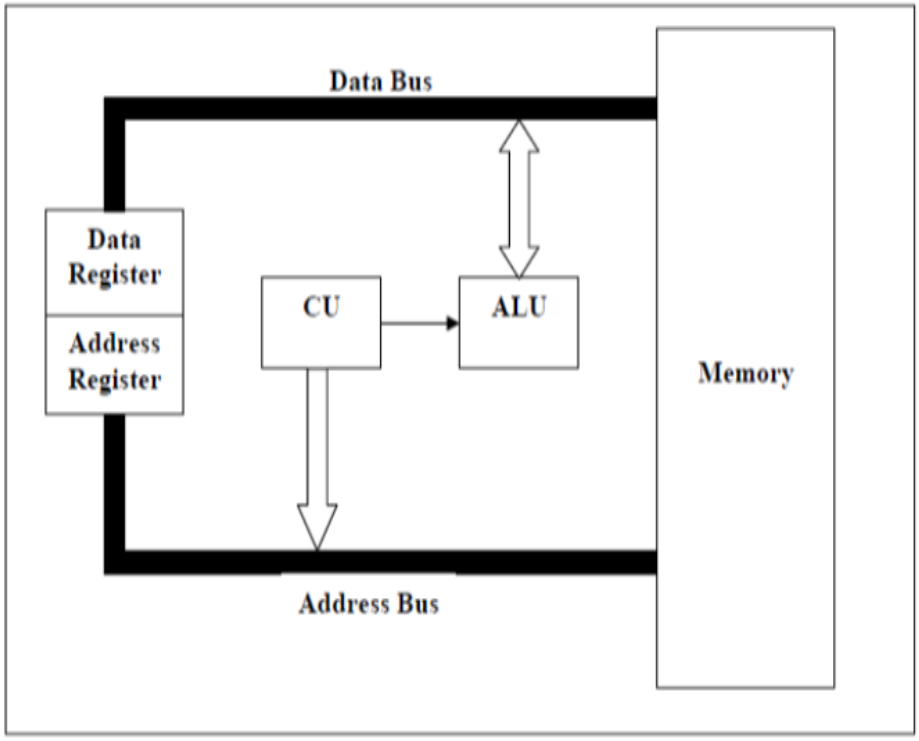


A block diagram of simple CPU.

Computer: An electronic device that accepts input, stores large quantities of data, execute complex instructions which direct it to perform mathematical and logical operations and outputs the answers in a human readable form.

Processor: The part of a computer which controls all the other parts. The CPU **fetches** instructions from memory, **decodes** and **executes** them. This may cause it to transfer data to or from memory or to activate peripherals to perform input or output.

Accumulators: are special classes of register which are closely connected with the ALU such that arithmetic and logical operations can be easily performed on their contents.

Registers: are storage elements, some of which have specific functions owing to the way in which they are implemented in hardware or the task they have to perform.

CU: A **control unit** is circuitry that directs operations within a computer's processor. It lets the computer's logic unit, memory, as well as both input and output devices know how to respond to instructions received from a program**.**

EU: The execution unit is responsible for decoding and executing instructions.

BIU: Responsible of performing all bus operations such as (instruction Fetching from memory, and data transfer between the processor and outside world).

Interrupt: An interrupt is a signal to the processor which causes it to discontinue the current program sequence and start running a different program often called the interrupt service routine

Memory: A unit of a computer in which data is stored for later use, The store of things learned and retained from an organism's activity or experience as evidenced by modification of structure or behavior or by recall and recognition.

Microcomputer Architecture

A computer system has three main components: a Central processing Unit (CPU) or processor, a Memory Unit and Input Output Unit: 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.

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.

Processor: The CPU is divided into two general parts. Arithmetic Logic Unit (ALU) and Control Unit (CU) .

- The **ALU** comprises circuitry to perform arithmetic and logical operations, such a ADD, SUBTRACT, AND, OR, SHIFT.

- The **CU** fetches data and instruction, and decodes addresses for the ALU.

Processor has two speeds: -

1. Internal speed(internal clock) which is the speed of data exchange within the processor and the greater the internal processor frequency increased the amount of mutual orders within the processor and thus implement more operations per second.

2 external speed (system bus), a speed exchange of data between the processor and the memory card and the screen and the information passed on by the front carrier FSB(front side bus).

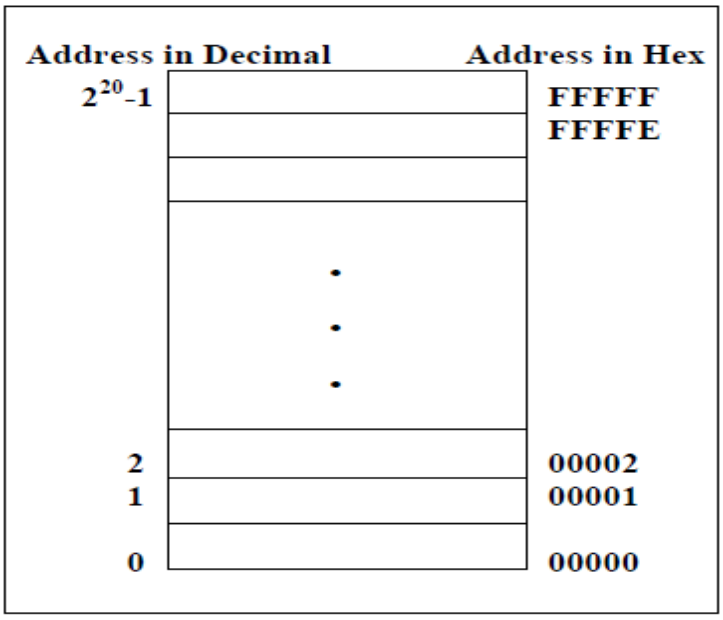
Memory:

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. This is referred to as memory address of the byte. Such memory is called byte addressable memory.

8086 can address up to 1 MB (220 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.



Logical view of the system memory

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.

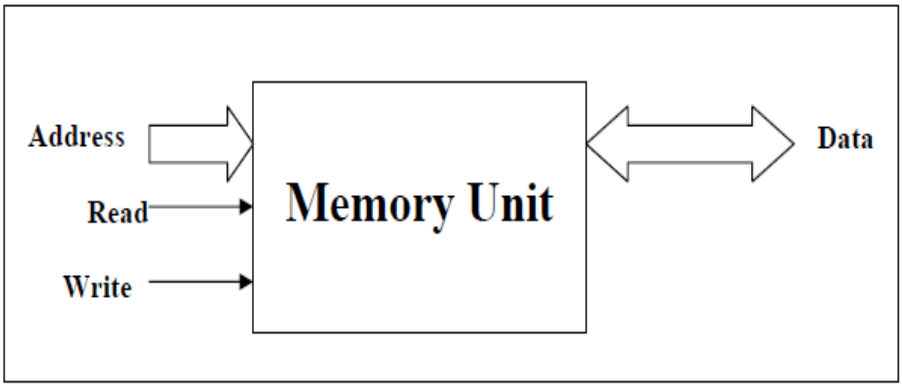
Dynamic RAM: main memory, or RAM is where program, data are kept when a program is running. It must be refreshed with in less than a millisecond or losses its contents.

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.

Two basic memory operations

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



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.

Intel 8086 Microprocessor

Processor deal with memory method:

Should be required program moves executed on the computer first, from the storage unit where there to electronic computer memory to the processor to implement the orders required within the program, when the ends of the execution of an order and needs to move to execute the following command, the processor sends a request to the charge of the control program memory in order to provide him the next order as processor also be

performed in the memory control program by providing it with data that may be needed and also asked to run an example:

If the program is a request from the processor to collect the number in the field is a with the number in the field b must be transferred from electronic memory to the processor first is the combination and then transferred to him the contents of the contents of the field a B to be carried out which program he wants and the process of transition These are called access time It is Atstgrq more than 1on 1000 of a second, and the speed of the processor to get a big effect on the results of instruction and measured its speed MHZ, GHZ .

Operations of a CPU:

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 struction:

- 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.

The fetch and Execute Cycle:

The organization of the processor into a separate BIU and EU allows the fetch and execute cycle to overlap. To see this, consider what happens when the 8086 is first started.

1- The BIU outputs the contents of the instruction pointer (IP)onto the address bus, causing the selected byte or word in memory to be read into the BIU.
2- Register IP is incremented by one to prepare for the next instruction fetch.
3- Once inside the BIU, the instruction is passed to the queue: a first-in/first-out storage register sometimes likened to a pipeline.
4- Assuming that the queue is initially empty, the EU immediately draws this instruction from the queue and begins execution.
5- While the EU is executing this instruction , BIU proceeds to fetch a new instruction. Depending on the execution time of the first instruction, the BIU may fill the queue with several new instructions before the EU is ready to draw its next instruction.
6- The cycle continues, with the BIU filling the queue with instructions and the EU fetching and executing these instructions.
   The BIU is programmed to fetch anew instruction whenever the queue has room for two additional bytes. The advantage to this **pipelined** architecture is that the EU can execute instructions (almost) continually instead of having to wait for the BIU to fetch anew instruction.

System Bus:

A **Bus** is a bunch of wires, and electrical path on the printed IC to which everything in the system is connected.

The registers and ALU are not permanently connected together, but are joined by a DATA BUS. This bus allows data to be passed from a register to the ALU at one instant and from the ALU to a register at another instant. More generally, any device which needs to read or write data will be connected to the data bus. The timing and control circuitry ensures that only one pair of devices is using the data bus at any time (one writing data and the other reading it).

The amount of data storage provided internally in the microprocessor is limited. Only a few operands can be stored in the internal registers. If the required program steps were to be stored internally in registers, this would place an unacceptable limit on the size of program allowed.

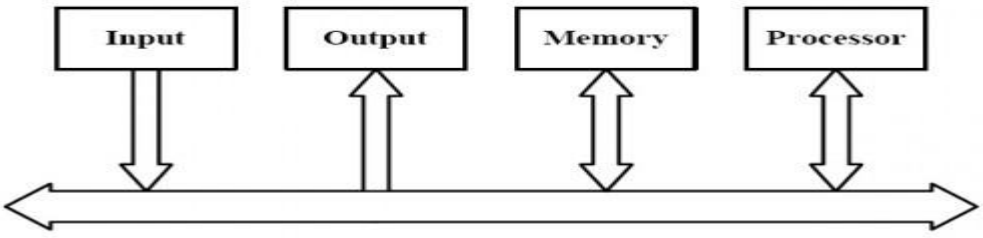
So some means of extending the available storage is provided. External storage elements, or memory, can be read or written to using an extension of the internal data bus. Buffers are provided to interface with external memory chips over an external data bus. These memory chips allow program instructions to be stored, which can be read into the microprocessor at the appropriate time. Results of calculations and data to be input to the ALU can also be stored in this external memory. The arrows on the diagram show that the data bus is bidirectional, that is, data must be able to pass to and from external memory as the microprocessor writes or reads it. Clearly, a control signal is provided to indicate in which direction the data is travelling. This line is called READ/NOT-WRITE or R/W for short. When HIGH, a memory read operation is taking place; when LOW, a memory write (ie. the microprocessor is transferring data to external memory).

Since there will be a large number of external memory locations, we need to specify which one is to be used for transferring data to or from the

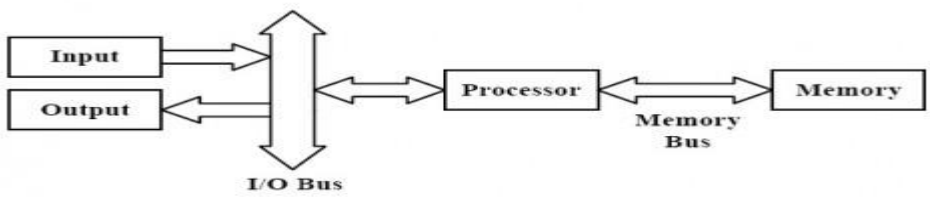
microprocessor. This is achieved by the address bus. The address bus carries a 16-bit code which uniquely identifies the external memory location we require. Each memory location will only respond to its own code (called its address) and, according to the setting of the R/W line, will take data from the data bus or place data on the data bus. The address bus is unidirectional since it only makes sense for the microprocessor to generate addresses as it is the system controller.

<u>There are three types of Bus</u>:

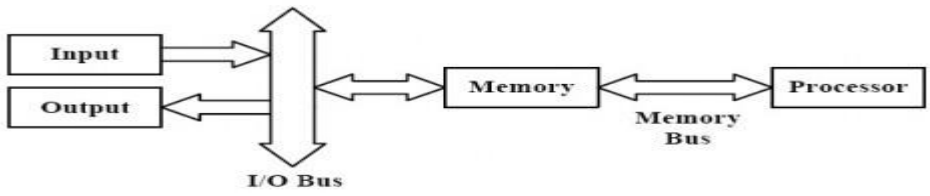
1- **Address Buss (AB):** the width of AB determines the amount of physical memory addressable by the processor.
2- **Data Bus (DB):** the width of DB indicates the size of the data transferred between the processor and memory or I/O device.
3- **Control Bus (CB):** consists of a set of control signals, 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.



Configuration 1



Configuration 2



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 represents Bit,Byte,and Larger units.

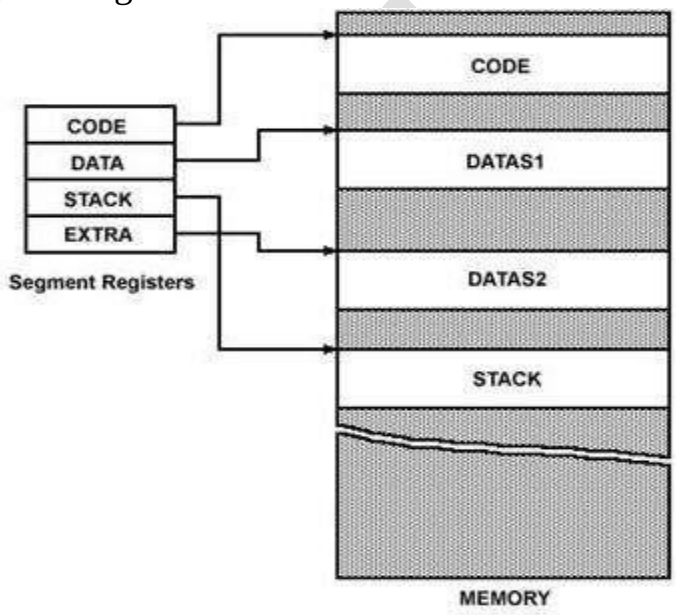
Name	Number of Byte
Bit	0 or 1
Byte	is a group of bits used to represent a character, typically 8-bit.
Word	2-bytes (16-bit) data item
Double Word	4-byte (32-bits)
Quadword	8-Bytes (64-bit)
Paragraph	16-bytes (128-bit)
KiloByte (KB)	the number 2^{10} = 1024 = 1K for KiloByte, (thus 640K = 640 * 1024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

Segmented memory:

The memory of 8086 microprocessor is divided into sixteen parts or segments. In the given diagram we will take or use only four segments that are:

1- Code segment
2- Data segment
3- Stack segment
4- Extra segment

These segments store the specific data. Four segments registers are used to store or hold the initial address or base address. Each of the segment stores 64KB. Which means that another register is required in which the memory locations address change. We know that basically there is no change in the base address so, the address on the special purpose registers are added with the base address of the segment register.



CS 64K	4000H
DS 64K	3000H
SS 64K	2000H
ES 64K	1000H

So in this manner the actual address is made the segment register are special in 8086 microprocessor. They were designed to solve the problem that is index register and pointer register are 16 bit and the memory in 8086 microprocessor is 1 MB which requires a 20 bit address, the index and pointer register are not wide enough to address directly any memory location a segment of memory is a block of 64 KB of memory addresses by special registers called segment register.

The data are pointed in a segment by the index register, pointer register, base register and instruction register. Each segment register holds or stores the 16 bit portion of the 20 bit address of a64 KB segment of memory.

The 20 bit address is made by adding the segment register with a 0H or 0000. That is placed on the least significant end of the number in the segment register.

Code Segment

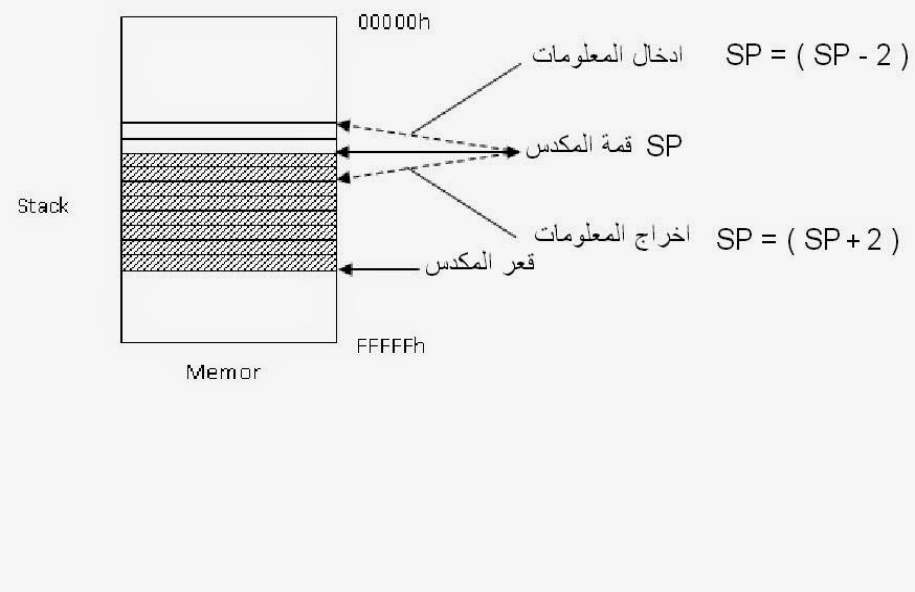
The code segment is that section of memory that stores the different codes used by the microprocessor. The code segment register is used for this segment to store the starting address off the code segment. Its length is 64 KB.

Data segment

This is the section of memory that stores the general data used by programmers. The data are usedor accessed in the data segment by an offset address of other register that hold the offset address. It is limited to 64KB.

Extra segment

This segment is used to store the data which is extra for.



Stacksegment:

This segment stores the data in such a manner which is in successive form and its capacity is 64 KB.

Physical address = Segment address + offset address

Physical address = [Segment address*10H] + [offset address]

Example :

If the data segment register contain 1000H and the offset register contain 2000H . findbellow:

1- Start address of this segment
2- End address.
3- Physical address.

Sol:

DS=1000H

Start address of the segment = 1000H*10H = 10000H

End address of the segment = 10000H+FFFFH= 1FFFFH.

Physical address=Start address(segment address)+offset address

10000H + 2000H= 12000H

Example:

If the code Segment register contain 1400H and the IP register contain 1200H,find :

- ths start address?

- the end address?

- the address of the next instruction to be feched by the microprocessor?

Sol:

The start address of the code segment = CS*10H=1400H*10H=14000H

The end address of the code segment = start address + FFFFH

14000H + FFFFH = 23FFFH.

Physical address = start address = offset address = 14000H + 1200H=14200H

Example:

If SS:SP=1234:4267 H find the start , end and the physical address?

Sol:

SS mean Stack segment register = 1234H.

SP mean stack pointer register = 4267H.

The start address of SS = SS * 10H = 1234H * 10H = 12340H

The end address of SS = start address + FFFFH =12340 + FFFF = 2233FH

The physical address = Start address + offset address

= 12340 H + 4267H = 165A7H.

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.

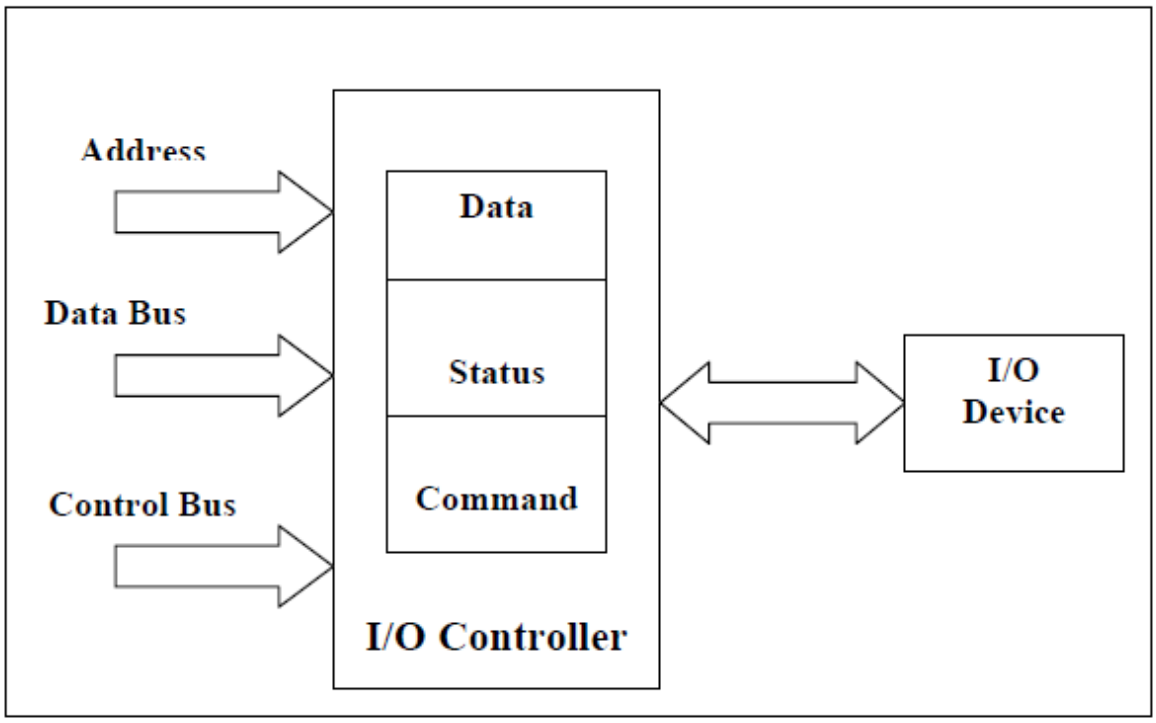
Devices that are used to communicate like, printer, keyboard, modem, Devices that are used to store data like disk drive. I/O devices are connected to the system bus through **I/O controller** (interface) – which acts as interface between the system bus and I/O devices.

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.



Block diagram of a generic I / O device interface.

Interfaces:

It is the device that manages the connection between two devices in the digital computer system. And its way of management is differ and depend on which devices is connected. Therefore we can simply classified it into:

1- Interface between CPU and Main Memory
   In this case the interface is a simple device offers these jobs:
   a- Specified the type of operation with Main Memory if it read or write.
   b- Specified the location which will consider to execute the operation.
2- Interface between CPU and Mass Storage Memory
   In this case the interface is a more complicated device offers these jobs:
   a- Specified the type of operation with Mass Storage Memory if it read or write.
   b- Specified the location which will consider to execute the operation.
   c- Synchronize the CPU with Mass Storage device.
   d- Present a temporally storage for transferred information.
3- Interface between CPU and I/O Sub System
   In this case the interface is a very complicated device offers these jobs:
   a- Specified the type of operation with I/O sub system if it read or write.
   b- Specified the device which will consider to execute the operation.
   c- Synchronize the CPU with I/O sub system.
   d- Present a temporally storage for transferred information.
   e- Convert the information in suitable form that match the required task.

Registers:

Small memories and very fast exist within the processor in order to save the digits to be processed from the unit of arithmetic and logic as it are not taken to implement any process in the register, without saves its data in registers until the implementation and the register have memories RAM of a temporary type SRAM This is the secret of being very fast and when we say that the processor 8086 supports 16Bit it means that the main recorders length equal to 16 Bit.

Registers are not part of the main memory, but a special temporary data store central processing unit and because the recorders are on-chip electronic processor During addressed the central processing unit will be accelerated much memory to use in obtaining the required data as the access to the memory location may require a cycle one time or more while access to the recorders may take the zero of the number of cycles of time and for this reason it must always try to keep variables in registers rather than using memory locations and more mathematical and logical operations is carried out by these recorders.

Registers types:

1- General purpose registers:

Despite the name of a register, it's the programmer who determines the usage for each general purpose register. The main purpose of a register is to keep a number (variable). The size of the above registers is 16 bit, 4 general purpose registers (AX, BX, CX, DX) are made of two separate 8 bit registers. Therefore, when you modify any of the 8 bit registers 16 bit register is also updated, and vice-versa. Because registers are located inside the CPU, they are much faster than memory

1- AX : Called the Accumulator register. It is used for arithmetic and logic, I/O port access, interrupt (divided into AH / AL).

2- BX : Called the Base register It is used to To carry not directly titles in memory and can be used for private purposes (divided into BH / BL).

3- CX: Called the Counter register It is used as a loop counter and for shifts Gets some interrupt values (divided into CH / CL).

4- DX: Called the Data register It is used for I/O port access, arithmetic, some interrupt calls (divided into DH / DL).

2-Segment registers :

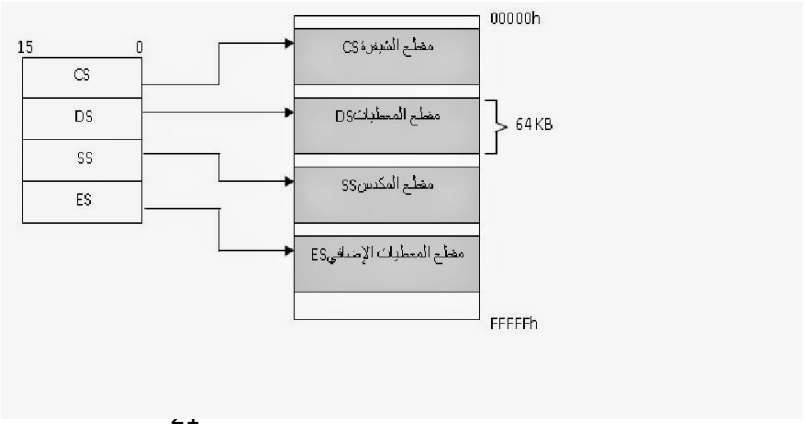
The segment registers have a very special purpose pointing at accessible blocks of memory. Segment registers work together with general purpose register to access any memory value.

CS : Holds the Code segment It contains a primary site to instruct the executable.

DS : Holds the Data segment that your program accesses. Changing its value might give erroneous data.

ES : Extra piece contains a memory where additional data and contain variables BL exist

SS : Holds the Stack segment your program uses. Sometimes has the Same value as DS. Changing its value can give unpredictable results, mostly data related.



3- **Pointer and index Registers**

These registers are four general-purpose registers: two pointer registers and two index registers. They store what are called **offset addresses**. An **offset address** selects any location within 64 k byte memory segment.

1- SP: Stack poiter
2- BP: Base poiter
3- DI: destination index
4- SI: Source index

4-Special Purpose Registers:

They recorders that are connected to the address bus and can not access to these registers as in the rest of the other types, but that the only processor deals with these recorders directly and there are two types of which are: -

1. Registered instruct pointer (IP):

The job as containing instruct currently being implemented and contains on offset of the next instruction to be executed that's mean amendment next, which will be implemented where it can refer to any position in the memory blade containing many memory different location and connects with CS segment register to get to the data that cannot be accessed directly you need to clip starting address.

2. Flags register :

It is a special recorder deals with 16-bit and every bit is a reference to a particular subject and 16-bit simply selective group of bits and these bits are working to determine the current state of the processor and despite the fact that the number of these bits is 16-bit, but we use the 9 bit only , 4 It is commonly used zero case (ZF) and pregnancy (CF) and the status of the signal (SF) and the case of pregnancy surplus (OF).

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C

O → over flow **T → trap** **A → axulary**

D → direction **S → sign** **P → parity**

I → interrupt **Z → zero** **C → carry**

DF Direction flag : Controls incr. direction in string ops (0=inc, 1=dec)

IF Interrupt flag : Controls whether interrupts are enabled

TF Trap flag : Controls debug interrupt generation after instructions

SF Sign flag : Indicates a negative result or comparison

ZF Zero flag : Indicates a zero result or an equal comparison

AF Auxiliary flag : Indicates adjustment is needed after BCD arithmetic

PF Parity flag : Indicates an even number of 1 bits

CF Carry flag : Indicates an arithmetic carry occurred

OF The overflow flag : indicates that the signed result is out of range. If the result is not out of range, OF remains reset.

The other three flags are control flags. These three flags provide control functions of the 8086 as follows:

1. The Trap flag(TF)
2. The interrupt flag(IF)
3. The direction flag(DF)

The Trap Flag:

- Setting TF puts the processor into single step mode for debugging, In single stepping microprocessor executes a instruction and enters into single step ISR.
- If TF=1, the CPU automatically generates an internal interrupt after each instruction, allowing a program to be inspected as it executes instruction by instruction.

The Interrupt Flag:-

- If IF=1, the CPU will recognize external interrupt request (Interrupt Disabled). If IF=0, then interrupt disabled.
- Clearing IF disables these interrupts.
- IF has no effect on either non-maskable external or internally generated interrupt.

The Direction Flag:-

- This bit is specially for string instructions.
- If DF=1, the string instruction will automatically decrement the pointer. If DF=0,the string instruction will automatically increment the pointer.

Programming languages:

The program is written in sequence of statements in form that people prefer to think in when solving a problem. The program which is written by high level language is not understand by PC directly.

Languages are systems of communication. When a person speaks English meet person speak Arabic, both cannot understand each other unless one of them speak with the second's language.

Programming languages allow the programmer to communicate with computers. Programming languages can be categorized as

- Low level languages
- High level languages

Computer PC understand Machine language(0,1) to that we use Translator between HLL and LLL or machine language such as (compiler, interpreter and assembler).

Low level languages (LLL)	High level languages (HLL)
1.oriented toward the computer 2.easier and faster for the computer to Execute. 3.the programmer must be familiar With H/W (microprocessor) for Which the program is being written. 4.programs written in many forms like(Binary, Hexadecimal and Assembly) 5.it is difficult to programmer to work With it.	1.oriented toward the programmer. 2.Easier for people to use and Understand. 3.the programmer must be familiar With high level languages to write his/her program. 4.programs writen in (Basic, pascal, C, C++) 5. it is easy to programmer to work With it.

Assembly Code Programming:

Until now we have been writing programs in Machine Code - that is the actual binary codes which the computer can understand and execute directly. Actually we have been using the hexadecimal equivalent of the instructions to save typing long strings of ones and zeros. In the lab when we type in a hex instruction there is a program running which converts the hex code into the appropriate binary and stores it in memory. Thus we have been using a very simple translation program to make life easier for ourselves. Early computers did not even have that and programming was done on a row of switches, one for each bit.

If we can get the computer to do a simple translation like hex to binary then we can get it to do a slightly more useful translation of mnemonics to machine code. The mnemonics are easily remembered shorthand versions of the instructions which we have already met LDA, LSR, ADDA etc. A program which converts mnemonics to machine code is called an Assembler and the language based on these mnemonics is known as Assembly Code.The Assembler uses a look-up table to convert each mnemonic into the appropriate code, The assembler works by reading the source code program and building up a symbol table which lists all the labels and their equivalent values.

Assembly Code Programming is basically written in four columns. It's easiest to use 'tab' to move to these columns:

• **First column: labels.** Labels are used instead of memory addresses. When Your program is compiled, the labels are replaced with the actual memory address – the assembler calculates these for you.

• **Second column: instructions.** The symbolic instructions that the assemble Will translate into opcodes.

• **Third column: operands.** The data or registers that the instructions Operate on.

• **Fourth column: comments.** You do comment your code, don't you?

Assembler Instruction format:

Label: Mnemonic operand, operand ; Remarks

(Destination),(Source)

MOV A,M ; ADD element of location 0f Memory to A

MOV B,D ; Register to Register transfer

MVI E,06H ; transfer of immediate operand to register

Example Assembly Programs

Some simple arithmetic:

Source program	*Generated assembly*
b = 4	.begin
c = 10	load FOUR
a = b + c - 7	store b
	load TEN
	store c
	load b
	add c
	subtract SEVEN
	store a
	halt

Advantages of Assembly language:

- It is easier to anderstand and use compared to machine language.
- It is easy to locate and correct errors.
- It is easy to modified.

Disadvantages of Assembly language:

- Like machine language it is also machine dependent.
- Since it is machine dependent therefore programmer should have the knowledge of the hardware also.

There is a relationship between the offset register and the segment register must pay attention to this relationship

Seg register	CS	DS	SS
Offset register	IP	SI , DI , BX	SP , BP

Addressing Mode:

The term addressing modes refers to the way in which the operand of an instruction is specified. Information contained in the instruction code is the value of the operand or the address of the result/operand. Following are the main addressing modes that are used on various platforms and architectures.

1) Register Addressing Modes:
   By specifying the name of the register as an operand to the instruction You may access the contents of that register. Consider the 8086 mov instruction.

   ```
   mov  destination, source
   ```

   this instruction copies the data from the source operand to the destination operand.

   ```
   mov  Ax, bx  ; copies the value from bx into Ax
   mov  dl, al  ; copies the value from al into dl
   ```

2) Immediate Addressing Mode:
   This addressing mode transfers the source-immediate byte or word of data into the destination register or memory location.

   ```
   mov  Al, 22H
   ```

   This instruction copies a byte size 22H into register Al.

   ```
   mov  ESI, 12345678H
   ```

   this instruction copies a double-word sized 12345678H into register ESI.
3) Displacement Mode:
   This mode consists of a 16 bit constant that specifies the address of the target location. The instruction mov al, ds:[8088h] load the al register with a copy of the byte at memory location 8088h.
   Likewise, the instruction mov ds:[1234h],dl
   stores the value in the dl register to memory location 1234h:

Example: statement memory condition after implementation of this instruct when DS = 1512

```
MOV AL,99H
MOV [3518H],AL
```

Sol /

1- Ph =1512 * 10 =15120
2- 15120 +3518 =18638 99H put in this location

4) Indirect Mode:

The 80x86 CPUs let you access memory indirectly through a register using the register indirect addressing modes. There are four forms of this addressing mode on the 8086, best demonstrated by the following instructions:

```
mov al, [bx]
mov al, [bp]
mov al, [si]
mov al, [di]
```

As with the x86 [bx] addressing mode, these four addressing modes reference the byte at the offset found in the bx, bp, si, or di register,

Example: statement memory condition after implementation of this instruct when DS = 1120 , SI = 2498 , AX = 17FE ,

```
MOV [SI],AX
```

Sol /

1- Log DS = 1120
2- Ph .A = 1120 * 10 = 11200
3- 11200 + 2498 =13698

4- FE put in 13698 location but 17 put in 13698 +1=13699loc.

5) Indexed Addressing Mode:

The indexed addressing modes use the following syntax:

mov al, disp[bx]
mov al, disp[bp]
mov al, disp[si]
mov al, disp[di]

If bx contains 1000h, then the instruction mov cl,20h[bx]will load cl from memory location ds:1020h. Likewise, if bp contains 2020h, mov dh,1000h[bp] will load dh from location ss:3020.

The offsets generated by these addressing modes are the sum of the constant and the specified register. The addressing modes involving bx, si, and di all use the data segment, the disp[bp] addressing mode uses the stack segment by default. As with the register indirect addressing modes, you can use the segment override prefixes to specify a different segment:

mov al, ss:disp[bx]
mov al, es:disp[bp]
mov al, cs:disp[si]
mov al, ss:disp[di]

6) Based Indexed Addressing Mode:

The based indexed addressing modes are simply combinations of the register indirect addressing modes. These addressing modes form the offset by adding together a base register (bx or bp) and an index register (si or di). The allowable forms for these addressing modes are

mov al, [bx][si]
mov al, [bx][di]
mov al, [bp][si]

mov al, [bp][di]

Example: statement memory condition after implementation of this instruct when DS = 4500 , SS = 2000 , BX =2100 , SI = 1486 , DI = 8500 , BD = 7814 , AX = 1512.

a) MOV [BX] +20,AX

sol /

1- Ph.A = 4500*10=45000
2- 45000 + 2100 =47100
3- 47100 + 20 = 47120
   AX = 15 12

Put 12 in location 47120

Put 25 in location 47121

b) MOV [SI]+10,AX

sol /

1- Ph.A= 4500*10=45000
2- 45000+1486 = 46486
3- 46486+10 = 46496 this location for 12
   46496+1=46497 this location for 25

c) MOV CL,[BX][DI]+8

sol /

1- Ph.A=4500*10=45000
2- 45000+2100+8500+8= 4F608 this location for 12 and 4F609 for 25

8086 Address Modes

Type	Instruction	Source	Address Generation	Destination
1-Register	MOV AX,BX	register BX	→	register AX
2-Immediate	MOV CH3,3AH	Data 3AH	→	register CH
3-Direct	MOV [1234], AX	register AX	(DS*10H)+Displacement → 10000H + 1234	Memory 11234H
4-Indirect	MOV [BX],CL	register CL	(DS*10H)+BX → 10000+0300H	Memory 10300H
5-Index	MOV [BX+SI],BP	register BP	(DS*10H)+BX+SI → 10000H+0300H+0200H	Memory 10500H
6-Relative	MOV CL, [BX+4]	memory 10304H	(DS*10H)+BX+4 → 10000H+0300H+4	Register CL

ASSUME BX= 0300H, SI= 0200H, ARRAY= 1000H, DS= 1000H

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.

ADD destination, source1, source2

EX: A=B+C

```
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 AX, BX
ADD AX, CX
```

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

```
MOV AX, X
ADD AX, D
MOV BX, N
ADD BX, 1
MUL BX
MOV Y,AX
```

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.

ADD source

LDA source; copy value from memory to ACC.
STA destination; copy value from Acc into memory.

EX: A=B+C

```
LDA B
ADD C
STA A
```

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

```
LDA X
ADD D
STA T1
LDA N
ADD 1
MUL T1
STA Y
```

4. Zero Operand instruction

Some computers have arithmetic instruction in which all operands are implied, these zero operand instruction use a stack, a stack is a list structure in which all insertion and deletion occur at one end, the element on a stack may be removed only in the reverse of the order in which they were entered. The process of inserting an item is called **Pushing**, removing an item is called **Popping**.

Computers that use Zero operand instruction for arithmetic operations also use one operand **PUSH** and **POP** instruction to copy value between memory and the stack.

PUSH source; Push the value of the memory operand onto the Top Of the stack.
POP destination; POP value from the Top of the stack and copy it into The memory operand.

EX: A=B+C

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

```
PUSH B          PUSH X
PUSH C          PUSH D
ADD              ADD
POP A           PUSH N
                PUSH 1
                ADD
                MUL
                POP Y
```

NOTE IN ADD Pop the two value of the stack, add them, and then push the sum back into the stack.

Instructions set:

8086 has 117 instructions, these instructions divided into groups:

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

MOV use to transfer a byte or a word of data from a source operand to a destination operand. It's more useful data transfer instruction because it transfers the data from one memory location to another. 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 CX=0100H DX=CS=0100H CS → DX
2. MOV AX, 05H Transform the value 05 to AX
3. MOV BX, [0ABCD] Transform the value that saved in location 0ABCD to BX

XCHG

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

EXAMPLES:

XCHG AL, DL Exchanges the contents of AL with DL.

Load Effective Address LEA, LDS, LES, LFS, LGS, and LSS

There are several load-effective address instructions in the microprocessor instruction set. The LES instruction loads any 16 bit register with the offset address of the data specified by the operands, as determined by the addressing mode selected for the instruction.

EXAMPLES:

LED BX,[DI] MOV BX,[DI]
The LDS, LES, LCS and LSS instructions load any 16 bit or 32 bit register with an offset address and the DS, ES, CS or SS segment register with a segment address.

EXAMPLES:
LDS BX,[DI] ,This instruction transfers the 32bit number addressed by DI in the data segment into BX and DS register.

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:
((SP)-1) (AH)
((SP)-2) (AL)

This shows that the two bytes of the AX are saved in the stack part of memory and the stack pointer is decrement 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

(AL) ((SP))

(AH) ((SP) + 1)

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.

2. Arithmetic Instructions

Arithmetic instructions include instructions for the addition, subtractions, multiplication and division 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, OF, SF, AF, ZF, and PF.

Addition: ADD, ADC, and INC

Addition (ADD) appears in many forms in the microprocessor, which may be come as add-with-carry instruction (ADC) and Increment instruction (INC) that add 1 to a register or a memory location.

Format:

```
ADD AX,BX          AX= AX+BX
ADC AX, BX         AX=AX+BX+CF
INC AH             AH= AH +1
```

EXAMPLES:

```
ADD BX,044H        BX=BX+44H
ADD AX,[1234H]     AX add with the contents of the address [1234]
ADD AX, BX         AX= 1100H, BX=0ABCH
                   AX= 1100H+ 0ABCH = 1BBCH
```

EXAMPLES:

The original contents of AX, BL, memory location SUM, and CF are AX=1234H, BL= ABH, Sum=00CDH and CF=0 respectively, describe the result of execution the following sequence of instruction:

```
ADD AX, SUM
ADC BL, 05H
INC SUM
1. AX= 1234H + 00CDH = 1301H        CF=0
```

2. BL= ABH +05H +0=B0H CF=0
3. SUM=00CDH + 1=00CEH CF=0

Instructions	AX	BL	SUM	CF
Initial state	1234H	ABH	00CDH	0
ADD AX, SUM	1301H	ABH	00CDH	0
ADC BL, 05H	1301H	B0H	00CDH	0
INC SUM	1301H	B0H	00CEH	0

Subtraction: SUB, SBB, DEC, and NEG

Many forms of subtraction (SUB) appear in the instruction set, which may be come as subtract-with-borrow instruction (SBB) and decrement instruction (DEC) that subtracts 1 from a register or a memory location.

Format:

SUB AX, BX AX=AX - BX
SBB AX, BX AX= AX - BX - CF
DEC AH AH= AH -1

EXAMPLES:

SUB DH,06FH DH=DH-6FH
SUB AX,[1234H] AX sub value equal to contents the address [1234]
SBB BX, CX BX=1234H, CX=0123H, CF=0
BX=1234H-0123H-0 =1111H
NEG BX BX= 3A H 0011 1010
1111 1111 1100 0101+1= 1111 1111 1100 0110
= FFC6H

Multiplication and Division MUL, DIV

Format:

MUL CX	(AX) = AX * CX
DIV CL	(AH), (AL) = AX/CL AL contain the quotient, AH is the reminder
DIV CX	DX, AX= (DX,AX)/CX AX contain the quotient, DX is the reminder

EXAMPLES:

MUL CL where AL=-1, CL= -2

AX = FF H * FE H

= FD02H

Comparison

The comparison instruction (CMP) is a subtraction that changes only the flag bits, the destination operands never changes. A comparison is useful for checking the entire contents of register or a memory location against another value.

EXAMPLES:

CMP CX, BX	CX – BX
CMP [DI], CX	CX subtracts from the byte contents of the data segment memory location by DI

The compare and exchange instruction (CMPXCHG) compares the destination operand with the accumulator. If they are equal, the source operand is copied into the destination, if they are not equal, the destination operand is copied into the accumulator.

EXAMPLES:

CMPXCHG CX, DX	CX compare with AX, if equal DX copied into AX, else CX copied into AX.

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

The basic logic instructions include AND, OR, Exclusive-OR and NOT. Logic operations always clear the carry and overflow flags, while the other flags change to reflect the condition of the result.

The AND operation performs logical multiplication. The OR operation performs logical addition and is often called the Inclusive-OR function. The Exclusive-OR instruction refers to XOR. NOT is one complement.

EXAMPLES:

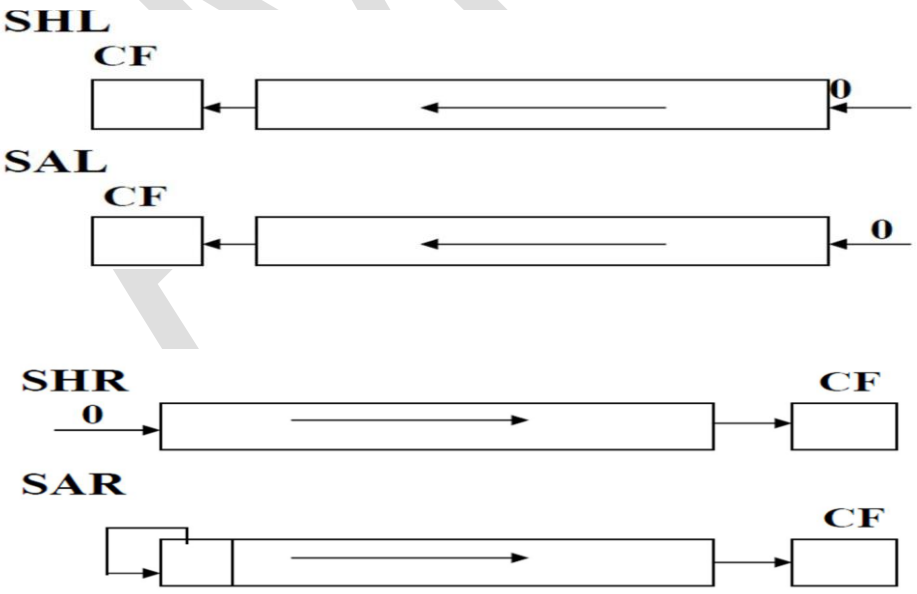
AND AL,BL	AL=AL and BL
OR AH,BH	AH=AH or BH
XOR CH,DL	CH=CH xor DL
NOT CX	

4. Shift and Rotate Instructions

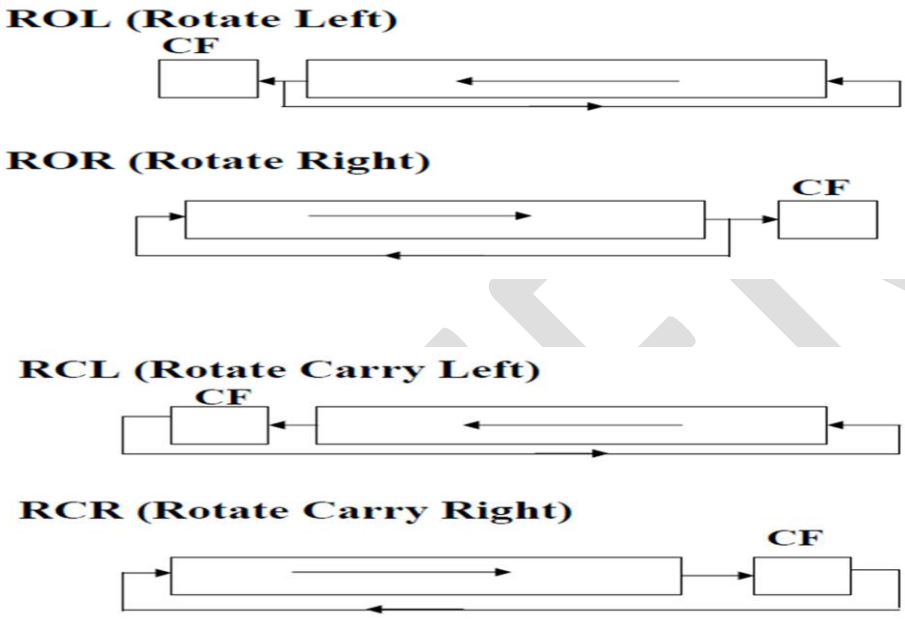
Shift and rotate instructions manipulate binary numbers at the binary bit level. Shift and rotate instructions find their most common applications in lowlevel software used to control I/O devices. The microprocessor contains a complete set of shift and rotate instruction that are used to shift or rotate any memory data or register.

Shift instructions position or move number to the left or right within register or memory location. They also perform simple arithmetic such as multiplication by powers of 2^{+2} (left shift) and division by powers of 2^{-2}. 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.

Instructions	Meaning	Format	Operation	Flags affected
SAL	Shift Arithmetic left	SAL/SHL D, Count	Shift the D left by the number of bit positions	OF, CF
SHL	shift logical left		equal to count and fill the vacated bits positions on the right with zeros	
SHR	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
SAR	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



Rotate instruction position binary data by rotating the information in a register or memory location, either from one end to another or through the carry flag. They are often used to shift or position numbers.



5. 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:

A. Flag control instructions
B. Compare instruction
C. Jump instructions
D. String instruction

A. 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
```

B. 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. The operands can reside in a storage location in memory, a register within the MPU.

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

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 10011001 and that the source operand equals 00011011. Subtraction the source from the destination,
we get

```
10011001
00011011-
```

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

```
10011001
11100101+
01111110
```

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 through a carry out of bit 7 is generated; there is no carry from bit 6 to bit
4. There is an even number of 1s, therefore, this makes parity flag PF equal to 1.
5. Bit 7 is zero and therefore signs flag SF is at logic 0.
6. The result that is produced is nonzero, which makes zero flag ZF logic 0.
7. This is an overflow condition and the OF flag is set.

C. 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 34.

Instruction	Meaning	Format	Operation	Flag affected
JMP	Unconditional jump	JMP operand	Jump is to the address specified by operand	None

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).

Instruction	Meaning	Format	Operation	Flag 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
JC	Jump if carry
JNC	Jump if not carry
JCXZ	Jump if CX is zero
JE/JZ	Jump if equal / jump if zero
JNE/JNZ	Jump if not equal / jump if not zero
JNO	Jump if not overflow
JO	Jump if overflow
JP/JPE	Jump if parity / jump if parity Even
JNP/JPO	Jump if parity / jump if parity odd

JNS	Jump if not sign
JS	Jump if sign

D. 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 instruction are:

a. Move byte or work 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 and operations for one element of a string.

Move String

The instructions MOVS, 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 aword 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. When an INT occurs, the CPU suspends its execution of the current program and transfer control to an interrupt service routine (ISR). That will provide the requested service by the interrupt. When the ISR is completed, the original program resumes execution as if it were not interrupted.

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 that request interrupts (INTR and NMI), and one hardware (INTA) that acknowledges the interrupt requested through INTR. 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.

Interrupts vs Procedures

Although the behavior of interrupts is analogous to procedures, there are some basic differences that make interrupts almost indispensable. These differences are highlighted below:-

- Interrupts can be initiated by both software and hardware. However, procedures can be initiated only by software.
- Interrupts mechanism provides an efficient way to handle unanticipated events. For example, if the program goes into an infinite loop, ctrl-break could cause an interrupt to suspend the program execution.
- Interrupt service routines are memory resident while procedures are

loaded with application programs.

- Interrupts are identified by numbers while procedures are identified by names.

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.

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 =0008H, 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) instruction 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 non maskable interrupt, and the reset interrupt. The function of the external hardware, software, and non-maskable interrupt and the reset interrupts can be defined by the user. On the other hand, the internal and reset interrupts have dedicated system functions.

Software Interrupt (SW INT)

Software interrupts are initiated by executing the interrupt instruction INT in a program. Software interrupts are mainly used in accessing I/O devices such as the keyboard, printer, screen, disk drive. Software interrupts can be classified into system-defined or user-defined. System-defined software interrupts are those whose interrupt service routines are supported by BIOS and DOS. User-defined interrupts are those whose interrupt service routines are provided by the user. The format of the interrupt instruction is INT interrupt type, where INT Type is an integer number in the range 0-255, thus a total of 256 different types are possible. The following table shows which of the 256 interrupt types:-

Interrupts Type	Allocation
0-1Fh	BIOS Interrupts
20h-3Fh	DOS Interrupts
40h-7Fh	Reserved
80h-F0h	ROM Basic
F1h-FFh	User Defined

Hardware Interrupt (HW INT)

This type is generated by hardware devices to get the attention of the CPU. Is usually use by peripheral I/O devices such as KB to alter CPU that they require its attention.

For example, when a key is pressed the keyboard generates an interrupt causing the CPU to suspend its present activity and execute the keyboard interrupt service routine to process the key.

HW INT can be divided into Maskable and Non-Maskable (NMI). Maskable interrupts are initiated through the INTR while non-maskable interrupts are

initiated through the NMI. The non-maskable interrupt are serviced by the CPU immediately after completing the execution of the current instruction. One example of non maskable interrupts is the RAM parity error indicating memory malfunction.
However, Maskable interrupts can be delayed until execution reaches a convenient point. As an example, let us assume that the CPU is execution main program , an INT occur, as a result, the CPU suspend the main as soon as it finish the current instruction of main and then control is transfer to the ISR. If ISR has to be executed without any interrupt, the CPU can mask further INT until ISR is complete. Suppose that, while executing ISR another mskable INT occurs, service to this INT would have to wait until ISR is completed.
A NMI can be generated by applying an electronic signal on the NMI pin this INT is called NMI because the CPU always respond to this signal. In other word, this INT cannot be disabling under program control, the NMI cussed by INT2.
Most HW INT are Maskable type, and electronic signal should be applied to the INTR (interrupt request) input pin of 8086, 8086 recognize the INTR only if IF=1, thus this INT can be masked or disable by clear IF(IF=0).

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.

STACK

There are many situations in which a program needs to temporarily store information and then retrieve it in reverse order. One example of such a situation is saving and restoring the counters when using loops. On the 8086 the CX register and the loop instructions can be conveniently used to provide the counting, testing, but because the loop instructions are designed to use only the CX register a problem arises when loops are nested. However, if as shown in Figure (36) there were an efficient means of saving the loop counters in order and then restoring them by retrieving the last stored counter first, at least part of the problem would be alleviated.

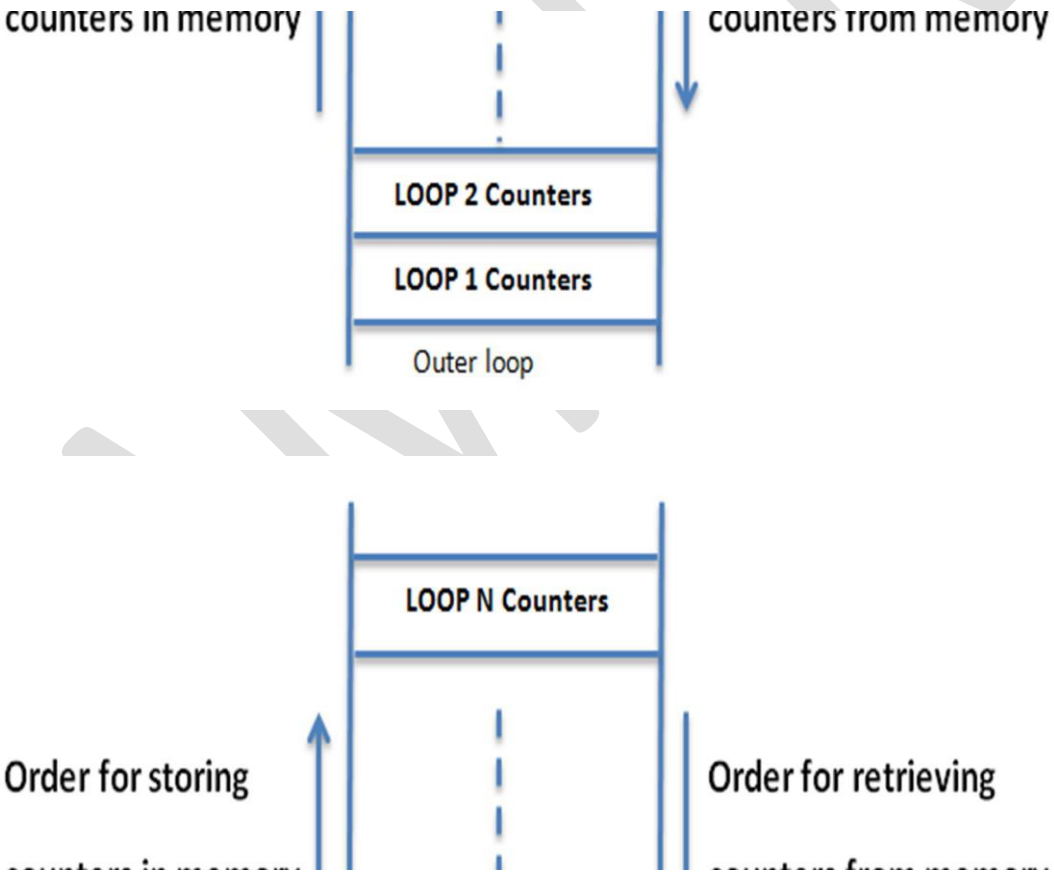


Figure ( ). Storing and retrieving counters for nested loop.

We can define the stack is an area of memory identified by the programmer for temporary storage of information. Most often a data storage situation such as the one mentioned above is resolved by designing into the computer last-inputfirst- output (LIFO) stack structure, which the stack itself is simply a part of memory.
The stack normally grows backwards into memory. In other words, the programmer defines the bottom of the stack and the stack grows up into reducing address range. Given that the stack grows backwards into memory, it is customary to place the bottom of the stack at the end of memory to keep it as far away from user programs as possible. In the 8086, the stack is defined by setting the SP (Stack Pointer) register. The Size of the stack is limited only by the available memory. The 8086 provides four instructions: PUSH, POP, CALL and RET for
storing information on the stack and retrieving it back.

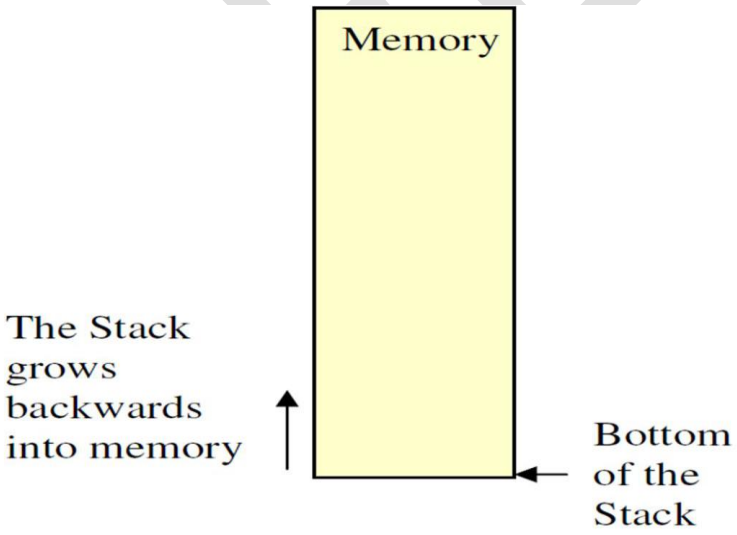


Figure ( ). The Stack.

Stack facilities normally involve the use of indirect addressing through a special register, the stack pointer, that is automatically decrement as items are put on the stack and incremented as they are retrieved. Putting something on the stack is called a push and taking it off is called a pop. The address of the last element pushed onto the stack is known as the top of the stack (TOS). Only words can be pushed or popped and the data cannot be immediate, but the pushed and pop instructions can use all of the other addressing modes.

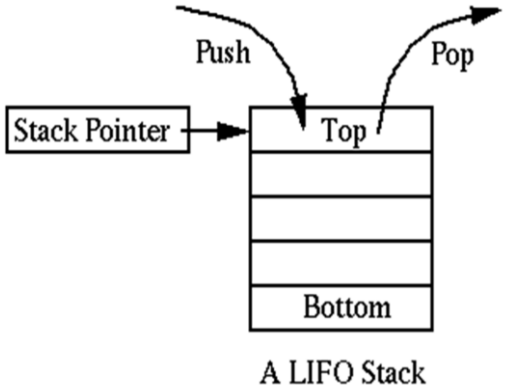


Figure ( ). The Stack Pointer.

During pushing, the stack operates in a “decrement then store” style. The stack pointer is decremented first, and then the information is placed on the stack. During popping, the stack operates in a “use then increment” style. The information is retrieved from the top of the stack and then the pointer is incremented. The SP pointer always points to “the top of the stack”. The order of PUSHs and POPs must be opposite of each other in order to retrieve information back into its original location.

```
PUSH A
PUSH B
  ...
  POP B
  POP A
```

Input and Output

Input & Output (I/O) devices provide the means by which a computer system can interact with the outside world. 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 (39) 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.

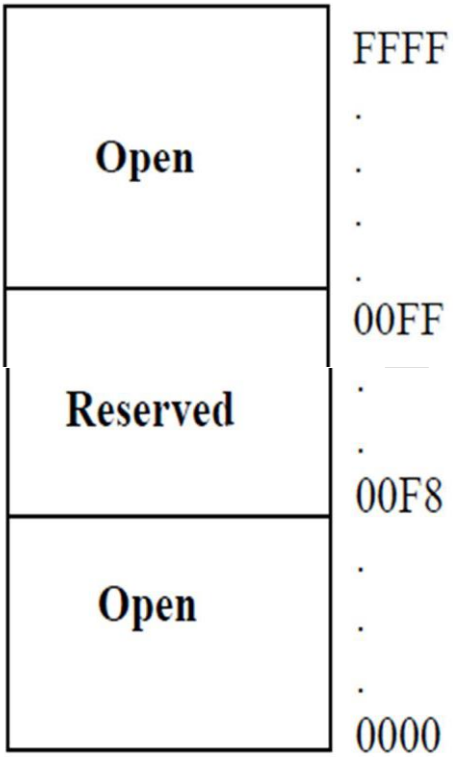


Figure ( ). I/O Address Space.

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

Where ACC = AL or AX

Example 1: write a sequence of instructions 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, 0FFH
```

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

```
OUT 0ABH, 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, 0B000H
```

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

```
MOV AL, 0FFH
```

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 AA_{16} and $A9_{16}$ respectively, and then output to a word wide output port at address $B000_{16}$. Write a sequence of instruction to perform this I/O operation:

Solution: we first read in a byte from the port at address AA_{16} into AL and move it to AH

IN AL, $0AA_H$
MOV AH, AL

The other byte can be read into AL

IN AL, $0A9_H$

To writhe 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 method of interfacing I/O to the MPU. In the isolated I/O scheme, the IN, OUT instruction transfer data between the MPU (ACC or memory) and the I/O device.

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

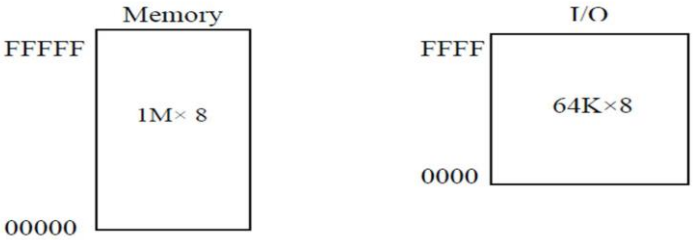


Figure ( ). Isolated I/O.

Memory- Map I/O

Unlike isolated I/O, memory mapped I/O does not use the IN or OUT instruction. Instead, it uses any instruction that transfer data between the MPU and memory. A memory mapped I/O device is treated as a memory location in 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 systems used as the I/O map. This reduced the amount of memory available to application. See Figure

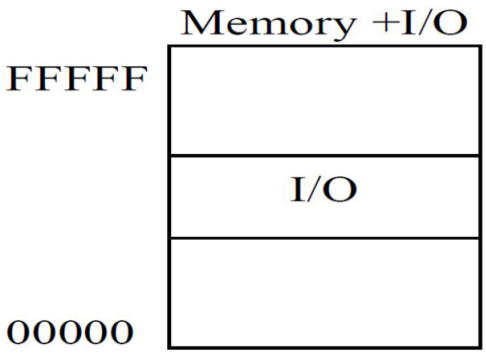


Figure ( ). Memory-Mapped I/O.