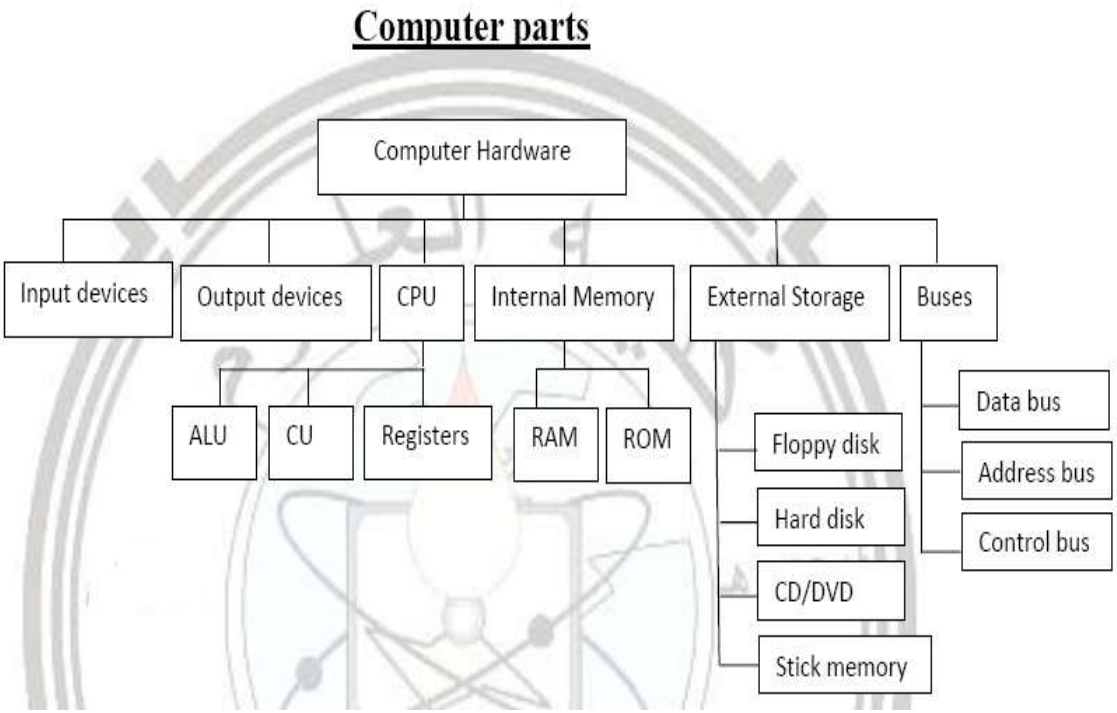
محاضره(1)


Computer parts
Computer Hardware
Input devices
Output devices
CPU
Internal Memory
External Storage
Buses
ALU
CU
Registers
RAM
ROM
Floppy disk
Hard disk
CD/DVD
Stick memory
Data bus
Address bus
Control bus

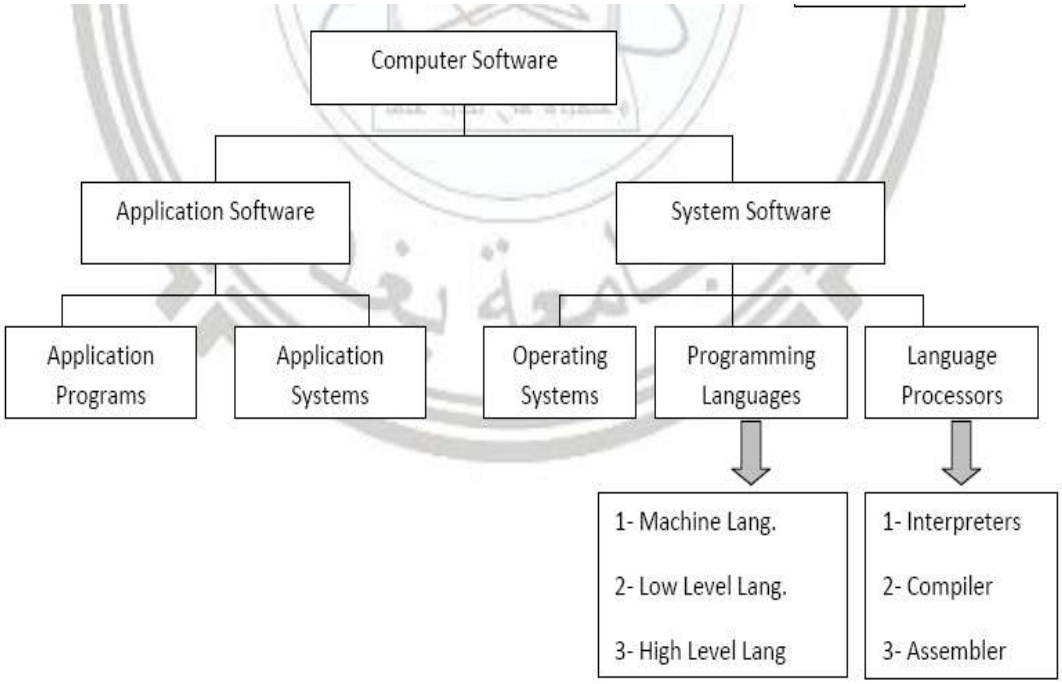


Figure1 (Computer parts)

How a program execute in a computer:

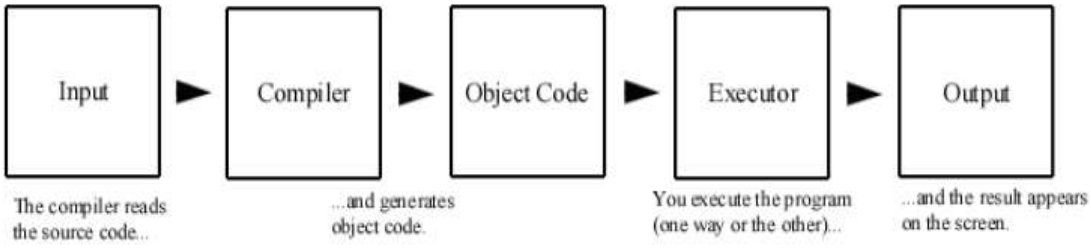


Figure2 (program cycle)

A Program in computer consists of two; first is the *source code*, other is the *object code*.

Source code is the version of a computer program as it is originally written i.e. typed into a computer by a programmer in a programming language.

A compiler is a specialized program that converts source code into object code and it converts the whole program at a time.

The object code is a machine code, also called a machine language, which can be understood directly by a specific type of CPU. So the Object code is the code which is executed by the compiler and is then sent for execution.

A machine code file can be immediately executable i.e. runnable as a programmer it might require linking with other object code files e.g. libraries to produce a complete executable program.
Thus object code is simply the machine language output of a compiler that is ready for execution on a particular computer and an object file format is a format that is used for the storage of object code and related data produced by a compiler

An object code file can contain not only the object code, but also relocation information that the linker uses to assemble multiple object files to form an executable program. It can also contain other information, such as program symbols and debugging information.

Introduction to computer system:

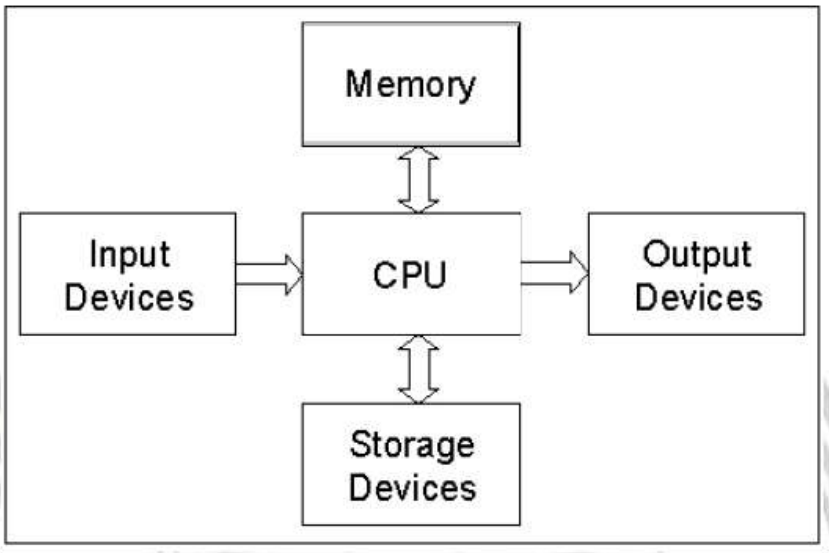


Figure3 (computer system Hardware)

Micro Computers (Micros): are small computer systems driven by a *microprocessor* chip, they designed to be used by one person at a time, commonly called personal computers (PCs).
Microprocessor: (physically) is a digital integrated circuit (IC) that can be programmed with a series of instructions to perform various operations on data. A microprocessor is the CPU of a computer. It can do arithmetic and logic operations, move data from one place to another, and make decisions based on certain instructions

Microprocessor elements (parts):

A MP (logically) consists of several units, each designed for a specific task. The design and organization of these units are called *architecture*. See figure4.

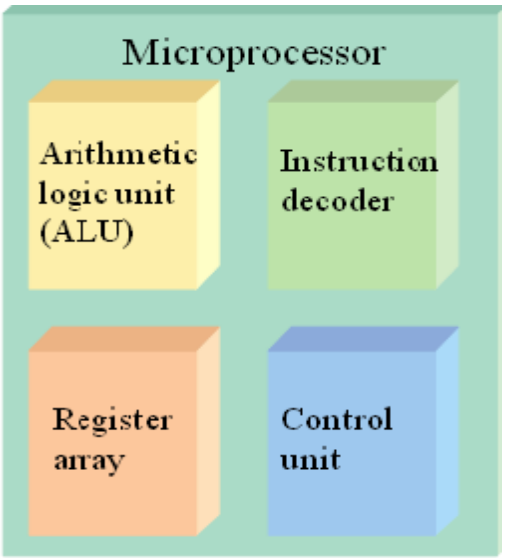


Figure4 (Microprocessor elements)

1- ALU: (Arithmetic Logic Unit) which perform arithmetic operations like addition, subtraction, multiplication and division) and logic operations like (NOT, AND, OR, X-OR), as well as many other types of operations. ALU obtains data from the registers.

2- Instruction Decoder: it translates the programming instruction into an address where microcode resides for executing the instruction.

3- Register set (array): It is a collection of registers that are contained within the microprocessor. Data and memory addresses are temporarily stored in these registers during the execution of a program. The registers work very quickly making the program run more efficiently.

 GPR: General-Purpose Register, meaning they can be used for multiple purposes and assigned to a variety of functions by the programmer.

 SPR: they have specific capacities and functions; they can't be used as GPR by the data user.

 Flags: they used to hold processor status. These bits (flags) are set by the CPU as the result of the execution of an operation. The status bits can be tested at a later time as part of another operation.

4- Control Unit: (CU) it provides the timing and control signals for getting data into and out of the MP.

محاضره(2)

Microprocessor Buses:

Three buses for microprocessors allow *data*, *addresses* and *instructions* to be moved. See figure5

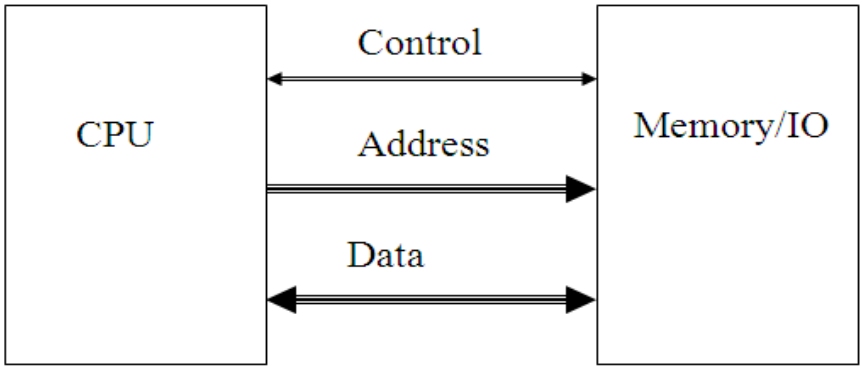


Figure5 (Microprocessor Buses)

1- The data bus: It is a *bi-directional* bus that connects the CPU, memory, and the other hardware devices on the motherboard. The number of wires in the bus affects the speed at which data can travel between hardware components. For example an 8-wire bus can move 8 bits at a time, which is a full byte. A 16-bit bus transfer 2 bytes, and a 32-bit bus can transfer 4 byte at a time because each wire can transfer 1 bit of data at a time.

2- The address bus: It is a *one-directional* bus that connects only the CPU and RAM and carries only *memory address*. The address bus used by the MP to specify the number of locations (words) in memory. For example if the address bus of a MP is 16 lines then it has 216 unique locations in memory (0 216-1).

3- The control bus: It is a *bi-directional* bus used by MP to coordinates its operations and carrying *control signal* (either write signal or read signal). Also control bus can carrying timing signals.

Microprocessor Basic Operations (functions):

A microprocessor executes a program by repeatedly cycling through following three basic steps: See figure6.

1- Fetch an instruction from main memory and place it in the CPU
2- Decodes the instruction; if other information is required by the instruction, fetch the other information.
3- Execute the instruction and store the results

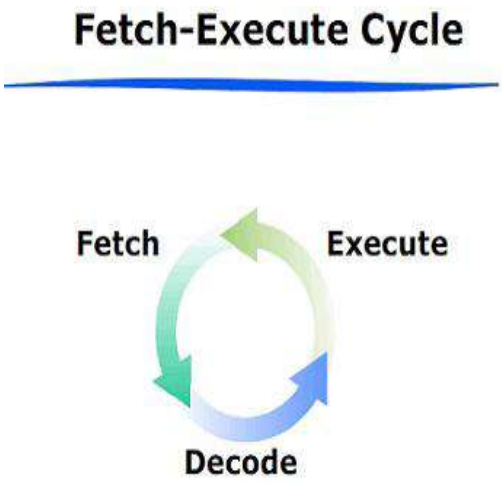


Figure6 (MP Fetch-Execute Cycle)

NOTE:

The Intel MPs (8080.8086, 8088, 80286, 80386, 80486 until Pentium) were all ***single core processors***, meaning they had only one microprocessor in an IC chip, they run multiple programs (*Multi-tasking*) to increase the processing speed.

Multi-Tasking is a technique that capable of running many tasks (programs) at the same time. By dividing the time of MP between all of the tasks, the MP switches from one task to another so quickly that it gives the appearance of executing all of them simultaneously.

NOTE:

Newer processors have more than one core on a single IC each one with its own cache memory. These ***multi-core processors*** operate in *parallel* and can run programs much faster than a single core chip; this process is also called *multiprocessing.*

Or the multi-core processors system can work on multi-tasking technique (but only one processor is involved at a time).

Brief History of Intel Family:

Intel 8080/8085 VS Intel 8086/8088

In 1979, Intel cooperation introduced a 16-bit microprocessor called 8086. This processor was a major improvement over the previous generation 8080/8085 series Intel microprocessor in several ways:

- ***First:*** 8086 capacity of 1 Megabyte of memory exceeded the 8080/8085 capacity of handling a maximum of 64K bytes of memory.

- ***Second:*** the 8080/8085 was an 8-bit system, meaning that the MP could work on only 8 bits of data at a time. Data larger than 8 bits had to be broken into 8-bit pieces to be processed by the CPU. In contrast, the 8086 is a 16-bit microprocessor.

- ***Third:*** the 8086 was a *pipelined* processor, as opposed to the *non-pipelined* 8080/8085.

Pipelining: A technique where the microprocessor begins executing the next instruction in a program before the previous instruction has been completed. That is several instruction are in the pipeline *simultaneously*, each at different processing stage.

A pipeline is divided into stages; each stage can execute its operation *concurrently* with the other stages. When the stages complete an operation, it passes the result to the next stage and fetches the next operation from the preceding stage in the pipeline. The final results of each instruction emerge at the end of the pipeline in rapid succession. Pipeline in 8086/88 has 2 stages only fetch and execute, but in more powerful MP have many stage. See figure6

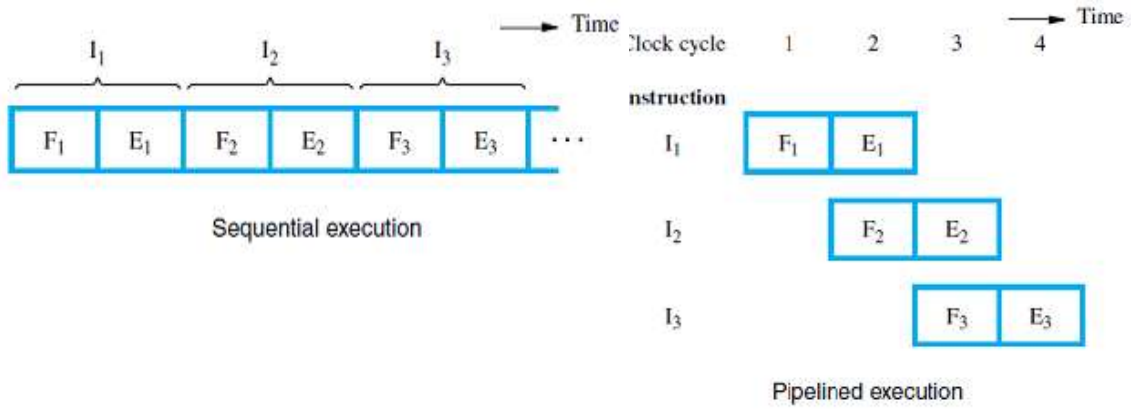


Figure6 (The idea of pipeline) execution)

There are two ways to make the CPU process information faster:

1.One way to improve performance is to exchange the IC chip by faster circuit technology to build the processor and the main memory, this way is technology dependent, i.e. depends on technology is available at the time, with consideration of cost.

2. Another way is to arrange the hardware to improve the internal work of CPU,i.e. more than one operation can be performed at the same time. In this way, the number of operations performed per second is increased and increase the over all performance of processor. (In 8086 pipe lining technique appears).

Data type (bits)	Physical memory	Address bus	External data bus	Internal data bus	year	product
8	64 K byte	16	8	8	1974	8080
8	64 K byte	16	8	8	1976	8085
8,16	1 M byte	20	16	16	1978	8086
8,16	1 M byte	20	8	16	1979	8088
8,16	16 M byte	24	16	16	1982	80286
8,16,32	4 G byte	32	32	32	1985	80386
8,16,32	4 G byte	32	32	32	1989	80486
8,16,32	4 G byte	32	64	32	1993	Pentium

Table1 Main Features of the Intel X86 Microprocessor Families

Introduction to 8086/8088 MP:

Intel implement pipelined concept in 8086/8088 by splitting internal architecture of the MP into two sections: the ***Execution Unit*** (***EU***) which *executes* the instructions, and the ***Bus Interface Unit*** (***BIU***) which *fetches* instructions, *reads* operands and *writes* results. The two sections work simultaneously.

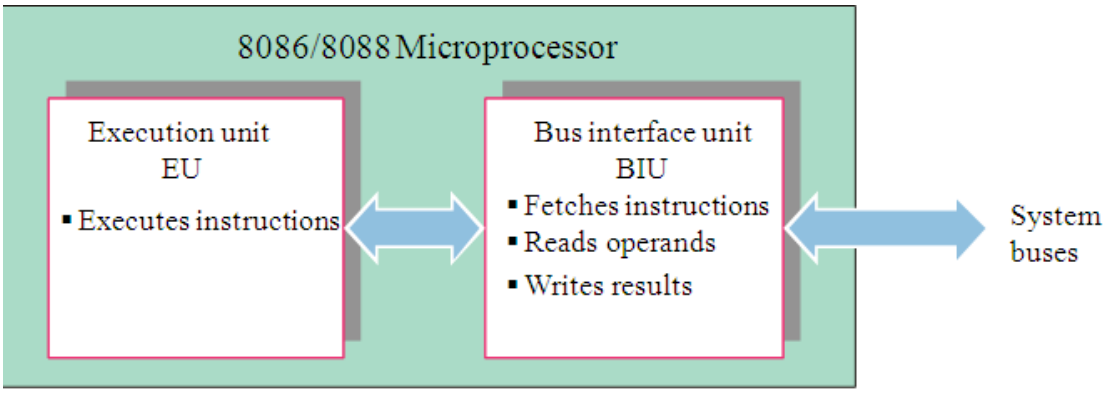


Figure7 (BIU & EU of 8086/8088 MP)

Inside to 8086/8088 MP:

The 8086/8088 processors are fully software compatible with each other; the 8088 had *20 address bits* that could address *1 MB (220)* of memory locations. 8088 used an *8-bit external data bus*, and *16-bit internal data bus (the same with the registers width)*. The 8088 had *4-byte instruction queue*.

The 8086 was identical to 8088 MP except that it had an *external 16-bit data bus* like its *internal 16-bit data bus* and *6-byte instruction queue*. The 8086 had also 20 address bits (220 Byte of memory).

محاضره(3)

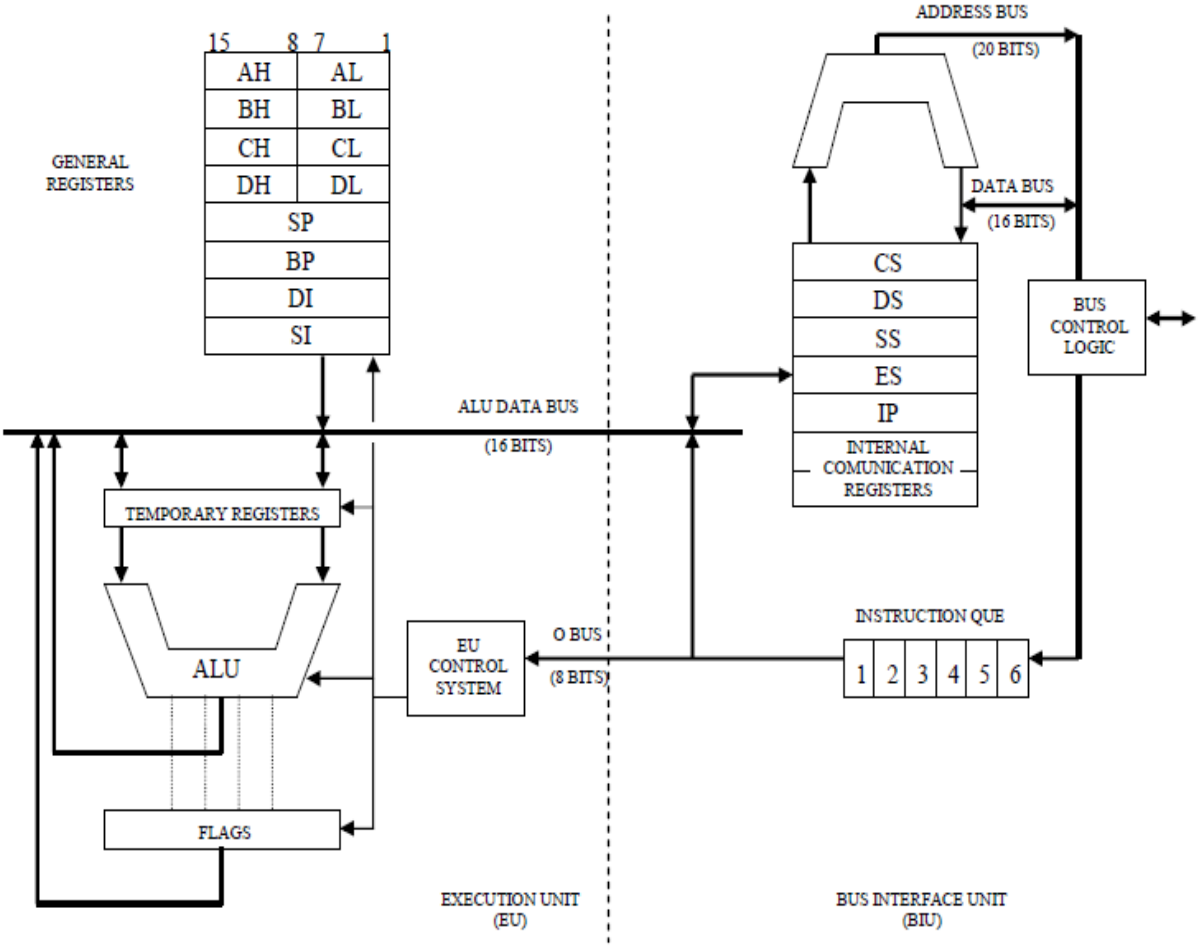


Figure8 (The internal organization of the of 8086MP)

Bus Interface Unit (BIU) in 8086/8088 MP:

BIU has the following functions: 1- Instruction fetch. 2- Instruction queuing.

3- Operand fetch and storage. 4- Address relocatable. 5- Bus control. The major parts of BIU are:

Instruction Queue: it increases the average speed with which a program is executed by storing up to *4 bytes in 8088* (*6 bytes in 8086*) this allows the next instructions or data to be fetched from memory while the processor is executing the current instruction at one time. If any instruction takes too long to execute, the queue is filled to its maximum capacity, and the buses will sit idle. The BIU fetches a new instruction whenever a bus has room for *2-bytes*.

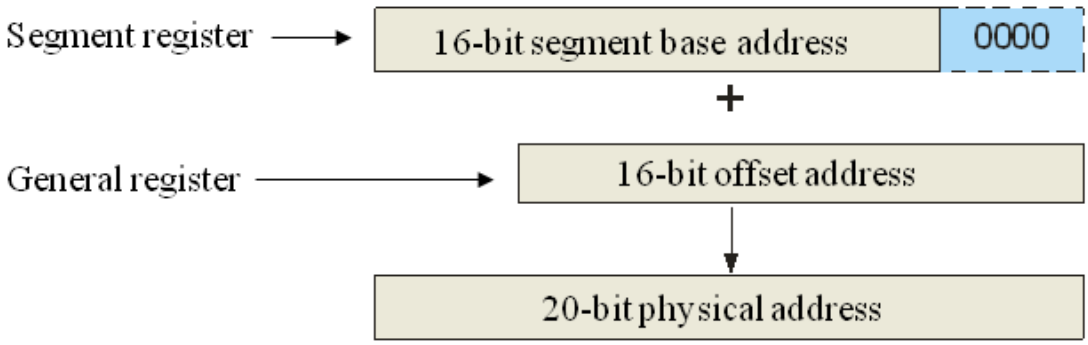
Segment Registers: The 8086/88 processors had *four segment registers* (CS, DS, SS, and ES) all of them 16-bit registers used in process of forming 20-bit address. A ***segment*** is 64 KB (216) block of memory and can begin at any point in the 1 MB of memory space. The 4 segment registers can be changed by the program to point to other 64 KB blocks. ***Code segment*** (***CS***)***:*** is a 16-bit register containing address of 64 KB segment with processor instructions. The processor uses CS segment for all accesses to instructions referenced by instruction pointer (IP) register. CS register cannot be changed directly. The CS register is automatically updated during far jump, far call and far return instructions.

Stack segment (SS): is a 16-bit register containing address of 64KB segment with program stack. By default, the processor assumes that all data referenced by the stack pointer (SP) and base pointer (BP) registers is located in the stack segment.

Data segment (DS): is a 16-bit register containing address of 64KB segment with program data. By default, the processor assumes that all data referenced by general registers (AX, BX, CX, and DX) and index register (SI, DI) is located in the data segment.

Extra segment (ES): is a 16-bit register containing address of 64KB segment, usually with program data. By default, the processor assumes that the DI register references the ES segment in string manipulation instructions.

Adder (Σ): perform *address re-locatable* operation it generates the 20-bit physical address (PA) by using two 16-bit register. One of the registers that formed the PA was always *a* ***segment register***; the other one was a 16-bit general register containing address information (***offset***).



4. ***Instruction pointer (IP):*** The IP contains the Offset Address of the nextinstruction, which is the distance in bytes from the base address given by thecurrent Code Segment (CS) register. The contents of the CS are shifted left by four. Bit 15 moves to the Bit 19 position. The lowest four bits are filled with zeros. The resulting value is added to the Instruction Pointer contents to make up a 20-bit physical address. The CS makes up a segment base address and the IP is looked as an offset into this segment.

Example*:* Assume IP = (20A0)16 and CS= (B200)16.

a)What is the location of the start and end of the block?

b)What physical address is formed?

Solution:

a)The start of the block is at B200016; it ends at

B200016 + FFFF16 = C1FFF16

b)The physical address is B200016 + 20A016 = B40A016

Execution Units (EU) in 8086/8088 MP:

EU has the following functions: 1- Decodes the instruction fetched by the BIU.

2- Generates appropriates control signal. 3- Executes the instructions.

The major parts of EU are:

1. Arithmetic Logic Unit (ALU): this unit does all the arithmetic and logic operations, working with either 8-bit or 16-bit operands.

2.The General Registers: All general registers of the 8086 microprocessor can be used for arithmetic and logic operations. The set of 16-bit general registers is divided into two sets of four registers; one set contains of ***data registers***, and the other set consists of the ***pointer and index registers***. See figure 9.

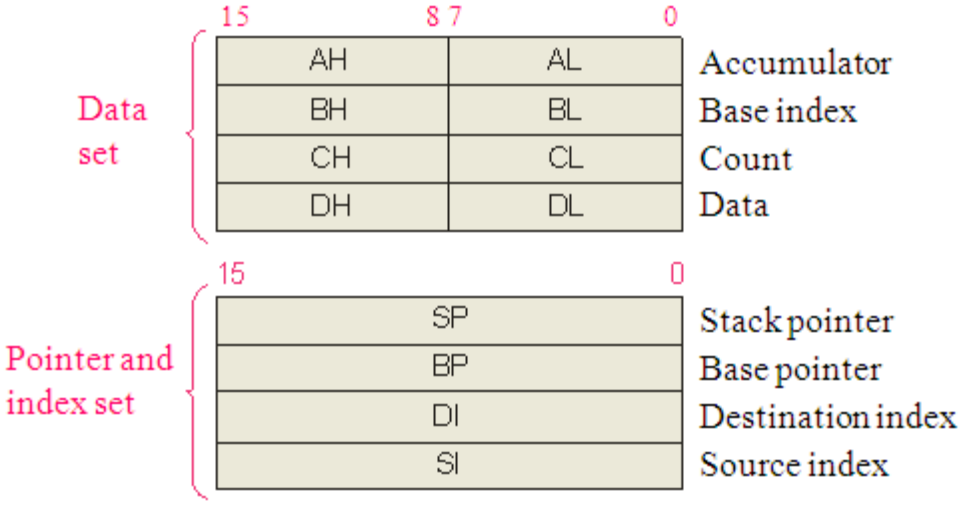


Figure9 (The 8086/8088 general registers)

2.1 Data Register Sets:

- ***Accumulator register:*** consists of two 8-bit registers **AL** and **AH**, which can be combined together and used as a 16-bit register **AX**. AL in this case contains the low-order byte of the word, and AH contains the high-order byte. Accumulator can be used for I/O operations and string manipulation.

- ***Base register:*** consists of two 8-bit registers **BL** and **BH**, which can be combined together and used as a 16-bit register **BX**. BL in this case contains the

- ***Count register:*** consists of two 8-bit registers **CL** and **CH**, which can be combined together and used as a 16-bit register **CX**. When combined, CL register contains the low-order byte of the word, and CH contains the high-order byte. Count register can be used as a counter in string manipulation and shift/rotate instructions.

- ***Data register:*** consists of two 8-bit registers **DL** and **DH**, which can be combined together and used as a 16-bit register **DX**. When combined, DL register contains the low-order byte of the word, and DH contains the high-order byte. Data register can be used as a port number in I/O operations. In integer 32-bit multiply and divide instruction the DX register contains high-order word of the initial or resulting number.

2.2 Pointer and Index Register Sets:

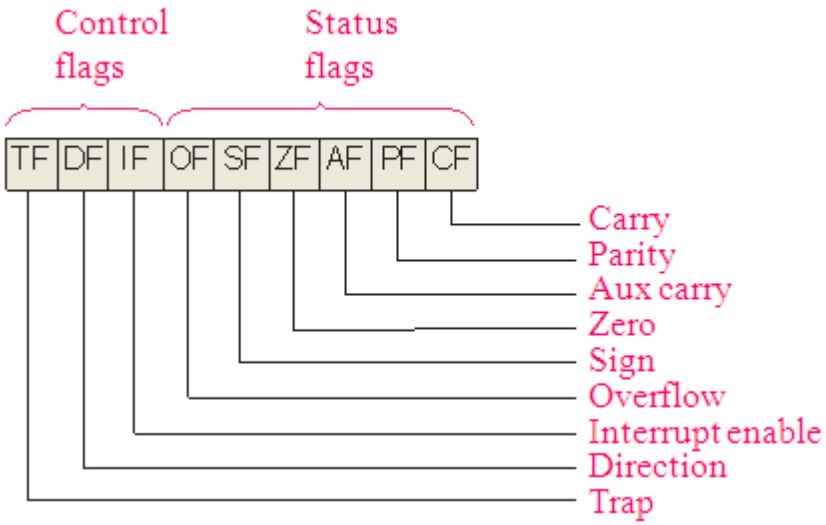
- ***Stack Pointer (SP):*** is a 16-bit register pointing to program stack.

- ***Base Pointer (BP)*** is a 16-bit register pointing to data in stack segment. BP register is usually used for *based*, *based indexed* or *register indirect addressing.*

- ***Source Index (SI):*** is a 16-bit register. SI is used for *indexed*, *based indexed* and *register indirect addressing*, as well as a source data address in string manipulation instructions.

- ***Destination Index (DI):*** is a 16-bit register. DI is used for *indexed*, *based indexed* and *register indirect addressing*, as well as a destination data address in string manipulation instructions.

3. *The Flags:* the flag register is a 16-bit register 9 bits from 16 bits are only used as control bits (flags). A status flag is a one – bit indicator used to reflect a certain condition after an arithmetic or logic operation by the ALU.



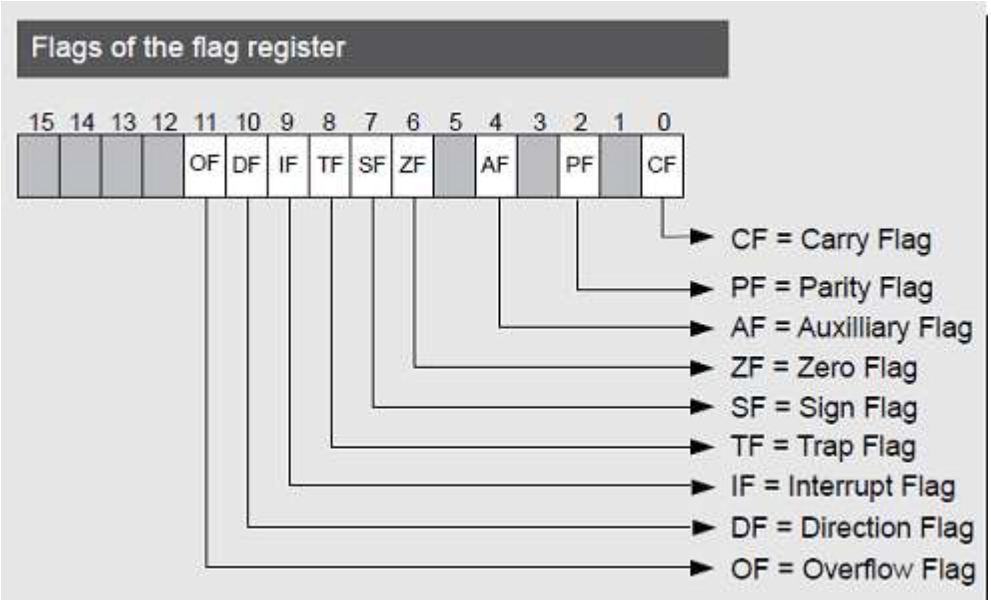


Figure 10 (The 8086/8088 16- bit Flag register)

- ***Overflow Flag (OF):*** set if the result is too large positive number, or is too small negative number to fit into destination operand.
- ***Direction Flag (DF):*** if set then string manipulation instructions will auto-decrement index register. If cleared then the index registers will be auto-incremented.
- ***Interrupt-enable Flag (IF):*** setting this bit enables maskable interrupts.
- ***Single-step Flag (TF):*** if set then single-step interrupt will occur after the next instruction.
- ***Sign Flag (SF):*** set if the most significant bit of the result is set.

- ***Zero Flag (ZF):*** set if the result is zero.
- ***Auxiliary carry Flag (AF):*** set if there was a carry from or borrow to bits 0-3 in the AL register.
- ***Parity Flag (PF):*** set if parity (the number of "1" bits) in the low-order byte of the result is even.
- ***Carry Flag (CF):*** set if there was a carry from or borrow to the most significant bit during last result calculation.

Segment Addressing:

A ***segment*** is an area of memory that includes up to 64K bytes and begins on an address evenly divisible by 16. The segment size of 64K bytes came about because the 8085 microprocessor could address a maximum of 64k Bytes of physical memory since it had only 16 pins for the address lines (216=64K). Whereas in the 8085 there was only 64K bytes of memory for all *code*, *data* and *stack* information. For this reason, the 8086/88 microprocessor can only handle a maximum of 64K bytes of code and 64K bytes of data and 64K bytes of stack at any given time. The 8086 processor has 20 bits for address, so the total memory size is 220=1 MB. This results in a range of addresses from 00000 – FFFFF

All segment registers in 8086 (CS, DS, SS and ES) that contain the memory address are 16-bit registers, then how can a 20 bits of address bus during each memory fetch operation be accommodated? The four 16-bit segment registers in 8086 are used to point to the beginning of a segment of memory that is 64 KB (216) long. . Any location within one of these 4 segments of total 1 MB memory is addressed *relative* (in the positive direction, or locations in higher memory-negative direction). See figure11.

The 8086 MP treats the contents of the segment register as ***shifting the binary contents of the segment register left by four places to obtain the physical address that puts on address bus to fetch its contents (data)***.

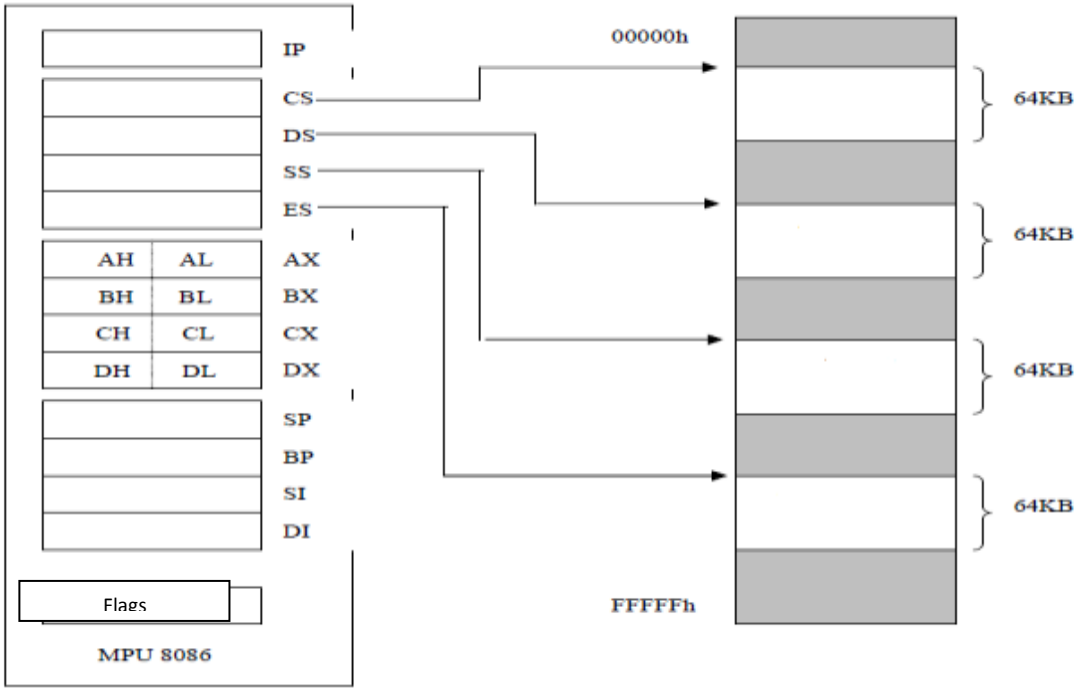


Figure11 (The programming model of 8086 MP)

محاضره(4)

Logical address and Physical address:

The BIU contains a dedicated ***Adder*** which is used to generate the 20 bits physical address that is out put on the address bus. This address is formed by adding the 16-bit shifting segment address with the offset address.

Physical address (PA) = Shifted Segment register + Offset

Segment address: It is located with one of the segment register and defines the beginning address of any 64 KB memory segment.

Offset address: is a location within a 64 KB segment range, therefore, an offset address can range from 0000 – FFFF.

Logical address: consists of a segment value and offset address. Sometimes the segment and offset is written as (**segment: offset**).

 The different combination used in 8086 MP for Segment: offset addressshown in table below:-

Purpose	Offset	Segment	Logical address
Instruction address	IP	CS	CS:IP
Stack address	SP or BP	SS	SS:SP or SS:BP
Data address	BX, DI or SI	DS	DS:BX, DS:DI or DS:SI
String destination address	DI	ES	ES:DI

Example Assume IP = $20A0_{16}$ and CS=$B200_{16}$.

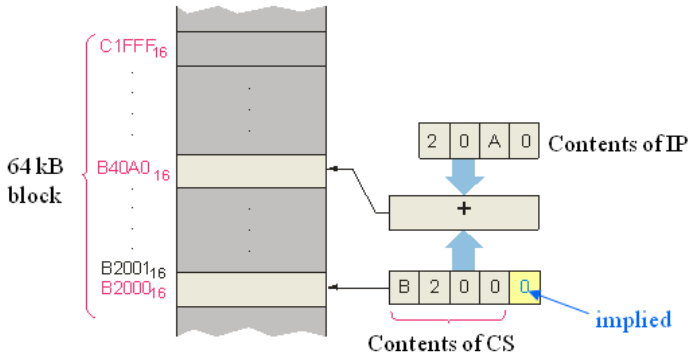
a) What is the location of the start and end of the block?

b) What physical address is formed?

Solution The addressing is diagramed:

a) The start of the block is at $B2000_{16}$;
it ends at $B2000_{16} + FFFF_{16} = C1FFF_{16}$

b) The physical address is
$B2000_{16} + 20A0_{16} = B40A0_{16}$



Example1:
If CS=24F6h and IP=634Ah, show:

a- The Logical address and the offset address. And calculate:
b- The physical address.
c- The lower range and upper range of the code segment.

Sol:
a- 24F6:634A, the offset is 634A

b- PA= 24F60+634A PA=2B2AA

c- The lower range 24F60+0000=24F60
The upper range of the code segment is 24F60+FFFF=34F5F

Example2:

If DS = 7FA2h and the offset is 438Eh, show:

a- The logical address, and calculate :

b- The physical address. c- The lower range and upper range of the code segment.

Sol:

a- 7FA2:438E

b- PA= 7FA20+438E=83DAE

c- The lower range 7FA20+0000=7FA20

The upper range of the code segment is 7FA20+FFFF=8FA1F

H.W1: If the logical address of SS: SP is 3500: FFFE, Calculate the PA and the upper range of the stack segment?

H.W2: Assume that the DS register is 578C. To access a given byte of data at physical memory location 67F66, does the data segment cover the range where the data is located?

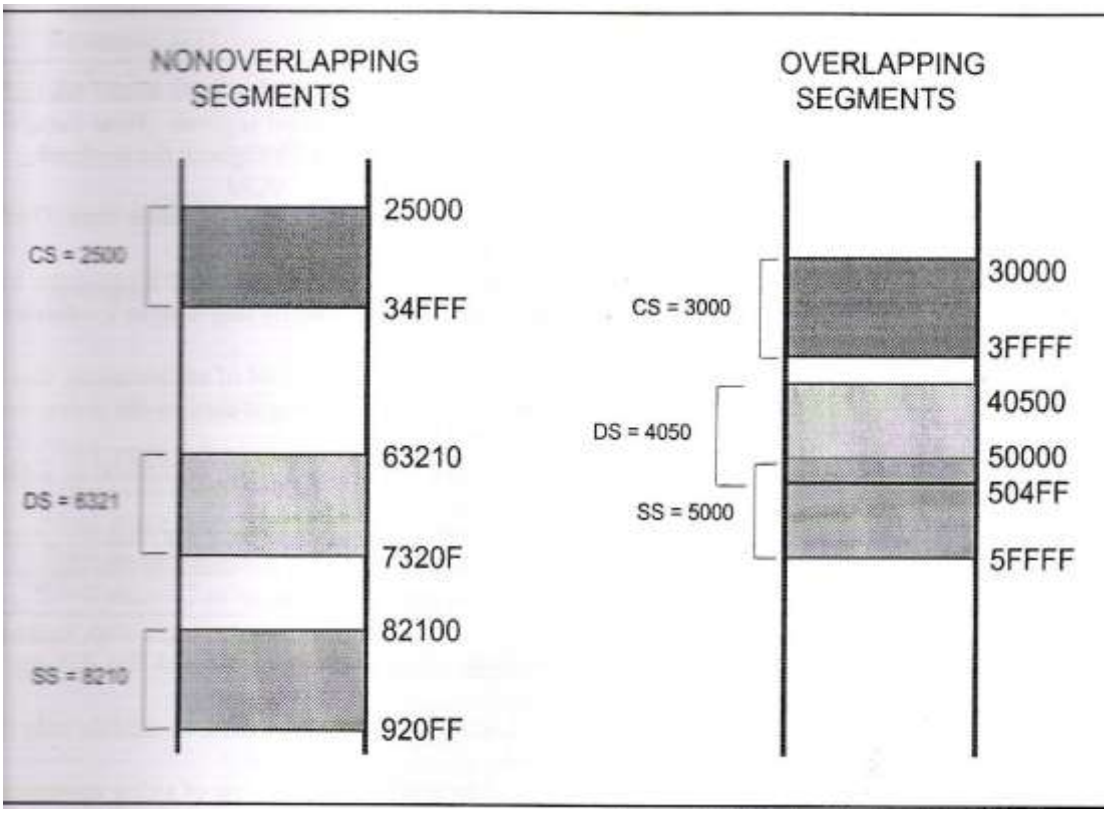
H.W3: Assume memory locations with the following contents:

DS:6826h=48 and DS:6827h=22. Show the contents of register BX after the execution of this instruction MOV BX, [6826h]

H.W4: What is the range of physical addresses if CS=FF49h? Draw the segment throw the total 1MB memory

.Segment Overlapping:

In calculation the physical address, it is possible that two segments can overlap. See figure below



Example3:

If you know the CS=800h and DS= 500h, if the CS and DS are overlapped or not?

Sol:

CS range from 8000h to (8000h+FFFFh) =17FFFF

DS rand from 5000h to (5000h+FFFFh) = 14FFFh, The two segments are overlapped

محاضره(5)

Machine language and Assembly language:

Machine language: is the native language of a given processor. Every type of CPU understands its own machine language. Instructions in machine language are *numbers* stored as bytes in memory (consists of 0's and 1's). Each instruction has its own unique numeric code called its *operation code* or *opcode.* Machine language is very fast processed by CPU, but it is very difficult to program in directly. De-coding the meanings of the numerical-coded instructions is tedious for humans.

Assembly language: is the symbolic form of machine language. An assembly language program is stored as text (just as a higher level language program). Each assembly instruction represents exactly *one machine* instruction. It is written with short abbreviations called *mnemonics*. A mnemonic is an abbreviation that represents the actual machine instruction.

Assembly language programming: is the writing of machine instructions in *mnemonic* form, where each machine instruction (binary or hex value) is replaced by a mnemonic. The use of mnemonics is more meaningful than that of hex or binary values, which would make programming at this low level easier and more manageable.

Assembler: is a program that reads a text file with assembly instruction and converts the assembly into machine code. Every assembly language statement directly represents a single machine instruction. High-level language statements are much more complex and may require many machine instructions. **Assembly language Instruction Format:**

An assembly program consists of a sequence of assembly statements. Each line of an assembly program is split into the following four fields: ***label***, ***operation code*** (***opcode***), ***operand***, and ***comments***. See figure12

Label (Optional)	Operation Code (Required)	Operand (Required in some instructions)	Comment (Optional)

figure12**Assembly instruction format)**

Labels are used to provide symbolic names for memory addresses. A label is an identifier that can be used on a program line in order to *branch to* the labeled line. It can also be used to access data using symbolic names.

Operation code (***opcode***) field contains the symbolic abbreviation of a given operation.

Operand field consists of additional information or data that the opcode requires.

Operands can have the following types:

1.Register: These operands refer directly to the contents of the CPU's registers.

2.Memory: These refer to data in memory. The address of the data may be aconstant into the instruction or may be computed using values of registers.Address are always offsets from the beginning of a segment.

3.Immediate: These are fixed values that are listed in the instruction itself. Theyare stored in the instruction itself (in the code segment), not in the data segment.

4.Implied: There operands are not explicitly shown. For example, the incrementinstruction adds one to a register or memory.

INC CX ➔CX=CX+1 (The one is implied)

MUL BL➔AX=AL×BL (AL register is implied)

Comments field provides a space for documentation to explain what has been done for the purpose of debugging and maintenance.

Types of Instructions:

To simplify learning the Intel 8086 MP instruction set, instructions are divided into seven categories these groups are:

1.Data Transfer: the basic data transfer instruction is MOV this instruction can be used in several ways to copy a byte, a word (16 bit) or a double word between various *sources* and *destinations* such as registers, memory and I/O port.

NOTE:

 In 8086 MP, the data can be moved among all registers***except*** the ***Flag register.***

 Values can't be loaded directly into any segment register (CS, DS, SS or ES).First we must load it to a non-segment register and then move it to the segment register. First MOV AX,2345h then MOV DS,AX

 The CS and IP registers can't be used as a destination operand, because they store the address of the next instruction.

 If a value less than FFh is moved into a 16-bit register, the rest of bits are assumed to be all zeros. MOV BX, 05h --- BX=0005h.

 Moving a value that is too large into a register will cause an error. For example the following instructions are illegal MOV BL, 7F2h and MOV AX,2FE456h.

2.Arithmetic: ADD, SUB, MUL and DIV instructions. Increment or decrement INC, DEC instruction is also included. These instructions allow for carry operation and for signed & unsigned arithmetic. The operand located in register or memory location or I/O port.

3.Bit Manipulation: This group of instructions includes *three class of operations*: -- Logical (Boolean) instructions: NOT, AND, OR and XOR.

- Shifts instruction: SAR, SAL …

- Rotate instruction: ROL, ROR. All of these 3 sub-groups of instructions operate on bytes or words in registers or memory location.

4.Loops and Jump: These instructions are designed to alter the normal (one after the other) sequence of instructions. Most of these instructions test the processor's flags to determine which instruction should be processed next. For example JMP,JNZ, JA …. Loop (decrement the CX register and repeats if not zero).

5.Strings: A ***string*** is a ***contigous*** (one after the other) sequence of bytes, for example MOVSB (copy one byte at a time), MOVSW (copy word).

6.Subroutine and Interrupts: A ***subroutine*** is a ***mini program*** that can be used repeatedly but programmed once. For example CALL (begin the subroutine) and RET(return to the main program).

7.Processor Control: this is a small group of instructions that allow direct control of some of the processor's Flags and other miscellaneous tasks. For example STC (set carry flag CF).

Each Type of those instructions will be discussed in details in Laboratory lessons

Instructions types According to the number of operands contain:

Instructions can be classified based on the number of operands as: *three-address*, *two-address*, *one address* and *zero-address*.

For the instruction layout: **Operation *Destination***, S***ource1,*** S***ource2 …***

- the **operation field** represents the operation to be performed, for example, ADD, SUB, MOV, etc …
- The **source field** represents the source operand(s), which can be an immediate date, a value stored in a register, or a value stored in the memory.
- The **destination field** represents the place where the result of the operation is tobe stored which can be a register or a memory location.

1. Three – address instruction: **Operation *Destination***, S***ource1,*** S***ource2***

The 3-address instruction means that the three operands in the instruction refer to a memory locations. For example ADD A, B, C. Adds the contents of a memory location C to the contents of memory location B and stores the result in memory location A.

2. Two– address instruction: **Operation *Destination***, S***ource***

The 2-address instruction means that the two operands in the instruction refer to a memory locations. For example ADD A, B. Adds the contents of memory location B to the contents of memory location A and stores the result in memory location A.

3. One– address instruction: **Operation** S***ource***

In this case the instruction *implicitly* refers to a register, called the *Accumulator* .The only operand appeared in the instruction refers to a memory location. For example ADD B. Adds the contents of memory location B to the contents of the Accumulator register and stores the result back into the accumulator.

4. Zero– address instruction: **Operation *Operand***

The operand in the instruction not refers to a memory location, may be a registercontents or immediate data. For example

INC BL →BL=BL+1.
MUL 05h→multiply the contents of AL with 05h, the result will be in the AX.

8086 Addressing Modes

When the 8086 MP execute an instruction, it performs the specified function on data. These data are called ***operand*** and may be part of the instruction, reside in one of the internal registers of 8086 or stored at an address in memory. To access these different types of operand, the 8086 provide with various addressing mode.

An ***addressing mode*** can be defined as the way, in which the operand is specified in an instruction, i.e. (Addressing mode is a method of specifying an operand), and can be categorized into Three main types:

1.Register Addressing Mode:

The register addressing mode involves the use of registers to hold the data to be manipulated. Memory is not accessed when this addressing mode is executed; therefore, it is relatively fast.

Examples:

MOV BX, DX Copy the contents of DX into BX.

MOV ES, AX Copy the contents of AX into ES.

ADD Al, BL AL=AL+BL

2.Immediate Addressing Mode:

In this mode, the Source operand is a constant and part of the instruction. Immediate addressing mode can be used to load information into any of the registers **except** the segment registers and flag register. Also this type of addressing executed quickly because when the instruction is assembled the operand comes immediately after code.

Examples:

MOV BX, 2550h → Move 2550h into BX, where BL=50 and BH=25.

MOV CX, 625 → Load the decimal value 625 into CX. CX=0271h

ADD BL, 40h → BL=BL+40h

3.Memory Addressing mode:

To reference an operand in memory. The 8086 MP must calculate the physicaladdress of the operand and then read or write this storage location.

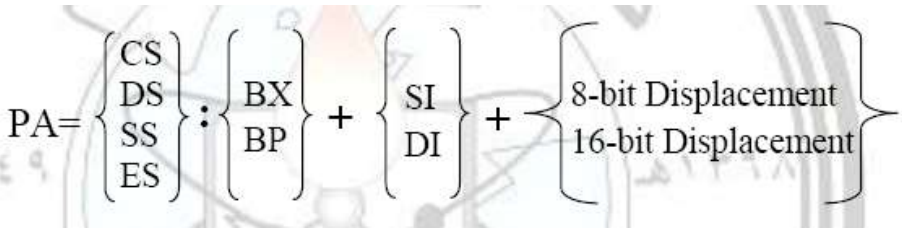
The physical address (PA) = Segment Address (SA) + Offset Address (EA)

- Segment address (SA), which is usually stored at one of the segment register
- Offset address or Effective address (EA): is the address of the offset that can be made up from as many as three elements: ***Base***, ***Index***, and ***Displacement***.

Logical Address SA: EA

Physical Address PA= SA+ EA

EA=Base + Index + Displacement



Not all effective address elements are always used in the effective address calculation. In fact a number of memory addressing modes are defined by using various combinations are:

3.1- Direct addressing mode:

In direct addressing mode the data is in some memory location(s) and the address of the data in memory comes immediately after the instruction. See figure 13

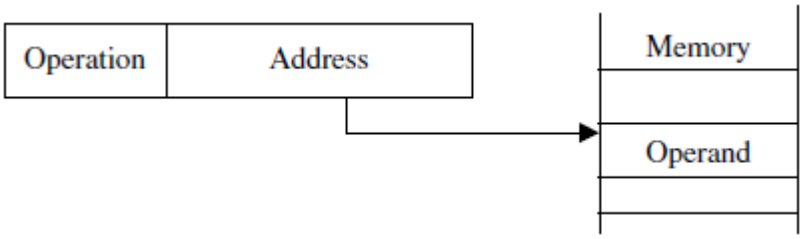
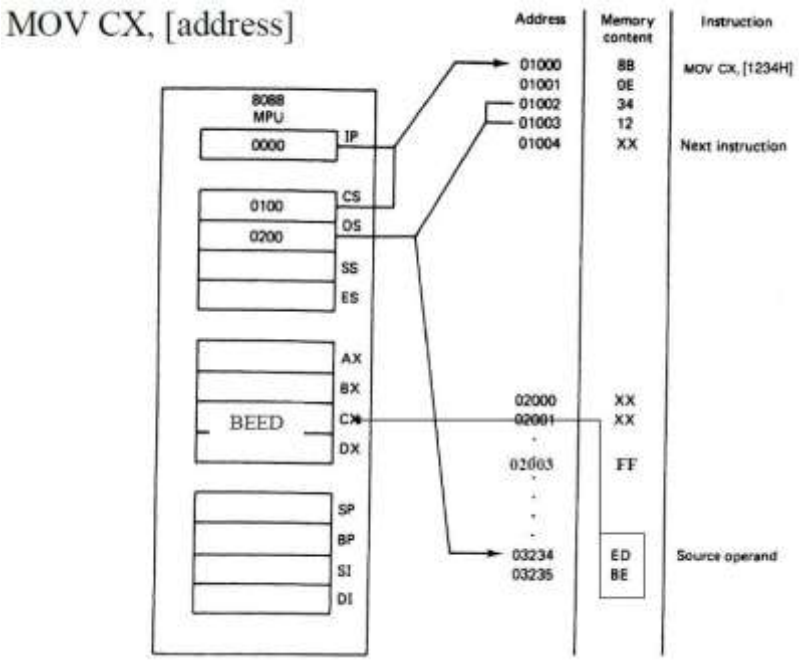


Figure13 (Direct addressing mode)

Example1:

MOV DX, [2400] → DL= [DS0+2400], DH= [DS0+2400+1]

MOV BL, [235F] → BL= [DS0+235F]



Example2:
MOV CX, [Beta]
IF you know Beta=1234h, DS=0200h, CS=0100h, IP= =0000h, the contents of memory location 3234 = EDh and the contents of next location = BEh. Calculates:

a)PA of the operand and the PA of this instruction.

b)Write the machine code for this instruction, (MOV CX = 0E8Bh)

c)PA of the next instruction.

d)How many bytes this instruction takes?

Sol:

a)PA of the operand = DS0 +Beta □ 2000+1234=3234h

PA of the instruction = CS0+IP □ =1000+0000=1000h

b)Machine code:- 01000 8B
01001 0E
01002 34
01003 12

c)so , the PA to the next instruction will be □ 01004h

d)4 bytes

H.W1: Find the PA of the memory location and its contents after the execution of the following assuming that DS=1512h.
MOV AL, 99h,
MOV [3518], AL

H.W2: MOV DX, [453Fh]
IF you know DS=800h, CS=200h, IP= =100h, Calculates:

a)PA of the operand and the PA of this instruction.

b)Write the machine code for this instruction, (opcode = 168Bh)

c)PA of the next instruction.

d)How many bytes this instruction takes?

3.2- Register indirect addressing mode:

In the register indirect addressing mode, the address of the memory location where the operand resides is held by a register. The registers used for this purpose are SI, DI, and BX. See figure14

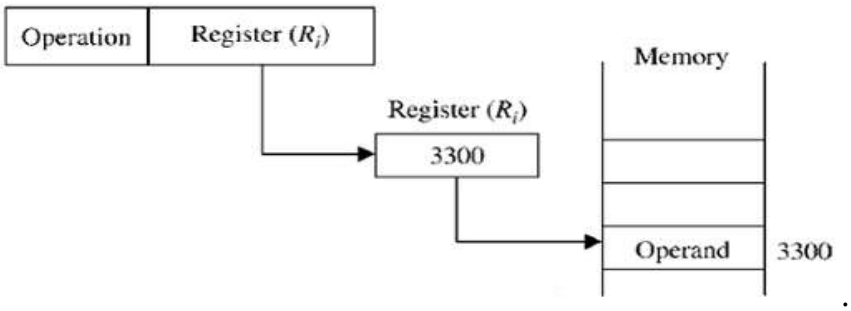


Figure14 (Register indirect addressing mode)

Example1:
MOV CL, [SI] □ CL= [DS0+SI]
MOV [DI], AX □ [DS0+DI] =AL, and [DS0+DI+1] =AH.
Example2:
MOV AX, [SI] , If IP=0000h, CS=300h, SI=1456h, DS=220h, Calculates:

a)PA of the operand and the PA of this instruction.

b)Write the machine code for this instruction, (MOV AX = 048Bh)

c)PA of the next instruction.

d)How many bytes this instruction takes?

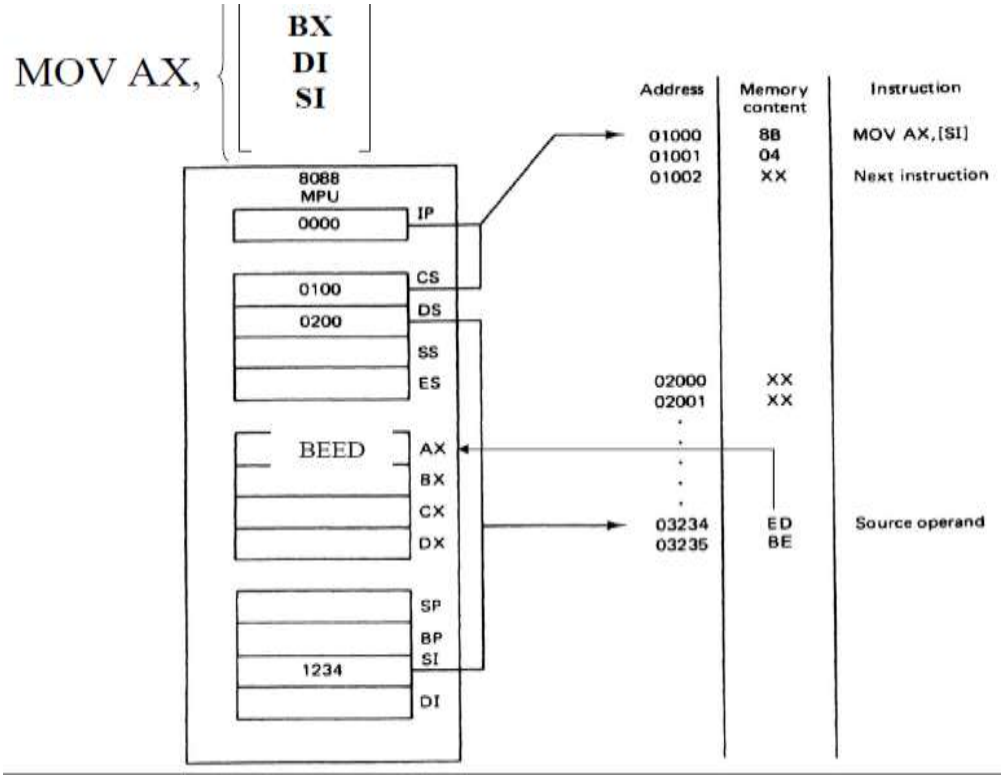
Sol:

a)PA of the operand = DS0 + SI □ 2200+1456 = 3656h

PA of the instruction = CS0+IP □ 3000+0000=3000h

a)Machine code:- 03000 8B

03001 04
c)so , the PA to the next instruction will be □ 03002h
d)2 bytes.



NOTE:

BX, SI, DI and sometimes BP are the only register that can be used with this mode. The BP register is an exception. If the BP is used with register indirect addressing, the data item is not assumed to be in data segment DS, but instead is found in the stack segment SS.

Example3:

If the contents of DS=300h, SS=500h, BX=4573h and BP=89A4h then calculates the PA of the operand for the following instructions:

MOV [BX], 23 PA= DS0+BX □ PA=3000 + 4573=7573h

MOV [BP], 23 PA= SS0+BP □ PA=5000 + 89A4=D9A4h

H.W:

Assume that DS=1120h, SI=2498h, and AX=17FEh, CS=400h, IP=900h

MOV [SI], AX

1. Calculates the PA for operand and the PA for the next instruction.
2. Show the contents of memory locations after the execution of instruction.
3. Write the machine code for the instruction (Assume the opcode = 345Dh).

3.3- Based relative addressing mode:

In the based relative addressing mode, the register BX and BP , as well as the Displacement value, are used to calculate the effective address (EA).

The default segments used to calculate the physical address (PA) are DS for BX

and SS for BP.

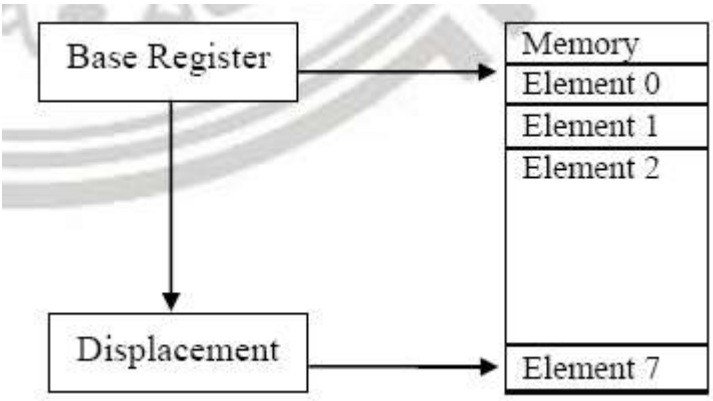
PA= SA+EA

EA in this mode calculate as :

EA= BX or BP + Displacement (8-bits or 16-bits)

- The displacement must be a constant. It can not be a variable.
- The displacement can be added to the register value (BX or BP) to form theoffset address (EA) which is combined with the shifted (DS or SS) to produce the PA where data reside.
- The value in the base register defines the *beginning of a group of data in amemory* such as an array of data, and the displacement selects an *element of data within this group*. See figure

MOV [BX].7, Element7



➢There are a number of ways to specify the displacement, but not all these methods work with all assemblers. The following are the most common:
MOV AL, [Reg + Disp] ex: MOV AL, [BX+7] □ AL = [DS0+BX+7]
MOV AL, [Reg][Disp] ex: MOV AL, [BX][7] □ AL = [DS0+BX+7]
MOV AL, Disp. [Reg] ex: MOV AL, 7. [BX] □ AL = [DS0+BX+7]
MOV AL, [Reg] +Disp ex: MOV AL, [BX] +7 □ AL = [DS0+BX+7]

Example1:

MOV DL, [BX]. 1234h □ DL= [DS0+BX+1234h]
MOV AX, [BP+05h] □ AL= [SS0+BP+05h] and AH= [SS0+BP+05h +1]
MOV CX, [BX] +10 □ CL= [DS0+BX+10h] and CH= [DS0+BX+10h +1]

Example2:

Assume that CS= 400h, IP= 0000h, DS=200h, BX= 1000h, Beta=1497h, AX= BEEDh
When the following instruction MOV [BX]. Beta, AL is executed:

1- Calculates the PA for operand and the PA for the instruction.

2- Show the contents of memory locations after the execution.

3- Write the machine code for the instruction (opcode for instruction= 678Bh).

4- What is the PA for the next instruction ? and how many bytes this instruction takes?

Sol:

1- The PA for operand= DS0+BX+Beta □ PA= 2000+1000+1497= 4497h

The PA for the instruction= CS0+ IP □ PA=4000+ 0000 = 4000h

2- [4497h] = EDh

3- Machine code:- 04000 8B

04001 67
04002 97
04003 14

4- 04004 □ the next instruction (IP), this inst. takes 4 bytes.

H.W:

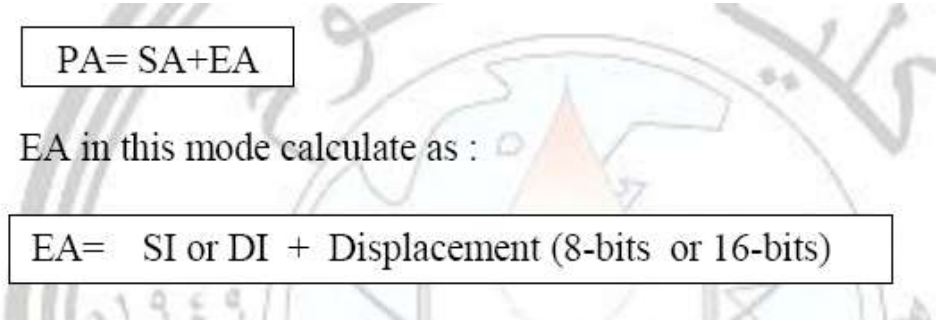
Assume that CS=0100h, SS=1512h, IP= 0000h and AX=17FEh BP=2101h

Calculates the PA and show the contents of memory location after execution

MOV [BP] +20, AL instruction:

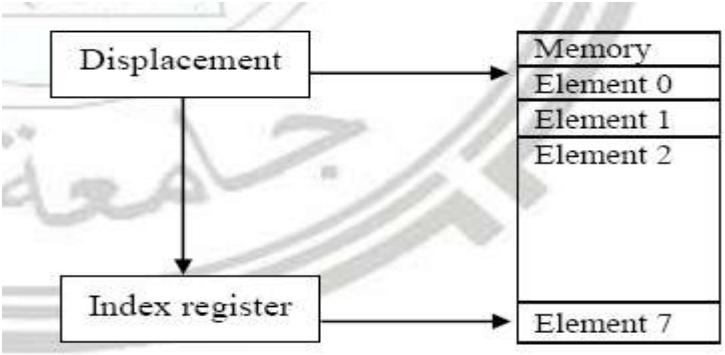
3.4- Indexed relative addressing mode:

The indexed relative a dressing mode works the same as the based relative addressing mode , except that the registers DI and SI is used to hold the offset address(EA) of data, and used the displacement value as an index into the group of data. The default segment used to calculate the physical address (PA) is DS.



With indexed relative addressing mode, the displacement value is used as apointer to the starting point of a group of data (array) in memory, while theindex register (DI or SI) content select the specified element in the array

MOV 7[SI]. Element7



- The general form for indexed relative addressing mode is:

MOV X [SI],AL ex: MOV 4A32h [SI], AL [DS0+ SI +4A32] = AL

MOV AL, X [SI+ Disp] ex: MOV AL, 1235[SI+6]

AL=[DS0+1235+SI+6]

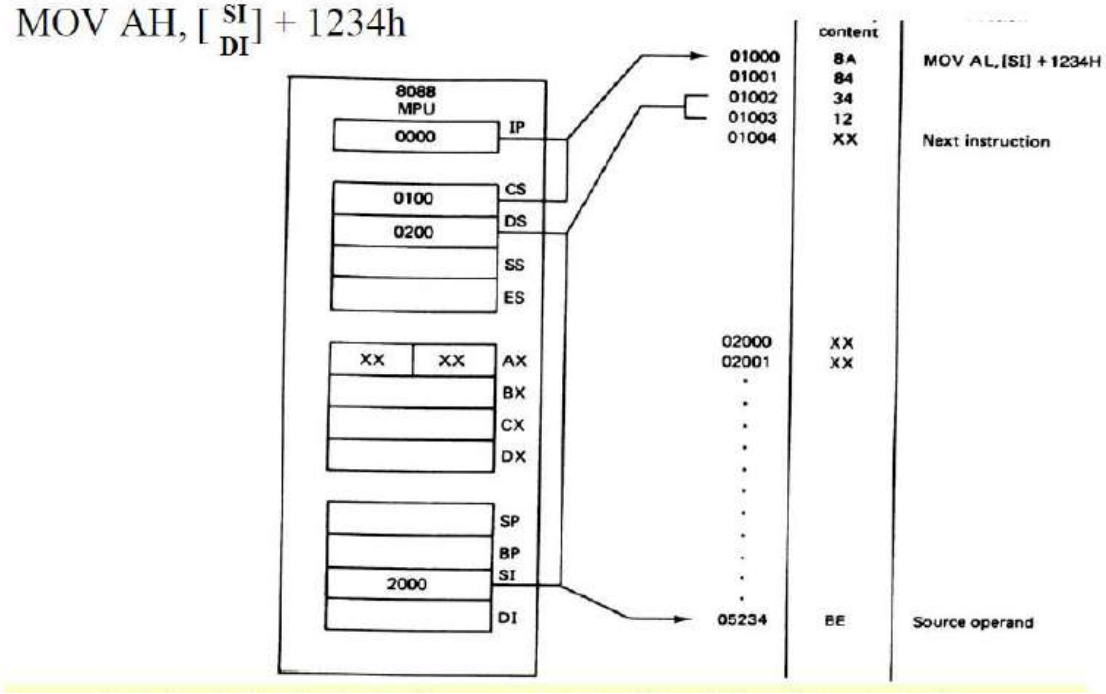
Where: X represents a variable that define the displacement

Example1:

MOV DX, [SI]+5 → DL= [DS0 +SI+5] and DH= [DS0 +SI+5+1]

Example2:

MOV AH, [SI / DI] + 1234h



Example3:
MOV AL, Array [SI] DS=500h, SI= 2000h, CS=200h, IP=0000h and Array=1234h
The contents of memory location 08234h= BEh

- PA for operand= DS0 + SI + Array □ PA=5000+2000+1234= 8234h

PA for the instruction = CS0+ IP □ PA= 2000 + 0000= 2000h

- The contents of AL= BEh

- If the opcode of instruction = 448Bh, then

Machine code: - 02000 8B
02001 44
02002 34
02003 12
02004 □ the next instruction (IP)

- This instruction occupided 4 bytes from memery.

EA= BX or BP + SI or DI + Displacement (8-bits or 16-bits)

H.W:
Assume that CS=2100h, DS=4500h, IP= 0100h, AX=2512h. SI=1486h and Alpha= 1234h. answer the following for MOV [SI]+ Alpha, AX

1.Calculates the PA for code and PA for data.

2.Show the contents of memory location after execution.

3.Write the machine code for the instruction (op code = 0489h)

4.What is the PA of next instruction? And how many bytes this instructionoccupied?

3.5- Based indexed addressing mode:

By combining based and indexed addressing modes, a new addressing mode is derived called the based indexed addressing mode. In this mode, one base register and one index register are used. Cannot use BX with BP nor can we use SI with DI.

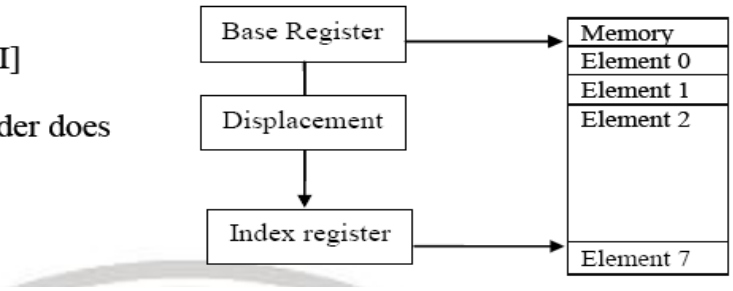
EA in this mode calculate as :

EA= BX or BP + SI or DI + Displacement (8-bits or 16-bits)

Then PA= SA+EA

MOV AH, [BX+7+SI]

Note: the register order does Not matter.



 We can place the beginning address of a group of data (array) in the baseregister, and use an index register to hold an index of selected elements intoan array.

 The general form for this mode is:

- MOV AL, [BX]+X [SI]
ex: MOV AL, [BX]+4A32h [SI] AL=[DS0+ BX +4A32+ SI]

- MOV AX, [BX][SI+ Disp]

ex: MOV AL, [BX][SI+6] AL=[DS0+BX+SI+6]

 The advantage of this addressing mode is that we can access the elements ofan array without using the array name.

Example1:

- MOV CX, [BX][DI]. Beta CL= [DS0+ BX+DI+ Beta]

CH= [DS0+ BX+DI+ Beta+1]

- MOV AH, [BP][DI]+12 AH= [SS0 +BP + DI +12]

محاضره(6)

Example2:

BP=5h, SI=7h, DS=700h, SS=500h ,CS=200h, IP=200h, MOV AX, [BP].100[SI]
1- What is the EA for the data.
2- Calculate the PA for the code and PA for the data.
Sol:
1- EAdata=Disp+SI+BP EAdata= 100+7+5=10Ch
2- PAcode=CS0+IP PAcode=2000+200=2200h
PAdata= Shifted segment+EA PAdata=SS0+10Ch=5000+10C=510Ch
NOTE:
MOV AX, [SI][DI]+ 5 this is illegal instruction , because the two registers are an index registers.
NOTE:
We can use any instruction as long as that instruction supports the addressing mode, but we used the MOV instruction as an example to illustrate addressing mode.
NOTE:
If the displacement is used with this addressing mode then it is called ***based indexed relative addressing mode*** instead of ***based indexed addressing mode.***
MOV CX, [SI][BP] CL=[SS0+SI+BP]
CH=[SS0+SI+BP+1]
Based indexed addressing mode
MOV DX, [SI][BX]+6hDL=[DS0+SI+BX+6h]
DH=[DS0+SI+BX+6h+1]
Based indexed relative addressing mode
►There are other Types of addressing modes such as:

1.String Addressing mode:

The String instruction automatically use the source and destination indexregister (SI, DI) to specify PA.

2.Port Addressing mode:

This mode is used in conjunction with IN and OUT instructions to access inputand output ports. For ports in the I/O address space, only the direct addressingmode and an indirect addressing mode using DX are available
Example:

IN AL, DX and OUT DX, AL (which explained later)

Segment Overrides:

The 8086 CPU allows the program to override the default segment and use any segment register. To do that, specify the segment in the code.
For example:
MOV AL, [BX] → the logical address DS: BX, because DS is the default segment for pointer BX to override that default, specify the desired segment in the instruction as:
MOV AL, ES: [BX] →the logical address ES: BX instead of DS: BX.
Example1:

- MOV AX, SS:[SI] → SS:SI instead of default DS: SI.

- MOV SS:[BX][DI]+32, AX→ SS:BX+DI+32 instead of default DS:BX+ID+32.

- MOV AX, CS:[BP] → CS:BP instead of default SS:BP.

Example2:

Segment override

Segment Registers	CS	DS	ES	SS
Offset Register	IP	SI,DI,BX	SI,DI,BX	SP,BP

Instruction Examples	Override Segment Used	Default Segment
MOV AX,CS:[BP]	CS:BP	SS:BP
MOV DX,SS:[SI]	SS:SI	DS:SI
MOV AX,DS:[BP]	DS:BP	SS:BP
MOV CX,ES:[BX]+12	ES:BX+12	DS:BX+12
MOV SS:[BX][DI]+32,AX	SS:BX+DI+32	DS:BX+DI+32

Summary of the addressing modes

Addressing Mode	Operand	Default Segment
Register	Reg	None
Immediate	Data	None
Direct	[offset]	DS
Register Indirect	[BX] [SI] [DI]	DS DS DS
Based Relative	[BX]+disp [BP]+disp	DS SS
Indexed Relative	[DI]+disp [SI]+disp	DS DS
Based Indexed Relative	[BX][SI or DI]+disp [BP][SI or DI]+disp	DS SS

H.W:
Assume that DS=4500h, SS=2000h, BX =2100h, SI=1486h, DI 8500h, BP=7814h, and AX=2512h.
1-Show the the PA location where AX content is stored in each of the following.

A)MOV [BX] +20, AXB)MOV [SI] +10. AX

C)MOV [DI][BX]+4, AXD)MOV [BP] +12. AX

2- What is the type of addressing mode for each previous instruction?

محاضره(7)

Encoding of 8086 Instructions:

8086 Instructions are represented as binary numbers Instructions require between 1 and 6 bytes

byte	7	6	5	4	3	2	1	0	
1	opcode						d	w	Opcode byte
2	mod		reg			r/m			Addressing mode byte
3	[optional]								low disp, addr, or data
4	[optional]								high disp, addr, or data
5	[optional]								low data
6	[optional]								high data

***Note*:**

The order of bytes in an assembled instruction:

[Prefix] Opcode [Addr Mode] [Low Disp] [High Disp] [Low data] [High data]

Prefix Bytes

One type of prefix instructions is (Segment Overrides) which decodes as follows:

CS 2Eh

DS 3Eh

SS 36h (See example7)

Byte one:

- Opcode field specifies the operation performed (mov, add, sub etc)

- d (direction) field specifies the direction of data movement:

d = 1 data moves from operand specified by R/M field to operand specified by REG field
d = 0 data moves from operand specified by REG field to operand specified by R/M field
Note: d position may be replaced by "s" bit
s = 1 one byte of immediate data is present which must be sign-extended to produce a 16-bit operand
s = 0 two bytes of immediate are present
Note: d position is replaced by "c" bit in Shift and Rotate instructions indicates whether CL is used for shift count

- W (word/byte) specifies operand size

W = 1 data is word
W = 0 data is byte

Byte two:

Contains three fields (Mod, Reg and R/M)

- Mod Bits 6,7 (mode; determines how R/M field is interpreted

- Reg Bits 3, 4, 5 (register) or SREG (Seg register)

- R/M Bits 0, 1, 2 (register/memory) Specifies details about operands

MOD

00 Memory mode, no displacment follows.
01 memory mode with 8-bit displacement follows.
10 memory mode with 16-bit displacement follows.
11 Register mode no displacement follows.

MOD = 11			EFFECTIVE ADDRESS CALCULATION			
R/M	W = 0	W = 1	R/M	MOD = 00	MOD = 01	MOD = 10
000	AL	AX	000	(BX) + (SI)	(BX) + (SI) + D8	(BX) + (SI) + D16
001	CL	CX	001	(BX) + (DI)	(BX) + (DI) + D8	(BX) + (DI) + D16
010	DL	DX	010	(BP) + (SI)	(BP) + (SI) + D8	(BP) + (SI) + D16
011	BL	BX	011	(BP) + (DI)	(BP) + (DI) + D8	(BP) + (DI) + D16
100	AH	SP	100	(SI)	(SI) + D8	(SI) + D16
101	CH	BP	101	(DI)	(DI) + D8	(DI) + D16
110	DH	SI	110	DIRECT ADDRESS	(BP) + D8	(BP) + D16
111	BH	DI	111	(BX)	(BX) + D8	(BX) + D16

REG

Used when the mod field = 11, *look at the left part of the above table*

SREG

000 ES
001 CS

010 SS
110 DS
R/M
Used when the mod field = 00, 01 or 10, *look at the right part of the above table*
Notes:

- In general is not present if instruction has no operands.

- For one-operand instructions the R/M field indicates where the operand is to be found.

- For two-operand instructions (except those with an immediate operand) one is a register determined by REG (SREG) field and the other may be register or memory and is determined by R/M field.

- Direction bit has meaning only in two-operand instructions Indicates whether "destination" is specified by REG or by R/M, note that this allows many instructions to be encoded in two different ways.

Example

MOV BL,AL

- Opcode for MOV = 100010
- We'll encode AL so
  - D = 0 (AL source operand)
- W bit = 0 (8-bits)
- MOD = 11 (register mode)
- REG = 000 (code for BL)
- R/M = 011

OPCODE	D	W	MOD	REG	R/M
100010	1	0	11	011	000

MOV BL,AL => 100010 10 11011000 = 8AD8 h

Example

MOV AL, BL => 100010 10 11000011 = 8AC3h
ADD AX,[SI] => 00000011 00000100 = 03 04 h
ADD [BX][DI] + 1234h, AX => 00000001 10000001 __ __ h
=> 01 81 34 12 h

Example

- To POP into AX:

```
MOD = 11 (Use REG table)    R/M = 000 ->11 000 000
Encoding: 8F C0
```

- To POP into BP:

```
MOD = 11 (Use REG table)    R/M = 101 ->11 000 101
Encoding: 8F C5
```

MOV reg/mem to/from reg/mem

- **MOV [BX+10h],CL**
  - w = 0 because we are dealing with a byte
  - d = 0 because we need R/M Table 2 to encode [BX+10h]
- therefore first byte is **10001000 = 88H**
- since 10H can be encoded as an 8-bit displacement, we can use

```
MOD=01 REG=001 and R/M=111  = 0100 1111 = 4FH
```

- and the last byte is 10H

```
result: 88 4F 10
Note: MOV [BX+10H],CX = 89 4F 10
```

- since 10H can also be encoded as a 16-bit displacement, we can use

```
MOD=10 REG=001 and R/M=111  = 1000 1111 = 8FH
```

- and the last bytes are 00 10

```
result: 88 8F 00 10
```

Example

MOV reg/mem, imm

- This instruction has the structure:

```
1100 011w   MOD 000 R/M disp1    disp2
```

- Where 0, 1 or 2 displacement bytes are present depending on value of MOD
- **`MOV BYTE PTR [100h],10h`**
  - w = 0 because we have byte operand
  - MOD = 00 (R/M Table 1) R/M = 110 (Direct Addr)
  - bytes 3 and 4 are address; byte 5 immediate data
- Result

```
C6 06 00 01 10
```

- **`MOV WORD PTR [BX+SI],10h`**
  - w = 1 because we have byte operand
  - MOD = 00 (R/M Table 1) R/M = 000 ([BX+SI])
  - bytes 3 and 4 are immediate data
- Result

```
C7 00 10 00
```

محاضره(8)

Example

MOV imm to reg

- This instruction is optimized as a 4-bit opcode, with register encoded into the instruction

  1011wreg

- Examples

```
MOV bx, 3          1011 w=1 reg=011=BX
10111011 imm       BB 03 00
MOV bh, 3          1011 w=0 reg=111=BH
10110111 imm       B7 03
MOV bl, 3          1011 w=0 reg=011=BL
10110011 imm       B3 03
```

Example : prefix byte

POP Reg/Mem

- To POP into memory location DS:1200H

```
MOD = 00      R/M = 110    00 000 110
Encoding      8F 06 00 12
```

-
- To POP into memory location CS:1200H add a prefix byte

```
CS = 2Eh
Encoding = 2E 8F 06 00 12
```

Immediate Mode Instructions

- Immediate mode instructions have only one register or memory operand; the other is encoded in the instruction itself

  The Reg field is used an "opcode extension"

  The addressing mode byte has to be examined to determine which operation is specified

```
add imm to reg/mem   1000 00sw   mod000r/m
or imm to reg/mem    1000 00sw   mod001r/m
```

- In instructions with immediate operands the "d" bit is interpreted as the "s" bit
- When the s bit is set it means that the single byte operand should be sign-extended to 16 bits

ADD imm to reg/mem

- add dx, 3 ;Add imm to reg16

  1000 00sw mod000r/m

  w=1 (DX is 16 bits)

  mod = 11 (use REG table) r/m = 010 =DX

- With s bit set we have

  1000 0011 11 000 010 operand = 83 C2 03

- With s bit clear we have

  1000 0001 11 000 010 operand = 81 C2 03 00

Assembler Directives:

Assembler directives also called ***pseudo – operations***: are commands that are understood by the assembler and do not correspond to actual machine instructions.
They are generally used to either instruct the assembler to do something or inform the assembler of something. They are not translated into machine code.
► Common uses of directives are:

1. Define simple constants (predefined constants).

2. Define memory to store data into (by defining different data type).

3. Group memory into segments.

- The EQU (equate) directive:

EQU directive used to define a *constant* without occupying a memory location. The constant value cannot be redefined later in the program.
The format is: *Constant name* **EQU** *constant value*
Example:
Count EQU 25 (as CONST count = 25 in Pascal language)
When executing the instruction (MOV CX, Count). The CX register will be loaded with the value 25. CX=0019 $_h$

- The ORG (Origin) directive:

ORG directive is used to indicate the *beginning of the offset address*. The number that comes after ORG can be either in hex or in decimal. It can also be used for the offset of the code segment (IP).
Example:
Org 100h in the beginning of the program means the IP=100
Then the beginning PA of the program is CS_0+100

محاضره(9)

Assembler Data directives:

The 8086 MP supports many data types, but none are longer than 16-bit wide since the size of the registers is16-bits. It is the job of the programmer to break down data larger than 16-bits to be processed by the CPU. The 8086 data types can be 8-bit or 16-bit, positive or negative. If the number is less than 8-bits wide, it still must be coded as an 8-bit register with the higher digits as zero.

The following are some of the data directives used by 8086 microprocessor:

- The DB (Define Byte) directive: *label* **DB** *value 1byte*

DB directive is one of the most widely used data directives in the assembler. It allows allocation of memory in ***byte-sized***. This is the smallest allocation unit permitted. DB can be used to define numbers in decimal, binary, hex, and ASCII. DB is the only directive that can be used to define ASCII string larger than two characters; therefore it should be used for all ASCII data definitions.

Example:

memory contents	meaning	Data directive
2Dh	decimal value	Data1 DB 45
89h	binary	Data2 DB 10001001B
12h	Hex	Data3 DB 12h
32 35 39 31	ASCII numbers (4 bytes)	Data4 DB '2591'
??	Set aside byte	Data5 DB ?
6D 69 63 72 6F	ASCII character (5 bytes)	Data6 DB 'micro'

Note: Either ' or " can be used with ASCII string

Example:
Z=A+B, Where each a, b and z are memory locations

```
org 100h

mov si, offset x
lea di, y
mov al,[si]
add al,[di]
lea si, z
mov [si],al

ret
X DB 07h          ; X=07
Y DB 02h          ; Y=02
Z DB ?
```

The DUP (Duplicate) directive:

DUP directive is used to ***duplicate*** a given number of characters to avoid a lot of typing.
Format: *label no. of duplicated location* **DUP (***duplicated value***)**
Example1:
The following two methods of filling six memory locations (each of one byte) with
FFh is equivalent:
Data1 DB 0FFh, 0FFh, 0FFh, 0FFh, 0FFh, 0FFh Data1 DB 6DUP (0FFh)
Example2:
XX DB 32DUP(?) ; means aside 32 bytes with no initial value given.
Example3:
DUP can be used inside another DUP
Data4 DB 5 DUP (2 DUP(99)) ; means fill 10 bytes with 99.

The DW (Define Word) directive:

DW directive is used to allocate memory ***2 bytes*** (***one word at a time***). DW is used widely in 8086 and 286 MP since the registers are 16-bit length.

Equivalent in hex	meaning	Data directive
03 BA	decimal value	Data1 DW 954
09 54	binary	Data2 DW 100101010100B
25 3F	hex	Data3 DW 253Fh
00 09 00 02 00 07 00 0C 00 20 00 05 48 49	Miscellaneous data	Data4 DW 9, 2, 7, 0Ch, 00100000B, 5, 'HI'
?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??	Set aside 8 word (16 bytes)	Data5 DW 8 DUP(?)

The DD (Define Double) directive:

The **DD** directive is used to allocate memory locations that are ***4 bytes*** (***two words***) in size. The data can be in decimal, binary, or hex. In any case the data is converted to hex and placed in memory locations according to the rule of low byte to low address and high byte to high address.

Example:

Equivalent in hex	meaning	Data directive
00 00 03 FF	decimal value	Data1 DD 1023
00 08 965C	binary	Data2 DD 10001001011001011100B
5C 2A 57 F2	hex	Data3 DD 5C2A57F2 h
00 00 00 23 00 03 47 89 00 00 FF FD	Miscellaneous data	Data4 DD 23h, 34789h, 65533

The DQ (Define Quadword) directive:

The **DQ** is used to allocate memory ***8 bytes*** (***four words***) in size. This can used to represent any variable up to ***64 bits*** wide.

- The DT (Define Ten bytes) directive:

The **DT** directive is used for memory allocation of ***BCD numbers***. This allocates ***10 bytes***, but maximum of ***18 digits*** can be entered.

What is a MACRO and how it is used:

There are applications in assembly language programming where a group of instruction performs a specific task that is used repeatedly, so it doesn't suitable to rewrite them every time they are needed. ***MACRO*** allows the programmer to rewrite the task once only and invoke it whenever and wherever it is needed. The concept of MACRO was found to *reduce* the time that it takes to write these codes and *reduce* the possibility of errors.

MACRO Definition:

Every macro definition must have three parts, as follows:

Macro Name **MACRO** paramerter1, paremeter2, paremeter3 ....

……

……

ENDM

MACRO directive indicated the beginning of the macro definition and **ENDM** signals the end of it. The body of macros will be written between these two directives.

When a macro is called with actual arguments, these are *substituted* for the dummy arguments (macro parameters), and the statements are assembled.

Example1

Let us define a macro SUM that adds one byte in memory X to another byte in memory Y and puts the result after location Y in memory.

```
SUM MACRO X, Y ; macro definition
LEA SI, X
LEA DI, Y
MOV Al, [SI]
ADD Al, [DI]
MOV [DI+1], Al
ENDM
org 100h
sum m, n ; macro call
ret
m DB 12h
n DB 11h
```

Macro definition

```
SUM MACRO X, Y
LEA SI, X
LEA DI, Y
MOV AX, [SI]
ADD AX, [DI]
MOV [DI+1], AX
ENDM
```

Example2:

Multiply Two 8 bits numbers, and store the result in memory location its name RESULT. (use MACRO)

```
multiply  MACRO  d1, d2    ; macro definition
              lea si , d1     ; the same as mov si, offset d1
              mov al,[si]
              lea si, d2
              mul [si]      ; the result now in AX
              lea si, result
              mov [si],ax
          ENDM
org 100h
   multiply d1, d2

ret

result DW ?
d1 db 8h
d2 db 4h
```

NOTE:
Without using MACRO solved as follows:

```
org 100h
mov al, 8h
mov cl, 4h
mul cl ;the result now in AX
lea si, result
mov [si], ax
ret
```

محاضره(10)

H.W1:
Write assembly program that define a MACRO to swap two memory locations begins with offset BYTE1 and BYTE2.
H.W2:
Find the average for the contents of three 8 bit numbers and store result in memory location with offset 300h. (Use MACRO)
H.W3:
You have two numbers, the first one is 32 bits and the second is a 16 bits number. Find the result of division and store it in memory location called RESULT then store the reminder of division in memory location called MOD. (Use MACRO)

Computer Program

The user program in its original alphanumeric text format is called a ***source program*** (*.asm), and the assembled machine language program is called an ***object program*** (*.obj).
This is done by the **assembler program**, which replaces all symbols denoting operations and addressing modes with the binary codes used in machine instructions, and replaces all names and labels with their actual values.
One or more object code files linked together and possibly with one or more library files to produce ***executable program*** (*.exe) that can be executed by the Microprocessor. This is done by **linker program.** See figure

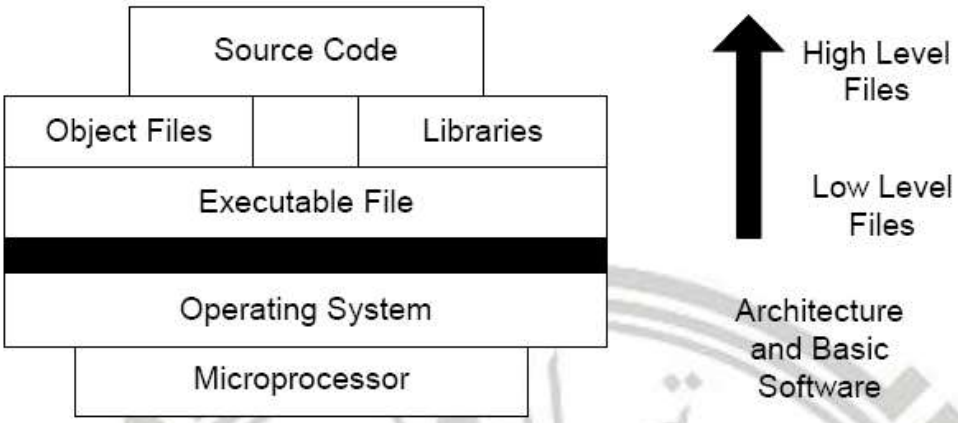
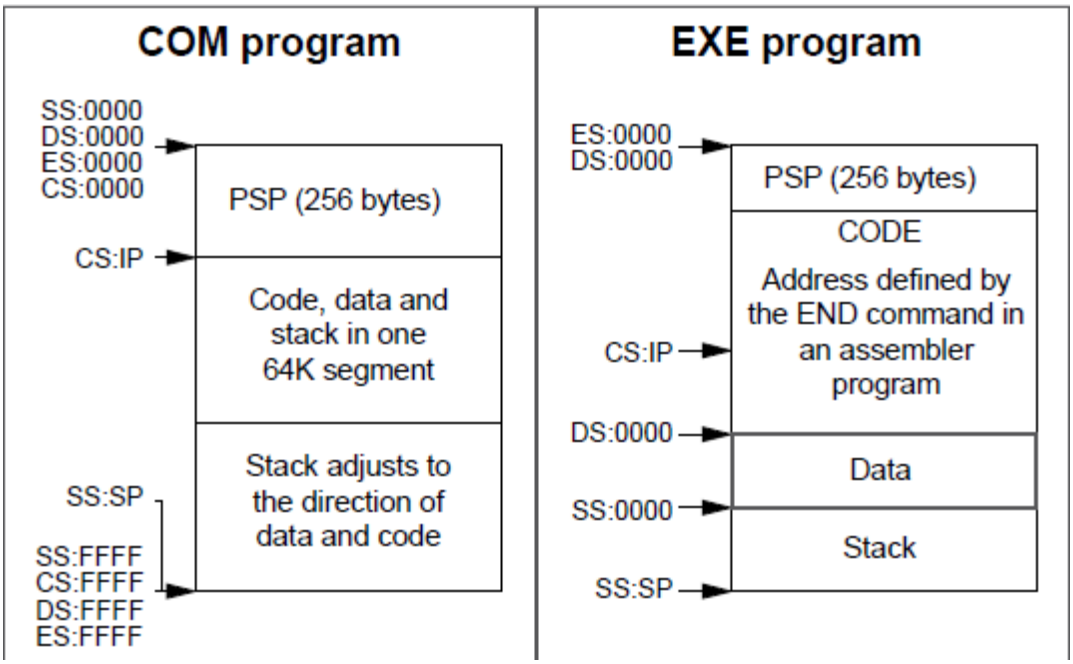


Figure1 (Computer program)

COM Files vs. EXE Files

The COM file is like the EXE file, which contents the executable machine code and can be run at DOS Operating System. COM program files are stored on disk as an image copy of memory, because of this no further processing is needed during loading. The COM files load and start execution faster than EXE files because other information besides the program code and initialized variables are also stored there.

The COM file is useful when there is a limited amount of memory, and when a very compact code is needed. The COM file can't be greater than 64 Kbytes in size because that it is must fit into a ***single segment***, and since in the 8086 MP the size of a segment is 64 KB. To limit the size of the COM file to 64 KB requires defining the data inside the code segment and also using the end area of the code segment for the stack; therefore the COM file has no separate data segment definition. See figure



COM vs. EXE files)

COM File	EXE File
Maximum size 64 K bytes	Unlimited size
Data segment defined in code segment	Data segment is defined
No stack segment definition (define in the end area of CS)	Stack segment is defined
Code and data must be begin at offset 0100h (Org 0100h))	Code, data can be defined at any offset address
Smaller file (takes less memory	Larger file (takes more memory)

Tabel2 (Comparing between COM & EXE files)