Introduction to assembly language

The following is an introduction to the iAPx86 family of microprocessors that form the basis of IBM PC computers, starting with the 8088 and 8086 processors, continuing with the 80286, 80386, 80486, Pentium, and so on. The 8086 processor is, in fact, the basis of what is known briefly as the x86 microprocessor family. This is why this architecture (8086) will be referred to hereafter.

Basic architectural elements of the microprocessor

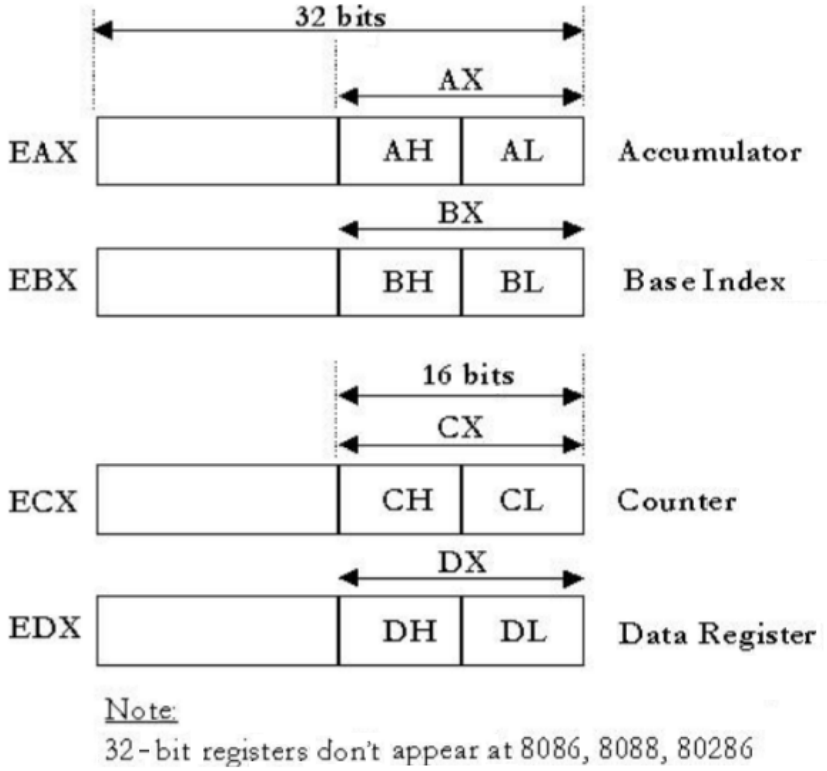


Figure 1. General purpose registers - accumulator, base index, counter and data

Microprocessor registers

The microprocessor registers are special memory locations located directly on the chip, making them the fastest type of memory. Another special thing about registers is that each of them has a specific purpose, providing some special, unique functionality. There are four main register categories: general purpose registers, *flags* register, segment registers and instruction pointer register.

General-purpose registers

General-purpose registers (see Figure 1 and Figure 2) are involved in the operation of most instructions, as source or destination operands for computations, data copies, pointers to memory locations or counting functions. Each of the 8 general purpose registers AX, BX, CX, DX, SP, BP, DI, SI are 16-bit registers for the 8086 microprocessor, and since the 80386 processor they have become 32-bit registers, called EAX, EBX, ECX, EDX, ESP, EBP, EDI, ESI respectively. Furthermore, the least significant 8 bits of the AX, BX, CX, DX registers form respectively the AL, BL, CL, DL registers (L stands for *Low*) and the most significant 8 bits of the same registers form the AH, BH, CH, DH registers (H stands for *High*) (Figure 1).

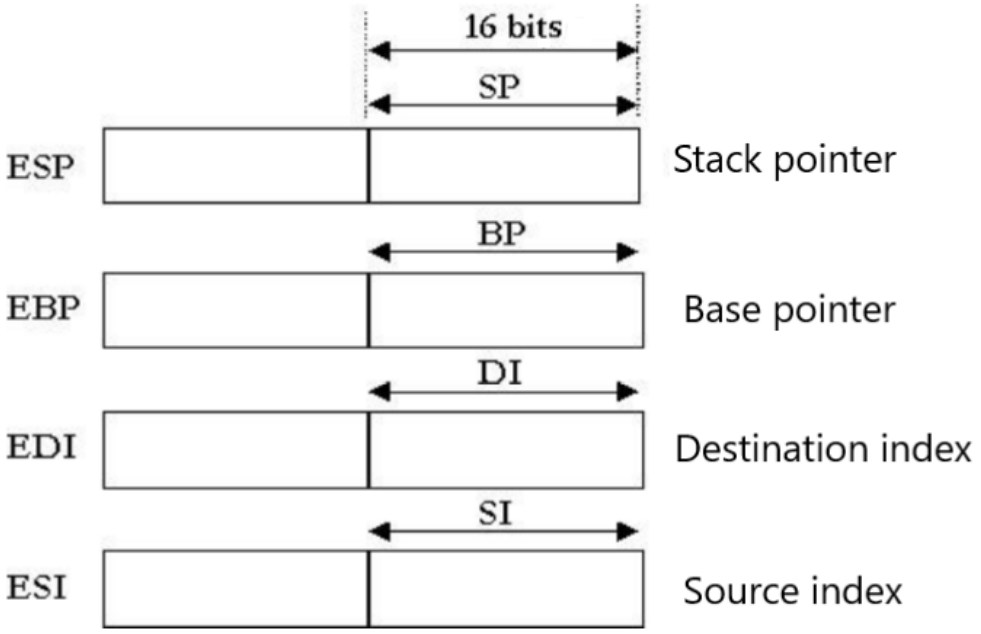


Figure 2. General purpose index and pointer rulers

We will focus next on general 16-bit registers; each of these can store a 16-bit value, can be used to store a value in memory, or can be used for arithmetic and logic operations. For example, the following instructions:

```
...
MOV BX, 2
MOV DX, 3
ADD BX, DX
...
```

loads value 2 into the BX register, value 3 into the DX register, adds the two values and the result (5) is stored in the BX register. In the above example we

can use any of the general purpose registers instead of the BX and DX registers. Apart from the property of storing values and using them as source or destination operands for data manipulation instructions, each of the 8 general purpose registers has its own "personality". We will see below what the specific characteristics of each of the general purpose registers are.

AX Registry (EAX for 32-bit)

The AX register (EAX) is also referred to as the *accumulator* register and is the main general purpose register used for arithmetic, logic and data movement operations. Multiplication and division operations always involve the AX register. Some of the instructions are optimized to execute faster when AX is used. In addition, the AX register is also used for all data transfers to/from the I/O ports. It can be accessed on 8, 16 or 32 bit portions and is referred to as AL (least significant 8 bits of AX), AH (most significant 8 bits of AX), AX (16 bits) or EAX (32 bits). We present below some other examples of instructions using the AX register. Note that data transfers are done for Intel instructions (also called *mnemonics*) from right to left, just the opposite of Motorola (we will see another similar example when writing data to memory in a different format in Motorola than in Intel), where the transfer is done from left to right.

Instruction: **MOV AX, 1234H** loads the value 1234H (4660 in decimal) into the AX accumulator register. As I said, the least significant 8 bits of the AX register are identified as AL (A-Low) and the most significant 8 bits of the same register are identified as AH (A-High). This is used to work with single-byte data, allowing the AX register to be used in place of two separate registers (AH and AL). The same rule applies to the general purpose registers BX, CX, DX. The following three instructions set the AH register to 1, increment this value by 1 and then copy it to the AL register:

```
MOV AH, 1
INC AH
MOV AL,AH
```

The final value of register AX will be 22 (AH = AL = 2).

BX register (EBX for 32-bit)

The BX (Base) register, or base register, can store addresses to reference various data structures, such as vectors stored in memory. A 16-bit represented value stored in the BX register can be used as a portion of the address of a memory location to be accessed. For example, the following instruction loads the AH register with the value in memory at address 21.

```
MOV AX, 0
MOV DS, AX
MOV BX, 21
MOV AH, [ BX ]
```

Note that we loaded the value 0 into the DS register before accessing the memory location referenced by the BX register. This is due to memory segmentation (segmentation discussed in more detail in the section on segment registers); by default, when used as a memory pointer, BX references the DS segment register relatively (the address it references is an address relative to the segment address contained in the DS register).

CX Registry (ECX for 32-bit)

The specialization of the CX (Counter) register is counting; therefore, it is also called the counter register. The CX register also plays a special role when the LOOP instruction is used. The counter role of the CX register is immediately apparent from the following example:

```
MOV CX, 5
start:
...
<instructions to be executed 5 times>
...
SUB CX, 1
JNZ start
```

Since the initial value of CX is 5, the instructions between the start label and the JNZ instruction will execute 5 times (until the CX register becomes 0). The SUB CX, 1 instruction decrements the CX register by 1 and the JNZ start instruction causes the jump back to the *start* label if CX is not 0. In the microprocessor language there is also a special instruction related to cycling. This is the LOOP instruction, which is used in combination with the CX register. The following lines of code are equivalent to the previous ones, but here the LOOP instruction is used:

```
MOV CX, 5
start:
...
<instructions to be executed 5 times>
...
LOOP start
```

Note that the LOOP instruction is used in place of the two previous SUB and JNZ instructions; LOOP automatically decrements the CX register by 1 and executes the jump to the specified label (*start*) if CX is non-zero, all in one instruction.

DX register (EDX)

The general-purpose DX register (Data register) can be used for Input/Output data transfers or when a multiplication or division operation takes place. The **IN AL, DX** instruction copies a Byte value from an input port whose address is in the DX register. The following instruction writes the value 101 to I/O port 1002:

```
...
MOV AL, 101
MOV DX, 1002
OUT DX, AL
...
```

Regarding multiplication and division operations, when dividing a 32-bit number by a 16-bit number, the most significant 16 bits of the dividend must be in DX. After division, the remainder of the division will be in the DX. The least significant 16 bits of the dividend must be in AX and the quotinent will be in AX. On multiplication, when multiplying two 16-bit numbers, the most significant 16 bits of the product will be stored in DX and the least significant 16 bits in the AX register.

SI Register

The SI (Source Index) register can be used, like BX, to reference memory addresses. For example, the following sequence of instructions:

```
MOV AX, 0
MOV DS, AX
MOV SI, 33
MOV AL, [ SI ]
```

Load the value (8 bits) from memory at address 33 into the AL register. The SI register is also very useful when used in conjunction with string instructions. The following sequence :

```
CLD
MOV AX, 0
MOV DS, AX
MOV SI, 33
LODSB
```

not only loads the AX register with the value from the memory address referenced by the SI register, but also adds the value 1 to the SI. This is particularly effective when sequentially accessing a number of memory locations, such as strings. String instructions can be repeated many times, so a single instruction can result in hundreds or thousands of operations.

DI Register

The DI (Destination Index) register is used in a similar way to the SI register. In the following sequence of instructions:

```
MOV AX, 0
MOV DS, AX
MOV     DI,     1000
ADD BL, [ DI ]
```

the 8-bit value stored at address 1000 is added to the BL register. The DI register is slightly different from the SI register for string instructions; while the SI is always the memory source pointer, the DI register serves as the memory destination pointer. Furthermore, in the case of string instructions, the SI register addresses memory relative to the DS segment register, while DI contains memory references relative to the ES segment register. If SI and DI are used with other instructions, they reference the DS segment register.

BP Register

In order to better understand the role of the BP and SP registers, it is time to say a few things about the portion of memory called the *stack*. The stack (see Figure 3) is a special portion of adjacent locations in memory. It is contained within a memory segment and identified by a segment selector stored in the SS register (except when using the non-segmented memory model where the stack can be located anywhere in the program's linear address space). The stack is a portion of memory where values can be stored and accessed on the LIFO (Last In - First Out) principle, therefore the last value stored in the stack is the first to be read from the stack. Usually the stack is used when calling a procedure or returning from a procedure call (the main instructions used are CALL and RET).

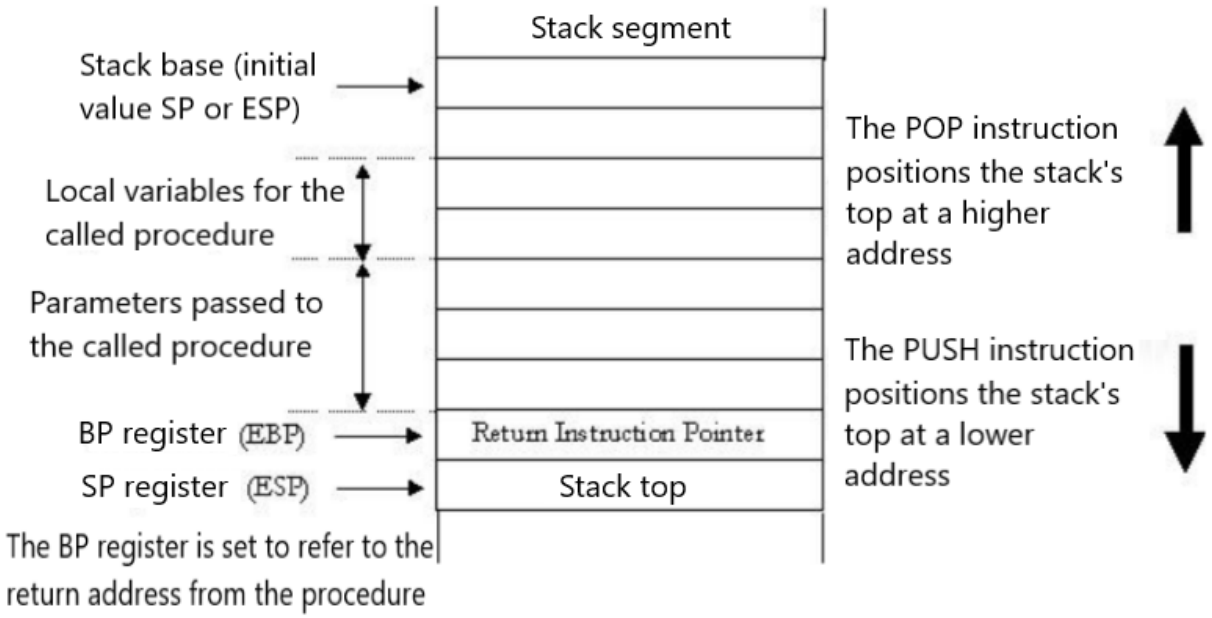


Figure 3. Stack structure

The base pointer register, BP (Base Pointer) can be used as a memory pointer like the BX, SI and DI registers. The difference is that while BX, SI and DI are normally used as memory pointers relative to the DS segment, the BP register refers relative to the SS stack segment. The principle is as follows: one way to pass parameters to a subroutine is to use the stack (this is commonly done in high-level languages - C, for example). If the stack is in the portion of memory referenced by the SS segment register (Stack Segment), the data is normally in the memory segment referenced by the DS, the data segment register. Since BX, SI and DI refer to the data segment, there is no efficient way to use the BX, SI, DI registers to reference parameters saved in the stack because the stack is located in a different memory segment. The BP register provides the solution to this problem by providing addressing in the stack segment. For example, the instructions:

```
PUSH BP
MOV BP, SP
MOV AX, [ BP+4 ]
```

cause the stack segment to be accessed in order to load the AX register with the first parameter sent by a C call to a routine written in assembly language. In summary, the BP register is designed to provide support for parameter access, local variables, and other needs related to accessing the stack portion of memory.

SP Register

The SP (Stack Pointer) register, or stack pointer, usually holds the travel address of the next available item within the stack segment. This register is probably the least "general" of the general purpose registers, as it is dedicated most of the time to stack management.

The BP register refers to the top of the stack at all times - this top of the stack is the address of the memory location where the next item in the stack will be inserted. The action of inserting a new item into the stack is called a *push*; hence the instruction is called PUSH. Similarly, the operation of removing an item from the stack top is called a *pop*, and the instruction equivalent to the operation is called POP. Figures 3 and 4 show the changes in the contents of the stack and the SP, BX and CX registers as a result of the execution of the following instructions (the SP register is assumed to have the initial value of 1000):

```
MOV BX, 9
PUSH BX
MOV CX, 10
PUSH CX
POP BX
POP CX
```

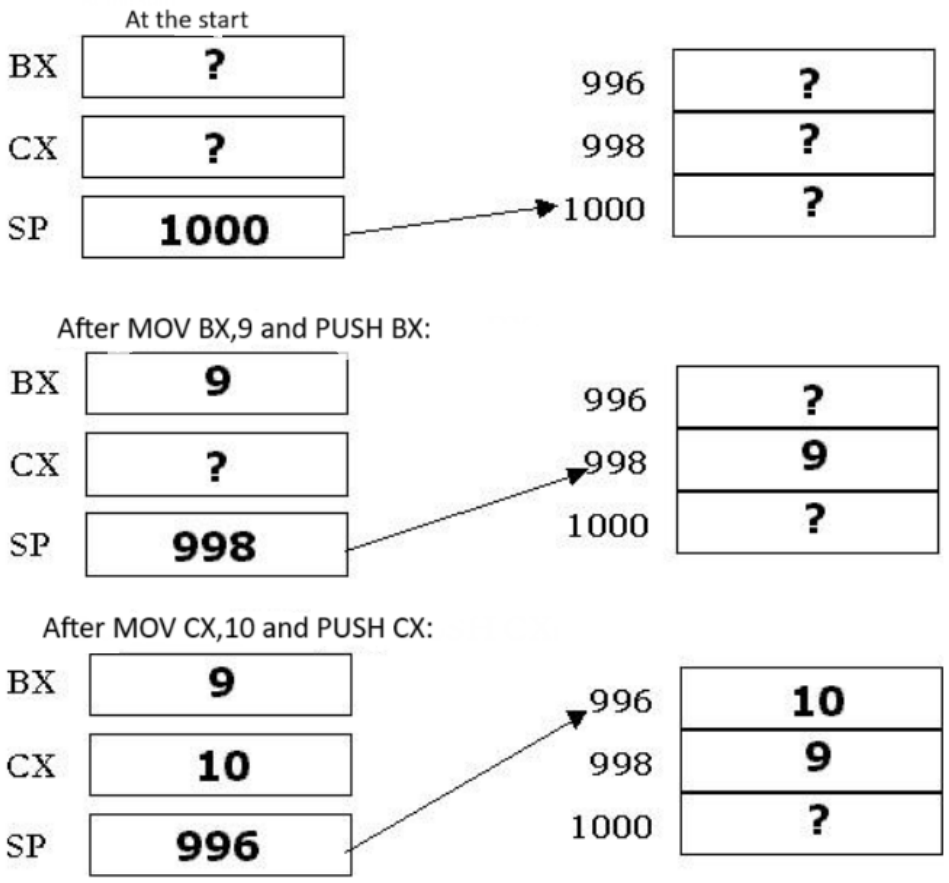


Figure 3. How the stack works after executing the first 4 instructions

It is allowed to store values in the SP register as well as to change its value by addition or subtraction just like the other general purpose registers; however, this is not recommended if we are not very sure what we are doing. By modifying the SP register, we will change the memory address of the stack vertex, which can have unintended effects, this is because the PUSH and POP instructions are not the only ways to use the stack. Whether we are calling a subroutine or returning from such a subroutine call, either procedure or function, in this case the stack is used. Some system resources, such as the keyboard or system clock, may use the stack when sending an interrupt to the microprocessor. This assumes that the stack is used continuously, so if the SP register (i.e. the stack address) is changed, the data in the new memory locations will no longer be correct. In conclusion, the SP register does not need to be changed directly; it is changed automatically by POP, PUSH, CALL, RET instructions. Any of the other general purpose registers can be changed directly at any time.

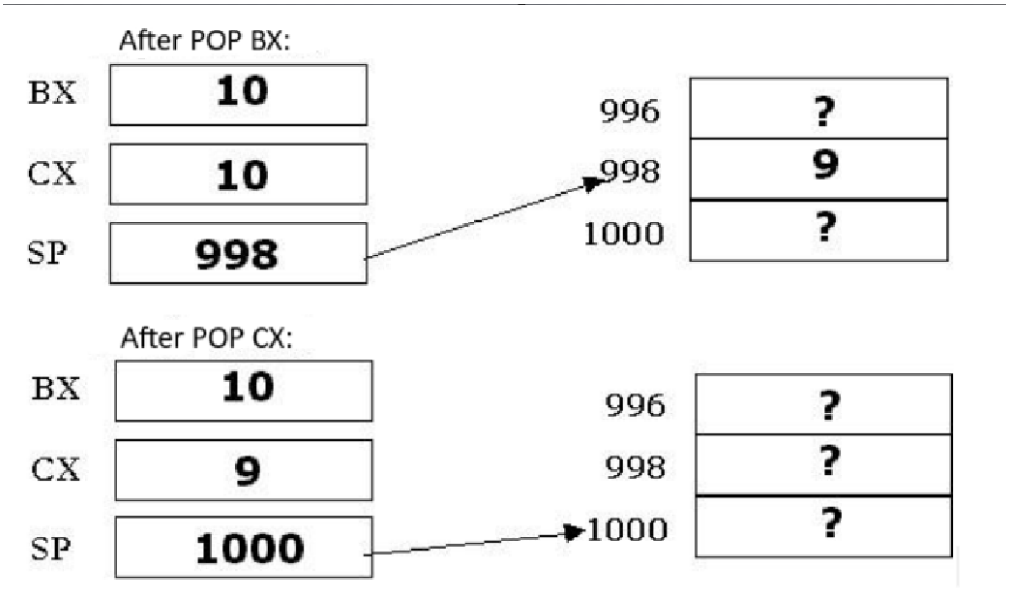


Figure 4. Stack operation after the last two POP instructions

Instruction pointer register (IP)

The instruction pointer register (IP - Instruction Pointer, see figure 5) is always used to store the address of the next instruction to be executed by the microprocessor.As an instruction is executed, the instruction pointer is incremented and will refer to the next memory address (where the next instruction to be executed is stored). Usually, the instruction that has to be executed is at the address immediately following the instruction that just have been executed, but there are special cases (resulting either from calling a subroutine via the CALL instruction or returning from a subroutine via the RET instruction). The instruction pointer cannot be modified or read directly; only special instructions can load this register with a new value. The instruction pointer register does not specify the full memory address of the next instruction to be executed, for the same reason of memory segmentation. To fetch an instruction from memory, the CS register provides a base address and the instruction pointer register indicates the address of the move from this base address.

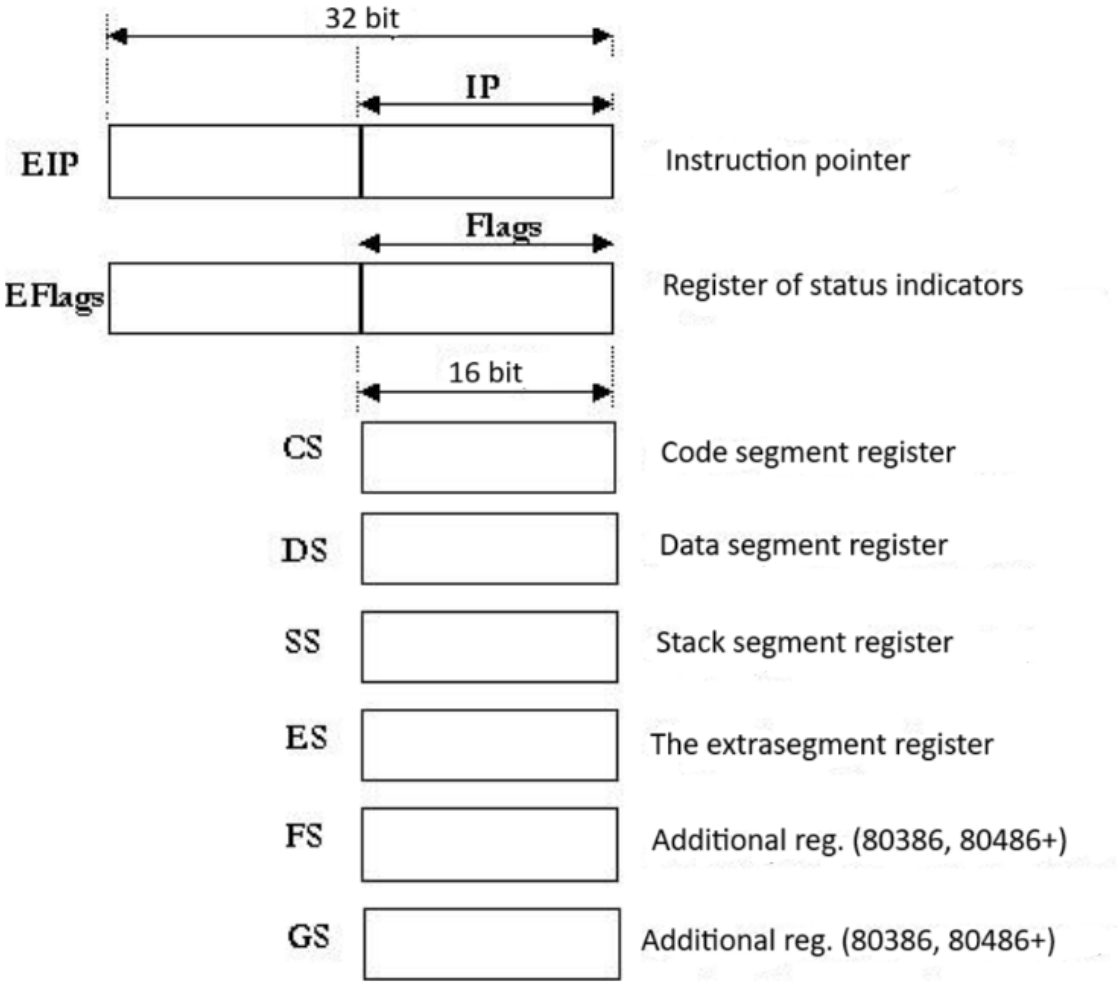
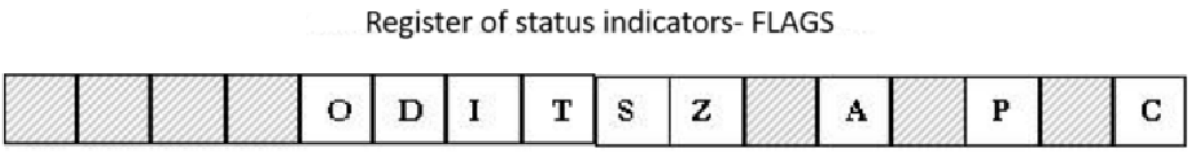


Figure 5. Segment rulers, instruction pointer and status indicator register Register of Status Indicators (FLAGS)



O - Overflow Flag

D - Direction Flag

I - Interrupt Flag

T - Trap Flag

S - Sign Flag

Z - Zero Flag

A - Auxiliary Carry Flag

P - Parity Flag

C - Carry Flag

Figure 6. Status indicators register - detail

The 16-bits Status Indicator Register (FLAGS) contains information about the status of the microprocessor as well as the results of the last executed instructions. A *flag* is itself a 1-bit memory location indicating the current state of the microprocessor and its method of operation. A flag is said to be 'set' if it has a value of 1 and 'not set' otherwise. Status indicators change after the execution of arithmetic or logic instructions. Examples of status indicators (see Figure 6):

- C (Carry) indicates the occurrence of a binary carry figure, in the case of an addition, or borrow, in the case of a subtraction;
- O (Overflow) occurs after an arithmetic operation. If it is set, it means that the result does not fit in the destination operand;
- Z (Zero) indicates that the result of an arithmetic or logical operation is zero;
- S (Sign) indicates the sign of the result of an arithmetic operation;
- D (Direction) - when zero, the processing of the string elements is from the lowest to the highest address, otherwise it is the other way around;
- I (Interrupt) controls the ability of the microprocessor to respond to external events (interrupt calls);
- T (Trap) is used by debugger programs, enabling or disabling the possibility of step-by-step program execution. If set, UCP interrupts each instruction, leaving the debugger program to execute that program step by step;
- A (Auxiliary carry) supports operations in BCD code. Most programs do not support representing numbers in this format, so it is rarely used;
- P (Parity) is set according to the parity of the least significant bits of a data operation. Thus, if the result of an operation contains an even number of bits 1, this flag is set. If the number of bits 1 in the result is odd, then the PF flag is zero. It is commonly used by communications software, but Intel introduced this flag not to perform a specific functionality, but to ensure compatibility with older x86 family microprocessors.

Segment managers

The properties of segment registers (see Figure 5) are closely related to the notion of memory segmentation. The assumption is as follows: the 8086 is capable of addressing 1MB of memory, so 20-bits addresses are required to encompass all locations in the 1MB of memory space. However, the registers used are 16-bits registers, so a solution to this problem had to be found. The solution found is called *memory segmentation*; in this case the 1MB memory is divided into 16 segments of 64 KB (16*64 KB = 1024 KB = 1 MB).

The notion of memory segmentation involves the use of memory addresses consisting of two parts. The first part represents the segment address and the second part represents the offset address (Figure 7).

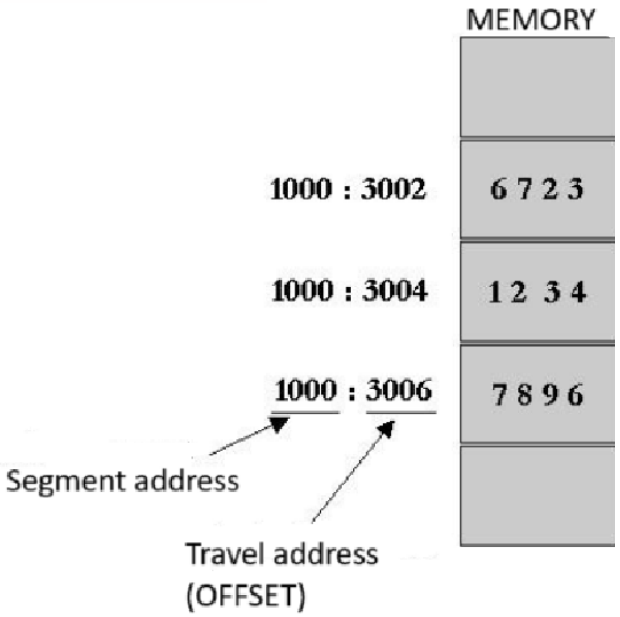


Figure 7. The two portions of a segmented address

Each 16-bits memory pointer is combined with the contents of a 16-bits segment register to form a complete 20-bit address. The segment address together with the offset address are combined in the following way: the segment value is shifted to the left by 4 bits (multiplied by 16 = 2^4) and then added with offset address value. The address thus constructed is called the effective address; being a 20-bits address it can access 2^{20} bytes of memory, i.e. 1 MB of memory. The construction of an effective address is shown in Figure 8.

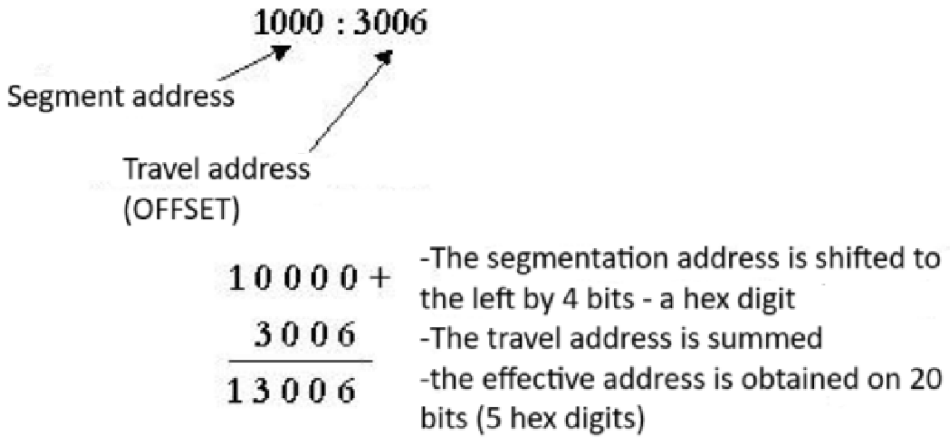


Figure 8. Example of effective address calculation

CS register - this register refers to the beginning of the 64 KB block of memory in which the program code (code segment) resides. The 8086

microprocessor cannot fetch any instruction for execution other than that defined by the CS. The CS register can be modified by a number of instructions, such as jump, call or return instructions. It cannot be loaded directly with a value, but only via another general register.

DS register - refers to the beginning of the data segment, where the data crowd with which the running program works is located.

ES register - refers to the beginning of the 64KB block known as the extra-segment. It is not dedicated to any particular purpose, but is available for various actions. Sometimes it can be used to create an additional 64KB memory block for data. This extra-segment works very well for STRING instructions. All STRING instructions that write to memory use the ES address: DI as the memory address.

SS register - refers to the beginning of the stack segment, which is the 64 KB block where the stack is located. All instructions that use the SP register by default (POP, PUSH, CALL, RET instructions) work in the stack segment because the SP register is only able to address memory in the stack segment.

General format of an assembly language instruction

A line of code written in assembly language has the following general format:

<name> <instructive/directive> <operators> <;comment>

where:

- **<name>** - represents an optional symbolic name;
- **<instruction/directive>** - represents the mnemonic (name) of an instruction or directive;
- **<operands>** - represents a combination of one, two or more operands (or even none), which may be constant, memory references, register references, strings, depending on the particular structure of the instruction;
- **<;comment>** - represents an optional comment that can be placed after the ";" character until the end of the respective line of code.

Variable names and labels

Names used in a program written in assembly language can identify numeric variables, string variables, memory locations or labels. For example, the following code sequence, which calculates the value of three factorial (3!=1x2x3=6), contains several variable names and labels:

```
.MODEL small
.STACK 200h
.DATA
Factorial_Value DW ?
Factorial DW?
.CODE

Three_Factorial PROC
MOV ax, @data
MOV ds, ax
MOV [Factorial_Value], 1
MOV [Factorial], 2
MOV      cx,      2
Cycle:
MOV ax, [Factorial_Value]
MUL [Factorial]
MOV [Factorial_Value], ax
INC [Factorial]
LOOP Cycling
RET
Three_Factorial ENDP
END
```

The names *Factorial_Value* and *Factorial* are used to define two word variables (16-bits), *Three_Factorial* identifies the name of the procedure (subroutine) containing the code for the factor calculation, allowing it to be called from elsewhere in the program. *Cycling* represents a tag name, identifying the address of the instruction MOV ax, [Factor_Value], so that the LOOP instruction used below can jump back to this instruction. Variable names may contain the following characters: the letters a-z and A-Z, the digits 0-9 as well as the special characters _ (*underscore*), @ (*"at"* - also read "*a round"* or *"monkey's tail"*), $ and ?. The period character (".") can also be used as the first character of a label name. The digits 0-9 may not be used in the first position of the name; nor may names containing only a single $ or ? character. Each name can be defined *only once* (names are *unique*) and can be used as operands as many times as desired in a program. A name may appear in a program on a line by itself (that line contains no other instruction or directive), in which case the value of the name is given by the address of the instruction or directive on the next line of the program. For example, in the following sequence:

```
...
```

```
JMP decrease
... decrease:
SUB AX, CX
…
```

the next instruction to be executed after the **JMP decrease** instruction will be the **SUB AX, CX** instruction. The above example is equivalent to the sequence:

```
...
JMP decrease
...
decrease: SUB AX, CX
...
```

There are some advantages to writing instructions on separate lines. First, when we write a tag name on a single line, it is easier to use long tag names without spoiling the "shape" of the program written in assembly language. Second, it is easier to add a new instruction to the label later if it is not written on the same line as the instruction.

Variable or label names used in a program should not be confused with *reserved* assembly names, such as directive and instruction names, registry names, etc. For example, a statement like:

```
...
ax DW 0
BYTE:
…
```

cannot be accepted because AX is the name of the accumulator register, AX, and BYTE is a reserved keyword.

Any label name appearing on a line without instructions or appearing on a line with instructions must have a ":" after its name. At the same time, an attempt is made to give a suggestive name to the labels in the program. Take the following example:

```
...
CMP AL, 'a'
JB Is_Not_lowercase
CMP AL, 'z'
JA Is_Not_lowercase
SUB AL, 20H ; turns into capital letter
Is_Not_lowercase:
...
```

compared to:

```
...
CMP AL, 'a'
JB x5
CMP AL, 'z'
JA x5
SUB AL, 20H          ; turns into capital letter
x5:
...
```

If in the first case we used a suggestive tag name (Is_Not_lowercase), in the second case, identical in functionality to the first, the tag was named x5, absolutely unsuggestive!

Remark:

The assembly language is not *case sensitive*. This means that, in a program written in assembly language, variable names, labels, instructions, directives, mnemonics, etc., can be written in either upper or lower case, and there is no difference between them (*is_not_lowercase* is the same as *is_not_lowercase* or *Is_Not_Lowercase*, etc.).

Simplified segment directives

Due to the fact that the 8086 microprocessor registers are 16-bit registers, it was necessary to use 64KB memory segments (the maximum that can be addressed with 16 bits - 64KB=2^16=65536). In a program written in assembly language (we will use the abbreviation ASM) there are three segments: the code segment, the data segment and the stack segment.

Segment directives (either in standard or simplified form) are required in any program written in assembly language to define and control the use of segments and the END directive is always used to terminate program code.

Examples of simplified segment directives are:

```
.STACK
.CODE
.DATA
.MODEL
DOSSEG
END
```

.STACK,.CODE,.DATA define the stack, code and date segments..

For example, **.STACK 200H** defines a 512-byte stack (in ASM the values ending with the letter H means hexadecimal). Such a stack value is normally sufficient; however, some programs (particularly recursive ones) may require larger stack sizes.

The **.CODE** directive marks the beginning of the code segment.

The **.DATA** directive marks the beginning of the data segment, i.e. where we will place the memory variables. Representative here is the fact that the DS segment register must be explicitly loaded with the value "@data" before accessing the memory locations in the segment defined by .DATA. Since a segment register can be loaded either from a general register or from a memory location but cannot be loaded directly with a constant, the DS segment register is generally loaded in a sequence of 2 instructions:

```
...
mov ax, @data
mov ds, ax
...
```

(another general register can be used instead of AX).

The preceding sequence means that DS will refer to the data segment starting with the .DATA directive.

We consider below an example of a program that displays the text stored in the DataString on the screen:

```
;Program p01.asm
.MODEL small          ;specify SMALL memory model
.STACK 200H           ;a 512-byte stack is defined
.DATA                 ;specify the beginning of the segment of
                      ;date
 DataString DB 'Hello!$' ;the variable DataString is defined
                      ;initialised with the value ; "Hello!"
.CODE                 ;the start of the code segment
ProgramStart:         ;any program has a start label
mov bx,@data          ;sequence setting the DS register to
                      ;refer to the data segment which
                      ;starts with .DATA
mov ds,bx
mov dx, OFFSET DataString ;load in DX the address
                      ;of the DataString variable
mov ah,09             ;DOS function code to display a string

int 21H               ;DOS call to display string
mov ah, 4cH           ;DOS function code to terminate the
                      ;program
int 21H               ;DOS program termination call
```

```
END ProgramStart    ;the code termination directive of the
                    ;program
```

Explanation:

1.Comments can be inserted into an ASM program by using ";". Everything after ";" and up to the end of the line is considered a comment.

2.It does not matter whether the program is written using upper or lower case (it is not "*case sensitive*").

3.Without the two instructions that set the DS register to the segment defined by .DATA, the string display function would not work properly. The DataString variable is in the .DATA segment and cannot be accessed unless the DS is set to that segment. This is explained in the following way: when we make the DOS call to display a string, we must traverse the entire segment:offset address of the string in DS:DX. Therefore, only after loading DS with the .DATA segment and DX with the address (offset) of the DataString do we have a complete segment:offset reference to the DataString.

Remarks:

We don't have to explicitly load the CS segment register because DOS does this automatically when we run a program. Thus, if CS were not already set when the first instruction in the program was executed, the processor would not know where to find the instruction and the program would never run. Similarly, the SS segment register is set by DOS prior to program execution and usually remains unchanged during program execution.

With the DS segment register things are different. While the CS register refers to instructions (code), SS refers ("points") to the stack, DS "points" to data. Programs do not directly manipulate instructions or stacks but deal directly with data. Also, programs will access data located in different segments at any time. One may wish to load a segment into DS, access the data in that segment and then load DS with another segment to access a different block of data. In small or medium programs we will not need more than one data segment but more complex programs often use multiple data segments.

The next program will display a character on the screen, using the ES register load instead of DS.

```
;Program p02.asm
.MODEL small
.STACK 200H
.DATA
OutputChar DB 'B'          ;OutputChar variable definition
                           ;initialised with the value "B"
```

```
.CODE
ProgramStart:
mov dx, @data
mov es, dx          ;unlike the previous program, use
                    ES to specify the data segment
mov bx, offset OutputChar     ;load BX with the address of
                              ;the OutputChar variable
mov dl, es:[bx]          ;load AL with the value from
                         ;the explicit address es:[bx]
                         ;(indexed addressing)
mov ah,02           ;DOS function code for displaying a
                    ;character

int 21H             ;display execution DOS call
mov ah, 4cH         ;DOS termination function code of the
                    ;program
int 21H             ;DOS program termination call
END ProgramStart    ;termination directive of the
                    ;code of the program
```

DOSSEG is the directive that causes segments in a program to be grouped according to Microsoft segment addressing conventions.

.MODEL Directive

This is the directive that specifies the memory model for an ASM program using simplified segment directives.

Definitions: 'near' means the 16-bit address (offset) within the same segment, while 'far' means a full segment:offset address within a segment other than the current one.

Memory models that can be specified via the .MODEL directive are:

- ***tiny*** - both code and program data fit in the same 64KB segment. Both the code and the data are of type near.

- ***small*** - program code must be in a single 64KB segment and data in a separate 64KB block; code and data are not

- ***medium*** - the program code can be larger than 64KB but the data must be in a single 64KB segment. The code is far, the data is near.

- ***compact*** - program code can be in one segment, data can be larger than 64 KB. The code is non-zero, the data is far.

- ***large*** - both code and data can exceed 64KB, but no bulk data can exceed 64KB. Both code and data are far.

- ***huge*** - both code and data can exceed 64KB and bulk data can exceed 64KB. Both code and data are far. Pointers to elements in a bulk are far.

The following are some examples of how to declare variables and address memory.

```
 var1 DW 01234h      ;define a word variable with
                     ;the value 1234h
 var2 DW 01234       ;define a word variable with
                     ;decimal value 1234 (4D2 in hex)
 var3 RESW 1         ;space is reserved for a variable
                     ;word (of value 0)
 var4 DW ABCDh       ;illegal assignment

  messagesco2 DB 'SCO 2 is the preferred course!'

...start:

   mov ax,cs         ;segment setting data
   mov ds,ax         ;DS=CS

; any memory reference is assumed to be relative to the DS
segment

   mov ax,[var2]           ; AX <- var2
                           ; == mov ax,[2]
   mov si,var2
                     ;use SI as pointer to var2
                     ;(C code equivalent SI=&var2)
   mov ax,[si]       ;read from memory the value of
                     ;var2 (*(&myvar2))
                     ;(indirect reference)
   mov bx,messagesco2 ; BX is a pointer to a string
                       ;(C equivalent: BX=&messagesco2)

   dec BYTE [bx+1]  ; transform 'C' to 'B' !

   mov si, 1         ;Use SI as index

inc byte [messageco2+SI]; == inc byte[SI + 8]
                         ; == inc byte [9]

; Memory can be addressed using 4 registers:
; SI -> Implies DS
; DI -> Implies DS
; BX -> Implies DS
; BP -> Implies SS ! (not very often used)
;
;Example:
```

```
mov ax,[bx] ; ax <- word in memory referenced by BX
mov al,[bx] ; al <- byte in memory referenced by BX
mov ax,[si] ; ax <- word referenced by SI
mov ah,[si] ; ah <- byte referenced from SI
mov cx,[di] ; di <- word referenced from DI
mov ax,[bp] ; AX <- [SS:BP] Stack operation!

; In addition, BX+SI and BX+DI are allowed:

mov ax,[bx+si]

mov ch,[bx+di]

; 8 or 16 bit displacements:

mov ax,[23h]        ;ax <- word in memory DS:0023
mov ah,[bx+5]       ;ah <- byte in memory [DS:BX+5]
mov ax,[bx+si+107]  ;ax <- word at[DS:BX+SI+107]
mov ax,[bx+di+47]   ;ax <- word at [DS:BX+DI+47]
;WARNING: copying from memory to memory is illegal!
;Always pass the copied value through a register

mov [bx], [si]      ;Illegal

mov [di], [si]      ;Illegal

;Special case: stack operations!

pop word [var]         ; var <- [SS:SP]
```

Memory addresses and values

A program written in assembly language can refer either to a memory address (OFFSET) or to a variable value stored in memory. Unfortunately, assembly language is neither strict nor intuitive about the ways in which these two types of reference are made, and as a result, references to OFFSET or value are often confused. Figure 9 illustrates the concepts of offset and value stored in memory.

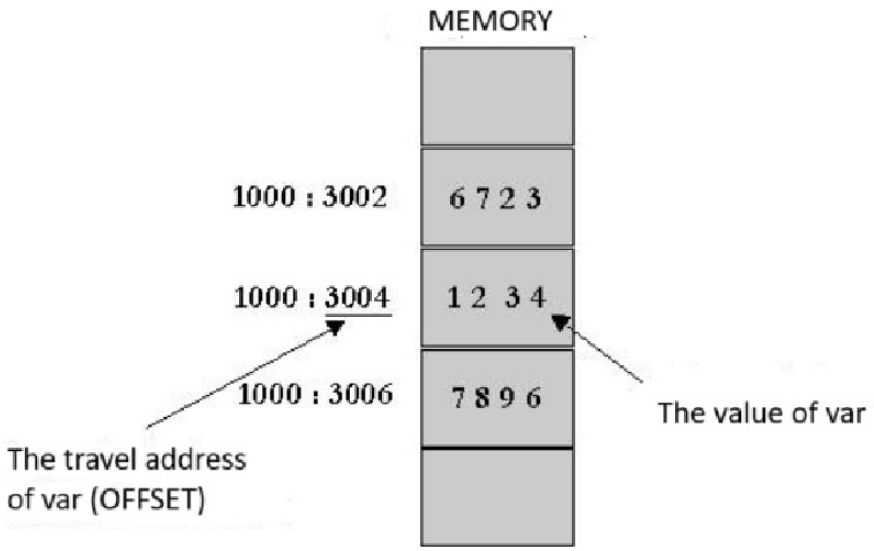


Figure 9. Illustration of the notions of displacement address and value stored in memory

The offset of a word-sized *var* memory variable is the constant value 5004H, obtained with the OFFSET operator. For example, the instruction:

MOV BX, OFFSET var

Load the value 5004H into the BX register. The value 5004H does not change; it is constructed within the instruction. The value of *var* is 1234H, read from memory at the address given by offset 5004H in the data segment. One way to read this value is to load the BX, SI, DI or BP registers with the offset of *var* and then use that register to address the memory. Instructions:

MOV BX, OFFSET var
MOV AX, [BX]

Has the effect of loading the value of *var* (1234H) into the AX register.
You can also load the value of *var* directly into AX using:

MOV AX, var
Sau
MOV AX, [var]
While the displacement value remains constant, the value 1234H is not permanently associated with *var*. For example, the instructions:

MOV [var], 5555H
MOV AX, [var]
Has the effect of loading the value 5555H into the AX register.

In other words, while the offset of *var* is a constant value describing a fixed address in a data segment, the value of *var* is a modifiable number stored at that (memory) address. Instructions:

MOV[var], 1

ADD [var], 2

Changes the value of var to 3, but the instruction:

ADD OFFSET var, 2 is equivalent to **ADD 5002H, 2**, which is nonsense because it is impossible to sum one constant with another.

A problem that can often arise during programming is that of omitting OFFSET; for example, if we write **MOV SI, var** when we actually want to load the displacement of *var* into SI. No error will be reported in this case, since *var* is a word variable. However, at program execution time, the SI register will be loaded with the value of *var* (1234H) instead of OFFSET, which can lead to unpredictable results. In this case, references to address constants should be preceded by OFFSET and references to values in memory should be enclosed in square brackets ("[" and "]"), thus eliminating ambiguity.

Intel microprocessor instructions

Intel x86 microprocessors have an impressive instruction set, as do all processors in the CISC (Complex Instruction Set Computer) class. Instructions can be divided into: logic, arithmetic, transfer and control instructions. We present below some examples of each class of instructions.

Logical instructions

Logic statements implement basic logic functions, on a per byte or per word basis. They act bit by bit, so they apply the respective logical function to all bits or bit pairs corresponding to the operands. The logic instructions are as follows:

- **NOT**: A =~A
- **AND**: A &= B
- **OR**: A |= B
- **XOR**: A ^= B
- **TEST**: A & B

As a rule, logical statements have an effect on status indicators, except for the NOT statement, which has no effect on any flag (status indicator). These effects are as follows:

- Delete the carry indicator (C)
- Clear the overflow indicator (O)
- Set zero flag (Z) if the result is zero, or clear it otherwise
- Copy the "higher" bit of the result to the sign pointer (S)
- Set the parity bit (P) according to the *parity of* the result

Instruction NOT

It is a single operand instruction (*unary* instruction) with general form:
NOT *destination*
Where *destination* is either a register or an 8-bit or 16-bit memory location. The instruction has the effect of inverting (negating) all bits of the operand, i.e. bringing it into reverse code form - complement to 1.

AND instruction

It is a two-operand instruction (*binary* instruction) with general form:
AND *destination, source*
Where *destination* is either a register or an 8-bit or 16-bit memory location, and *source* can be a register, memory location or an 8-bit or 16-bit constant. The statement has the operation: <destination> == <destination> AND <source>. The modified status flags are: SF, ZF, PF, CF, OF = 0, AF undefined.

TEST instruction (AND "non-destructive")

It is a two-operand instruction (*binary* instruction) with general form:
TEST *destination, source*
Where *destination* is either a register or an 8-bit or 16-bit memory location, and *source* can be a register, memory location or an 8-bit or 16-bit constant. The instruction has the same effect as the AND instruction, except that the destination operand is not changed, and the status pointers are changed in the same way as the AND instruction.

OR instruction

It is a two operand instruction with general form:
OR *destination, source*
Where *destination* is either a register or an 8-bit or 16-bit memory location, and *source* can be a register, memory location or an 8-bit or 16-bit constant. The statement has the effect: <destination> == <destination> OR <source>. The modified status flags are: SF, ZF, PF, CF, OF = 0, AF undefined.

XOR instruction (OR-Exclusive)

It is a two operand instruction with general form:

XOR *destination, source*

Where *destination* is either a register or an 8-bit or 16-bit memory location, and *source* can be a register, memory location or an 8-bit or 16-bit constant. The statement has the effect: <destination> == <destination> XOR <source>. The modified status flags are: SF, ZF, PF, CF, OF = 0, AF undefined. The XOR function, called OR-Exclusive (or *anti-coincidence*) has logical value 1 when its operands are different (one has value 0 and the other has value 1) and logical value 0 when both operands have the same value (either both have value 0 or both have value 1).

Remark:

Most of the time, AND and OR statements are used in place of "masking" of data; in this sense, a "*mask*" value is used to force certain bits to take the value zero or the value 1 within another value. O
Such a logical "mask" has an effect on some bits, while leaving others unchanged. Examples:

- **AND CL, 0Fh** - causes the 4 most significant bits to take the value 0, while the less significant bits are left unchanged;
  Thus, if the CL register has the initial value **1001 1101**, after executing the **AND CL** instruction**, 0Fh** will have the value **0000 1101**.

- **OR CL, 0Fh** instruction - causes the least significant 4 bits to take the value 1, while the more significant bits remain unchanged. If the CL register has the initial value **1001 1101**, after execution of the **CL OR** instruction**, 0Fh** will have the value **1001 1111**.

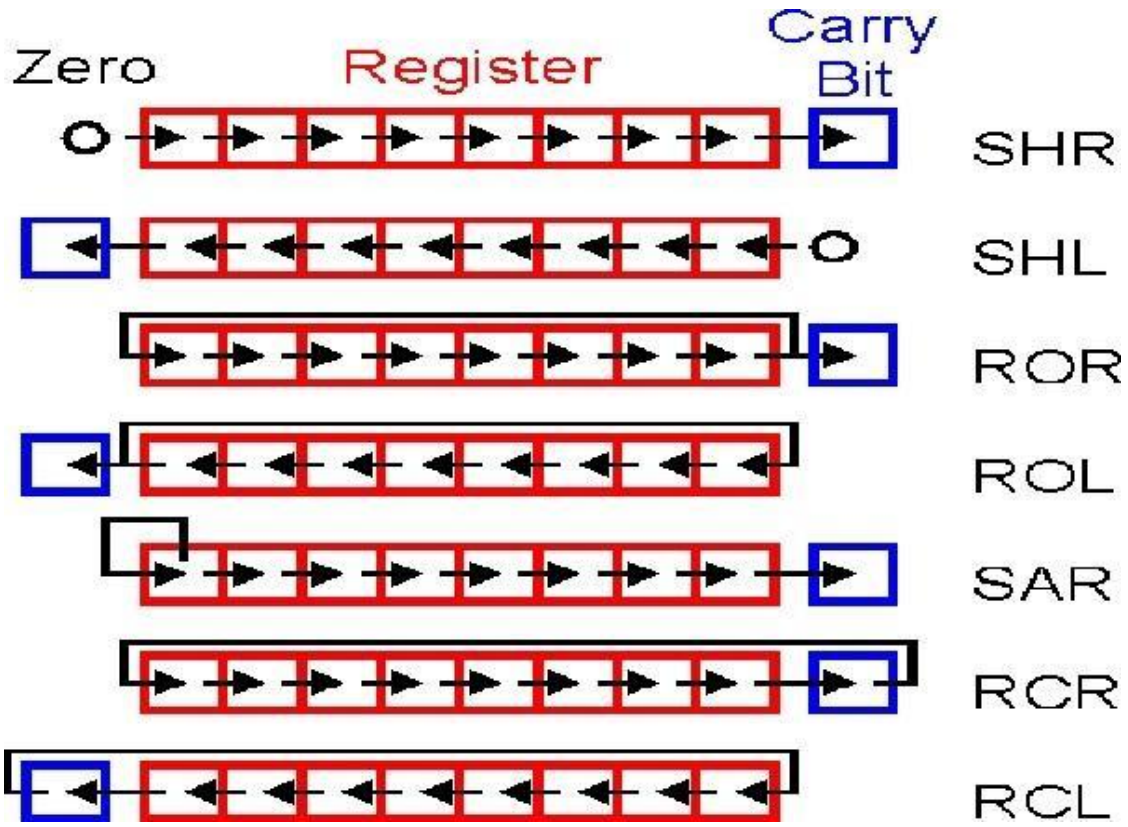


Figure 10. Movement and rotation instructions

Driving and turning instructions

This type of instruction (see Figure **10**) allows bit-level shift and rotation operations to be performed. They have two operands, the first operand being the one to which the bit shift operation is applied, and the second (the *counter* operand) signifying the number of bits by which the bit shift is performed. Operations can be performed from right to left or vice versa. Shift means translation of all bits in the operand to the left/right, with a fixed value filled in the remaining free position and the loss of the right/left bits. Rotation involves translating the bits in the operand to the left/right, with the bits that are lost on the opposite side being filled in on the right/left. The general syntax of move and rotate instructions is as follows:

INSTR<operand> , <contor>

Where **INSTR** represents the instruction name, **<operand>** represents an 8-bit or 16-bit register or memory location, and **<count>** signifies the number of bits by which the move is made, i.e. either a constant or the CL register (thus confirming its role as a counter).

Remark.

There are always two ways to travel:

- By using an effective counter - e.g. SHL AX, 1
- By using the CL register as a counter - for example: SHL AX, CL

SHL/SAL (Shift Left/Shift Arithmetic Left) instruction

This instruction translates the operand bits one position to the left whenever the numerator operand specifies. Positions left vacant by the left shift are padded with zeros to the least significant bit, while the most significant bit is shifted to the CF (Carry Flag) flag.

Represents a fast way to multiply by a power of 2 (depending on the number of bits for which the left shift is made).

Example:

1. Multiply AX by 10 (1010 in binary) (multiply by 2 and 8, then add the results)

```
shl     ax, 1          ; AX ori 2
mov   bx, ax           ; save 2*AX to BX
shlax , 2              ; 2*AX(original) * 4 = 8*AX(original)
add     ax, bx         2*AX + 8*AX = 10*AX
```

2. Multiply AX by 18 (10010 in binary) (multiply by 2 and 16, then add the results)

```
shl     ax, 1          ; AX ori 2
mov   bx, ax           ; save 2*AX
shlax , 3              ; 2*AX(original) times 8 =
16*AX(original) add       ax, bx2*AX + 16*AX = 18*AX
```

SHR (Shift Right) instruction

This instruction translates the bits of the operand one position to the right whenever the numerator operand specifies. The least significant bit moves to the CF (Carry Flag) indicator.

Represents a quick way of *unsigned* division to a power of 2 (if the move is made with *one position* to the right, the operation is equivalent to a division by 2,

if the move is made with two positions, the operation is equivalent to a division by 2^2 , etc.). The division operation is performed *unsigned*, completed with a leftmost bit 0 (the most significant bit).

SAR (Shift Arithmetic Right) instruction

This instruction translates the bits of the operand one position to the right whenever the numerator operand specifies. The most significant bit remains unchanged, while the least significant bit is copied to the CF (Carry Flag) flag.

Represents a fast way to *sign* division to a power of 2 (depending on the number of bits by which the right shift is made).

RCL (Rotate through Carry Left) instruction

This instruction causes the operand bit to rotate to the left via CF (Carry Flag). Thus, the most significant bit moves from the operand to the CF, then moves all bits in the operand one position to the left and the original CF moves to the least significant bit in the operand. **ROL (Rotate Left) instruction**

This instruction causes the operand bits to rotate to the left.
Thus, the most significant bit passes from the operand into the least significant bit.

Example:

After executing the instructions:
ROL AX, 6
AND AX, 1Fh
Bits 10-14 of AX move to bits 0-4.

RCR (Rotate through Carry Right) instruction

This instruction causes the operand bit to be rotated to the right via CF (Carry Flag). Thus, the bit in CF is written back to the most significant bit of the operand.

ROR (Rotate Right) instruction

This instruction causes the operand bits to rotate to the right.
The least significant bit is passed to the most significant bit.

Example:

```
MOV ax,3     ; Initial values               AX = 0000 0000 0000 0011
MOV bx,5     ;                              BX = 0000 0000 0000 0101
OR ax,9      ;      ax <- ax | 0000 1001    AX = 0000 0000 0000 1011
AND ax,10101010b ; ax <- ax & 1010 1010 AX = 0000 0000 0000 1010
XOR ax,0FFh         ; ax <- ax ^ 1111 1111 AX = 0000 0000 1111 0101
NEG ax              ; ax <- (-ax)           AX = 1111 1111 0000 1011
NOT ax              ; ax <- (~ax)           AX = 0000 0000 1111 0100
OR ax,1      ; ax <- ax | 0000 0001 AX = 0000 0000 1111 0101
SHL ax,1     ; logical left shift by 1 bit AX = 0000 0001 1110 1010
SHR ax,1     ; 1-bit right logic depl       AX = 0000 0000 1111 0101
ROR ax,1     ; left rotation (LSB=MSB)      AX = 1000 0000 0111 1010
ROL ax,1     ; right rotation (MSB=LSB)     AX = 0000 0000 1111 0101
MOV cl,3     ; we use CL for depl with 3-bit           CL = 0000 0011
SHR ax,cl    ; divide AX by 8               AX = 0000 0000 0001 1110
MOV cl,3     ; we use CL for depl for with 3-bit       CL = 0000 0011
SHL bx,cl    ; multiply BX by 8             BX = 0000 0000 0010 1000
```

Arithmetic instructions

ADD instruction (ADDition)

The ADD instruction has the general format:

ADD <destination> <source>

Where <*destination*> can be a general register or memory location, and <*source*> can be a general register, memory location or an immediate value. However, the two operands cannot be memory locations at the same time. The result of the operation is next: <destination> == <destination> + <source>. The status indicators changed by this operation are: AF, CF, PF, SF, ZF, OF. The operands can be 8-bit or 16bit and must be the same size. If there is an ambiguity in the way the operands are expressed (8 or 16 bits) the PTR operator shall be used.

Example:

ADDAX, BX gather between registers - AX ? AX + BX

```
ADDDL, 33h        actual gather - DL DL + 33h MOV
DI, NUMB ; address of NUMB
MOV AL, 0         delete amount ADD
AL, [DI]          add [NUMB]
ADDAL, [DI + 1] ; add [NUMB + 1]
ADD word ptr [DI], -2   ; destination in memory, immediate source
ADD byte for VAR, 5     ; forcing instruction on one byte, VAR being
                        ; declared DW
```

INC (Increment addition) instruction

The INC instruction has the general format:

INC <destination>

Where <*destination*> is an 8-bit or 16-bit register or operand in memory and the meaning of the operation is to increment the destination value by 1. All status flags are affected except CF (Carry Flag).

Example:

```
MOV DI, NUMB          ; address of
NUMBMOV AL, 0         delete amount
ADD AL, [DI]          ; add [NUMB]
INC   DI              DI = DI + 1
ADD AL, [DI]          ; add [NUMB + 1]
```

ADC (ADdition with Carry) instruction

The ADD instruction has the general format:

ADD <destination> <source>

Where <*destination*> can be a general register or memory location, and <*source*> can be a general register, memory location or an immediate value.

The instruction acts just like ADD, except that the CF bit is added to the result. It is usually used to add numbers larger than 16 bits (8086- 80286) or larger than 32 bits to 80386, 80486, Pentium.

Example:

Two 32-bit numbers can be added together as (BXAX) + (DXCX):

ADD AX, CX
ADC BX, DX

SUB instruction (SUBstract)

The SUB instruction has the general format:

SUB <destination> <source>

Where <*destination*> can be a general register or memory location, and <*source*> can be a general register, memory location or an immediate value. The result of the operation is as follows: <destination> == <destination> - <source>. The status indicators changed as a result of this operation are: AF, CF, PF, SF, ZF, OF. The operands can be 8-bit or 16-bit and must be the same size. The subtraction can be seen as an addition with the 2's complement representation of the source operand and the bit CF inverted, in the sense that if transport occurs in the operation (equivalent addition), CF=0 and if transport does not occur in the equivalent addition, CF=1.

For instructions:

MOV CH, 22h
SUB CH, 34h
The result is **-12 (1110 1110)** and the status indicators change as follows:

ZF = 0 (non-zero result)
CF = 1 (loan)
SF = 1 (negative result)
PF = 0 (even parity)
OF = 0 (no overrun)

DEC instruction (DECrement substraction)

The DEC instruction has the general format:

DEC <destination>

Where <*destination*> is an 8-bit or 16-bit register or operand in memory and the meaning of the operation is to decrement the destination value by 1. All status flags are affected except CF (Carry Flag).

SBB Instruction (SuBstract with Borrow)

The SBB instruction has the general format:

SBB <destination>, <source>

Where <*destination*> and <*source*> can be register or operand in memory, 8bit or 16-bit. The result of the operation is as follows: <destination> == <destination> - <source> - CF, so the same as for the SUB instruction, but the CF bit is subtracted from the result. The status indicators changed as a result of this operation are: AF, CF, PF, SF, ZF, OF. This instruction is usually used to subtract numbers larger than 16 bits (in 8086 - 80286) or 32 bits (in 80386, 80486, Pentium).

Example

The subtraction of two 32-bit numbers can be done as follows (BXAX) - (SIDI):

```
SUB   AX, DI
SBB BX, SI
```

Program examples

1. Program that reads a number from the keyboard and displays whether the number is even or not:

```
; Program reads a number and displays a message about parity
dosseg
.model small
.stack
.data

message db 13,10,'Enter number:(<=9)$'
mesg_even db 13,10,'Number entered is even!$'
mesg_odd   db   13,10,'Number entered is odd!$'
.code

pstart:
      mov ax,@data
      mov ds,ax

      mov ah,09
        mov dx,offset message
      int 21h
```

```
        mov ah,01h ; a character is read from the keyboard ;
        ; the ASCII code of the entered character will be in AL
        int 21h
        mov bx,2
        div bx ; divide AX by BX, the remainder will be in AX,
        the rest in DX
        cmp dx,0
        jnz odd
        mov ah,09
        mov dx,offset mesg_even
        int 21h
        jmp end
odd: mov ah,09
       mov dx,offset mesg_odd
        int 21h
end:
       mov ah,4ch
       int 21h; end of the programme

END pstart
```

2. Program that calculates the square of a number entered from the keyboard.

; The program calculates the square of a number (<=256) entered from the keyboard

; The value of the square is calculated in the AX register (maximum value 2^16 = 65536)

```
dosseg
.model small
.stack
.data

nr DB 10,10 dup(0)
r DB 10, 10 dup(0)
message db 13,10,'Enter number:(<=256)$'
square db 13,10,'The square of the number is:$'

.code

pstart:
```

```
        mov ax,@data
        mov ds,ax

        mov ah,09
          mov dx,offset message
        int 21h

        mov ah,0ah
        mov dx,offset nr
        int 21h

        mov cl,nr[1] ; load in CL the number of digits of the number entered
        inc cl          ; in the string it will go to position cl+1
        mov si,1        ; use the SI register as a counter
        xor ax,ax       ;initialise AX with 0
        mov bl,10     ; will multiply by the value 10 that is stored in the BL
multiplication:
        mul bl
        inc si
        mov dl,nr[si]
        sub dl,30h
        add ax,dx
        cmp si,cx
        jne multiply

        mul ax

        xor si,si
        mov bx,10
digit: ; this is where the AX div bx result display starts
        div bx
        add dl,30h
        mov r[si],dl
        inc si
        xor dx,dx
        cmp ax,0
        jne digit
        mov ah,9
        mov dx, offset square
        int 21h
character:
```

```
        dec si
        mov ah,02      ;call function 02 to display a character
        mov dl,r[si]  ;whose ASCII code is in DL
        int 21h
        cmp si,0
        jne character
        jmp end

        mov ah,9
        mov dx,offset square
        int 21h
end:
        mov ah,4ch
        int 21h; stop program

END pstart
```

3. Program that calculates the value of a number raised to a power. Both the number and the exponent (power) are entered from the keyboard.

```
        ; The program calculates a high number to a power

        ; Observation. Since the result is calculated in the AX register which is a
        ; 16-bit register, the maximum value calculated correctly is 2^16= 65536

        .model small
        .stack
        .data

        message1 db 13,10,'Enter number:(<=9)$'
        message2 db 13,10,'Enter power:(<=9)$'
        message_final db 13,10,'Result is: $'
        message_power_0 db 13,10, 'Any number raised to power 0 is 1! $'
        r db 30 dup(0) ; the variable r will store the result

        .code

        pstart:
        mov ax,@data
        mov ds,ax

        mov ah,09
```

```
        mov dx,offset message1
       int 21h
        mov ah,01h ; a character is read from the keyboard
        ; the ASCII code of the entered character will be in AL
        int 21h
        and ax,00FFh
        sub ax, 30h ; the numerical value is obtained
           ; by subtracting the code of 0 in ASCII (30H)
        push ax ; save the value of ax in the stack

        mov ah,09
        mov dx,offset message2
        int 21h
        mov ah,01h ; a character is read from the keyboard
              ; the ASCII code of the entered character will be in AL
        int 21h
        and ax,00FFh
        sub ax, 30h ; the numerical value is obtained
                        ; by dropping its code 0 in ASCII (30H)
        mov cx,ax  ; register CX counts the number of increments
        cmp cx,0
        jne power_0
        mov ah,09
        mov dx, offset message_power_0
        int 21h
        jmp end

power_0:
        pop bx       ;save in BX the value that multiplies
        mov ax,0001
multiplication:
        mul bx
        loop multiplication
        xor si,si
        mov bx,10
digit:
        div bx
        add dl,30h
        mov r[si],dl
        inc si
        xor dx,dx
```

```
        cmp ax,0
        jne digit

        mov ah,9
        mov dx, offset final_message
        int 21h
character:
        dec si
        mov ah,02 ;call function 02 to display a character
       mov dl,r[si]  ;whose ASCII code is in DL
       int 21h
       cmp si,0
       jne character

end:
        mov ah,4ch
        int 21h ; end of programme
END pstart
```

4. Program that checks if a number is a palindrome (a number is called a palindrome if written from right to left or vice versa it has the same value).

```
       ; Program checks if a number or string is palindrome


       dosseg
       .model small
       .stack
       .data
       nr DB 10,10 dup(0)

       message db 13,10,'Enter number:$'
       message_no db 13,10,'Number is not palindromic!$'
       message_da db 13,10,'Number is palindromic!$'

       .code
       pstart:
              mov ax,@data
              mov ds,ax

              mov ah,09
```

```
                mov dx,offset message
               int 21h
                mov ah,0ah
                mov dx,offset nr
                int 21h

                mov si,1
                mov cl,nr[si] ; load in CL the number of digits of the number
                              ;entered
                and cx,00FFh

               mov ax,cx
               mov bl,2
               div bl ; in AL is the division of AX by 2
               and ax,00FFh
               inc ax
               inc cx
               mov di,cx
next_character:
               inc si          ; SI increases from the beginning of the
                               ;string to the middle
               mov bl,nr[di]
               cmp nr[si],bl
               jne is_not
               dec di           ; DI decreases from the end of the string to the middle
               cmp and,ax   ;in the string it will go to the position cl+1
               jne next_character
               mov ah,9
               mov dx,offset message_yes
               int 21h
               jmp end
is_not:
               mov ah,9
               mov dx,offset message_no
               int 21h
end:
               mov ah,4ch
               int 21h; stop program
END pstart
```

5. Program that calculates the sum of the digits of a number entered from the keyboard.

```
; The program calculates the sum of the digits of a number entered from the
keyboard


dosseg
.model small
.stack
.data
nr DB 10,10 dup(?)
result DB 10,10 dup(?)

message db 13,10,'Enter number:$'
message_sum db 13,10,'The sum of the digits of the number is: $'

.code

pstart:
        mov ax,@data
        mov ds,ax

        mov ah,09                ; this displays the initial input message
        mov dx,offset message ; of the number
        int 21h

        mov ah,0ah          ; function 10(0ah) reads a string from
                            ; keyboard in a memory variable
        mov dx,offset nr
        int 21h

        mov si,1
        mov cl,nr[si] ; load in CL the number of digits of the number entered
        and cx,00FFh
        inc cx ;CX now stores the last position in the digit string
        xor ax,ax ; we store the result in AX, which we initialise with zero
next_character:
        inc si ; SI increases from the beginning of the string to the end
        add al,nr[si]

        sub al,30h ; we are writing the ASCII code of zero

        cmp and,cx ; in the line go to the position cl+1
```

```
                jne next_character

                xor si,si       ; SI is the index in the string that will contain the result

  digit: ; this is where the display of the AX result starts
                mov bx,0ah
                div bx
                add dl,30h
                mov result[si],dl
                inc si
                xor dx,dx
                cmp ax,0
                jne digit

                mov ah,9
                mov dx,offset message_sum
                int 21h
character:
            dec if
            mov ah,02       ;call function 02 to display a character
     mov dl,result[and]     ;whose ASCII code is in DL
   int 21h
   cmp si,0
   jne character

  mov ah,4ch
  int 21h ; end of the  program

  END pstart
```