

# Micro-Processors and Assembly Programming



## References

- **The 8086 Microprocessors Architecture, software and Interfacing techniques**  
By: *Walter A. Triebel*
- **The 8086/8088 MPU, Architecture, programming and interfacing** BY: *Barry B. Brey*

## **Microprocessor**

A **microprocessor** or processor (sometimes abbreviated  $\mu\text{P}$ ) is the heart of the computer and it performs all the computational tasks, calculations and data processing etc. inside the computer. Microprocessor is the brain of the computer. A **Central processing unit (CPU)** in a **computer system** or handheld device consists of one or more microprocessors. The microprocessors can be classified based on the following features:

## **Instruction Set**

It is the set of the instructions that the Microprocessor can execute.

## **Bandwidth**

The number of bits processed by the processor in a single instruction.

4 bit. e.g. Intel 4004

8 bit. e.g. Intel 8085, 8088, Zilog Z80, Z180

16 bit. e.g. Intel 8086, 80186, 80286, 80386,

32 bit. e.g. Intel Pentium, Celeron, AMD Sempron

64 bit. e.g. AMD Athlon.

## **Clock Speed**

Clock speed is measured in the MHz and it determines that how many instructions a processor can processed.

The speed of the microprocessor is measured in the MHz or GHz. The processor is also known as the CPU (Central Processing Unit). It contains the control unit and the arithmetic unit and both works together to process the commands. CPU is used in every computer whether it is a workstation, server or a laptop. CPU is a complete computational engine that is designed as a chip. It starts the work when you turn on your computer.

CPU is designed to perform the arithmetic and logical operations inside the computer. Common operations inside the computer include adding, subtracting, multiplying, comparing the values and fetching the different numbers to process them. The higher the CPU clocks' speed the more efficient will be the performance of the computer.

Computer's performance is also influenced by the system bus architecture, memory used, type of the processor and software program being running.

## **Basic Block Diagram of Microprocessor:**

Figure 1: Basic Block Diagram of Microprocessor

**ALU** – Performs all arithmetic and logical operations

**Register array** – Holds the data temporarily for processing

**Control Unit** – It supervises/ monitors all the operations carried out in the computer.

## **Microcomputer Architecture**

A computer system has three main components: a **Central Processing Unit** (CPU) or processor, a **Memory Unit** and **Input Output Units** (devices). In any microcomputer system, the component which actually processes data is entirely contained on a single chip called **Microprocessor** (MPU). This MPU can be programmed using assembly language. Writing a program in assembly language requires a knowledge of the computer hardware (or Architecture) and the details of its instruction set.

The main **internal hardware** features of a computer are the processor, memory and **registers** (registers are special processor components for holding address and data).

The **external hardware** features are the computer Input/Output devices such as keyboard, monitor...

**Software** consists of the operating system (O.S) and various programs and data files stored on disk.

Inside any computer based on a member of the 8086 family, the basic arrangement of the main components is shown in Figure 2.

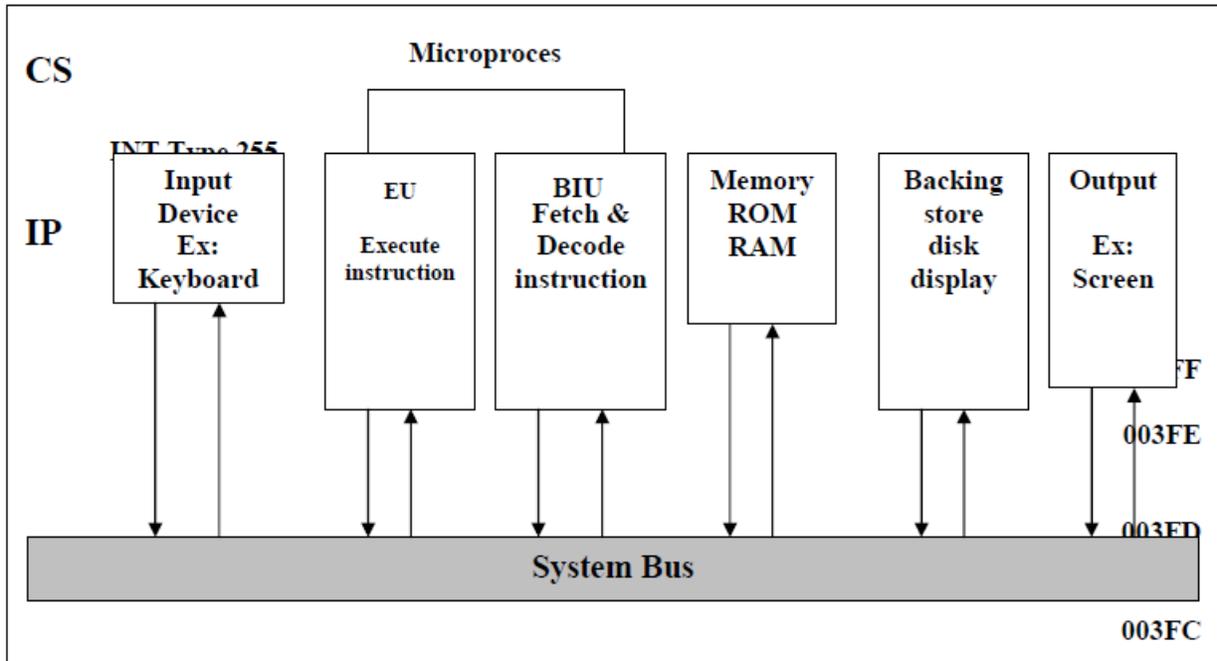


Figure 2: Data flow between the main components of an 8086 family computer.

Information is sent from one main component to another along the communication channel, which is often called the **System Bus**. Both programs and data are stored in the memory. The **Bus Interface Unit (BIU)** within the MPU fetches new instruction or data as necessary. It is also the BIU jobs to interpret or decode instruction and to route results to their proper destination.

The MPU **Execution Unit** carries out any arithmetic which is required, including memory calculation. Microcomputer memories consist of a collection of chips of two kinds **Read Only Memory (ROM)** and **Random Access Memories (RAM)**.

## System Bus

The components of the computer system must communicate with each other and with the outside world. Although it may be possible to connect each component to the CPU separately as a practical matter this would require too many physical connects. To keep the number of connections manageable, the processor is connected to memory and all peripherals using a bus. A **Bus** is a bunch of wires, and electrical path on the printed IC to which everything in the system is connected.

### **There are three types of Bus:**

- 1- **Address Buss (AB):** the width of AB determines the amount of physical memory addressable by the processor.
- 2- **Data Bus (DB):** the width of DB indicates the size of the data transferred between the processor and memory or I/O device.
- 3- **Control Bus (CB):** consists of a set of control signals, typical control signals includes memory read, memory write, I/O read, I/O write, interrupt acknowledge, bus request. These control signals indicates the type of action taking place on the system bus.

## **Personal Computer (PC) Components**

The main component of the PC is its **System Board** (or mother board). It contains the processor, co-processor, main memory, connectors, and expansion slots for optional cards.

The slots and connectors provide access to such components as ROM, RAM, hard disk, CD-ROM drive, additional memory, video unit, keyboard, mouse, parallel and serial device, sound adapter and cache memory (the processor use high speed cache memory to decrease its need to access the slower main memory). A bus with wires attached to the system board connect the components. It transfers data between the processor, memory and external devices.

### **A. The processor**

The CPU or processor acts as the controller of all actions or services provided by the system. The operations of a CPU can be reduced to three basic steps: **fetch**, **decode**, and **execute**. Each step includes intermediate steps, some of which are:

#### **1- Fetch the next instruction:**

- Place it in a holding area called a queue.
- Decode the instruction.

#### **2- Decode the instruction**

- Perform address translation.
- Fetch operand from memory.

### 3- Execute the instruction.

- Perform the required calculation.
- Store results in memory or register.
- Set status flag attached to the CPU.

Figure 3 shows a block diagram of a simple imaginary CPU. The CPU is divided into two general parts. **Arithmetic Logic Unit (ALU)** and **Control Unit (CU)**.

- The **ALU** carry Arithmetic, logical, and shifting operations.
- The **CU** fetches data and instruction, and decodes addresses for the ALU.

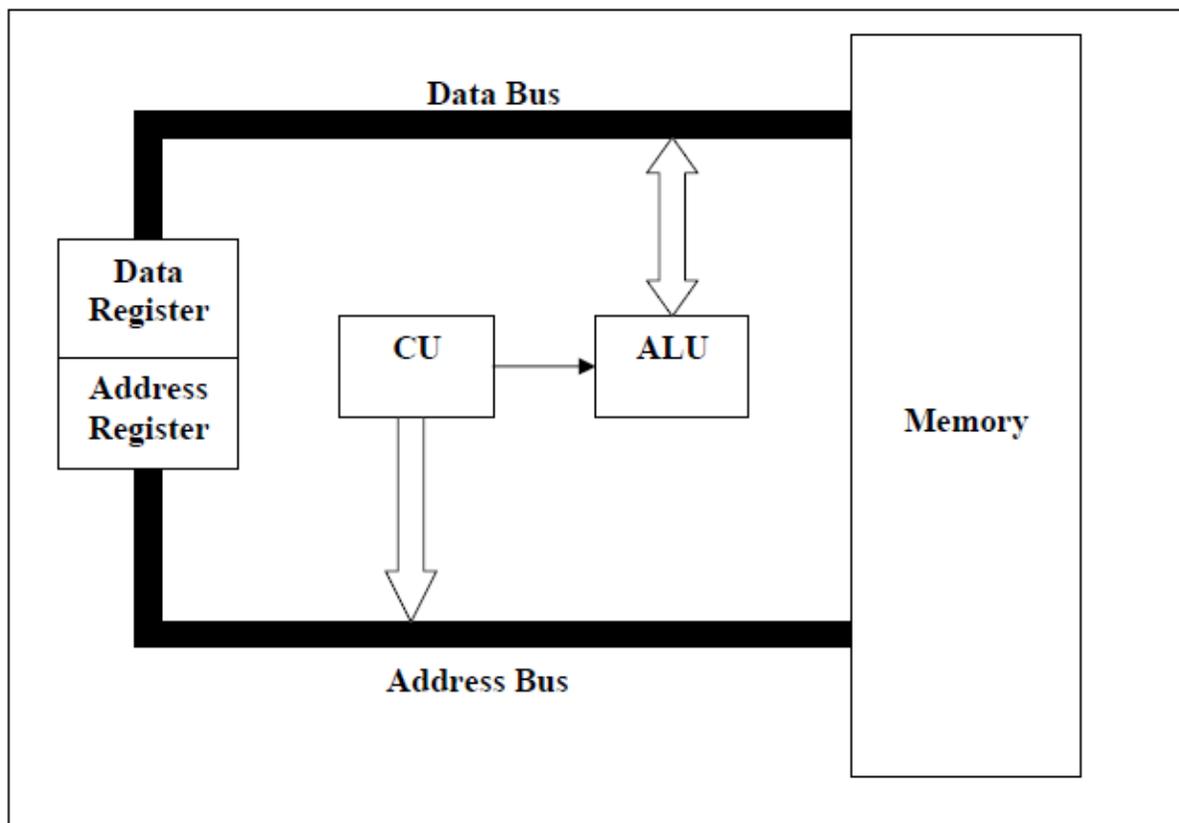


Figure 3: A block diagram of a simple CPU.

## B. Memory

The memory of a computer system consist of tiny electronic switches, with each switch set in one of two states: open or close. It is however more convenient to think of these states **as 0 and 1**.

Thus each switch can represent a binary digit or bit, as it is known, the memory unit consists of millions of such bits, bits are organized into groups of eight bits called **byte**.

Memory can be viewed as consisting of an ordered sequence of bytes. Each byte in this memory can be identified by its sequence number starting with 0, as shown in Figure 4. This is referred to as memory address of the byte. Such memory is called **byte addressable memory**.

**8086** can address up to 1 MB ( $2^{20}$  bytes) of main memory this magic number comes from the fact that the address bud of the 8086 has 20 address lines. This number is referred to as the **Memory Address Space (MAS)**. The memory address space of a system is determined by the address bus width of the CPU used in the system. The actual memory in a system is always less than or equal to the MAS.

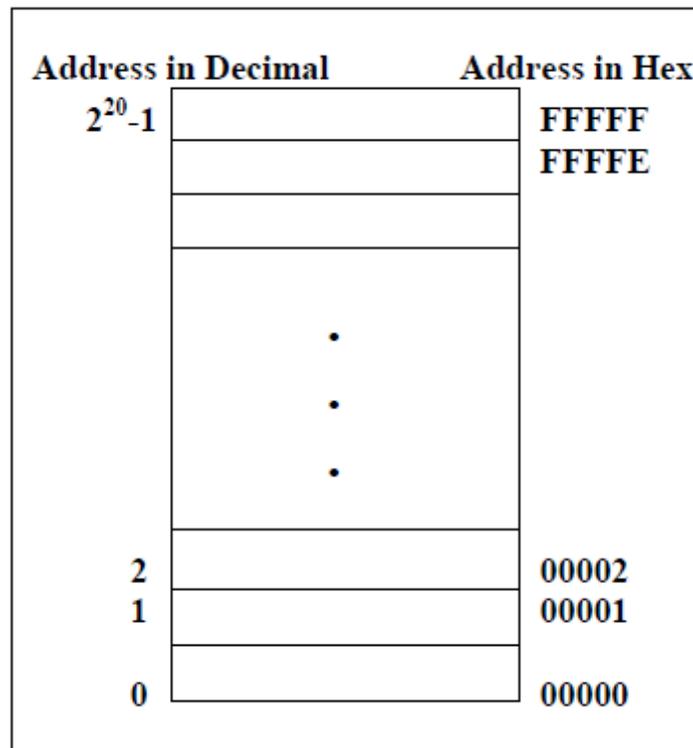


Figure 4: Logical view of the system memory

## Two basic memory operations

The memory unit supports two fundamental operations: Read and Write. The read operation reads a previously stored data and the write operation stores a value in memory. See Figure 5

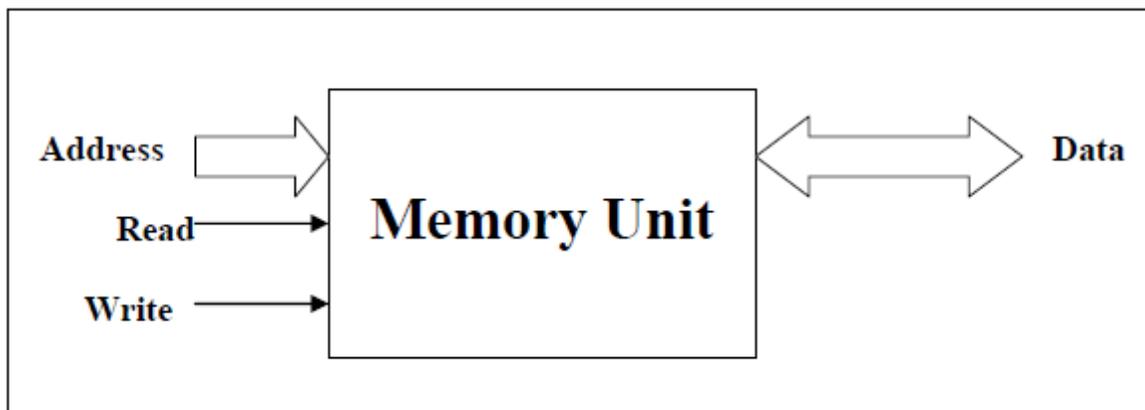


Figure 5: Block diagram of system memory

### Steps in a typical read cycle:

- 1- Place the address of the location to be read on the address bus.
- 2- Activate the memory read control signal on the control bus.
- 3- Wait for the memory to retrieve the data from the address memory location.
- 4- Read the data from the data bus.
- 5- Drop the memory read control signal to terminate the read cycle.

### Steps in a typical write cycle:

- 1- Place the address of the location to be written on the address bus.
- 2- Place the data to be written on the data bus.
- 3- Activate the memory write control signal on the control bus.
- 4- Wait for the memory to store the data at the address location.
- 5- Drop the memory write control signal to terminate the write cycle.

**Addresses:** group of bits which are arranged sequentially in memory, to enable direct access, a number called address is associated with each group. Addresses

start at 0 and increase for successive groups. The term location refers to a group of bits with a unique address. Table 1 represents Bit, Byte, and Larger units.

Table1: Bit, Byte, and Larger units.

Name	Number of Byte
<b>Bit</b>	0 or 1
<b>Byte</b>	is a group of bits used to represent a character, typically 8-bit.
<b>Word</b>	2-bytes (16-bit) data item
<b>Double Word</b>	4-byte (32-bits)
<b>Quadword</b>	8-Bytes (64-bit)
<b>Paragraph</b>	16-bytes (128-bit)
<b>KiloByte (KB)</b>	the number $2^{10} = 1024 = 1K$ for KiloByte, (thus $640K = 640 * 1024 = 655360$ bytes)

## Types of memory

The memory unit can be implemented using a variety of memory chips- different speeds, different manufacturing technology, and different sizes. The two basic types are RAM and ROM.

### 1- Read Only Memories (ROM):

ROMs allow only read operation to be performed. This memory is non-volatile. Most ROMs are programmed and cannot be altered. This type of ROM is cheaper to manufacture than other types of ROM. The program that controls the standard I/O functions (called BIOS) is kept in ROM, configuration software.

#### Other types of ROM include:

- Programmable ROM (PROM).
- Erasable PROM (EPROM) is read only memory that can be reprogrammed using special equipment.
- EAPROM, Electrically Alterable Programmable ROM is a Read Only Memory that is electrically reprogrammable.

## **2- Read/Write Memory**

Read/Write memory is commonly referred to as Random Access Memory (RAM); it is divided into static and dynamic. Static RAM (SRAM): used for implementing CPU registers and cache memories. Dynamic RAM (DRAM), the bulk of main memory in a typical computer system consists of dynamic ram.

**Dynamic RAM:** main memory, or RAM is where program, data are kept when a program is running. It must be refreshed with in less than a millisecond or losses its contents.

**Static RAM,** used for special high speed memory called cache memory which greatly improves system performance. Static RAM keeps its value without having to be refreshed.

## **C. INPUT/OUTPUT**

Input/output (I/O) devices provide the means by which the computer system can interact with the outside world. Computers use I/O devices (also called peripheral devices) for two major purposes:

- 1- To communicate with the outside world and,
- 2- Store data.

Devices that are used to communicate like, printer, keyboard, modem, Devices that are used to store data like disk drive. I/O devices are connected to the system bus through **I/O controller** (interface) – which acts as interface between the system bus and I/O devices.

### **There are two main reasons for using I/O controllers**

1- I/O devices exhibit different characteristics and if these devices are connected directly, the CPU would have to understand and respond appropriately to each I/O device. This would cause the CPU to spend a lot of time interacting with I/O devices and spend less time executing user programs.

2- The amount of electrical power used to send signals on the system bus is very low. This means that the cable connecting the I/O device has to be very short (a

few centimeters at most). I/O controllers typically contain driver hardware to send current over long cable that connects I/O devices. See Figure 6.

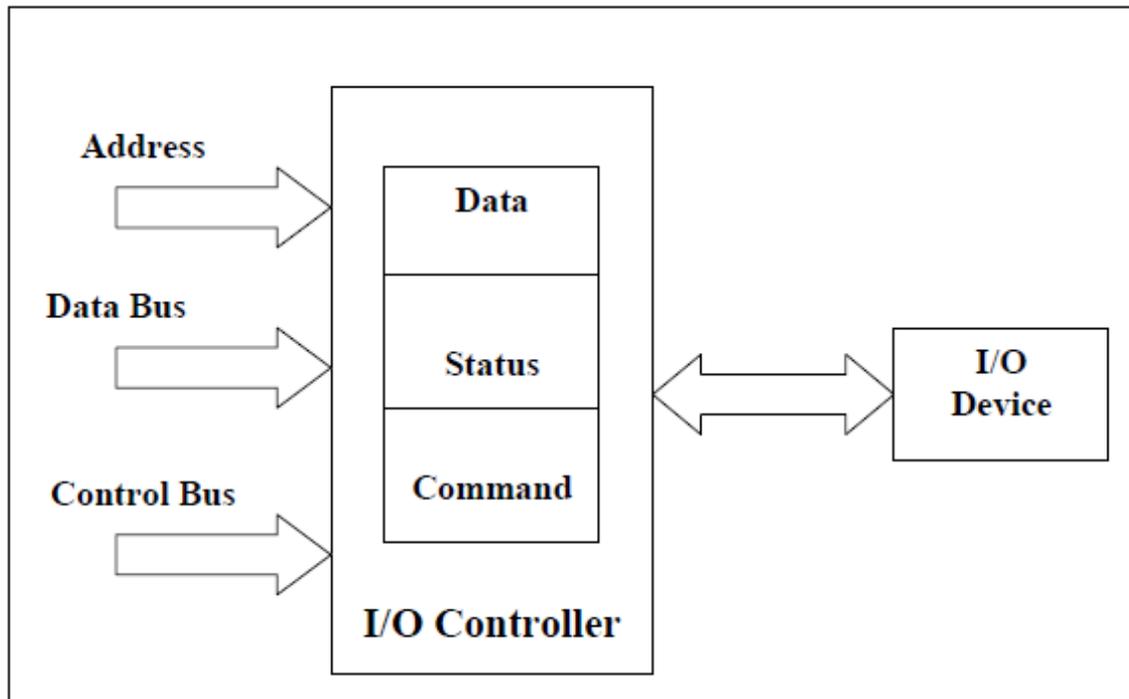


Figure 6: Block diagram of a generic I/O device interface.

## Evolution of Intel Microprocessor

The principle way in which MPU & microcomputer are categorized in term of the maximum number of binary bit 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 2.

**8088** and **8086** functionally identical but **8088** lower performance, **80186** run all **8088** and **8086** software, but have 10 new instructions. **80188** in function are identical to **80186** but lower performances. **80286** run all **8086**, **80186** program, but has extra instruction, more powerful than **8086**. **80386** has various operation mode, which allow it to act as **80286** chip or multiple **8086** chip, as well as a set of instruction capable of 32 bit operations such as arithmetic.

Table 2: Different Microprocessor features descriptions

Microprocessor Name	Features Descriptions		
	Width of (DB)	Width of (AB)	Instruction queue length
8086	16 bit	20 bit	6 Byte
8080	8 bit	20 bit	4 Byte
80186	16 bit	20 bit	6 Byte
80188	8 bit	20 bit	4 Byte
80286	16 bit	24 bit	6 Byte
80386	32 bit	32 bit	6 Byte

## Execution Unit and Bus Interface Unit

In the Figure 7, the processor is partitioned into two logical units:

An **Execution Unit (EU)** and **Bus Interface Unit (BIU)**. The role of the EU is to execute instruction, whereas the BIU delivers instruction and data to EU.

The EU contains **ALU**, **CU** and number of **registers**. This feature enables the EU to execute instructions and perform arithmetic and logical operations. The most important function of BIU is to manage the bus control unit, segment registers instruction queue. The BIU controls the busses that transfer data to the EU, to memory, and to external input/output devices, whereas the segment registers control the memory addressing.

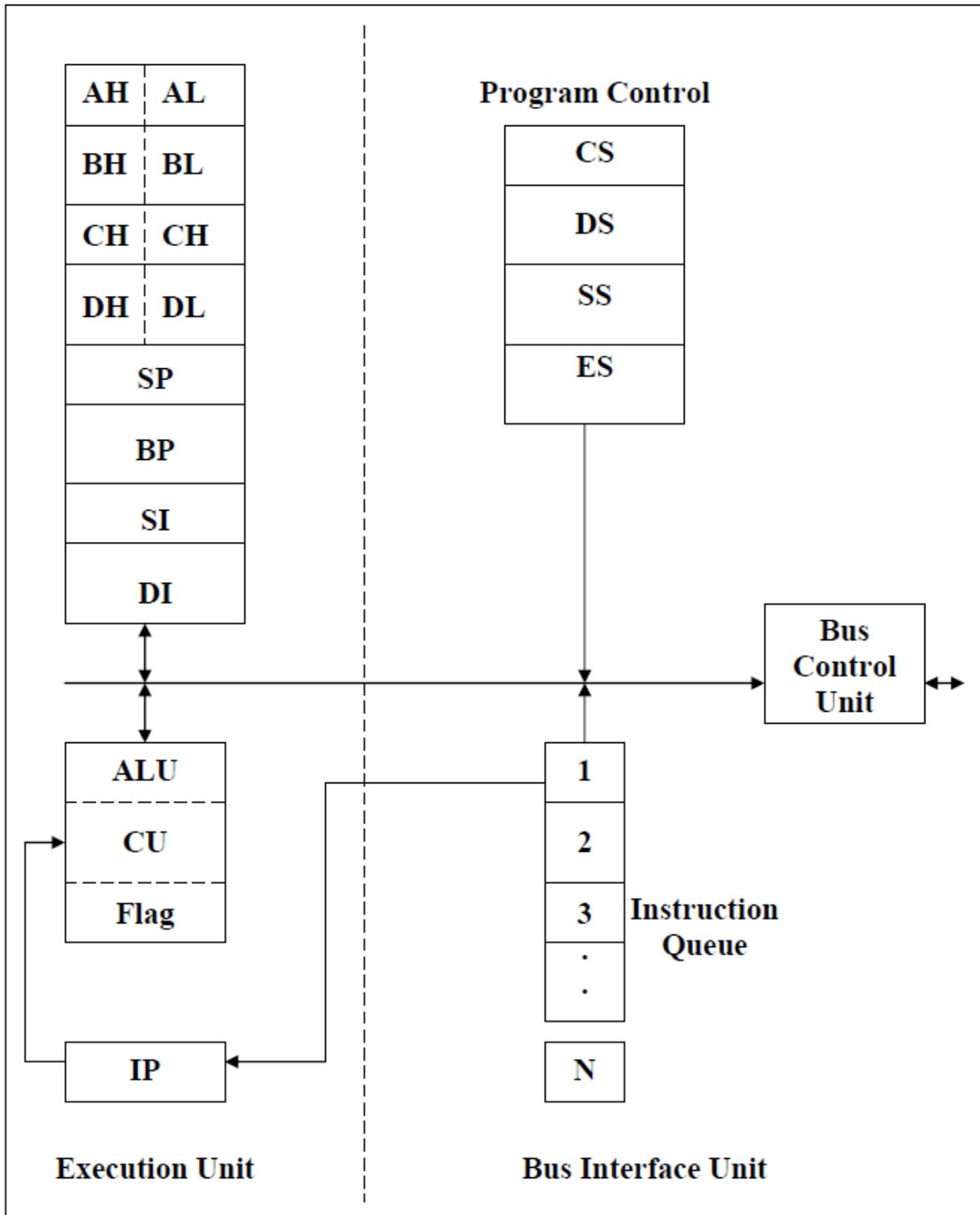


Figure 7: Execution unit and Bus interface unit.

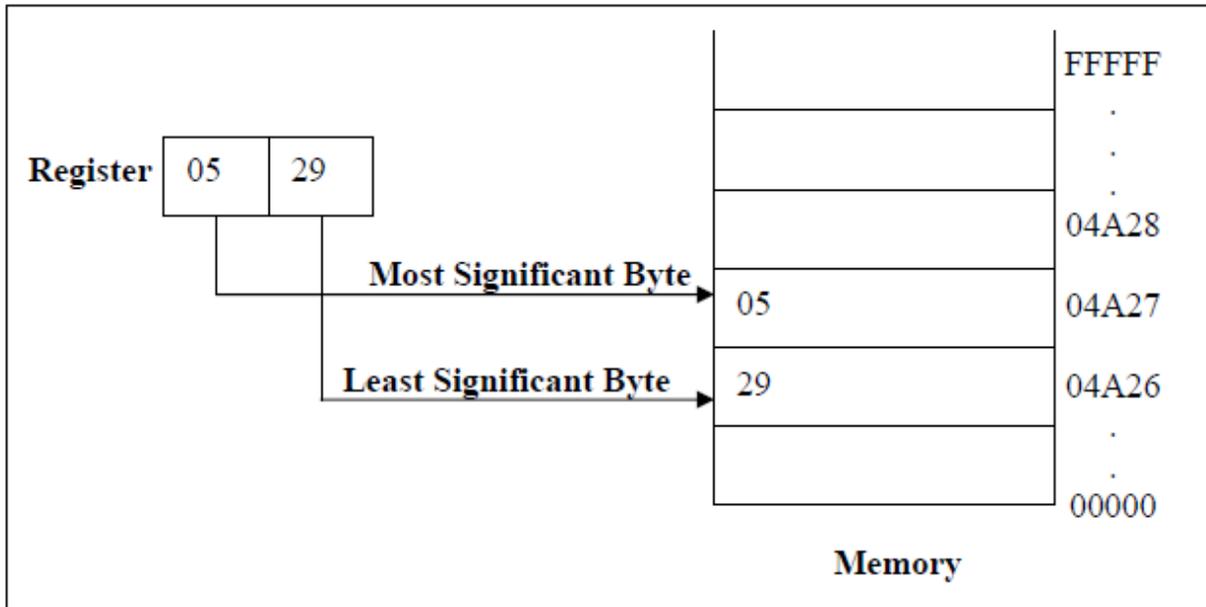
Another function of the BIU is to provide access to instructions, because the instructions for a program that is executing are kept in memory, the BIU must access instruction from memory and place them in an instruction queue, which varies in size depending on the processor. This feature enables the BIU to look ahead and prefetch instructions, so that there is always a queue of instructions ready to execute.

The EU and BIU work in parallel, with the BIU keeping one step ahead. The EU notifies the BIU when it needs access to data in memory or I/O devices. Also the EU request machine code instructions from the BIU instruction queue. The top instruction is the currently executable one, and while the EU is occupied executing an instruction, the BIU fetch another instruction from memory. This fetching overlaps with execution and speeds up processing.

## **Addressing Data in Memory**

Depending on the model, the processor can access one or more bytes of memory at a time. Consider the Hexa value (**0529<sub>H</sub>**) which requires two bytes or one word of memory. It consist of high order (**most significant**) byte 05 and a low order (**least significant**) byte 29.

The processor store the data in memory in reverse byte sequence i.e. the low order byte in the low memory address and the high order byte in the high memory address. For example, the processor transfer the value **0529<sub>H</sub>** from a register into memory address **04A26<sub>H</sub>** and **04A27<sub>H</sub>** like this:



The processor expects numeric data in memory to be in reverse byte sequence and processes the data accordingly, again reverses the bytes, restoring them to correctly in the register as hexa **0529<sub>H</sub>**.

When programming in assembly language, you have to distinguish between the address of a memory location and its contents. In the above example the content of address **04A26<sub>H</sub>** is 29, and the content of address **04A27<sub>H</sub>** is 05.

## There are two types of addressing schemes:

1. **An Absolute Address**, such as **04A26<sub>H</sub>**, is a 20 bit value that directly references a specific location.
2. **A Segment Offset Address**, combines the starting address of a segment with an offset value.

## Segments and Addressing

Segments are special area defined in a program for containing the code, the data, and the stack. **Segment Offset** within a program, all memory locations within a segment are relative to the segment starting address. The distance in bytes from the segment address to another location within the segment is expressed as an **offset** (or displacement).

To reference any memory location in a segment, the processor combine the segment address in a segment register with the offset value of that location, that is, its distance in byte from the start of the segment.

## Specifying addresses

To represent a segment address and its relative offset we use the notation:

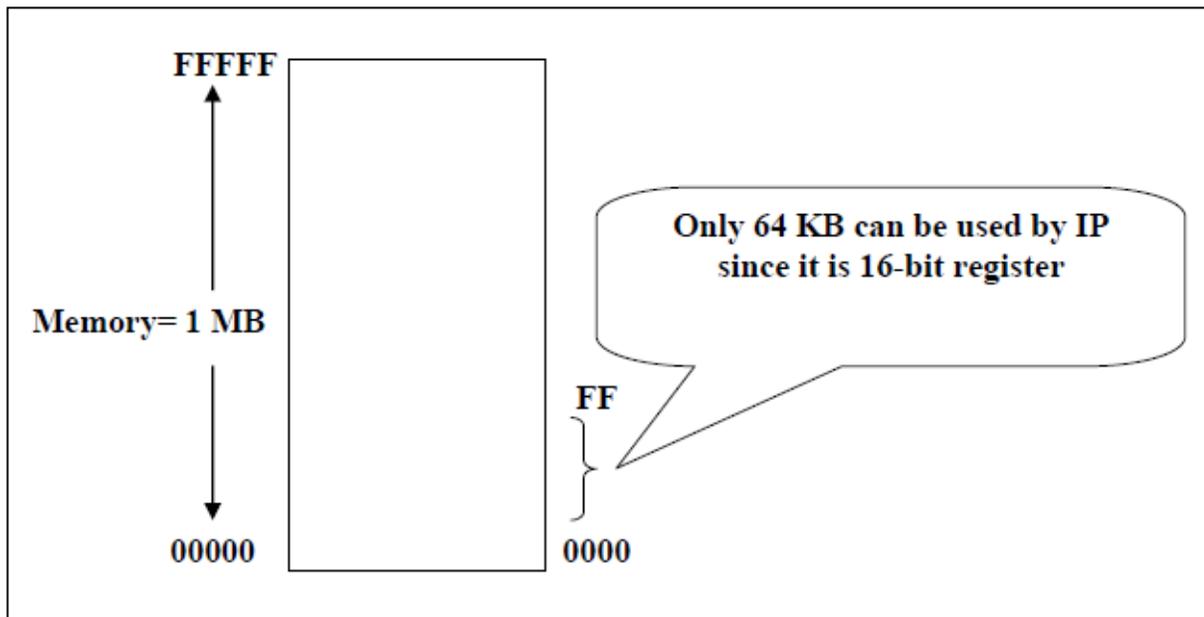
*Segment: offset*

Thus **020A:1BCD** denotes offset **1BCD<sub>H</sub>** from segment **020A<sub>H</sub>**.

The actual address it refers to is obtained in the following way:

- 1- Add zero to the right hand side of the segment address.
- 2- Add to this the offset.

Hence the actual address referred to by 020A:1BCD is 03C6D.



Address Bus in the **8086** is 20 bits wide (20 lines) i.e. the processor can access memory of size  $2^{20}$  or 1048576 bytes (**1MB**).

Instruction Pointer = 16 bit register which means the processor can only address  $0 - 2^{16}$  (65535) bytes of memory. But we need to write instructions in any of the 1MB of memory. This can be solved by using memory segmentation., where each segment register is 16-bit (this 16-bit is the high 16-bit of Address Bus ( $A_4- A_{19}$ )) i.e. each of the segment registers represent the actual address after shifting the address 4-bit to get 20 bits.

## **Registers**

Registers are 8, 16, or 32-bit high speed storage locations directly inside the CPU, designed to be accessed at much higher speed than conventional memory.

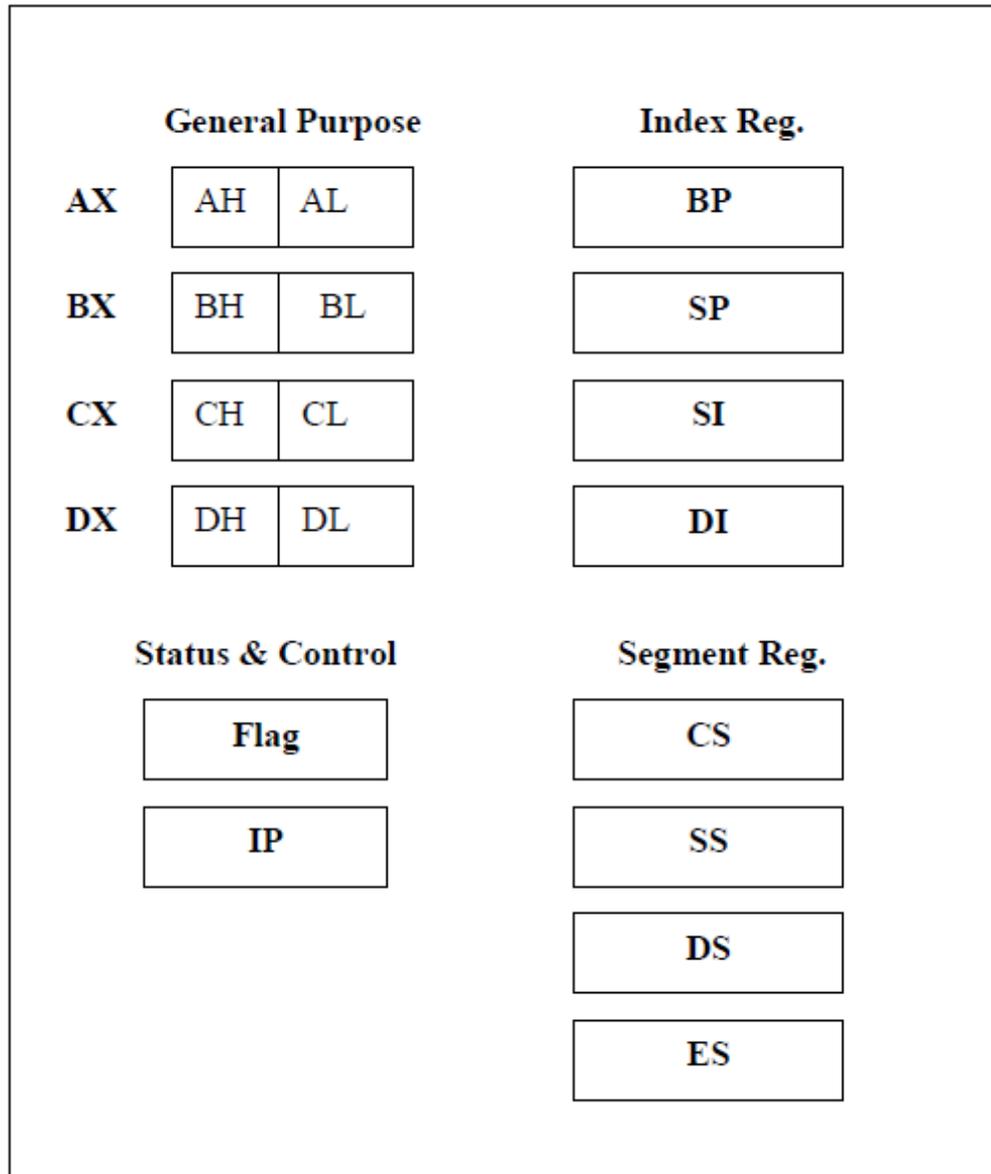


Figure 8: Intel 16-bit registers

The CPU has an internal data bus that is generally twice as wide as its external data bus.

**Data Registers:** The general purpose registers, are used for arithmetic and data movement. Each register can be addressed as either 16-bit or 8 bit value. Example, **AX** register is a 16-bit register, its upper 8-bit is called **AH**, and its lower 8-bit is called **AL**. Bit 0 in AL corresponds to bit 0 in AX and bit 0 in AH corresponds to bit 8 in AX. See Figure 9.

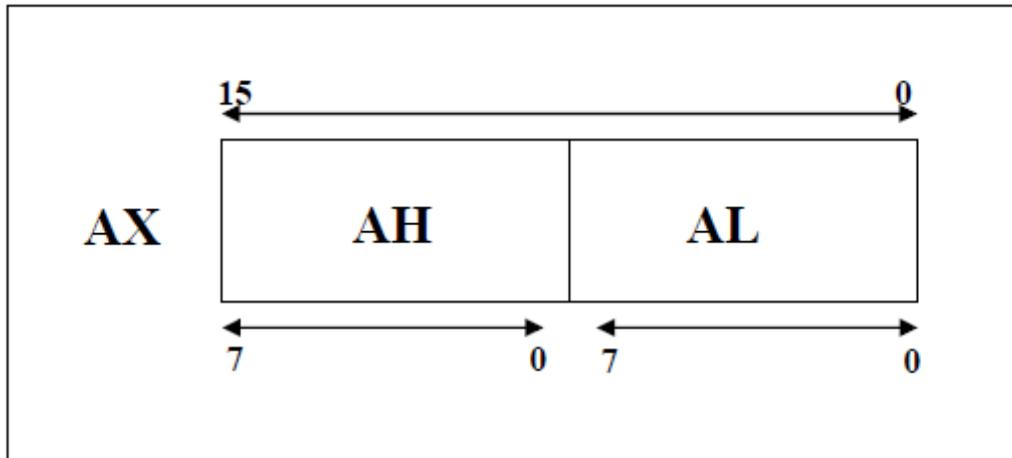


Figure 9: AX register

Instructions can address either 16-bit data register as **AX**, **BX**, **CX**, and **DX** or 8-bit register as **AL**, **AH**, **BL**, **BH**, **CL**, **CH**, **DI**, and **DH**. If we move **126FH** to **AX** then **AL** would immediately **6FH** and **AH** = **12H**.

\* Each **general purpose register** has special attributes:

- 1- **AX (Accumulator):** AX is the accumulator register because it is favored by the CPU for arithmetic operations. Other operations are also slightly more efficient when performed using **AX**.
- 2- **BX (Base):** the BX register can hold the address of a procedure or variable. Three other registers with this ability are **SI**, **DI** and **BP**. The **BX** register can also perform arithmetic and data movement.
- 3- **CX (Counter):** the CX register acts as a counter for repeating or looping instructions. These instructions automatically repeat and decrement **CX**.
- 4- **DX (Data):** the DX register has a special role in multiply and divide operation. When multiplying for example **DX** hold the high 16 bit of the product.

\* **Segment Registers:** the CPU contain four segment registers, used as base location for program instruction, and for the stack.

- 1- **CS (Code Segment):** The code segment register holds the base location of all executable instructions (code) in a program.
- 2- **DS (Data Segment):** the data segment register is the default base location for variables. The CPU calculates their location using the segment value in DS.
- 3- **SS (Stack Segment):** the stack segment register contain the base location of the stack.
- 4- **ES (Extra Segment):** The extra segment register is an additional base location for memory variables.

\* **Index registers:** index registers contain the offset of data and instructions. The term offset refers to the distance of a variable, label, or instruction from its base segment. The index registers are:

- 1- **BP (Base Pointer):** the **BP** register contain an assumed offset from the stack segment register, as does the stack pointer. The base pointer register is often used by a 23 subroutine to locate variables that were passed on the stack by a calling program.
- 2- **SP (Stack Pointer):** the stack pointer register contain the offset of the top of the stack. The stack pointer and the stack segment register combine to form the complete address of the top of the stack.
- 3- **SI (Source Index):** This register takes its name from the string movement instruction, in which the source string is pointed to by the source index register.
- 4- **DI (Destination Index):** the **DI** register acts as the destination for string movement instruction.

## **Status and Control register:**

1- **IP (Instruction Pointer):** The instruction pointer register always contain the offset of the next instruction to be executed within the current code segment. The instruction pointer and the **code segment** register combine to form the complete address of the next instruction.

2- **The Flag Register:** is a special register with individual bit positions assigned to show the status of the CPU or the result of arithmetic operations. The Figure 10 describes the **8086/8088** flags register:

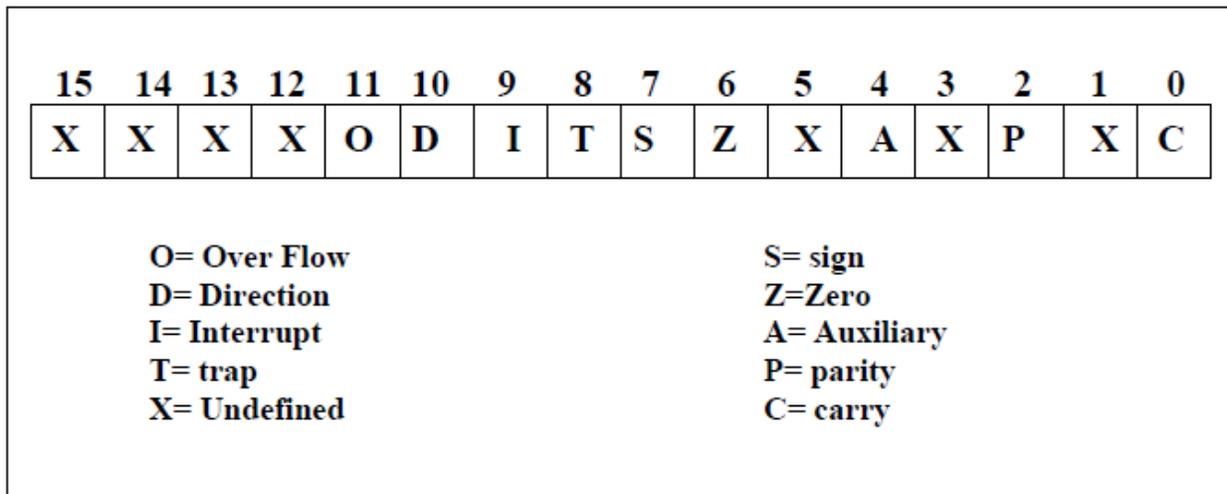


Figure 10: Flag Register.

**There two basic types of flags: (control flags and status flags)**

- 1- **Control Flags:** individual bits can be set in the flag register by the programmer to control the CPU operation, these are - The **Direction Flag (DF):** affects block data transfer instructions, such as **MOVS, CMPS, and SCAS**. The flag values are **0 = up** and **1 = down**.
  - The **Interrupt flag (IF):** dictates whether or not a system interrupt can occur. Such as keyboard, disk drive, and the system clock timer. A program will sometimes briefly disable the interrupt when performing a critical operation that cannot be interrupted. The flag values are **1 = enable**, **0 = disable**.
  - The **Trap flag (TF):** Determine whether or not the CPU is halted after each instruction. When this is set, a debugging program can let a programmer to enter single stepping (trace) through a program one instruction at a time.

The flag values are **1 = on**, **0 = off**. The flag can be set by **INT 3** instruction.

- 2- **Status Flags**: The status flags reflect the outcomes of arithmetic and logical operations performed by the CPU, these are:
- The **Carry Flag (CF)**: is set when the result of an unsigned arithmetic operation is too large to fit into the destination for example, if the sum of 71 and 99 were stored in the 8-bit register AL, the result causes the carry flag to be 1. The flag values = **1 = carry**, **0 = no carry**.
  - The **Overflow (OF)**: is set when the result of a signed arithmetic operation is too wide (too many bits) to fit into destination. **1 = overflow**, **0 = no overflow**.
  - **Sign Flag (SF)**: is set when the result of arithmetic or logical operation generates a negative result, **1 = negative**, **0 = positive**.
  - **Zero Flag (ZF)**: is set when the result of an arithmetic or logical operation generates a result of zero, the flag is used primarily by jump or loop instructions to allow branching to a new location in a program based on the comparison of two values. The flag value = **1 = zero**, & **0 = not zero**.
  - **Auxiliary Flag**: is set when an operation causes a carry from bit 3 to bit 4 (or borrow from bit 4 to bit 3) of an operand. The flag value = **1 = carry**, **0 = no carry**.
  - **Parity Flag**: reflects the number of 1 bits in the result of an operation. If there is an **even number** of bits, the parity is **even**. If there is an **odd number** of bits, parity is **odd**. This flag is used by the OS to verify memory integrity and by communication software to verify the correct transmission of data.

## Instruction Execution and Addressing

An assembly language programmer writes a program in **symbolic code** and uses the **assembler** to translate it into **machine code** as .EXE program. For program execution, the system looks only at the machine code in memory.

Every instruction consists of at least one **operation**, such as MOV, ADD. Depending on the operation, an instruction may also have one or more **operands** that reference the data the operation is to process.

The basic steps the processor takes in executing an instruction are:

1. **Fetch the next instruction** to be executed from memory and place it in the instruction queue.
2. **Decode the instruction** calculates addresses that reference memory, deliver data to the Arithmetic Logic Unit, and increment the instruction pointer (**IP**) register.
3. **Execute the instruction**, performs the requested operation, store the result in a register or memory, and set flags such as zero or carry where required.

For an .EXE program the **CS** register provides the address of the beginning of a program code segment, and **DS** provide the address of the beginning of the data segment.

The **CS** contains instructions that are to be executed, whereas the **DS** contain data that the instruction references. The **IP** register indicates the offset address of the current instruction in the **CS** that is to be executed. An instruction operand indicates an offset address in the **DS** to be referenced.

Consider an example in which the program loader has determined that it is to be loaded on .EXE program into memory beginning at location **05BE0<sub>H</sub>**. The loader accordingly initializes **CS** with segment address **05BE[0]<sub>H</sub>** and **IP** with zero.

**CS: IP** together determine the address of the first instruction to execute **05BE0<sub>H</sub> + 0000<sub>H</sub> = 05BE0<sub>H</sub>**. In this way the first instruction in **CS** being execution, if the first instruction is two bytes long, the processor increments **IP** by **2**, so that the next instruction to be executed is **05BE0<sub>H</sub> + 0002<sub>H</sub> = 05BE2<sub>H</sub>**.

Assume the program continues executing, and **IP** contains the offset **0023<sub>H</sub>**. **CS: IP** now determine the address of the next instruction to execute, as follows:

$$\begin{array}{r} \text{CS address:} \quad 05BE0_H \\ \text{IP offset:} \quad 0023_H \\ \hline \end{array} +$$

**Instruction address: 05C03<sub>H</sub>**

**EX:** let's say that **MOV** instruction beginning at **0FC03<sub>H</sub>** copies the content of a byte in memory into the **AL** register. The byte is at offset **0016<sub>H</sub>** in the **DS**. Here are the machine code and the symbolic code for this operation.

Address	Symbolic Code	MIC code
<b>0FC03</b>	<b>MOV AL, [0016]</b>	<b>A0 1600</b>

Address **0FC03<sub>H</sub>** contain the first byte (**A0<sub>H</sub>**) of the MIC code instruction the processor is to access

	00000 <sub>H</sub>
	.
	.
<b>A0</b>	<b>0FC03<sub>H</sub></b>
<b>00</b>	<b>0FC04<sub>H</sub></b>
<b>16</b>	<b>0FC05<sub>H</sub></b>
	.
	.
	<b>FFFFF<sub>H</sub></b>

The second and third byte contains the offset value in reversed byte sequence. In symbolic code, the operand **[0016]** in square brackets (an index operator) indicates an offset value to distinguish it from the actual storage address 16.

Let say that the program has initialized the **DS** register with **DS** address **05D1[0]<sub>H</sub>**. To access the data item, the processor determines its location from the segment address in **DS** + the offset (**0016<sub>H</sub>**) in the instruction. Operand become **DS** contain **0FD1[0]<sub>H</sub>**, the actual location of the reference data item is

$$\begin{array}{r}
 \text{DS:} \\
 \text{Offset:} \\
 \text{Address of data item:}
 \end{array}
 \begin{array}{r}
 05D10_{\text{H}} \\
 0016_{\text{H}} \\
 \hline
 05D26_{\text{H}}
 \end{array}
 +$$

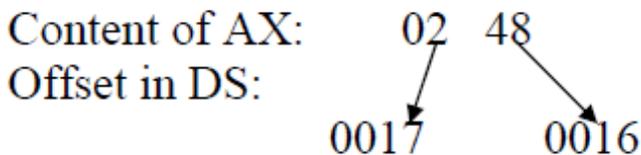
Assume the address **05D26<sub>H</sub>** contain **4A<sub>H</sub>**, the processor now extract the **4A<sub>H</sub>** at address **05D26<sub>H</sub>** and copy it into **AL** register.

An instruction may also access more than one byte at a time

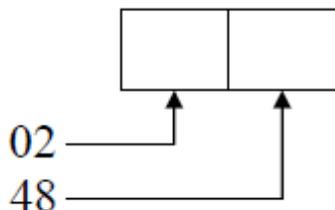
**EX:** Suppose an instruction is to store the content of the **AX** register (**0248<sub>H</sub>**) in two adjacent byte in the **DS** beginning at offset **0016<sub>H</sub>**.

The symbolic code **MOV [0016], AX**

The processor stores the two byte in memory in reversed byte sequence as



Another instruction, **MOV AX, [0016]**, subsequently could retrieve this byte by copy them from memory back into **AX**. The operation reverses (and corrects) the byte in **AX** as:



## Number of Operands

Operands specify the value an instruction is to operate on, and where the result is to be stored. Instruction sets are classified by the number of operands used. An instruction may have no, one, two, or three operands.

### 1. Three-Operand instruction:

In instruction that have three operands, one of the operand specifies the destination as an address where the result is to be saved. The other two operands specify the source either as addresses of memory location or constants.

**EX:**            **A=B+C**  
                  **ADD destination, source1, source2**  
                  **ADD A,B,C**

**EX:**            **Y=(X+D)\* (N+1)**

**ADD T1, X, D**  
                  **ADD T2, N, 1**  
                  **Mul Y, T1, T2**

### 2. Two operand instruction

In this type both operands specify sources. The first operand also specifies the destination address after the result is to be saved. The first operand must be an address in memory, but the second may be an address or a constant.

**ADD destination, source**

**EX: A=B+C**  
                  **MOV A, B**  
                  **ADD A, C**

**EX: Y=(X+D)\* (N+1)**

**MOV T1, X**  
                  **ADD T1, D**  
                  **MOV Y, N**  
                  **ADD Y, 1**  
                  **MUL Y, T1**

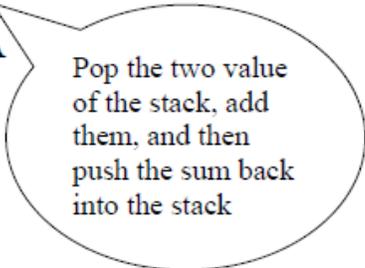
### 3. One Operand instruction

Some computer have only one general purpose register, usually called on Acc. It is implied as one of the source operands and the destination operand in memory instruction the other source operand is specified in the instruction as location in memory.



**EX:**  
**A=B+C**

PUSH B  
PUSH C  
ADD;  
POP A



**EX:**  
**Y=(X+D)\* (N+1)**

PUSH X  
PUSH D  
ADD  
PUSH N  
PUSH 1  
ADD  
MUL  
POP Y

## Assembly Language Instruction

Assembly language instructions are provided to describe each of the basic operations that can be performed by a microprocessor. They are written using **alphanumeric symbols** instead of the 0s and

1s of the microprocessor's machine code. Program written in assembly language are called **source code**. An assembly language description of this instruction is

**ADD AX, BX**

In this example, the contents of BX and AX are added together and their sum is put in AX. Therefore, BX is considered to be the **source operand** and AX the **destination operand**.

Here is another example of an assembly language statement:

**LOOP:     MOV AX, BX ; COPY BX INTO AX**

This instruction statement starts with the word LOOP. It is an address identifier for the instruction MOV AX, BX. This type of identifier is called a **label** or **tag**. The instruction is followed by "COPY BX INTO AX." This part of the statement is called a **comment**. Thus a general format for writing an assembly language statement is:

**LABEL:     INSTRUCTION     ; COMMENT**

## The 8086 Addressing Mode

When the 8086 executes an instruction, it performs the specified function on data. The data are called its **operands** and may be part of the instruction, reside in one of the internal registers of the 8086, stored at an address in memory, or held at an I/O port. To access these different types of operands, the 8086 is provided with

Various addressing modes:

## 1. Register Addressing Mode

With the register addressing mode, the operand to be accessed is specified as residing in an internal register of the 8086, an example of an instruction that uses this addressing mode is

**MOV AX, BX**

This stands for move the contents of BX, the source operand, to AX, the destination operand. Both the source and destination operands have been specified as the content of the internal registers of the 8086. See Figure 11(a, b).

## 2. Immediate Addressing Mode

If a source operand is part of the instruction instead of the contents of a register or memory location, it represents what is called an immediate operand and is accessed using the immediate addressing mode. Typically, immediate operands represent constant data. Immediate operands can be either a byte or word of data. In the Instruction

**MOV AL, 015<sub>H</sub>**

The source operand 15H is an example of a byte-wide immediate source operand. Note that the value of the immediate operand must always be preceded by a zero. See Figure 12(a, b).

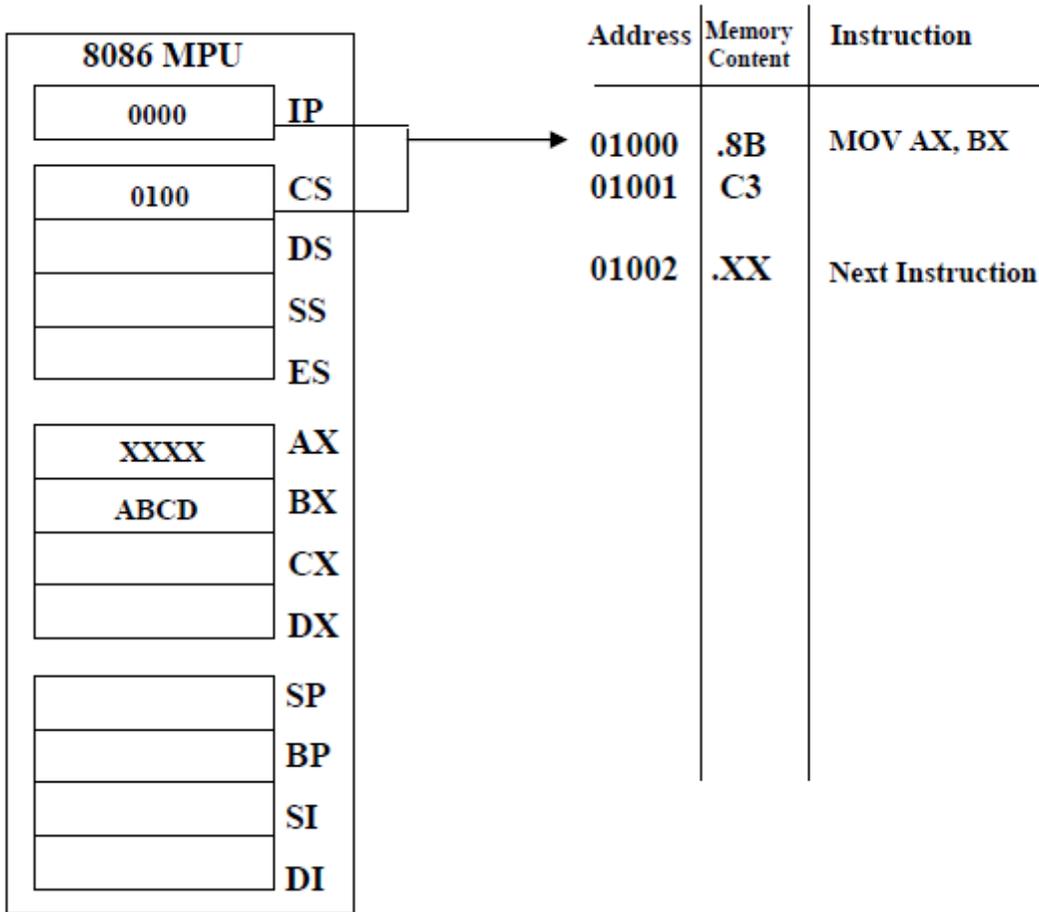


Figure 11(a): Register addressing mode before execution.

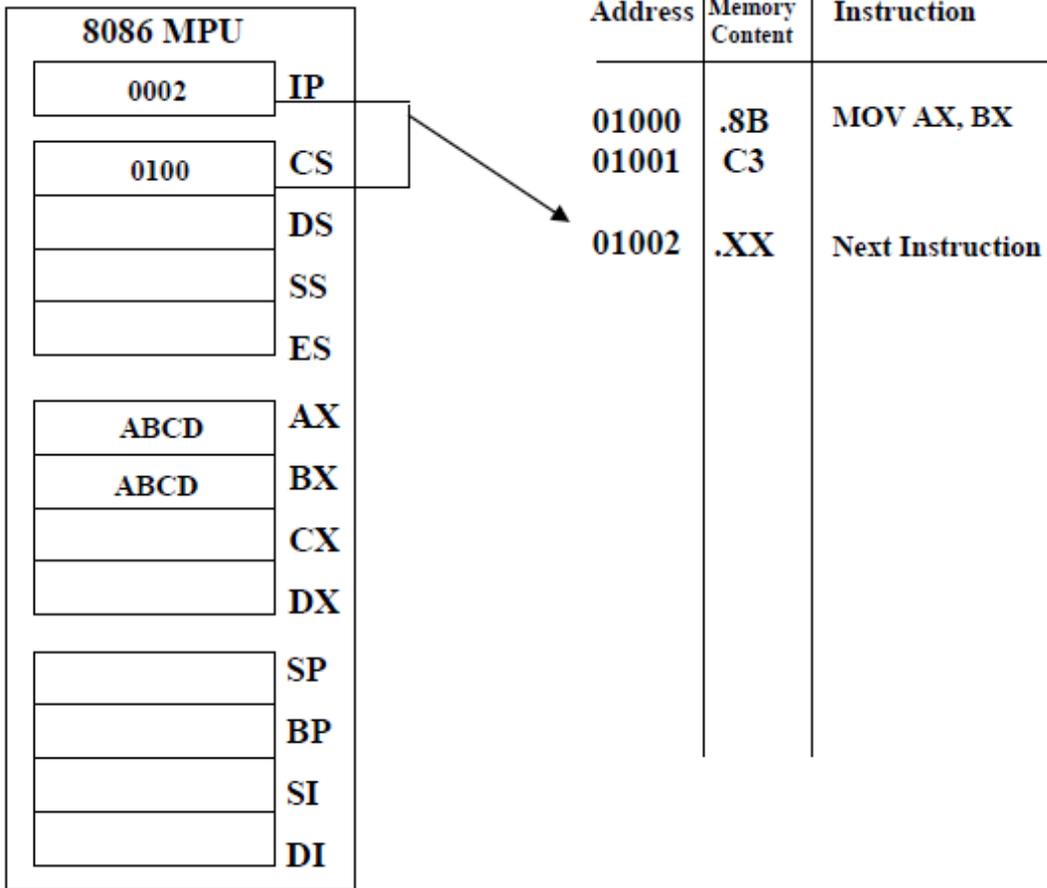


Figure 11(b): Register addressing mode after execution.

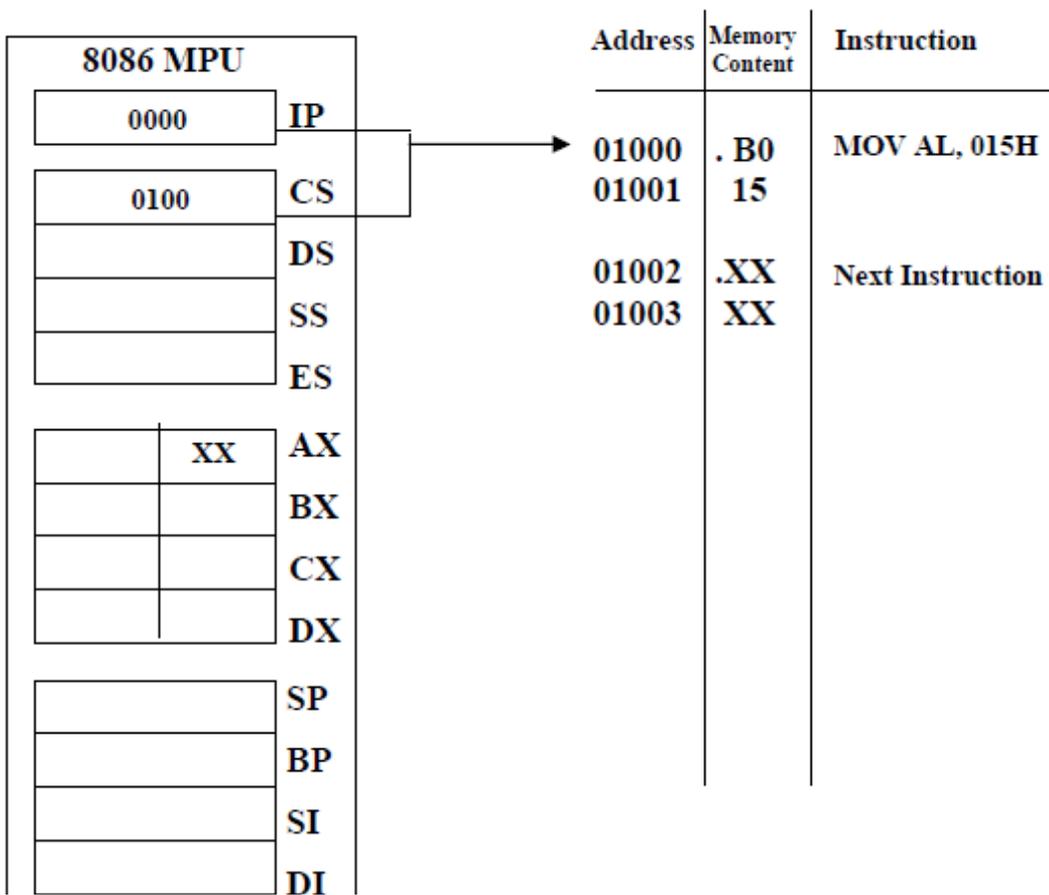


Figure 12(a): Immediate addressing mode before execution.

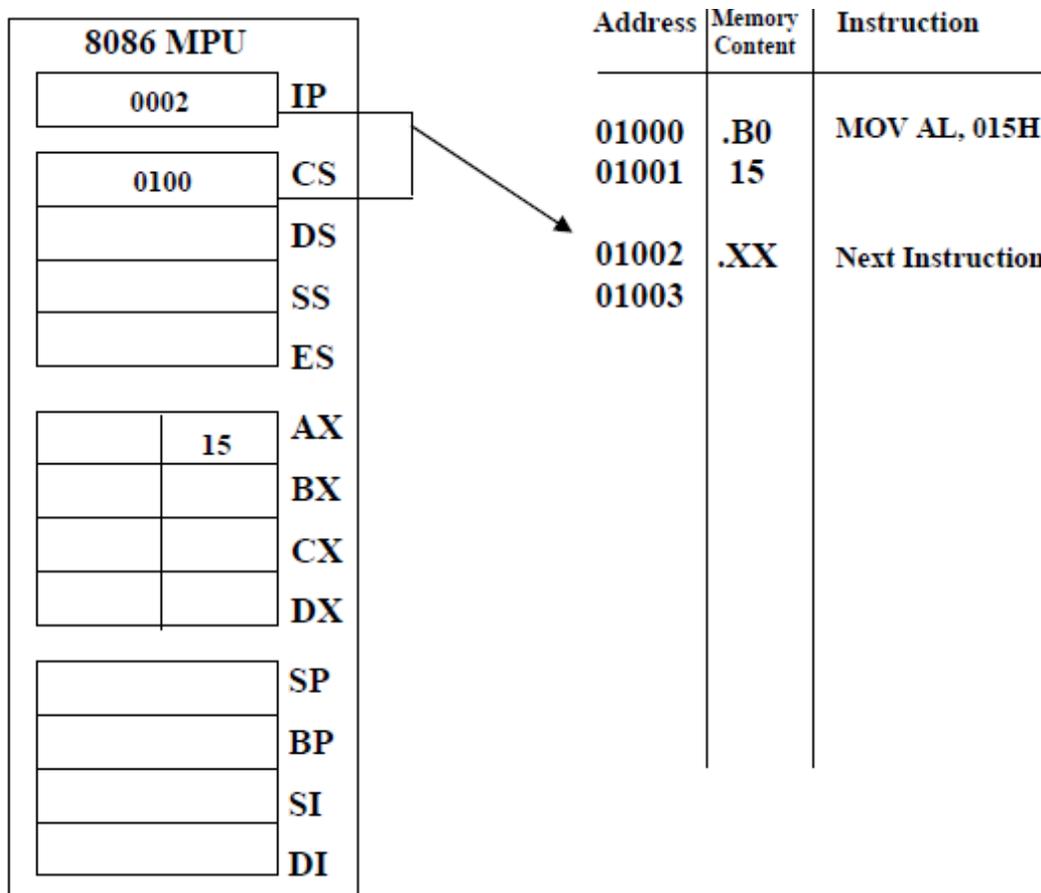


Figure 12(b): Immediate addressing mode after execution.

### 3. Direct Addressing Mode

Direct addressing differs from immediate addressing in that the locations following the instruction opcode hold an **effected memory address (EA)** instead of data. This effective address is a 16-bit offset of the storage location of the operand from the current value in the data segment (DS) register. EA is combined with the contents of DS in the BIU to produce the **physical address** for its source operand is

**MOV CX, BETA**

This stands for move the contents of the memory location which is offset by BETA from the current value in DS into internal register CX. See Figure 13(a, b). Notice that the value assigned to constant BETA is 1234<sub>H</sub>.

$$\begin{aligned} \text{PA} &= \text{02000}_H + \text{1234}_H \\ &= \text{03234}_H \end{aligned}$$

#### 4. Register Indirect Addressing Mode

Register indirect addressing is similar to direct addressing in that an effective address is combined with the contents of DS to obtain a physical address. However, it differs in the way the offset is specified. This time EA resides in either a pointer register or index register within the 8086. The pointer register can be either BX or BP and the index register can be SI or DI.

**MOV AX, [SI]**

This instruction moves the contents of the memory location offset by the value of EA in SI from the current value in DS to the AX register. See Figure 14(a, b). SI contains 1234<sub>H</sub> and DS contains 0200<sub>H</sub>.

$$\begin{aligned} \text{PA} &= \text{02000}_H + \text{1234}_H \\ &= \text{03234}_H \end{aligned}$$

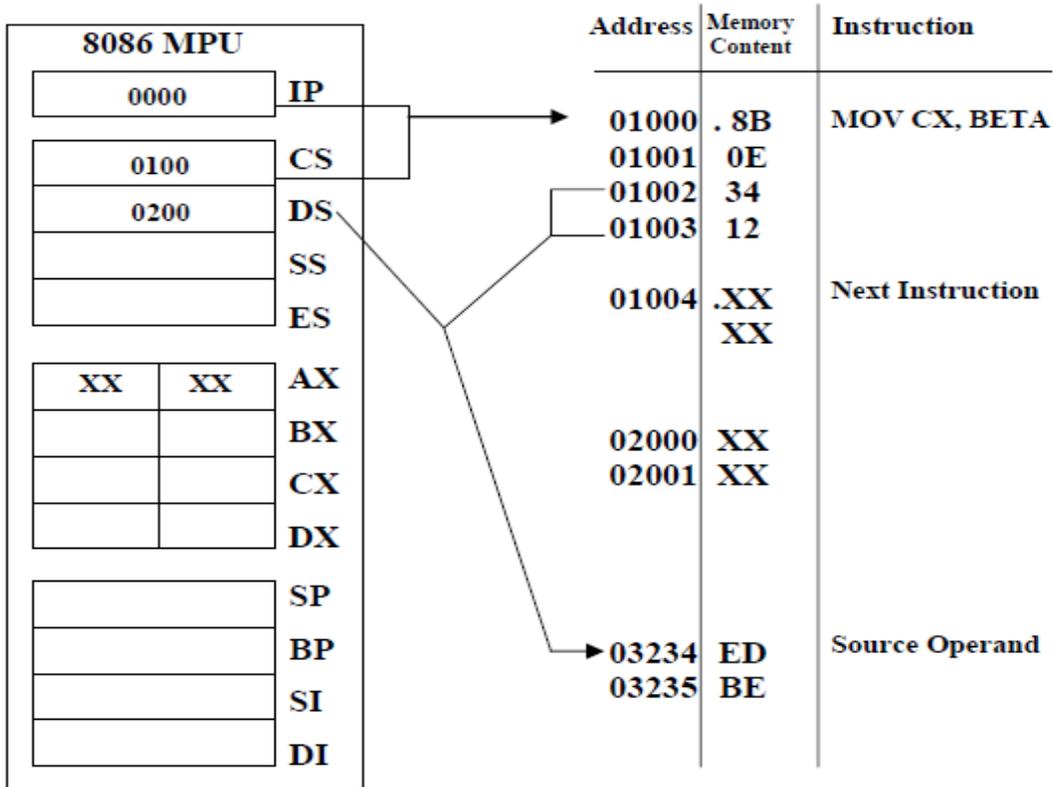


Figure 13(a): Direct Addressing mode before execution.

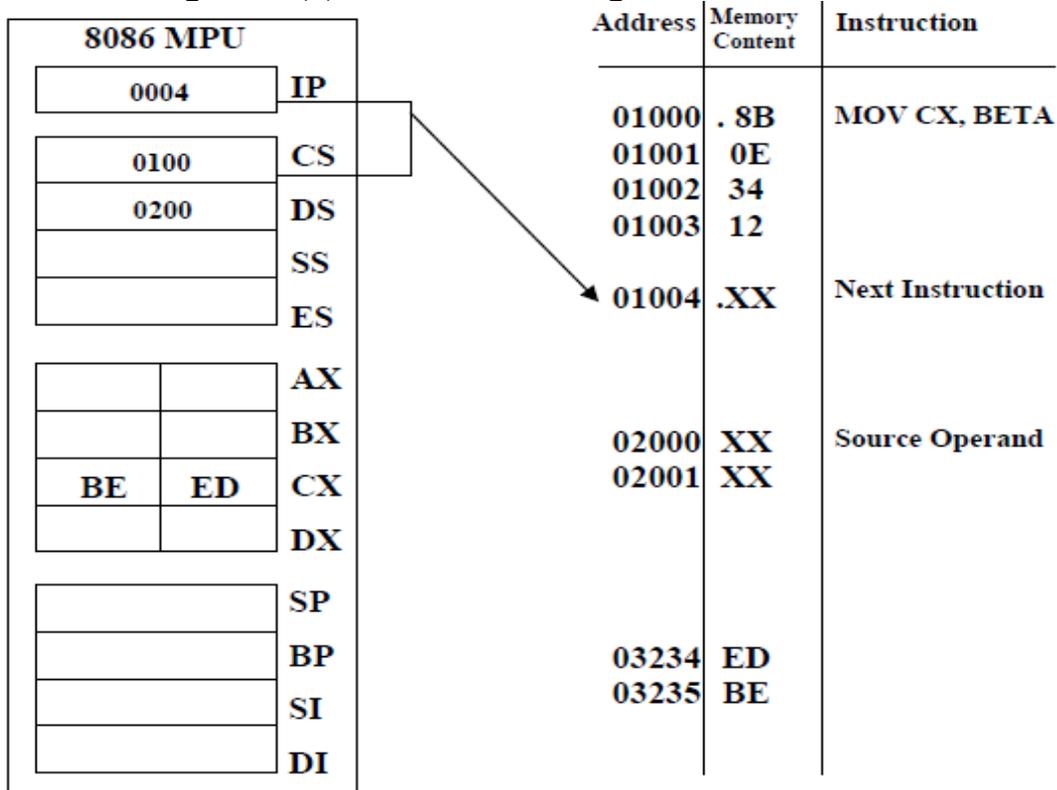


Figure 13(b): Direct Addressing mode after execution.

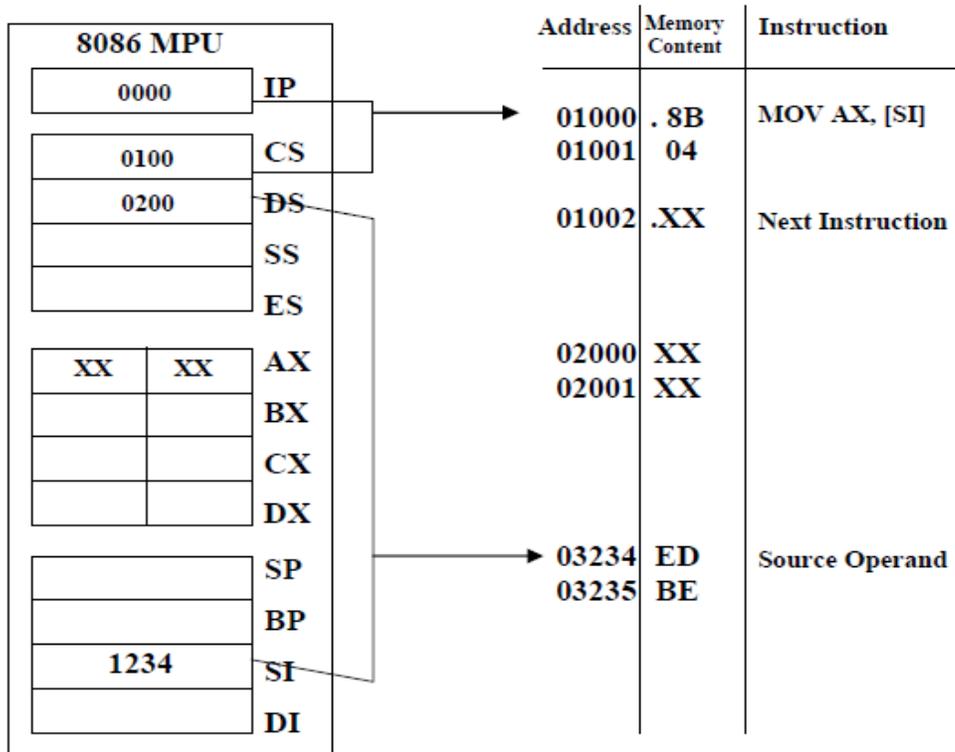


Figure 14(a): Register Indirect Addressing before execution.

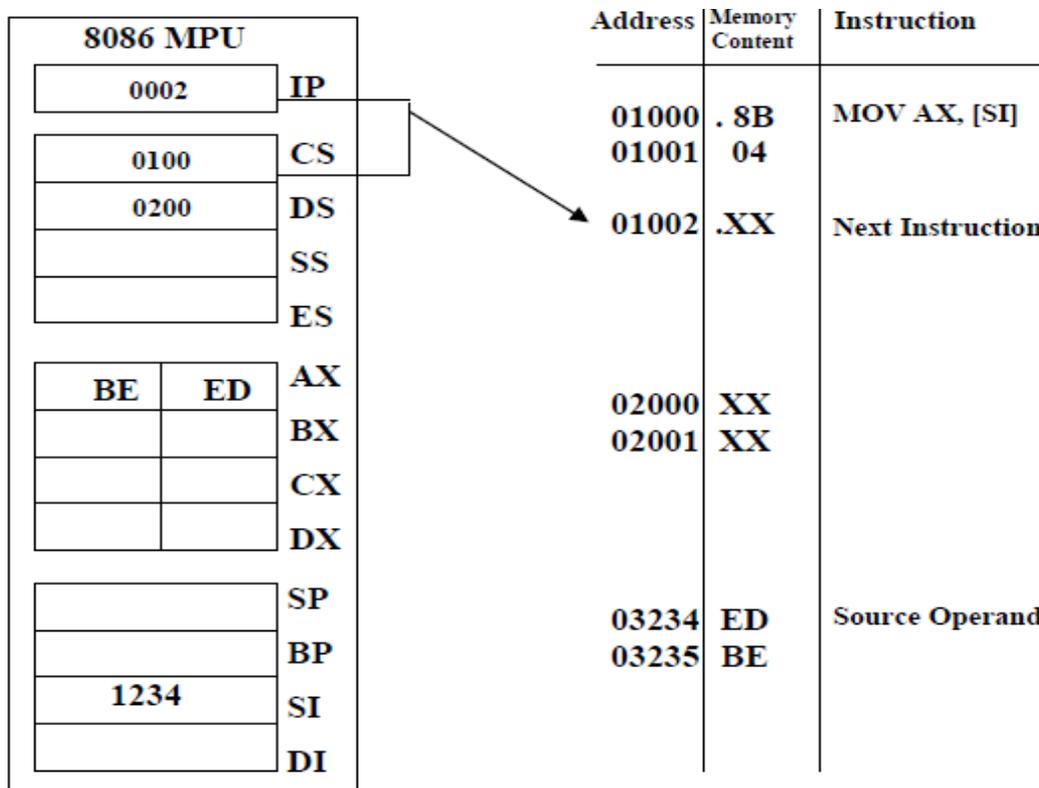


Figure 14(b): Register Indirect Addressing mode after execution.

## 5. Based Addressing Mode

In the based addressing mode, the physical address of the operand is obtained by adding a direct or indirect displacement to the contents of either BX or BP and the current value in DS and SS, respectively. A MOV instruction that uses based addressing to specify the location of its destination operand is as follows:

**MOV [BX].BETA, AL**

As shown in Figure 15(a,b) the fetch and execution of this instruction causes the BIU to calculate the physical address of the destination operand from the contents of DS, BX, and the direct displacement. The result is

$$\begin{aligned}
 \text{PA} &= 02000_{\text{H}} + 1000_{\text{H}} + 1234_{\text{H}} \\
 &= 04234_{\text{H}}
 \end{aligned}$$

## 6. Indexed Addressing Mode

Indexed addressing works identically to the based addressing, it uses the contents of one of the index registers, instead of BX or BP, in the generation of the physical address, here is an example:

**MOV AL, ARRAY[SI]**

The example in Figure 16(a,b) shows the result of executing the MOV instruction. First the physical address for the source operand is calculated from DS, SI, and the direct displacement.

$$\begin{aligned} \text{PA} &= 02000_{\text{H}} + 2000_{\text{H}} + 1234_{\text{H}} \\ &= 05234_{\text{H}} \end{aligned}$$

Then the byte of data stored at this location, which is BEH is read into lower byte AL of the accumulator register.

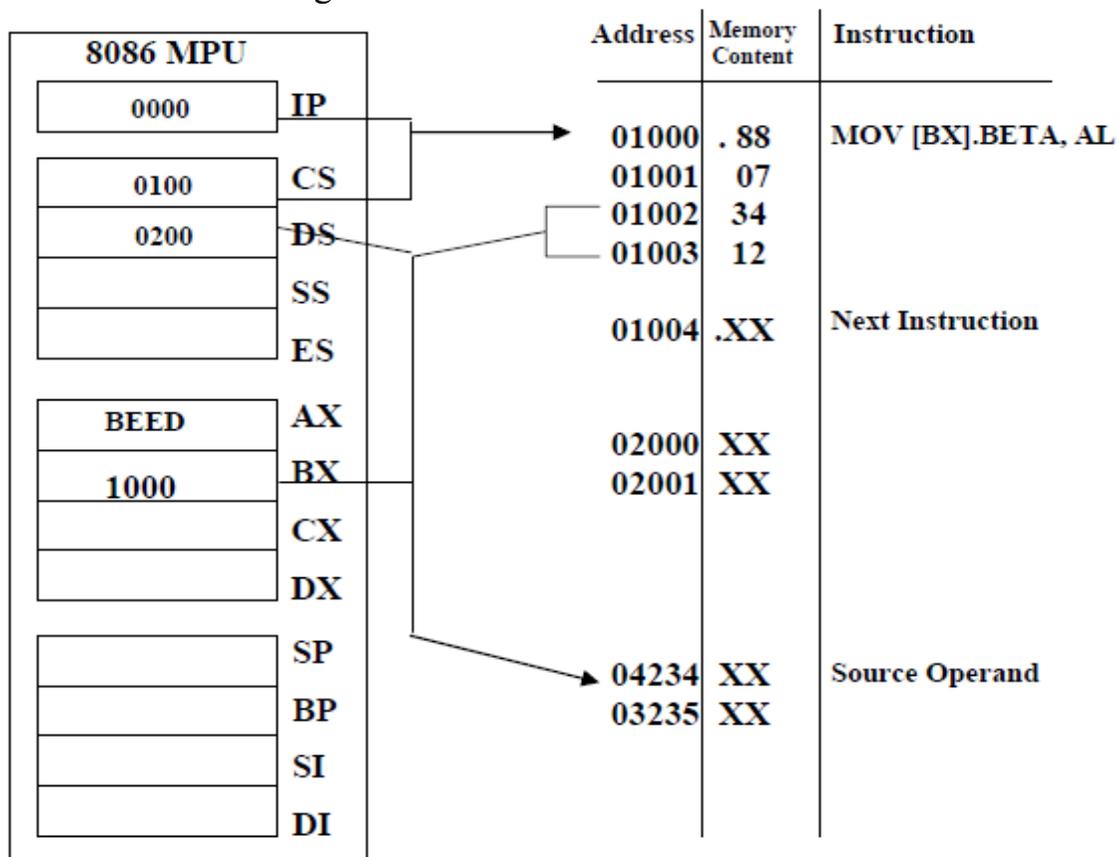


Figure 15(a): Based Addressing before execution.

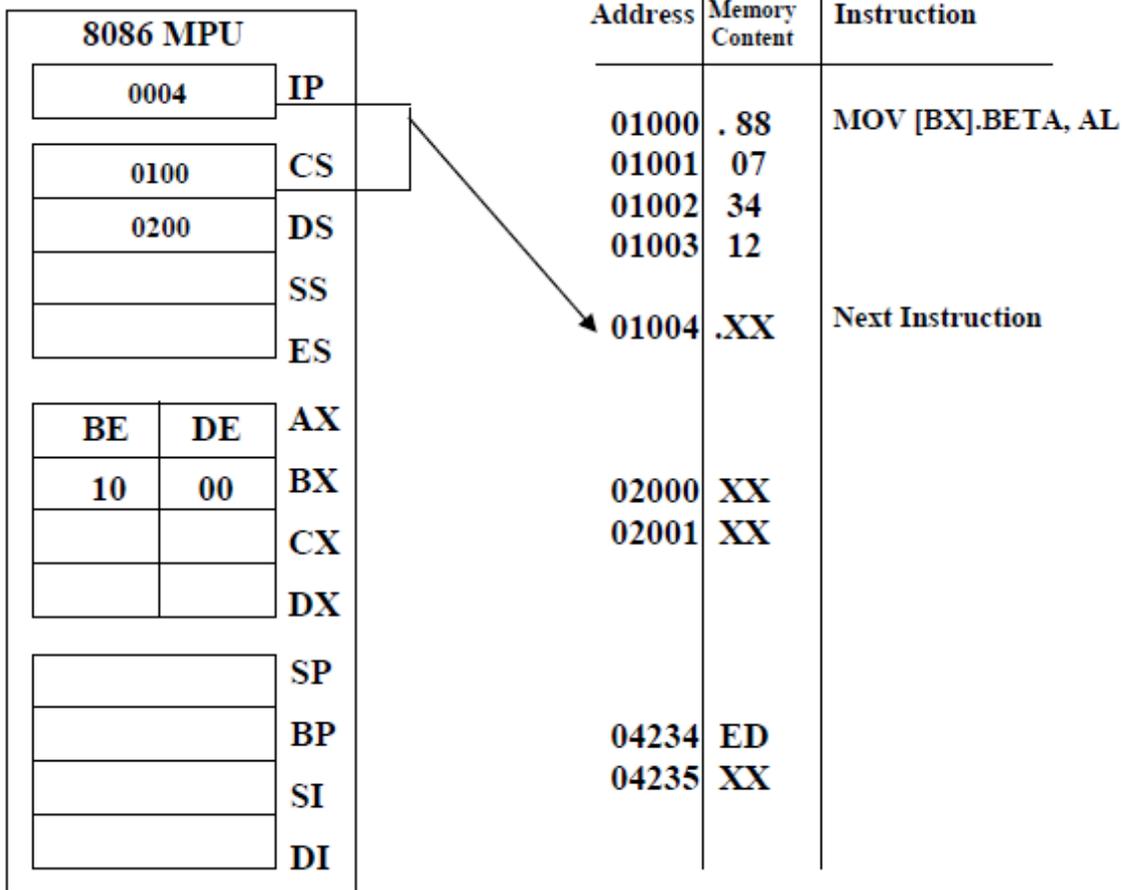


Figure 15(b): Based Addressing mode after execution.

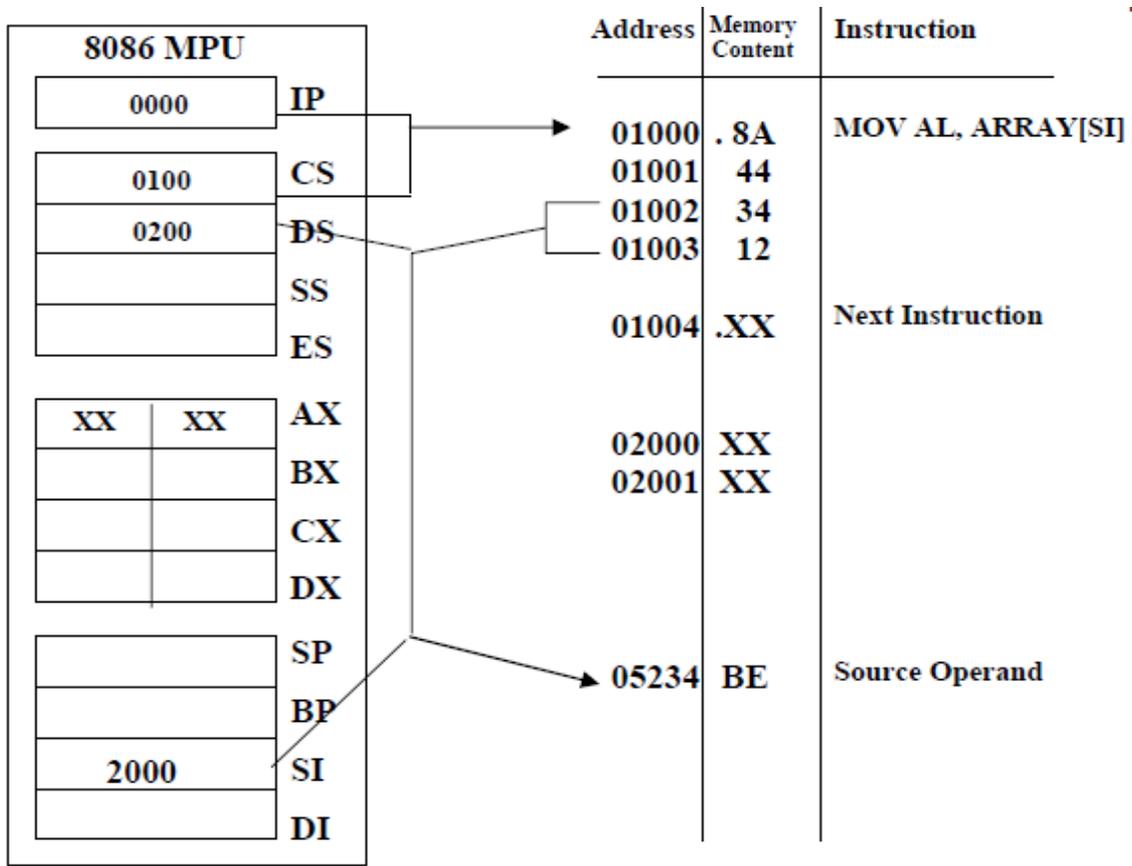


Figure 16(a): Direct Indexed Addressing before execution.

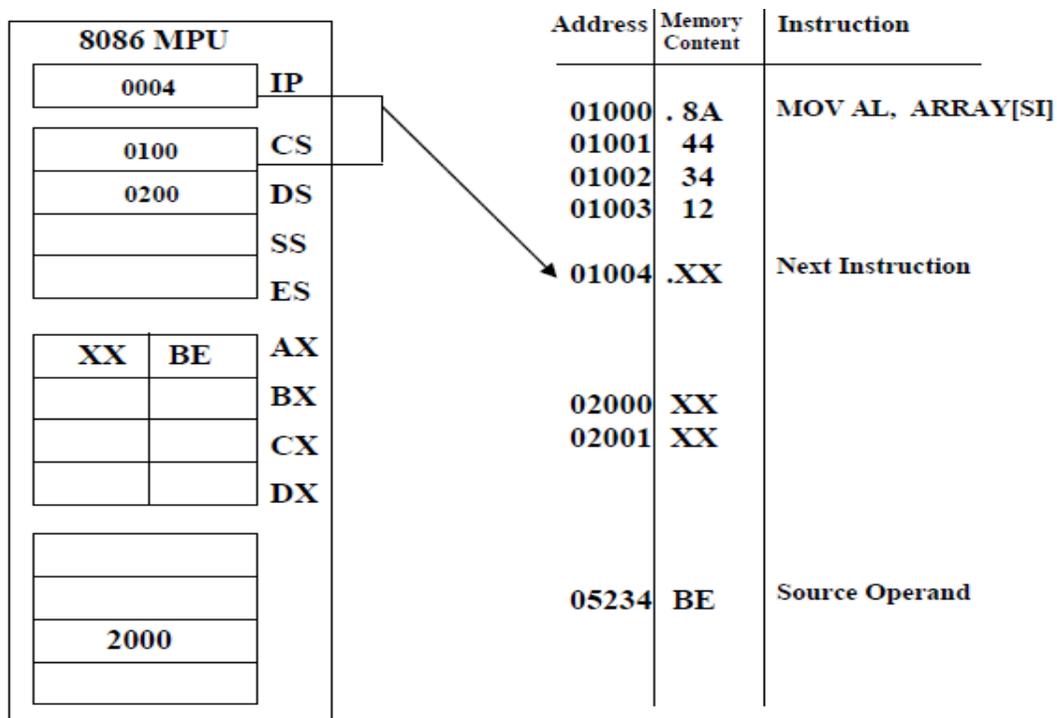


Figure 16(b): Direct Indexed Addressing mode after execution.

## 7. Based Indexed Addressing Mode

Combining the based addressing mode and the indexed addressing mode together results in a new, more powerful mode known as based indexed addressing. Let us consider an example of a MOV instruction using this type of addressing.

**MOV AH, [BX].BETA[SI]**

An example of executing this instruction is illustrated in Figure 17(a,b). The address of the source operand is calculated as

$$\begin{aligned}
 PA &= 02000_H + 1000_H + 1234_H + 2000_H \\
 &= 06234_H
 \end{aligned}$$

Execution of this instruction causes the Value stored at this location to be written into AH.

## 8. String Addressing Mode

The string instructions of the 8086's instruction set automatically use the source and destination index registers to specify the effective addresses of the source and destination operands, respectively. The move string instruction

### MOVS

is an example. Notice that neither SI nor DI appears in the string instruction, but both are used during its execution.

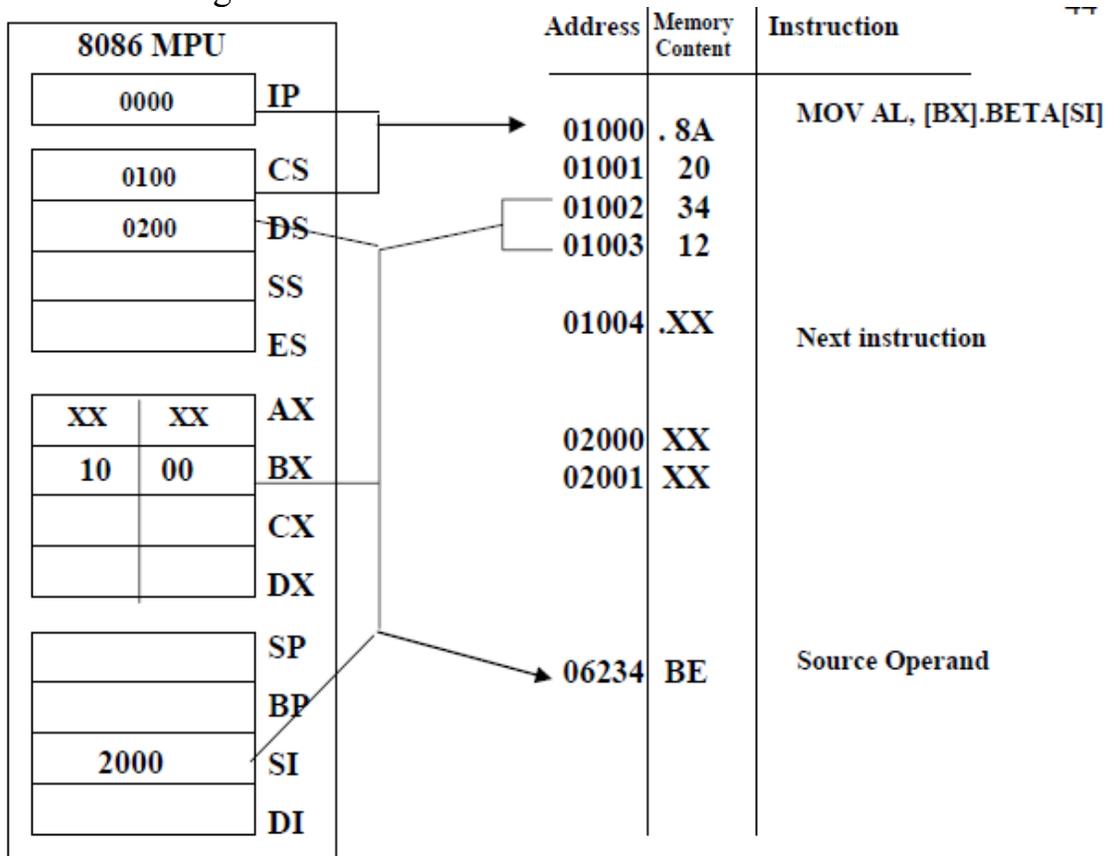


Figure 17(a): Based Indexed Addressing before execution.

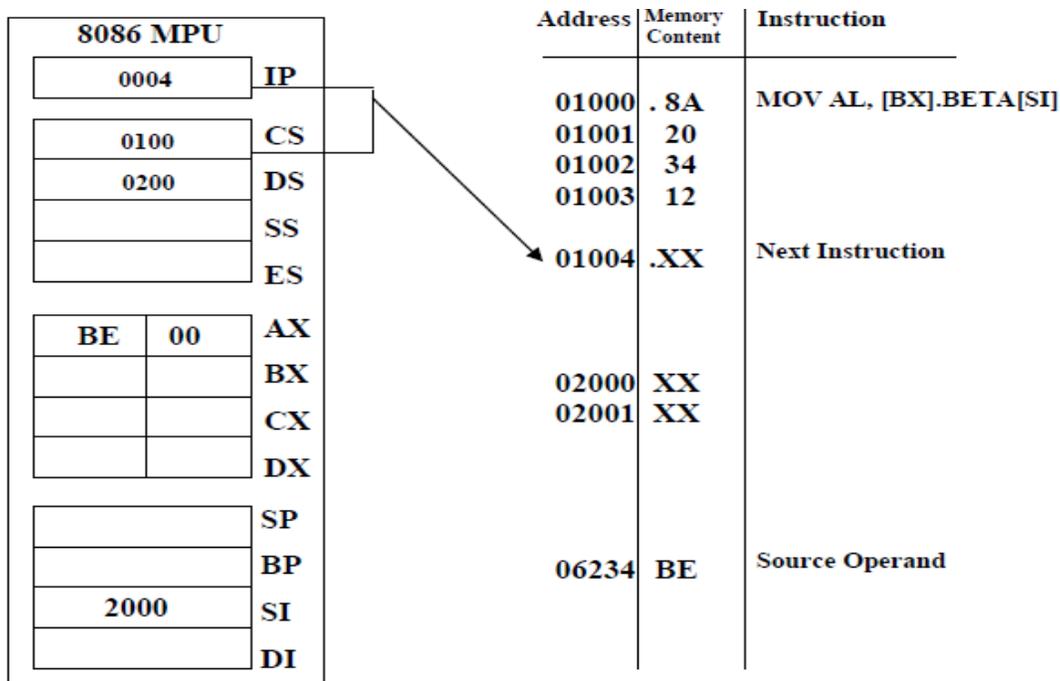


Figure 17(b): Based Indexed Addressing mode after execution.

## 9. Port Addressing Mode

Port addressing is used in conjunction with the IN and OUT instructions to access input and output ports. Any of the memory addressing modes can be used for the port address for memory mapped ports. For ports in the I/O address space, only the **Direct addressing mode** and an **Indirect addressing mode** using DX are available. For example, **Direct addressing** of an input port is used in the instruction

**IN AL, 15<sub>H</sub>**

This stands for input the data from the byte wide input port at address 15<sub>H</sub> of the I/O address space to register AL. Next, let us consider another example. Using **Indirect port addressing** for the source operand in an IN instruction, we get:

**IN AL, DX**

It means input the data from the byte wide input port whose address is specified by the contents of register DX. For instance, if DX equals 1234<sub>H</sub> the contents of the port at this I/O address are loaded into AL.

## **Problems:**

1. Which register holds a count for some instruction?
2. What is the purpose of the IP register?
3. The carry flag bit is set by which arithmetic operation?
4. A number that contain 3 one bit said to have----- parity?
5. Find the memory address of the next instruction execute by the micro processor for the following **CS:IP** combinations:

- a. CS=1000<sub>H</sub> and IP=2000<sub>H</sub>
- b. CS=2000<sub>H</sub> and IP=1000<sub>H</sub>

6. Which register or registers are used as an offset address for string instruction destination in the microprocessor?
7. The stack memory is addressed by a combination of the----- segment plus ----- offset.
8. Which registers of the 8086 are used in memory segmentation?
9. Categorize each flag bit of the 8086 as either a control flag or as a flag to monitor the effect of instruction execution.
10. Identify the three part of an assembly language instruction in each of the following statement:

**AGAIN: ADD AX, CX; ADD THE REGISTERS  
MOV BX, AX; SAVE RESULT**

11. Identify the source and destination operand for each of the statements in 10.

## **Instructions set**

8086 has 117 instructions, these instructions divided into 6 groups:

1. Data transfer instructions
2. Arithmetic instructions
3. Logic instructions
4. Shift instructions
5. Rotate instructions
6. Advance instructions

## **1. Data Transfer Instructions**

The microprocessor has a group of data transfer instructions that are provided to move data either between its internal registers or between an internal register and a storage location in memory. Some of these instructions are:

♣ **MOV** use to transfer a byte or a word of data from a source operand to a destination operand. These operands can be internal registers and storage locations in memory. Notice that the MOV instruction cannot transfer data directly between a source and a destination that both reside in external memory. For instance, flag bits within the microprocessors are not modified by execution of a MOV instruction.

**EXAMPLES:**

1. MOV DX, CS            where DX=0100<sub>H</sub>  
    DX=CS=0100<sub>H</sub>

2. MOV SUM, AX            DS=0200<sub>H</sub>            SUM=1212<sub>H</sub>  
    PA=02000<sub>H</sub>+1212<sub>H</sub> = 03212<sub>H</sub>

AL →Memory location 03212<sub>H</sub>  
 AH →Memory location 03213<sub>H</sub>

4. If DS contain 1234H what is the effect of executing the instruction MOV CX,[0ABCD<sub>H</sub>]

CL loaded with the content of Memory location 1234<sub>H</sub> + ABCD<sub>H</sub> = 1CF0D<sub>H</sub>

And CH is loaded with the content of Memory location 1234<sub>H</sub> + ABCD<sub>H</sub> +1 = 1CF0E<sub>H</sub>

♣ **XCHG**: in MOV instruction the original contents of the source location are preserved and the original contents of the destination are destroyed. But **XCHG** instruction can be used to swap data between two general purpose register or between a general purpose register and storage location in memory.

**EXAMPLES:**

1. XCHG AX, DX            (AX) ↔ (DX)

## 2. XCHG SUM, BX

(DS (0) + SUM)  $\longleftrightarrow$  BX  
DS=02000<sub>H</sub> + 1234<sub>H</sub> = 03234<sub>H</sub>  
(3234)  $\longleftrightarrow$  (BL)  
(3235)  $\longleftrightarrow$  (BH)

## 2. Arithmetic Instructions

Arithmetic instructions includes instructions for the addition, subtractions can be performed on numbers expressed in a variety of numeric data formats. The status that results from the execution of an arithmetic instruction is recoded in the flags of the microprocessor. The flags that are affected by arithmetic instructions are CF, AF, SF, ZF, and PF.

### ♣ Addition: ADD, ADC, and INC

- **ADD AX, BX**

AX = AX + BX

#### EXAMPLE:

AX = 1100<sub>H</sub>, BX = 0ABC<sub>H</sub> **ADD AX, BX**

1100<sub>H</sub> + 0ABC<sub>H</sub> = 1BBC<sub>H</sub> = AX

- **ADC AX, BX**

AX = AX + BX + CF

- **INC AH**

AH = AH + 1

#### EXAMPLE:

The original contents of AX, BL, memory location SUM, and CF are AX=1234<sub>H</sub>, BL= AB<sub>H</sub>, Sum=00CD<sub>H</sub> and CF=0 respectively, describe the result of execution the following sequence of instruction:

ADD AX, SUM  
 ADC BL, 05<sub>H</sub>  
 INC SUM

1. AX= 1234<sub>H</sub> + 00CD<sub>H</sub> = 4301<sub>H</sub>      CF=0
2. BL= AB<sub>H</sub> +05<sub>H</sub> +0=B0<sub>H</sub>              CF=0
3. SUM=00CD<sub>H</sub> + 1=00CE<sub>H</sub>              CF=0

Instructions	AX	BL	SUM	CF
Initial state	1234 <sub>H</sub>	AB <sub>H</sub>	00CD <sub>H</sub>	0
ADD AX, SUM	4301 <sub>H</sub>	AB <sub>H</sub>	00CD <sub>H</sub>	0
ADC BL, 05 <sub>H</sub>	4301 <sub>H</sub>	B0 <sub>H</sub>	00CD <sub>H</sub>	0
INC SUM	4301 <sub>H</sub>	B0 <sub>H</sub>	00CE <sub>H</sub>	0

♣ **Subtraction: SUB, SBB, DEC, and NEG**

- **SUB** AX, BX  
 AX=AX – BX

- **SBB** AX, BX  
 AX= AX - BX – CF

**EXAMPLE:**

BX=1234<sub>H</sub>, CX=0123<sub>H</sub>, CF=0

SBB BX, CX

BX=1234<sub>H</sub>-0123<sub>H</sub>-0 =1111<sub>H</sub>

- **DEC** subtract 1 from its operand
- **NEG** BX (2's complement)

00<sub>H</sub> – BX  
 0000 + 2's complement of BX

**EXAMPLE:**

**EXAMPLE:**

**BX = 3A<sub>H</sub> NEG BX**

0000<sub>H</sub> + FFC6<sub>H</sub>

0011 1010  
1100 0101 +  
          1  
-----  
1100 0110  
C    6



**♣ Multiplication and Division MUL, DIV**

- **MUL CL**

(AX) = AL \* CL

- **MUL CX**

(DX, AX) = AX \* CX

- **DIV CL**

(AH), (AL) = AX / CL

And AL the quotient

Where AH is the remainder

- **DIV CX**

DX, AX = (DX, AX) / CX

AX contain the Quotient

DX contain the remainder

**EXAMPLE:**

MUL CL where AL = -1 CL = -2

AX = FF<sub>H</sub> \* FE<sub>H</sub> = FD02<sub>H</sub>

**3. Logical Instructions (AND, OR, XOR, NOT)**

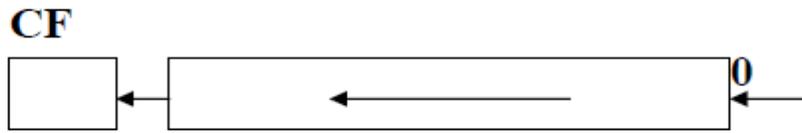
<b>Instructions</b>	<b>AL</b>
MOV AL, 0101 0101 <sub>B</sub>	0101 0101 <sub>B</sub>
AND AL, 0001 1111 <sub>B</sub>	0001 0101 <sub>B</sub>
OR AL, 1100 0000 <sub>B</sub>	1101 0101 <sub>B</sub>
XOR AL, 0000 1111 <sub>B</sub>	1101 1010 <sub>B</sub>
NOT AL	0010 0101 <sub>B</sub>

## 4. Shift Instructions

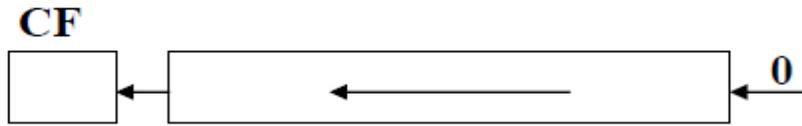
The four types of shift instructions can perform two basic types of shift operations. They are the logical shift and arithmetic shift. Each of these operations can be performed to the right or to the left.

<b>Instructions</b>	<b>Meaning</b>	<b>format</b>	<b>Operation</b>	<b>Flags affected</b>
<b>SAL/SHL</b>	Shift arithmetic left/shift logical left	SAL/SHL D, Count	Shift the D left by the number of bit positions equal to count and fill the vacated bits positions on the right with zeros	OF, CF
<b>SHR</b>	Shift logical right	SHR D, Count	Shift the D right by the number of bit position equal to count and fill the vacated bit positions on the left with zeros	OF, CF
<b>SAR</b>	Shift arithmetic right	SAR D, Count	Shift the D right by the number of bit positions equal to count and fill the vacated bit positions on the left with the original most significant bit	OF, SF, ZF, AF, PF, CF

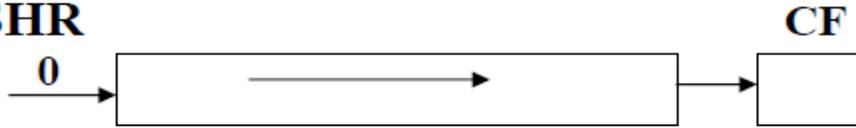
♣ SHL



♣ SAL



♣ SHR

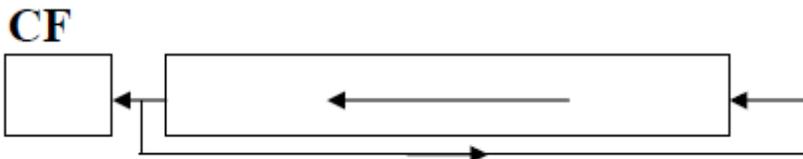


♣ SAR

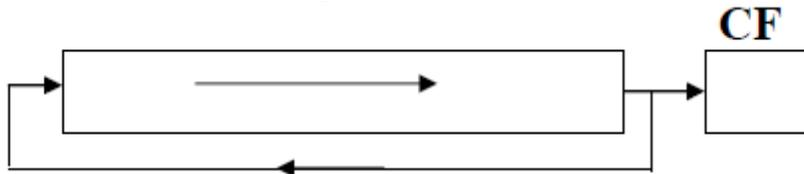


## 5. Rotate Instructions

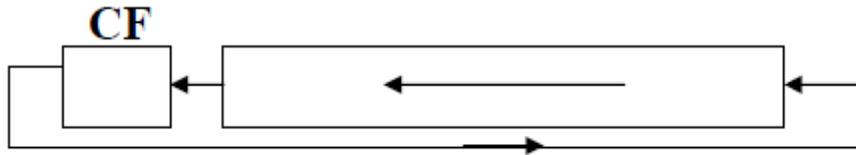
♣ ROL (Rotate Left)



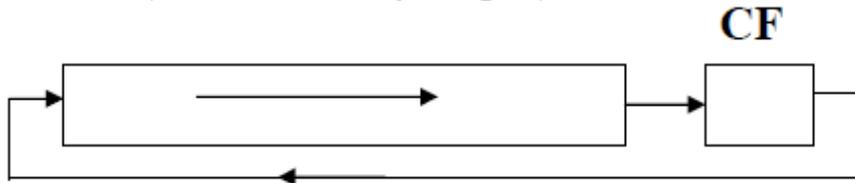
♣ ROR (Rotate Right)



♣ RCL (Rotate Carry Left)



### ♣ RCR (Rotate Carry Right)



## 5. Advance instruction (Program and Control Instruction)

In this section many of instructions that can be executed by the 8086 microprocessor are described, furthermore, these instructions use to write simple programs. The following topics are discussed in this section:

1. Flag control instructions
2. Compare instruction
3. Jump instructions
4. String instruction

### 1. Flag Control Instruction

The 8086 microprocessor has a set of flags which either monitor the status of executing instruction or control options available in its operation. The instruction set includes a group of instructions which when execute directly affect the setting of the flags. The instructions are:

**LAHF**: load AH from flags

**SAHF**: store AH into flags

**CLC**: clear carry, CF=0

**STC**: set carry, CF=1

**CMC**: complement carry, CF= CF

**CLI**: clear interrupt, IF=0

**STI**: set interrupt, IF=1

### EXAMPLE:

Write an instruction to save the current content of the flags in memory location MEM1 and then reload the flags with the contents of memory location MEM2

**Solution:**

```
LAHF
MOV MEM1, AH
MOV AH, MEM2
SAHF
```

**2. Compare Instruction**

There is an instruction included instruction set which can be used to compare two 8-bit number or 16-bit numbers. It is the compare (CMP) instruction.

Instruction	Meaning	Format	Operation	Flag affected
CMP	Compare	CMP D,S	D-S	CF,AF,OF,PF,SF,

Destination	Source
Register	Register
Register	Memory
Memory	Register
Register	Immediate
Memory	Immediate
Accumulator	Immediate

The operands can reside in a storage location in memory, a register within the MPU. Instruction Meaning Format Operation Flag affected the process of comparison performed by the CMP instruction is basically a subtraction operation. The source operand is subtracted from the destination operand. However the result of this subtraction is not saved. Instead, based on the result the appropriate flags are set or reset.

**EXAMPLE:**

Lets the destination operand equals 100110012and that the source operand equals 000110112. Subtraction the source from the destination, we get

$$\begin{array}{r}
 10011001 \\
 \bullet 00011011
 \end{array}$$

Replacing the destination operand with its 2's complement and adding

```
10011001
• 11100101
011111102
```

1. No carry is generated from bit 3 to bit 4, therefore, the auxiliary carry flag AF is at logic 0.
2. There is a carry out from bit 7. Thus carry flag CF is set.
3. Even though a carry out of bit 7 is generated; there is no carry from bit 6 to bit 7. This is an overflow condition and the OF flag is set.
4. There are an even number of 1s, therefore, this makes parity flag PF equal to 1.
5. Bit 7 is zero and therefore sign flag SF is at logic 0.
6. The result that is produced is nonzero, which makes zero flag ZF logic 0.

### 3. JUMP Instruction

The purpose of a jump instruction is to alter the execution path of instructions in the program. The code segment register and instruction pointer keep track of the next instruction to be executed. Thus a jump instruction involves altering the contents of these registers. In this way, execution continues at an address other than that of the next sequential instruction. That is, a jump occurs to another part of the program.

There two type of jump instructions:

- a. Unconditional jump.
- b. Conditional jump.

In an *unconditional jump*, no status requirements are imposed for the jump to occur. That is, as the instruction is executed, the jump always takes place to change the execution sequence. See Figure 17 Instruction Meaning Format Operation Flags affected

Instruction	Meaning	Format	Operation	Flags affected
<b>JMP</b>	<b>Unconditional jump</b>	<b>JMP operand</b>	<b>Jump is to the address specified by operand</b>	<b>None</b>

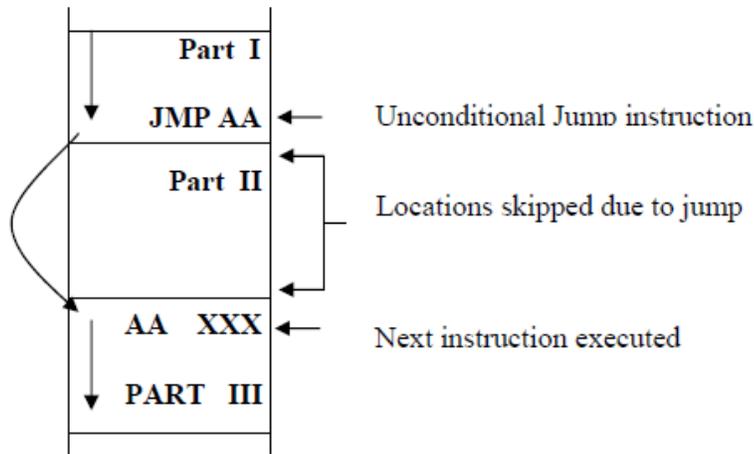


Figure 17: Unconditional jump program sequence.

On the other hand, for a *conditional jump* instruction, status conditions that exist at the moment the jump instruction is executed decide whether or not the jump will occur. If this condition or conditions are met, the jump takes place, otherwise execution continues with the next sequential instruction of the program. The conditions that can be referenced by a conditional jump instruction are status flags such as carry (CF), parity (PF), and overflow (OF). See Figure 18.

Instruction	Meaning	Format	Operation	Flags affected
JCC	Conditional jump	Jcc operand	If the specific condition cc is true, the jump to the address specified by the operand is initiated, otherwise the next instruction is executed	None

The following table lists some of the conditional jump instructions:

Instruction	Meaning
JAE/JNB	Jump if above or equal/jump if not below
JB/JNAE	Jump if below/jump if not above or equal
JC	Jump if carry
JCXZ	Jump if CX is zero
JE/JZ	Jump if equal/jump if zero
JNC	Jump if not carry
JNE/JNZ	Jump if not equal/ jump if not zero
JNO	Jump if not overflow
JNP/JPO	Jump if parity/jump if parity odd
JNS	Jump if not sign
JO	Jump if overflow
JP/JPE	Jump if parity/jump if parity Even
JS	Jump if sign

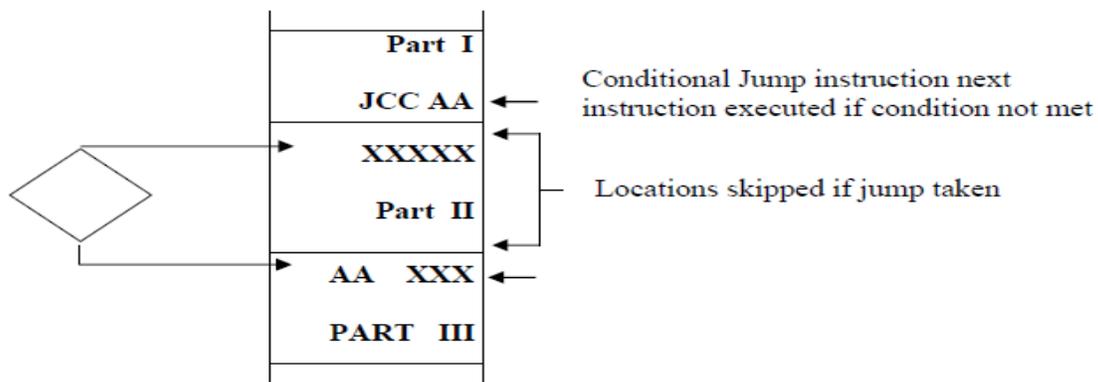


Figure 18: Conditional jump program sequence.

**EXAMPLE:**

Write a program to move a block of N bytes of data starting at offset address BLK1ADDR to another block starting at offset address BLK2ADDR. Assume that both blocks are in the same data segment, whose starting point is defined by the data segment address DATASEGADDR.

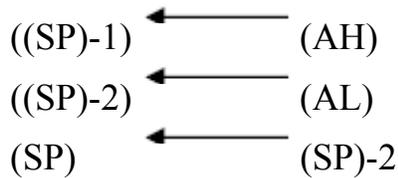
```
MOV AX, DATASEGADDR
MOV DS, AX
MOV SI, BLK1ADDR
MOV DI, BLK2ADDR
MOV CX, N
NXTPT: MOV AH, [SI]
MOV [DI], AH
INC SI
INC DI
DEC CX
JNZ NXTPT
HLT
```

## 4. Push and POP Instruction

It is necessary to save the contents of certain registers or some other main program parameters. These values are saved by pushing them onto the stack. Typically, these data correspond to registers and memory locations that are used by the subroutine.

The instruction that is used to save parameters on the stack is the push (PUSH) instruction and that used to retrieve them back is the pop (POP) instruction. Notice a general-purpose register, a segment register (excluding CS), or a storage location in memory as their operand.

♣ Execution of a PUSH instruction causes the data corresponding to the operand to be pushed onto the top of the stack. For instance, if the instruction is **PUSH AX** the result is as follows:

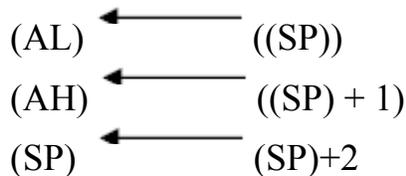


This shows that the two bytes of the AX are saved in the stack part of memory and the stack pointer is decremented by 2 such that it points to the new top of the stack.

♣ On the other hand, if the instruction is

### POP AX

Its execution results in



The saved contents of AX are restored back into the register.

♣ We also can save the contents of the flag register and if saved we will later have to restore them. These operations can be accomplished with the push flags (**PUSHF**) and pop flags (**POPF**) instructions, respectively. Notice the **PUSHF** save the contents of the flag register on the top of the stack. On the other hand, **POPF** returns the flags from the top of the stack to the flag register.

Instruction	Meaning	Operation	Flags affected
PUSHF	Push flags onto stack	$((SP)) \leftarrow (\text{flag})$	None
POPF	Pop flags from stack	$(\text{flag}) \leftarrow ((SP))$	OF, DF, IF, TF, SF, ZF, AF, PF, CF

## 5. String Instructions

The microprocessor is equipped with special instructions to handle string operations. By "string" we mean a series of data words or bytes that reside in consecutive memory locations.

There are five basic string instructions in the instruction set of the 8086, these instructions are:

- a. Move byte or word string (MOVS, MOVSB, and MOVSW).
- b. Compare string (CMPS).
- c. Scan string (SCAS).
- d. Load string (LODS)
- e. Store string (STOS).

They are called the basic string instructions because each defines an operation for one element of a string.

## **Move String**

The instructions MOVSB, MOVSB, and MOVSW all perform the same basic operation. An element of the string specified by the source index (SI) register with respect to the current data segment (DS) register is moved to the location specified by the destination index (DI) register with respect to the current extra segment (ES) register.

After the move is complete, the contents of both SI and DI are automatically incremented or decremented by 1 for a byte move and by 2 for a word move. Remember the fact that the address pointers in SI and DI increment or decrement depends on how the direction flag DF is set.

## **Compare Strings and Scan Strings**

The CMPS instruction can be used to compare two elements in the same or different strings. It subtracts the destination operand from the source operand and adjusts flags CF, PF, AF, ZF, SF, and OF accordingly. The result of subtraction is not saved; therefore, the operation does not affect the operands in any way

### **CMPS BYTE**

The source element is pointed to by the address in SI with respect to the current value in DS and the destination element is specified by the contents of DI relative

to the contents of ES. Both SI and DI are updated such that they point to the next elements in their respective string.

The scan string (SCAS) instruction is similar to CMPS, however, it compares the byte or word element of the destination string at the physical address derived from DI and ES to the contents of AL or AX, respectively. The flags are adjusted based on this result and DI incremented or decremented.

## **Interrupts (INTs)**

Interrupt is a mechanism by which a program's flow of control can be altered, INT provide a mechanism to transfer control to an interrupt service routine (ISR).

This mechanism is similar to that of a procedure call however, while procedure can be invoked only by a procedure call in software. INT can be invoked by both hardware and software. For instance, when an interrupt signal occurs indicating that an external device, such as a printer, requires service. The microprocessor must suspend what it is doing in the main part of the program and pass control to a special routine that performs the function required by the device.

The section of program to which control is passed is called the *interrupt service routine* (ISR). When the microprocessor terminates execution in the main program, it remembers the location where it left off and then picks up execution with the first instruction in the service routine. After this routine has run to completion, program control is returned to the point where the microprocessor originally left the main body of the program.

The interrupts of the microprocessors include two hardware pins that request interrupts (INTR and NMI), and one hardware pin (INTA) that acknowledges the interrupt requested through INTR. In addition to the pins, the microprocessor also has software interrupts INT, INTO, INT3 and BOUND. Two flag bits IF and TF are also used with the interrupt structure and a special return instruction IRET.

All Interrupt whether HW-initiated or SW-initiated, are identify by an INT-Type number that is between (0 and 255), this INT number is used to access the interrupt vector table (IVT) to get the associated interrupt vector. HW interrupt can be masked or disable by manipulating the INT flag using (STI and CLI) instruction.

## **Interrupt Processing**

The Interrupt Vector Table (IVT) is located at address 0, each vector takes 4 bytes. Each vector consist of a (CS:IP) pointer to the associated ISR, 2 byte for specifying the CS, and 2 byte for the offset (IP) within the CS. As shown in the below Figure (19).

The IVT layout in the memory since each entry in the IVT is 4 byte long, INT type is multiplied by 4 to get the corresponding ISR pointer in the table. For example , INT 2 can find the ISR pointer at memory address  $2*4 = 00008H$ , the first 2 byte at the specified address are taken as the offset value, and the next 2 byte as the CS value. Thus executing INT 2 causes the CPU to suspend its current program and calculate the address in the IVT (which is  $2*4=8$ ) and read CS:IP value and transfer control to that memory location.

Just like procedure ISR, should end with a (RET) inst to send control back to the INT program. The interrupt return (IRET) is used for this purpose. On receiving an INT, flag register is automatically saved on the stack. The INT enable flag is clear. This disable attending further INT until this flag is set. Usually, this flag is set in ISR unless there is a special reason to disable other INT.



The current CS and IP values are pushed onto the stack. In most cases, these value CS and IP point to the instruction following the current instruction the CS and IP register are loaded with the address of ISR from the IVI.

When an interrupt occur, the following action are taken:

1. Push flag register on the stack
2. Clear IF and TF
3. Push CS and IP register, on the stack
4. Load CS with the 16-bit data at memory address (INT-type \*4+2)
5. Load IP with the 16 bit data at memory address (INT-type \*4).

The last instruction of ISR is (IRET) instruction, it actions are:

1. POP the 16-value on top of stack into IP register
2. POP the 16-value on top of stack into CS register
3. POP the 16-value on top of stack into flag register.

## **Interrupt Type**

The 8086 microcomputer is capable of implementing any combination of up to 256 interrupts. They are divided into five groups: external hardware interrupts, software interrupts, internal interrupts, the nonmaskable interrupt, and the reset interrupt. The function of the external hardware, software, and nonmaskable interrupt and the rest interrupts can be defined by the user. On the other hand, the internal and reset interrupts have dedicated system functions.

### **Software Interrupt (SW INT)**

Are initiated by execution an INT instruction the format is

**INT INT-Type**

Where INT-Type is an integer number in the range 0-255, thus a total of 256 different types are possible.

## Hardware Interrupt (HW INT)

This type is usually used by peripheral I/O devices such as KB to alter CPU that they require its attention.

HW INT can be divided into Maskable and Non-Maskable (NMI).

A NMI can be generated by applying an electronic signal on the NMI pin this INT is called NMI because the CPU always responds to this signal. In other words, this INT cannot be disabled under program control, the NMI caused by INT2.

Most HW INT are maskable type, and electronic signal should be applied to the INTR (interrupt request) input pin of 8086, 8086 recognizes the INTR only if  $IF=1$ , thus this INT can be masked or disabled by clearing  $IF(IF=0)$ .

## NMI vs. Maskable INT

**NMI** is always attended to by the CPU immediately. Note that when we say immediately, the CPU does not suspend the execution of the current inst in the middle, it completes the current inst and then services the INT.

**Maskable INT** can be delayed until execution reaches a convenient point. As an example, let us assume that the CPU is executing a main program, an INT occurs, as a result, the CPU suspends the main as soon as it finishes the current inst of main and then control is transferred to the ISR. If the ISR has to be executed without any interrupt, the CPU can mask further INT until the ISR is complete. Suppose that, while executing the ISR another maskable INT occurs, service to this INT would have to wait until the ISR is completed.

There are several INT predefined by the microprocessor which have dedicated functions, some of them:

**INT 0 Divide error** INT the CPU generates an INT type whenever executing a DIV results in a quotient that is larger than the destination. The default ISR displays a divide overflow message and terminates a program.

**Single Step INT**, single step is a useful debugging tool to observe the behavior of a program instruction by instruction. To start single stepping the TF bit in flag

register should be set (TF=1), when TF=1, the CPU automatically generate a type 1 INT after execution each inst.

To end single step, TF should be cleared, the CPU however does not have any instruction to manipulate the TF directly

♣ Set trap flag (TF=1) to start single stepping

```
PUSHF
POP AX          ;COPY FLAGS INTO AX
OR AX, 100H    ;SET TF=1
PUSH AX        ;COPY MODIFY FLAG BIT BACK TO FLAG REGISTER
POPF
```

♣ Clear trap flag (TF=0) to end single stepping

```
PUSHF
POP AX          ; COPY FLAGS INTO AX
AND AX, 0FEFFH, ;CLEAR TF=0
PUSH AX        ; COPY MODIFY FLAG BIT BAXK TO FLAG REGISTER
POPF
```

## How can more than one device interrupt?

Computer typically have more than one I/O device requesting interrupt service, like keyboard, hard disk, floppy disk, printer all generate an INT when they required the attention to CPU.

When more than one device INT CPU, we need a mechanism to priority these INT (if they come at the same time) and forward only one INT request at a time to the CPU while keeping other INT request pending for their service.

## Input and Output

Input & Output (I/O) devices provide the means by which a computer system can interact with the outside worlds.

An I/O device can be a purely input device (e.g. KB, Mouse), a purely output device (printer, screen), or both input and output device like (e.g. disk).

Regardless of the intended purpose of I/O devices, all communication with these devices must involve the system bus. However, I/O devices are not directly connected to the system bus. Instead, there is usually, On I/O controller that acts as an interface between the system and the I/O devices.

## **Accessing I/O devices**

As programmer, you can have direct control to any of the I/O devices (through their associated I/O controller).

It is a waste of time and effort if everyone had to develop their own routines to access I/O devices. In addition system resource could be abused either intentionally or accidentally. For instance, and improper disk drive could erase the content of a disk due to a bug in the driver routine.

To avoid this problem and to provide a standard way of accessing I/O devices, OS provide routine to convent all access I/O devices. Typically, access to I/O devices can be obtain from two layer of system software, the basic I/O system (BIOS) and the OS, BIOS is ROM resident and is a collection of routine that control the I/O devices. Both provide access to routine that control I/O devices through a mechanism called INT (interrupt).

## **I/O Address Space and Data Transfer**

As we know I/O ports in the 8086 MPU can be either byte wide or word wide. The port that is accessed for input or output of data is selected by an I/O address. The address is specified as port of the instruction that performs the I/O operation.

I/O addresses are 16 bit in length and are output by the 8086 to the I/O interface over bus lines AD0 through AD15, the most significant bit A16-A19 of the memory address are held at the 0 logic (not used).

Below Figure 20 show a map of I/O address space of the 8086 system. This is an independent 64-KB address space that is dedicated for I/O devices. Notice that its address range is from 000016-FFFF16. Moreover, notice that the eight ports

located from address 00F8 to 00FF are specified as reserved. These port addresses are reserved by Intel for use in their future HW and SW products.

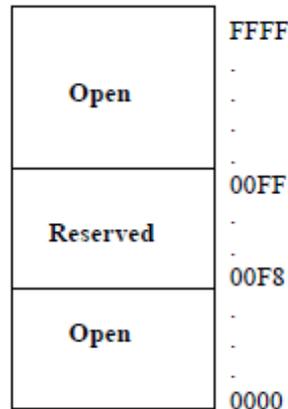


Figure 20: I/O Address Space

Data transfer between the MPU and I/O devices are performed over the data bus. Word transfer take place over the complete data bus D0 to D15, and can required either one or two bus cycle.

**Ports:** a port is a device that connects the processor to the external world through a port processor, receive a signal from an input device and send a signal to an output device.

## Input / Output Instruction

The instruction set contains one type of instruction that transfer information to an I/O device (OUT) and another to read information from an I/O device (IN).

Instruction	Meaning	Format	Operation
IN	Input direct	IN ACC, PORT	ACC ← PORT
	Input indirect	IN ACC, DX	ACC ← (DX)
OUT	Output direct	OUT PORT, ACC	PORT ← ACC
	Output indirect	OUT DX, ACC	(DX) ← ACC

- ACC = AL or AX

**Example 1:** write a sequence of inst that will output FF16 to a byte wide output port at address AB16 of the I/O addresses space.

**Solution:** first the AL register is loaded with FF16 as an immediate operand in the instruction

**MOV AL, 0FF<sub>H</sub>**

Now the data in AL can be output to the byte wide output port with the instruction

**OUT 0AB<sub>H</sub>, AL**

**Example2:** write a series of instruction that will output FF16 to an output port located at address B00016 of the I/O address space.

**Solution:** the DX register must first be loaded with the address of the output port

**MOV DX, 0B000<sub>H</sub>**

Next, the data that is to be output must be loaded into AL

**MOV AL, 0FF<sub>H</sub>**

Finally, the data are output with the instruction

**OUT DX, AL**

**Example 3:** data are to be read in from two byte wide input port at address AA16 and A916 respectively, and then output to a word wide output port at address B00016. Write a sequence of instruction to perform this I/O operation:

**Solution:** we first read in a byte from the port at address AA16 into AL and move it to AH

**IN AL, 0AA<sub>16</sub>**  
**MOV AH, AL**

The other byte can be read into AL

**IN AL, 0A9<sub>H</sub>**

To write out the word of data in AX, we can load DX with the address B00016 and use a variable output instruction

```
MOV DX, 0B000H  
OUT DX, AX
```

## Isolated and Memory I/O

There are two different methods of interfacing I/O to the MPU. In the isolated I/O scheme, the IN, OUT instructions transfer data between the MPU (ACC or memory) and the I/O device.

**Isolated I/O:** it is the most common I/O transfer technique. The addresses for isolated I/O devices, called ports, are separate from the memory. Because the ports are separate from the memory, the user can expand the memory to its full size without using any of the memory space for I/O devices. A disadvantage of isolated I/O is that the data transferred between I/O and the MPU must be accessed by the IN, OUT instructions. See Figure 21.

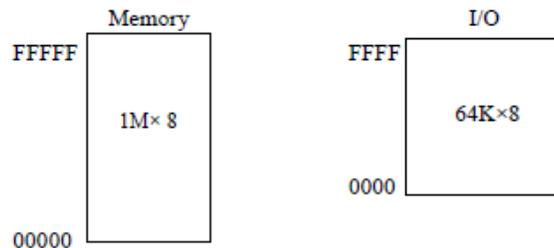


Figure 21: Isolated I/O.

**Memory- Map I/O:** Unlike isolated I/O, memory mapped I/O does not use the IN or OUT instructions. Instead, it uses any instruction that transfers data between the MPU and memory. A memory mapped I/O device is treated as a memory location in the memory map. The main advantage of memory-mapped I/O is that any memory transfer instruction can be used to access the I/O. The main disadvantage is that a portion of the memory system is used as the I/O map. This reduces the amount of memory available to applications. See Figure 22.

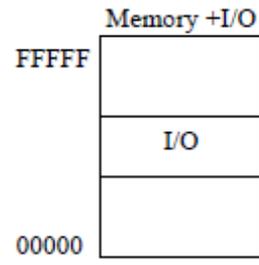


Figure 22: Memory-Mapped I/O