University of Technology
الجامعة التكنولوجية

Computer Science Department
قسم علوم الحاسوب

Microprocessor
المعالجات المايكروية

Dr. Khitam A. Salman
Assist. Lecturer Mohammed Thamer
Assist. Lecturer Sarab M. Taher
د. ختام عبدالنبي سلمان
م.م. محمد ثامر

cs.uotechnology.edu.iq

Microprocessors – 1’st course (Syllabus)

- Introduction to Microprocessor and Microcomputer system.
  - Microprocessor Architecture and Register Set.
  - System Buses
  - Memory types and physical addressing.
  - I/O devices
- Instruction Set and Format
- Addressing Modes
- Introduction to Assembly Programming Language.
  - Arithmetic and logical Instructions (Shift and Rotate).
  - Program Control (interrupt and subroutine call).

References:

1. Abel P., "IBM PC Assembly Language and Programming", 4th Edition, Prentice Hall, 1998..

2. Thorne M., "Computer Organization and Assembly Language Programming", 2nd Edition, Benjamin/Cummings, 1990.

Introduction to Microprocessors

Electronic and Logic Circuits

Modern devices contain two types of circuits:

Electronic circuits

The basic components of electronic circuits are transistors, resistors, capacitors, etc. Electronic circuits operate on a wide range of voltages such as (1V, 2.1V, 3.3V, 12V) positive or negative and deal with analog signals.

Logic Circuits

The basic component of digital circuits are logic gates such as AND, OR, NAND, NOR, XOR, XNOR, NOT... which deal with digital signals.

Integrated Circuits (IC)

Integrated circuits consist of logic and electronic circuits built on a single small block or chip of semiconductor that all work together to perform a specific task. The IC is easily breakable, so to be attached to a circuit board, it is often housed in a plastic package with metal pins.

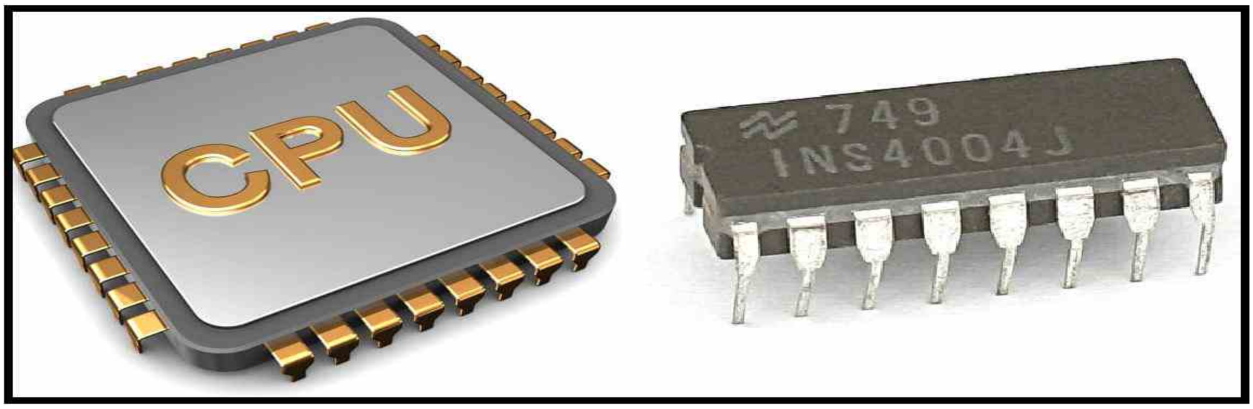


Figure 1 Integrated circuits of Microprocessor

Microprocessor

Computer's Central Processing Unit (CPU) built on a single Integrated Circuit (IC) is called a microprocessor. A digital computer with one microprocessor which acts as a CPU is called microcomputer. It is a programmable, multipurpose, clock-driven, register-based electronic device that reads binary instructions from a storage device called memory, accepts binary data as input and processes data according to those instructions and provides results as output.

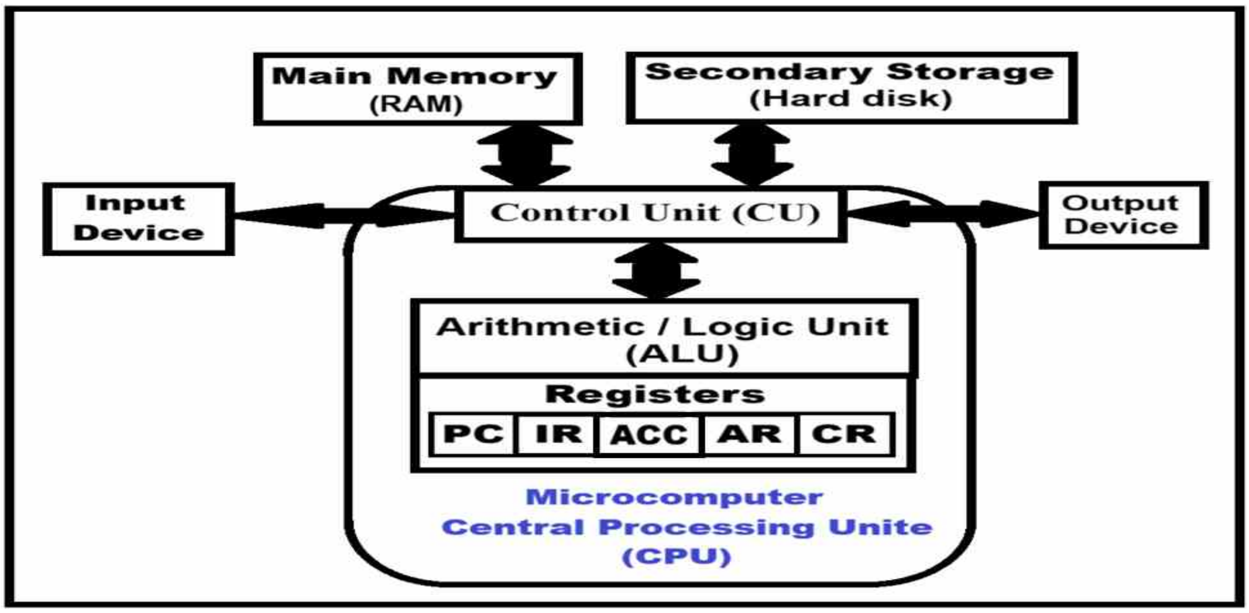

Main Memory (RAM)
Secondary Storage (Hard disk)
Input Device
Control Unit (CU)
Output Device
Arithmetic / Logic Unit (ALU)
Registers
PC
IR
ACC
AR
CR
Microcomputer Central Processing Unite (CPU)

Figure 2 Main Component of Microprocessor

Central processing unit (CPU)

The CPU supervises and controls all other computer units, transfers data to and from these units, and performs the arithmetic and logical operations necessary to transform data into meaningful information. It called "Processor" or "Microprocessor" in personal computer. It is divided into three parts:

1- Arithmetic and Logic unit (ALU).
2- Control unit (CU).
3- Register.

1. Arithmetic and Logic unit (ALU).

Perform the processing of data including arithmetic operations such as addition, subtraction, multiplication, division and logic operations including comparison (ex. A<B) and sorting.

2. Control Unit (CU).

The control unit coordinates the operation of the entire computer system automatically, and acts as a central nervous system that sends control signals to other computer units. The operations it performs are:

1- Control of input and output devices.
2- Sending and retrieving information to and from memory's (primary and secondary memory).

3- Routing of information between the main memory (RAM) and the arithmetic and logic unit (ALU).
4- Direct and coordinates all units of the computer to execute program steps.

3. Registers

Registers are a type of computer memory used to quickly accept, store, and transfer data and instructions that are being used immediately by the CPU. The most important registers are:

1- Instruction Register (IR): It contains the instruction being executed.
2- Program Counter Register (PC): It contains the address of the next instruction to be executed.
3- Address Register (AR): holds the address of memory location.
4- Data Register (DR): Holds data that is being transferred to or from memory.
5- Accumulator Register (ACC): Where intermediate arithmetic and logic results are stored.

The basic operations performed by microprocessor

A microprocessor does three basic things.

1- Using its ALU to perform arithmetic and logical operations. Modern microprocessors contain complete floating-point processors that can perform extremely sophisticated operations on large floating-point numbers.
2- A microprocessor can move data from one memory location to another.
3- A microprocessor can make decisions and jump to a new set of instructions based on those decisions.

Instructions Fetch-Execute Cycle

The microprocessor follows the following sequence to execute instructions.

1. Initially: the microprocessor loads the program instructions into main memory.
2. **Fetch**: The microprocessor fetches those instructions from the memory.
3. **Decode**: Separate the operation from the operands and replace the variables with their real values stored in main memory in preparation for execution.
4. **Execute**: Executes these instructions until the end of the program (until all instructions are performed).
5. Later: it sends the result in binary to the output device.

Between these processes the register stores the temporarily data and ALU performs the computing functions.

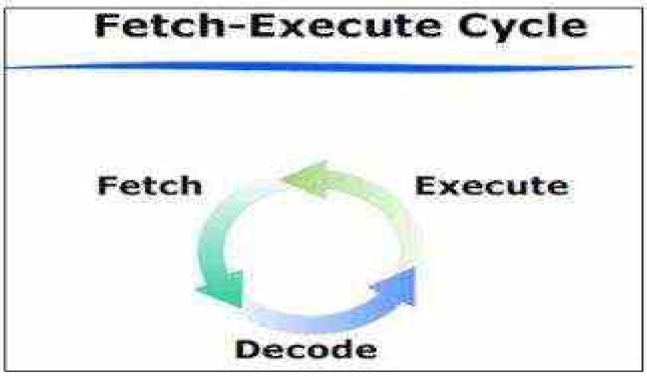


Figure 3 Sequence of Execute Instructions Cycle

<u>Q: How does the CPU Fetch and execute program instructions?</u>

1. **Fetch**: Fetch the instruction from memory to IR.
2. Change the address of Program Counter PC to the next instruction.
3. **Decode**: Determine the type of the instruction to be execute.
   - If the instruction uses data in memory, use the address in the AR register to fetch the data.
   - Fetch the data into DR register (Data Register).
4. **Execute**: Execute the instruction.
5. Store the result in the ACC register or in a proper place or send it to the output device.
6. Go to step 1 to Fetch the next instruction whose address in PC register.

List of Terms Used in a Microprocessor

Here is a list of some of the used terms in a microprocessor.

- **<u>Instruction Set:</u>** It is the set of instructions that the microprocessor can understand (Ex: MOV, ADD, SUB...). The instruction set acts as an interface between the software and hardware.
- **<u>Bus:</u>** The bus is used for the transmission of data, address and control information. This transmission occurs in different elements of the microprocessor. The bus in this is basically of three types which are **data bus, address bus** and **control bus**.

  This microprocessor has:

✓ A data bus (that may be 8, 16, 32 or 64 bits wide) that can send data to memory or receive data from memory.

✓ An address bus (that may be 8, 16, 32 or 64 bits wide) that sends an address to memory.

✓An RD (read) and WR (write) line to tell the memory whether it should set or get the addressed location.

✓ A clock line that lets a clock pulse sequence the processor

✓ A reset line that resets the program counter to zero (or whatever) and restarts execution

- **Word Length:** It depends upon the (number of bits) of internal data bus, registers, ALU, etc. A 16-bit microprocessor can process 16-bit data at a time. The word length ranges from 4 bits to 64 bits depending on the type of the microcomputer. A processor with longer word length is more powerful and can process data at a faster speed as compared to a processor with shorter word length.
  NOT: The power of the given microprocessor is measured in terms of bits.
- **Clock Speed:** It determines the number of operations per second the processor can perform. It is expressed in megahertz (MHz) or gigahertz (GHz). It is also known as Clock Rate.
- **Data Types:** The microprocessor has multiple data type formats like Decimal, Hexadecimal, Binary, signed and unsigned numbers.

Evolution of Microprocessors (

We can categorize the microprocessor according to the generations or according to the size of the microprocessor. The following table shows this categorization.

Table 1 Evolution of Microprocessor

Generation	Year	Size	Example	Features
1st generation	1971-1972	4 bits	4004	Basic operation and it has limited memory and the control unit executes fetch, decode and execute instructions
2nd generation	1973	8 bits	Intel 8008 & 8080	Improved processing speed, better instruction sets

Generation	Year	Size	Example	Features
3rd generation	1978	16 bits	Intel 8086, Zilog Z800 and 80286	Its performance like minicomputers. Enhanced memory access, pipelining introduced
4th generation	1985-1994	32 bits	Intel 80386, 80486, Motorola 68020	Integrated FPU, cache memory, faster processing
5th generation	195-2005	64 bits	Intel Pentium, AMD Athlon, PowerPC G4	Super scalar architecture, advanced power management
6th generation	2006 up now	Multi-core	Intel Core i3/i5/i7/i9, AMD Ryzen, Apple M-series	Multi-core processing, AI acceleration, high energy efficiency

Introduction to Microcomputers

Microcomputer is a digital computer with one microprocessor which acts as a CPU.

Microcomputer: small computers, also called personal computers (PC), can fit next to a desk or on a desktop, or can be carried around. They are either standalone machines or are connected to a computer network such as a local area network ***LAN***. LAN connects, usually by special cable, a group of desktop PCs and other devices, such as printers, in an office or a building.

Computers used nowadays can be known as general purpose machines or special purpose machines.

General purpose machines are machines that built with no specific application in mind, but rather are capable of performing computation needed by different applications.

Special purpose machines: are machines that to serve specific applications.

1. ***Desktop PCs***: are those in which the case or main housing sits on a desk, with keyboard in front and monitor (screen) often on top.
2. ***Tower PCs***: are those Microcomputer in which the case sits as a "tower," often on the floor beside a desk, thus freeing up desk surface space.

3. ***Laptop computers (notebook computers):*** are lightweight portable computers with built-in monitor, keyboard, hard-disk drive, battery, and AC adapter that can be plugged into an electrical outlet; their weight anywhere from 1.8 to 9 pounds.
4. ***Personal digital assistants (PDAs)*** (handheld computers or palmtops) combine personal organization tools-schedule planners, address books, to-do lists. Some are able to send e-mail and faxes. Some PDAs have touch-sensitive screens. Some also connect to desktop computers for sending or receiving information.
5. ***Microcontrollers (tiny computers***: Microcontrollers, also called embedded computers, are the tiny, specialized microprocessors installed in "smart" appliances and automobiles. These microcontrollers enable PDAs, microwave ovens, for example to store data about how long to cook your potatoes and at what temperature.

The Microprocessor-Based Personal Computer System

Machines that once filled large areas have been reduced to small desktop computer systems. Companies such as DEC (Digital Equipment Corporation now owned by Hewlett-Packard Company) have stopped producing mainframe computer systems in order to concentrate their resources on microprocessor-based computer systems. PC consists of three main blocks which are: microprocessor, memory system and I/O systems. These blocks are interconnected by buses. A bus is a set of common connections that carry the same type of information. Figure 4 shows the general block diagram of the microcomputer

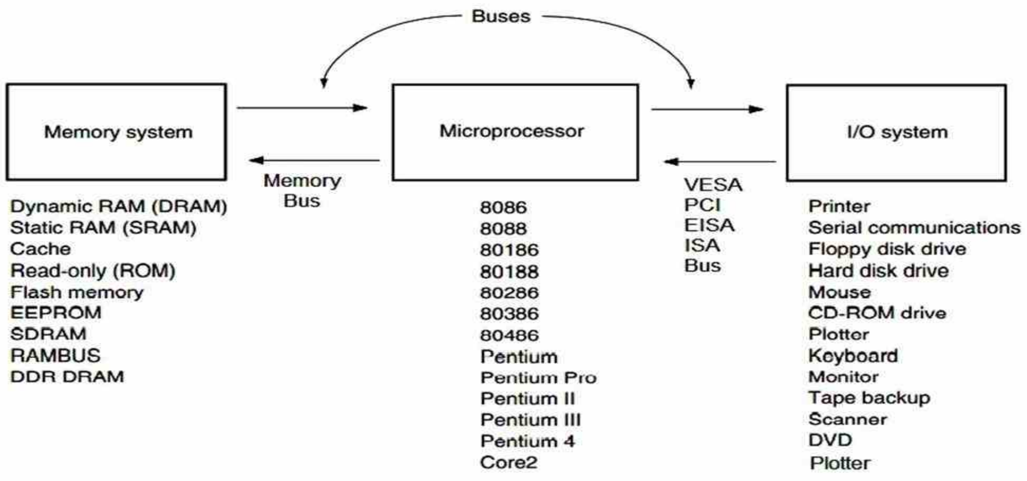


Figure 4 Microcomputer component

Memory system

Memory unit in microprocessor is used to store information such as numbers, characters and so on. Storing information means that the memory has the ability to hold this information for processing or for later use. Programs that define how the computer work are also stored in the memory.

Memory can be divided into: ***primary storge*** that is used for temporary storage and it is normally of small size and ***secondary storage*** which is used for long term storage.

The primary storage is further divided into ***Read Only Memory (ROM)*** and ***Random Access Memory (RAM).*** The information stored in ROM are nonvolatile, that is the information is not lost when the power turned off. On the other hand, the information stored in RAM are volatile can be modified.

Input/Output System

The microcomputer system contains input and output (I/O) devices that allow the system to communicate with the external environment. Keyboard, mouse, joystick and microphone are examples of the input devices. On the other hand the most used output devices are printers, displays and speakers.

Performance measure

The clock synchronizes the internal operations of the CPU with other components in the system. CPU speed is determined by a clock cycle, which is the time between two rising edges of a periodic clock signal. Figure 5 shows the clock cycle time

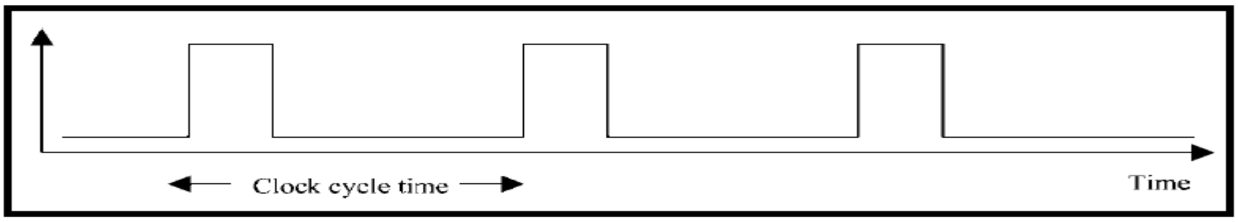


Figure 5 Clock signal

The higher number of pulses per second, the faster the processor. The clock speed is measured in Hz, often either MHz (megahertz) or GHz (gigahertz). For example, a 4 GHz processor performs 4,000,000,000 clock cycles per second.

Instructions Per Cycle (IPC): it measures how many instructions a CPU is capable of executing in a single clock. The performance of a processor can be measured by calculating the average number of ***Clock cycles Per Instruction (CPI)***.
The instruction set of a given machine consists of a number of instruction categories: ***ALU*** (simple assignment and arithmetic and logic instructions), ***load and store, branch*** and others In case that CPI for each instruction category is known then the overall CPI can be computed as:

$$CPI = \frac{\sum_{i=1}^{n} CPI_i \times I_i}{Instruction\ count}$$

Where I_i is the number of times an instruction of type i is executed in the program and CPI_i is the average number of clock cycles needed to execute such instruction.
Example:
Use the CPI to measure the performance of a computer, which records the following metrics in table 2when you run a set of benchmarking programs.

Table 2 Machine inforation

Instruction Category	Percentage of occurrence	No. of cycles per instruction
ALU	38	1
Load & store	15	3
Branch	42	4
Others	5	5

<u>Solution:</u>

$$CPI = \frac{\sum_{i=1}^{n} CPI_i \times I_i}{Instruction\ count}$$

Instruction count = 38+15+42+5 =100

$$CPI = \frac{38 \times 1 + 15 \times 3 + \ 42 \times 4 + \ 5 \times 5}{100} = 2.76$$

Million Instructions-Per-Second (MIPS)

MIPS is a measure of a processor's speed, providing a standard for representing the number of instructions that a CPU can process in one second. The number is meant to indicate how well a computer performs and how much work it can do.

$$MIPS = \frac{clock\ rate}{CPI \times 10^6}$$

Example:

Use MIPS to measure the performance of a computer for which the following metrics are recorded when running a set of benchmarking programs in table3. Assume that the clock rate of the CPU is 200 MHz

Table 3 Machine information

Instruction Category	Percentage of occurrence	No. of cycles per instruction
ALU	35	1
Load & store	30	2
Branch	15	3
Others	20	5

Solution:

$$CPI = \frac{\sum_{i=1}^{n} CPI_i \times I_i}{Instruction\ count}$$

Instruction count = 35+30+15+20 =100

$$CPI = \frac{35 \times 1 + 30 \times 2 + 15 \times 3 + 20 \times 5}{100} = 2.4$$

$$MIPS = \frac{clock\ rate}{CPI \times 10^6}$$

$$MIPS = \frac{200\ MHz}{2.4 \times 10^6}$$

Note:1 MHz = 1 000 000 hertz = 10^6 hertz

$$MIPS = \frac{200\ M \times 10^6}{2.4 \times 10^6} = 83.67$$

The MIPS scale can be used to compare computers and determine which one has better performance. For example, a computer that can process 12,000 MIPS should be able to outperform one that processes 10,000 MIPS.

Example3:

What is the MIPS for the devices (A, B) shown in table 4? assuming the clock rate is 200 MHz

Table 4 Machine A and B information

Machine A			Machine B		
Instruction Category	**Percentage of occurrence**	**No. of cycles per instruction**	**Instruction Category**	**Percentage of occurrence**	**No. of cycles per instruction**
ALU	38	1	ALU	35	1
Load & store	15	3	Load & store	30	2
Branch	42	4	Branch	15	3
Others	5	5	Others	20	5

Solution:

$$CPI_a = \frac{\sum_{i=1}^{n} CPI_i \times I_i}{Instruction\ count} = \frac{38 \times 1 + 15 \times 3 + 42 \times 4 + 5 \times 5}{100} = 2.76$$

$$MIPS_a = \frac{clock\ rate}{CPI \times 10^6} = \frac{200 \times 10^6}{2.76 \times 10^6} = 70.24$$

$$CPI_b = \frac{\sum_{i=1}^{n} CPI_i \times I_i}{Instruction\ count} = \frac{35 \times 1 + 30 \times 2 + 15 \times 3 + 20 \times 5}{100} = 2.4$$

$$MIPS_b = \frac{clock\ rate}{CPI \times 10^6} = \frac{200 \times 10^6}{2.4 \times 10^6} = 83.67 =$$

Thus: $MIPS_b(83.67) > MIPS_a(70.24)$

Microprocessor Architecture

8086 Microprocessor is an enhanced version of 8085 Microprocessor that was designed by Intel in 1976. This 16 Bit Microprocessor have 20-bit address lines and16-bit data lines that provides up to 1MB storage. It consists of powerful instruction set, which provides operations like multiplication and division easily. It supports two modes of operation ***Maximum mode and Minimum mode***. Maximum mode is suitable for system having multiple processors and Minimum mode is suitable for system having a single processor.

Features of 8086 Microprocessors

The most prominent features of an 8086 microprocessor are as follows:

- It is a 16-bit Microprocessor. Its ALU, internal registers work within 16-bit binary word.
- It has a 20-bit address bus which can access up to $2^{20} = 1\ MB$ memory locations.
- It has 16-bit data bus. It can read or write to a memory/port 16 bits or bit at a time.
- It provides 14,16-bit registers.
- The frequency range of the 8886 microprocessor is 6-10 Hz.
- It has multiplexed address bus and data bus D_0-D_{15} and A_{17}-A_{19}
- It has an instruction queue, which is capable of storing six instruction bytes from the memory resulting in faster processing.
- It can prefetch up to 6 instruction bytes from memory and queues them in order to speed up the instruction execution.
- It requires +5V power supply.

Comparison between 8085 & 8086 Microprocessor

- **Size** – 8085 is 8-bit microprocessor, whereas 8086 is 16-bit microprocessor.
- **Address Bus** – 8085 has 16-bit address bus while 8086 has 20-bit address bus.
- **Memory** – 8085 can access up to 64Kb, whereas 8086 can access up to 1 Mb of memory.
- **Instruction** – 8085 doesn't have an instruction queue, whereas 8086 has an instruction queue.

- **Pipelining** – 8085 doesn't support a pipelined architecture while 8086 supports a pipelined architecture.
- **I/O** – 8085 can address 2^8 = 256 I/O's, whereas 8086 can access 2^16 = 65,536 I/O's.
- **Cost** – The cost of 8085 is low whereas that of 8086 is high.
- **Operation mode**: 8085 has single operation mode while 8086 has ***minimum mode*** which suits systems that have one processor and maximum mode that suits for multiprocessors systems

Microprocessor 8086 Architecture

The internal architecture of 8086 family of microprocessor has changed from the original 8086 to the 80386. All members of 8086 family employ the parallel processing. That is, they are implemented with several simultaneously operating processing units. Each unit has a dedicated function and they operate at the same time. The more parallel processing the higher microprocessor performance.

8086 microprocessors contain two processing units: the ***Bus Interface Unit (BIU) and the Execution Unit (EU)***. Figure (6) depicts the architecture diagram of the microprocessor 8086.

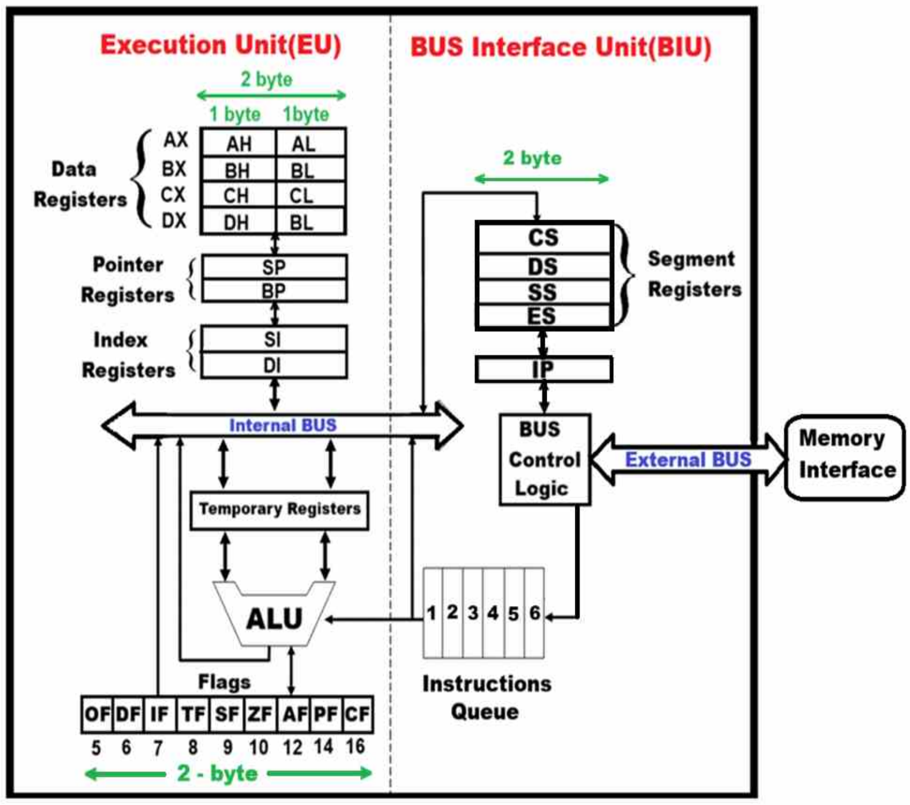


Figure 6 Microprocessor 8086 Architecture

Bus Interface Unit (BIU)

It provides a full 16-bit bidirectional data bus and 20-bit address bus. This unit fetches a set of program instructions from main memory and queuing them in an instruction queue to be executed later by the execution unit. This unit acts as a control unit that is connected to the ***memory and external computer parts*** via the ***external bus*** and is connected to the ***Execution Unit (EU) via the internal bus***. It is responsible for transmitting data, addresses and control signals on these buses. BIU performs the following operations:

1- Fetches instruction from main memory.
2- Supports instruction queuing.
3- Sends address of the memory or I/O.
4- Reads data from port/memory.
5- Writes data into port/memory.

The BIU uses a mechanism known as ***instruction stream queue*** to implement pipeline architecture. The ***Instruction queue***, as mentioned before can store up to 6-byte instruction to be executed by the EU. When the EU execute the current instruction, the next instruction to be executed will be ready in the instruction queue. This lead to increase the execution speed.

Execution Unit (EU)

The execution unit is responsible for decoding and executing all instructions. It consists of: ***ALU, status and control flag, general-purpose registers and temporary registers***.

The EU extract the instruction from the top of the instruction queue in the BIU, decode them, generates operands, if necessary, passes them to the BIU and requests it to perform the read or write by cycles to memory or to I/O and perform the operation specified by the instruction on the operands.

During the execution of the instruction, the EU tests the status and control flags and updates them based on the result of executing the instruction.

Microprocessor 8086 - Pin Configuration

8086 was the first 16-bit microprocessor available in 40-pin Dual Inline Package (DIP) chip. Let us now discuss in detail the pin configuration of an 8086 Microprocessor.

8086 Pin Diagram

The pin diagram of 8086 microprocessors is shown in Figure 7. Some of these signals are explained as follows

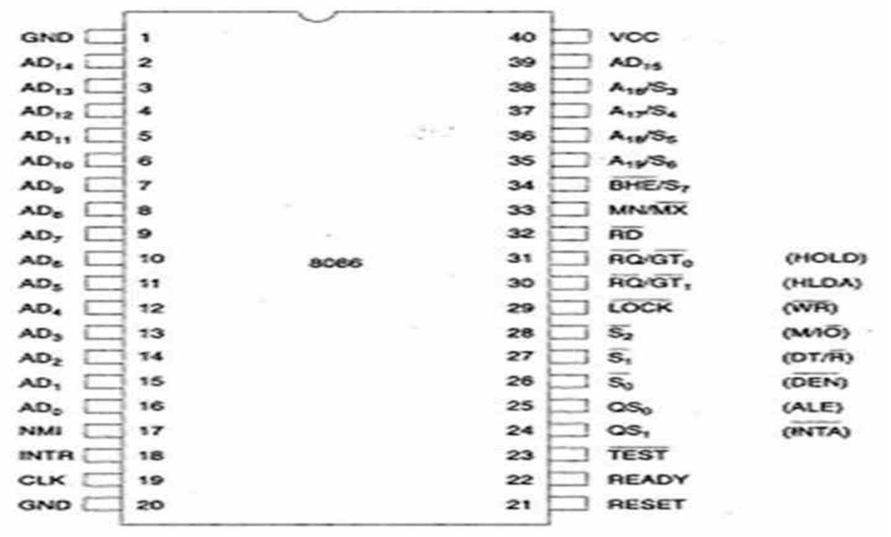


Figure 7 8086 Pin Diagram

1. **MN/*MX'*:** It stands for ***Minimum/Maximum*** and is available at pin 33. It indicates what mode the processor is to operate in; when it is high, it works in the minimum mode and vice-averse. 8086 supports two modes of operation Maximum mode and Minimum mode. Maximum mode is suitable for system having multiple processors and Minimum mode is suitable for system having a single processor.
2. **Address/data bus**: **AD0-AD15**. These are 16 address/data bus. AD0-AD7 carries low order byte data and AD8-AD15 carries higher order byte data. During the first clock cycle, it carries 16-bit address and after that it carries 16-bit data.
3. **Address/status bus: A16-A19/S3-S6**. These are the 4 address/status buses. During the first clock cycle, it carries 4-bit address and later it carries status signals.
4. **Read** (**RD**): It is available at pin 32 and is used to read signal for Read operation.
5. **Ready**: It is available at pin 22. It is an acknowledgement signal from I/O devices that data is transferred. It is an active high signal. When it is high, it indicates that the device is ready to transfer data. When it is low, it indicates wait state.

6. **RESET:** It is available at pin 21 and is used to restart the execution. It causes the processor to immediately terminate its present activity. This signal is active high for the first 4 clock cycles to RESET the microprocessor.
7. **M/IO**: This signal is used to distinguish between ***memory and I/O operations***. When it is high, it indicates I/O operation and when it is low indicating the memory operation. It is available at pin 28.
8. **WR:** It stands for ***write signal*** and is available at pin 29. It is used to write the data into the memory or the output device depending on the status of M/IO signal.
9. **QS_1 and QS_0:** These are queue status signals and are available at pin 24 and 25. These signals provide the status of **instruction queue**. Their conditions are shown in the following table

QS_0	QS_1	Status
0	0	No operation
0	1	First byte of opcode from the queue
1	0	Empty the queue
1	1	Subsequent byte from the queue

10. **S_0, S_1, S_2**: These are the status signals that provide the status of operation, which is used by the Bus Controller 8288 to generate memory and I/O control signals. These are available at pin 26, 27, and 28. Following is the table showing their status:

S_2	S_1	S_0	Status
0	0	0	Interrupt acknowledgement
0	0	1	I/O Read
0	1	0	I/O Write
0	1	1	Halt
1	0	0	Opcode fetch

1	0	1	Memory read
1	1	0	Memory write
1	1	1	Passive

11. **LOCK:** When this signal is active, it indicates to the other processors not to ask the CPU to leave the system bus. It is activated using the LOCK prefix on any instruction and is available at pin 29.

Registers Set

Registers are fast memory locations within the CPU that are used to create and store the results of CPU operations and other calculations. Different computers have different register sets. They differ in the number of registers, register types, and the length of each register. They also differ in the usage of each register.
Registers can be ***general-purpose registers*** used for multiple purposes and assigned to a variety of functions by the programmer. ***Special-purpose registers*** are restricted to only specific functions. In some cases, some registers are used only to hold data and cannot be used in the calculations of operand addresses.
Data registers' length must be long enough to hold values of most data types. Some machines allow two contiguous registers to hold double-length values.
The 8086 registers are classified into ***four*** types as follow

1. Data Registers (General Purpose Registers).
2. Pointers and Index Registers.
3. Flag Register (Status Register).
4. Segment Registers.

1. Data Registers (General Purpose Registers).

There are four registers that are generally used to store both data and addresses. All data registers can be used for arithmetic and logic operations and data movement. each of them has a special use. Each of these registers is 2-bytes (16-bit) long and each is divided into two smaller registers in 1-byte (8-bit) long as follows:

1.1. AX, AH, AL registers

Also known as ***the accumulator register***, it is used ***to transfer data, access I/O ports, and arithmetic and logical instructions***. The ***AX*** register is **2 bytes long** and can be divided into two ***one-byte registers AH*** (A High) and ***AL*** *(A Low)*. The AH register is specially used to store the O/I device number to be handled before using the *INT 21h* interrupt instruction.

1.2. BX, BH, BL registers

Also known as a ***Base index register,*** BX register usually contains a data pointer used for based. It is used ***to hold the address of a procedure or variable***. The **BX** register is **2 bytes long** and can be divided into two **one-byte registers BH** (B High) and **BL** (B Low). The BX register is specially used to store the starting base address of the memory area within the data segment.

1.3. CX, CH, CL registers

Also known as ***a Count register***, used in ***Loop, shift/rotate instructions and as a counter in string manipulation.*** The **CX** register is **2 bytes long** and can be divided into two **one-byte registers CH** (C High) and **CL** (C Low). The CX register is specially used in loop instruction to store the loop counter.

1.4. DX, DH, DL registers

Also known as ***Data register***. It is used as ***a port number in I/O operations. It is also used in multiplication and division***. The **DX** register is **2 bytes long** and can be divided into two **one-byte registers DH** (D High) and **DL** (D Low). The DX register is specially used to store the address of the string that will be processed before the INT 21 interrupt instruction is used.

2. Pointers and Index Registers

These 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 pointers will always store some address or memory location. In 8086 Microprocessor, they usually store the offset through which the actual address is calculated.

2.1. Stack Pointer (SP)

The Stack Pointer points at the current top value of the Stack. Like the BP, it also acts as an offset to the Stack Segment (SS).

2.2. Base Pointer (BP)

The Base pointer stores the base address of the memory. Also, it acts as an offset for Stack Segment (SS).

2.3. Source Index (SI)

It stores the offset address of the source in string manipulation instructions.

2.4. Destination Index (DI)

It stores the offset address of the Destination in string manipulation instructions.

3. Flag Register(Status Register)

It is a 16-bit register, only 9 bits are used as flags and the rest are ignored. These 9 flags provide information about the state of the processor after executing an instruction. The flag bit is changed to 0 or 1 depending on the value of the result after arithmetic or logical operations. The flags used to determine the behavior of many conditional jump and branch instructions. The 9 flags are divided into two groups, 6 for ***Conditional Flags*** and 3 for ***Control Flags***.

3.1. *Conditional Flags*

It represents the result of the last arithmetic or logical instruction executed. Following is the list of conditional flags:

a) ***Carry Flag (CF)***

The carry flag (commonly referred to as the C flag) is a single bit in the flag register used to indicate when to Carry/Borrow from the Most Significant Bit (MSB) during mathematical operations. Figure 8 illustrate it.

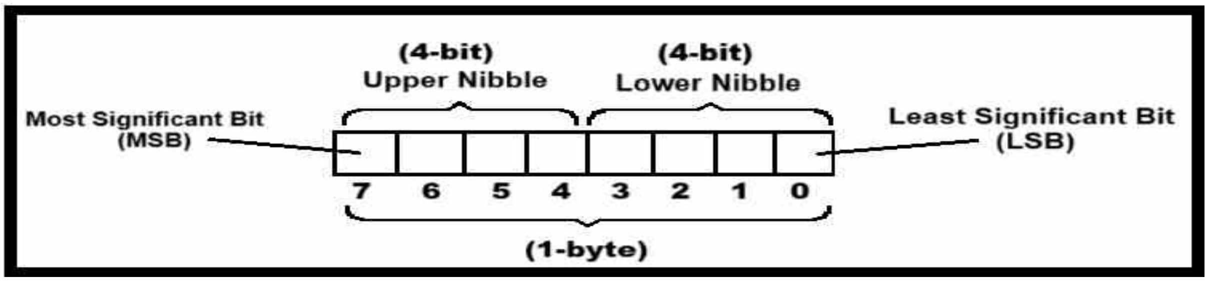


Figure 8 illustrates the terms most significant bit (MSB), least significant bit (LSB), Upper Nibble, Lower nibble on one byte

b) Auxiliary carry Flag (AF)

- This flag is used in BCD (Binary-coded Decimal) operations.
- The status of this flag is updated for every arithmetic or logical operation performed by ALU.
- This flag is set to one if there is a Carry/Borrow for the lower nibble (4 bits) in binary representation, else it is set to zero.

c) ***Parity Flag (PF)***

This flag is set to 1 when there is an even number of one bits in the result, and to 0 when there is an odd number of one bits. Even if the result is a word only 8 low bits are analyzed.

d) Zero Flag (ZF)

This flag is set to 1 when the result of the arithmetic or the logical operation is zero else it is set to 0.

e) ***Sign Flag (SF)***
This flag holds the sign of the result, i.e. when the result of the operation is negative, then the sign flag is set to 1 else set to 0.

f) ***Overflow Flag (OF)***
It sets to 1 when there is a signed overflow. For example, when you add signed bytes 100 + 50 (result is not in range -128...127).

3.2. Control Flags

The control flags control the operations of the execution unit. These flags are shown in figure 9. The control flags are as follows:

a) ***Trap Flag (TF)***
It is used for single step control and allows the user to execute one instruction at a time for debugging. If it is set, then the program can be run in a single step mode.

b) Interrupt Flag (IF)
It is an interrupt enable/disable flag, i.e. it is used to allow/prevent the interruption of a program. It is set to 1 for interrupt enabled condition and set to 0 for interrupt disabled condition.

c) ***Direction Flag (DF)***
This flag is used by some instructions to process data string, when this flag is set to 0 - the processing is done forward, when this flag is set to 1 the processing is done backward.

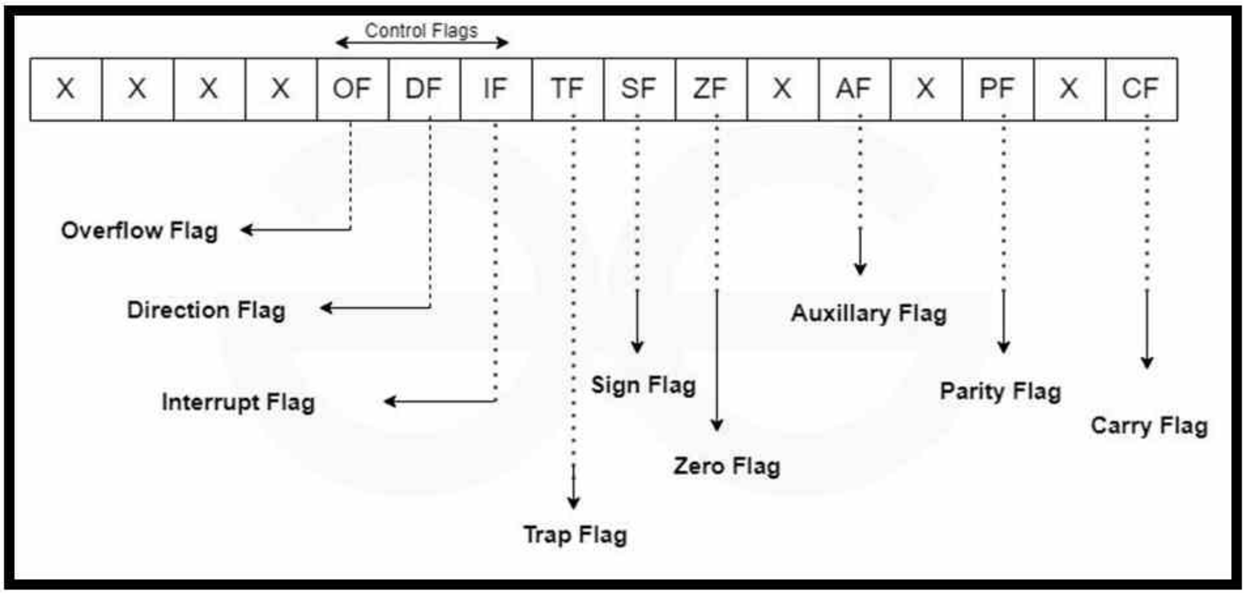


Figure 9 Flag Register

4. ***Segment Registers***

The processor can divide the main memory into four Segments (Sections) which are: ***Code Segment, Data Segment, Stack Segment, and Extra Segment***. For this reason, the BIU has four segments' registers, each 2 bytes long, and are used to store the addresses of the beginnings of the four Segment. The processor uses the addresses stored in these registers to determine the physical address and access any memory location in those segments figure 10 shows the segment registers. These registers are:

4.1.Code segment register (CS)

The CS register holds the base address of the code segment area in the memory. In this area of memory, the executable program instructions are stored.

4.2.Data Segment register (DS).

The DS register holds the base address of the data segment area in memory. In this area of memory, the stored data, variables and arrays are declared in the program.

4.3.Extra Segment register (ES)

The ES register holds the base address of the extra segment area in memory. In this area of memory more data, variables and arrays declared in the program are stored.

4.4.Stack Segment register (SS)

The SS register holds the base address of the Stack segment area in memory. This area of memory is used as a stack where data is stored according to the "first in, last out" (FILO) principle.

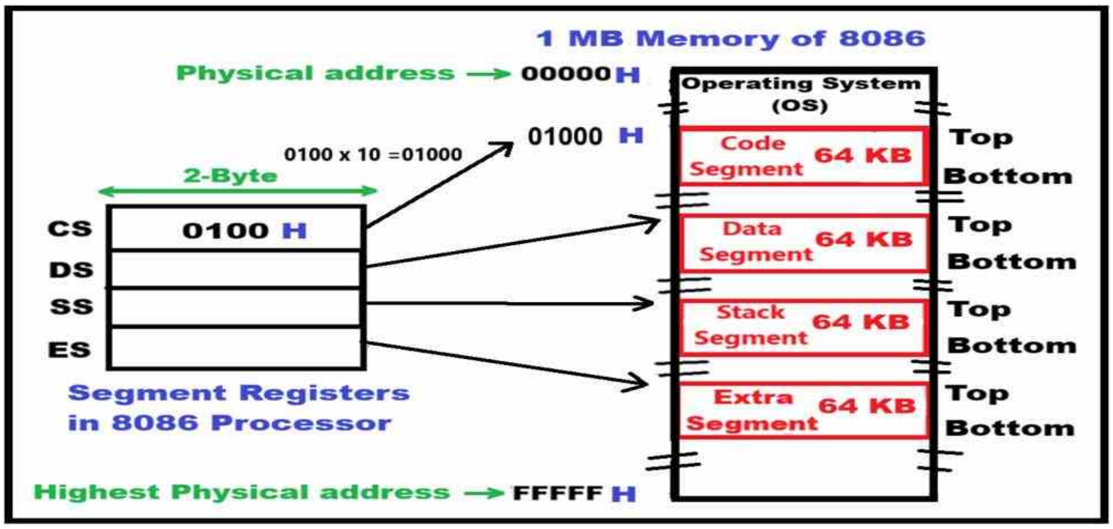


Figure 10 Segment Registers and 1MB Memory

System Buses

Bus structure

The system bus connects the CPU (Processor), memory and peripherals devices (input/output device or secondary memory) with each other. The bus system carries data, address and control information. The speed of the system bus is the part of performance of computer system figure 11 shows the system buses.

Note: The components of the computer system communicate with each other and with the outside world **through system bus. The processor connects to memory and peripheral devices by bus system.**

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 Buses

1. **Address Buss (AB):** is unidirectional (one direction) because address flow in one direction from processor to memory or from processor to input/ output devices. The width of AB determines the amount of physical memory addressable by the processor.
2. **Data Bus (DB):** Is bidirectional (two directions) because allow data to transfer between the processor (CPU) and memory (RAM). the width of DB indicates the size of the data transferred between the processor and memory or I/O device.
3. **Control Bus (CB):** is bidirectional (two directions) used by CPU for communicating with other devices within the computer. It carries control signals from CPU. The typical control signals include memory read, memory write, I/O read, I/O write, bus request. These signals indicate the type of action taking place in the computer system.

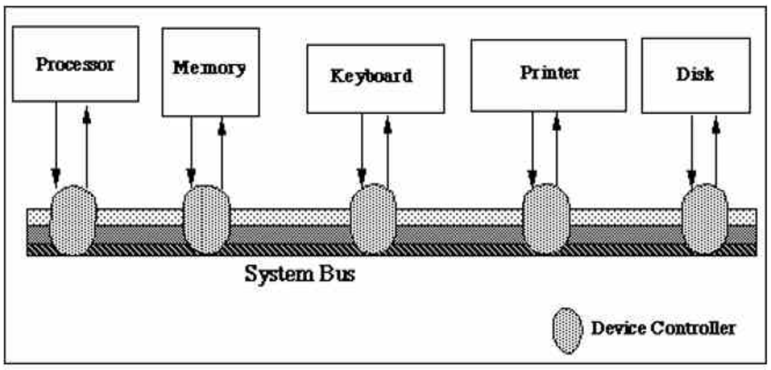


Figure 11 The system bus

Memory (Main Memory or Primary memory (RAM))

The primary memory (RAM) is a temporary storage area. It holds the data and instruction that the CPU needs. The memory of a computer system consists of tiny electronics witches, 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 12. This is referred to as memory address of the byte. Such memory is called **byte addressable memory.** The memory address space of a system is determined by the address bus width of the CPU used in the system.

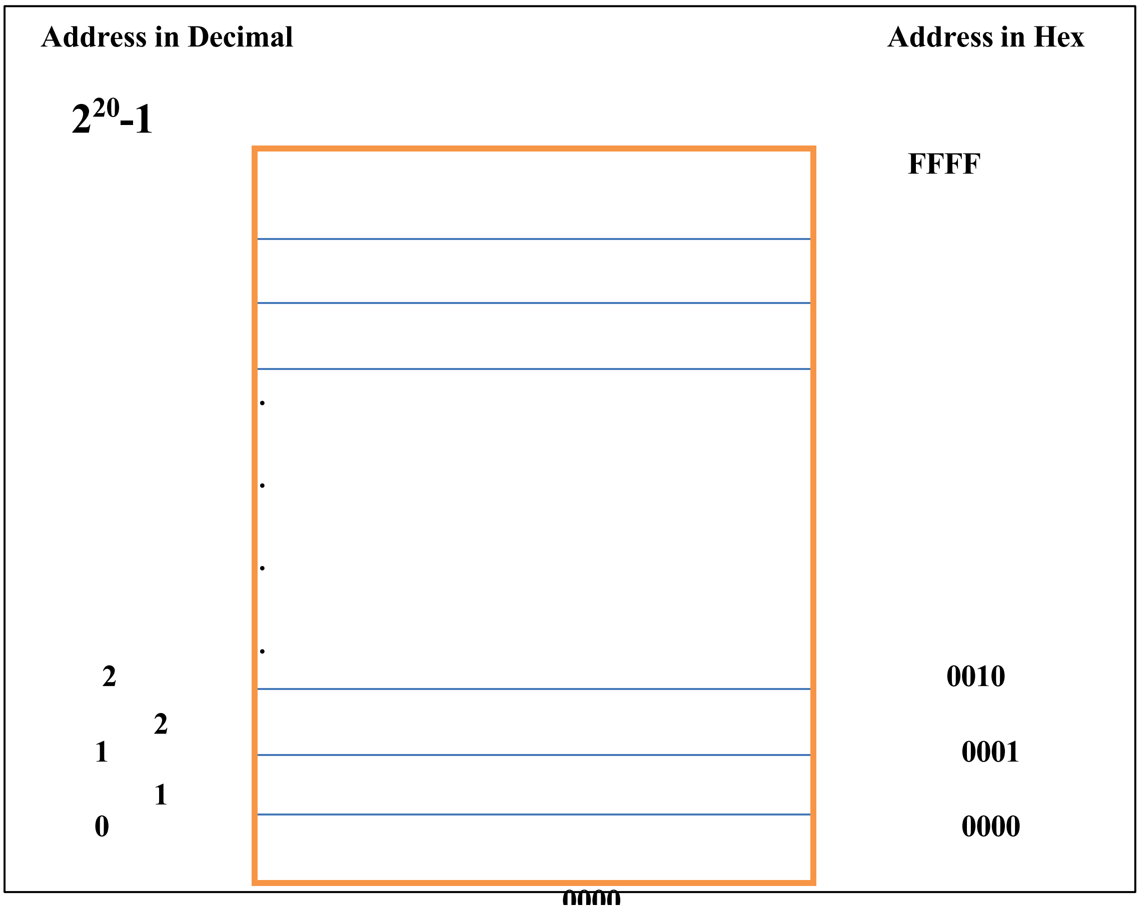


Figure 12 Logical view of the system memory

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.

Steps in a typical <u>Read cycle</u>:

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 <u>Write cycle</u>:

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 5 represents Bit, Byte, and larger units.

Table 5 address 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-byte (16-bit)
Double Word	4-byte (32-bits)
Quad word	8-byte (64-byte)

Name	Number of Byte
Kilo Byte (KB)	The number 210=1024=1 KB thus 640K=640*1024=655360 bytes)
Megabyte (MB)	(1024*1024) byte or 1,048,576 byte) approximately 1,000,000 bytes
Gigabyte (GB)	(1024*1024*10240byte) or (1,073,741,824 byte), approximately 1,000,000,000 bytes.
Terabyte (TB)	Approximately 1,000,000,000,000 bytes.

Memory Chips

Memory chips have two main properties that determine their application, **storage capacity or size** and **access time or speed**. A memory chip contains a number of locations, each of which stores one or more bits of data known as its bit width. The storage capacity of a memory chip is the **product of the number of locations and the bit width**. For example, a chip with 512 locations and a 2-bit data width has a memory size of 512×2=1024 bits.
Since the standard unit of data is a byte (8 bits), the above storage capacity is normally given as 1024/8 =128 bytes.
The number of locations may be obtained from the address width of the chip. For example, a chip with 10 address lines has 2^{10}= 1024 or 1 k locations. Given an 8-bit data width, a 10- bit address chip has a memory size of 2^{10} ×8 = 1024×8 = 1k ×1 byte = 1 KB. The computer's word size can be expressed in bytes as well as in bits.

For example, a word size of 8-bit is also a word size of one byte; a word size of 16- bit is a word size of two byte. Computers are often described in terms of their word size, such as an 8-bit computer, a 16-bit computer and so on.
For example, a 16-bit computer is one in which the instruction data are stored in memory as 16-bit units, and processed by the CPU in 16-bit units. The word size also indicates the size of the data bus which carries data between the CPU and memory and between the CPU and I/O devices. To access the memory, to store or retrieve a single word of information, it is necessary to have a unique address.
The word address is the number that identifies the location of a word in a memory. Each word stored in a memory device has a unique address.

Addresses are always expressed as binary number, although hexadecimal and decimal numbers are often used for convenience.

The second property of memory chips is the access time, access time is the speed with which a location within the memory chip may be made a variable to the data bus. It is defined as the time interval between the instant that an address is sent to the memory chip and the instant that the data stored in to the location appears on the data bus. Access time is given in nanosecond (ns) and varies from 25 ns to the relatively slow 200 ns.

NOTES:

- The large computers (mainframes) have word-sizes that are usually in the 32-to-64 –bits range.
- Mini computers have a word size from 8-to-32-bits range.
- Microcomputers have a word size from 4-to-32-bits range.

In general, a computer with a larger word size, can execute programs of instruction at a fast rate because more data and more instruction are stuffed into one word. The larger word sizes, however, mean more lines making up the data bus, and therefore more interconnections between the CPU and memory and I/O devices.

The word size is 4-bit therefore there are 4-data I/P lines and 4-data O/P lines. This memory has 32 different words, and therefore has 32 different addresses (storage location) from (00000) to (11111). Thus, we need a 5 address I/P lines.

Memory capacity = number of memory storage Location ×size of each word

= (number of word) × (number of bits per word)

= m (word)*n(bits)

= m*n bits

The capacity of memory depends on two parameters, the number of words (m) and the number of bits per word (n).

Every bit added to the length of address will double the number of words in the memory. The increase in the number of bits per bits requires that an increase the length of data I/P and data O/P lines.

Memory capacity units

Byte $\longrightarrow$ 8bits
KB $\longrightarrow$ 1024 Byte
MB $\longrightarrow$ 1024 KB
GB $\longrightarrow$ 1024 MG
TB $\longrightarrow$ 1024 GB

Figure 13 shows the unit of memory capacity measurement.

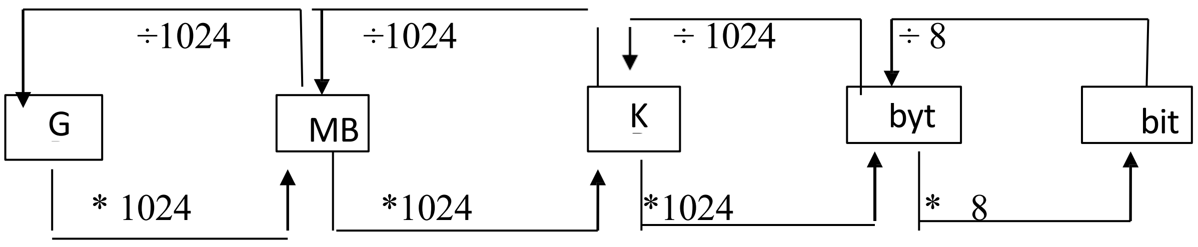


Figure 13 memory capacity measurements

Examples

1. If the capacity of memory is 2MG, what is the capacity in KB ?
   Solution: the capacity in KB
   Capacity = 2*1024= 2048 KB
2. the capacity of memory is 10MG, what is the capacity in byte?
   Solution: The capacity in byte
   Capacity = 10*1024 *1024 = 10485760
3. the capacity of memory is 2MG, what is the capacity in bit ?2*1024*1024*8= 16777216 bits
4. if the capacity of memory is 16 bits, what is the capacity in Byte?
   Solution: the capacity in byte
   Capacity = 16/ 8 = 2 byte
5. the capacity of memory is 20 KB, what is the capacity in MG?
   Solution: the capacity in MG
   capacity = 20/ 1024=0.01953125 MG
6. the capacity of memory is 15 KB, what is the capacity in GB?
   Solution: the capacity in KB
   Capacity = 15/1024 /1024 = 0.0000143051 GB

Memory example

Example1 A certain memory chip is specified as 2K×8:

1. How many words can be stored on this chip?
2. What is the words size?
3. How many total bits can this chip store?

SOLUTION:

1. 2K =2 × 1024 = 2048 words(bytes)
2. The word size is 8-bits (1 byte).
3. Capacity = 2048 × 8 = 16384 bits. Memory chip

Example2: A certain memory chip is specified as 2K × 16

1. How many words can be stored on this chip?
2. What is the words size?
3. How many total bits can this chip store?

Solution:

1. 2K = 2 × 1024 = 2048 words
2. The word size is 16-bits(2 byte).
3. Capacity = 2048 * 16 = 32768 bits.

Example3:- Which memory stores the most number of bits: 2MG × 8 memory or 2MG × 16 memory?

Solution:

2MG= 2×1024× 1024 = 2 ×(1048576) =words

1. Capacity 2MG ×8 =(2 × 1024 ×1024) × 8 = 16,777,216 bits.
2. Capacity 2MG ×16=(2 × 1024 ×1024) ×16= 33,554,432 bits.

So 2MG × 16 memory is bigger than 2MG × 8

Example4: Which memory stores the greatest number of bits: 4MG × 8 memory or 2MG × 16?

Solution:

1. Capacity = $(4 \times 1024 \times 1024) \times 8 = 33{,}554{,}432$ bits.
2. Capacity = $(2 \times 1024 \times 1024) \times 16 = 33{,}554{,}432$ bits.

Example5: A certain memory has a capacity of $4K \times 8$

1. How many data I/P & data O/P lines?
2. How many word address line?
3. What is its capacity in byte?

Solution

1. 8 each line: So, data I/P lines = data O/P lines =8
2. $4 \times 1024 = 4096$ words(bytes) Thus, there are 4096 memory addresses
   $2^{12} = 4096$

So, it required a 12- bit address line

3. The capacity = $(4 \times 1024) \times 8 = 32{,}768$ bit $= 32{,}769/8 = 4096$ byte (since 1byte = 8 bit).

Example6: - the a certain memory has a capacity of $4K \times 16$

1. How many data I/P & data O/P lines?
2. How many word address lines?
3. What is its capacity in byte?

Solution:

1. 16 each one.
   Data I/P lines = data O/P lines =16
2. $4 \times 1024 = 4096$ words
   Thus, there are 4096 memory addresses.
   $4096 = 2^{12}$ so, its require a 12-bit address line.
3. Capacity = $(4 \times 1024) \times 16 = 65{,}536$ bit
   $= 65{,}536 / 8 = 8.192$ byte

H.W: A computer system has **8bits data I/P** & data **O/P lines** and 12-**bit address bus (address lines)** find the **capacity of memory** in **bits and KB**.

Memory Types and Physical addressing

Memory Representation

The computer memory stores different kinds of data like input data, output data, intermediate results, etc., and the instructions. ***Binary digit*** or ***bit*** is the basic unit of memory. A ***bit*** is a single binary digit, i.e., 0 or 1. A bit is the smallest unit of representation of data in a computer. However, the data is handled by the computer as a combination of bits. A group of 8 bits form a **byte**.
One byte is the smallest unit of data that is handled by the computer.
One byte (8 bit) can store $\mathbf{2^8}$ = 256 different combinations of bits, and thus can be used to represent 256 different symbols. In a byte, the different combinations of bits fall in the range 00000000 to 11111111. A group of bytes can be further combined to form a **word**. A word can be a group of 2, 4 or 8 bytes.

1 bit = 0 or 1
1 Byte (B) = 8 bits
1 Kilobyte (KB) = 2^{10} = 1024 bytes
1 Megabyte (MB) = 2^{20} = 1024KB
1 Gigabyte (GB) = 2^{30} = 1024 MB = 1024 *1024 KB
1 Terabyte (TB) = 2^{40}= 1024 GB = 1024 * 1024 *1024 KB

I. Characteristics Of Memories

- **Volatility**
  - Volatile {RAM}
  - Non-volatile {ROM, Flash memory}
- **Mutability**
  - Read/Write {RAM, HDD, SSD, RAM, Cache, Registers…}
  - Read Only {Optical ROM (CD/DVD…), Semiconductor ROM}
- **Accessibility**
  - Random Access {RAM, Cache}
  - Direct Access {HDD, Optical Disks}
  - Sequential Access {Magnetic Tapes}

II. Memory Hierarchy

The memory is characterized on the basis of two key factors: ***capacity*** and ***access time***.

- ***Capacity*** is the amount of information (in bits) that a memory can store.
- ***Access time*** is the time interval between the read/ write request and the availability of data. The lesser the access time, the faster is the *speed of memory*.

Ideally, we want the memory with *fastest speed and largest capacity*. However, the cost of fast memory is very high. The computer uses a hierarchy of memory that is organized in a manner to enable the fastest speed and largest capacity of memory.

The hierarchy of the different memory types is shown in Figure 14.

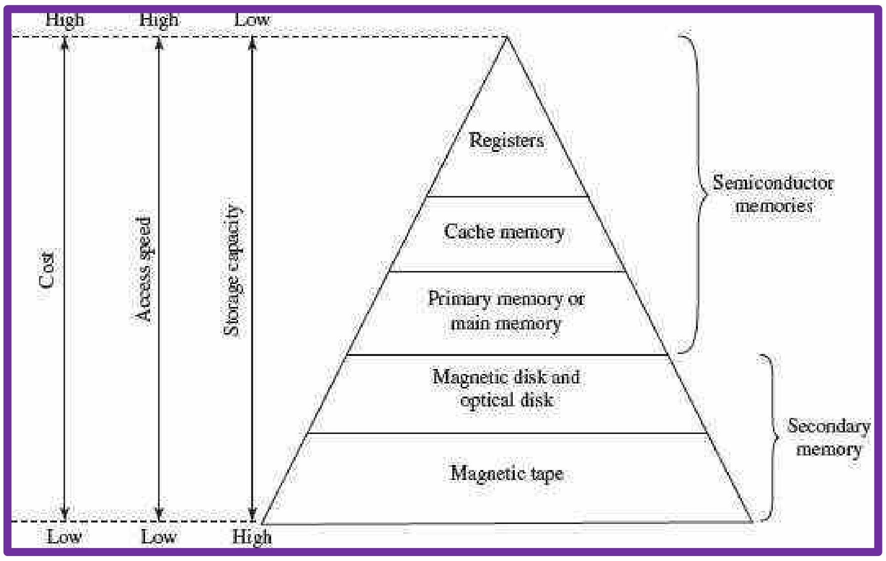


Figure 14 Memory hierarchy

The Internal Memory and ***External Memory*** are the two broad categories of memory used in the computer. *The Internal Memory* consists of the CPU registers, cache memory and primary memory. The internal memory is used by the CPU to perform the computing tasks. *The External Memory* is also called the ***secondary memory***. The secondary memory is used to store the large amount of data and the software.

In general, referring to the computer memory usually means the internal memory.

- **Internal Memory**

The key features of internal memory are:

1. Limited storage capacity.
2. Temporary storage.
3. Fast access.
4. High cost.

Registers, *cache memory*, and *primary memory* constitute the internal memory. *The primary memory* is further of two kinds: RAM and ROM. *Registers* are the fastest and the most expensive among all the memory types. The registers are located inside the CPU, and are directly accessible by the CPU. The speed of registers is between 1-2 ns (nanosecond). The sum of the size of registers is about 200B. *Cache memory* is next in the hierarchy and is placed between the CPU and the main memory. The speed of cache is between 2-10 ns. The cache size varies between 32 KB to 4MB. Any program or data that has to be executed must be brought into RAM from the secondary memory. Primary memory is relatively slower than the cache memory. The speed of RAM is around 60ns. The RAM size varies from 512KB to 64GB.

- **Secondary Memory**

The key features of secondary memory storage devices are:

1. Very high storage capacity.
2. Permanent storage (non-volatile), unless erased by user.
3. Relatively slower access.
4. Stores data and instructions that are not currently being used by CPU but may be required later for processing.
5. Cheapest among all memory.

To get the fastest speed of memory with largest capacity and least cost, the fast memory is located close to the processor. The secondary memory, which is not as fast, is used to store information permanently, and is placed farthest from the processor.

With respect to CPU, the memory is organized as follows:

- ***Registers*** are placed inside the CPU (small capacity, high cost, very high speed)
- ***Cache memory*** is placed next in the hierarchy (inside and outside the

CPU)
- ***Primary memory*** is placed next in the hierarchy
- ***Secondary memory*** is the farthest from CPU (large capacity, low cost, low speed) The speed of memories is dependent on the kind of technology used for the memory. The registers, cache memory and primary memory are **semiconductor memories**. They do not have any moving parts and are fast memories. The secondary memory is **magnetic or optical memory** has moving parts and has slow speed.

III. CPU Registers

Registers are very high-speed storage areas located inside the CPU. After CPU gets the data and instructions from the cache or RAM, the data and instructions are moved to the registers for processing. Registers are manipulated directly by the control unit of CPU during instruction execution. That is why registers are often referred to as the CPU's *working memory*. Since CPU uses registers for the processing of data, the number of registers in a CPU and the size of each register affect the power and speed of a CPU. The more the number of registers (ten to hundreds) and bigger the size of each register (8 bits to 64 bits), the better it is.

IV. Cache Memory

Cache memory is placed in between the CPU and the RAM. Cache memory is a fast memory, faster than the RAM. When the CPU needs an instruction or data during processing, it first looks in the cache. ***If the information is present in the cache***, it is called a ***cache hit***, and the data or instruction is retrieved from the cache. ***If the information is not present in cache***, then it is called a ***cache miss*** and the information is then retrieved from RAM.

Type of Cache memory

Cache memory improves the speed of the CPU, but it is expensive. Type of Cache Memory is divided into different levels that are L1, L2, L3:

Level 1 (L1) cache or Primary Cache

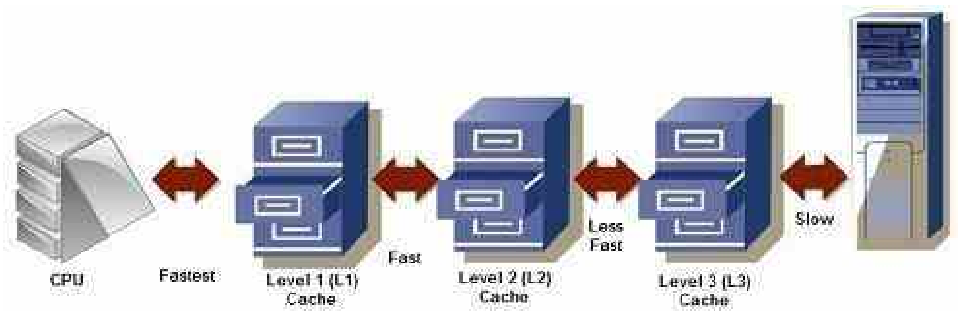
L1 is the primary type cache memory. The Size of the L1 cache very small comparison to others that is between ***2KB to 64KB,*** it depends on computer processor. It is an embedded register in the computer microprocessor (CPU).The Instructions that are required by the CPU that are firstly searched in L1 Cache. Example of registers are accumulator, address register, Program counter etc.

Level 2 (L2) cache or Secondary Cache

L2 is secondary type cache memory. The Size of the L2 cache is more capacious than L1 that is between ***256KB to 512KB***. L2 cache is located on computer microprocessor. After searching the Instructions in L1 Cache, if not found then it searched into L2 cache by computer microprocessor. The high-speed system bus interconnecting the cache to the microprocessor.

Level 3 (L3) cache or Main Memory

The L3 cache is larger in size but also slower in speed than L1 and L2, its size is between ***1MB to 8MB***. In Multicore processors, each core may have separate L1 and L2, but all cores share a common L3 cache. L3 cache double speed than the RAM.



The advantages and disadvantages of cache memory are as follows:

Advantages

The advantages of cache memory are as follows:

- Cache memory is faster than main memory.
- It consumes less access time as compared to main memory.
- It stores the program that can be executed within a short period of time.
- It stores data for temporary use.

Disadvantages

The disadvantages of cache memory are as follows:

- Cache memory has limited capacity.
- It is very expensive.

V. PRIMARY MEMORY (Main Memory)

Primary memory is the main memory of computer. It is a chip mounted on the motherboard of computer. Primary memory is categorized into two main types: Random access memory (ram) and read only memory (rom). **RAM** is used for the temporary storage of input data, output data and intermediate results. The input data entered into the computer using the input device, is stored in RAM for processing. After processing, the output data is stored in RAM before being sent to the output device. Any intermediate results generated during the processing of program are also stored in RAM. Unlike RAM, the data once stored in **ROM** either cannot be changed or can only be changed using some special operations. Therefore, ROM is used to store the data that does not require a change.

Types of Primary Memory

1. RAM (Random Access Memory)

The Word "**RAM**" stands for "random access memory" or may also refer to short- term memory. It's called "random" because you can read store data randomly at any time and from any physical location. It is a temporal storage memory. RAM is volatile that only retains all the data as long as the computer powered. It is the fastest type of memory. RAM stores the currently processed data from the CPU and sends them to the graphics unit.

There are generally two broad subcategories of RAM:

- **Static RAM** (**SRAM**): Static RAM is the form of RAM and made with flip-flops and used for primary storage are volatile. It retains data in latch as long as the computer powered. SRAM is more expensive and consumes more power than DRAM. It used as Cache Memory in a computer system. As technically, SRAM uses more transistors as compared to DRAM. It is faster compared to DRAM due to the latching arrangement, and they use 6 transistors per data bit as compared to DRAM, which uses one transistor per bit.
- **Dynamic Random Access Memory (DRAM)**: It is another form of RAM used as Main Memory, its retains information in Capacitors for a short period (a few milliseconds) even though the computer powered. The Data is Refreshed Periodically to maintain in it. The DRAM is cheaper, but it can store much more information. Moreover, it is also slower and consumes less power than SRAM.

2. ROM (Read Only Memory)

ROM is the long-term internal memory. ROM is "Non-Volatile Memory" that retains data without the flow of electricity. ROM is an essential chip with permanently written data or programs. It is similar to the RAM that is accessed by the CPU. ROM comes with pre-written by the computer manufacturer to hold the instructions for booting-up the computer.

There is generally three broad type of ROM:

- **PROM (Programmable Read Only Memory)**: PROM stands for programmable ROM. It can be programmed only be done once and read many. Unlike RAM, PROMs retain their contents without the flow of electricity. PROM is also nonvolatile memory. The significant difference between a ROM and a PROM is that a ROM comes with pre-written by the computer manufacturer whereas PROM manufactured as blank memory. PROM can be programmed by PROM burner and by blowing internal fuses permanently.
- **EPROM (Erasable Programmable Read Only Memory)**: EPROM is pronounced ee-prom. This memory type retains its contents until it exposed to intense ultraviolet light that clears its contents, making it possible to reprogram the memory.
- **EEPROM (Electrically Erasable Programmable Read Only Memory)**: EEPROM can be burned (programmed) and erased by first electrical waves in a millisecond. A single byte of a data or the entire contents of device can be erased. To write or erase this memory type, you need a device called a PROM burner.

Physical Addressing

Addresses are 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.

Addresses type

There are three types of addresses.

1- Physical address.
2- Offset address.
3- Logical address.

1. Physical Address (PA)

The physical address is the 20- bit address that actually put on the address pins of the 8086 microprocessors. The 8086 microprocessor handles 1 MB of memory and has an address range from 00000 H to FFFFF H. Since each hexadecimal digit is represented by 4-bits, so each memory address requires 20-bits to represent the 5 digits of physical address.

Physical address Generation

A 2-byte register can only store 4 hexadecimal digits (16 bits). So, the address stored in the segment register (Base address) is multiplied by 10 H (or shifting it 4 digits to the left or adding zeros to the bottom 4 bits) to get 5 hexadecimal digits of the physical address (20 bits) and then the offset address is added as in the following equation. Figure 15 shows how the physical address is generated.

Physical address = Base address x 10 + Offset address.

Example:

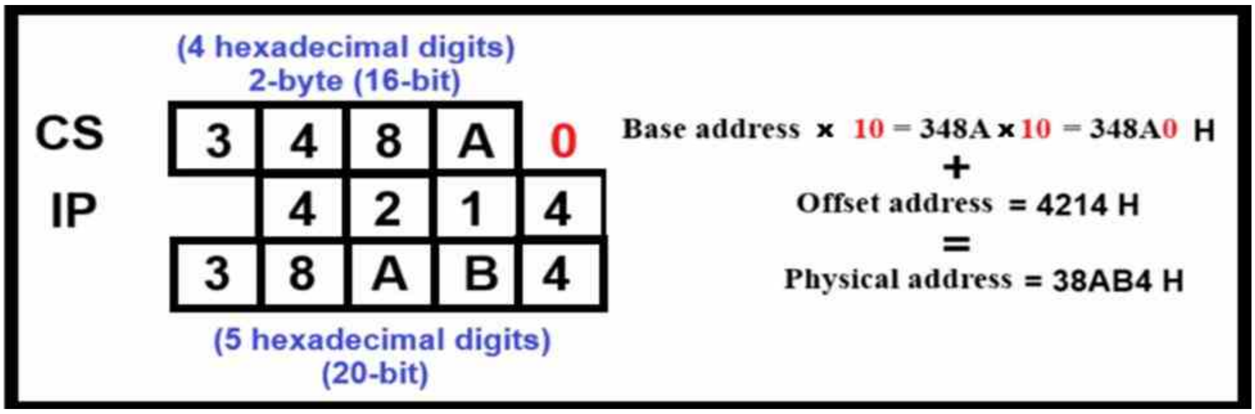


Figure 15 Physical address Generation

Offset address

Offset is the displacement of the memory location from the starting location of the memory segment (Data, Extra, Stack, or Code). The complete physical address which is 20-bits long is generated using segment and offset registers each of the size 16-bit. So, the addresses stored in these registers range from 0000 H to FFFF H. In Other word, **offset address is the number of address locations added to a base address in order to get to a specific absolute address**. See figure 16.

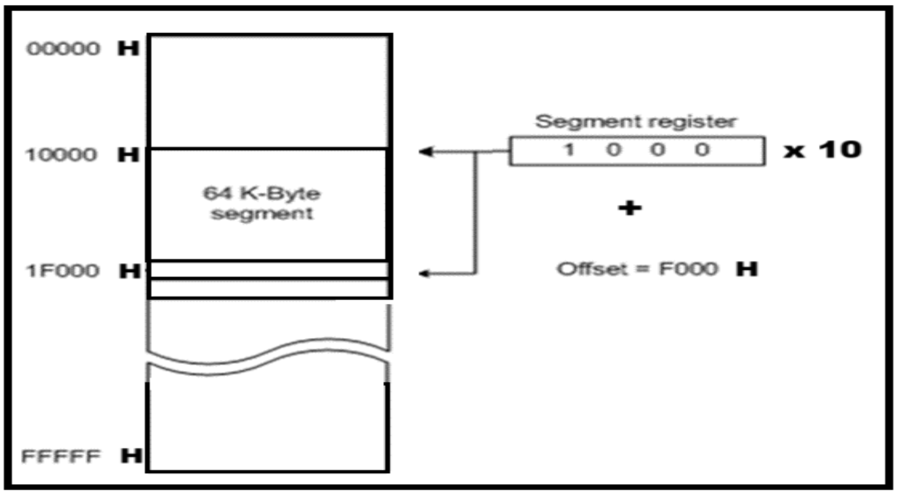


Figure 16 the implementation of offset address

Segment and Offset Registers

Segment registers contain base addresses for segments and each of these registers handles a specific type of register that contains offset addresses. By combining the base address and the offset address, the physical address is generated as shown in the following table:

Table 6 physical address

Segment register	Offset register	Physical address
CS	IP	Instruction address
SS	SP, BP	Stack address
DS	BX, DI, SI	Data address
ES	BX, DI, SI	String destination address for string instruction

Logical address (virtual address)

A logical address is an address generated by the CPU during program execution that refers to a location in the memory space. It acts as a reference point for where data or instructions are stored within a program. The logical address consists of the segment base address (located in the segment register) and the offset address and it is represented by the following format (**Segment :Offset**).

Q:/ What is the difference between logical and physical address in 8086?

Answer

Logical address is generated by CPU in perspective of a program whereas the physical address is a location that exists in the memory unit.

Examples shows how to add numbers in hexadecimal system which is used to calculate memory locations

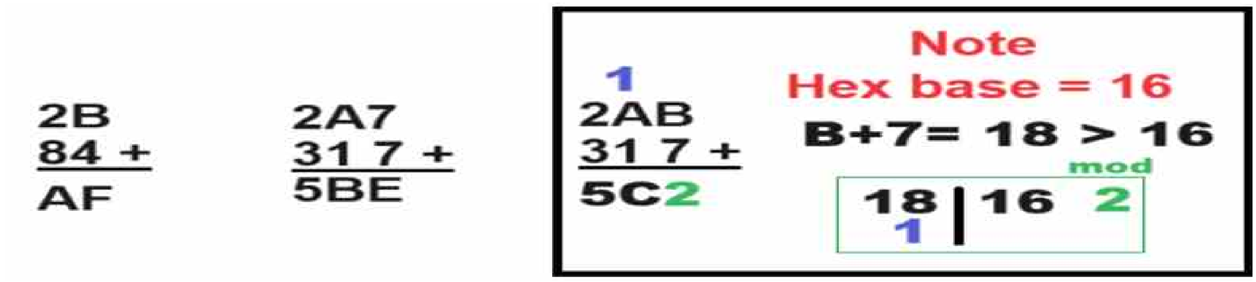


Figure 17 adding numbers

Q8: How physical address is calculated in 8086 explain with example?

The 8086 addresses a segmented memory. The complete physical address which is 20-bits long is generated **using segment and offset registers each of the size 16-bit**. The content of a segment registers also called as segment address, and content of an offset register also called as offset address.

Example:

If CD = 0100 H and IP = 9F2C H, determine the **physical address**.

Solution (steps are shown in figure 18):

Physical address = Base address × 10 + Offset address
= 0100 × 10 + 9F2C
= 01000 + 9F2C
= 0AF2C H

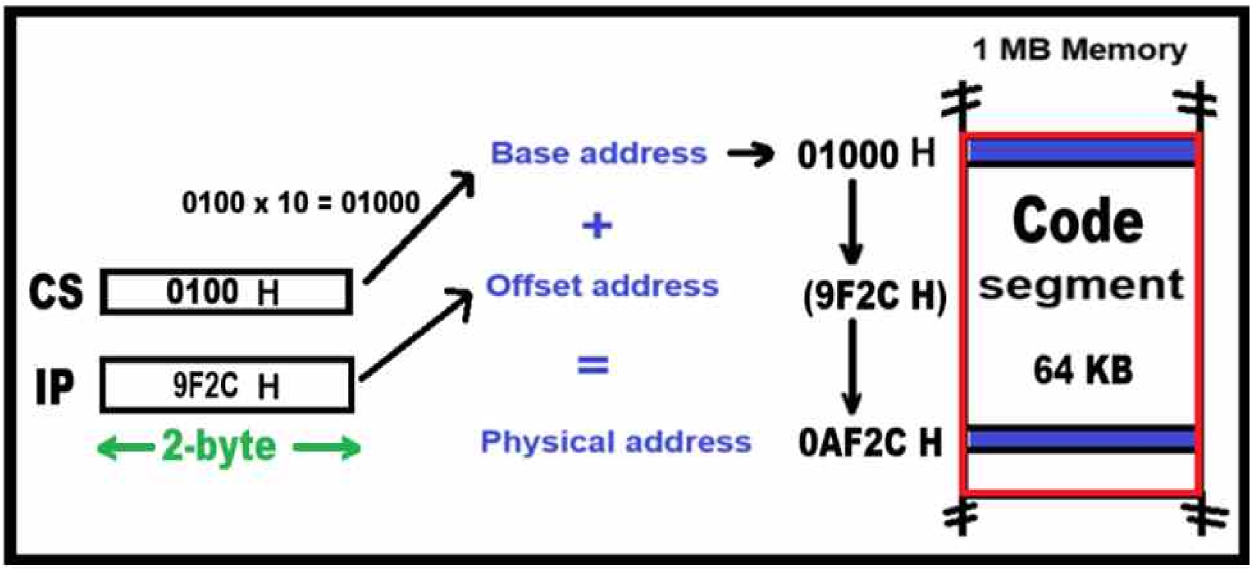


Figure 18 Determining the physical address

<u>**Example 1:**</u>

If DS= 24F6 H and SI = 634A H, **determine**:

a- The offset address

b- The physical address

c- The logical address

d- The lower range of the Data segment

e- The upper range of the Data segment

Solution:

a- 634A H

b- 24F6 x 10 + 634A = 2B2AA H

c- 24F6: 634A

d- 24F6 x 10 + 0000 = 24F60 H

e- 24F6 x 10 + FFFF = 34F5F H

Example 2:

If SS=7FA2 H and SP= 438E H, determine:

a- The offset address

b- The physical address

c- The logical address

d- The lower range of the Stack segment

e- The upper range of the Stack segment

Solution:

a- 438E H

b- 7FA20 + 438E = 83DAE H

c- 7FA2: 438E

d- 7FA20 + 0000 = 7FA20 H

e- 7FA20 + FFFF = 8FA1F H

Input/ Output devices

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.

Input devices (even output devices) can be distinguished from each other by their ***data processing rate*** which means the average number of characters that can be processed by a device per second.

For example, the data processing rate for ***the keyboard*** is about 10 characters (bytes)/second, a ***scanner*** can send data at a rate of about 200,000 characters/second. Similarly, while ***a laser printer*** can output data at a rate of about 100,000 characters/second, a ***graphic display*** can output data at a rate of about 30,000,000 characters/second.

A simple arrangement for connecting the processor and the memory in a given computer system to an input device, for example, a keyboard and an output device such as a graphic display. A single bus consisting of the required address, data, and control lines is used to connect the system's components is shown in **Figure 19**.

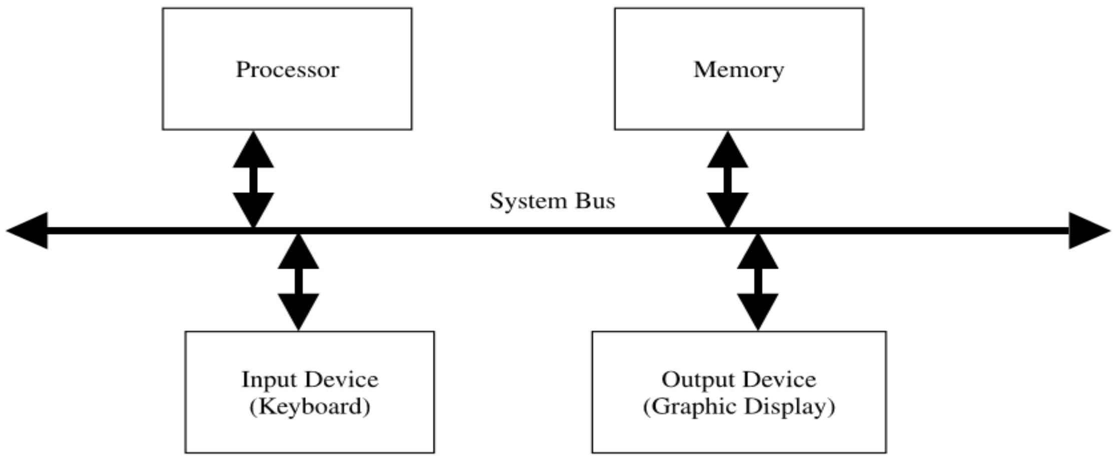


Figure 19 A single bus system

I/O protocol is a simple way of communication between the processor and I/O devices, it requires the availability of the input and output registers. In a typical computer system, there is a number of input registers, each belonging to a specific

input device. There is also a number of output registers, each belonging to a specific output device.
More than one arrangement exists to satisfy these requirements. Among these, two particular methods are ***Isolated I/O and Memory-Mapped I/O***.

Isolated I/O

Isolated I/O also called ***Shared I/O,*** is a method for managing I/O devices in a computer system. ***In the isolated I/O configuration,*** a distinct address space is assigned for I/O devices separated from the memory address space. the CPU uses specific instructions (like IN and OUT) to communicate with I/O devices, these instructions are different from the ones used to access memory. Isolated I/O often utilizes separate control lines for I/O operations, distinguishing them from memory operations.
The main advantage of Isolated I/O is ***the separation between the memory address space and that of the I/O devices***. While the main disadvantage is ***the need to have special input and output instructions in the processor instruction set.*** Figure 20 shows the arrangement of Isolated I/O devices.

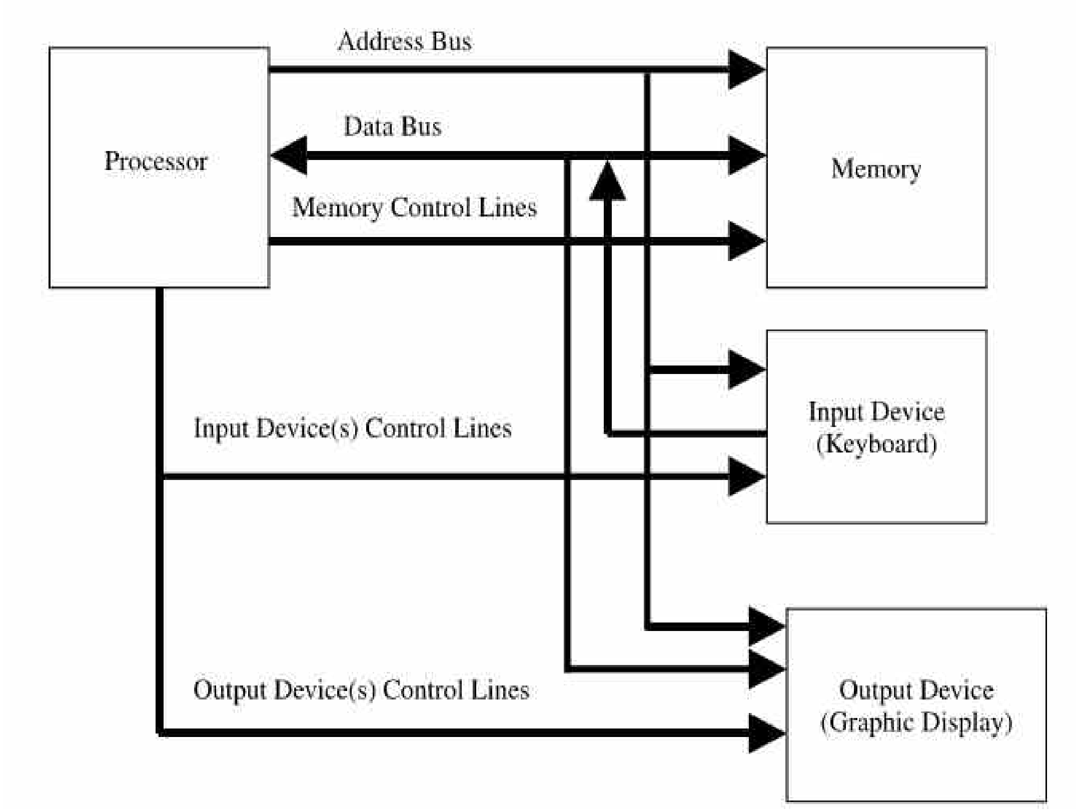


Figure 20 Isolated I/O arrangement

Memory mapped I/O

Memory-mapped I/O is a technique where input/output (I/O) devices and main memory share the same address space. This means that the CPU can interact with I/O devices using the same instructions it uses to access memory (like LOAD, Store. In this configuration there is no ***dedicated control lines*** for I/O operations. Instead, the same control lines used for memory access are also used for I/O. figure 21 show the memory mapped arrangement.

The **main advantage** of the memory-mapped I/O is the use of the read and write instructions of the processor to perform the input and output operations, respectively. It eliminates the need for introducing special I/O instructions. The **main disadvantage** of the memory-mapped I/O is the need to reserve a certain part of the memory address space for addressing I/O devices, that is, a reduction in the available memory address space.

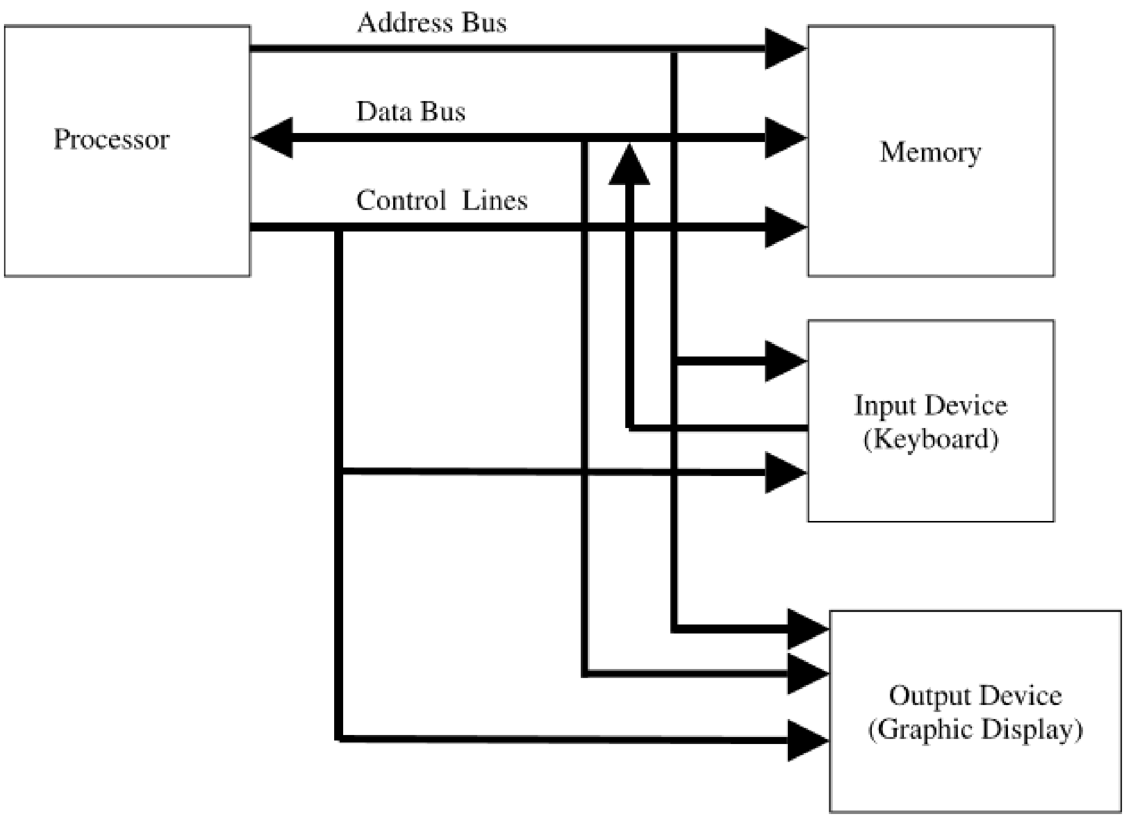


Figure 21 the memory mapped arrangement

Modes of Transfer

Data transfer between the processor and I/O devices may be handled in a variety of modes. three possible modes are Programmed I/O, Interrupt I/O and Direct Memory Access (MDA).

Programmed I/O

In this mode of the data transfer, the I/O operations are performed under the control of the CPU. A complete instruction fetches, decode, and execute cycle will have to be executed for every input and every output operation. The CPU sends commands to the I/O device, waits for the device to become ready, and then transfers the data. It is a simple method but its main disadvantage is that, the CPU is heavily involved and spends a lot of time waiting for the I/O device (wasting the CPU time), which is inefficient.

Interrupt-Driven I/O

Before we explain this mode of data transfer between the processor and I/O device, we will know what is an interrupt and how the CPU responds to it.

An ***interrupt*** is a signal to the processor that an event has occurred that requires its attention. This event can be triggered by various sources, such as: I/O devices (data transfer), hardware errors (disk failure) or software events (a program making a system call).

When the CPU is interrupted, it is required to discontinue its current activity, attend to the interrupting condition (serve the interrupt), and then resume its activity from wherever it stopped. Discontinuity of the processor’s current activity requires finishing executing the current instruction, saving the processor status (mostly in the form of pushing register values onto a stack), and transferring control (jump) to what is called the ***interrupt service routine*** (ISR). The service offered to an interrupt will depend on the source of the interrupt. For example, *in the case of an I/O interrupt, serving an interrupt means to perform the required data transfer*. Upon finishing serving an interrupt, the processor should restore the original status by popping the relevant values from the stack. Once the processor returns to the normal state, it can enable sources of interrupt again.

In the ***interrupt-Driven I/O*** the CPU initiates the I/O operation and then goes on to perform other tasks. When the I/O device is ready to transfer data, it sends an interrupt signal to the CPU. The CPU then handles the interrupt and transfers the data. The advantage of this transfer mode, it is more efficient than programmed I/O as the CPU is not constantly waiting. Its disadvantage is that it still requires the CPU intervention for each data transfer, which can be overhead for large data transfers.

Direct Memory Access (DMA)

The main idea of direct memory access (DMA) is to enable peripheral devices to transfer data directly from and to memory without the intervention of the CPU. This mechanism allows the CPU to do other work, which would lead to improved performance, especially in the cases of large transfers.
The DMA controller is a piece of hardware that controls one or more I/O devices. It allows direct data transfer between I/O devices and the system's memory without the help of the processor. In a typical DMA transfer, some event notifies the DMA controller that data needs to be transferred to or from memory. Both the DMA and CPU use memory bus and only one or the other can use the memory at the same time. The DMA controller requests CPU permission to use a bus, which is granted by the CPU. The DMA then conducts memory transfer independently, and after completion it leaves the control to the CPU. Figure 22 shows how the DMA controller shares the CPU's memory bus.

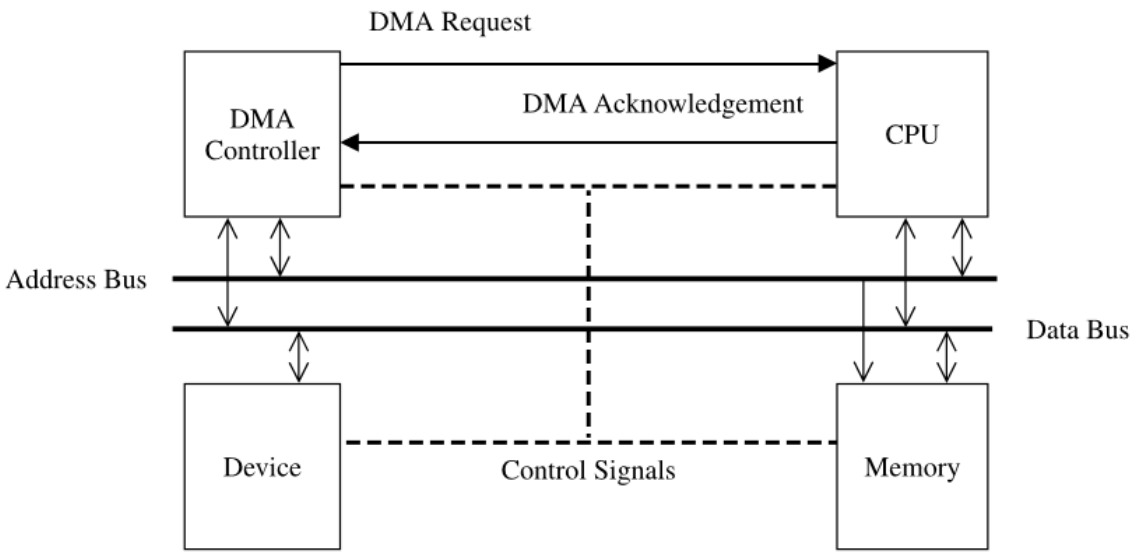


Figure 22 DMA controller shares the CPU's memory bus

Instruction Set and Format

Microprocessor - 8086 Instruction Sets

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

A three-address instruction takes the form ***operation add-1, add-2, add-3***. In this form, each of add-1, add-2, and add-3 refers to a register or to a memory location. For example, the instruction ***ADD R_1, R_2, R_3***. This instruction indicates that the operation to be performed is addition. It also indicates that the values to be added are those stored in registers R_1 and R_2 and the results should be stored in register R_3.

A two-address instruction takes the form ***operation add-1, add-2***. In this form, each of add-1 and add-2 refers to a register or to a memory location. For example, the instruction ***ADD R_1,R_2***. This instruction adds the contents of register R_1 to the contents of register R_2 and stores the results in register R_2. The original contents of register R_2 are lost due to this operation while the original contents of register R_1 remain intact.

A one-address instruction takes the form ADD R_1. In this case the instruction implicitly refers to a register, called the Accumulator R_{acc}, such that the contents of the accumulator is added to the contents of the register R_1 and the results are stored back into the accumulator R_{acc}.

The 8086 microprocessor supports 8 types of instructions

1. Data Transfer Instructions
2. Arithmetic Instructions
3. Bit Manipulation Instructions
4. String Instructions
5. Branch Instructions
6. Processor Control Instructions
7. Loop & Iteration Instructions
8. Interrupt Instructions

Let us now discuss these instruction sets in detail.

1. Data Transfer Instructions

These instructions are used to transfer the data from the source operand to the destination operand. The data can be of any type. They are classified into four groups as explain in table 7.

Table 7 data transfer instructions

General-purpose Byte or word transfer instructions	Special address transfer instructions	Simple input/output port transfer instructions	Flag transfer instructions
MOV XCHG XLAT PUSH POP	LEA LDS LES	IN OUT	LAHF SAHF PUSHF POPF

a) **MOV** − Used to copy a byte or a word from a specified source to specified destination. Data can be moved between general-purpose register, between a general-purpose register and a segment register, between a general-purpose register or a segment register and a memory, or between a memory location and the accumulator. Details of this instruction is shown in table 8.

Table 8 move instruction

Mnemonic	**Meaning**	**Format**	**Operation**	**Flags effected**
MOV	Move	MOV D, S	(S) → (D)	None

Example1: MOV CX, 037AH; it means move 037AH into CX, 037A -> CX
Example2: MOV AX, BX; it means copy the content of register BX to AX; BX-> AX
Example3: MOV DL, [BX]; it means copy byte from memory at BX to DL; DS*10+BX-> DL

b) **XCHG-** exchange the content of a register with the content of another register (or) the content of the register with the content of a memory location. ***Direct memory to memory exchange is not supported. Both operands must be of the same size and one operand must be a register.*** Table 9 shows the instruction details.

Table 9 XCHG instruction

Mnemonic	**Meaning**	**Format**	**Operation**	**Flags effected**
XCHG	Exchange	XCHG D, S	(D) ←→ (S)	None

Example1: CX, [037A]H ; [[DS*10]+037A] ←→ CX
Example2: AX, [BX] ; [[DS*10]+BX] ←→ AX

Example3: CX, [BP+200H] ; [[DS*10]+BP+200] ←→ CX

Example 4: what is the result of executing the following instruction?

XCHG AX, [0002]

Solution: figure 23 shows the solution

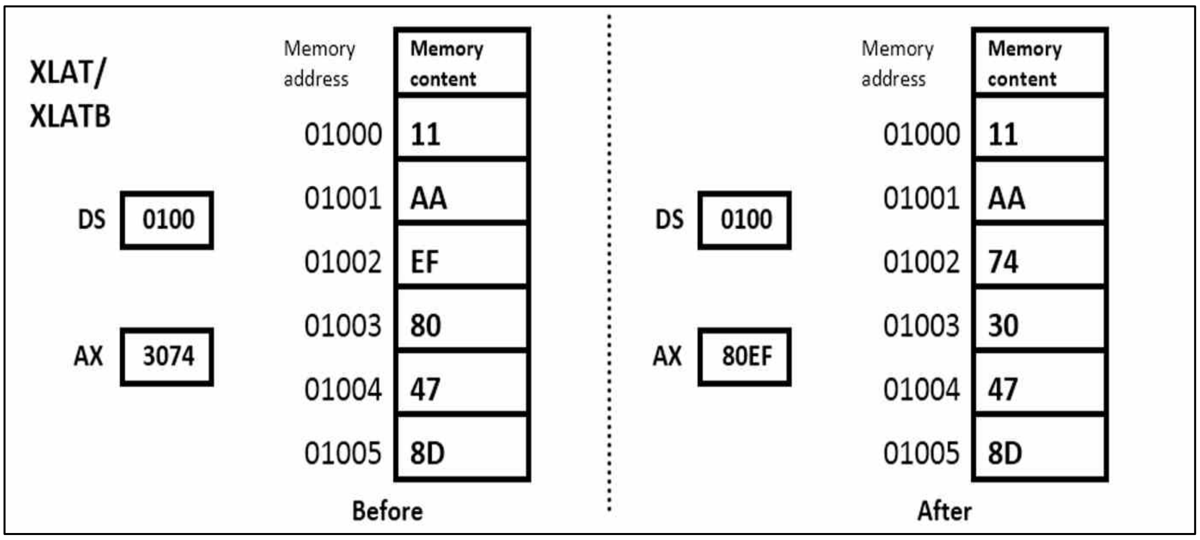


Figure 23 Solution of example 4

c) LEA, LDS and LES (Load Effective Address) Instructions

These instructions load a segment and general-purpose registers with an address directly from the memory. These instructions described in table 10.

Table 10 Load effective address instructions

Mnemonic	Meaning	Format	Operation	Flags effected
LEA	Load register with Effective Address	LEA reg16, EA	EA → (reg16)	none
LDS	Load register and DS with words from memory	LDS reg16, EA	[PA]→ (reg16) [PA+2] → [DS]	None
LES	Load register and ES with word from memory	LES reg16, EA	[PA]→ (reg16) [PA+2] → [ES]	

Example1: LEA BX, PRICE ; load BX with offset of PRICE in DS
Example2: LEA BP, SS:STAK ; Load BP with offset of STACK in SS

Example3: LEA CX, [BX][DI] ; Load CX with EA= BX + DI
Example4: LDS BX, [4326] ; copy the content of the memory at displacement 4326H in DS to BL, the content of the 4327H to BH, copy the content the content of the 4328H and 4329H in DS to DS register.
Example 5: Assuming that (BX)=100H, DI==200H, DS=1200H, SI=F002H, AX=0105H and the following memory content. What is the result of executing the following instructions?
Solution: is explained in figure 24

a. LEA SI , [DI + BX +2H
SI = (DI) + (BX) + 2H= 0200H+0100H+0002H= 0302H

b. MOV SI , [DI+BX+2H]
EA=(DI+BX+2H)= 0302H
PA=DS*10+EA=1200*10+0302=12302
SI = 80EFH

c. LDS CX , [300]
PA = DS*10+EA= 1200H*10+300H = 12300H
CX= AA11H and DS=80EFH

d. LES BX , [DI+AX]
EA = (DI+AX)= 0200H+0105H =0305H
PA= DS*10+EA = 1200H*10+0305H = 12305H
BX = 5A8DH and ES = C592H

Memory address	Memory content
12300	11
12301	AA
12302	EF
12303	80
12304	47
12305	8D
12306	5A
12307	92
12308	C5

Figure 24 solution of example 4

d) XLAT/XLATB Instruction- Translate a byte in AL

XLAT exchanges the byte in AL register from the user table index to the table entry, addressed by BX. It transfers 16-bit information at a time. The no-operands form XLAT provides a short form of XLAT instructions which explained in table11.

Table 11 XLAT instruction

Mnemonic	Meaning	Format	Operation	Flags effected
XLAT	Translate	XLAT	EA ← (DS*10 +(AL)+(BX))	none

Example: for the figure below, the result of executing the following instruction XLAT

Solution: is explained in figure 25.

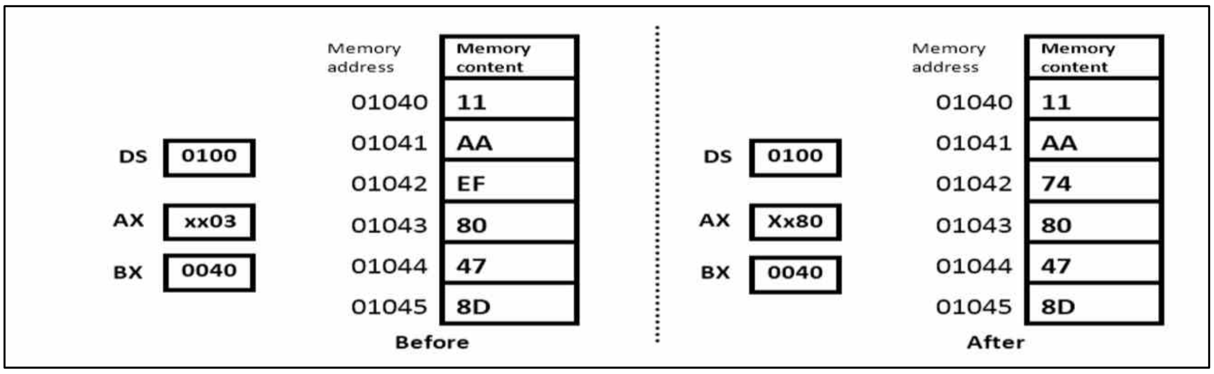


Figure 25 solution of XLAT instruction

The Stack

The stack is implemented in the memory and it is used for temporary storage of information such as data and addresses. The stack is 64kbutes long and is organized from a software point of view as 32kwords

- SS register points to the lowest address word in the stack.
- SP and BP point to the address within the stack.
- Data transferred to and from the stack are word-wide, not byte-wide.
- The first address in the stack segment (SS:0000) is called End of Stack.
- The last address in the stack segment (SS: FFFE) is called Bottom of Stack.
- The address (SS:SP) is called Top of Stack.

Figure 26 shows the stack representation diagram.

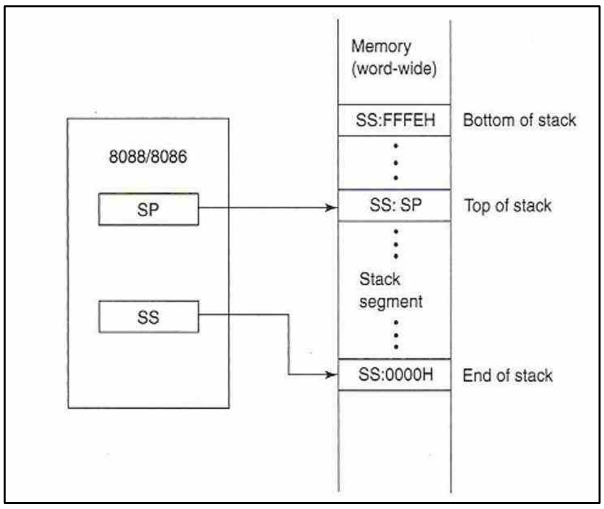


Figure 26 the Stack

e) PUSH and POP Instructions

The PUSH and POP instructions are important instructions that store and retrieve data from the stack memory. The instruction formats are as in table 12.

Table 12 Push and Pop instructions

Mnemonic	Meaning	Format	Operation	Flags effected
PUSH	Push a word into a stack	PUSH S	(SP) ← (SP-2) ((SP)) ← (S)	none
POP	Pop a word off a stack	POP D	(d) ← ((SP)) (SP) ← (SP+2)	None

- POP instruction is used to read word from the stack
- PUSH instruction is used to write word to the stack
- When a word to be pushed into the top of the stack
  - The value of SP is first automatically decremented by two this is because the stack typically grows downwards in memory (towards lower addresses) and then the content of the register written into the stack.
- When a word is to be popped from the top of the stack
  - The content is first moved from the top of the stack to a specific register then the value of SP is incremented by two.

Example: let AX= 1234H, SS = 0105H and SP= 0006H. Below is the state of stack prior and after the execution of next program instruction
PUSH AX,
POP BX,
POP AX
Solution: is shown in figure 27

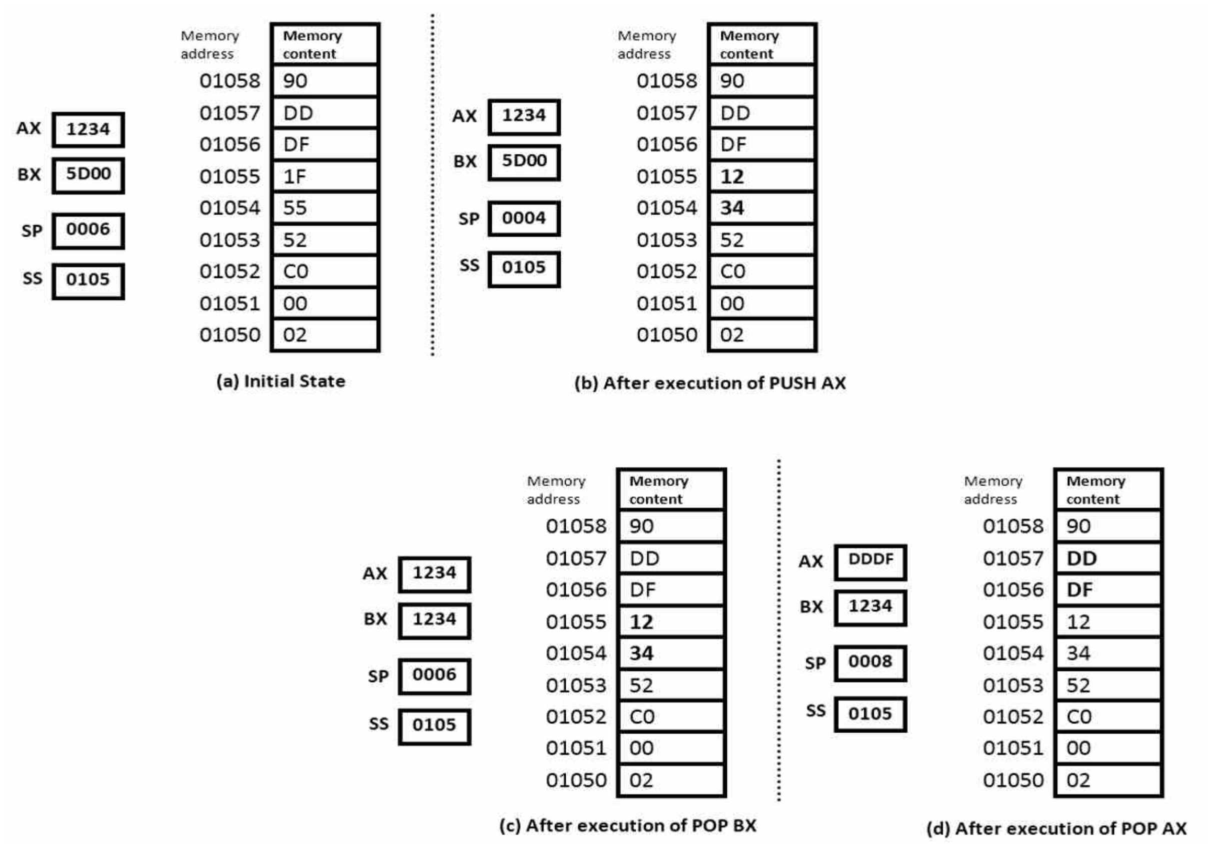


Figure 27 solution of stack example

f) INPUT/ OUTPUT Instructions

There are two different forms of Input and Output instructions: the ***direct I/O instructions*** and ***variable I/O instructions***. These two types of instructions can be used to transfer a byte or word of data. The data transfer takes place between the I/O device and the *microprocessor unit's (MPU) accumulator register. Table 13 shows these instructions.*

Table 13 Input and Output Instructions

Mnemonic	Meaning	Format	Operation	Flags effected
IN	Input direct Input variable	IN Acc, Port IN Acc, DX	(Acc) ← (Port) (Acc) ← (DX)	none
OUT	Output direct Output variable	OUT Port, Acc Out DX, Acc	(Port) ← (Acc)	None

			(DX) ← (Acc)	

Example1: IN AL, 0C8H ; Input a byte from port 0C8H to AH
Example1: IN AX, 34H ; Input a byte from port 34H to AX
Example1: OUT 3BH, AL ; Copy the content of AL to port 3BH
Example1: OUT 2CH, AX ; Copy the content of AX to port 2CH

g) LAHF Instruction- Load Register AH from Flags

LAHF instruction copies the values of SF, ZF, AF, PF and CF into 7,6,4,2 and 0 respectively of AH register the LAHF was provided to make conversion of assembly language programs written for 8080 and 8085 to 8986 easier.

h) SAHF instruction AH Register into FLAGS

SAHF instruction transfers the bits 0-7 of AH of SF, ZF, AF, PF and CF into the flag register

i) PUSHF Instruction- Push flag register on the stack

This instruction decrements the SP by 2 and copies the word in flag register to the memory location pointed by SP.

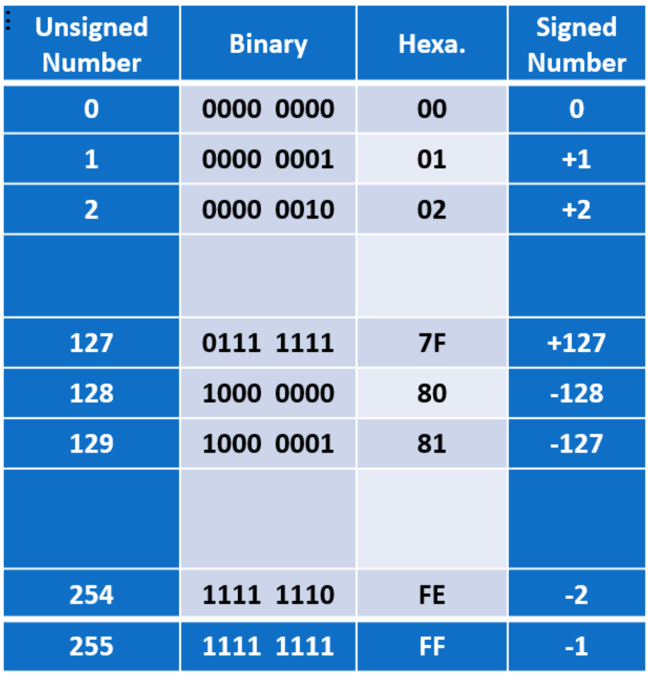
j) POPF –Pop word from top of stack to flag- register

This instruction copies a word from the two-memory location at the top of the stack to flag register and increments the stack pointer by 2.

1. Arithmetic Instructions

These instructions are used to perform arithmetic operations like addition, subtraction, multiplication, division, etc.

An 8-bit number system can be used to create 256 combinations from (0-255). The first 128 combination represent the positive numbers from (0-127) and the next 128 combination from (128-255) represent the negative numbers ((-128) - (-1)). The binary numbers are shown in figure 28.



Unsigned Number	Binary	Hexa.	Signed Number
0	0000 0000	00	0
1	0000 0001	01	+1
2	0000 0010	02	+2
127	0111 1111	7F	+127
128	1000 0000	80	-128
129	1000 0001	81	-127
254	1111 1110	FE	-2
255	1111 1111	FF	-1

Figure 28 the binary numbers

In binary, when the Most Significant Bit (MSB) position set to 1 then the number is negative otherwise it is a positive number. Table 14 explain the classification of the arithmetic instructions

Table 14 arithmetic instructions

Addition instructions	Subtraction instructions	Multiplication instruction	Division instruction
ADD ADC INC AAA DAA	SUB SBB DEC NEG CMP AAS DAS	MUL IMUL AAM	DIV IDIV AAD CBW CWD

The result of the execution of an arithmetic instruction is recorded in the flags registers. the C, A, S, Z, P and O flags are affected by the arithmetic instructions.

a) Addition Instruction: these instructions are described in table 15.

Table 15 addition instructions

Mnemonic	Meaning	Format	Operation	Flags effected
ADD	Addition	ADD D, S	(S)+(D) → (D) carry → (CF)	O,S,Z,A,P,C
ADC	Add with carry	ADC D, S	(S)+(D)+(CF)→ (D) CARRY → (CF)	O,S,Z,A,P,C
INC	Increment by 1	INC D	(D) +1 → D	O,S,Z,A,P
DAA	Decimal adjust for addition	DAA		S,Z,A,P,C
AAA	ASCII adjust for addition	AAA		A, C

Examples

ADD AL, 74H /add immediate number 74H to the content of AL
ADC CL, BL /add the content of BL plus carry status to the content of CL
ADD DX, [SI] /add a word from a memory at offset [SI] in DS to the content of DX

When adding unsigned numbers

ADD, CL, BL

Assuming that Cl= 01110011= 115(decimal); BL= 01001111 =79(decimal) the result in CL= 11000010 = 194(decimal)

Now assume we add two signed numbers

ADD CL, BL

Assume that CL= 01110011 = +115(decimal); BL = 01001111 = +79(decimal)
The result in CL= 11000010 = -62 (decimal); incorrect result because it is too large to fit in 7 bits.

More examples

7. ADD AX, 2
   ADC AX,2
8. INC BX
   INC WORD PTR [BX]

9. If you know the ASCII code FOR 0-9 = 30-39H
   ADD CL, DL
   suppose that [CL]=32 (ASCII for 2) AN [DL] = 35 (ASCII for 5) the result [CL] = 67
   MOV AL, CL // move the ASCII result into AL since AAA adjust only [AL]
   AAA // [AL] = 07, unpacked BCD for 7

b) **Subtraction instruction:** table 16 explain the detail of these instructions.

Table 16 subtraction instructions

Mnemonic	**Meaning**	**Format**	**Operation**	**Flags effected**
SUB	Subtraction	SUB D, S	(S)-(D) → (D) borrow → (CF)	O,S,Z,A,P,C
SBB	Subtract with borrow	SBB D, S	(S)-(D)-(CF)→ (D) borrow → (CF)	O,S,Z,A,P,C
DEC	Decrement by 1	DEC D	(D) – 1 → (D)	O,S,Z,A,P
NEG	Negative	NEG	0 - (D) → D 1→ (CF)	S,Z,A,P,C
CMP	Compare	CMP D,S	(S) – (D)	A, C
DAS	Decimal adjust for Subtraction	DAS		S,Z,A,P,C
AAS	ASCII adjust for Subtraction	AAS		A, C

c) NEG instruction

This instruction performs the 2's complement subtraction of the operand from zero and sets the flags according to the results

Example:

MOV AX, 2CBh
NEG AX // after executing NEG the result AX= FD35h

d) CMP Instruction- Compare byte or word-CMP destination, source

This instruction compares the destination and the source i.e., it subtracts the source from the destination. The result is not stored anywhere. It neglects the result, but sets the flags accordingly. This instruction is usually used before conditional JUMP instruction

Example:

MOV AL, 5

MOV BL, 5

CMP AL, BL **// AL= 5, ZF= 1 → EQUAL**

3. Multiplication And Division Instructions

The instructions format and operation are in table 17.

Table 17 multiplication and division instructions

Mnemonic	Meaning	Format	Operation
MUL	Multiply (unsigned)	MUL S	(AL)*(S8) → (AX); (AX) *(S16) →(DX)(AX)
DIV	Division (Unsigned)	DIV S	Q((AX) / (S8))→ (AL); R((AX)/(S8))→(AH) Q((DX,AX) / (S16)) → (AX); R((DX,AX) / (S16)) → (DX)
IMUL	Integer Divide (signed)	IMUL S	(AL) * (S8) → (AX)* (S16) → (DX)(AX)
IDIV	Integer Divide (Signed)	NEG	Q((AX) / (S8))→ (AL); R((AX)/(S8))→(AH) Q((DX,AX) / (S16)) → (AX); R((DX,AX) / (S16)) → (DX)
AAM	Adjust AL FOR Multiplication	AAM	Q(AL) /(10) → (AH); R((AL) /(10))→(AL)
AAD	Adjust AL FOR Division	AAD	(AH)*10 + (AL) →(AL) ; 00→(AH)
CBW	Convert byte to word	CBW	(MSB of AL) →(All bits of AH)
CWD	Convert word to double word	CWD	(MSB of AX) →(All bits of DX)

Examples:

1. Assume that each instruction starts these values, AL=85H, BL = 35H, AH= 0H.
   MUL BL→ AL. BL= 85H*35H = 1B89H → AX = 1B89H)
2. IMUL BL →AL.BL = 85H-2'S AL= 2'S(85H) *(35H)
   = 7BH*35H = 1977H→ 2'S comp→ E689H→AX

AH AL

3. DIV BL→ $\frac{AX}{BL} = \frac{0085H}{35H}$ = 02 (85-02*35=1B) → $\boxed{1B \quad 02}$

4. IDIV BL→ $\frac{AX}{BL} = \frac{0085H}{35H} =$

AH	AL
1B	02

Example 2: Al= F3H, BL=91H, AH=00H

1. MUL BL→ AL*BL = F3H* 91H = 89A3H→AX = 89A3
2. IMUL BL→ AL*BL = 2'S AL *2'S BL = 2'S(F3H)* 2'S(91H)
   = 0DH*6FH = 05A3H
3. IDIV BL →$\frac{AX}{BL} = \frac{00F3H}{2's(91H)} = \frac{00F3H}{6FH}$ = 2→(00F3- 2 *6F=1FH)

AH	AL
15	02
R	Q

$\rightarrow \frac{pos}{NEG} = NEG --\rightarrow 2's(o2) = FEH \rightarrow$

AH	AL
15	FE

4. DIV BL →$\frac{AX}{BL} = \frac{00F3H}{91H}$ = 01→(F3-1*91 = 62)

AH	AL
62	01
R	Q

AAA (ASCII Adjust after Addition)
- The data entered from the terminal in ASCII format.
- In ASCII, 0-9 are represented by 30H-39H
- This instruction allowed to add the ASCII codes
- This instruction does not have any operand.

Other ASCII Instructions:
- AAS (ASCII Adjust after Subtraction)
- AAM (ASCII Adjust after Multiplication)
- AAD (ASCII Adjust after Division)

DAA (Decimal Adjust after Addition)
- It is used to make sure that the result of adding two BCD numbers is adjusted to be a correct BCD number
- It only works on AL register

DAS (Decimal Adjust after Subtraction)
- It is used to make sure that the result of subtracting two BCD numbers is adjusted to be a correct BCD number
- It only works on AL register

NEG Source

- It creates 2's complement of a given number. That means, it changes the sign of the number

CMP Destination, Source

- It compares two specified bytes or words.
- The source and the destination can be a constant, a register or a memory location.
- Both operands cannot be a memory location at the same time.
- The comparison is done simply by internally subtracting the source from the destination.
- The value of the source and the destination does not change. But the flags are updated to indicate the result.

CBW (Convert Byte to Word)

- This instruction converts a byte in AL to Word into AX.
- The conversion is done by extending the sign bit of AL throughout AH.\

CWD (Convert Word to Byte)

- This instruction converts a word in AX to Double word in DX:AX.
- The conversion is done by extending the sign bit of AX throughout DX.

3. Bit Manipulation Instructions

These instructions are used to perform operations on bit level. These instructions are used for testing a zero bit, set or reset a bit, shift bit across registers. Table 18 shows the classification of these instruction.

Table 18 bit manipulation instructions

Logical instructions	Shift instruction	Rotate instructions
NOT	SHL/SAL	ROL
AND	SHR	ROR
OR	SAT	RCL
XOR		RCR
TEST		

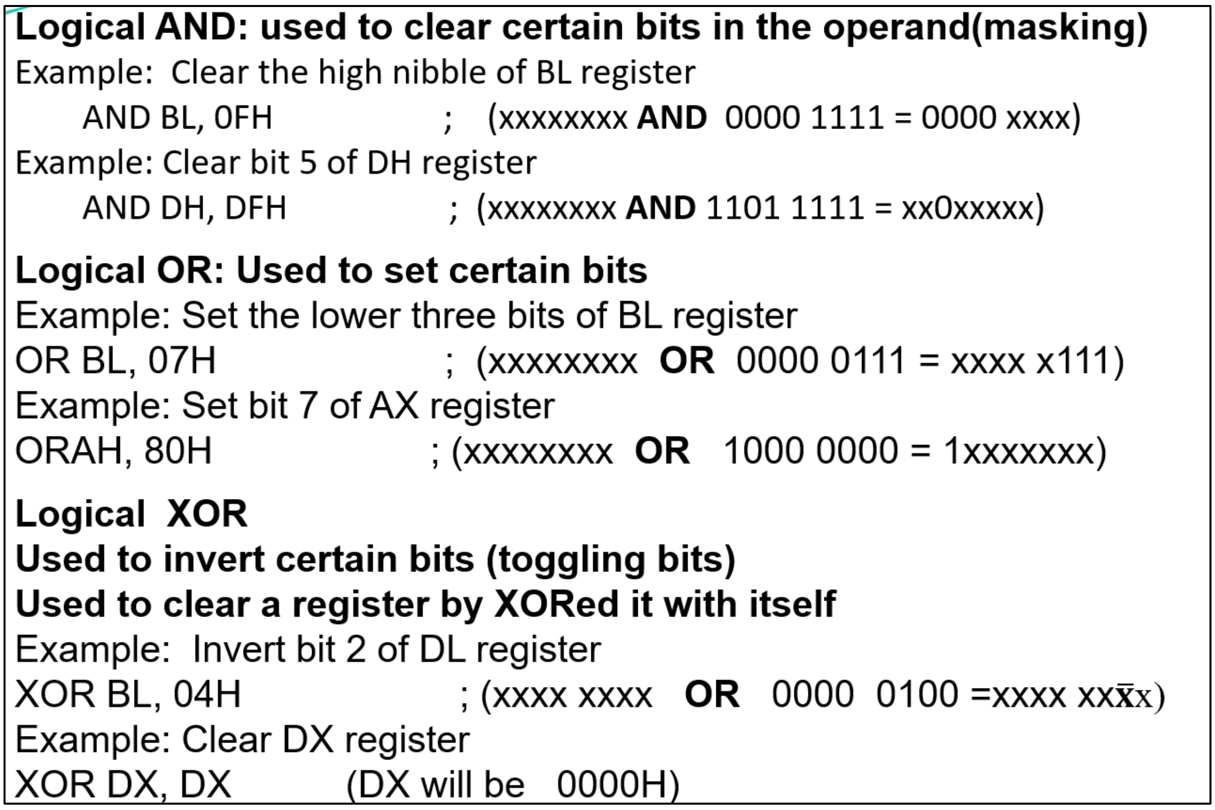
a) Logical instructions

Details about these instructions is provided in table 19.

Table 19 logical instructions

Mnemonic	Meaning	Format	Operation	Flags effected
NOT	Logical NOT	NOT D	(D-) → (D)	Non
AND	Logical AND	AND D, S	(S).(D) → (D)	O, S, Z, P, C
OR	Logical Inclusive OR	OR D, S	(S)+(D) →(D)	O, S, Z, P, C
XOR	Logical Exclusive OR	XOR D, S	(S) ⊕ (D) → (D)	O, S, Z, P, C

Examples of logical instructions are shown in figure 29



Logical AND: used to clear certain bits in the operand(masking)

Example: Clear the high nibble of BL register

```
      AND BL, 0FH              ;   (xxxxxxxx AND  0000 1111 = 0000 xxxx)
```

Example: Clear bit 5 of DH register

```
      AND DH, DFH              ;  (xxxxxxxx AND 1101 1111 = xx0xxxxx)
```

Logical OR: Used to set certain bits

Example: Set the lower three bits of BL register

```
OR BL, 07H                 ;  (xxxxxxxx  OR  0000 0111 = xxxx x111)
```

Example: Set bit 7 of AX register

```
ORAH, 80H                 ; (xxxxxxxx  OR   1000 0000 = 1xxxxxxx)
```

Logical XOR

Used to invert certain bits (toggling bits)

Used to clear a register by XORed it with itself

Example: Invert bit 2 of DL register

XOR BL, 04H ; (xxxx xxxx **OR** 0000 0100 =xxxx xx$\bar{x}$x)

Example: Clear DX register

```
XOR DX, DX          (DX will be   0000H)
```

Figure 29 examples of logical instructions

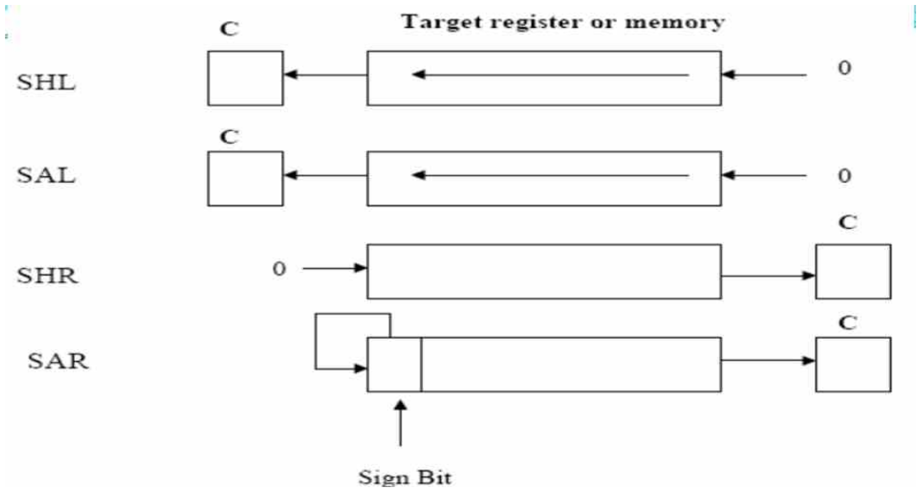
Shift instructions

These instructions are used to

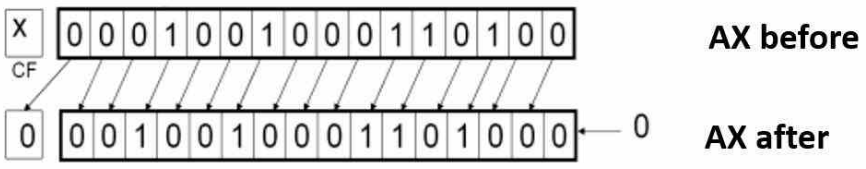
- Align data
- Isolate bit or a byte of a word so that it can be tested
- Perform simple multiplication and division computations. The meaning and the format of these instruction is shown table 20.
-

Table 20 shift instruction format

Mnemonic	Meaning	Format	Operation	Flags effected
SAL/SHL	Shift arithmetic left/shift logical left	SAL D, Count 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	C, P, S, Z A undefined O undefined if count =1
SHR	shift logical right	SHR D, Count	Shift the (D) right by the number of bit positions equal to Count and fill the vacated bits positions on the left with zeros	C, P, S, Z A undefined O undefined if count =1
SAR	shift arithmetic right	OR D, S	Shift the (D) right by the number of bit positions equal to Count and fill the vacated bits positions on the right with the original most significant bit	C, P, S, Z A undefined O undefined if count =1

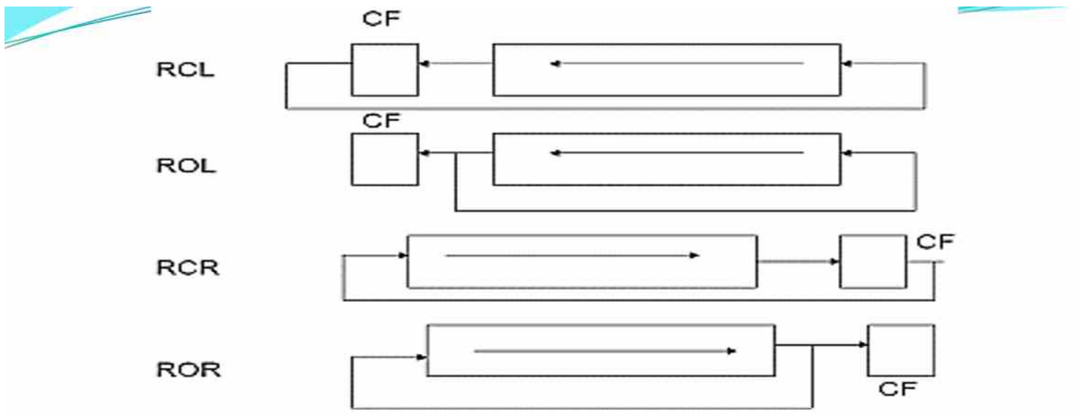


Examples on shift instructions
Let AX= 1234H what is the value of AX after execution of this instruction AHL AX, 1
Solution:



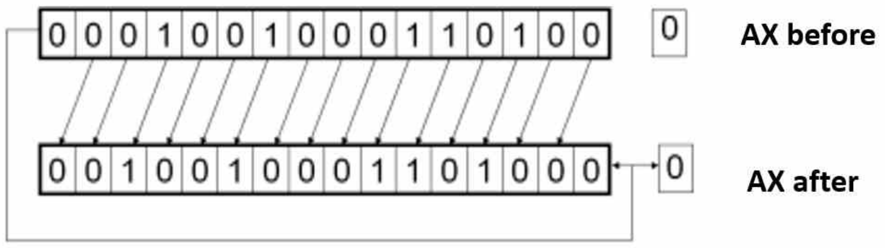
Rotate Instructions

Mnemonic	Meaning	Format	Operation	Flags effected
ROL	Rotate left	ROL D, Count	rotate the (D) left by the number of bit positions equal to Count. Each bit shifted out from the leftmost goes back into the rightmost bit position.	C, O undefined if count =1
ROR	Rotate right	ROR D, Count	Rotate the (D) right by the number of bit positions equal to Count. Each bit shifted out from the rightmost goes back into the leftmost bit position	C and O undefined if count =1
RCL	Rotate left through carry	RCL D, S	Same as ROL with an addition that the carry is attached to (D) for rotation.	C and O undefined if count =1
RCR	Rotate right through carry	RCR D, Count	Same as ROR with an addition that the carry is attached to (D) for rotation.	C and O undefined if count =1



Example: Assume AX = 1234H , what is the result of executing the instruction
ROL AX, 1

Solution:

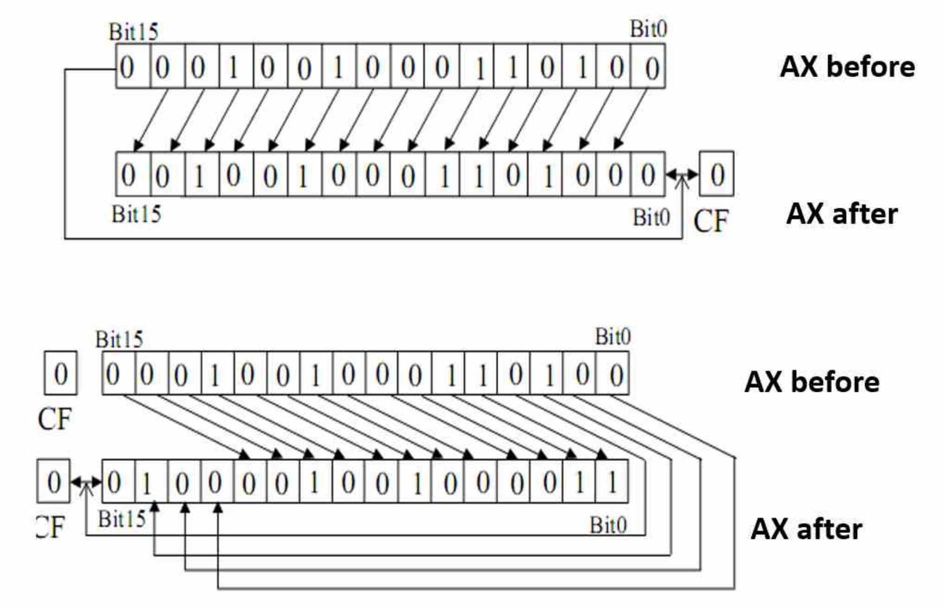


Example: if CL =04H and AX=1234H. determine the new content of AX and the carry flag after executing the instruction

a) ROL AX, 1

b) ROR AX, CL

solution



Process Control Instructions

These instructions are used to control the processor action by setting/resetting the flag values. They are described in details in table 21.

Table 21 Flag Instruction

Mnemonic	Meaning	Format	Operation	Flags effected
CLC	Clear Carry Flag		(CF)← 0	CF
STC	Set Carry Flag		(CF)← 1	CF
CMC	Complement Carry Flag		(CF)← (CF)’	
CLD	Clear Direction Flag		(DF)← 0 SI& DI will be auto incremented while string instruction are executed	DF
STD	Set Direction Flag		(DF)← 1	DF
CLI	Clear Interrupt Flag		(IF)← 0	IF
STI	Set Interrupt Flag		(IF)← 1	IF

Test instruction

This instruction is similar to AND instruction with a difference that the AND instruction change the destination operand while test instruction does not change it. The instruction affects the condition of the flag register, which indicates the test result. The same addressing modes that used with AND instruction are used with TEST instruction.

5. String Instructions

String is a group of bytes/words and their memory is always allocated in a sequential order. Table 22 shows the instructions under this group.

Notes: consider the following tips when deal with string instructions.

- The source (***DS:SI***), the destination (***ES:DI***).
- You must ensure that SI and DI are offsets into DS and ES respectively.
- The direction flag (0=up, 1=down)
  5. CLC-increment address (left to right)
  6. STD- decrement address (right to left)

Table 22 string instructions

Mnemonic	Meaning	Format	Operation	Flags effected
MOVS	Move string	MOVSB/ MOVEW	**((DS)*10+(SI))→((ES)*10+(DI)) (SI)± 1→(SI);(DI)± 1→(DI)[byte] (SI)± 2→(SI);(DI)± 2→(DI)[word]**	NONE
CMPS	Compare string	CMPS/ CMPW	**((DS)*10+(SI)) - ((ES)*10+(DI)) (SI) ± 1→(SI);(DI) ± 1→(DI)[byte] (SI) ± 2→(SI);(DI) ± 2→(DI)[word]**	O,S,Z,A,P, C
SCAS	Scan string	SCASB/ SCASW	**(AL) or (AX) – ((ES)*10+ (DI)) (DI) ± 1→(DI)[byte] (DI) ± 2→(DI)[word]**	O,S,Z,A,P, C
LODS	Load string	LODSB/ LODSW	**((DS)*10+(SI)) → (AL) or (AX)) (SI) ± 1→(SI)[byte] (SI) ± 2→(SI)[word]**	NONE
STOS	Store string	STOSB/ STOSW	**(AL) or (AX)) → (ES)*10+(DI) (DI) ± 1→(DI)[byte] (DI) ± 2→(DI)[word]**	NOOE

Executing these instruction causes the address indices in SI and DI to be either increased or decreased automatically. The status of direction flag during the string operations determines whether to increase or decrease the SI and DI. When (D=0) it will be auto incremented while (D=1) decide to be decremented.

In most applications, the string operations must be repeated in order to process arrays of data. The following repeat prefixes are used to repeat the instructions.

- **REP** – is used with (MOVS, STOS, LODS) to repeat them while not end of the string CX ≠ 0.
- **REPE/REPZ** – is used with (CMPS, SCAS) instructions to repeat them while not the end of the string and strings are equal, CX ≠ 0 or zero flag ZF = 1.
- **REPNE/REPNZ** – is used with (CMPS, SCAS) to repeat them while not the end of the string and strings are not equal CX ≠ 0 or zero flag ZF = 0.
- **INS/INSB/INSW** − Used as an input string/byte/word from the I/O port to the provided memory location.
- **OUTS/OUTSB/OUTSW** − Used as an output string/byte/word from the provided memory location to the I/O port.

6. Control Transfer Instructions

These instructions are used to branch the instructions during an execution, i.e. transfer the program control from one address to other address (not in sequence address). The classification of these instructions is shown in table 23.

Table 23 control transfer instructions

Unconditional Transfer Instructions	Conditional Transfer Instructions		Iteration Control Instructions	Interrupt Instructions
JMP	JA/JNBE	JLE/JNG	LOOP	INT
CALL	JAE/JNB	JNC	LOOP/LOOPZ	INTO
RET	JB/JNAE	JNE/JNZ	LOOPNE/LOOPNZ	IRET
	JBE/JNA	JNO		
	JC	JNP/JPO		
	JE/JZ	JNS		
	JG/JNLE	JO		
	JGE/JNL	JP/JPE		
	JL/JNGE	JS		

Jump Instructions

Two types of jump operation are allowed in 8086. Conditional jump and unconditional jump.

a) Unconditional Jump

The instruction is (JMP) that means unconditional jump. A simple illustration of unconditional jump is shown in figure 30.

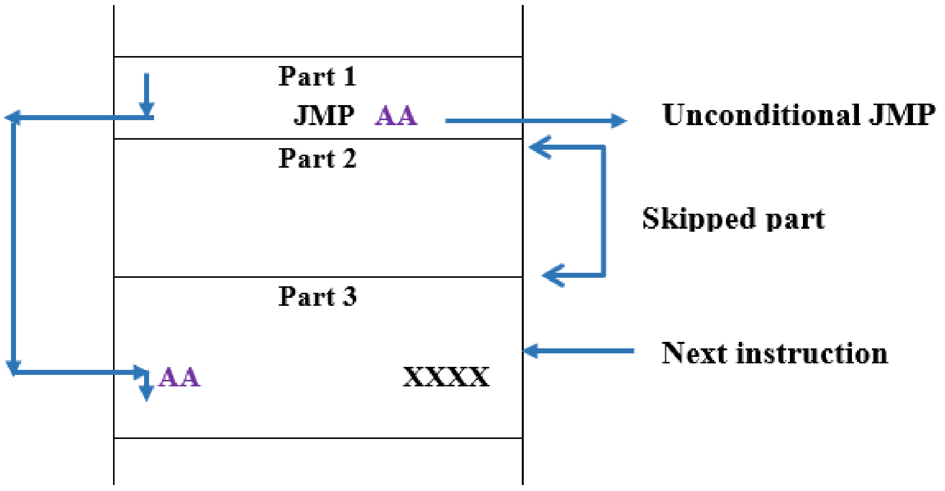
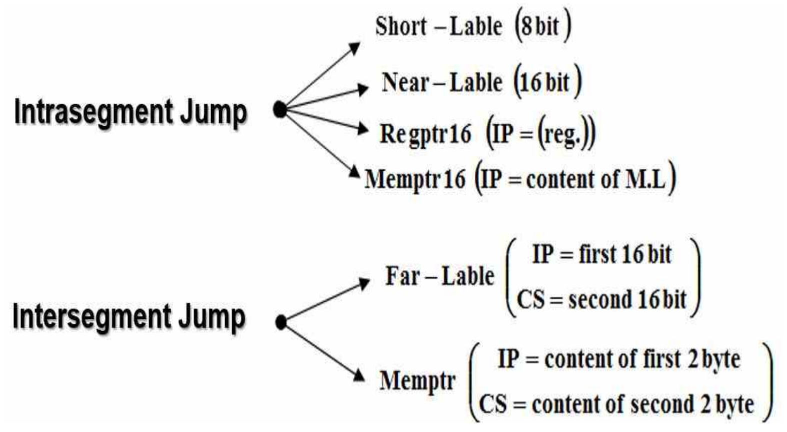


Figure 30 unconditional jump

The unconditional jump instruction has two types:

- **Intrasegment jump**: which is short, near jump instruction. It is limited to addresses within the current code segment and achieved by just modifying the value in IP
- **Intersegment jump**: far jump instruction that requires modification of the content of both CS and IP. It permits jump from one code segment into another.


Intrasegment Jump
Short – Lable (8 bit)
Near – Lable (16 bit)
Regptr16 (IP = (reg.))
Memptr16 (IP = content of M.L)
Intersegment Jump
Far – Lable (IP = first 16 bit, CS = second 16 bit)
Memptr (IP = content of first 2 byte, CS = content of second 2 byte)

Example:

Assume the following state of 8086: (CS)=1075H, (IP)=0300H, (SI)=A00H, (DS)=400H, (DS:A00)=10H. (DS:A01)=B3H, (DS:A02)=22H, (DS:A03)=1AH. To what address is the program control pass if each of the following JMP instruction is executed:

a) JMP 85 **b) JMP 1000H** **c) JMP [SI]**

d) JMP SI **e) JMP FAR[SI]** **f) JMP 3000:1000**

Solution:

1. JMP 85	→	1075: 85	→	short jump
2. JMP 1000H	→	1075: 1000	→	Near jump
3. JMP [SI]	→	1075:B310	→	Near jump
4. JMP SI	→	1075: 0A00	→	Near jump
5. JMP FAR[SI]	→	1A22:B310	→	Far jump
6. JMP 3000:1000	→	3000:1000	→	Far jump

b) Conditional Jump

This type of jump instructions tests the flags bits (S, Z, C, P and O). if the tested condition is TRUE, then a brunch to the label associated with jump instruction occurs. Otherwise, the next subsequential step in the program will be execute. An illustration of unconditional jump is shown in figure 31.

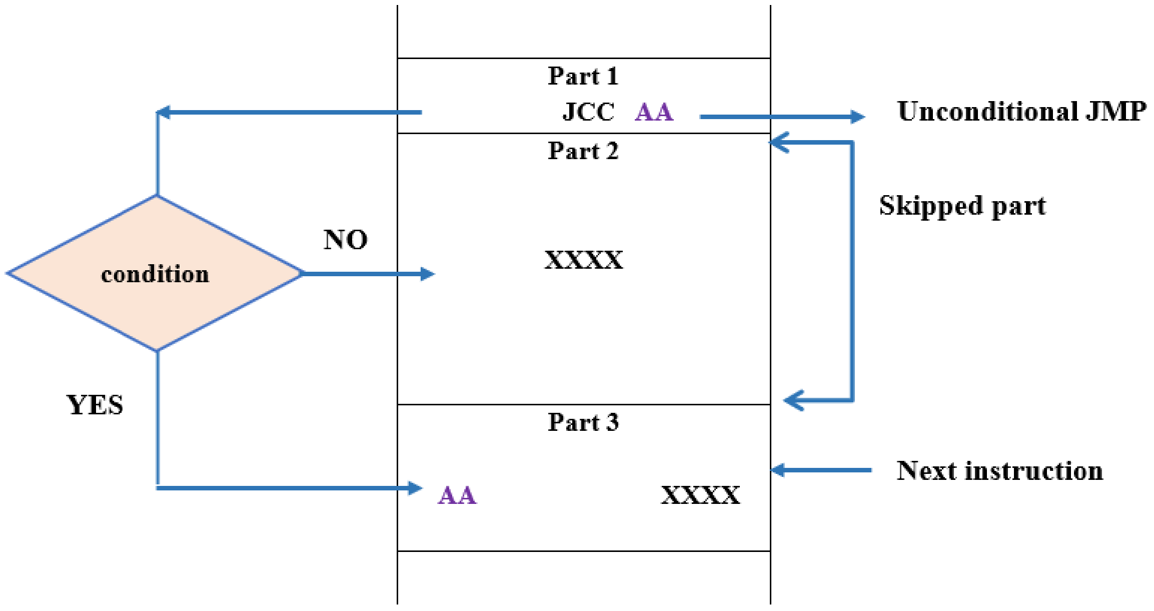


Figure 31 Unconditional jump

These instructions are explained below.

- **JMP** – Used to jump to the provided address to proceed to the next instruction.
- **JA/JNBE** – Used to jump if above/not below/equal instruction satisfies.
- **JAE/JNB** – Used to jump if above/not below instruction satisfies.
- **JBE/JNA** – Used to jump if below/equal/ not above instruction satisfies.

- **JC** – Used to jump if carry flag CF = 1.
- **JE/JZ** – Used to jump if equal/zero flag ZF = 1.
- **JG/JNLE** – Used to jump if greater/not less than/equal instruction satisfies.
- **JGE/JNL** – Used to jump if greater than/equal/not less than instruction satisfies.
- **JL/JNGE** – Used to jump if less than/not greater than/equal instruction satisfies.
- **JLE/JNG** – Used to jump if less than/equal/if not greater than instruction satisfies.
- **JNC** – Used to jump if no carry flag (CF = 0)
- **JNE/JNZ** – Used to jump if not equal/zero flag ZF = 0
- **JNO** – Used to jump if no overflow flag OF = 0
- **JNP/JPO** – Used to jump if not parity/parity odd PF = 0
- **JNS** – Used to jump if not sign SF = 0
- **JO** – Used to jump if overflow flag OF = 1
- **JP/JPE** – Used to jump if parity/parity even PF = 1
- **JS** – Used to jump if sign flag SF = 1
- **JCXZ** – Used to jump to the provided address if CX = 0

Call Subroutine Instructions

A subroutine is a special segment of the program that can be called for execution from any point in the program. Two basics instructions for handling the subroutine which are.

- **CALL** this instruction is used to call a subroutine and save their return address to the stack.
- **RET** this instruction is used to return from the subroutine to the main program. It should be included at the end of the subroutine to initiate the return sequence to the main program environment.

Every subroutine must be ended by executing of (RET) instruction which returns the control to the main program. Executing this instruction causes the original value of IP and CS to be POPOed from the stack. The illustration of subroutine and how to handle it is figured out in figure 32.

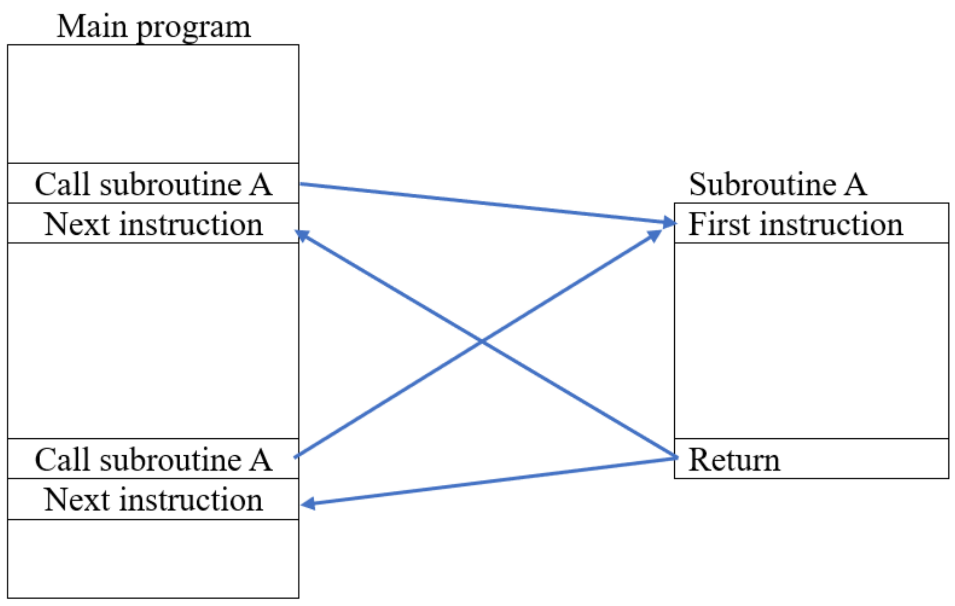


Figure 32 subroutine handling

Example: write a procedure to square the content of BL and places the result in BX.
Solution:
Let we name the procedure Square, the procedure is as follow

```
Square      PUSH AX
            MOV AL, BL
            MUL BL
            MOV BX, AX
            POP AX
            RET
```

Now we can call this procedure to compute y= $(AL)^2 + (AH)^2 + (DL)^2$ and place the result in CX (assuming that y doesn't exceed 16bit)
The code will be.

```
MOV CX, 0000H
MOV BL, AL
CALL Square
ADD CX, BX
MOV BL, AH
CALL Square
ADD CX, BX
MOV BL, DL
CALL Square
ADD CX, BX
HLT
```

Loop & Iteration Instructions

These instructions are used to execute the given instructions for number of times. Following is the list of instructions under this group −

- **LOOP** − Used to loop a group of instructions until the condition satisfies, i.e., CX = 0
- **LOOPE/LOOPZ** − Used to loop a group of instructions till it satisfies ZF = 1 & CX = 0
- **LOOPNE/LOOPNZ** − Used to loop a group of instructions till it satisfies ZF = 0 & CX = 0

Interrupt Instructions

These instructions are used to call the interrupt during program execution.

- **INT** − Used to interrupt the program during execution and calling service specified.
- **INTO** − Used to interrupt the program during execution if OF = 1
- **IRET** − Used to return from interrupt service to the main program.

Addressing Modes

Information involved in any operation performed by the CPU needs to be addressed. In computer terminology, such information is called the operand. Therefore, any instruction issued by the processor must carry at least two types of information. These are the operation to be performed, encoded in what is called the op-code field, and the address information of the operand on which the operation is to be performed, encoded in what is called the address field.

The different ways in which operands can be addressed are called the ***addressing modes.*** Addressing modes differ in the way the address information of operands is specified.

There are 8 different addressing modes in 8086 programming:

1. Immediate addressing mode.
2. Register addressing mode.
3. Direct addressing mode.
4. Register indirect addressing mode.
5. Based addressing mode.
6. Indexed addressing mode.
7. Based-index addressing mode.
8. Based indexed with displacement mode.

These modes are explained as follows.

1. Immediate addressing mode

Immediate addressing transfers data from a register or memory location to a destination register or memory location. the data operand is a part of the instruction itself. The data can be of different types: decimal data is represented as it is without adding any symbol or letter, the hexadecimal data which is indicated by adding **H** at the end of the data value, the ASCII coded data is represented by enclosing it in the apostrophes ‘ **’,** which differs from the single quote ’ **’,** the binary data is represented by adding B at the end. Below examples of different versions of MOV instruction shown in table 24.

Table 24 immediate addressing mode examples

Instruction	Size	Operation
MOV AL, 20	**8 bit**	**Copies 20 decimal (14H) into register AL**
MOV BL, 44	**8 bit**	**Copies 44 decimal (2CH) into register BL**
MOV AX, 44H	**16 bit**	**Copies 0044H into register AX**
MOV BX, 55H	**16 bit**	**Copies a 0055H into register BX**
MOV SI, 0	**16 bit**	**Copies 0000H into register SI**
MOV CH, 100	**8 bit**	**Copies 100 decimal (64H) into register CH**
MOV AL, 'A'	**8 bit**	**Copies ASCII of A into register AL**
MOV AH, 1	**8 bit**	**Copies 1 decimal (01H) into register AH**

Instruction	Size	Operation
MOV AX, 'AB'	16 bit	Copies ASCII of AB into register AX
MOV DX, 'Ahmed'	16 bit	Copies an ASCII Ahmed into register DX
MOV CL, 11001110B	8 bit	Copies 11001110 binary into register CL

Note that if the hexadecimal data begins with a letter, the assembler requires that the data start with a 0. For example, to move F2H to AL, the instruction will be **MOV AL, 0F2H**

2. Register addressing mode

This addressing mode transfers data from the source register or a memory location to the destination register or a memory location. This is one of the fastest addressing modes because the CPU can access registers very quickly. Registers are located directly within the CPU, so there's no need to fetch data from main memory.

Notes:

- There is no need to compute the effective address because the operand is in a register and no memory access involved.
- CS (code segment register) ***cannot be*** as the destination operand.
- You cannot move data from one segment register to another with a single MOV instruction. So, only one of the operands can be a segment register.
- Segment registers cannot be used to hold arbitrary values, they contain segment address only.
- Transfer data should be done within registers of the same sizes. Different registers sizes are not allowed. For example, **MOV AX, AL** will indicate an error because AX is 16-bit register while AL is 8-bit register.

Table 25 shows some examples of register addressing mode

Table 25 register addressing mode examples

Instruction	Size	Operation
MOV AL, 20	8 bit	Copies 20 decimal (14H) into register AL
MOV BL, 44	8 bit	Copies 44 decimal (2CH) into register BL
MOV AX, 44H	16 bit	Copies 0044H into register AX
MOV BX, 55H	16 bit	Copies a 0055H into register BX
MOV SI, 0	16 bit	Copies 0000H into register SI
MOV CH, 100	8 bit	Copies 100 decimal (64H) into register CH
MOV AL, 'A'	8 bit	Copies ASCII of A into register AL
MOV AH, 1	8 bit	Copies 1 decimal (01H) into register AH
MOV AX, 'AB'	16 bit	Copies ASCII of AB into register AX
MOV DX, 'Ahmed'	16 bit	Copies an ASCII Ahmed into register DX
MOV CL, 11001110B	8 bit	Copies 11001110 binary into register CL

3. Direct addressing mode

In this type of the addressing modes, the address of the memory location that holds the operand is included in the instruction. **Figure 33** shows a simple illustration of this mode. There are two basic forms of direct data addressing:

Direct addressing: this form specifically involves transferring data between a memory location and the AL(8-bit) or AX (16-bit0. The instruction directly contains the memory address of the data.

Example: **MOV AX, [1234H]** (Move the word from memory location 1234H into the AX register).

- The effective address is the displacement value.
- The segment register that is used by default is the data segment register (DS).

Displacement addressing: this is a more versatile form that can be used with almost any instruction that involves memory access. It involves a ***displacement value*** (a constant offset) that is added to the base address. The base address is typically determined by the default data segment (DS) or can be overridden by specifying another segment register (CS, SS, ES, FS, GS).

Example: **MOV BX, [DI + 10H]** (Move the word from the memory location pointed to by **DI + 10H** into the BX register).

- In that example, 10H is the displacement.
- DI is an index register, and by default, if an index register is used without a segment override, the DS register is used.

Example of segment override:

MOV BX, ES:[DI + 10H] (In this case, the ES segment register is used)

- The effective address is the sum of the displacement and the contents of the index or base register, and then offset by the segment register.

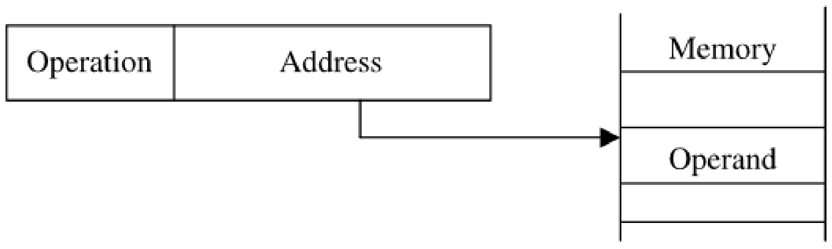


Figure 33 illustration of direct addressing mode

Table 26, some examples of using direct and displacement addressing mode

Table 26 direct addressing mode instructions

Instruction	Size	Operation
MOV AL, NUMBER	**8 bit**	**Copies the byte contents of data segment memory location NUMBER into register AL**
MOV AX, COW	**16 bit**	**Copies the word contents of data segment memory location COW into register AX**
MOV NEWS, AL	**8 bit**	**Copies AL into byte memory location NEWS**
MOV THERE, AX	**16 bit**	**Copies AX into word memory location THERE**
MOV ES:[2000H], AL	**8 bit**	**Copies AL into extra segment memory at offset address 2000H**
MOV AL, MOUSE	**8 bit**	**Copies the contents of location MOUSE into**
MOV CH, DOG	**8 bit**	**Copies the byte contents of data segment memory location DOG into register CH**
MOV CH, DS:[1000H]	**8 bit**	**Copies the byte contents of data segment memory offset address 1000H into register CH**
MOV ES, DATA6	**16 bit**	**Copies the word contents of data segment memory location DATA6 into register ES**
MOV DATA7, BP	**16 bit**	**Copies BP into data segment memory location DATA7**
MOV NUMBER, SP	**16 bit**	**Copies SP into data segment memory location NUMBER**

4. Register indirect addressing mode

transfers a byte or word between a register and a memory location through an offset address held by in any of the following registers: BP, BX, DI & SI.

The DS is used by default with register indirect addressing or any other addressing mode that uses BX, DI, or SI to address memory. If the BP register addresses memory, the SS is used by default. For example, the **MOV AL, [DI]** instruction is clearly a byte-sized move instruction, but the MOV [DI], 10H instruction is ambiguous. Does the MOV [DI], 10H instruction address a byte-, or word-sized memory location? The assembler can't determine the size of the 10H. The instruction MOV DI],10H clearly designates the location addressed by DI as a byte-sized memory location.

The [BX], [SI], and [DI] modes use the DS segment by default (1000H). The [BP] addressing mode uses the stack segment (SS) by default (3400H). You can use the segment override prefix symbols if you wish to access data in different segments. The following instructions demonstrate the use of these overrides:

```
MOV AL, CS:[BX]
        MOV AL, DS:[BP]
        MOV AL, SS:[SI]
        MOV AL, ES:[DI]
```

For example: **MOV SI, 1234H**

MOV AL, [SI]

If SI contains 1234H and DS contains 0200H the result produced by executing the instruction is that the contents of the memory location at address:

PA = 02000H + 1234H = 03234 are moved to the AL register

5. Based addressing mode

In this addressing mode, the offset address is located in the base registers (BX/BP) and deals with an 8-bit/16-bit displacement number. In this type of the addressing mode the offset address that is found in (BX/BP) are used to access the memory locations. The effective memory address is calculated by adding the contents of a base register to a displacement value.

Example 1

```
ORG 100h
MOV DX, [BX+4]
MOV AX, [BX+7]
ADD CL, [BX+8]
RET
```

The registers address can be increased using another way as shown in example2

Example 2

```
ORG 100h
MOV DX, [BX]+4
MOV AX,7[BX]
ADD CL, [BX]+8
RET
```

Q:/What is the displacement number?

The displacement is an 8 bit/16-bit number that is contained in the instruction.

6. Indexed addressing mode

In this addressing mode, the offset address is located in the index registers (SI/DI) and deals with an 8-bit/16-bit displacement number. In this type the offset address that is found in (SI/DI) is used to access the memory locations. It is used to access the one-dimensional array items

Example

```
MOV AL, [DI+16]
MOV [DI]+12, AL
MOV BX, [SI+16]
MOV [SI]+10, BX
ADD [SI],5
INC SI          ; Incremented   SI by 1
DCR SI          ; Decremented SI by 1
```

Example: Program to print an array using loop

```
    . STACK 100                    ;stack segment
    .DATA                          ;data segment
    ARR1 DB 'A','B','C','D'     ; Array definition
    CODE                           ;code segment
    MOV AX, @DATA              ; Initialize data segment
    MOV DS, AX
    LEA SI, ARR1                 ; Storing the offset address of array arr1 in
    SI
    MOV CX, 4                    ; Initializing counter register loop to 4
    C: MOV DX, [SI]                ; Storing address value at DX
    INC SI                       ; Incrementing  SI by 1
    MOV AH, 2                    ; Printing DX Contain
    INT 21H
    LOOP   C        ; loop to c and decrement CX by 1 until CX =0 then stop
    MOV AH,4CH
    INT 21H
END
```

Q3:/ What is the main difference between base and index register addressing modes?

Answer

In the 8086 through the 80286, this type of addressing uses one base register (BP/BX), and one index register (DI/SI) to indirectly address memory. The base register often holds the **beginning of array** locations in memory, while the index register holds the relative **position of an element in the array**.

7. Based-index addressing mode

In this addressing mode, the offset address of the operand is computed by summing the base register to the contents of an Index register.

This addressing mode is used to handle one-dimensional arrays (including character strings). The base register contains the memory address of the first element of the array, and the index register contains the offset of the desired element. By adding them together (base + offset), the desired element is .accessed

Example

ADD CX, [BX+SI]
MOV AX, [BX+DI]
; Or
MOV AX, [BX][DI]

Note that: BX and BP can be used together in the same instruction. Also, SI and DI cannot be used together in one instruction. Such as shown in the following ***wrong examples***.

MOV AX, [BX+BP]
MOV AX, [SI+DI]

8. Based indexed with displacement mode.

In this addressing mode, the operands offset is computed by adding the base register contents. An Index registers contents and 8 or 16-bit displacement number. This addressing mode is used to handle two-dimensional arrays.

Example

MOV AX, [BX+DI+08]
ADD CX, [BX+SI+16]
MOV AX, [BX+SI+20]
MOV AX, [SI+BX+20]
Or
MOV AX, [SI][BX]+20

Q:/ How is Effective Address (EA) calculated in 8086?

Answer

The (EA) is used as an offset for the physical address of the destination data. In 8086 we have base and index registers. Thus, the execution unit (EU) calculates the effective address by adding the displacement number to the contents of the base and index registers as in the following equation:

EA = [Base Register] + [Index Register] + [8/16-bit displacement]

Introduction To Assembly Language

Assembly language unlocks the secret of your computer's hardware and software. It teaches you about the way the computer's hardware and operating system work together and how, the application programs communicate with the operating system. Assembly language, unlike high level languages, is machine dependent. Each microprocessor has its own set of instructions, that it can support. Here we will discuss, only the IBM-PC assembly language. It consists of the Intel 8086/8088 instruction set. The instructions for the Intel 8088 may be used without modification on all its enhancements - 80186,80286,80386,80486 and Pentium.

Why learn Assembly Language?

You must learn assembly language for various reasons:

1. It helps you understand the computer architecture and operating system.
2. Certain programs, requiring close interaction with computer hardware, are sometimes difficult or impossible to do in high level languages. Example: a telecommunication program for the IBM-PC.
3. High level languages, out of necessity, impose rules about what is allowed in a program. For example, Pascal does not allow, a character value to be assigned to an integer variable. Assembly language, in contrast, has very few restrictions or rules; nearly everything is left to the discretion of the programmer. The price for such freedom is the need to handle many details that would otherwise be taken care by the programming language itself.
4. One of the most important advantages of assembly language, is that the programs written in assembly language are at least 30% dense than the same program written in high level language. The reason for this is, that as of today the compilers are still not so intelligent to take advantage of some of the complex instructions of the assembly language. Example: if you write a high-level program to compare two strings, it will translate the code, using simple instruction like MOV, CMP, JMP etc. While the same thing can be written in assembly, by using REPE and CMPSB. Obviously, the code is much smaller.

We can summaries the above reasons by:

1. Accessibility to system hardware.
2. Space and time efficiency.

Assembly Language Syntax and Program Structure

Introduction

A processor can directly execute a machine language program. Though it is possible to program directly in machine language, assembly language uses mnemonics to make programming easier. An assembly language program uses mnemonics to represent symbolic instructions and the raw data that represent variables and constants.

A machine language program consists of: a list of numbers representing the bytes of machine instructions to be executed and data constants to be used by the program.

Assembly Language Syntax

An assembly language program consists of statements. The syntax of an assembly language program statement obeys the following rules:

- Only one statement is written per line.
- Each statement is either an instruction or an assembler directive.
- Each instruction has an opcode and possibly one or more operands.
- An opcode is known as a mnemonic.
- Each mnemonic represents a single machine instruction.
- Operands provide the data to work with.

Assembler Directives

Pseudo instructions or assembler directives are instructions that are directed to the assembler. Assembler directives affect the generated machine code, but are not translated directly into machine code. Directives can be used to declare variables, constants, segments, macros, and procedures as well as supporting conditional assembly.

In general, a directive contains pseudo-operation code, tells the assembler to do a specific thing, and is not translated into machine code.

Segment directives

Segments are declared using directives. The following directives are used to specify the following segments:

- .stack
- .data
- .code

Stack Segment

- Used to set aside storage for the stack.
- Stack addresses are computed as offsets into this segment.
- Use: .stack followed by a value that indicates the size of the stack.

Data Segment

- Used to set aside storage for variables.
- Constants are defined within this segment in the program source.
- Variable addresses are computed as offsets from the start of this segment.
- Use: .data followed by declarations of variables or definitions of constants.

Code Segment

The code segment contains executable instructions macros and calls to procedures. Use: .code followed by a sequence of program statements.

Memory Models

The memory model specifies the memory size assigned to each of the different parts or segments of a program. There exist different memory models for the 8086 processor.

The *.MODEL* Directive

The memory model directive specifies the size of the memory the program needs. Based on this directive, the assembler assigns the required amount of memory to data and code. Each one of the segments (stack, data and code), in a program, is called a ***logical segment***.

Depending on the model used, segments may be in one or in different physical segments. This directive is placed at the very beginning of the program.

The general structure for this directive is:

.MODEL memory_model

Where memory_model can be :

- TINY
- SMALL
- COMPACT
- MEDIUM
- LARGE
- HUGE

SMALL Model

In the SMALL model all code is placed in one physical segment and all data in another physical segment. In this model, all procedures and variables are addressed as NEAR by pointing to their offsets only.

Instructions

Definition:

An instruction in assembly language is a symbolic representation of a single machine instruction. In its simplest form, an instruction consists of a mnemonic and a list of operands.

A mnemonic is a short alphabetic code that assists the CPU in remembering an instruction. This mnemonic can be followed by a list of operands. Each instruction in assembly language is coded into one or more bytes. The first byte is generally an OpCode, i.e. a numeric code representing a particular instruction. Additional bytes may affect the action of the instruction or provide information about the data needed by the instruction.

Instruction Semantics:

The following rules have to be strictly followed in order to write correct code.

1. Both operands have to be of the same size:

Instruction	Correct	Reason
MOV AX, BL	No	Operands of different sizes
MOV AL, BL	Yes	Operands of same sizes
MOV AH, BL	Yes	Operands of same sizes
MOV BL, CX	No	Operands of different sizes

2. Both operands cannot be memory operands simultaneously:

Instruction	Correct	Reason
MOV i , j	No	Both operands are memory variables
MOV AL, i	Yes	Move memory variable to register
MOV j, CL	Yes	Move register to memory variable

3. First operand, or destination, cannot be an immediate value:

Instruction	Correct	Reason
ADD 2, AX	No	Move register to constant
ADD AX, 2	yes	Move constant to register

Writing a Program

How to write an assembly language program?

These are the steps that should be followed for writing an assembly language program:

1. Define the problem.
2. Write the algorithm.
3. Translate into assembly mnemonics.
4. Test and debug the program in case of errors.

The translation phase consists of the following steps:

- Define type of data the program will deal with.
- Write appropriate instructions to implement the algorithm.

Assembly Language Program Development Tools

Now that you have some idea, about how to go about writing assembly language programs, you might want to write your own programs, and try them out on the machine. To do that, there are some developmental tools required. Let us study them now. The discussion is from the point of view of the end user, and not the system programmer.

7. Editor

An editor is a program which, when run on a system, lets you type in text, and store in a file. This text could also be your assembly language program. There are a number of editors available on PC. The editor helps you type the program in required format. This form of the program is called as the source program. The editor gives you all the flexibility, to insert lines, delete lines, insert words, characters, delete words, characters etc. In short all the features that you can think of while writing text, and more. After the program is typed, it can be stored in some secondary storage, like hard disk, floppy diskette etc, for permanent storage.

2. Assembler

An assembler program is used to translate assembly language mnemonics to the binary code for each instruction. After the complete program has been written, with the help of an editor, it is then assembled with the help of an assembler.

An assembler works in two phases, i.e., it reads your source code two times. In the first pass, the assembler, collects all the symbols defined in the program, along with their offsets, in symbol table. On a second pass through the source program, it produces a binary code for each instruction of the program, and give all the symbols an offset with respect to the segment, from the symbol table.

The assembler generates two files: the object file and the list file. The object file contains the binary code for each instruction in the program. It is created only when your program has been successfully assembled, with no errors. The errors that are detected by the assembler, are called the syntax errors. These are like:

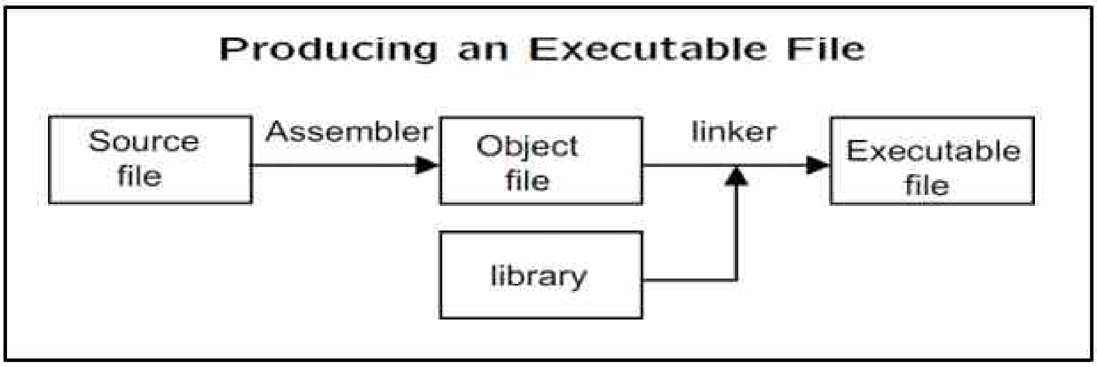
MOVE AX,BX ; undeclared identifier

MOVE. MOV AX,BL ; illegal operands

These are just two of the syntax errors that you can get when your program contains such kind of mistakes. (Exact description of the errors defer from assembler to assembler). In the first statement, it reads the word MOVE, it tries to match with its mnemonics set, as there is no mnemonic with this spelling, it assumes it to be an identifier, and looks for its entry in the symbol table. It does not even find it there, therefore, gives an error 'undeclared identifier'. In the second error, the two operands are of different kind. 8086 expects, both the identifier to be of the same kind, byte or word. But in the above case, one is a byte variable, while the other is a word variable. An assembler does not detect logical errors in your programs, that is your responsibility. List file is optional, and contains, the source code, the binary equivalent of each instruction, and the offsets of the symbols in the program. This file is for documentation purposes. Some of the assemblers available on PC are, MASM (Microsoft Assembler), TASM (TURBO) etc.

3. Linker

For modularity of your program, it is better to break your programs, into several subroutines. It is even better, to put the common routine, like reading a hexadecimal number, writing a hexadecimal number etc., which could he used by a lot of your other programs also, into a separate file. These files are assembled separately. After each, has been successfully assembled, they can be linked together to form a large file, which constitutes your complete program. The file containing the common routines, can be linked to your other programs also. The program that links your programs is called the linker. The linker produces a link file which contains the binary codes for all compound modules. The linker also produces a link map which contains the address information about the linked files. The linker, however, does not assign absolute addresses to your program. It only assigns continuous relative addresses to all the modules linked, starting from zero. This form of program is said to be relocatable, because it can be put anywhere in memory to be run. This form of code can be even be carried to other machines, of the same kind, or compatible to the present machine, to be run successfully. The linker available on your PC is LINK ,TURBO has a built in linker.



4. Loader

Loader is a program, which assigns absolute addresses to the program. These addresses are generated, by adding to all the offsets, the address from where the program is loaded into the memory. Loader comes into action, when you execute your program. This program is brought from the secondary memory, like disk, or floppy diskette, into the main memory at a specific address. Let us assume the program was loaded at address 1000h, then 1000h is added to all the offsets to get the absolute address. Once the program has been loaded, it is now ready to run.

5. Debugger

If your program requires no external hardware or requires hardware directly accessible from your system, then you can use a debugger to debug your program. Debugger allows you to load your program into just like a loader, and, troubleshoot your program. While debugging, you can run your program in single step, set breakpoints, view the contents of registers or memory locations. You can even change the contents of the register or memory location, and run your program with new value. This helps you to isolate the problems in your programs. The problems can be corrected with the help of an editor, and the whole procedure of assembling, linking and executing your program can be repeated. Debugger helps you detect the logical errors, that could not be detected by the assembler.
The following steps showed the process of translating an assembly program into executable file:

1. The *assembler* produces an object file from the assembly language source.
2. The object file contains machine language code with some external and relocatable addresses that will be resolved by the linker. Their values are

undetermined at that stage.

3. The *linker* extract object modules (compiled procedures) from a library and links them with the object file to produce the executable file.
4. The addresses in the executable file are all resolved but they are still virtual addresses.