Chapter IV: Case study Intel 8086 processor

I/ Introduction:

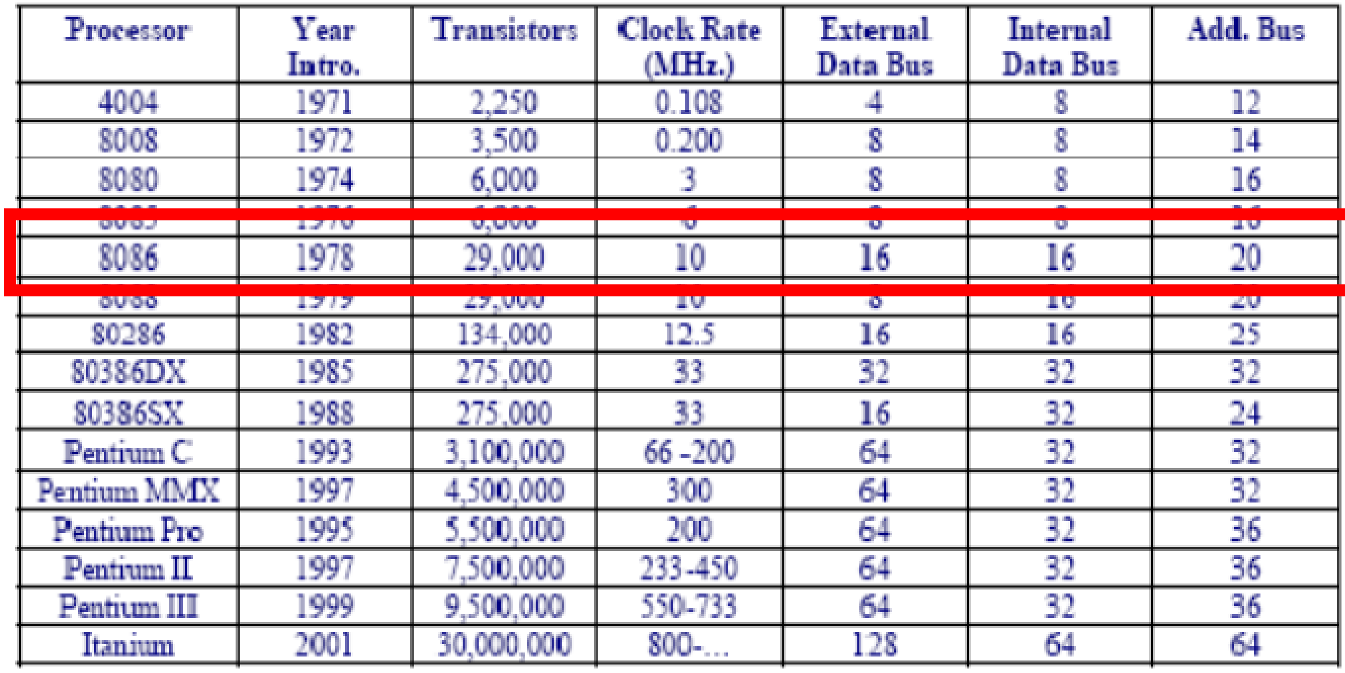
This chapter studies an example of architecture to reinforce all information required from the last chapters. The case is Intel 8086 processor.

We will take a look for the contents and registers in this processor and practice the assembling language.

II/ History and generalities:

Intel introduced the microprocessor in November 1971 with the advertisement, "Announcing a New Era in Integrated Electronics." The fulfillment of this prophecy has already occurred with the delivery of the 8008 in 1972, the 8080 in 1974, the 8085 in 1976, and the 8086 in 1978. During this time, throughput has improved 100-fold, the price of a CPU chip has declined from $300 to $3, and microcomputers have revolutionized design concepts in countless applications. They are now entering our homes and cars.

The principal way in which MPU & microcomputer are categorized in term of the maximum number of binary bits in the data they process that is, their word length. Processor vary in their speed, capacity of memory, register and data bus, below are a brief description of various Intel processor in Table.



Processor	Year Intro.	Transistors	Clock Rate (MHz.)	External Data Bus	Internal Data Bus	Add. Bus
4004	1971	2,250	0.108	4	8	12
8008	1972	3,500	0.200	8	8	14
8080	1974	6,000	3	8	8	16
8085	1976	6,000	6	8	8	16
8086	1978	29,000	10	16	16	20
8088	1979	29,000	10	8	16	20
80286	1982	134,000	12.5	16	16	25
80386DX	1985	275,000	33	32	32	32
80386SX	1988	275,000	33	16	32	24
Pentium C	1993	3,100,000	66 -200	64	32	32
Pentium MMX	1997	4,500,000	300	64	32	32
Pentium Pro	1995	5,500,000	200	64	32	36
Pentium II	1997	7,500,000	233-450	64	32	36
Pentium III	1999	9,500,000	550-733	64	32	36
Itanium	2001	30,000,000	800-…	128	64	64

Intel introduced 8086 microprocessor in 1978. This 16-bit microprocessor was a major improvement over the previous generation of 8080/8085 series of microprocessors.

It has characteristics:

- 133 instructions
- 9 flags
- 1 megabytes memory size
- 64K input/output ports
- 40 processor pins
- 20 address bus
- 16 data lines
- 8086 microprocessors come in many variants, such as 8086-1 and 8086-2. The clock speed of 8086 is 5 MHz, 8086-2 is 8MHz and 8086-1 is 10 MHz. It consists of 29,000 transistors.
- It was mainly designed for assembly languages and high-level programming languages. Also, it can be operated in both single processor and multiprocessor configurations for performing task more accurately and speedily.
- It was the first 16-bit processor having 16 bit ALU and 16-bit external data bus.
- Addressing mode Memory direct/ Memory indirect/ Register/ Immediate/ Indexing

III/ Intel 8086 architecture:

Internal organization of the 8086 microprocessor:

The 8086 consists of two units operating in parallel:

- **the Execution Unit (EU)**: executes the instructions contained in the queue.
- **the Bus Interface Unit (BIU)**: fetches instructions from memory and stores them in a queue.

Both units operate simultaneously, speeding up the program execution process (**pipe-line** operation (we will see later)).

The 8086 microprocessor contains 14 registers divided into 4 groups:

- **General registers**: 4 16-bit registers

AX = (**AH**,**AL**) ;

BX = (**BH**,**BL**) ;

CX = (**CH**,**CL**) ;

DX = (**DH**,**DL**)

They can also be considered as 8 8-bit registers. They are used to temporarily hold data. They are general registers, but can be used for specific operations. Example: AX = accumulator, CX = counter.

- **Pointer and index registers:** we have 4 registers (16-bit registers).

Pointers :

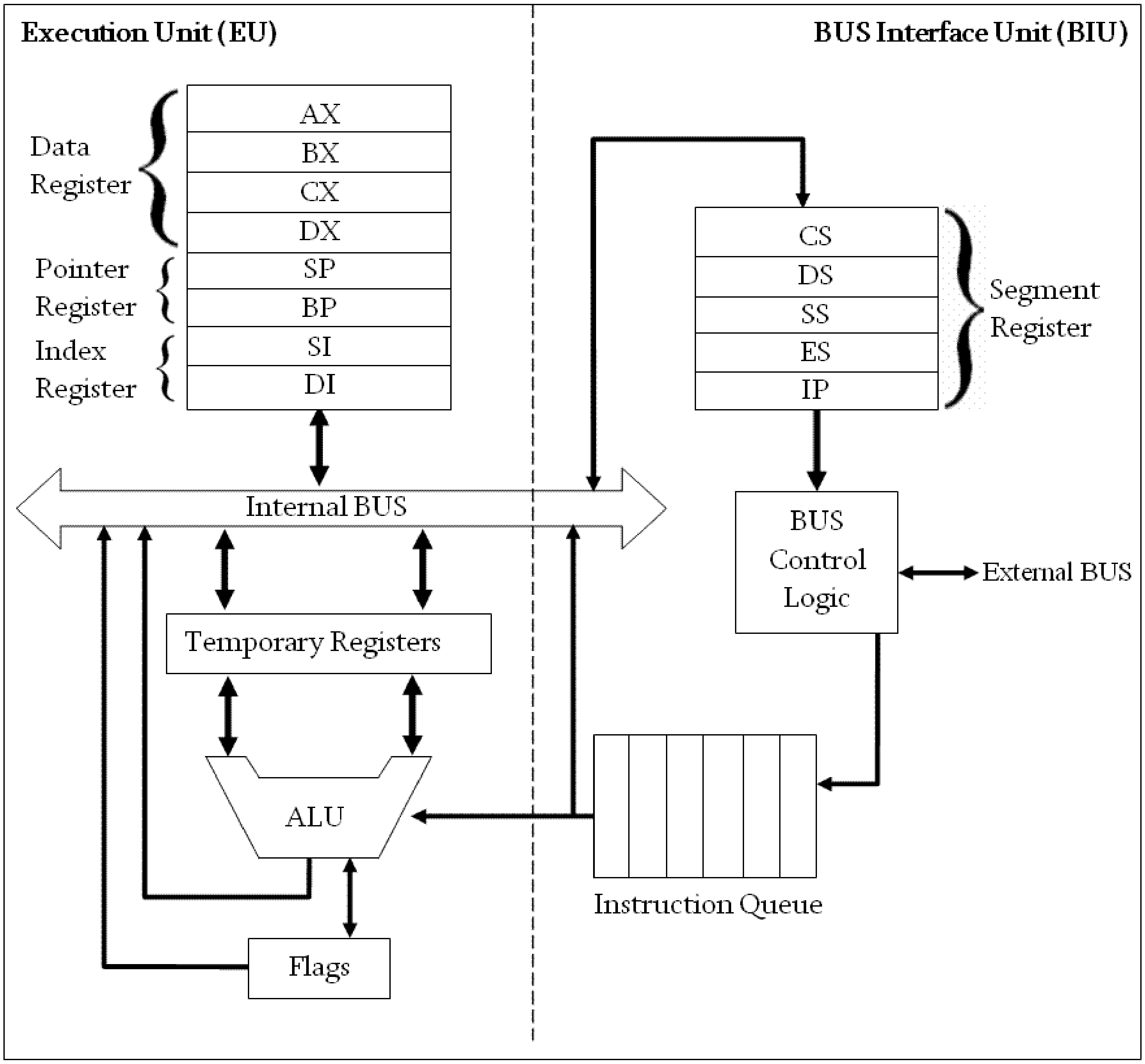
SP: Stack Pointer (the stack is an area where data is saved during program execution);

BP: Base Pointer, used to address data on the stack.

Index :

SI: Source Index ;

DI: Destination Index. They are used to transfer byte strings between two memory areas.



***Note**: Pointers and indexes contain memory cell addresses.*

- **Instruction pointer (ordinal or program counter) and flags (PSW Process Stat Word flags):** we have 2 registers (16-bit registers).

Instruction pointer IP: contains the address of the next instruction to be executed.

PSW Flags :

				O	D	I	T	S	Z		A		P		C
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

CF: carry indicator;
PF: parity indicator;
AF: auxiliary hold indicator;
ZF: zero indicator;
SF: sign indicator;
TF: trap indicator;
IF: interrupt authorization flag;
DF: decrement indicator;
OF: overflow flag.

- **Segment registers:** we have 4 registers (16-bit registers).

CS: Code Segment register;

DS: Data Segment register;

SS: Stack Segment;

ES: Extra Segment, additional segment registers for data;

<u>**Note**</u>: Segment registers, together with pointers and indexes, enable the 8086 microprocessor to address the entire memory.

- **Instruction decoder (ID):** analyzes the instruction's operation code to distribute the various elementary commands.
- **Sequencer:** orders all micro-commands to all units (memory, ALU, etc.) to process the instruction.

<u>**Note**</u>: The registers are used according to the 8086 architectures:

16 bits registers	High part 8 to 15 bits	Low part 0 to 7 bits	use
AX	AH	AL	Accumulator, multiplication, division
BX	BH	BL	Access memory
CX	CH	CL	Counter, repetition, offset
DX	DH	DL	In, out, multiplication, division
SI	/	/	Index source, lods, movs
DI			Index destination, stos, movs
BP,			Base pointer, stack
SP			Stack pointer, stack top

IV/ The structure of the main memory in intel 8086:

The 8086's addressable memory space is 220 = 1,048,576 bytes = 1 MB (**20 address bits**). This space is divided into segments. A segment is a 64 KB (65,536 bytes) memory area defined by its starting address, which must be a multiple of 16.

In such an address, the 4 least significant bits are set to zero. We can therefore represent a segment address with only its 16 most significant bits, the 4 least significant bits being implicitly set to 0.

To designate one of the 216 = 65,536 memory locations contained in a segment, all you need is a 16-bit value.

Thus, a memory location is identified by the 8086 using two 16-bit quantities:

- a segment address;
- a displacement or offset (also called effective address) within this segment.

➙ This method of memory management is called memory segmentation.

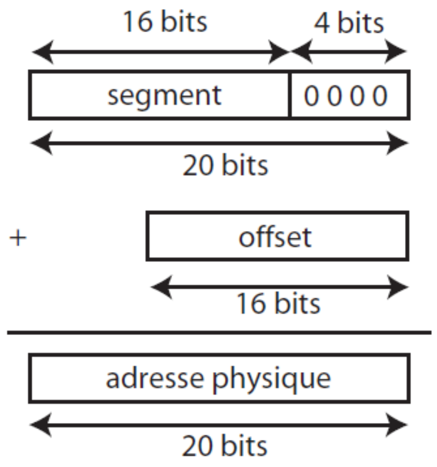
A (**segment,offset**) pair defines a logical address, expressed as ➙ **segment : offset**

The address of a memory cell given as a 20-bit quantity (5 hex digits) is called the physical address, as it corresponds to the value actually sent on the A0 - A19 address bus.

At any given moment, the 8086 has access to 4 segments whose addresses are in the segment registers CS, DS, SS and ES. The code segment contains the program instructions, the data segment contains the data manipulated by the program, the stack segment contains the backup stack, and the additional segment may also contain data.

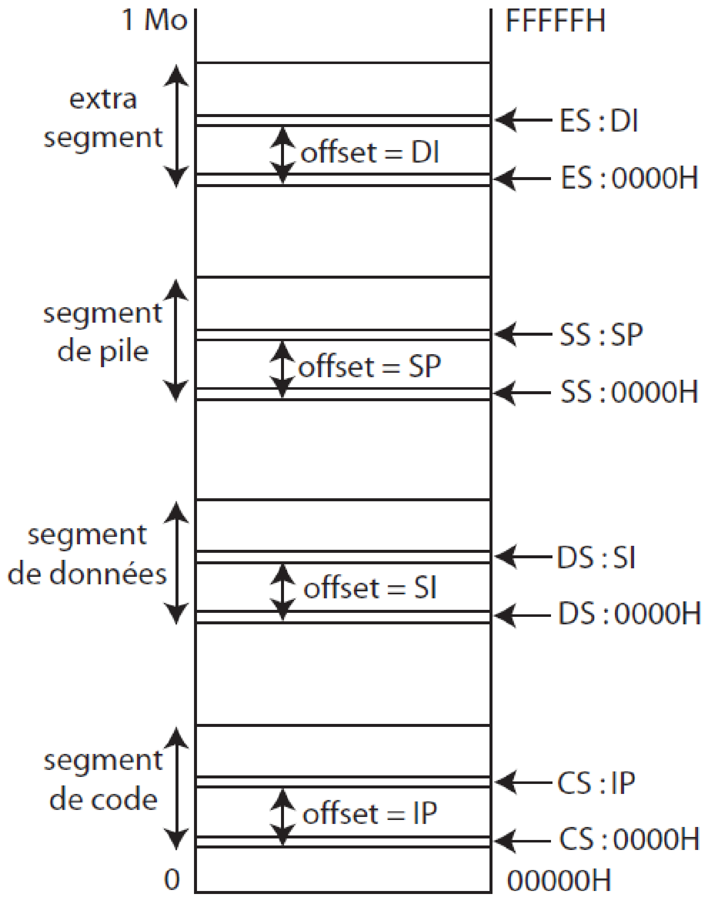
The CS register is associated with the IP instruction pointer, so the next instruction to be executed is at the logical address CS : IP.

The physical address for the location of the instruction is generated by shifting the CS left one hex digit and then adding it to the IP. IP contains the offset address.



physical address = 16 X segment + offset

Similarly, the segment registers DS and ES can be associated with an index register. Example: DS: SI, ES: DI. The stack segment register can be associated with pointer registers: SS: SP or SS: BP.



Note: in other systems, memory is organized into pages - Memory pagination.

V/ What is assembling language:

- ✓ Assembly language or simply Assembler - ASM ➝ is a low-level language that represents machine language in human-readable form.
- ✓ The bit combinations of machine language are represented by so-called "mnemonic" symbols (from the Greek mnêmonikos, relating to memory), ➝ easy to remember.
- ✓ The assembler program converts these mnemonics into machine language to create, for example, an executable file.
- ✓ On early computers, the assembly task was performed manually by the programmer.
- ✓ Each computer is equipped with a processor that performs the computer's arithmetic, logic and control activities.
- ✓ Each processor family has its own set of instructions for handling different operations, such as keyboard input, displaying information on the screen and performing various other tasks. We call this sequence "machine language instructions".
- ✓ A processor only understands machine language instructions, which are sequences of ones and zeros.
- ✓ However, machine language is too obscure and complex to use for software development. So, assembly language is designed for a specific family of processors that represent different instructions in symbolic code and in a more comprehensible form.
- ✓ Assembly language is the symbolic programming language closest to machine language in form and content. Assembly language is useful when :
  - We need to carefully control the steps in our program, down to byte and even bit level.
  - We need to write subroutines for functions that are not provided by other symbolic programming languages, such as COBOL, Java or C.

Advantages of assembly language :

Knowing this basic language makes you more aware of how:

- Programs work with the operating system, processor and BIOS.
- Data is displayed in memory.
- Processor executes instructions.
- Process instruction data.
- A program accesses external device.

The advantages of using assembly language are :

- Requires less memory and execution time.
- Assembler simplifies complex hardware-specific tasks.
- Suitable for urgent tasks.
- Ideal for writing interrupt routines and other memory-resident programs.

Special features:

- Processor-specific language
- Machine instructions
- Assembly language directives: assembly languages have additional directives for assembling data blocks and assigning addresses to instructions by defining labels.
- Machine language reversibility: it is theoretically possible to translate code in both directions without loss of information. The transformation of assembly code into machine language is accomplished by a program called assembler, and in the other direction by a disassembler program. These operations are called assembly and disassembly respectively.
- A disassembler: is a computer program that translates machine language (an executable file) into assembly language (also known as "low-level" language). This operation, disassembly, is the reverse of that performed by an assembler program, assembly.

How it works :

- Computers come with a specific set of basic instructions that correspond to the basic operations the computer can perform. For example, a "Load" instruction causes the processor to move a series of bits from a location in processor memory to a special repository called a register.
- The programmer can write a program using one of these assembly instructions.
- This sequence of assembler instructions, known as the source code or source program, is then specified in the assembler program when we start the program.
- The assembler program takes each program instruction in the source program and generates a corresponding bitstream or pattern (a series of zeros and ones of a certain length).
- The output of the assembler program is called object code or object program in relation to the input source program. The series of zeros and ones that make up the object program is also called machine code.
- We can then run the object program at any time.

II/ The assembler language:

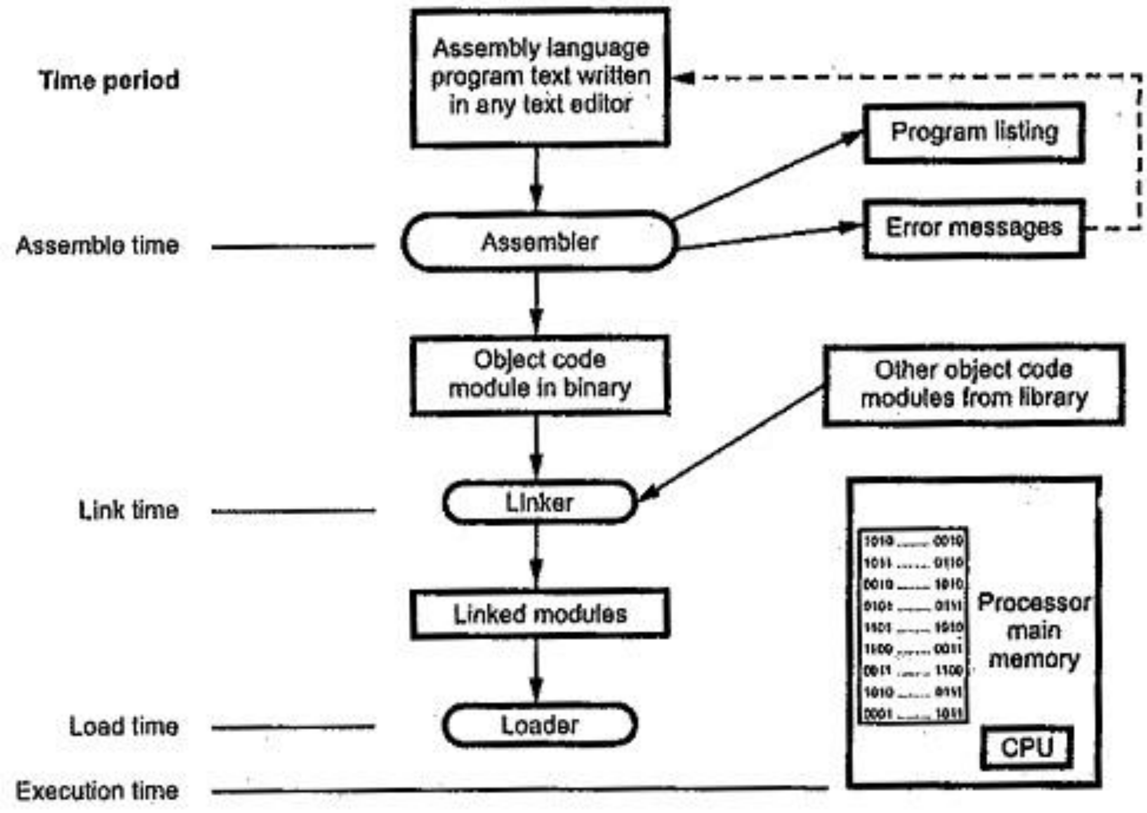
A/ Source file:

Machine language and assembler language:

- ✓ A machine language program is a sequence of instructions and binary data blocks in memory.
- ✓ Assembly language is a symbolic form of a machine-language program
  - human-readable (+/-)
  - symbolic constants, addresses and instruction names
  - arithmetic during assembly - address calculations, constants
  - synthetic instructions (not supported by our assembler)
  - macro instruction expansion (not supported by our assembler)
  - assembly directives (pseudo-instructions)
- ✓ memory allocation, memory initialization, conditional assembly (not supported by our assembler).

To create an assembler program, we go through the following steps:

- ✓ Assembler is a non-interactive utility. The program you wish to translate into machine language (known as an assembler) must be placed in a text file (with the extension **.ASM** under **DOS**).
- ✓ Keyboard entry of the source program requires a program called a text editor.
- ✓ The assembly operation translates each instruction in the source program into a machine instruction. The result of assembly is saved in a file with the extension **.OBJ** (object file).
- ✓ The **.OBJ file** is not directly executable. In fact, it often happens that an executable program is built from several source files. You need to link the object files using a utility called a link editor (even if you only have one). The linker produces an executable file with the **.EXE** extension.
- ✓ The **.EXE** file is directly executable. A special utility in the operating system (**DOS** in this case), the loader, is responsible for reading the executable file, placing it in main memory and then launching the program.



- ✓ There are several software packages that allow you to switch between the three phases shown in the previous figure: MASM (Microsoft Assembler: with LINK as link editor), TASM (Turbo Assembler: with TLINK as link editor) and NASM etc...
- ✓ Other file formats can be generated from an object file for systems other than the PC (IBM compatible). The best-known forms are INTEL HEX, ASCII HEX etc. ...
- ✓ The assembler is used to be closer to the machine, to know exactly what instructions are being generated (to control or optimize an operation):

- programming of basic machine systems (keyboard control, screen control, etc.),
- certain parts of the operating system,
- control of new peripherals (printers, scanners, etc.).
- access to system resources,

✓ Like any program, a program written in assembler (source program) comprises definitions, data and instructions, each of which is written on a line of text.

✓ Data are declared by directives, special key words understood by the assembler and therefore intended for the assembler. Instructions are intended for the microprocessor.

B/ <u>Instructions</u>:

✓ An assembly language instruction in a source file has the form:

{Label} {Mnemonic {Operand}} {;Comment}

✓ Each entity is a field → label, mnemonic, operands and a comment field.

❖ ***Label field***: is (usually) an optional field containing a symbolic label for the current instruction → mark target lines for use in GOTO jumps.

- We can also specify variable names, procedure names and other entities using symbolic labels.
- Some mnemonics, however, require a label, while others do not.
- In general, labels should always begin in the first column (this makes programs easier to read).

❖ ***Mnemonic field:***

- A mnemonic is an instruction name (e.g. mov, add, etc.). The mnemonic → memory load MOV is much easier to remember than the binary equivalent of MOV.
- - The mnemonic field contains an assembler instruction.
- - Instructions are divided into three classes: 80x86 Machine instructions, assembler directives and pseudo-opcodes.
  - ✓ 80x86 instructions, of course, are assembler mnemonics that correspond to the actual 80x86 Instructions.
  - ✓ Assembler directives are special instructions that provide information to the assembler but do not generate any code. Example: segment, equ, assume, end, ...
  - ✓ A pseudo-opcode is a message to the assembler, just like an assembler directive, however a pseudo-opcode will emit bytes of object code. Example: byte, word, dword, tbyte, ...

❖ ***The operand field:***

- The operand field contains the operands or parameters for the instruction specified in the mnemonic field.
- Operands never appear on lines by themselves
- The type and number of operands (zero, one, two or more) depend entirely on the specific instruction.

❖ ***The comments field:***

- The comment field is used to annotate each line of program source code.

- Note that the comment field always begins with a semicolon.
- When the assembler processes a line of text, it completely ignores everything on the source line following a semi colon.

Each assembly language instruction then appears on its own line in the source file:

Example:

```
  ET1 : MOV  AX , 500   ; set the value 500 in the AX register
; this section to position the cursor
; left position:
           mov X, 0
           mov Y, 0
; change cursor position
; screen to clear the video display:
           ; etc.
```

C/ Assembler directives:

- A directive is an information provided by the programmer to the compiler. It is not transformed into a machine language instruction. It therefore adds no bytes to the compiled program. So directives are declarations that guide the assembler.
- A directive is used, for example, to create memory space for variables, to define constants, etc...
- To declare a directive, use the following syntax:

{Name} Directive {operand} { ; comment}

- The operand field depends on the directive
- The comment field has the same properties as seen in the previous paragraph.
- The Name field indicates the name of variables: this is an optional field (depending on the directive).

D/ Location or line counter:

- **Remember**: all addresses in 80x86 memory space consist of a segment address and an offset within that segment.
  - ✓ The assembler, in the process of converting your source file into object code, must keep track of the offsets in the current segment.
- The location (line) counter is an assembler variable that manages this.
- Each time a segment is created in the source file, the assembler associates the current value of the location counter with it. The location counter contains the current offset in the segment.
- The value of the location counter varies throughout the assembly process.
- MASM increments the location counter for each byte written to the object code file.
  - ✓ For example, MASM increments the location counter by two after encountering mov ax, bx, since this instruction is two bytes long.

- The **or**, **and** and **xor** instructions are each three bytes long.
- the **mov** instruction is two bytes long
- the remaining instructions are all one byte long
- If these instructions appear at the beginning of a segment, the location counter would be the same as the numbers immediately to the left of each instruction above.
- For example, the or instruction above starts at offset zero. Since the **or** instruction is three bytes long, the following and instruction follows at offset three. Similarly, and is three bytes long, so **xor** follows at offset six, ...

```
0: or ah, 9
3: and ah, 0c9h
6: xor ah, 40h
9: pop cx
A: mov al, cl
C: pop bp
D: pop cx
E: pop dx
F: pop ds
10: ret
```

E/ Symbols:

- A symbol, identifier or label is a name associated with a particular value.
  - ✓ An offset in a segment, a constant, a string, a segment address, an offset in a record, or even an operand for an instruction.
- In all cases, a label provides the ability to represent a value otherwise incomprehensible with a familiar name.
- A symbolic name consists of a sequence of letters, digits and special characters, with the following restrictions:
  - ✓ A symbol cannot begin with a numeric digit.
  - ✓ A name can have any combination of upper- or lower-case alphabetic characters (assembler treats them the same).
  - ✓ A symbol can contain any number of characters, but only the first 31 are used.
  - ✓ The symbols _, $, ? and @ can appear anywhere in a symbol.
  - ✓ A symbol cannot match any name that is a reserved symbol.
  - ✓ Register names are also reserved

%out	.186	.286	.286P
.287	.386	.386P	.387
.486	.486P	.8086	.8087
.ALPHA	.BREAK	.CODE	.CONST
.CREF	.DATA	.DATA?	.DOSSEG
.ELSE	.ELSEIF	.ENDIF	.ENDW
.ERR	.ERR1	.ERR2	.ERRB
.ERRDEF	.ERRDIF	.ERRDIFI	.ERRE
.ERRIDN	.ERRIDNI	.ERRNB	.ERRNDEF
.ERRNZ	.EXIT	.FARDATA	.FARDATA?
.IF	.LALL	.LFCOND	.LIST
.LISTALL	.LISTIF	.LISTMACRO	.LISTMACROALL
.MODEL	.MSFLOAT	.NO87	.NOCREF
.NOLIST	.NOLISTIF	.NOLISTMACRO	.RADIX
.REPEAT	.UNTIL	.SALL	.SEQ
.SFCOND	.STACK	.STARTUP	.TFCOND
.UNTIL	.UNTILCXZ	.WHILE	.XALL
.XCREF	.XLIST	ALIGN	ASSUME
BYTE	CATSTR	COMM	COMMENT
DB	DD	DF	DOSSEG
DQ	DT	DW	DWORD
ECHO	ELSE	ELSEIF	ELSEIF1
ELSEIF2	ELSEIFB	ELSEIFDEF	ELSEIFDEF
ELSEIFE	ELSEIFIDN	ELSEIFNB	ELSEIFNDEF

END	ENDIF	ENDM	ENDP
ENDS	EQU	EVEN	EXITM
EXTERN	EXTRN	EXTERNDEF	FOR
FORC	FWORD	GOTO	GROUP
IF	IF1	IF2	IFB
IFDEF	IFDIF	IFDIFI	IFE
IFIDN	IFIDNI	IFNB	IFNDEF
INCLUDE	INCLUDELIB	INSTR	INVOKE
IRP	IRPC	LABEL	LOCAL
MACRO	NAME	OPTION	ORG
PAGE	POPCONTEXT	PROC	PROTO
PUBLIC	PURGE	PUSHCONTEXT	QWORD
REAL4	REAL8	REAL10	RECORD
REPEAT	REPT	SBYTE	SDWORD
SEGMENT	SIZESTR	STRUC	STRUCT
SUBSTR	SUBTITLE	SUBTTL	SWORD
TBYTE	TEXTEQU	TITLE	TYPEDEF
UNION	WHILE	WORD	

Examples

```
     jmp target
```

- **Valid symbols**

```
L1          Bletch     RightHere
Right_Here       Item1            __Special
$1234            @Home            $_@1
Dollar$     WhereAmI?  @1234
```

- **Invalid symbols**

```
1TooMany - starts with a number.
Hello.There - contains a point in the middle.
$ - we can't have $ or ? alone.
LABEL - reserved symbol.
Right Here - the space.
Hi,There - , prohibited
```

F/ Constants:

A literal constant:

A literal constant is a constant whose value is implied from the characters that make it up.

123
3.14159
"Literal String Constant"
0FABCh
'A'
<Text Constant>

- The representation of a literal constant corresponds to what we would normally expect for its "real value".
- Literal constants are also called non-symbolic constants, because they use the real representation of the value, rather than a symbolic name.

An integer constant:

- An integer constant is a numerical value that can be specified in binary, decimal or hexadecimal.
- All integer constants must begin with a decimal digit, including hexadecimal constants.
- To represent the value "FDED", specify 0FDEDh.
- The first decimal digit is required by the assembler to differentiate between symbols and numeric constants.

Name	Base	Valid Digits
Binary	2	0 1
Decimal	10	0 1 2 3 4 5 6 7 8 9
Hexadecimal	16	0 1 2 3 4 5 6 7 8 9 A B C D E F

A string constant:

- A string constant is a sequence of characters enclosed in apostrophes or quotation marks.

"is a string"
'like this'

A real constant:

- The decimal form is just a sequence of digits containing a decimal point at a certain position of the number.

625.25 ***-6.02e-10***

A text constant:

- A text constant consists of a sequence of characters surrounded by "<" and ">" symbols.
- For example, the text constant 5[bx] is normally written as <5[bx]>.

Declaration of a constant using equations:

symbol equ expression
symbol = expression

"=" for integers :

- The purpose of the "=" directive is to define symbols that have an integer (or a single character).
- This directive does not allow real, string or text values.

```
NumElements = 16
Array byte NumElements dup (?)
mov cx, NumElements
mov bx, 0
ClrLoop: mov Array[bx], 0
inc bx
loop ClrLoop
```

The directive “equ”:

- The **equ** directive provides almost a superset of the capabilities of the **"="** and **textqu** directives.
- It authorizes operands that are numeric, textual or literal string constants.

```
One          equ 1
Minus1       equ -1
Essaye       equ ’Y’
StringEqu    equ  “Hello student”
TxtEqu       equ <4[si]>
HTString byte StringEqu ;same HTString equ “Hello student”
mov   ax, TxtEqu ;same mov ax, 4[si]
mov   bl, One ;same mov bl, 1
cmp   al, try ;same cmp al,  ‘Y’
```

G/ Variables:

The directives used to declare variables are : DB, DW, DD, DF, DP, DQ, DT

DB (Define Byte) :

This directive defines an 8-bit variable: in other words, it reserves a memory space of one byte: so the values that can be stored for this directive range from 0 to 255 (for unsigned numbers) and from -128 to 127 for signed numbers.

Name DB expression

```
Vil DB 12H; Defines a variable (one byte) with value Initial 12.
Tab DB 18H, 15H, 13H ; defines a 3 table cells
                 ;(3 octet) Starts from TAB address.
Mess DB 'ISET'; also defines an array, but the values of each cell
           ;  simply the ascii code of each letter.
Name DB  ? ; defines an 8-bit variable with any initial value.
```

DW (Define Word) :

This directive defines a 16-bit variable: in other words, it reserves a memory space of one word: so the values that can be stored for this directive range from 0 to 65535 (for unsigned numbers) and from -32768 to 32767 for signed numbers.

Name DW expression

```
TT1 DW    500H ; reserves two memory cells (one word) from address TT1.
TAB1 DW  10H,11H,14H ; reserve a table of 6 squares each value will be put on two cells
YY   DW    ? ; reserves a word in memory of any initial value.
```

DD (Define Double)

This directive reserves a 32-bit space (4 memory cells or 4 bytes).

Name DD expression

Example : ff DD 15500000F

Dup directive:

To declare an array of n cells, all initialized to the same value, use the dup directive:

```
tab DB 100 dup (15) ; 100 octets as 15
y DW 10 dup (?) ; 10 mots de 16 bits not initialized
```

The Word PTR et Byte PTR directives :

are directives used to remove size ambiguity in the Assembler

```
MOV byte ptr [BX], val ; concerns 1 octet
MOV word ptr [BX], val ; concerns 1 word of 2 octets
```

H/ Segments:

The SEGMENT directive controls the relationship between object code generation and the management of the logical segments generated.

The SEGMENT instruction is used to:

- control the placement of object code in specific segments.
- associate symbols representing addresses with a segment, considering their value as a displacement relative to the beginning of the segment.
- specify directives for the linker (segment name, operand fields of the SEGMENT instruction determining segment processing by the linker); this information is passed as is:

SEGMENT : Name SEGMENT operand

Name ends

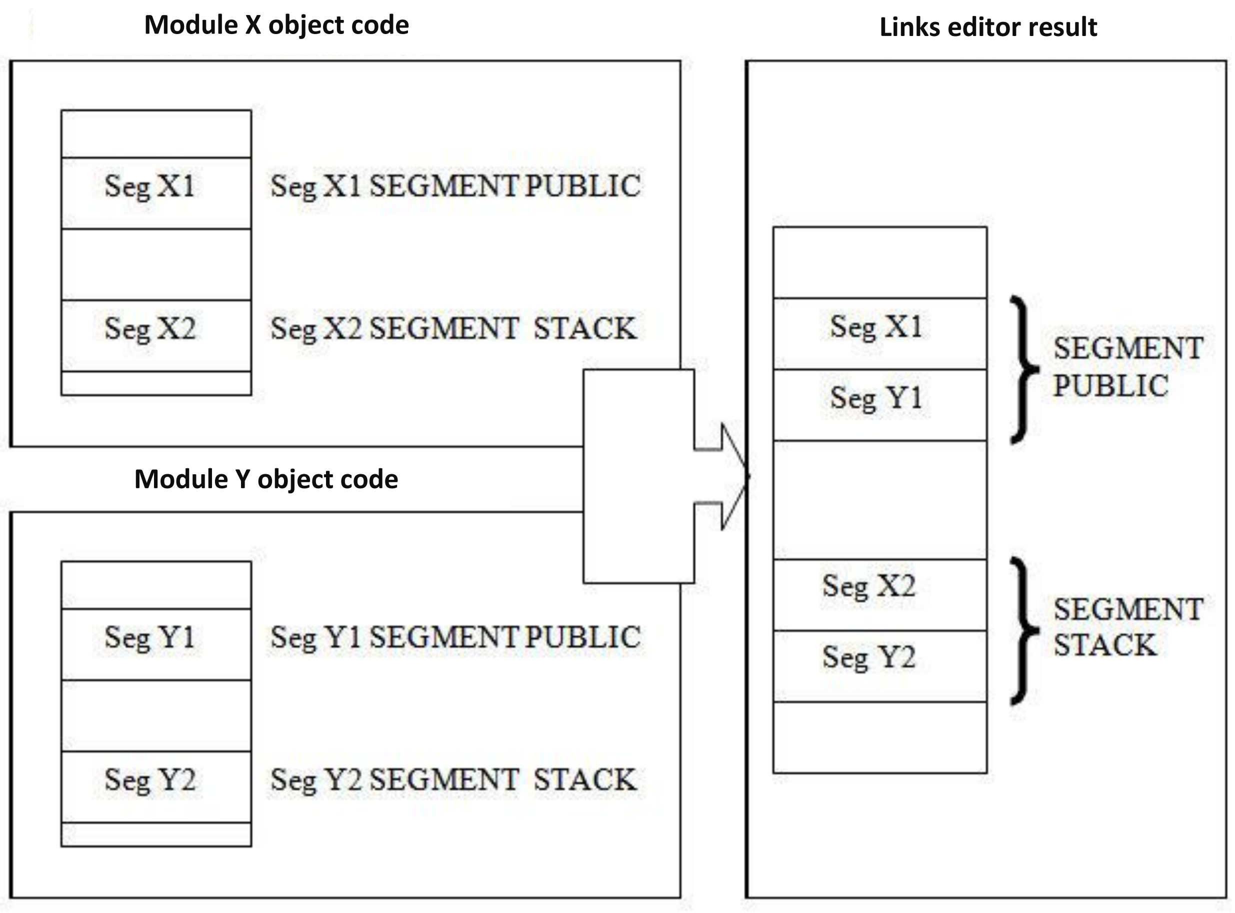
The 'Name' label field is used to:

- indicate the name of the segment.
- establish an obvious relationship between pairs of SEGMENT and ENDS instructions.

Alternating segments :

A program can alternate between different segments to generate code:

The SEGMENT instruction is used to re-open an existing segment (i.e. SEGMENT either creates a new segment, or opens a segment to add additional code).



The operands of the SEGMENT statement determine how the linker will handle the segment:

$$\text{Nom} \qquad \text{SEGMENT} \quad \left\{ \begin{array}{l} \text{COMMON} \\ \text{PUBLIC} \\ \text{STACK} \\ \text{MEMORY} \\ \text{AT adr} \end{array} \right. \quad \left\{ \begin{array}{l} \text{BYTE} \\ \text{WORD} \\ \text{PARA} \\ \text{PAGE} \end{array} \right. \qquad \text{TITO}$$

COMMON :

All segments with the label (class) will be placed at the same base address (in a contiguous block); zones of type (COMMON) with different names ((class)) will be placed one behind the other.

PUBLIC :

All segments with this qualifier will be grouped together in a single resulting segment, one behind the other.

STACK :

Only one segment with this qualifier is accepted, intended for stack management.

MEMORY:

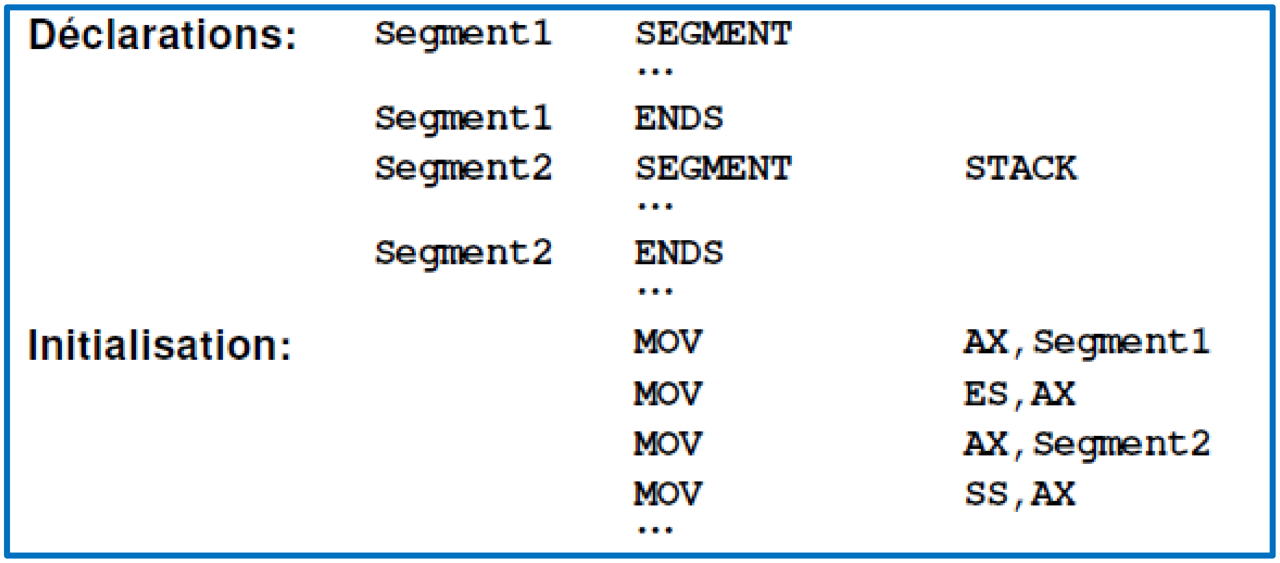
The first segment with this qualifier will be placed at a memory position above any other segment; if there are more segments of this kind, they will be treated as segments of type (COMMON).

AT :

Tags defined in such a segment are defined as being relative to the value ((adr)) 16) * 16.

Managing basic registers :

At the start of execution, the program must initialize the base registers: their contents must correspond to the location of the physical segments, containing the code of the corresponding logical segments:



```
Déclarations:   Segment1    SEGMENT
                            ...
                Segment1    ENDS
                Segment2    SEGMENT         STACK
                            ...
                Segment2    ENDS
                            ...
Initialisation:             MOV             AX,Segment1
                            MOV             ES,AX
                            MOV             AX,Segment2
                            MOV             SS,AX
                            ...
```

The assembler creates a symbol designating the logic segment; this symbol can be used as an external variable (a symbol defined in another module);

➔ the linker defines the value of the symbol as the segment origin (= the value of the base address / 16).

Note: The contents of the CS and SS segment registers (as well as the IP and SP registers) are defined by the operating system before program execution begins (segment containing the code, small stack provided by the system).

However, it is preferable to redefine the stack (SS and SP) to ensure that a sufficient memory area is available during program execution.

Association of logic segments and base registers (= physical segments)

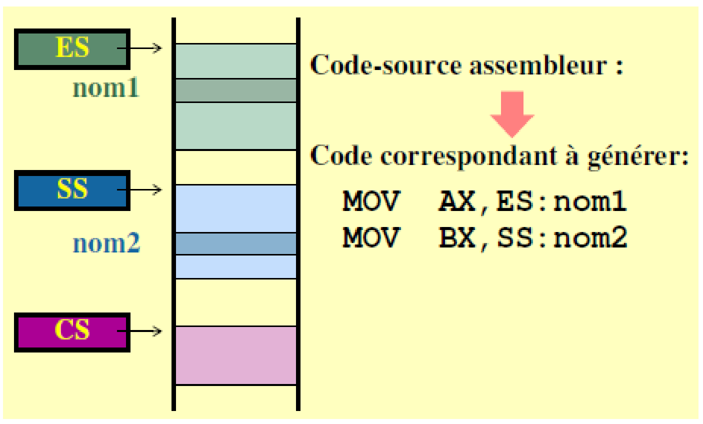
```
        CS          CS
        ES          ES
ASSUME  SS  :nom1[, SS  :nom2 ... ]
        DS          DS
```

The **ASSUME** pseudo-operation tells the assembler which segment register to use to access the operands:

- as a result, the assembler associates the base register(s) (CS, ES, SS, DS) with the logical segment name "namek" ;
- thereafter, the assembler will automatically add segment prefixes where necessary.

Note: The assembler will only add a prefix to instructions where the default choice of base register (see "Registers") does not provide the correct result.

```
seg1  SEGMENT

nom1  DW

seg1  ENDS
seg2  SEGMENT  STACK

nom2  DW

seg2  ENDS
seg3  SEGMENT

ASSUME  ES:seg1,SS:seg2,CS:seg3…

MOV  AX,seg1
MOV  ES,AX
MOV  AX,seg2
MOV  SS,AX

MOV  AX,nom1
MOV  BX,nom2
```



```
MOV   AX,ES:nom1
MOV   BX,SS:nom2
```

```
INCLUDE MACRO.LIB ; add the instructions contained in the MACRO.LIB
pile SEGMENT STACK ; stack reservation
DW 100 DUP (?) ; words without initialization
haut LABEL WORD ; word address according to stack
pile ENDS
data SEGMENT PUBLIC ; where the program data will be placed
data ENDS
code SEGMENT PUBLIC
ASSUME CS:code,SS:pile,DS:data ;
start LABEL FAR
MOV AX,data ; linking DS to the data segment
MOV DS,AX ; (No  MOV imm. DS !)
MOV AX,pile
MOV SS,AX
MOV SP,haut ; top of the stack (empty!)
EXIT
code ENDS
END start ; end of source code, address of first instruction
```

Relationship between location counter and segment :

- It defines the relative addresses at segment level.
- There is one counter per segment
- The current counter value indicates the position where the next byte in the object code will be placed.
- To initialize the position counter value, use the following syntax

ORG value
EVEN must be even

- $ this function returns the current value of the position counter.

THE 8086 ASSEMBLER INSTRUCTION SET

The 8086 processor instructions can be divided into 7 groups:

- Data transfer instructions
- Arithmetic instructions
- Bit instructions (logic)
- Program jump instructions
- String instructions
- Process control instructions
- Interrupt instructions

I/ Data transfer instructions:

A/ Instructions for general use:

1/ MOV Instruction:

MOV operand1, operand2

It copies the contents of the second operand to the first.

Operand1 = Destination can be register or memory address

Operand2 = Source can be immediate value, register or memory address

- MOV register, memory
- MOV memory, register
- MOV register, register
- MOV memory, immediate
- MOV register, immediate

Register = AX, BX, CX, DX, AH, AL, BH, BL; CH, CL, DH, DL, DI, SI, BP, SP

Memory = [BX], [BX + SI + 7], variable

Immediate = 5, -24, 3FH, 100111B, ...

Not allowed = MOV memory, memory / MOV CS or IP

Address calculation (View in Chapter III) :

@physical = 16*segment + offset

Example : DS = 100 (segment data)

BX = 30 and SI = 70

[BX + SI] + 25 = (100 * 16 + 30) + 70 + 25 = 1725

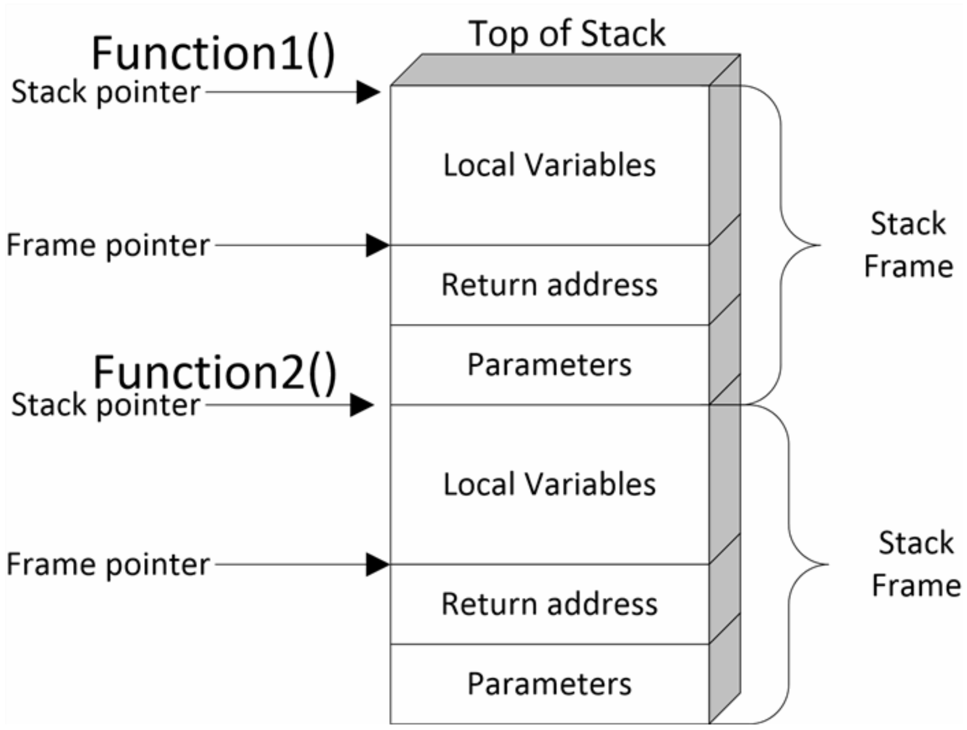
- **Based addressing** = base register content + offset (registers SS, CS, DS, ES)
- **Indexed addressing** = index register content + register content (BP, SI, SP, DI registers)
- **Indexed base addressing** = index register content + base register content + offset
- **Relative addressing** = IP register content + operand (used in jumps and branches)

2/ PUSH Instruction:

The stack is the memory area containing the execution context of functions in programs. Each time the program starts executing a function, it sets up a stack frame, which is a dynamic data structure. The stack contains local variables and parameters passed to the function.

Two registers work with the stack: SP (Stack Pointer) and BP (Base Pointer). SP contains the address of the last occupied cell, called the stack vertex, and the BP register contains the stack base address.

The stack is managed according to the LIFO (Last In First Out) principle, so that when a new item of data is stacked on top, it is unstacked from the top when it is deleted.



PUSH stacks CPU registers on top of the stack

PUSH source

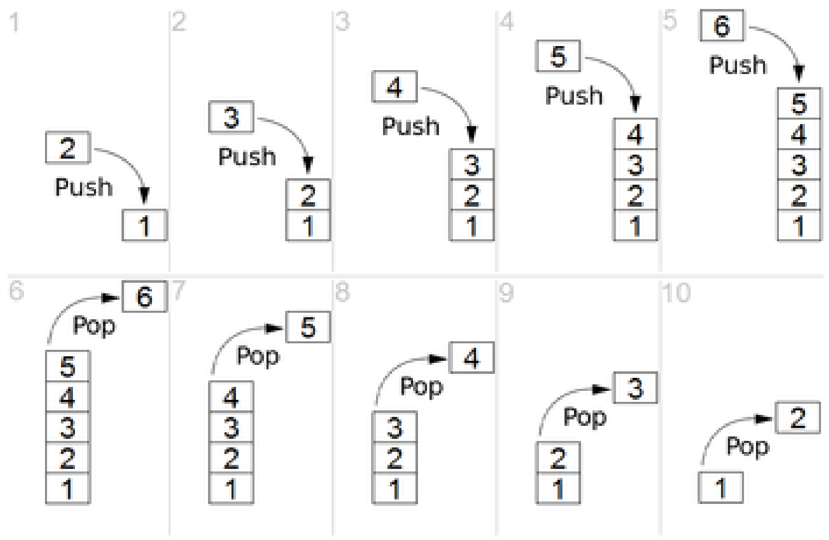
Source can be register, memory, immediate

3/ POP Instruction:

It is used to unstack CPU registers from the top of the stack.

POP destination

Destination can be register, memory



```
ORG 100h

MOV AX, 1234h
PUSH AX ; saves the value of AX in stack.
MOV AX, 5678h ; changes the value AX .
POP AX ; restore the original Value of AX.
RET
END
```

4/ PUSHA Instruction:

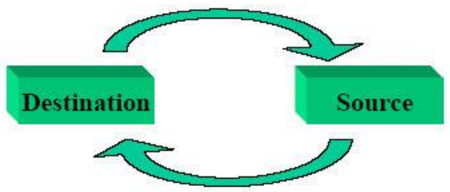
This instruction stacks all the microprocessor's internal registers on the stack.

5/ POPA Instruction:

This instruction unloads all the microprocessor's internal registers onto the stack.

6/ XCHG Instruction:

It switches the source to the destination as follows:



XCHG AX, BX ; It swaps between AX and BX
Example: AX=0123 and BX=5678
after executing the instruction, we'll have
AX=5678 and BX=0123

7/ XLAT Instruction:

This instruction is used to convert data from one code to another, in fact it places in the AL accumulator the contents of the memory cell addressed in base address (8 bits), the base being the BX register and the offset being AL itself in the DS segment.

(AL)<--------[(BX) + (AL)]

XLAT tab_source

Example: converting 4-bit binary code into ASCII-coded hexadecimal digits

```
Tab db ‘0123456789ABCDEF’

MOV AL, 1110B ; loading the value to be converted (07) MOV BX, OFFSET TAB pointed
                    ; on the table
                XLAT ; Al is charged by the code ASCII of ‘E’
```

B/ Address transfer instructions:

1/ LEA Instruction (Load Effective Adress) :

It transfers the offset address of a memory operand to a 16-bit register (pointer or index). This command has the same function as the MOV instruction with offset, but is more powerful, as it can be used with any addressing technique.

LEA register, operand

16-bit register and 16-bit hard operand

2/ LDS/LES Instruction:

This instruction loads the segment and offset of an address.

LDS load pointer with DS

LES load pointer with ES

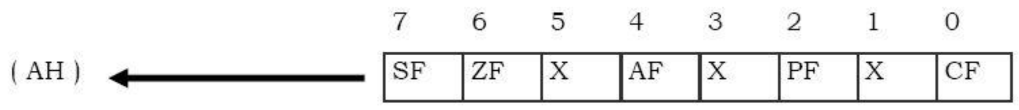
Instead of: **MOV BX, offset tab_val**
MOV AX, seg tab_val
MOV DS, AX
We write **LDS BX, tab_val**

C/ Indicator instructions:

1/ LAHF/SAHF Instruction:

LAHF (Load AH from Flags): places the low byte of the status register (FLAGS) in the AH register as follows:

LAHF



SAHF (Store AH into Flags): Places the contents of AH in the low byte of the status register (FLAGS).

SAHF

2/ PUSHF/POPF Instruction:

PUSHF: Stacks the entire status register (FLAGS).
POPF: Unstacks the status register (FLAGS).

D/ Input/output instructions:

IN/OUT Instruction:

It is used to retrieve data from a port (i.e. from the periphery) or to restore data to a port. In both cases, if a byte is to be sent or received, the AL accumulator is used, and if a word is to be sent or received, the AX accumulator is used.

IN Accumulateur, DX

OUT DX, Accumulateur

DX: contains the port address.
ACCUMULATOR: contains the data (to be received or stored).

```
MOV AH, 9

INT 21H ; string display

OUT DX, AL ; sets the byte in AL to the port specified by DX
```

II/ Arithmetic instructions:

Arithmetic instructions can handle four types of numbers:

- unsigned binary numbers

- Signed binary numbers.
- unsigned binary coded decimal (DCB) numbers.
- DCB numbers, unsigned.

A/ Addition:

1/ ADD Instruction:

ADD destination, source

Destination = destination + source

Example:

ADD AX, BX ; AX = AX + BX (16-bit addition) ADD AL,BH
AL = AL + BH (8-bit addition)
ADD AL, [SI]; AL = AL + the contents of the memory cell pointed to by SI
ADD [DI], AL ; the contents of the memory cell pointed to by DI is added with AL, the result is put into the memory cell pointed to by DI
Interdiction : ADD [DI], [SI]
ADD tab2, tab2
ADD tab, AX

2/ ADC Instruction (addition with carry):

ADC destination, source

CF holdback and result put into destination = destination + source + carry
Example:

ADC AX,BX ; AX = AX + BX + CF(16-bit addition)
ADC AL,BH ; AL = AL + BH + CF(8-bit addition)
ADC AL,[SI] ; AL = AL + the contents of the memory cell pointed to by SI + CF
ADC [DI],AL ; the content of the memory cell pointed to by DI is added with AL + CF ; the result is placed in the memory cell pointed to by DI
Note: CF represents the carryover from a previous addition

3/ INC Instruction (Incrementation):

INC destination

Increment destination content destination = destination + 1
Example:

INC AX INC AL INC [SI]
An immediate value cannot be incremented

4/ AAA/DAA Instruction (ASCII/Decimal Addition Ajustement):

The addition of two BCD numbers sometimes generates a result that is not a BCD number, so corrections must be made to these numbers to obtain a consistent result. This instruction examines the lower fourth of AL and checks whether it is consistent or not:

- If yes (sets AF and CF to zero for information) deletes AL's top quarter.
- else :
  - Adds 6 to AL
  - Adds 1 to AH
  - Clears AL's top quarter
  - Sets AF and CF to 1 (for information)

Example: we want to add 73 + 88 in BCD

```
 73            0111 0011
        +
 88            1000 1000
_________________________
 163           1111 1011
```

Note that neither 1111 nor 1011 is a BCD number, so we're going to add 6 to the first fourth, so the operation becomes :

```
      1111 1011
+          0110
___________________
    =      10000 0001
    +      0110
___________________
    =      0110 0001        (63)
```

Highest byte :

```
      00000001
+     00000000
__________________
=     00000001            (01)
```

resulting in 161

B/ Subtraction:

1/ SUB Instruction:

SUB destination, source
Destination = destination – source

Example :

SUB AX, BX (16 bits)
SUB AL, BH
SUB AL, [SI]
SUB [DI],AL

Note : same interdiction as ADD

2/ SBB Instruction (Subtraction with carry):

SBB destination, source
Destination = destination – source – carry

Example :

SBB AX, BX AX = AX – BX - CF

3/ DEC Instruction (decrementation):

DEC destination
Destination = destination - 1

4/ NEG Instruction (Negatif):

NEG destination

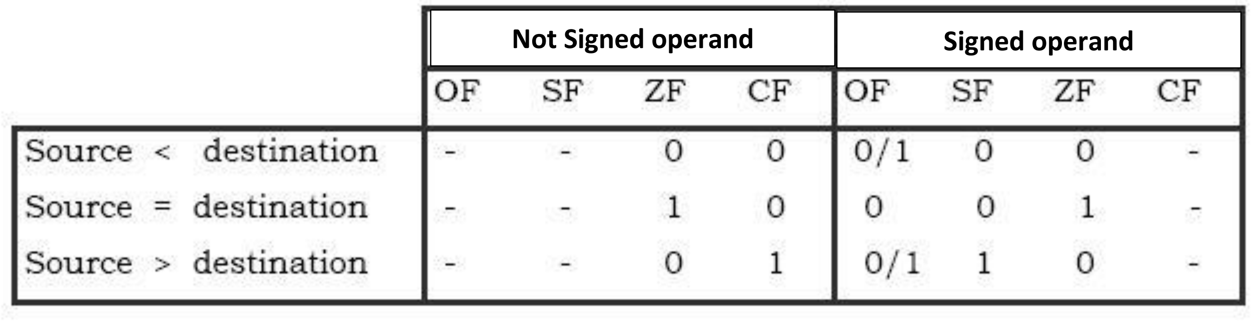
Destination = 0 - destination have the second complement of an operand destination =(-destination) CF=1.

Indicators affected by this operation are **AF, CF, OF, PF, SF, ZF.**

5/ CMP Instruction (Comparison):

CMP destination, source

Destination - source and the result is not saved in the destination, we will have a comparison and the affected indicators are **AF, CF, OF, PF, SF, ZF**



	Not Signed operand				**Signed operand**			
	OF	SF	ZF	CF	OF	SF	ZF	CF
Source < destination	-	-	0	0	0/1	0	0	-
Source = destination	-	-	1	0	0	0	1	-
Source > destination	-	-	0	1	0/1	1	0	-

Example :

CMP AL, BL **CMP AX, 200H** **CMP AX, BX** **CMP [DI], CH**

6/ AAS/DAS Instruction :

Same as AAA, DAA

C/ Multiplication:

1/ MUL Instruction (multiplication for not signed numbers):

MUL source

MUL performs an unsigned multiplication of the source operand with the accumulator

- If the source is a byte, it will be multiplied by the AL accumulator, and the 16-bit result will be stored in the AX register.
- If the source is a word then it will be multiplied with the AX accumulator, the 32-bit result will be stored in the AX and DX register pair.

(AX) <------------- (AL) X Source (byte)
(AX)(DX) <------------ (AX) X Source (words)

Multiplication	Operand 1	Operand 2	Result
Byte * Byte	AL	Register or memory	AX
Word * Word	AX	Register or memory	DX AX
Word * Byte	AL = Byte, AH=0	Register or memory	DX AX

2/ IMUL Instruction (multiplication for signed numbers):

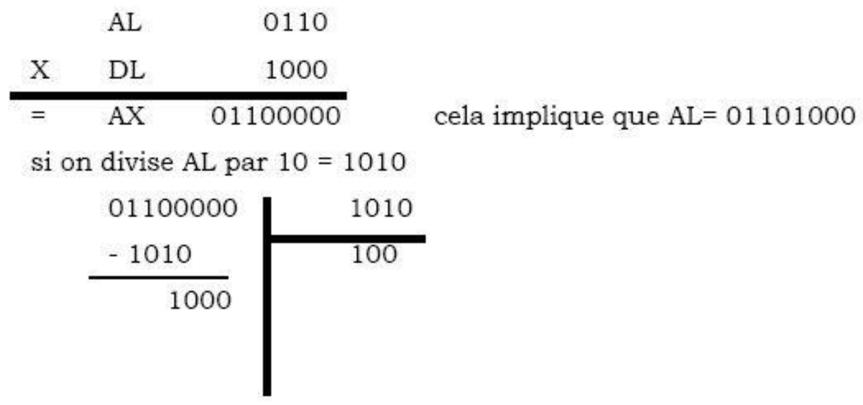
IMUL source

It has the same principle except that the numbers are signed.

3/ AAM Instruction (ASCII Adjust for Multiplication):

Like AAA and AAS, this instruction will enable us to correct the result of a multiplication of two BCD numbers. To correct the result of the AAM instruction, divide AL by 10.

Example :
MOV AL , 6
MOV DL , 8
MUL DL
AAM



```
     AL          0110
X    DL          1000
=    AX      01100000        cela implique que AL= 01101000
si on divise AL par 10 = 1010
     01100000  |    1010
     - 1010    |    100
         1000  |
```

D/ Division:

1/ DIV Instruction (division for not signed numbers):

DIV source

It performs an unsigned division of the accumulator by the source operand:

Examples:

- If the operand is a byte: then the quotient is recovered in register AL and the remainder in register AH.
- If the operand is a word: then the quotient is recovered in the AX register and the remainder in the DX register.

2/ <u>IDIV Instruction (division for signed numbers):</u>

IDIV source

It performs a signed division of the accumulator by the source operand:

- If the operand is a byte: then the quotient is recovered in register AL and the remainder in register AH.
- If the operand is a word: then the quotient is recovered in the AX register and the remainder in the DX register.

3/ <u>AAD Instruction (ASCII Adjust for Division):</u>

To correct the result, it multiplies the content of AH by 10 and adds it to that of AL.
Note: For this instruction, the adjustment must be performed before the division instruction.

4/ <u>CBW Instruction (Convert byte to word):</u>

This instruction doubles the size of the signed operand one byte becomes one word.

5/ <u>CWD Instruction (Convert word to double):</u>

This instruction doubles the size of the signed operand a word becomes doubled.

III/ <u>Logical (bit) instructions</u>:

Divided into three subgroups: logic, shift and rotation

A/ <u>Logical instructions</u>:

1/ <u>NOT Instruction (negation):</u>

NOT destination

$$\text{Destination} \longleftarrow \overline{\text{Destination}}$$

<u>Example</u> :
MOV AX, 500 ; AX = 0000 0101 0000 0000
NOT AX ; AX = 1111 1010 1111 1111

2/ <u>AND Instruction (logical and):</u>

AND destination, source

It is used to perform a logical AND between the destination and the source (byte or word); the result is put into the destination.

Destination = destination . source

Example :

MOV AX , 503H ; AX = 0000 0101 0000 0011
AND AX , 0201H ; 0000 0101 0000 0011 AND 0000 0010 0000 0001= 0000 0000 0000 0001
AND AX,BX ; AX = AX . BX (And logic between AX and BX)
AND AL,BH ; AL = AL . BH (logical AND on 8 bits)
AND AL,[SI] ; AL = AL AND the contents of the memory cell pointed to by SI
AND [DI],AL ; logical AND between the memory cell pointed to by DI and AL , the result is set in the memory cell pointed to by DI.

2/ OR Instruction (logical or):

OR destination, source

It is used to perform a logical OR between the destination and the source (byte or word). The result is put into the destination.

Destination = destination + source

Example :

MOV AX , 503H ; AX = 0000 0101 0000 0011
OR AX , 0201H ;0000 0101 0000 0011 OR 0000 0010 0000 0001 = 0000 0111 0000 0011
OR AX,BX ; AX = AX + BX (Logical OR between AX and BX)
OR AL,BH ; AL = AL + BH (8-bit logical OR)
OR AL,[SI ; AL = AL OR the contents of the memory cell pointed to by SI
OR [DI],AL ; logical OR between the memory locations DI and AL, the result is stored in the memory location DI

3/ XOR Instruction (exclusive or):

XOR destination, source

It is used to perform a logical exclusive OR between the destination and the source (byte or word). The result is put into the destination.

Examples :

MOV AX , 503H ; AX = 0000 0101 0000 0011
XOR AX , 0201H ; 0000 0101 0000 0011 XOR 0000 0010 0000 0001 = 0000 0011 0000 0010
XOR AX,BX ; AX = AX + BX (Exclusive OR between AX and BX)
XOR AL,BH ; AL = AL + BH (8-bit exclusive OR)
XOR AL,[SI] ; AL = AL OR exclusive the contents of the Memory box pointed to by SI
XOR [DI],AL ; Logical XOR between the memory cell pointed to by DI and AL, the result is set in the memory cell pointed to by DI.

4/ <u>Test Instruction:</u>

TEST destination, source

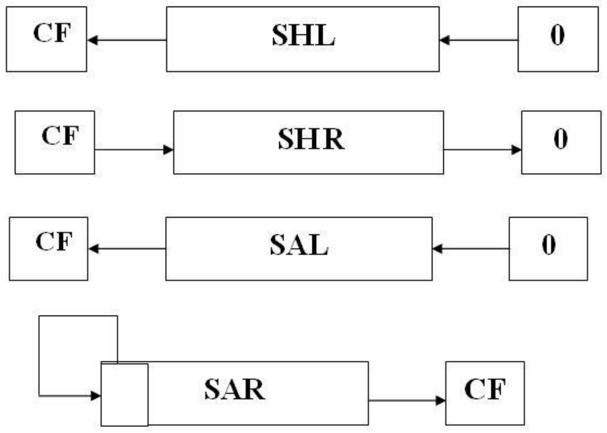
It is used to perform a logical AND between the destination and the source (byte or word), but the destination is not affected, as this instruction only affects flags.

<u>Examples</u> :

MOV AX, 503H ; AX = 0000 0101 0000 0011
TEST AX, 0201H ; 0000 0101 0000 0011

B/ Shifting instructions:

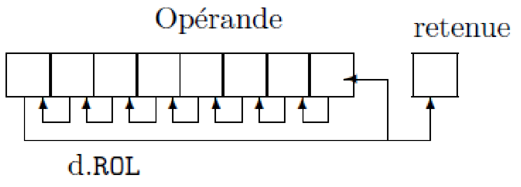
We have SHL (logical left shift), SHR (logical right shift), SAL (arithmetic left shift) and SAR (arithmetic right shift).



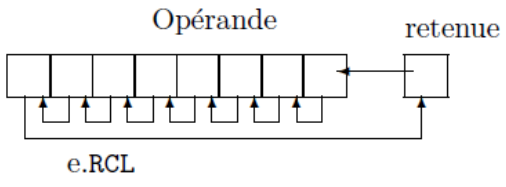
SHL destination, 1
SHL destination, CL
SHL destination, number of bits

C/ Rotation instructions:

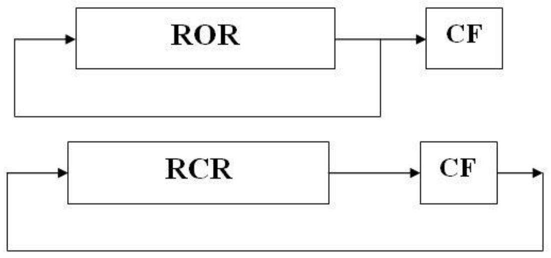
We have ROL (rotation to the left), RCL (rotation through the retainer to the left), ROR (rotation to the right) and RCR (rotation through the retainer to the right).



d.ROL



e.RCL



ROR destination, compteur

IV/ Program jumping instructions :

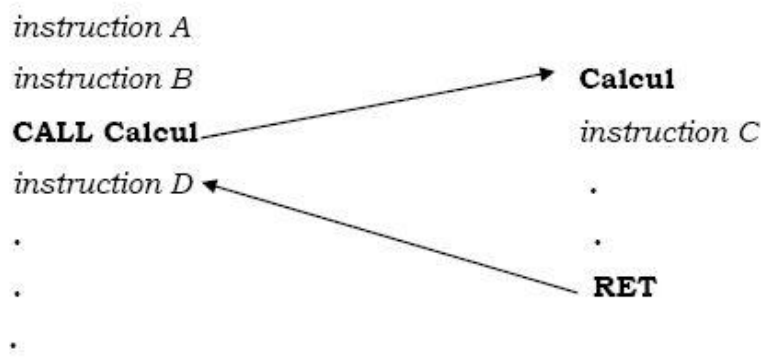
Usage	**Name**	**Function**
Unconditional jumps	CALL RET JMP	Call a sub-program or function Return of a sub-program or function Jump
Conditional jumps (not signed arithmetic)	JA/ JNBE JAE/ JNB JB/ JNAE JBE/ JNA	if greater/ if not less than or equal to if greater than or equal/ if not less than if less/ if not greatre than or equal to if less than or equal/ if not greater than
Conditional jumps (signed arithmetic)	JG/ JNLE JGE/ JNL JL/ JNGE JLE/ JNG	if greater/ if not less than or equal to if greater than or equal/ if not less than if less/ if not greatre than or equal to if less than or equal/ if not greater than
Conditional jumps (flags)	JC JE/ JZ JNC JNE/ JNZ JNO JNP/ JPO JNS JO JP/ JPE JS	If carry If equal/ if zero If no carry If not equal/ if not zero If no overflow if no parity/ if odd parity If not signed If overflow If parity/ if even parity If sgned (negatif)
Loops	LOOP LOOPE/ LOOPZ LOOPNE/ LOOPNZ JCXZ	Loop Loop if equal/ if zero Loop if different/ if different to zero jump if CX=0
Interruptions	INT INTO IRET	Interruption Interruption if overflow Return of interruption

A/ Unconditional jumping instructions:

1/ CALL instruction:

CALL procedure label

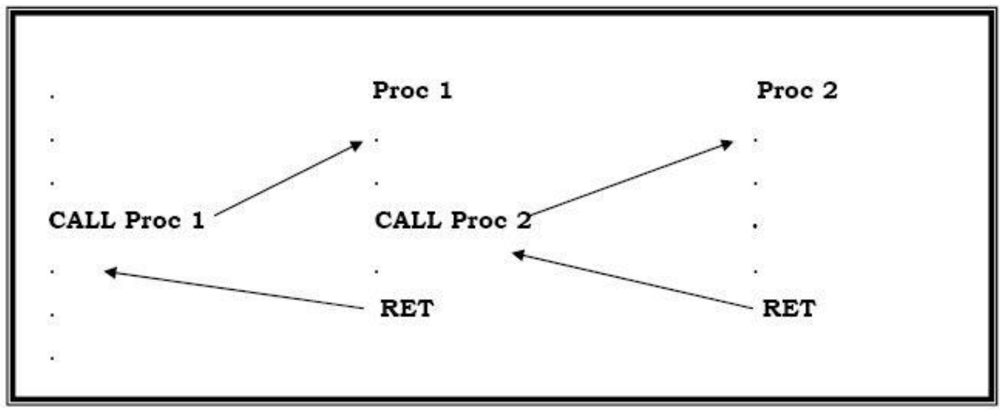
Jump to a procedure to execute it



to see in chapter 6

2/ RET instruction:

This instruction is used to return the value of the IP register to resume the instruction after a procedure has been executed.



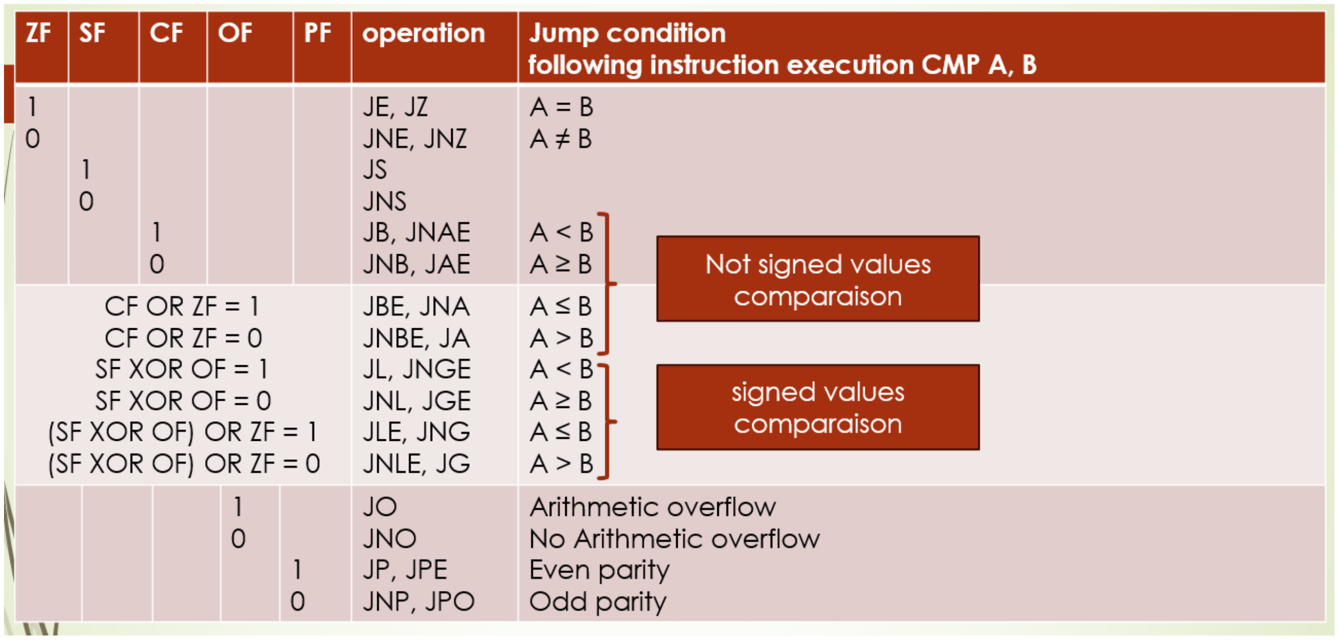
3/ JMP instruction:

JMP destination

- If the JMP is of type NEAR, then IP = IP + Displacement
- If JMP is FAR, then CS and IP are replaced by the new values obtained from the instruction.

JMP unconditionally transfers the command to the destination location. The Target operand can be obtained from the instruction itself (direct JMP), from memory or from a register indicated by the instruction.

B/ <u>Conditional jumping instructions</u>:



ZF	SF	CF	OF	PF	operation	Jump condition following instruction execution CMP A, B	
1					JE, JZ	A = B	
0					JNE, JNZ	A ≠ B	
	1				JS		
	0				JNS		
		1			JB, JNAE	A < B	Not signed values comparaison
		0			JNB, JAE	A ≥ B	
CF OR ZF = 1					JBE, JNA	A ≤ B	
CF OR ZF = 0					JNBE, JA	A > B	
SF XOR OF = 1					JL, JNGE	A < B	signed values comparaison
SF XOR OF = 0					JNL, JGE	A ≥ B	
(SF XOR OF) OR ZF = 1					JLE, JNG	A ≤ B	
(SF XOR OF) OR ZF = 0					JNLE, JG	A > B	
			1		JO	Arithmetic overflow	
			0		JNO	No Arithmetic overflow	
				1	JP, JPE	Even parity	
				0	JNP, JPO	Odd parity	

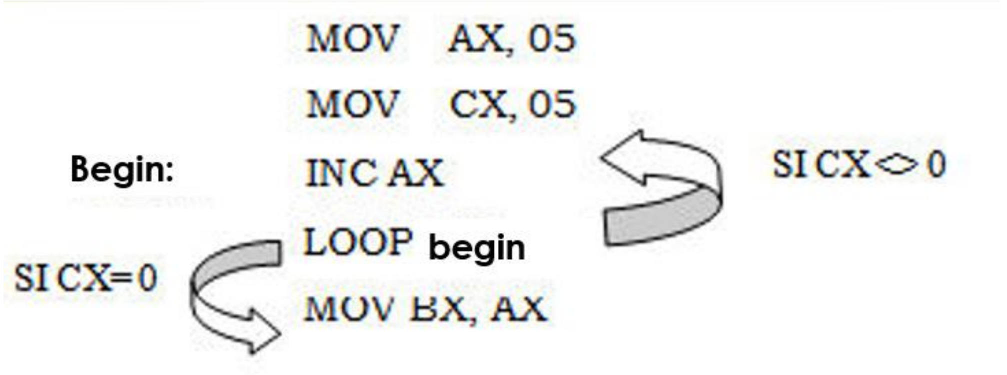
J?? destination

C/ <u>Loop instructions</u>:

1/ <u>LOOP instruction:</u>

LOOP label

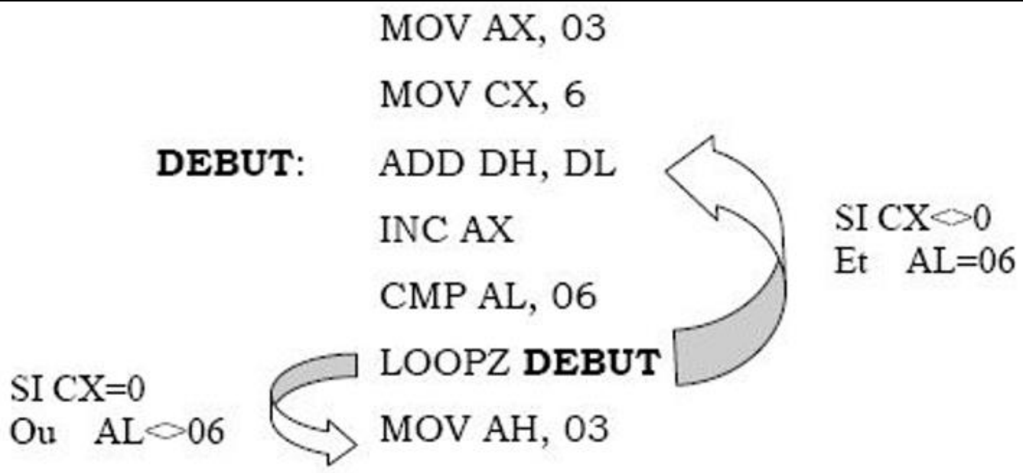
It decrements the content of CX by 1.
If CX is non-zero, then IP = IP + displacement
If CX = 0, the next instruction is executed.



```
        MOV  AX, 05
        MOV  CX, 05
Begin:  INC AX
        LOOP begin
        MOV BX, AX
```

2/ <u>LOOPE/LOOPZ instruction:</u>

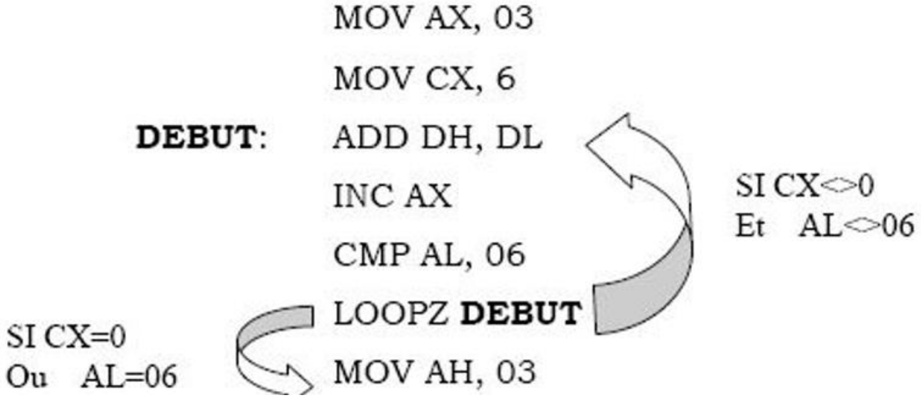
The CX register is automatically decremented by 1.
If CX is non-zero and ZF=1 then IP = IP + displacement



```
        MOV AX, 03
        MOV CX, 6
DEBUT:  ADD DH, DL
        INC AX
        CMP AL, 06
        LOOPZ DEBUT
        MOV AH, 03
```

3/ LOOPNE/LOOPNZ instruction:

The CX register is automatically decremented by 1.
If CX is non-zero and ZF=0 then IP = IP + displacement



```
        MOV AX, 03
        MOV CX, 6
DEBUT:  ADD DH, DL
        INC AX
        CMP AL, 06
        LOOPZ DEBUT
        MOV AH, 03
```

4/ Loop while:

```
WH: condition
        JCC EndWH
        Action
        …
        JMP WH
EndWH:
```

5/ Loop repeat:

```
Repeat:
        actions
        …
        Condition
        JCC repeat
```

V/ String instructions :

They can work on byte or word blocks of up to 64 Kbytes.

A/ Repeat prefixes:

1/ REP instruction:

These instructions are used with string instructions to ensure repetition of the instruction if it is to be applied to a set of information.

- REP automatically decrements CX to test whether it is equal to zero or not. If CX = 0 REP stops.
- REPE/REPZ: REPE/REPZ is the same as REP, i.e. it automatically decrements the CX register, but can exit the loop if ZF<>0.
- REPNE/REPNZ

2/ MOVE-string instruction:

MOVSB copies the byte (respectively the word for the MOVSW instruction) pointed to by SI to the byte (respectively the word) pointed to by **ES:DI**. The destination segment is necessarily ES. However, the source segment, which is DS by default, can be redefined.
Once copied, SI and DI are automatically modified. If the direction indicator is set to 0, 1 (respectively 2) is added to SI and DI. If the direction indicator is set to 1, 1 (respectively 2) is removed from SI and DI.
Example:

```
MOV AX, 8000H    ;Assign source segment address to AX
MOV DS, AX       ;Load source segment address into DS
MOV AX, 9000H   ;Assign destination segment address to AX
MOV CX, 0E0H    ;Move the length of the string to the counter register CX
MOV SI, 4000H    ;Assign source index address to SI
MOV DI, 5000H   ;Assign destination index address to DI
CLD     ;Ensure auto-increment mode is set by clearing the direction flag
REP MOVSB        ;Repeat the move byte from source to destination instruction CX times
```

B/ COMPARE-string instruction:

String comparison: subtracts the destination byte or word (pointed to by DI) from the source byte or word (pointed to by SI). CMPS affects the flags but does not change the operands.
If CMPS is used with the REPE/REPZ repeat prefix, it is interpreted as "compare as long as the string is not finished (CX <>0) and the elements to be compared are not equal (ZF=1).
If CMPS is used with the REPNE/REPNZ repeat prefix, it is interpreted as "compare as long as the string is not finished (CX <>0) and the elements to be compared are not equal (ZF=0)".
Note: You can't use the REP prefix with the CMPS instruction, as this would mean comparing only the last two elements of the two strings.

CMPSB destination, source
CMPSW destination, source

C/ SCAN-string instruction:

SCAS string
SCASB
SCASW

SCAS subtracts the destination string element (byte or word) addressed by DI in the extra segment from the contents of AL (a byte) or AX (a word) and acts on the flags. Neither the destination string nor the accumulator changes value.

D/ LOAD-string and STORE-string instructions:

1/ LOAD string instruction:

LODS source_string
LODSB
LODSW

LODS transfers the string element (byte or word) addressed by SI to the AL or AX register and updates SI to point to the next element in the string.

2/ STORE string instruction:

STOS destination_string
STOSB
STOSW

STOS transfers a byte or word from the AL or AX register to the string element addressed by DI, and modifies DI to point to the next location in the string.

Operation	destination	Source
MOVSW; MOVSB	[ES:DI]	[DS:SI]
CMPSW; CMPSB	[ES:DI]	[DS:SI]
SCAW	[ES:DI]	AX
SCAB	[ES:DI]	AL
LODW	AX	[DS:SI]
LODB	AL	[DS:SI]
STOW	[ES:DI]	AX
STOB	[ES:DI]	AL

VI/ Processor commands instructions :

- **STC:** Used to set carry flag CF to 1
- **CLC:** Used to clear/reset carry flag CF to 0
- **CMC:** Used to put complement at the state of carry flag CF.
- **STD:** Used to set the direction flag DF to 1
- **CLD:** Used to clear/reset the direction flag DF to 0
- **STI**: Used to set the interrupt enable flag to 1, i.e., enable INTR input.
- **CLI**: Used to clear the interrupt enable flag to 0, i.e., disable INTR input.

- **HLT:** Stop the CPU before the execution of the next instruction
- **Wait:** Suspend the CPU
- **Lock:** Bus lock instruction prefix

Procedure and interruptions in 8086 assembler language

I/ Assembly language subprograms:

In assembler, subroutines play the role of procedures and functions in high-level languages. They structure programs by giving a name to a process and reducing the size of programs that use the same sequence of code several times.

Calling a sub-program consists in breaking a sequence, continuing program execution and then resuming execution where the sequence was previously broken. This is exactly the same notion as in high-level languages, where calling a procedure suspends execution of the current sequence of instructions, executes the body of the procedure and then resumes execution of the previously suspended sequence. Note that a procedure can be called again within a procedure, or that a procedure can call itself.

In a high-level language, you don't have to worry about it working properly, as the whole mechanism is taken care of by the compiler, which generates the sequence of instructions in assembler so that everything works as expected. In assembler, you need to understand the mechanism in order to get the most out of it, and you also need to be able to pass parameters to the sub-programs.

Basically, two instructions are required:

- A subroutine call instruction, which breaks the sequence of instructions being executed and gives control to the subroutine to be called;
- A subroutine return instruction which, placed at the end of the subroutine, tells the processor to resume execution of the interrupted program where it left off, i.e. to execute the instructions following the subroutine call instruction.

A/ Actions to be taken when calling a sub-program:

We can briefly summarize the actions to be performed when calling and returning from a sub-program. Suppose we have a return stack. Calling a sub-program is done simply by stacking the current value of IP and storing the sub-program's call address in IP. Subroutine return is achieved simply by unstacking the top of the stack and placing this unstacked value in the IP register.

1/ The subroutine call stack:

The stack itself is a memory area (a kind of table) where the processor stores the data to be saved.

The only problem to be solved is where to find the top of the stack. It's hard to say that you can see the top of a computer stack. So we need an indicator that will show us this vertex. Since the elements of the stack are located in a memory area, and the elements are placed one after the other, all we need is a register that indicates the address of the top of the stack. The element below the top of the stack is then located in the neighboring memory location, ... This register is called the stack top pointer. This is the SP register (SP stands for stack pointer).

When data is stacked, this stack pointer must be updated to indicate the new stack top (as the stack rises). When data is unstacked, this pointer must also be updated (as the stack goes down). Given the importance of these operations, subroutine call and return instructions automatically modify the stack pointer so that subroutine calls and returns function correctly.

Classically, stacks are implemented with their base above their top, i.e. the stack top address is lower than the base address. So when data is stacked, the stack top pointer is decremented, while when data is unstacked, the stack top pointer is incremented.

2/ Back to subroutine call and return instructions:

Having described the implementation of the return stack, we can now fully describe the operation of subroutine call and return instructions.

Subroutine call instruction

When a sub-program is called, the following operations are automatically performed:

- SP is decremented
- (SP) ➝ return address (the contents of the IP register)
- IP ➝ operand of the subroutine call instruction

Sub-program return instruction

- IP ➝ (SP)
- SP is incremented

B/ Calling a subprogram without passing parameters:

Calls and subroutine returns:

In the calling program we have :

CALL procedure_name

Subroutine calls cause the contents of the IP register only to be saved on the stack. The value of the IP register is updated with the value of the operand supplied to the CALL instruction, so that program execution continues at the start of the sub-program. The value of the SP stack pointer register is, of course, updated after stacking the value of IP. The contents of the CS register remain unchanged.

And the procedure has the structure:

Name_procedure proc

…..

…..

Ret

Name_procedure endp

- The sub-program is declared by the proc pseudo-instruction, with its name indicated in the label field.
- This is followed by the subroutine instructions
- The RET instruction indicates the return to the caller.
- The pseudo-instruction endp indicates the end of the sub-program. The name of the sub-program is indicated in the label field

Context backup and restore:

- The context of the calling program is saved on the stack using push/pushf instructions (as seen in the previous chapter).
- Context restoration after the return of the sub-program is performed from the stack via pop/popf instructions (already seen).

C/ Passing parameters and local variables:

Passing parameters via register :

Example:

```
Caller

...
mov ax, 8
call facto
```

```
Sub-program

facto proc
        push cx
        mov cx, ax
        mov ax, 1
loop1:
        mul ax, cx
        loop loop1
        pop cx
        ret
facto endp
```

Passing parameters via stack :

For the called party, the BP register must be used to access parameter values. BP must therefore be initialized at the start of the called party's execution. Its value must also be saved. This must be done by the called party.

Suppose we pass three parameters, the values of registers AX, BX and CX

```
Caller

...
push ax
push bx
push cx
call sbr
add sp, 6
...
```

```
Sub-program

sbr proc
push bp
mov bp, sp
...
... ...[bp+4]...
... ...[bp+6]...
... ...[bp+8]...
...
...
pop bp
ret
sbr endp
```

- In the caller, parameter values are stacked before the subroutine is called.
- In the sub-program (i.e., the value of the ordinal counter IP has been stacked 2 bytes), the value of the BP register is stacked.
- The value of the BP register is initialized with the value of the top of the stack contained in SP
- Parameter values can now be accessed via the BP register. For example:
  - [bp+4] accesses the last stacked parameter (i.e. the value of register CX)
  - [bp+6] accesses the parameter stacked before CX (i.e. the value in register BX)
  - [bp+8] accesses the parameter stacked before BX (i.e. the AX register value)
- At the end of the called sub-program, the value of the BP register must be restored so that the caller has no problems. This is the role of the POP instruction.
- After execution of the POP, the top of the stack contains the return address, so a RET can be safely executed.
- Back in the caller, the memory area used to store parameters must be freed. To do this, we simply modify the value of the stack top pointer register SP by adding the value 6, which is the number of bytes occupied on the stack by the stacked values (2 for AX, 2 for BX and 2 for CX). We could also have freed the stack with POP instructions. However, it is customary to write programs as described here. Moreover, using POP instructions modifies the value of registers, which can have undesirable effects.

__Example__: calculating the PGCD of two numbers x and y

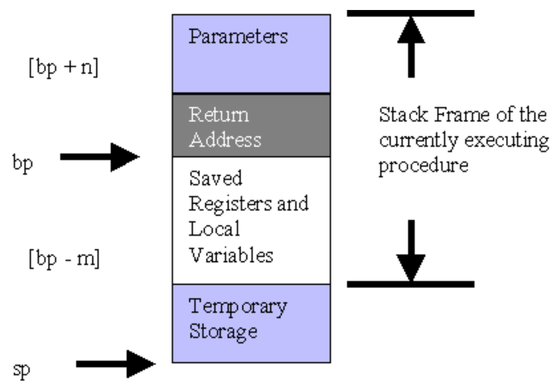
__Caller__

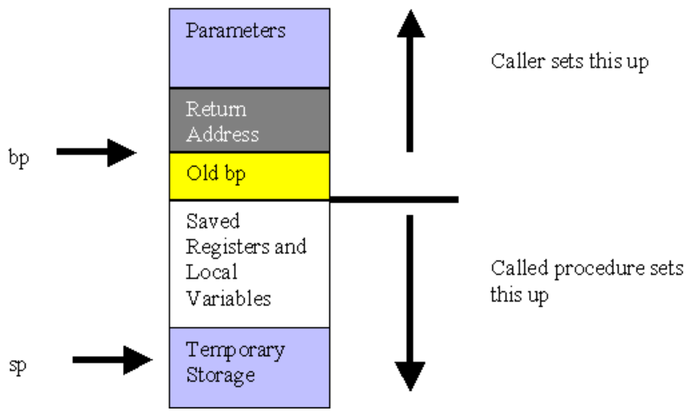
```
...
mov ax, 564
push ax          ; empile x
mov ax, 55
push ax          ; empile y
call pgcd
add sp, 4
...
```

__Sub-program__

```
pgcd proc
        push bp
        mov bp, sp
        push ax                 ; saving
        push bx                 ; ax and bx (contexte)
        mov ax, [bp+4]          ; load y in ax
        mov bx, [bp+6]          ; load x in bx
        tq1: cmp ax, bx
                jg sup          ; jump if y>x
                je ftq1         ; end if x=y
                sub ax, bx
                jmp tq1
                sup: sub bx, ax
                jmp tq1
        ftq1: call affiche_nombre
        pop bx
        pop ax
        pop bp
ret
```

__Local variables :__


[bp + n]
bp
[bp - m]
sp
Parameters
Return Address
Saved Registers and Local Variables
Temporary Storage
Stack Frame of the currently executing procedure



Caller

```
...
push ax
push bx
call sbr
add sp, 6
...
```

Sub-program

```
sbr proc
push bp
mov bp, sp
push di            ; DI and SI are local variables
push si
sub sp, 10
...
... ...[bp+4]...
... ...[bp+6]...
...
... ...[bp-6]...
...
add sp, 10
pop si
pop di
pop bp
ret
sbr endp
```

- Save the BP register (as before)
- initialize the BP register so that it points to the top of the stack
- Save the value of registers DI and SI. Note that BP points to the value of the saved DI register.
- Reservation of 10 bytes on the stack (using the sub sp, 10 instruction), which can be freely manipulated in the subprogram and whose role is to contain the value of local variables.
- Access to local variables will then be via addressing such as [bp-6] ... [bp-15], depending on the size of the data. Importantly, the programmer must define the use of these 10 bytes.
- At the end of the call, the stack area used to store the value of local variables is freed (ADD instruction).
- We then restore the input value of the SI, DI and BP registers.

II/ DOS interruptions:

A/ Input/output management:

1/ Definition:

Input/Output refers to any information transfer operation between the processor, memory and external devices.

- From outside ➝ Input (Read)
- External ➝ Output (Write)

2/ <u>I/O types:</u>

- Physical I/O: managed by the lowest level of the system (hardware).
- Logical I/O: an instruction within a program that calls an I/O procedure (e.g. Scanf, Printf).
- Virtual I/O: between processor, main memory and virtual memory.
- Spool I/O: linked to a slow peripheral, access is via a disk file called "Spool".

3/ <u>Driver (manager):</u>

The driver is a specific program which initiates a physical transfer, controls the operation and handles errors.

Example: the printer driver allows you to :

- Ensure that the printer is plugged in.
- Ensure that it is not busy.
- Make sure there's paper.
- Insert the first sheet.
- Fill the pad with the characters to be printed.
- Start printing and move on to the next set of characters.

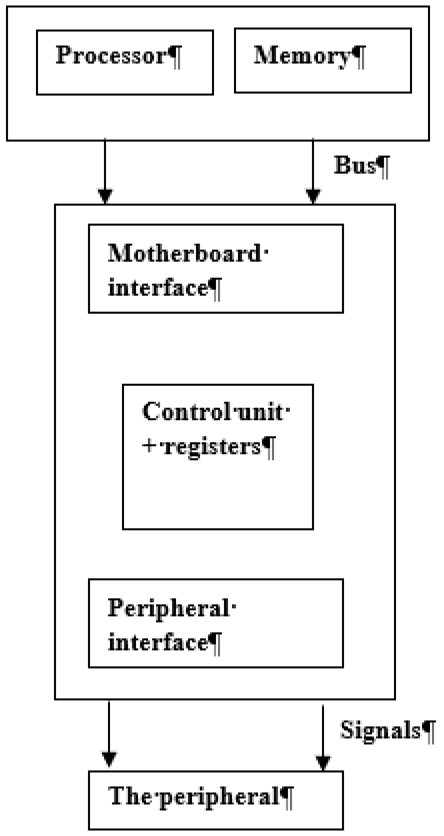
4/ <u>I/O interfaces:</u>

The interface is a device that connects peripherals to the computer, i.e. it adapts the characteristics of each of the two parts, motherboard and peripheral, these characteristics are :

- Type of connection: serial or parallel.
- Transmission mode: synchronous or asynchronous.
- Transmission speed: fast or slow.

5/ <u>Peripheral controllers:</u>

Used to drive peripherals, each peripheral is linked to a bus by a controller. The role of the controller is to adapt the diversity of peripherals to a common interface. It's an electronic card inserted into the motherboard.



6/ Physical input/output control mode:

Processor control of a peripheral unit requires synchronization between processor actions and the unit being controlled. Several I/O control techniques are used:

a/ Direct physical I/O:

This is to control the flow of physical I/O by the processor itself, where we have :

- *Synchronous mode:*

The processor controls the peripheral from I/O start to finish. To start an I/O, the processor first initializes the appropriate controller registers with the necessary information, then issues a command to the controller to initiate the I/O operation. The controller, in turn, examines the contents of these registers to determine the action to be taken and initiates the transfer.

During the physical transfer of the character, the processor remains mobilized to monitor the I/O operation by consulting a device-specific flag designating its state, when it detects the end of the transfer. (Also known as active waiting).

The major drawback is the loss of CPU time.

- *Interval scan mode:*

In this mode, the processor initiates the I/O operation and then returns to the execution of user processes. At regular short intervals, the processor is interrupted to check whether this operation has been completed.

Same disadvantage of time loss due to handling frequent clock interruptions.

- *Asynchronous mode:*

The processor is freed from transfer completion control, where the controller is provided with an interrupt line that will be used to inform the processor. Once the transfer has been completed, the controller generates an interrupt that informs the processor of the end of the transfer.

The disadvantage of this mode is the time lost in executing these interrupt routines. **The interrupts already seen in Chapter III**

b/ Indirect physical I/O:

This technique is designed to improve processor performance by freeing the processor from tracking I/O until it has finished. An independent device known as a channel is used.

- *Channel mode :*

A channel is a programmable processor that can execute a sequence of I/O operations. It executes a channel program resident in MC.

The processor initiates an I/O operation by sending a special instruction to the channel, telling it where to find the channel program that starts the I/O. The channel then executes all the commands of this program independently of the processor. At the end of execution, the channel sends an interrupt to the processor to warn it of the end of I/O.

- *Direct memory access (DMA):*

DMA is a simplified channel used in small computers, which can be connected between the controller and the memory bus, enabling peripheral devices to access memory without passing through the CPU. If the DMA and CPU simultaneously request memory access, the DMA takes priority.

B/ DOS interruptions:

1/ On-screen character display:

Display a character :

```
mov dl, ’a’
mov ah, 2
int 21h
displays character 'a' on screen.
```

Display a message :

```
.data
msg db ’hello world’, 13, 10, ’$’
.code
mov ax, @data
mov ds, ax
mov ah, 9
lea dx, msg
int 21h
mov ah, 4ch
int 21h
end
```

This pseudo-code displays the message hello world on the screen. The character string must end with a display the message hello world on the screen. The string must end with a char-return.

2/ Enter a keystroke:

With echo:

```
mov ah, 1
int 21h
returns the code of the character read from the keyboard to the al register.
```

```
.data
msg db 80 dup (?)
.code
mov ax, @data
mov ds, ax
mov bx, 0
mov cx, 80
lea dx, msg
mov ah, 3fh
int 21h
```

this pseudo-code captures a keyboard string of up to cx characters and stores it in the variable msg, which is declared as an array of 80 bytes (i.e. 80 characters). The number of characters read is placed in the ax register. Note that the char-return is stored in the form of two successive characters: the first in code 13, the second in code 10.

Without echo:

```
mov ah, 8
int 21h
```

reads a character from the keyboard and returns it to register al. This character is not displayed on the screen.

3/ File access:

In assembler, files are processed in a similar way to an advanced language such as Pascal. To access data in a file, either for reading or writing, it must first be opened. On opening, the program retrieves a handle. All operations on this file are then performed by specifying this handle. At any given moment, several files may have been opened by a program. Each one has a different handle, enabling the program to distinguish between them.

File creation:

Before the call
- cx contains the value 0 ;
- ds:dx point to the file name. This is a string terminated by an ASCII character code 0.

After the call
The C bit of the flags register is set to 1 if an error has occurred.

The call has the following format:

```
mov ah, 3ch
int 21h
```

File opening:

Before the call
- al contains the access mode:
0 for read, 1 for write,
2 for both;
- ds:dx point to the filename.
file name. This is a string of
terminated by an
ASCII character code 0.

After the call
The C bit of the flags register is set to 1 if an error has occurred. Otherwise, the ax register contains the handle on the open file.

The call has the following format:
mov ah, 3dh
int 21h

File closing:

Before the call
- bx contains the handle of the file to be closed

After the call
The C bit of the flags register is set to 1 if an error has occurred. Otherwise, the file is closed. It can no longer be accessed via its handle. If you wish to access it again it must first be re-opened.

The call has the following format :
mov ah, 3eh
int 21h

File reading:

Before the call
- bx contains the file handle ;
- cx contains the number of characters to be read ;
- ds:dx contains the address of the buffer to store the characters read.

After the call
The C bit of the flags register is set to 1 if an error has occurred. Otherwise, the ax register contains the number of characters read. In principle, this is the number of characters that should have been read, unless the end of the is reached.

The call has the following format :
mov ah, 3fh
int 21h

Read the current time :

```
mov ah, 2ch
int 21h
When the call returns, the registers contain the following information:
ch hours
cl minutes
dh seconds
dl hundredths of a second
```

Read the current date :

```
mov ah, 2ah
int 21h
When the call returns, the registers contain the following information:
al coded day of the week (0: Sunday, 1: Monday, ...)
cx year
dh month
dl date
```