COMPUTER ORGANIZATION
(PC232CS)

BE IV SEM

(MODEL CURRICULUM)

A.Y 2019-20

By

SANDEEP RAVIKANTI
ASSISTANT PROFESSOR
CSE,MCET

Course Objectives

- To understand basic components of computers.
- To explore the I/O organizations in depth.
- To explore the memory organization.
- To understand the basic chip design and organization of 8086 with assembly language

Course Outcomes

1. After this course students understand in a better way the I/O and memory organization in depth.
2. Ability to understand the merits and pitfalls in computer performance measurements.
3. Identify the basic elements and functions of 8086 microprocessors.
4. Understand the instruction set of 8086 and use them to write assembly language programs.
5. Demonstrate fundamental understanding on the operation between the microprocessor and its interfacing devices.

Suggested Readings:

1. Computer system Architecture: Morris Mano (UNIT-1,2,3).
2. Advanced Micro Processor and Peripherals- Hall/ A K Ray(UNIT-4,5).

Reference

3. Computer Organization and Architecture – William Stallings Sixth Edition, Pearson/PHI.
4. Structured Computer Organization – Andrew S. Tanenbaum, 4th Edition PHI/Pearson.
5. Fundamentals or Computer Organization and Design, - Sivaraama Dandamudi Springer Int. Edition.
6. Computer Architecture a quantitative approach, John L. Hennessy and David A. Patterson, Fourth Edition Elsevier.
7. Computer Architecture: Fundamentals and principles of Computer Design, Joseph D. Dumas II, BS Publication.

UNIT-I

Basic Computer Organization: Functions of CPU, I/O Units, Memory: Instruction: Instruction Formats- One address, two addresses, zero addresses and three addresses and comparison; addressing modes with numeric examples: Program Control-Status bit conditions, conditional branch instructions, Program Interrupts: Types of Interrupts.

Basic Computer Organization

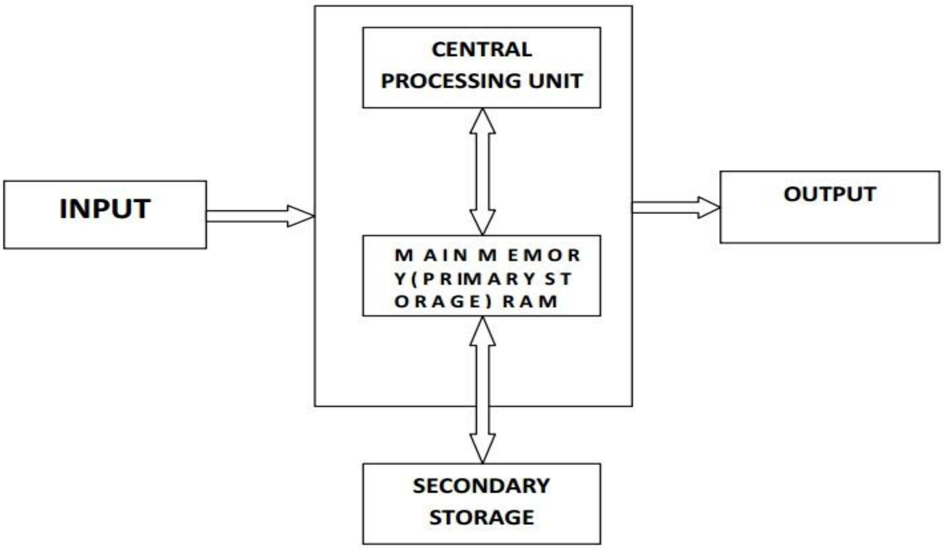


Fig: Block diagram of computer

A standard fully featured desktop configuration has basically four types of featured devices(I/O Units, Memory)

1. Input Devices 2. Output Devices 3. Memory 4. Storage Devices

1. Introduction to CPU
2. CPU
3. The Arithmetic / Logic Unit (ALU)
4. The Control Unit
5. Main Memory
6. External Memory
7. Input / Output Devices
8. The System Bus

CPU OPERATION

The fundamental operation of most CPUs

- To execute a sequence of stored instructions called a program.

1. The program is represented by a series of numbers that are kept in some kind of computer memory.

2. There are four steps that nearly all CPUs use in their operation: fetch, decode, execute, and write back.

3. Fetch:

o Retrieving an instruction from program memory.

o The location in program memory is determined by a program counter (PC)

o After an instruction is fetched, the PC is incremented by the length of the instruction word in terms of memory units.

Decode :

1.The instruction is broken up into parts that have significance to other portions of the CPU.

2.The way in which the numerical instruction value is interpreted is defined by the CPU's instruction set architecture (ISA).

3.Opcode, indicates which operation to perform.

4.The remaining parts of the number usually provide information required for that instruction, such as operands for an addition operation.

5.Such operands may be given as a constant value or as a place to locate a value: a register or a memory address, as determined by some addressing mode.

Execute :

1.During this step, various portions of the CPU are connected so they can perform the desired operation.

2.If, for instance, an addition operation was requested, an arithmetic logic unit (ALU) will be connected to a set of inputs and a set of outputs.

3.The inputs provide the numbers to be added, and the outputs will contain the final sum.

4. If the addition operation produces a result too large for the CPU to handle, an arithmetic overflow flag in a flags register may also be set.

Write back :

1.Simply "writes back" the results of the execute step to some form of memory.

2.Very often the results are written to some internal CPU register for quick access by subsequent instructions.

3.In other cases results may be written to slower, but cheaper and larger, main memory.

Some types of instructions manipulate the program counter rather than directly produce result data.

INPUT DEVICES

Anything that feeds the data into the computer. This data can be in alpha-numeric form which needs to be keyed-in or in its very basic natural form i.e. hear, smell, touch, see; taste & the sixth sense …feel?

Typical input devices are:

1.Keyboard
2. Mouse
3. Joystick
4. Digitizing Tablet
5. Touch Sensitive Screen
6. Light Pen
7. Space Mouse
8.Digital Stills Camera
9. Magnetic Ink Character
10.OpticalMarkReader
Recognition (MICR)   (OMR)
11. Image Scanner
12.Bar Codes

13. Magnetic Reader
14.Smart Cards
15.Voice Data Entry
16.Sound Capture
17. Video Capture

The Keyboard is the standard data input and operator control device for a computer. It consists of the standard QWERTY layout with a numeric keypad and additional function keys for control purposes.

The Mouse is a popular input device. You move it across the desk and its movement is shown on the screen by a marker known as a 'cursor'. You will need to click the buttons at the top of the mouse to select an option.

Track ball looks like a mouse, as the roller is on the top with selection buttons on the side. It is also a pointing device used to move the cursor and works like a mouse. For moving the cursor in a particular direction, the user spins the ball in that direction. It is sometimes considered better than a mouse, because it requires little arm movement and less desktop space. It is generally used with Portable computers.

Magnetic Ink Character Recognition (MICR) is used to recognize the magnetically charged characters, mainly found on bank cheques. The magnetically charged characters are written by special ink called magnetic ink. MICR device reads the patterns of these characters and compares them with special patterns stored in memory. Using MICR device, a large volume of cheques can be processed in a day. MICR is widely used by the banking industry for the processing of cheques.

The joystick is a rotary lever. Similar to an aircraft's control stick, it enables you to move within the screen's environment, and is widely used in the computer games industry.

A Digitizing Tablet is a pointing device that facilitates the accurate input of drawings and designs. A drawing can be placed directly on the tablet, and the user traces outlines or inputs coordinate positions with a hand-held stylus.

A Touch Sensitive Screen is a pointing device that enables the user to interact with the computer by touching the screen. There are three types of Touch Screens: pressure-sensitive, capacitive surface and light beam.

A Light Pen is a pointing device shaped like a pen and is connected to a VDU. The tip of the light pen contains a light-sensitive element which, when placed against the screen, detects the light from the screen enabling the computer to identify the location of the pen on the screen. Light pens have the advantage of 'drawing' directly onto the screen, but this can become uncomfortable, and they are not as accurate as digitising tablets.

The Space mouse is different from a normal mouse as it has an X axis, a Y axis and a Z axis. It can be used for developing and moving around 3-D environments.

Digital Stills Cameras capture an image which is stored in memory within the camera. When the memory is full it can be erased and further images captured. The digital images can then be downloaded from the camera to a computer where they can be displayed, manipulated or printed.

The Optical Mark Reader (OMR) can read information in the form of numbers or letters and put it into the computer. The marks have to be precisely located as in multiple choice test papers.

Scanners allow information such as a photo or text to be input into a computer. Scanners are usually either A4 size (flatbed), or hand-held to scan a much smaller area. If text is to be scanned, you would use an Optical Character Recognition (OCR) program to recognise the printed text and then convert it to a digital text file that can be accessed using a computer.

A Bar Code is a pattern printed in lines of differing thickness. The system gives fast and error-free entry of information into the computer. You might have seen bar codes on goods in supermarkets, in libraries and on magazines. Bar codes provide a quick method of recording the sale of items.

Card Reader: This input device reads a magnetic strip on a card. Handy for security reasons, it provides quick identification of the card's owner. This method is used to run bank cash points or to provide quick identification of people entering buildings.

Smart Card: This input device stores data in a microprocessor embedded in the card. This allows information, which can be updated, to be stored on the card. This method is used in store cards which accumulate points for the purchaser, and to store phone numbers for cellular phones.

OUTPUT DEVICES :

Output devices display information in a way that you can you can understand. The most common output device is a monitor. It looks a lot a like a TV and houses the computer screen. The monitor allows you to 'see' what you and the computer are doing together.

Brief of Output Device

Output devices are pieces of equipment that are used to get information or any other response out from computer. These devices display information that has been held or generated within a computer. Output devices display information in a way that you can understand. The most common output device is a monitor.

Types of Output Device

Printing:	Plotter, Printer
Sound	: Speakers
Visual	: Monitor

A Printer is another common part of a computer system. It takes what you see on the computer screen and prints it on paper. There are two types of printers; Impact Printers and Non-Impact Printers.

Speakers are output devices that allow you to hear sound from your computer. Computer speakers are just like stereo speakers. There are usually two of them and they come in various sizes.

MEMORY OR PRIMARY STORAGE :

Purpose of Storage

The fundamental components of a general-purpose computer are arithmetic and logic unit, control circuitry, storage space, and input/output devices. If storage was removed, the device we had would be a simple calculator instead of a computer. The ability to store instructions that form a computer program, and the information that the instructions manipulate is what makes stored program architecture computers versatile.

Primary Storage

Primary storage is directly connected to the central processing unit of the computer. It must be present for the CPU to function correctly, just as in a biological analogy the lungs must be present (for oxygen storage) for the heart to function (to pump and oxygenate the blood). As shown in the diagram, primary storage typically consists of three kinds of storage:

Processors Register

It is the internal to the central processing unit. Registers contain information that the arithmetic and logic unit needs to carry out the current instruction. They are technically the fastest of all forms of computer storage.

Main memory

It contains the programs that are currently being run and the data the programs are operating on. The arithmetic and logic unit can very quickly transfer information between a processor register and locations in main storage, also known as a "memory addresses". In modern computers, electronic solid-state random access memory is used for main storage, and is directly connected to the CPU via a "memory bus" and a "data bus".

Cache memory

It is a special type of internal memory used by many central processing units to increase their performance or "throughput". Some of the information in the main memory is duplicated in the cache memory, which is slightly slower but of much greater capacity than the processor registers, and faster but much smaller than main memory.

Memory

Memory is often used as a shorter synonym for Random Access Memory (RAM). This kind of memory is located on one or more microchips that are physically close to the microprocessor in your computer. Most desktop and notebook computers sold today include at least 512 megabytes of RAM (which is really the minimum to be able to install an operating system). They are upgradeable, so you can add more when your computer runs really slowly.

The more RAM you have, the less frequently the computer has to access instructions and data from the more slowly accessed hard disk form of storage. Memory should be distinguished from storage, or the physical medium that holds the much larger amounts of data that won't fit into RAM and may not be immediately needed there.

Storage devices include hard disks, floppy disks, CDROMs, and tape backup systems. The terms auxiliary storage, auxiliary memory, and secondary memory have also been used for this kind of data repository.

RAM is temporary memory and is erased when you turn off your computer, so remember to save your work to a permanent form of storage space like those mentioned above before exiting programs or turning off your computer.

TYPES OF RAM:

There are two types of RAM used in PCs - Dynamic and Static RAM.

Dynamic RAM (DRAM): The information stored in Dynamic RAM has to be refreshed after every few milliseconds otherwise it will get erased. DRAM has higher storage capacity and is cheaper than Static RAM.

Static RAM (SRAM): The information stored in Static RAM need not be refreshed, but it remains stable as long as power supply is provided. SRAM is costlier but has higher speed than **DRAM.**

Additional kinds of integrated and quickly accessible memory are Read Only Memory (ROM), Programmable ROM (PROM), and Erasable Programmable ROM (EPROM). These are used to keep special programs and data, such as the BIOS, that need to be in your computer all the time. ROM is "built-in" computer memory containing data that normally can only be read, not written to (hence the name read only).

ROM contains the programming that allows your computer to be "booted up" or regenerated each time you turn it on. Unlike a computer's random access memory (RAM), the data in ROM is not lost when the computer power is turned off. The ROM is sustained by a small long life battery in your computer called the CMOS battery. If you ever do the hardware setup procedure with your computer, you effectively will be writing to ROM. It is non volatile, but not suited to storage of large quantities of data because it is expensive to produce. Typically, ROM must also be completely erased before it can be rewritten,

PROM (Programmable Read Only Memory)

A variation of the ROM chip is programmable read only memory. PROM can be programmed to record information using a facility known as PROM-programmer. However once the chip has been programmed the recorded information cannot be changed, i.e. the PROM becomes a ROM and the information can only be read.

EPROM (Erasable Programmable Read Only Memory)

As the name suggests the Erasable Programmable Read Only Memory, information can be erased and the chip programmed a new to record different information using a special PROM-Programmer. When EPROM is in use information can only be read and the information remains on the chip until it is erased.

STORAGE DEVICES

The purpose of storage in a computer is to hold data or information and get that data to the CPU as quickly as possible when it is needed. Computers use disks for storage: hard disks that are located inside the computer, and floppy or compact disks that are used externally.

Computers Method of storing data & information for long term basis i.e. even after PC is switched off.

It is non - volatile

Can be easily removed and moved & attached to some other device

Memory capacity can be extended to a greater extent

- Cheaper than primary memory

Storage Involves Two Processes

a) Writing data b) Reading data

Floppy Disks

The floppy disk drive (FDD) was invented at IBM by Alan Shugart in 1967. The first floppy drives used an 8-inch disk (later called a "diskette" as it got smaller), which evolved into the 5.25-inch disk that was used on the first IBM Personal Computer in August 1981. The 5.25-inch disk held 360 kilobytes compared to the 1.44 megabyte capacity of today's 3.5-inch diskette.

The 5.25-inch disks were dubbed "floppy" because the diskette packaging was a very flexible plastic envelope, unlike the rigid case used to hold today's 3.5-inch diskettes.

By the mid-1980s, the improved designs of the read/write heads, along with improvements in the magnetic recording media, led to the less-flexible, 3.5-inch, 1.44-megabyte (MB) capacity FDD in use today. For a few years, computers had both FDD sizes (3.5-inch and 5.25-inch). But by the mid-1990s, the 5.25-inch version had fallen out of popularity, partly because the diskette's recording surface could easily become contaminated by fingerprints through the open access area.

When you look at a floppy disk, you'll see a plastic case that measures 3 1/2 by 5 inches. Inside that case is a very thin piece of plastic that is coated with microscopic iron particles. This disk is much like the tape inside a video or audio cassette. Basically, a floppy disk drive reads and writes data to a small, circular piece of metal-coated plastic similar to audio cassette tape.

At one end of it is a small metal cover with a rectangular hole in it. That cover can be moved aside to show the flexible disk inside. But never touch the inner disk - you could damage the data that is stored on it. On one side of the floppy disk is a place for a label. On the other side is a silver circle with two holes in it. When the disk is inserted into the disk drive, the drive hooks into those holes to spin the circle. This causes the disk inside to spin at about 300 rpm! At the same time, the silver metal cover on the end is pushed aside so that the head in the disk drive can read and write to the disk.

Floppy disks are the smallest type of storage, holding only 1.44MB.

3.5-inch Diskettes (Floppy Disks) features:

Spin rate: app. 300 revolutions per minute (rpm)

High density (HD) disks more common today than older, double density (DD) disks

Storage Capacity of HD disks is 1.44 MB

Floppy Disk Drive Terminology

Floppy disk - Also called diskette. The common size is 3.5 inches.

Floppy disk drive - The electromechanical device that reads and writes floppy disks.

Track - Concentric ring of data on a side of a disk.

Sector - A subset of a track, similar to wedge or a slice of pie.

It consists of a read/write head and a motor rotating the disk at a high speed of about 300 rotations per minute. It can be fitted inside the cabinet of the computer and from outside, the slit where the disk is to be inserted, is visible. When the disk drive is closed after inserting the floppy inside, the monitor catches the disk through the Central of Disk hub, and then it starts rotating.

There are two read/write heads depending upon the floppy being one sided or two sided. The head consists of a read/write coil wound on a ring of magnetic material. During write operation, when the current passes in one direction, through the coil, the disk surface touching the head is magnetized in one direction. For reading the data, the procedure is reverse. I.e. the magnetized spots on the disk touching the read/write head induce the electronic pulses, which are sent to CPU.

The major parts of a FDD include:

Read/Write Heads: Located on both sides of a diskette, they move together on the same assembly. The heads are not directly opposite each other in an effort to prevent interaction between write operations on each of the two media surfaces. The same head is used for reading and writing, while a second, wider head is used for erasing a track just prior to it being written. This allows the data to be written on a wider "clean slate," without interfering with the analog data on an adjacent track.

Drive Motor: A very small spindle motor engages the metal hub at the center of the diskette, spinning it at either 300 or 360 rotations per minute (RPM).

Stepper Motor: This motor makes a precise number of stepped revolutions to move the read/write head assembly to the proper track position. The read/write head assembly is fastened to the stepper motor shaft.

Mechanical Frame: A system of levers that opens the little protective window on the diskette to allow the read/write heads to touch the dual-sided diskette media. An external button allows the diskette to be ejected, at which point the spring-loaded protective window on the diskette closes.

Circuit Board: Contains all of the electronics to handle the data read from or written to the diskette. It also controls the stepper-motor control circuits used to move the read/write heads to each track, as well as the movement of the read/write heads toward the diskette surface.

Electronic optics check for the presence of an opening in the lower corner of a 3.5-inch diskette (or a notch in the side of a 5.25-inch diskette) to see if the user wants to prevent data from being written on it.

Hard Disks

Your computer uses two types of memory: primary memory which is stored on chips located on the motherboard, and secondary memory that is stored in the hard drive. Primary memory holds all of the essential memory that tells your computer how to be a computer. Secondary memory holds the information that you store in the computer.

Inside the hard disk drive case you will find circular disks that are made from polished steel. On the disks, there are many tracks or cylinders. Within the hard drive, an electronic reading/writing device called the head passes back and forth over the cylinders, reading information from the disk or writing information to it. Hard drives spin at 3600 or more rpm (Revolutions Per Minute) - that means that in one minute, the hard drive spins around over 7200 times!

Optical Storage

Compact Disk Read-Only Memory (CD-ROM)

CD-Recordable (CD-R)/CD-Rewritable (CD-RW)

Digital Video Disk Read-Only Memory (DVD-ROM)

DVD Recordable (DVD-R/DVD Rewritable (DVD-RW)

Photo CD

Optical Storage Devices Data is stored on a reflective surface so it can be read by a beam of laser light. Two Kinds of Optical Storage Devices

CD-ROM (compact disk read-only memory)

DVD-ROM (digital video disk read-only memory)

Compact Disks

Instead of electromagnetism, CDs use pits (microscopic indentations) and lands (flat surfaces) to store information much the same way floppies and hard disks use magnetic and non-magnetic storage. Inside the CD-Rom is a laser that reflects light off of the surface of the disk to an electric eye. The pattern of reflected light (pit) and no reflected light (land) creates a code that represents data.

CDs usually store about 650MB. This is quite a bit more than the 1.44MB that a floppy disk stores. A DVD or Digital Video Disk holds even more information than a CD, because the DVD can store information on two levels, in smaller pits or sometimes on both sides.

Recordable Optical Technologies

CD-Recordable (CD-R)

CD-Rewritable (CD-RW)

PhotoCD

DVD-Recordable (DVD-R)

- DVD-RAM

CD ROM - Compact Disc Read Only Memory.

Unlike magnetic storage device which store data on multiple concentric tracks, all CD formats store data on one physical track, which spirals continuously from the center to the outer edge of the recording area. Data resides on the thin aluminum substrate immediately beneath the label. The data on the CD is recorded as a series of microscopic pits and lands physically embossed on an aluminum substrate. Optical drives use a low power laser to read data from those discs without physical contact between the head and the disc which contributes to the high reliability and permanence of storage device.

To write the data on a CD a higher power laser are used to record the data on a CD. It creates the pits and land on aluminum substrate. The data is stored permanently on the disc. These types of discs are called as WORM (Write Once Read Many). Data written to CD cannot subsequently be deleted or overwritten which can be classified as advantage or disadvantage depending upon the requirement of the user. However if the CD is partially filled then the more data can be added to it later on till it is full. CDs are usually cheap and cost effective in terms of storage capacity and transferring the data.

The CD‘s were further developed where the data could be deleted and re written. These types of CDs are called as CD Rewritable. These types of discs can be used by deleting the data and making the space for new data. These CD‘s can be written and rewritten at least 1000 times.

CD ROM Drive

CD ROM drives are so well standardized and have become so ubiquitous that many treat them as commodity items. Although CD ROM drives differ in reliability, which standards they support and numerous other respects, there are two important performance measures.

Data transfer rate

Average access

Data transfer rate: Data transfer rate means how fast the drive delivers sequential data to the interface. This rate is determined by drive rotation speed, and is rated by a number followed by ‗X‘. All the other things equal, a 32X drive delivers data twice the speed of a 16X drive. Fast data transfer rate is most important when the drive is used to transfer the large file or many sequential smaller files. For example: Gaming video.

CD ROM drive transfers the data at some integer multiple of this basic 150 KB/s 1X rate. Rather than designating drives by actual KB/s output drive manufacturers use a multiple of the standard 1X rate. For example: a 12X drive transfer data at (12*150KB/s) 1800 KB/s and so on.

The data on a CD is saved on tracks, which spirals from the center of the CD to outer edge. The portions of the tracks towards center are shorter than those towards the edge. Moving the data under the head at a constant rate requires spinning the disc faster as the head moves from the center where there is less data per revolution to the edge where there is more data. Hence the rotation rate of the disc changes as it progresses from inner to outer portions of the disc.

CD Writers

CD recordable and CD rewritable drives are collectively called as CD writers or CD burners. They are essentially CD ROM drives with one difference. They have a more powerful laser that, in addition to reading discs, can record data to special CD media.

Pen Drives / Flash Drives

- Pen Drives / Flash Drives are flash memory storage devices.
- They are faster, portable and have a capability of storing large data.
- It consists of a small printed circuit board with a LED encased in a robust plastic
- The male type connector is used to connect to the host PC
- They are also used a MP3 players

Functions of CPU:

The CPU, or Central Processing Unit, is both the heart and brains of every computer. Many of us do not know how important this unit is to the performance of a computer. How many of you have wondered about the basic functions of CPU? This article will answer that question, plus others including: Why it is important to have a good cooling system to keep the CPU at the right temperatures. Why it is so important to keep the CPU from overheating.

Common CPU Terms

Term	What it Means or Does
CPU	Central Processing Unit, or the heart and brains of the computer.
Binary Code	A sequence of ones and zeros, or the language into which your CPU translates all data.
ALU	Arithmetical Logical Unit, responsible for all mathematical and logical operations
Program Counter	Tracks which instructions the CPU should execute next when processing data.
One hertz	The speed at which one operation is performed per second.
Multi-core processor	A CPU with two or more independent cores, so it can do more than one thing at once.

The Four Primary Functions of the CPU

The CPU processes instructions it receives in the process of decoding data. In processing this data, the CPU performs four basic steps:

Fetch: Each instruction is stored in memory and has its own address. The processor takes this address number from the program counter, which is responsible for tracking which instructions the CPU should execute next.

Decode: All programs to be executed are translated to into Assembly instructions. Assembly code must be decoded into binary instructions, which are understandable to your CPU. This step is called decoding.

Execute: While executing instructions the CPU can do one of three things: Do calculations with its ALU, move data from one memory location to another, or jump to a different address.

Store: The CPU must give feedback after executing an instruction, and the output data is written to the memory.

INSTRUCTION FORMATS

The physical and logical structure of computers is normally described in reference manuals provided with the system. Such manuals explain the internal construction of the CPU, including the processor registers available and their logical capabilities. They list all hardware-implemented instructions, specify their binary code format, and provide a precise definition of each instruction. A computer will usually have a variety of instruction code formats. It is the function of the control unit within the CPU to interpret each instruction code and provide the necessary control functions needed to process the instruction.

The format of an instruction is usually depicted in a rectangular box symbolizing the bits of the instruction as they appear in memory words or in a control register. The bits of the instruction are divided into groups called fields. The most common fields found in instruction formats are:

1 An operation code field that specifies the operation to be performed.

An address field that designates a memory address or a processor registers.

A mode field that specifies the way the operand or the effective address is determined.

Other special fields are sometimes employed under certain circumstances, as for example a field that gives the number of shifts in a shift-type instruction.

The operation code field of an instruction is a group of bits that define various processor operations, such as add, subtract, complement, and shift. The bits that define the mode field of an instruction code specify a variety of alternatives for choosing the operands from the given address. The various addressing modes that have been formulated for digital computers are presented in Sec. 5.5. In this section we are concerned with the address field of an instruction format and consider the effect of including multiple address fields is an instruction.

Operations specified by computer instructions are executed on some data stored in memory or processor registers, Operands residing in processor registers are specified with a register address. A register address is a binary number of k bits that defines one of 2^k registers in the CPU. Thus a CPU with 16 processor registers R0 through R15 will have a
register address field of four bits. The binary number 0101, for example, will designate register R5.

Computers may have instructions of several different lengths containing varying number of addresses. The number of address fields in the instruction format of a computer depends on the internal organization of its registers. Most computers fall into one of three types of CPU organizations:

1. Single accumulator organization.
2. General register organization.
3. Stack organization.

An example of an accumulator-type organization is the basic computer presented in Chap. 5. All operations are performed with an implied accumulator register. The instruction format in this type of computer uses one address field. For example, the instruction that specifies an arithmetic addition is defined by an assembly language instruction as   ADD.

Where X is the address of the operand. The ADD instruction in this case results in the operation AC ← AC + M[X]. AC is the accumulator register and M[X] symbolizes the memory word located at address X.

An example of a general register type of organization was presented in Fig. 7.1. The instruction format in this type of computer needs three register address fields. Thus the instruction for an arithmetic addition may be written in an assembly language as

ADD R1, R2, R3

To denote the operation R1 ← R2 + R3. The number of address fields in the instruction can be reduced from three to two if the destination register is the same as one of the source registers. Thus the instruction

ADD R1, R2

Would denote the operation R1 ← R1 + R2. Only register addresses for R1 and R2 need be specified in this instruction.

Computers with multiple processor registers use the move instruction with a mnemonic MOV to symbolize a transfer instruction. Thus the instruction

MOV R1, R2

Denotes the transfer R1 ← R2 (or R2 ← R1, depending on the particular computer). Thus transfer-type instructions need two address fields to specify the source and the destination.

General register-type computers employ two or three address fields in their instruction format. Each address field may specify a processor register or a memory word. An instruction symbolized by

ADD R1, X

Would specify the operation R1 ← R + M [X]. It has two address fields, one for register R1 and the other for the memory address X.

The stack-organized CPU was presented in Fig. 8-4. Computers with stack organization would have PUSH and POP instructions which require an address field. Thus the instruction

PUSH X

Will push the word at address X to the top of the stack. The stack pointer is updated automatically. Operation-type instructions do not need an address field in stack-organized computers. This is because the operation is performed on the two items that are on top of the stack. The instruction

ADD

In a stack computer consists of an operation code only with no address field. This operation has the effect of popping the two top numbers from the stack, adding the numbers, and pushing the sum into the stack. There is no need to specify operands with an address field since all operands are implied to be in the stack.

Most computers fall into one of the three types of organizations that have just been described. Some computers combine features from more than one organization structure. For example, the Intel 808-microprocessor has seven CPU registers, one of which is an accumulator register As a consequence; the processor has some of the characteristics of a general register type and some of the characteristics of a accumulator type. All arithmetic and logic instruction, as well as the load and store instructions, use the accumulator register, so these instructions have only one address field. On the other hand, instructions that transfer data among the seven processor registers have a format that contains two register address fields. Moreover, the Intel 8080 processor has a stack pointer and instructions to push and pop from a memory stack. The processor, however, does not have the zero-address-type instructions which are characteristic of a stack-organized CPU.

To illustrate the influence of the number of addresses on computer programs, we will evaluate the arithmetic statement X = (A + B) ∗ (C + D).

Using zero, one, two, or three address instruction. We will use the symbols ADD, SUB, MUL, and DIV for the four arithmetic operations; MOV for the transfer-type operation; and LOAD and STORE for transfers to and from memory and AC register. We will assume that the operands are in memory addresses A, B, C, and D, and the result must be stored in memory at address X.

THREE-ADDRESS INSTRUCTIONS

Computers with three-address instruction formats can use each address field to specify either a processor register or a memory operand. The program in assembly language that evaluates X = (A + B) ∗ (C + D) is shown below, together with comments that explain the register transfer operation of each instruction.

ADD R1, A, B
R1 ← M [A] + M [B]
ADD R2, C, D
R2 ← M [C] + M [D]
MUL X, R1, R2
M [X] ← R1 ∗R2

It is assumed that the computer has two processor registers, R1 and R2. The symbol M [A] denotes the operand at
memory address symbolized by A.

The advantage of the three-address format is that it results in short programs when evaluating arithmetic expressions. The disadvantage is that the binary-coded instructions require too many bits to specify three addresses. An example of a commercial computer that uses three-address instructions is the Cyber 170. The instruction formats in the Cyber computer are restricted to either three register address fields or two register address fields and one memory address field.

TWO-ADDRESS INSTRUCTIONS

Two address instructions are the most common in commercial computers. Here again each address field can specify either a processor register or a memory word. The program to evaluate X = (A + B) ∗ (C + D) is as follows:

MOV R1, A	**R1 ← M [A]**
ADD R1, B	**R1 ← R1 + M [B]**
MOV R2, C	**R2 ← M [C]**
ADD R2, D	**R2 ← R2 + M [D]**
MUL R1, R2	**R1 ← R1*R2**

MOV X, R1	**M [X] ← R1**

The MOV instruction moves or transfers the operands to and from memory and processor registers. The first symbol listed in an instruction is assumed to be both a source and the destination where the result of the operation is transferred.

ONE-ADDRESS INSTRUCTIONS

One-address instructions use an implied accumulator (AC) register for all data manipulation. For multiplication and division there is a need for a second register. However, here we will neglect the second and assume that the AC contains the result of tall operations. The program to evaluate X = (A + B) ∗ (C + D) is

LOAD	**A**	**AC ← M [A]**
ADD	**B**	**AC ← A [C] + M [B]**
STORE	**T**	**M [T] ← AC**
LOAD	**C**	**AC ← M [C]**
ADD	**D**	**AC ← AC + M [D]**
MUL	**T**	**AC ← AC ∗ M [T]**
STORE	**X**	**M [X] ← AC**

All operations are done between the AC register and a memory operand. T is the address of a temporary memory location required for storing the intermediate result.

ZERO-ADDRESS INSTRUCTIONS

A stack-organized computer does not use an address field for the instructions ADD and MUL. The PUSH and POP instructions, however, need an address field to specify the operand that communicates with the stack. The following program shows how X = (A + B) ∗ (C + D) will be written for a stack organized computer. (TOS stands for top of stack)

PUSH	**A**	**TOS ← A**
PUSH	**B**	**TOS ← B**
ADD		**TOS ← (A + B)**
PUSH	**C**	**TOS ← C**
PUSH	**D**	**TOS ← D**
ADD		**TOS ← (C + D)**
MUL		**TOS ← (C + D) ∗ (A + B)**
POP	**X**	**M [X] ← TOS**

To evaluate arithmetic expressions in a stack computer, it is necessary to convert the expression into reverse Polish notation. The name “zero-address” is given to this type of computer because of the absence of an address field in the computational instructions.

ADDRESSING MODES

The operation field of an instruction specifies the operation to be performed. This operation must be executed on some data stored in computer registers or memory words. The way the operands are chosen during program execution in dependent on the addressing mode of the instruction. The addressing mode of the instruction. The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced. Computers use addressing mode techniques for the purpose of accommodating one or both of the following provisions:

To give programming versatility to the user by providing such facilities as pointers to Memory, counters for loop control, indexing of data, and program relocation

To reduce the number of bits in the addressing field of the instruction.

The availability of the addressing modes gives the experienced assembly language programmer flexibility for writing programs that are more efficient with respect to the number of instructions and execution time.

To understand the various addressing modes to be presented in this section, it is imperative that we understand the basic operation cycle of the computer. The control unit of a computer is designed to go through an instruction cycle that is divided into three major phases:

1. Fetch the instruction from memory
2. Decode the instruction.
3. Execute the instruction.

There is one register in the computer called the program counter of PC that keeps track of the instructions in the program stored in memory. PC holds the address of the instruction to be executed next and is incremented each time an instruction is fetched from memory. The decoding done in step 2 determines the operation to be performed, the addressing mode of the instruction and the location of the operands. The computer then executes the instruction and returns to step 1 to fetch the next instruction in sequence.

In some computers the addressing mode of the instruction is specified with a distinct binary code, just like the operation code is specified. Other computers use a single binary code that designates both the operation and the mode of the instruction. Instructions may be defined with a variety of addressing modes, and sometimes, two or more addressing modes are combined in one instruction.

An example of an instruction format with a distinct addressing mode field is shown in Fig. 1. The operation code specified the operation to be performed. The mode field is sued to locate the operands needed for the operation. There may or may not be an address field in the instruction. If there is an address field, it may designate a memory address or a processor register. Moreover, as discussed in the preceding section, the instruction may have more than one address field, and each address field may be associated with its own particular addressing mode.

Although most addressing modes modify the address field of the instruction, there are two modes that need no address field at all. These are the implied and immediate modes.

Implied Mode: In this mode the operands are specified implicitly in the definition of the instruction. For example, the instruction “complement accumulator” is an implied-mode instruction because the operand in the accumulator register is implied in the definition of the instruction. In fact, all register reference instructions that sue an accumulator are implied-mode instructions.

Op code	Mode	Address

Figure 1: Instruction format with mode field

Zero-address instructions in a stack-organized computer are implied-mode instructions since the operands are implied to be on top of the stack.

Immediate Mode: In this mode the operand is specified in the instruction itself. Inother words, an immediate- mode instruction has an operand field rather than an address field. The operand field contains the actual operand to be used in conjunction with the operation specified in the instruction. Immediate-mode instructions are useful for initializing registers to a constant value.

It was mentioned previously that the address field of an instruction may specify either a memory word or a processor register. When the address field specifies a processor register, the instruction is said to be in the register mode.

Register Mode: In this mode the operands are in registers that reside within the CPU .The particular register is selected from a register field in the instruction. A k-bit field can specify any one of 2^k registers.

Register Indirect Mode: In this mode the instruction specifies a register in the CPU whose contents give the address of the operand in memory. In other words, the selected register contains the address of the operand rather than the operand itself. Before using a register indirect mode instruction, the programmer must ensure that the memory address fo the operand is placed in the processor register with a previous instruction. A reference to the register is then equivalent to specifying a memory address. The advantage of a register indirect mode instruction is that the address field of the instruction sues fewer bits to select a register than would have been required to specify a memory address directly.

Auto increment or Auto decrement Mode: This is similar to the register indirect mode except that the register is incremented or decremented after (or before) its value is used to access memory. When the address stored in the register refers to a table of data in memory, it is necessary to increment or decrement the register after every access to the table. This can be achieved by using the increment or decrement instruction. However, because it is such a common requirement, some computers incorporate a special mode that automatically increments or decrements the content of the register after data access.

The address field of an instruction is used by the control unit in the CPU to obtain the operand from memory. Sometimes the value given in the address field is the address of the operand, but sometimes it is just an address from which the address of the operand is calculated. To differentiate among the various addressing modes it is necessary to distinguish between the address part of the instruction and the effective address used by the control when executing the instruction. The effective address is defined to be the memory address obtained from the computation dictated by the given addressing mode. The effective address is the address of the operand in a computational-type instruction. It is the address where control branches in response to a branch-type instruction. We have already defined two addressing modes in previous chapter.

Direct Address Mode: In this mode the effective address is equal to the address part of the instruction. The operand resides in memory and its address is given directly by the address field of the instruction. In a branch-type instruction the address field specifies the actual branch address.

Indirect Address Mode: In this mode the address field of the instruction gives the address where the effective address is stored in memory. Control fetches the instruction from memory and uses its address part to access memory again to read the effective address.

Relative Address Mode: In this mode the content of the program counter is added to the address part of the instruction in order to obtain the effective address. The address part of the instruction is usually a signed number (in 2's complement representation) which can be either positive or negative. When this number is added to the content of the program counter, the result produces an effective address whose position in memory is relative to the address of the next instruction. To clarify with an example, assume that the program counter contains the number 825 and the address part of the instruction contains the number 24. The instruction at location 825 is read from memory during the fetch phase and the program counter is then incremented by one to 826 + 24 = 850. This is 24 memory locations forward from the address of the next instruction. Relative addressing is often used with branch-type instructions when the branch address is in the area surrounding the instruction word itself. It results in a shorter address field in the instruction format since the relative address can be specified with a smaller number of bits compared to the number of bits required to designate the entire memory address.

Indexed Addressing Mode: In this mode the content of an index register is added to the address part of the instruction to obtain the effective address. The index register is a special CPU register that contains an index value. The address field of the instruction defines the beginning address of a data array in memory. Each operand in the array is stored in memory relative to the beginning address. The distance between the beginning address and the address of the operand is the index value stores in the index register. Any operand in the array can be accessed with the same instruction provided that the index register contains the correct index value. The index register can be incremented to facilitate access to consecutive operands. Note that if an index-type instruction does not include an address field in its format, the instruction converts to the register indirect mode of operation. Some computers dedicate one CPU register to function solely as an index register. This register is involved implicitly when the index-mode instruction is used. In computers with many processor registers, any one of the CPU registers can contain the index number. In such a case the register must be specified explicitly in a register field within the instruction format.

Base Register Addressing Mode: In this mode the content of a base register is added tothe address part of the instruction to obtain the effective address. This is similar to the indexed addressing mode except that the register is now called a base register instead of an index register. The difference between the two modes is in the way they are used rather than in the way that they are computed. An index register is assumed to hold an index number that is relative to the address part of the instruction. A base register is assumed to hold a base address and the address field of the instruction gives a displacement relative to this base address. The base register addressing mode is used in computers to facilitate the relocation of programs in memory. When programs and data are moved from one segment of memory to another, as required in multiprogramming systems, the address values of the base register requires updating to reflect the beginning of a new memory segment.

Numerical Example

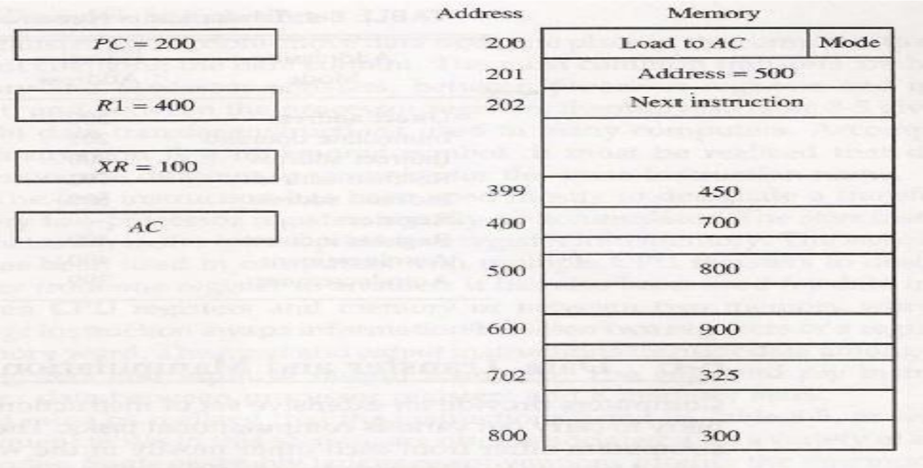


Figure 8-7 Numerical example for addressing modes.

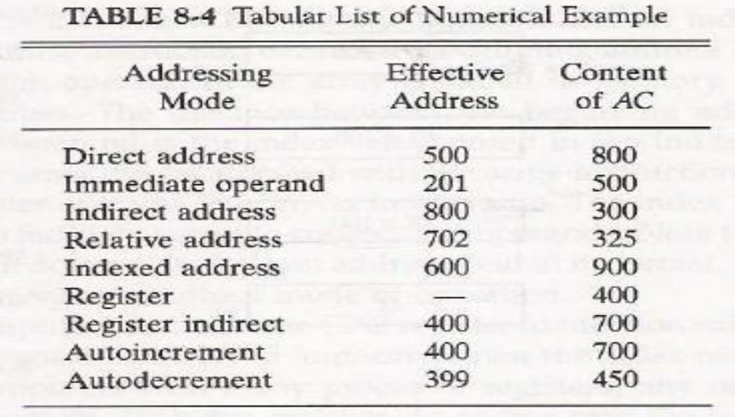
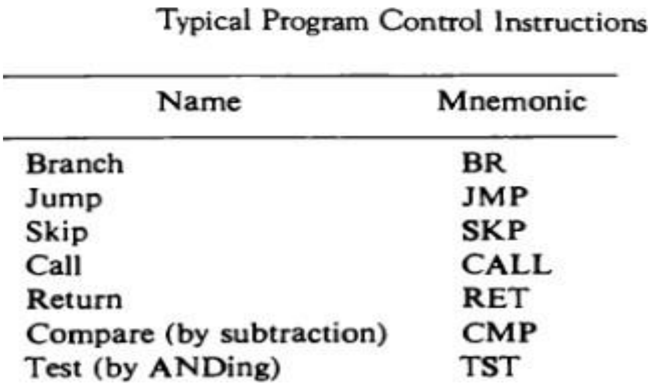


TABLE 8-4 Tabular List of Numerical Example

Addressing Mode	Effective Address	Content of AC
Direct address	500	800
Immediate operand	201	500
Indirect address	800	300
Relative address	702	325
Indexed address	600	900
Register	—	400
Register indirect	400	700
Autoincrement	400	700
Autodecrement	399	450

Program Control

Instructions are always stored in successive memory locations. When processed in the CPU, the instructions are fetched from consecutive memory locations and executed



Typical Program Control Instructions

Name	Mnemonic
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RET
Compare (by subtraction)	CMP
Test (by ANDing)	TST

Status Bit Conditions

It is sometimes convenient to supplement the ALU circuit in the CPU with a status register where status bit conditions can be stored for further analysis. Status bits are also called condition-code bits or flag bits.

The four status bits are symbolized by C, S, Z, and V. The bits are set or cleared as a result of an operation performed in the ALU.

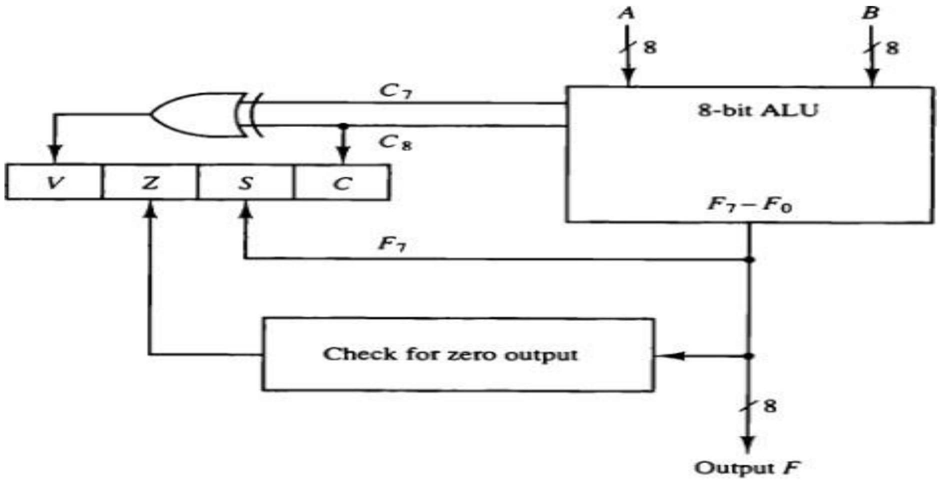
Bit C (carry) is set to 1 if the end cany C8 is 1. It is cleared to 0 if the canyisO.

Bit S (sign) is set to 1 if the highest-order bit F? is 1. It is set to 0 if the bit is 0.

Bit Z (zero) is set to 1 if the output of the ALU contains all 0's. It is cleared to 0 otherwise. In other words, Z = 1 if the output is zero and Z = 0 if the output is not zero.

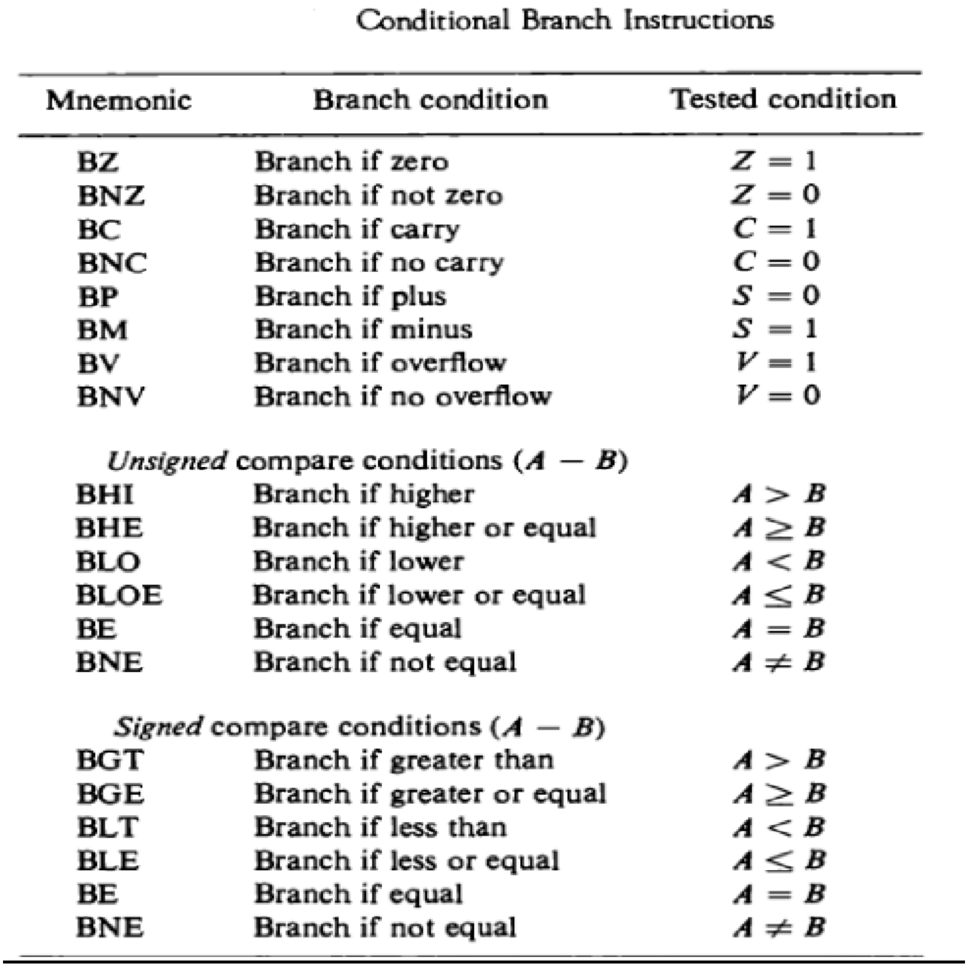
Bit V (overflow) is set to 1 if the exclusive-OR of the last two carries is equal to 1, and Cleared to 0 otherwise. This is the condition for an overflow when negative numbers are In 2's complement.

For the 8-bit ALU, V = 1 if the output is greater than +127 or less than -128.



Status register bits.

Conditional Branch Instructions



Conditional Branch Instructions

Mnemonic	Branch condition	Tested condition
BZ	Branch if zero	$Z = 1$
BNZ	Branch if not zero	$Z = 0$
BC	Branch if carry	$C = 1$
BNC	Branch if no carry	$C = 0$
BP	Branch if plus	$S = 0$
BM	Branch if minus	$S = 1$
BV	Branch if overflow	$V = 1$
BNV	Branch if no overflow	$V = 0$
Unsigned compare conditions ($A - B$)		
BHI	Branch if higher	$A > B$
BHE	Branch if higher or equal	$A \geq B$
BLO	Branch if lower	$A < B$
BLOE	Branch if lower or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$
Signed compare conditions ($A - B$)		
BGT	Branch if greater than	$A > B$
BGE	Branch if greater or equal	$A \geq B$
BLT	Branch if less than	$A < B$
BLE	Branch if less or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$

Some computers consider the C bit to be a borrow bit after a subtraction operation A — B. A Borrow does not occur if

A ^ B, but a bit must be borrowed from the next most significant Position if A < B. The condition for a borrow is the complement of the carry obtained when the subtraction is done by taking the 2's complement of B. For this reason, a processor that considers the C bit to be a borrow after a subtraction will complement the C bit after adding the 2's complement of the subtrahend and denote this bit a borrow.

Subroutine Call and Return

A subroutine is a self-contained sequence of instructions that performs a given computational task.

The instruction that transfers program control to a subroutine is known by different names. The most common names used are call subroutine, jump to subroutine, branch to subroutine, or branch and save address.

The instruction is executed by performing two operations:

The address of the next instruction available in the program counter (the return address) is Stored in a temporary location so the subroutine knows where to return

Control is transferred to the beginning of the subroutine.

Different computers use a different temporary location for storing the return address.

- Some store the return address in the first memory location of the subroutine, some store it in a fixed location in memory, some store it in a processor register, and some store it in a memory stack. The most efficient way is to store the return address in a memory stack. The advantage of using a stack for the return address is that when a succession of subroutines is called, the sequential return addresses can be pushed into the stack. The return from subroutine instruction causes the stack to pop and the contents of the top of the stack are transferred to the program counter.

A subroutine call is implemented with the following micro operations:

SP ←SP - 1 Decrement stack pointer

M [SP] ←PC Push content of PC onto the stack

PC ← effective address Transfer control to the subroutine

If another subroutine is called by the current subroutine, the new return address is pushed into The stack and so on. The instruction that returns from the last subroutine is implemented by the Micro operations:

PC←M [SP] Pop stack and transfer to PC SP ←SP + 1 Increment stack pointer

Program Interrupt

Program interrupt refers to the transfer of program control from a currently running program to another service program as a result of an external or internal generated request. Control returns to the original program after the service program is executed.The interrupt procedure is, in principle, quite similar to a subroutine call except for three Variations:

The interrupt is usually initiated by an internal or external signal rather than from the Execution of an instruction (except for software interrupt as explained later);

The address of the interrupt service program is determined by the hardware rather than from the address field of an instruction.

An interrupt procedure usually stores all the information

The state of the CPU at the end of the execute cycle (when the interrupt is recognized) is determined From:

The content of the program counter

The content of all processor registers

The content of certain status conditions

- Program status word the collection of all status bit conditions in the CPU is sometimes called a program status word or PSW. The PSW is stored in a separate hardware register and contains the status information that characterizes the state of the CPU.

Types of Interrupts

There are three major types of interrupts that cause a break in the normal execution of a Program. They can be classified as:

1. External interrupts
2. Internal interrupts
3. Software interrupts

External interrupts come from input-output (I/O) devices, from a timing device, from a circuit monitoring the power supply, or from any other external source.

Internal interrupts arise from illegal or erroneous use of an instruction or data. Internal interrupts are also called traps. Examples of interrupts caused by internal error conditions are register overflow, attempt to divide by zero, an invalid operation code, stack overflow, and protection violation.

A software interrupt is initiated by executing an instruction. Software interrupt is a special call instruction that behaves like an interrupt rather than a subroutine call. It can be used by the programmer to initiate an interrupt procedure at any desired point in the program.

****

UNIT-II

Input-Output Organizations: I/O Interface, I/O Bus and Interface modules: I/O Vs Memory Bus, Isolated Vs Memory-Mapped I/O, Asynchronous data Transfer- Strobe Control, Hand Shaking: Asynchronous Serial transfer- Asynchronous Communication interface, Modes of transfer Programmed I/O, Interrupt Initiated I/O, DMA; DMA Controller, DMA Transfer, IOP-CPU-IOP Communication, Intel 8089 IOP.

INPUT-OUTPUT INTERFACE

Input-output interface provides a method for transferring information between internal storage and external I/O devices. Peripherals connected to a computer need special communication links for interfacing them with the central processing unit. The purpose of the communication link is to resolve the differences that exist between the central computer and each peripheral. The major differences are:

Peripherals are electromechanical and electromagnetic devices and their manner of operation is different from the operation of the CPU and memory, which are electronic devices. Therefore, a conversion of signal values may be required.

The data transfer rate of peripherals is usually slower than the transfer rate of the CPU, and consequently, a synchronization mechanism may be need.

Data codes and formats in peripherals differ from the word format in the CPU and memory.

The operating modes of peripherals are different from each other and each must be controlled so as not to disturb the operation of other peripherals connected to the CPU.

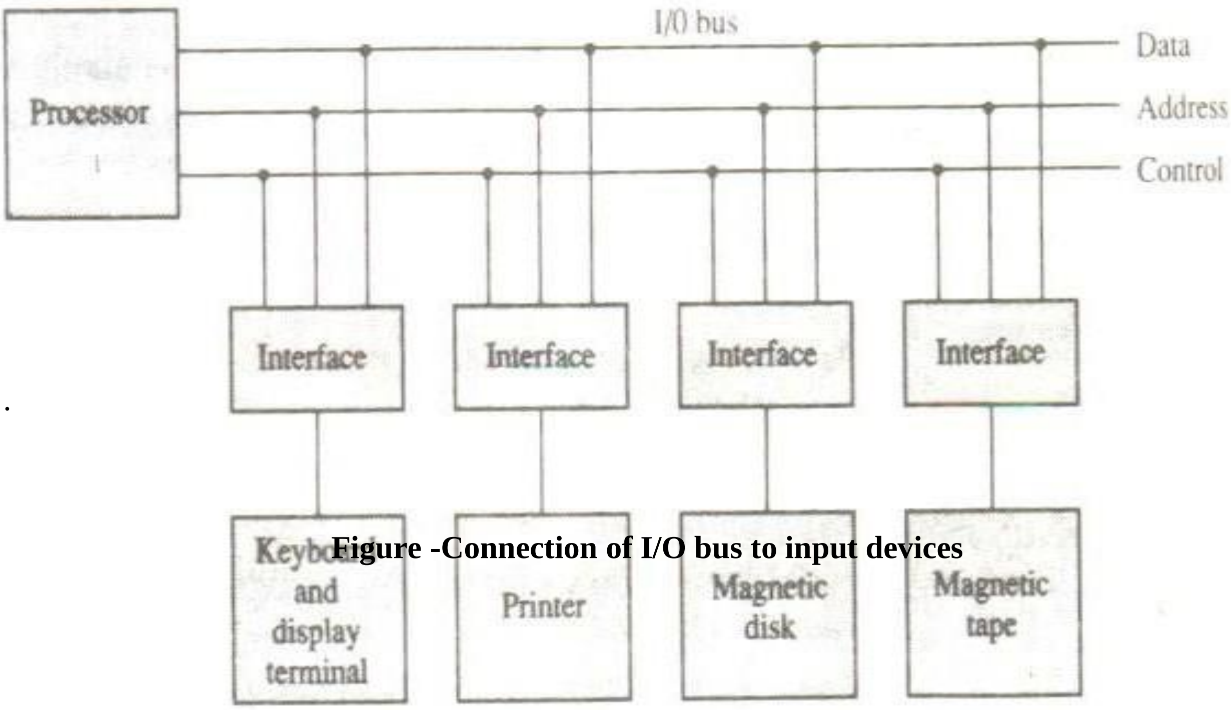
To resolve these differences, computer systems include special hardware components between the CPU and peripherals to supervise and synchronize all input and output transfers. These components are called interface units because they interface between the processor bus and the peripheral device. In addition, each device may have its own controller that supervises the operations of the particular mechanism in the peripheral.

I/O BUS AND INTERFACE MODULES

A typical communication link between the processor and several peripherals is shown in below Fig.The I/O bus consists of data lines, address lines, and control lines. The magnetic disk, printer, and terminal are employed in practically any general-purpose computer. The magnetic tape is used in some computers for backup storage. Each peripheral device has associated with it an interface unit. Each interface decodes the address and control received from the I/O bus, interprets them for the peripheral, and provides signals for the peripheral controller. It also synchronizes the data flow and supervises the transfer between peripheral and processor. Each peripheral has its own controller that operates the particular electromechanical device. For example, the printer controller controls the paper motion, the print timing, and the selection of printing characters. A controller may be housed separately or may be physically integrated with the peripheral.

The I/O bus from the processor is attached to all peripheral interfaces. To communicate with a particular device, the processor places a device address on the address lines. Each interface attached

to the I/O bus contains an address decoder that monitors the address lines. When the interface detects its own address, it activates the path between the bus lines and the device that it controls. All peripherals whose address does not correspond to the address in the bus are disabled their interface.
At the same time that the address is made available in the address lines, the processor provides a function code in the control lines. The interface selected responds to the function code and proceeds to execute it.



.

Figure -Connection of I/O bus to input devices

The function code is referred to as an I/O command and is in essence an instruction that is executed in the interface and its attached peripheral unit. The interpretation of the command depends on the peripheral that the processor is addressing. There are four types of commands that an interface may receive. They are classified as control, status, status, data output, and data input.

A control command is issued to activate the peripheral and to inform it what to do. For example, a magnetic tape unit may be instructed to backspace the tape by one record, to rewind the tape, or to start the tape moving in the forward direction. The particular control command issued depends on the peripheral, and each peripheral receives its own distinguished sequence of control commands, depending on its mode of operation.

A status command is used to test various status conditions in the interface and the peripheral. For example, the computer may wish to check the status of the peripheral before a transfer is initiated. During the transfer, one or more errors may occur which are detected by the interface. These errors are designated by setting bits in a status register that the processor can read at certain intervals.

A data output command causes the interface to respond by transferring data from the bus into one of its registers. Consider an example with a tape unit. The computer starts the tape moving by issuing a control command. The processor then monitors the status of the tape by means of a status command. When the tape is in the correct position, the processor issues a data output command. The interface responds to the address and command and transfers the information from the data lines in the bus to its buffer register. The interface then communicates with the tape controller and sends the data to be stored on tape.

The data input command is the opposite of the data output. In this case the interface receives an item of data from the peripheral and places it in its buffer register. The processor checks if data are available by means of a status command and then issues a data input command. The interface places the data on the data lines, where they are accepted by the processor.

I/O VERSUS MEMORY BUS

In addition to communicating with I/O, the processor must communicate with the memory unit. Like the I/O bus, the memory bus contains data, address, and read/write control lines. There are three ways that computer buses can be used to communicate with memory and I/O:

Use two separate buses, one for memory and the other for I/O.

Use one common bus for both memory and I/O but have separate control lines for each.

Use one common bus for memory and I/O with common control lines.

In the first method, the computer has independent sets of data, address, and control buses, one for accessing memory and the other for I/O. This is done in computers that provide a separate I/O processor (IOP) in addition to the central processing unit (CPU). The memory communicates with both the CPU and the IOP through a memory bus. The IOP communicates also with the input and output devices through a separate I/O bus with its own address, data and control lines. The purpose of the IOP is to provide an independent pathway for the transfer of information between external devices and internal memory.

ISOLATED VERSUS MEMORY-MAPPED I/O

Many computers use one common bus to transfer information between memory or I/O and the CPU. The distinction between a memory transfer and I/O transfer is made through separate read and write lines. The CPU specifies whether the address on the address lines is for a memory word or for an interface register by enabling one of two possible read or write lines. The I/O read and I/O writes control lines are enabled during an I/O transfer. The memory read and memory write control lines are enabled during a memory transfer. This configuration isolates all I/O interface addresses from the addresses assigned to memory and is referred to as the isolated I/O method for assigning addresses in a common bus.

In the isolated I/O configuration, the CPU has distinct input and output instructions, and each of these instructions is associated with the address of an interface register. When the CPU fetches and decodes the operation code of an input or output instruction, it places the address associated with the instruction into the common address lines. At the same time, it enables the I/O read (for input) or I/O write (for output) control line. This informs the external components that are attached to the common bus that the address in the address lines is for an interface register and not for a memory word. On the other hand, when the CPU is fetching an instruction or an operand from memory, it places the memory address on the address lines and enables the memory read or memory write control line. This informs the external components that the address is for a memory word and not for an I/O interface.

The isolated I/O method isolates memory word and not for an I/O addresses so that memory address values are not affected by interface address assignment since each has its own address space. The other alternative is to use the same address space for both memory and I/O. This is the case in computers that employ only one set of read and write signals and do not distinguish between memory and I/O addresses. This configuration is referred to as memory-mapped I/O. The computer treats an interface

Register as being part of the memory system. The assigned addresses for interface registers cannot be used for memory words, which reduce the memory address range available.

In a memory-mapped I/O organization there are no specific inputs or output instructions. The CPU can manipulate I/O data residing in interface registers with the same instructions that are used to manipulate memory words. Each interface is organized as a set of registers that respond to read and write requests in the normal address space. Typically, a segment of the total address space is reserved for interface registers, but in general, they can be located at any address as long as There is not also a memory word that responds to the same address.

Computers with memory-mapped I/O can use memory-type instructions to access I/O data. It allows the computer to use the same instructions for either input-output transfers or for memory transfers. The advantage is that the load and store instructions used for reading and writing from memory can be used to input and output data from I/O registers. In a typical computer, there are more memory-reference instructions than I/O instructions. With memory-mapped I/O all instructions that refer to memory are also available for I/O.

ASYNCHRONOUS DATA TRANSFER

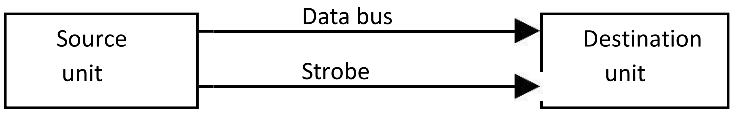
The internal operations in a digital system are synchronized by means of clock pulses supplied by a common pulse generator. Clock pulses are applied to all registers within a unit and all data transfers among internal registers occur simultaneously during the occurrence of a clock pulse. Two units, such as a CPU and an I/O interface, are designed independently of each other. If the registers in the interface share a common clock with the CPU registers, the transfer between the two units is said to be synchronous. In most cases, the internal timing in each unit is independent from the other in that each uses its own private clock for internal registers. In that case, the two units are said to be asynchronous to each other. This approach is widely used in most computer systems.

Asynchronous data transfer between two independent units requires that control signals be transmitted between the communicating units to indicate the time at which data is being transmitted. One way of achieving this is by means of a strobe pulse supplied by one of the units to indicate to the other unit when the transfer has to occur. Another method commonly used is to accompany each data item being transferred with a control signal that indicates the presence ofdata in the bus. The unit receiving the data item responds with another control signal to acknowledge receipt of the data. This type of agreement between two independent units is referred to as handshaking.

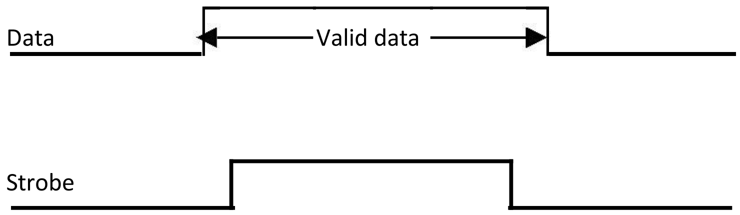
The strobe pulse method and the handshaking method of asynchronous data transfer are not restricted to I/O transfers. In fact, they are used extensively on numerous occasions requiring the transfer of data between two independent units. In the general case we consider the transmitting unit as the source and the receiving unit as the destination. For example, the CPU is the source unit during an output or a write transfer and it is the destination unit during an input or a read transfer. It is customary to specify the asynchronous transfer between two independent units by means of a timing diagram that shows the timing relationship that must exist between the control signals and the data in buses. The sequence of control during an asynchronous transfer depends on whether the transfer is initiated by the source or by the destination unit.

STROBE CONTROL

The strobe control method of asynchronous data transfer employs a single control line to time each transfer. The strobe may be activated by either the source or the destination unit. Figure-(a) shows a source-initiated transfer.



Block diagram



Timing diagram Figure- Source-initiated strobe for data transfer.

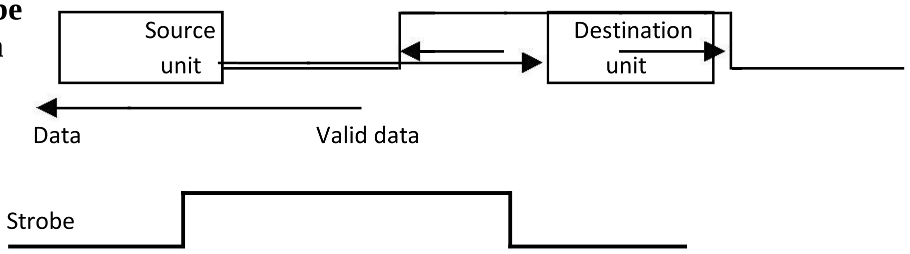
The data bus carries the binary information from source unit to the destination unit. Typically, the bus has multiple lines to transfer an entire byte or word. The strobe is a single line that informs the destination unit when a valid data word is available in the bus.

As shown in the timing diagram of Fig.-(b), the source unit first places the data on the data bus. After a brief delay to ensure that the data settle to a steady value, the source activates the strobe pulse. The information on the data bus and the strobe signal remain in the active state for a sufficient time period to allow the destination unit to receive the data. Often, the destination unit uses the falling edge of the strobe pulse to transfer the contents of the data bus into one of its internal registers. The source removes the data from the bus a brief period after it disables its strobe pulse. Actually, the source does not have to change the information in the data bus. The fact that the strobe signal is disabled indicates that the data bus does not contain valued data. New valid data will be available only after the strobe is enabled again.

Below Figure shows a data transfer initiated by the destination unit. In this case the destination unit activates the strobe pulse, informing the source to provide the data. The source unit responds by placing the requested binary information on the data bus. The data must be valid and remain in the bus long enough for the destination unit to accept it. The falling edge of the Strobe pulse can beused again to trigger a destination register. The destination unit then disables the strobe. The source removes the data from the bus after a predetermined time interval.

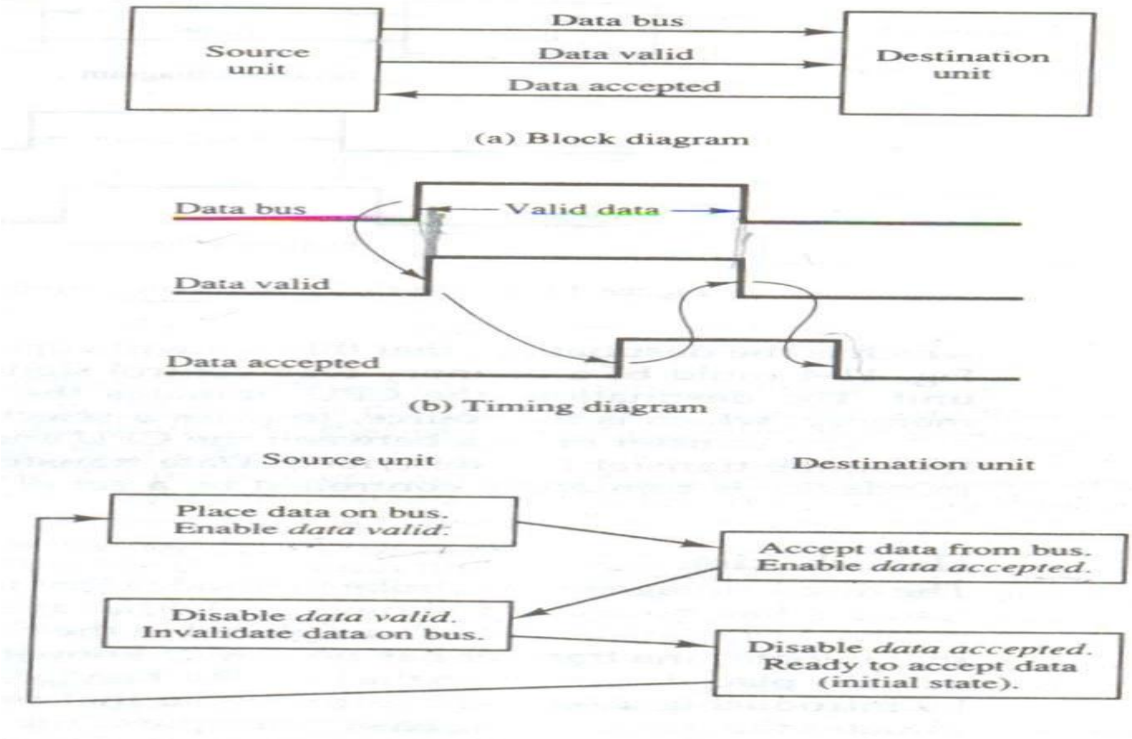
In many computers the strobe pulse is actually controlled by the clock pulses in the CPU. The CPU is always in control of the buses and informs the external units how to transfer data. For example, the strobe of above Fig. could be a memory -write control signal from the CPU to a memory unit. The source, being the CPU, places a word on the data bus and informs the memory units.

Data bus Strobe Block diagram



Timing diagram Figure -Destination-initiated strobe for data transfer.

This is the destination, that this is a write operation. Similarly, the strobe of Figure -Destination-initiated strobe for data transfer. Could be a memory -read control signal from the CPU to a memory unit. The destination, the CPU, initiates the read operation to inform the memory, which


Source unit
Data bus
Data valid
Data accepted
Destination unit
Data bus
Valid data
Data valid
Data accepted
Source unit
Destination unit
Place data on bus. Enable *data valid*.
Accept data from bus. Enable *data accepted*.
Disable *data valid*. Invalidate data on bus.
Disable *data accepted*. Ready to accept data (initial state).

(a) Block diagram

(b) Timing diagram

(c) Sequence of events

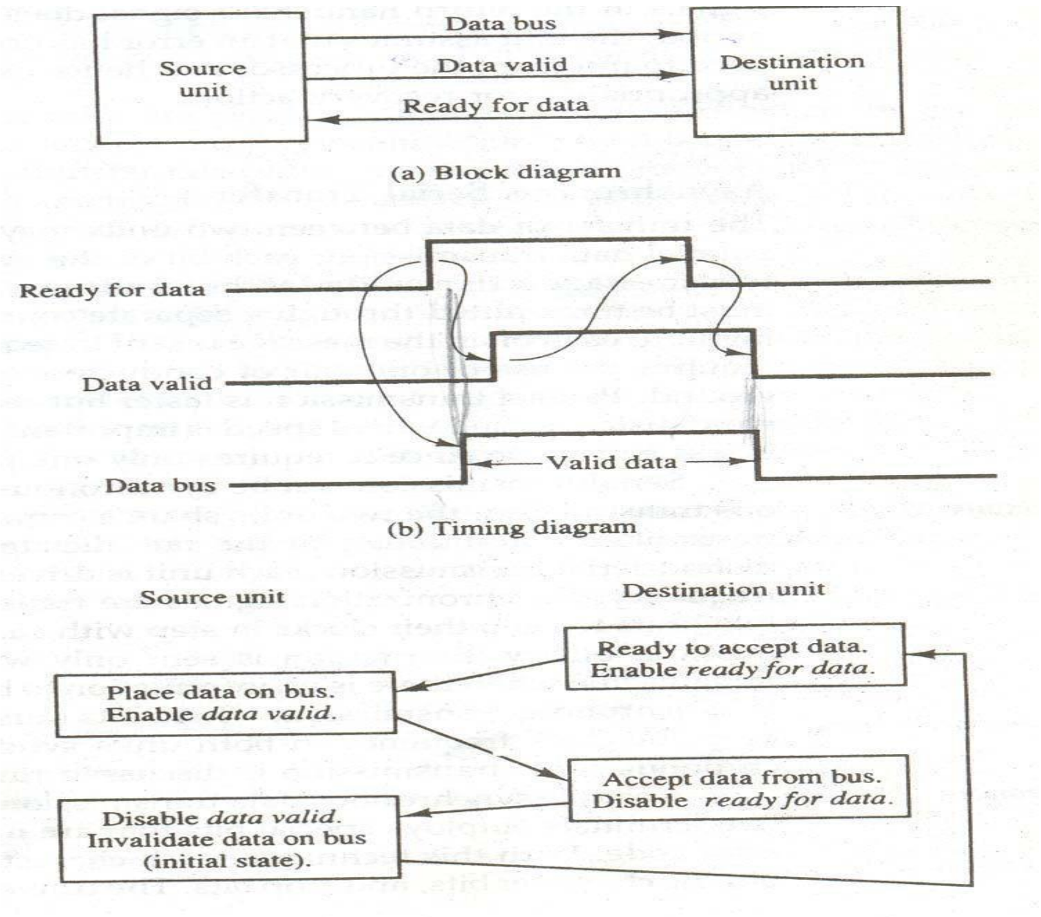
is the source, to place a selected word into the data bus. The transfer of data between the CPU and an interface unit is similar to the strobe transfer just described. Data transfer between an interface and an I/O device is commonly controlled by a set of handshaking lines.

HANDSHAKING

The disadvantage of the strobe method is that the source unit that initiates the transfer has no way of knowing whether the destination unit has actually received the data item that was placed in the bus. Similarly, a destination unit that initiates the transfer has no way of knowing whether the source unit has actually placed the data on the bus,. The handshake method solves this problem by introducing a second control signal that provides a reply to the unit that initiates the transfer. The basic principle of the two-write handshaking method of data transfer is as follows. One control line is in the same direction as the data flow in the bus from the source to the destination. It is used by the source unit to inform the destination unit whether there are valued data in the bus. The other control line is in the other direction from the destination to the source. It is used by the destination unit to inform the source whether it can accept data. The sequence of control during the transfer depends on the unit that initiates the transfer.

Figure shows the data transfer procedure when initiated by the source. The two handshaking lines are data valid, which is generated by the source unit, and data accepted, generated by the destination unit. The timing diagram shows the exchange of signals between the two units. The sequence of events listed in part (c) shows the four possible states that the system can be at any given time. The source unit initiates the transfer by placing the data on the bus and enabling its data valid signal. The data accepted signal is activated by the destination unit after it accepts the data from the bus. The source unit then disables its data valid signal, which invalidates the data on the bus. The destination unit then disables its data accepted signal and the system goes into its initial state. The source dies not send the next data item until after the destination unit shows its readiness to accept new data by disabling its data accepted signal. This scheme allows arbitrary delays from one state to the next and permits each unit to respond at its own data transfer rate. The rate of transfer is determined by the slowest unit.

The destination -initiated transfer using handshaking lines is shown in Fig. Note that the name of the signal generated by the destination unit has been changed to ready from data to reflect its new meaning. The source unit in this case does not place data on the bus until after it receives the ready for data signal from the destination unit. From there on, the handshaking procedure follows the same pattern as in the source-initiated case. Note that the sequence of events in both cases would be identical if we consider the ready for data signal as the complement of data accepted. In fact, the only difference between the source-initiated and the destination-initiated transfer is in their choice of initial state.


Source unit
Data bus
Data valid
Ready for data
Destination unit
Ready for data
Data valid
Data bus
Valid data
Source unit
Destination unit
Ready to accept data. Enable *ready for data*.
Place data on bus. Enable *data valid*.
Accept data from bus. Disable *ready for data*.
Disable *data valid*. Invalidate data on bus (initial state).

(a) Block diagram

(b) Timing diagram

(c) Sequence of events

The handshaking scheme provides a high degree of flexibility and reality because the successful completion of a data transfer relies on active participation by both units. If one unit is faulty, the data transfer will not be completed. Such an error can be detected by means of a timeout mechanism, which produces an alarm if the data transfer is not completed within a predetermined time. The timeout is implemented by means of an internal clock that starts counting time when the unit enables one of its handshaking control signals. If the return handshake signal does not respond within a given time period, the unit assumes that an error has occurred. The timeout signal can be used to interrupt the processor and hence execute a service routine that takes appropriates error recovery action.

ASYNCHRONOUS SERIAL TRANSFER

The transfer of data between two units may be done in parallel or serial. In parallel data transmission, each bit of the message has its own path and the total message is transmitted at the same time. This means that an n-bit message must be transmitted through in *n* separate conductor paths. In serial data transmission, each bit in the message is sent in sequence one at a time. This method requires the use of one pair of conductors or one conductor and a common ground. Parallel transmission is faster but requires many wires. It is used for short distances and where speed is important. Serial transmission is slower but is less expensive since it requires only one pair of conductors.

Serial transmission can be synchronous or asynchronous. In synchronous transmission, the two units share a common clock frequency and bits are transmitted continuously at the rate dictated by the clock pulses. In long-distant serial transmission, each unit is driven by a separate clock of the same frequency. Synchronization signals are transmitted periodically between the two units to keep their clocks in step with each other. In asynchronous transmission, binary information is sent only when it is available and the line remains idle when there is no information to be transmitted. This is In contrast to synchronous transmission, where bits must be transmitted continuously to deep the clock frequency in both units synchronized with each other.

Serial asynchronous data transmission technique used in many interactive terminals employs special bits that are inserted at both ends of the character code. With this technique, each character consists of three parts: a start bit, the character bits, and stop bits. The convention is that the transmitter rests at the 1-state when no characters are transmitted. The first bit, called the start bit, is always a 0 and is used to indicate the beginning of a character. The last bit called the stop bit is always a 1. An example of this format is shown in below Fig.

A transmitted character can be detected by the receiver from knowledge of the transmission rules:

When a character is not being sent, the line is kept in the 1-state.

The initiation of a character transmission is detected from the start bit, which is always 0.

The character bits always follow the start bit.

After the last bit of the character is transmitted, a stop bit is detected when the line returns To the 1-state for at least one bit time.

Using these rules, the receiver can detect the start bit when the line gives from 1 to 0. A clock in the receiver examines the line at proper bit times. The receiver knows the transfer rate of the bits and the number of character bits to accept. After the character bits are transmitted, one or two stop bits are sent. The stop bits are always in the 1-state and frame the end of the character to signify the idle or wait state.

At the end of the character the line is held at the 1-state for a period of at least one or two bit times so that both the transmitter and receiver can resynchronize. The length of time that the line stays in this state depends on the amount of time required for the equipment to resynchronize. Some older electromechanical terminals use two stop bits, but newer terminals use one stop bit. The line remains in the 1-state until another character is transmitted. The stop time ensures that a new character will not follow for one or two bit times.

As illustration, consider the serial transmission of a terminal whose transfer rate is 10 characters per second. Each transmitted character consists of a start bit, eight information bits, of

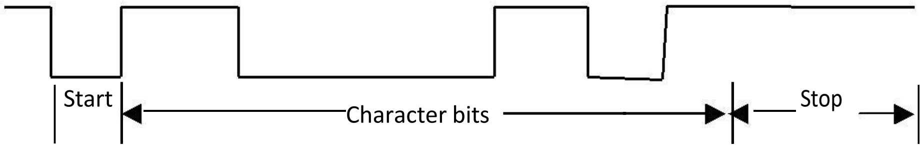


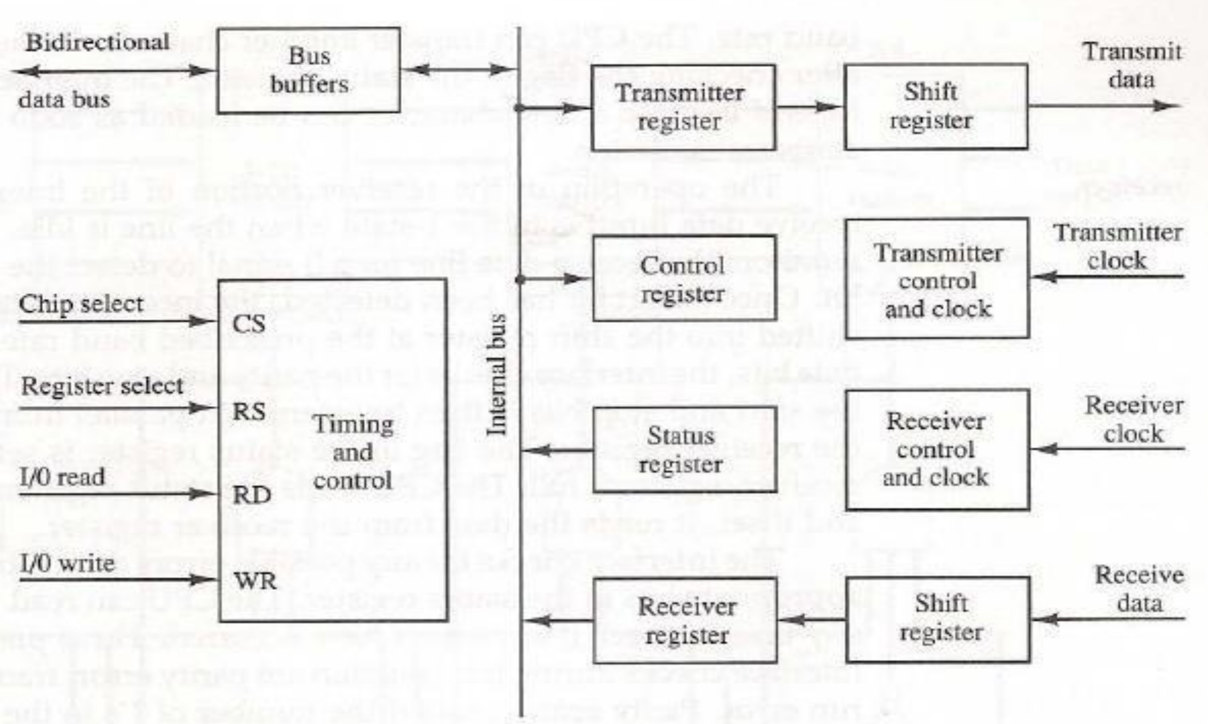
Figure - Asynchronous serial transmission

a start bit, eight information bits, and two stop bits, for a total of 11 bits. Ten characters per second means that each character takes 0.1s for transfer. Since there are 11 bits to be transmitted, it follows that the bit time is 9.09 ms.The baud rate is defined as the rate at which serial information is transmitted and is equivalent to the data transfer in bits per second. Ten characters per second with an 11-bit format has a transfer rate of 110 baud.

The terminal has a keyboard and a printer. Every time a key is depressed, the terminal sends 11 bits serially along a wire. To print a character. To print a character in the printer, an 11-bit message must be received along another wire. The terminal interface consists of a transmitter and a receiver. The transmitter accepts an 8-bit character room the computer and proceeds to send a serial 11-bit message into the printer line. The receiver accepts a serial 11-bit message from the keyboard line and forwards the 8-bit character code into the computer. Integrated circuits are available which are specifically designed to provide the interface between computer and similar interactive terminals. Such a circuit is called an asynchronous communication interface or a universal asynchronous receiver-transmitter (UART).

Asynchronous Communication Interface

Fig shows the block diagram of an asynchronous communication interface is shown in Fig. It acts as both a transmitter and a receiver. The interface is initialized for a particular mode of transfer by means of a control byte that is loaded into its control register. The transmitter register accepts a data byte from the CPU through the data bus. This byte is transferred to a shift register for serial transmission. The receiver portion receives serial information into an- other shift register, and when a complete data byte is accumulated, it is transferred to the receiver register. The CPU can select the receiver register to read the byte through the data bus. The bits in the status register are used for input and output flags and for recording certain errors that may occur during the transmission. The CPU can read the status register to check the status of the flag bits and to determine if any errors have occurred. The chip select and the read and write control lines communicate with the CPU. The chip select (CS) input is used to select the interface through the address bus. The register select (RS) is associated with the read (RD)andwrites (WR) controls. Two registers are write-only and two are read-only. The register selected is a function of the RS value and the RD and WR status, as listed in the table accompanying the diagram.



CS	RS	Operation	Register selected
0	×	×	None: data bus in high-impedance
1	0	WR	Transmitter register
1	1	WR	Control register
1	0	RD	Receiver register
1	1	RD	Status register

MODES OF TRANSFER

Binary information received from an external device is usually stored in memory for later processing. Information transferred from the central computer into an external device originates in the memory unit. The CPU merely executes the I/O instructions and may accept the data temporarily, but the ultimate source or destination is the memory unit. Data transfer between the central computer and I/O devices may be handled in a variety of modes. Some modes use the CPU as.An intermediate path; other transfer the data directly to and from the memory unit. Data transfer to and from peripherals may be handled in one of three possible modes:

Programmed I/O

Interrupt-initiated I/O

Direct memory access (DMA)

Programmed I/O operations are the result of I/O instructions written in the computer program. Each data item transfer is initiated by an instruction in the program. Usually, the transfer is to and from a CPU register and peripheral. Other instructions are needed to transfer the data to and from CPU and memory. Transferring data under program control requires constant monitoring of the peripheral by the CPU. Once a data transfer is initiated, the CPU is required to monitor the interface to see when a transfer can again be made. It is up to the programmed instructions executed in the CPU to keep close tabs on everything that is taking place in the interface unit and the I/O device.

In the programmed I/O method, the CPU stays in a program loop until the I/O unit indicates that it is ready for data transfer. This is a time-consuming process since it keeps the processor busy needlessly. It can be avoided by using an interrupt facility and special commands to inform the interface to issue an interrupt request signal when the data are available from thedevice. In the meantime the CU can proceed to execute another program. The interface meanwhile keeps monitoring the device. When the interface determines that the device is ready for data transfer, it generates an interrupt request to the computer. Upon detecting the external interrupt signal, the CPU momentarily stops the task it is processing, branches to a service program to process the I/O transfer, and then returns to the task it was originally performing.

Transfer of data under programmed I/O is between CPU and peripheral. In direct memory access (DMA), the interface transfers data into and out of the memory unit through the memory bus. The CPU initiates the transfer by supplying the interface with the starting address and the number of words needed to be transferred and then proceeds to execute other tasks. When the transfer is made, the DMA requests memory cycles through the memory bus. When the request is granted by the memory controller, the DMA transfers the data directly into memory. The CPU merely delays its memory access operation to allow the direct memory I/O transfer. Since peripheral speed is usually slower than processor speed, I/O-memory transfers are infrequent compared to processor access to memory.

Many computers combine the interface logic with the requirements for direct memory access into one unit and call it an I/O processor (IOP). The IOP can handle many peripherals through a DMPA and interrupt facility. In such a system, the computer is divided into three separate modules: the memory unit, the CPU, and the IOP.

EXAMPLE OF PROGRAMMED I/O

In the programmed I/O method, the I/O device dies not have direct access to memory. A transfer from an I/O device to memory requires the execution of several instructions by the CPU, including an input instruction to transfer the data from the device to the CPU, and a store instruction to transfer the data from the CPU to memory. Other instructions may be needed to verify that the data are available from the device and to count the numbers of words transferred.

An example of data transfer from an I/O device through an interface into the CPU is shown in below Fig. The device transfers bytes of data one at a time as they are available. When a byte of data is available, the device places it in the I/O bus and enables its data valid line. The interface accepts the byte into its data register and enables the data accepted line. The interface sets it in the status register that we will refer to as an F or “flag” bit. The device can now disable the data valid line, but it will not transfer another byte until the data accepted line is disabled by the interface. This is according to the handshaking procedure established in Fig.

A program is written for the computer to check the flag in the status register to determine if a byte has been placed in the data register by the I/O device. This is done by reading the status register into a CPU register and checking the value of

The flag bit. If the flag is equal to 1, the CPU reads the data fromthe data register. The flag bit is then cleared to 0 by either the CPU or the interface, depending on how the interface circuits are designed. Once the flag is cleared, the interface disables the data accepted line and the device can then transfer the next data byte.

A flowchart of the program that must be written for the CPU is shown in Fig. Flowcharts for CPU program to input data. It is assumed that the device is sending a sequence of bytes that must be stored in memory. The transfer of each byte requires three instructions:

Read the status register.

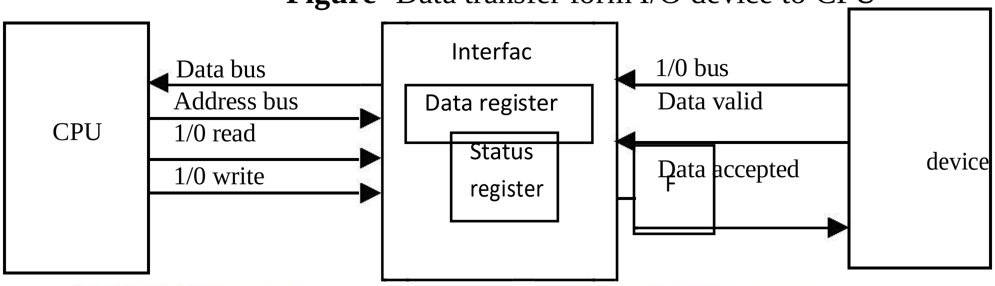
Check the status of the flag bit and branch to step 1 if not set or to step 3 if set.

Read the data register.

Each byte is read into a CPU register and then transferred to memory with a store instruction. A common I/O programming task is to transfer a block of words form an I/O device and store them in a memory buffer.

A program that stores input characters in a memory buffer using the instructions mentioned in the earlier chapter.

Figure -Data transfer form I/O device to CPU



F = Flag bit

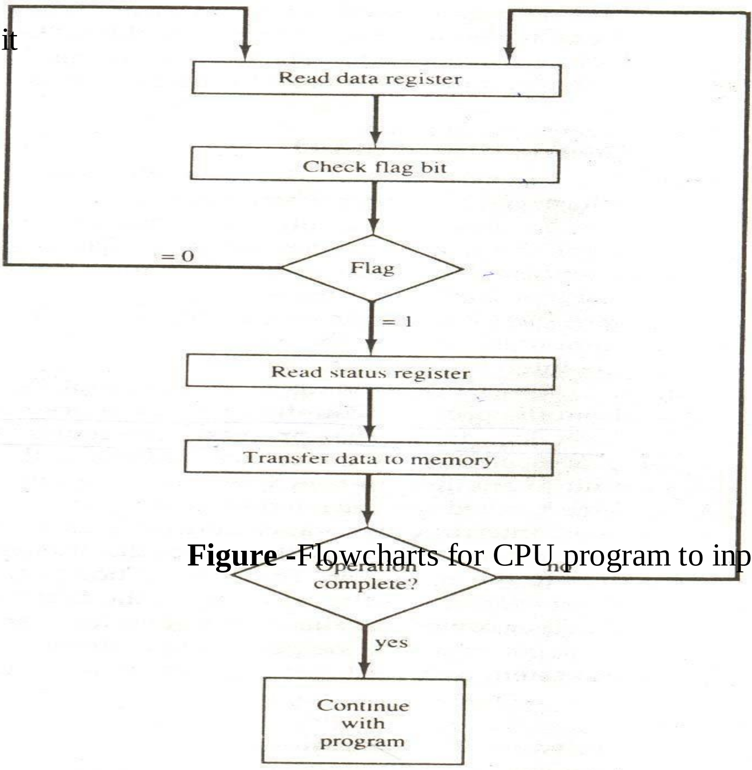


Figure -Flowcharts for CPU program to input data

The programmed I/O method is particularly useful in small low-speed computers or in systems that are dedicated to monitor a device continuously. The difference in information transfer rate between the CPU and the I/O device makes this type of transfer inefficient. To see why this is inefficient, consider a typical computer that can execute the two instructions that read the status register and check the flag in 1 πs. Assume that the input device transfers its data at an average rate of 100 bytes per second. This is equivalent to one byte every 10,000 π s. This means that the CPU will check the flag 10,000 times between each transfer. The CPU is wasting time while checking the flag instead of doing some other useful processing task.

INTERRUPT-INITIATED I/O

An alternative to the CPU constantly monitoring the flag is to let the interface inform the computer when it is ready to transfer data. This mode of transfer uses the interrupt facility. While the CPU is running a program, it does not check the flag. However, when the flag is set, the computer is momentarily interrupted from proceeding with the current program and is informed of the fact that the flag has been set. The CPU deviates from what it is doing to take care of the input or output transfer. After the transfer is completed, the computer returns to the previous program to continue what it was doing before the interrupt.

The CPU responds to the interrupt signal by storing the return address from the program counter into a memory stack and then control branches to a service routine that processes the required I/O transfer. The way that the processor chooses the branch address of the service routine varies from tone unit to another. In principle, there are two methods for accomplishing this. One is called vectored interrupt and the other, no vectored interrupt. In a non-vectored interrupt, the branch address is assigned to a fixed location in memory. In a vectored interrupt, the source that interrupts supplies the branch information to the computer. This information is called the interrupt vector. In some computers the interrupt vector is the first address of the I/O service routine. In other computers the interrupt vector is an address that points to a location in memory where the beginning address of the I/O service routine is stored.

DIRECT MEMORY ACCESS (DMA)

The transfer of data between a fast storage device such as magnetic disk and memory is often limited by the speed of the CPU. Removing the CPU from the path and letting the peripheral device manage the memory buses directly would improve the speed of transfer. This transfer technique is called direct memory access (DMA). During DMA transfer, the CPU is idle and has no control of the memory buses. A DMA controller takes over the buses to manage the transfer directly between the I/O device and memory.

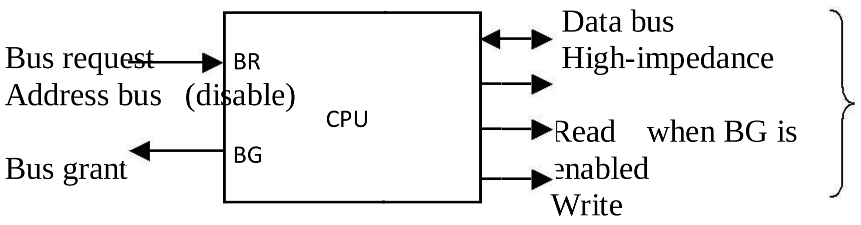
The CPU may be placed in an idle state in a variety of ways. One common method extensively used in microprocessors is to disable the buses through special control signals. Figure- CPU bus signals for DMA transfer. Shows two control signals in the CPU that facilitate the DMA transfer. The bus request (BR) input is used by the DMA controller to request the CPU to relinquish control of the buses. When this input is active, the CPU terminates the execution of the current instruction and places the address bus, the data bus, and the read and write lines into a high-impedance state behaves like an open circuit, which means that the output is disconnected and does not have a logic significance. The CPU activates the Bus grant (BG) output to inform the external DMA that the buses are in the high -impedance state. The DMA that originated the bus request can now take control of the buses to conduct memory transfers without processor intervention. When the DMA terminates the transfer, it disables the bus request line. The CPU disables the bus grant, takes control of the buses, and returns to its normal operation.

When the DMA takes control of the bus system, it communicates directly with the memory. The transfer can be made in several ways. In DMA burst transfer, a block sequence consisting of a number of memory words is transferred in a continuous burst while the DMA controller is master of the memory buses. This mode of transfer is needed for fast devices such as magnetic disks, where data transmission cannot be stopped or slowed down until an entire block is transferred. An alternative technique called cycle stealing allows the DMA controller to transfer one data word at a time after which it must return control of the buses to the CPU. The CPU merely delays its operation for one memory cycle to allow the direct memory I/O transfer to “steal” one memory cycle.

DMA CONTROLLER

The DMA controller needs the usual circuits of an interface to communicate with the CPU and I/O device. In addition, it needs an address register, a word count register, and a set of address lines. The address registers and addresses lines

Figure -CPU bus signals for DMA transfer.



Are used for direct communication with the memory. The word count register specifies the number of words that must be transferred. The data transfer may be done directly between the device and memory under control of the DMA.

Below Figure shows the block diagram of a typical DMA controller. The unit communicates with the CPU via the data bus and control lines. The registers in the DMA are selected by the CPU through the address bus by enabling the DS (DMA select) and RS (register select) inputs. The RD (read) and WR (write) inputs are bidirectional. When the BG (bus grant) input is 0, the CPU can communicate with the DMA registers through the data bus to read from or write to the DMA registers. When BG = 1, the CPU has relinquished the buses and the DMA can communicate directly with the memory by specifying an address in the address bus and activating the RD or WR control. ; The DMA communicates with the external peripheral through the request and acknowledge lines by using a prescribed handshaking procedure.

The DMA controller has three registers: an address register, a word count register, and a control register. The address register contains an address to specify the desired location in memory. The address bits go through bus buffers into the address bus. The address register is incremented after each word that is transferred to memory. The word count register is incremented after each word that is transferred to memory. The word count register holds the number of words to be transferred. This register is decremented by one after each word transfer and internally tested for zero. The control register specifies the mode of transfer. All registers in the DMA appear to the CPU as I/O interface registers. Thus the CPU can read from or write into the DMA registers under program control via the data bus

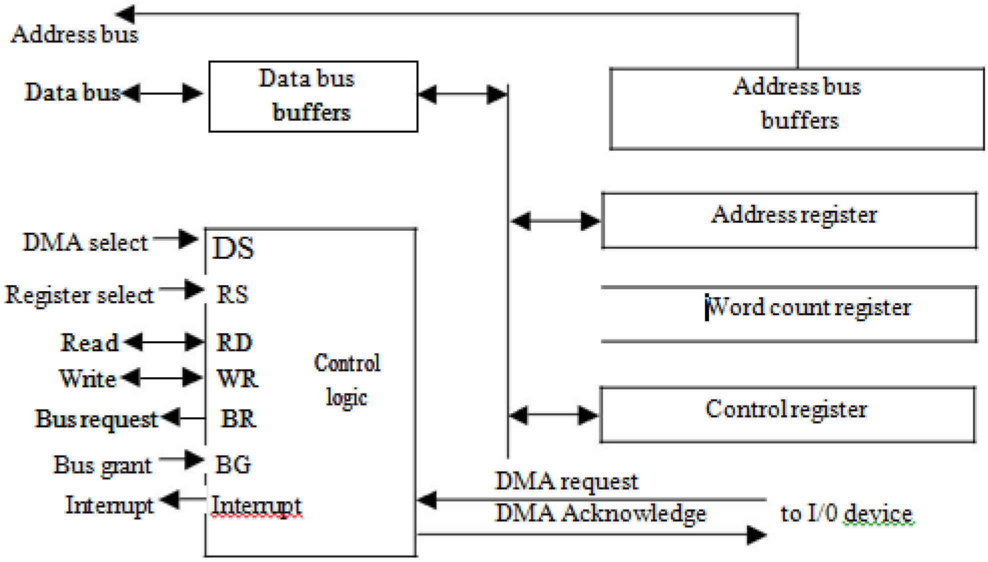


Figure-Block diagram of DMA controller.

The DMA is first initialized by the CPU. After that, the DMA starts and continues to transfer data between memory and peripheral unit until an entire block is transferred. The initialization process is essentially a program consisting of I/O instructions that include the address for selecting particular DMA registers. The CPU initializes the DMA by sending the following information through the data bus:

The starting address of the memory block where data are available (for read) or where data are to be stored (for write)

The word count, which is the number of words in the memory block

Control to specify the mode of transfer such as read or write

A control to start the DMA transfer

The starting address is stored in the address register. The word count is stored in the word count register, and the control information in the control register. Once the DMA is initialized, the CPU stops communicating with the DMA unless it receives an interrupt signal or if it wants to check how many words have been transferred.

DMA TRANSFER

The position of the DMA controller among the other components in a computer system is illustrated in below Fig. The CPU communicates with the DMA through the address and data buses as with any interface unit. The DMA has its own address, which activates the DS and RS lines. The CPU initializes the DMA through the data bus. Once the DMA receives the start control command, it can start the transfer between the peripheral device and the memory.

When the peripheral device sends a DMA request, the DMA controller activates the BR line, informing the CPU to relinquish the buses. The CPU responds with its BG line, informing the DMA that its buses are disabled. The DMA then puts the current value of its address register into the address bus, initiates the RD or WR signal, and sends a DMA acknowledge to the peripheral device. Note that the RD and WR lines in the DMA controller are bidirectional. Thedirection of transfer depends on the status of the BG line. When BG line. When BG = 0, the RD and WR are input lines allowing the CPU to communicate with the internal DMA registers. When BG = 1, the RD and WR and output lines from the DMA controller to the random- access memory to specify the read or write operation for the data.

When the peripheral device receives a DMA acknowledge, it puts a word in the data us (for write) or receives a word from the data bus (for read). Thus the DMA controls the read or write operations and supplies the address for the memory. The peripheral unit can then communicate with memory through the data bus for direct transfer between the two units while the CPU is momentarily disabled.

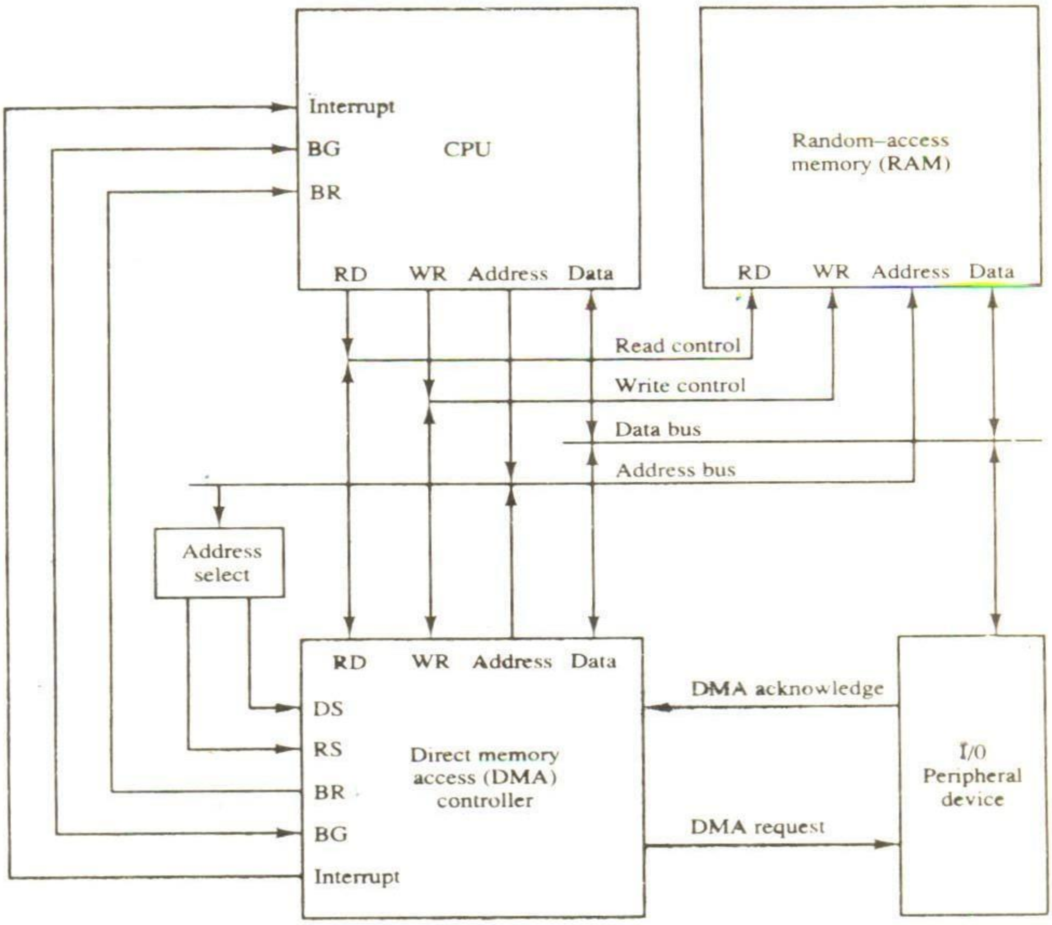


Figure-DMA transfer in a computer system.

For each word that is transferred, the DMA increments its address registers and decrements its word count register. If the word count does not reach zero, the DMA checks the request line coming from the peripheral. For a high-speed device, the line will be active as soon as the previous transfer is completed. A second transfer is then initiated, and the process continues until the entire block is transferred. If the peripheral speed is slower, the DMA request line may come somewhat later. In this case the DMA disables the bus request line so that the CPU can continue to execute its program. When the peripheral requests a transfer, the DMA requests the buses again.

It the word count register reaches zero, the DMA stops any further transfer and removes its bus request. It also informs the CPU of the termination by means of an interrupt. When the CPU responds to the interrupt, it reads the content of the word count register. The zero value of this register indicates that all the words were transferred successfully. The CPU can read this register at any time to check the number of words already transferred.

A DMA controller may have more than on channel. In this case, each channel has a request and acknowledges pair of control signals which are connected to separate peripheral devices. Each channel also has its own address register and word count register within the DMA controller. A priority among the channels may be established so that channels with high priority are serviced before channels with lower priority.

DMA transfer is very useful in many applications. It is used for fast transfer of information between magnetic disks and memory. It is also useful for updating the display in an interactive terminal. Typically, an image of the screen display of the terminal is kept in memory which can be updated under program control. The contents of the memory can be transferred to the screen periodically by means of DMA transfer.

Input-output Processor (IOP)

The IOP is similar to a CPU except that it is designed to handle the details of I/O processing. Unlike the DMA controller that must be set up entirely by the CPU, the IOP can fetch and execute its own instructions. IOP instructions are specially designed to facilitate I/O transfers. In addition, the IOP can perform other processing tasks, such as arithmetic, logic, branching, and code translation.

The block diagram of a computer with two processors is shown in below Figure. The memory unit occupies a central position and can communicate with each processor by means of direct memory access. The CPU is responsible for processing data needed in the solution of computational tasks. The IOP provides a path for transfer of data between various peripheral devices and the memory unit.

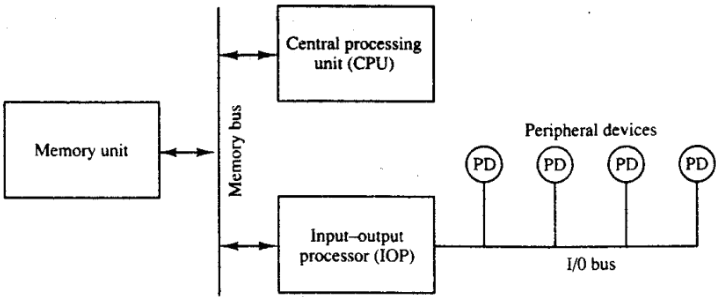


Figure- Block diagram of a computer with I/O processor

The data formats of peripheral devices differ from memory and CPU data formats. The IOP must structure data words from many different sources. For example, it may be necessary to take four bytes from an input device and pack them into one 32-bit word before the transfer to memory. Data are gathered in the IOP at the device rate and bit capacity while the CPU is executing its own program. After the input data are assembled into a memory word, they are transferred from IOP directly into memory by "stealing" one memory cycle from the CPU. Similarly, an output word transferred from memory to the IOP is directed from the IOP to the output device at the device rate and bit capacity.

The communication between the IOP and the devices attached to it is similar to the program control method of transfer. The way by which the CPU and IOP communicate depends on the level of sophistication included in the system. In most computer systems, the CPU is the master while the IOP is a slave processor. The CPU is assigned the task of initiating all operations, but I/O instructions are execute in the IOP. CPU instructions provide operations to start an I/O transfer and also to test I/O status conditions needed for making decisions on various I/O activities. The IOP, in turn, typically asks for CPU attention by means of an interrupt. It also responds to CPU requests by placing a status word in a prescribed location in memory to be examined later by a CPU program. When an I/O operation is desired, the CPU informs the IOP where to find the I/O program and then leaves the transfer details to the IOP.

<u>CPU-IOP Communication</u>

The communication between CPU and IOP. These are depending on the particular computer considered. In most cases the memory unit acts as a message center where each processor leaves information for the other. To appreciate the operation of a typical IOP, we will illustrate by a specific example the method by which the CPU and IOP communicate. This is a simplified example that omits many operating details in order to provide an overview of basic concepts.

The sequence of operations may be carried out as shown in the flowchart of below Fig. The CPU sends an instruction to test the IOP path. The IOP responds by inserting a status word in memory for the CPU to check. The bits of the status word indicate the condition of the IOP and I/O device, such as IOP overload condition, device busy with another transfer, or device ready for I/O transfer. The CPU refers to the status word in memory to decide what to do next. If all is in order, the CPU sends the instruction to start I/O transfer. The memory address received with this instruction tells the IOP where to find its program.

The CPU can now continue with another program while the IOP is busy with the I/O program. Both programs refer to memory by means of DMA transfer. When the IOP terminates the execution of its program, it sends an interrupt request to the CPU. The CPU responds to the interrupt by issuing an instruction to read the status from the IOP. The IOP responds by placing the contents of its status report into a specified memory location.

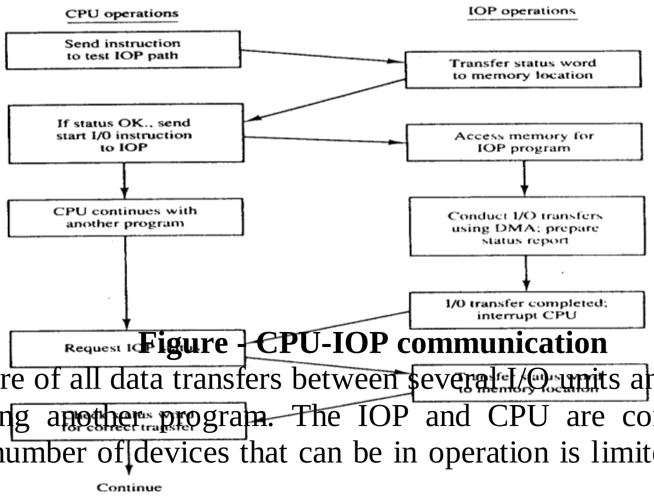

CPU operations
IOP operations
Send instruction to test IOP path
Transfer status word to memory location
If status OK., send start I/O instruction to IOP
Access memory for IOP program
CPU continues with another program
Conduct I/O transfers using DMA; prepare status report
I/O transfer completed; interrupt CPU
Request IOP status
Transfer status word to memory location
Check status word for correct transfer
Continue

Figure - CPU-IOP communication

The IOP takes care of all data transfers between several I/O units and the memory while the CPU is processing another program. The IOP and CPU are competing for the use of memory, so the number of devices that can be in operation is limited by the access time of the memory.

<u>INTEL 8089 IOP</u>

The Intel 8089 I/O processor is contained in a 40-pin integrated circuit package. Within the 8089 are two independent units called channels. Each channel combines the general characteristics of a processor unit with those of a direct memory access controller. The 8089 is designed to function as an IOP in a

Microcomputer system where the Intel 8086 microprocessor is used as the CPU. The 8086 CPU initiates an I/O operation by building a message in memory that describes the function to be performed. The 8089 IOP reads the message from memory, carries out the operation, and notifies the CPU when it has finished.

In contrast to the IBM 370 channel, which has only six basic I/O commands, the 8089 IOP has 50 basic instructions that can operate on individual bits, on bytes, or 16-bit words. The IOP can execute programs in a manner similar to a CPU except that the instruction set is specifically chosen to provide efficient input-output processing. The instruction set includes general data transfer instructions, basic arithmetic and logic operations, conditional and unconditional branch operations, and subroutine call and return capabilities. The set also includes special instructions to initiate DMA transfers and issue an interrupt request to the CPU. It provides efficient data transfer between any two components attached to the system bus, such as I/O to memory, memory to memory, or I/O to I/O.

A microcomputer system using the Intel 8086/8089 pair of integrated circuits is shown in Fig. (a). The 8086 functions as the CPU and the 8089 as the IOP. The two units share a common memory through a bus controller connected to a system bus, which is called a "multibus" by Intel. The IOP uses a local bus to communicate with various interface units connected to I/O devices. The CPU communicates with the IOP by enabling the channel attention line. The select line is used by the CPU to select one of two channels in the 8089. The IOP gets the attention of the CPU by sending an interrupt request.

The CPU and IOP communicate with each other by writing messages for one another in system memory. The CPU prepares the message area and signals the IOP by enabling the channel attention line. The IOP reads the message, performs the required I/O functions, and executes the appropriate channel program. When the channel has completed its program, it issues an interrupt request to the CPU.

The communication scheme consists of program sections called "blocks," which are stored in memory as shown in Fig. (b). each block contains control and parameter information as well as an address pointer to its successor block. The address of the control block is passed to each IOP channel during initialization. The busy flag indicates whether the IOP is busy or ready to perform a new I/O operation. The CCW (channel command word) is specified by the CPU to indicate the of operation required from the IOP. The CCW in the 8089 does not have the same meaning as the command word in the IBM

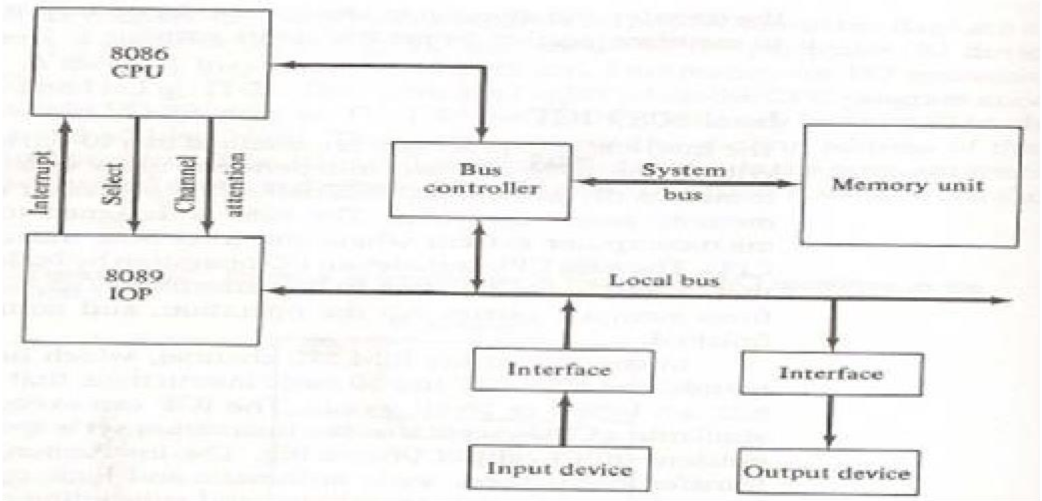


Fig (a)- Intel 8086/8089 Microcomputer system block diagram.

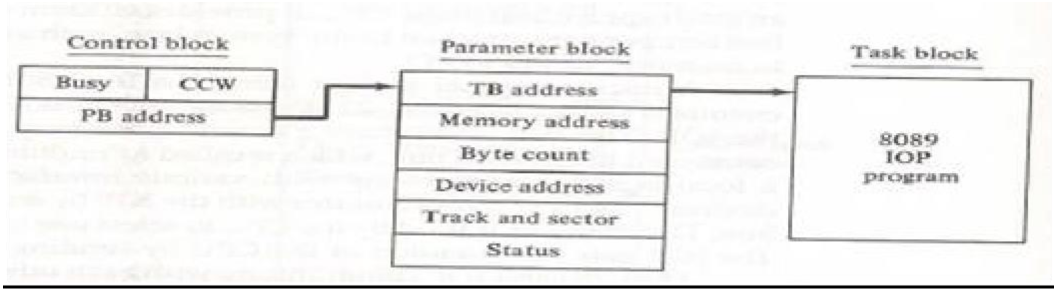


Fig (b) – Location of information in memory for I/O operations in the Intel8086/8089 microcomputer system.

The CPU and IOP work together through the control and parameter blocks. The CPU obtains use of the shared memory after checking the busy flag to ensure that the IOP is available. The CPU then fills in the information in the parameter block and writes a "start operation" command in the CCW. After the communication blocks have been set up, the CPU enables the channel attention signal to inform the IOP to start its I/O operation. The CPU then continues with another program. The IOP responds to the channel attention signal by placing the address of the control block into its program counter. The IOP refers to the control block and sets the busy flag. It then checks the operation in the CCW. The PB (parameter block) address and TB (task block) address are then transferred into internal IOP registers. The IOP starts executing the program in the task block using the information in the parameter block. The entries in the parameter block depend on the I/O device. The parameters listed in Fig. (b)are suitable for data transfer to or from a magnetic disk. The memory address specifies the beginning address of a memory buffer. The byte count gives the number of bytes to be transferred. The device address specifies the particular I/O device to be used. The track and sector numbers locate the data on the disk. When the I/O operation is completed, the IOP stores its status bits in the status word location of the parameter block and interrupts the CPU. The CPU can refer to the status word to check if the transfer has been completed satisfactorily.

****

UNIT-III

Memory Organizations: Memory hierarchy, Main Memory, RAM, ROM Chips, Memory Address Map, Memory Connection to CPU, associate memory, Cache Memory, Data Cache, Instruction cache, Miss and Hit ratio, Access time, associative, set associative, mapping, waiting into cache, Introduction to virtual memory.

MEMORY HIERARCHY

The memory unit is an essential component in any digital computer since it is needed for storing programs and data. A very small computer with a limited application may be able to fulfill its intended task without the need of additional storage capacity. Most general-purpose computers would run more efficiently if they were equipped with additional storage beyond the capacity of the main memory. There is just not enough space in one memory unit to accommodate all the programs used in a typical computer. Moreover, most computer users accumulate and continue to accumulate large amounts of data-processing software. Not all accumulated information is needed by the processor at the same time. Therefore, it is more economical to use low-cost storage devices to serve as a backup for storing the information that is not currently used by the CPU. The memory unit that communicates directly with the CPU is called the main memory. Devices that provide backup storage are called auxiliary memory. The most common auxiliary memory devices used in computer systems are magnetic disks and tapes. They are used for storing system programs, large data files, and other backup information. Only programs and data currently needed by the processor reside in main memory. All other information is stored in auxiliary memory and transferred to main memory when needed.

The total memory capacity of a computer can be visualized as being a hierarchy of components. The memory hierarchy system consists of tall storage devices employed in a computer system from the slow but high-capacity auxiliary memory to a relatively faster main memory, to an even smaller and faster cache memory accessible to the high-speed processing logic. **Figure** illustrates the components in a typical memory hierarchy. At the bottom of the hierarchy are the relatively slow magnetic tapes used to store removable files. Next are the magnetic disks used as backup storage. The main memory occupies a central position by being able to communicate directly with the CPU and with auxiliary memory devices through an I/O processor. When programs not residing in main memory are needed by the CPU, they are brought in from auxiliary memory. Programs not currently needed in main memory are transferred into auxiliary memory to provide space for currently used programs and data.

A special very-high speed memory called a cache is sometimes used to increase the speed of processing by making current programs and data available to the CPU at a rapid rate. The cache memory is employed in computer systems to compensate for the speed differential between main memory access time and processor logic. CPU logic is usually faster than main memory access time, with the result that processing speed is limited primarily by the speed of main memory. A technique used to compensate for the mismatch in operating speeds is to employ in extremely fast, small cache between the CPU and main memory whose access time is close to processor logic clock cycle time. The cache is used for storing segments of programs currently being executed in the CPU and temporary data frequently needed in the present calculations by .Making programs and data available at a rapid rate, it is possible to increase the performance rate of the computer.

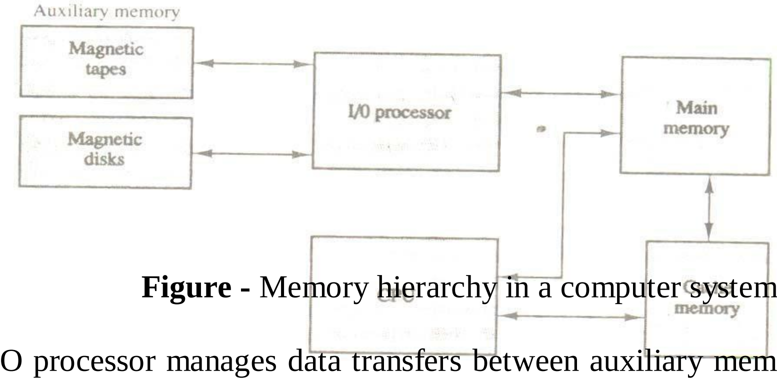


Figure - Memory hierarchy in a computer system

While the I/O processor manages data transfers between auxiliary memory and main memory, the cache organization is concerned with the transfer of information between main memory and CPU. Thus each is involved with a different level in the memory hierarchy system. The reason for having two or three levels of memory hierarchy is economics. As the storage capacity of the memory increases, the cost per bit for storing binary information decreases and the access time of the memory becomes longer. The auxiliary memory has a large storage capacity, is relatively inexpensive, but has low access speed compared to main memory. The cache memory is very small, relatively expensive, and has very high access speed. Thus as the memory access speed increases, so does its relative cost. The overall goal of using a memory hierarchy is to obtain the highest-possible average access speed while minimizing the total cost of the entire memory system. While the I/O processor manages data transfers between auxiliary memory and main memory, the cache organization is concerned with the transfer of information between main memory and CPU. Thus each is involved with a different level in the memory hierarchy system. The reason for having two or three levels of memory hierarchy is economics. As the storage capacity of the memory increases, the cost per bit for storing binary information decreases and the access time of the memory becomes longer. The auxiliary memory has a large storage capacity, is relatively inexpensive, but has low access speed compared to main memory. The cache memory is very small, relatively expensive, and has very high access speed. Thus as the memory access speed increases, so does its relative cost. The overall goal of using a memory hierarchy is to obtain the highest-possible average access speed while minimizing the total cost of the entire memory system.

Auxiliary and cache memories are used for different purposes. The cache holds those parts of the program and data that are most heavily used, while the auxiliary memory holds those parts that are not presently used by the CPU. Moreover, the CPU has direct access to both cache and main memory but not to auxiliary memory. The transfer from auxiliary to main memory is usually done by means of direct memory access of large blocks of data. The typical access time ratio between cache and main memory is about 1 to 7. For example, a typical cache memory may have an access time of 100ns, while main memory access time may be 700ns. Auxiliary memory average access time is usually 1000 times that of main memory. Block size in auxiliary memory typically ranges from256 to 2048 words, while cache block size is typically from 1 to 16 words.

Many operating systems are designed to enable the CPU to process a number of independent programs concurrently. This concept, called multiprogramming, refers to the existence of two or more programs indifferent parts of the memory hierarchy at the same time. In this way it is possible to keep all parts of the computer busy by working with several programs in sequence. For example, suppose that a program is being executed in the CPU and an I/O transfer is required. The CPU initiates the I/O processor to start executing the transfer. This leaves the CPU free to execute another program. In a multiprogramming system, when one program is waiting for input or output transfer, there is another program ready to utilize the CPU.

MAIN MEMORY

The main memory is the central storage unit in a computer system. It is a relatively large and fast memory used to store programs and data during the computer operation. The principal technology used for the main memory is based on semiconductor integrated circuits. Integrated circuit RAM chips are available in two possible operating modes, static and dynamic. The static RAM consists essentially of internal flip-flops that store the binary information. The stored information remains valid as long as power is applied to unit. The dynamic RAM stores the binary information in the form of electric charges that are applied to capacitors. The capacitors are provided inside the chip by MOS transistors. The stored charges on the capacitors tend to discharge with time and the capacitors must be periodically recharged by refreshing the dynamic memory. Refreshing is done by cycling through the words every few milliseconds to restore the decaying charge. The dynamic RAM offers reduced power consumption and larger storage capacity in a single memory chip. The static RAM is easier to use and has shorted read and write cycles.

Most of the main memory in a general-purpose computer is made up of RAM integrated circuit chips, but a portion of the memory may be constructed with ROM chips. Originally, RAM was used to refer to a random- access memory, but now it is used to designate a read/write memory to distinguish it from a read-only memory, although ROM is also random access. RAM is used for storing the bulk of the programs and data that are subject to change. ROM is used for storing programs that are permanently resident in the computer and for tables of constants that do not change in value one the production of the computer is completed.

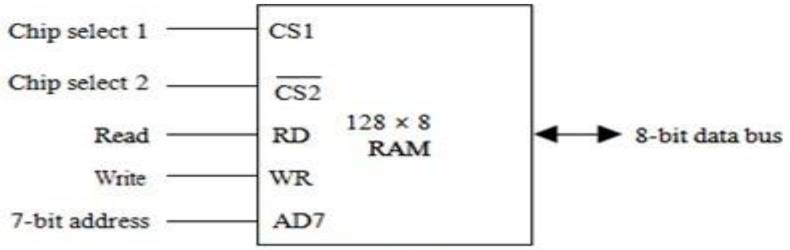
Among other things, the ROM portion of main memory is needed for storing an initial program called a bootstrap loader. The bootstrap loader is a program whose function is to start the computer software operating when power is turned on. Since RAM is volatile, its contents are destroyed when power is turned off. The contents of ROM remain unchanged after power is turned off and on again. The startup of a computer consists of turning the power on and starting the execution of an initial program. Thus when power is turned on, the hardware of the computer sets the program counter to the first address of the bootstrap loader. The bootstrap program loads a portion of the operating system from disk to main memory and control is then transferred to the operating system, which prepares the computer from general use.

RAM and ROM chips are available in a variety of sizes. If the memory needed for the computer is larger than the capacity of one chip, it is necessary to combine a number of chips to form the required memory size. To demonstrate the chip interconnection, we will show an example of a 1024 × 8 memory constructed with 128 × 8 RAM chips and 512 × 8 ROM chips.

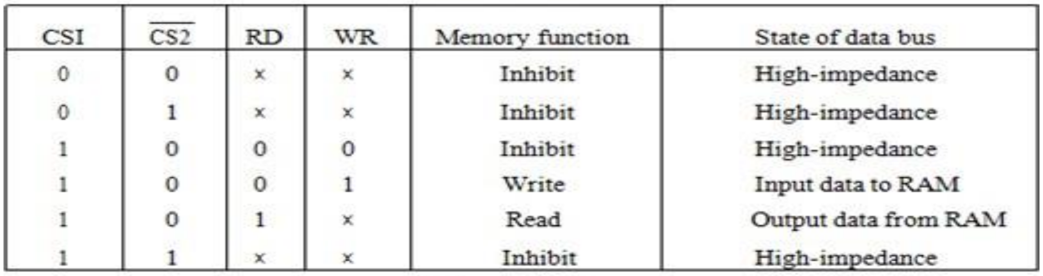
RAM AND ROM CHIPS

A RAM chip is better suited for communication with the CPU if it has one or more control inputs that select the chip only when needed. Another common feature is a bidirectional data bus that allows the transfer of data either from memory to CPU during a read operation or from CPU to memory during a write operation. A bidirectional bus can be constructed with three-state buffers. A three-state buffer output can be placed in one of three possible states: a signal equivalent to logic 1, a signal equivalent to logic 0, or a high-impedance state. The logic 1 and 0 are normal digital signals. The high-impedance state behaves like an open circuit, which means that the output does not carry a signal and has no logic significance. The block diagram of a RAM chip is shown in Fig. The capacity of the memory is 128 words of eight bits (one byte) per word. This requires a 7-bit. Address and an 8-bit bidirectional data bus. The read and write inputs specify the memory operation and the two chips select (CS) control inputs are for enabling the chip only when it is selected by the microprocessor. The availability of more than one control input to select the chip facilitates the decoding of the address lines when multiple chips are used in the microcomputer. The read and write inputs are sometimes combined into one line labeled R/W. When the chip is selected, the two binary states in this line specify the two operations or read or write.

The function table listed in Fig. (b) Specifies the operation of the RAM chip. The unit is in operation only when CSI = 1 and CS2 = 0. The bar on top of the second select variable indicates that this input in enabled when it is equal to 0. If the chip select inputs are not enabled, or if they are enabled but the read but the read or write inputs are not enabled, the memory is inhibited and its data bus is in a high-impedance state. When SC1 = 1 and CS2 = 0, the memory can be placed in a write or read mode. When the WR input is enabled, the memory stores a byte from the data bus into a location specified by the address input lines. When the RD input is enabled,



(a) Block diagram



CSI	$\overline{CS2}$	RD	WR	Memory function	State of data bus
0	0	×	×	Inhibit	High-impedance
0	1	×	×	Inhibit	High-impedance
1	0	0	0	Inhibit	High-impedance
1	0	0	1	Write	Input data to RAM
1	0	1	×	Read	Output data from RAM
1	1	×	×	Inhibit	High-impedance

(b) Function table

the content of the selected byte is placed into the data bus. The RD and WR signals control the memory operation as well as the bus buffers associated with the bidirectional data bus.

A ROM chip is organized externally in a similar manner. However, since a ROM can only read, the data bus can only be in an output mode. The block diagram of a ROM chip is shown in below Fig. For the same-size chip, it is possible to have more bits of ROM occupy less space than in RAM. For this reason, the diagram specifies a 512-byte ROM, while the RAM has only 128 bytes.

The nine address lines in the ROM chip specify any one of the 512 bytes stored in it. The two chip select inputs must be CS1 = 1 and CS2 = 0 for the unit to operate. Otherwise, the data bus is in a high-impedance state. There is no need for a read or write control because the unit can only read. Thus when the chip is enabled by the two select inputs, the byte selected by the address lines appears on the data bus.

MEMORY ADDRESS MAP

The designer of a computer system must calculate the amount of memory required for the particular application and assign it to either RAM or ROM. The interconnection between memory and processor is then established form knowledge of the size of memory needed and the type of RAM and ROM chips available. The addressing of memory can be established by means of a table that specifies the memory address assigned to each chip. The table, called a memory address map, is a pictorial representation of assigned address space for each chip in the system.

To demonstrate with a particular example, assume that a computer system needs 512 bytes of RAM and 512 bytes of ROM. The RAM and ROM chips

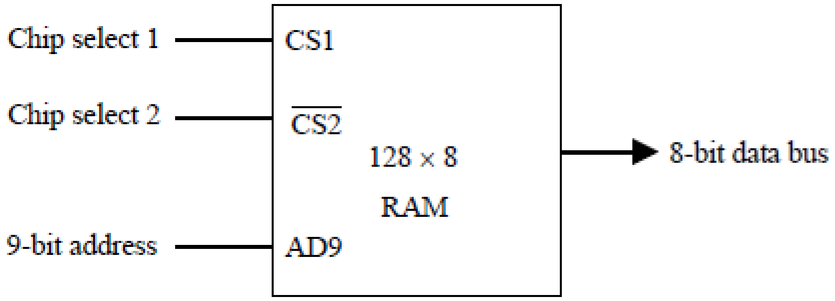


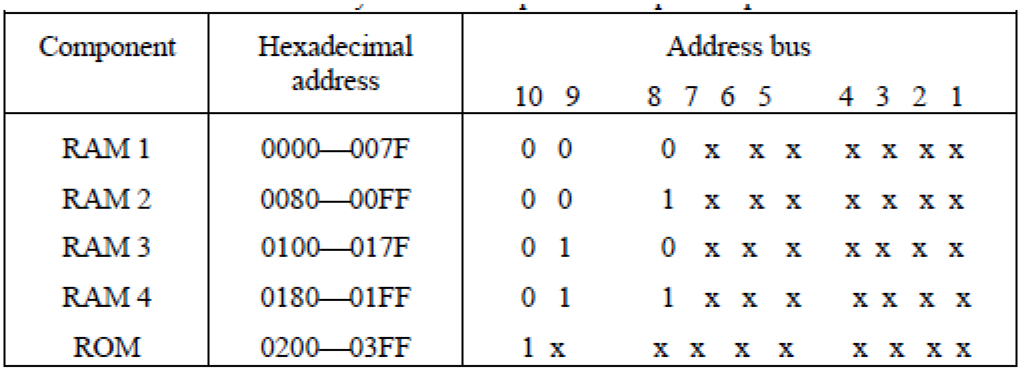
Figure-Typical ROM chip.

To be used are specified in Fig Typical RAM chip and Typical ROM chip. The memory address map for this configuration is shown in Table 7-1. The component column specifies whether a RAM or a ROM chip is used. The hexadecimal address column assigns a range of hexadecimal equivalent addresses for each chip. The address bus lines are listed in the third column. Although there are 16 lines in the address bus, the table shows only 10 lines because the other 6 are not used in this example and are assumed to be zero. The small x's under the address bus lines designate those lines that must be connected to the address inputs in each chip. The RAM chips have 128 bytes and need seven address lines. The ROM chip has 512 bytes and needs 9 address lines. The x's are always assigned to the low-order bus lines: lines 1 through 7 for the RAM and lines 1 through 9 for the ROM. It is now

necessary to distinguish between four RAM chips by assigning to each a different address. For this particular example we choose bus lines 8 and 9 to represent four distinct binary combinations. Note that any other pair of unused bus lines can be chosen for this purpose. The table clearly shows that the nine low-order bus lines constitute a memory space from RAM equal to 2^9 = 512 bytes. The distinction between a RAM and ROM address

is done with another bus line. Here we choose line 10 for this purpose. When line 10 is 0, the CPU selects a RAM, and when this line is equal to 1, it selects the ROM. The equivalent hexadecimal address for each chip is obtained forms the information under the address bus assignment. The address bus lines are subdivided into groups of four bits each so

TABLE-Memory Address Map for Microprocomputer



Component	Hexadecimal address	Address bus									
		10	9	8	7	6	5	4	3	2	1
RAM 1	0000—007F	0	0	0	x	x	x	x	x	x	x
RAM 2	0080—00FF	0	0	1	x	x	x	x	x	x	x
RAM 3	0100—017F	0	1	0	x	x	x	x	x	x	x
RAM 4	0180—01FF	0	1	1	x	x	x	x	x	x	x
ROM	0200—03FF	1	x	x	x	x	x	x	x	x	x

That each group can be represented with a hexadecimal digit. The first hexadecimal digit represents lines 13 to 16 and is always 0. The next hexadecimal digit represents lines 9 to 12, but lines 11 and 12 are always

The range of hexadecimal addresses for each component is determined from the x's associated with it. This x's represent a binary number that can range from an all-0's to an all-1's value.

MEMORY CONNECTION TO CPU

RAM and ROM chips are connected to a CPU through the data and address buses. The low-order lines in the address bus select the byte within the chips and other lines in the address bus select a particular chip through its chip select inputs. The connection of memory chips to the CPU is shown in below Fig. This configuration gives a memory capacity of 512 bytes of RAM and 512 bytes of ROM. It implements the memory map of Table 7-1. Each RAM receives the seven low-order bits of the address bus to select one of 128 possible bytes. The particular RAM chip selected is determined from lines 8 and 9 in the address bus. This is done through a 2 × 4 decoder whose outputs go to the SCI input in each RAM chip. Thus, when address lines 8 and 9 are equal to 00, the first RAM chip is selected. When 01, the second RAM chip is selected, and so on. The RD and WR outputs from the microprocessor are applied to the inputs of each RAM chip.

The selection between RAM and ROM is achieved through bus line 10.

The RAMs are selected when the bit in this line is 0, and the ROM when the bit is 1. The other chip select input in the ROM is connected to the RD control line for the ROM chip to be enabled only during a read operation. Address bus lines 1 to 9 are applied to the input address of ROM without going through the decoder. This assigns addresses 0 to 511 to RAM and 512 to 1023 to ROM. The data bus of the ROM has only an output capability, whereas the data bus connected to the RAMs can transfer information in both directions.

The example just shown gives an indication of the interconnection complexity that can exist between memory chips and the CPU. The more chips that are connected, the more external decoders are required for selection among the chips. The designer must establish a memory map that assigns addresses to the various chips from which the required connections are determined.

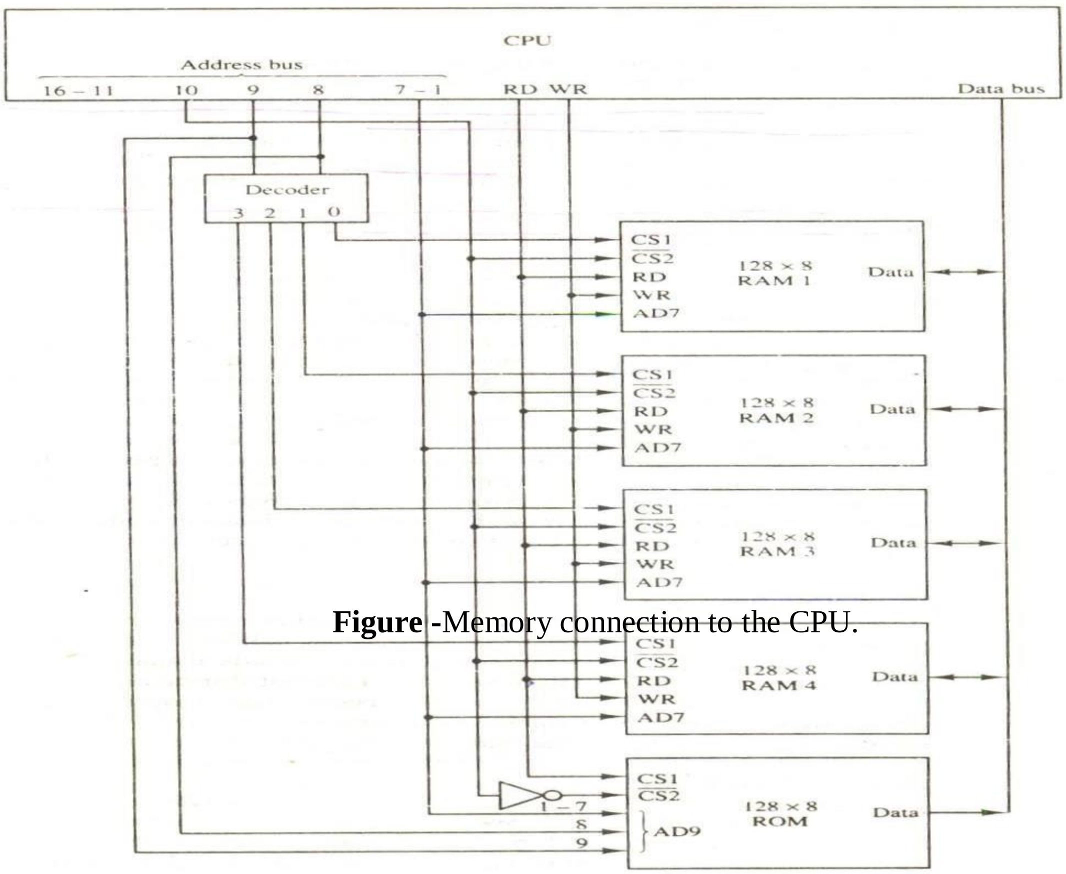


Figure -Memory connection to the CPU.

ASSOCIATIVE MEMORY

Many data-processing applications require the search of items in a table stored in memory. An assembler program searches the symbol address table in order to extract the symbol's binary equivalent. An account number may be searched in a file to determine the holder's name and account status. The established way to search a table is to store all items where they can be addressed in sequence. The search procedure is a strategy for choosing a sequence of addresses, reading the content of memory at each address, and comparing the information read with the item being searched until a match occurs. The number of accesses to memory depends on the location of the item and the efficiency of the search algorithm. Many search algorithms have been developed to minimize the number of accesses while searching for an item in a random or sequential access memory.

The time required to find an item stored in memory can be reduced considerably if stored data can be identified for access by the content of the data itself rather than by an address. A memory unit accessed by content is called an associative memory or content addressable memory (CAM). This type of memory is accessed simultaneously and in parallel on the basis of data content rather than by specific address or location. When a word is written in an associative memory, no address is given,. The memory is capable of finding an empty unused location to store the word. When a word is to be read from an associative memory, the content of the word, or part of the word, is specified. The memory locaters all words which match the specified content and marks them for reading.

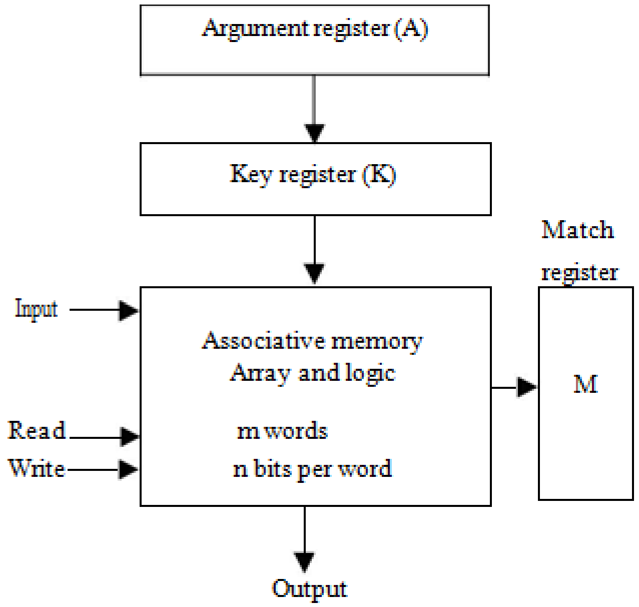
Because of its organization, the associative memory is uniquely suited to do parallel searches by data association. Moreover, searches can be done on an entire word or on a specific field within a word. An associative memory is more expensive then a random access memory because each cell must have storage capability as well as logic circuits for matching its content with an external argument. For this reason, associative memories are used in applications where the search time is very critical and must be very short.

HARDWARE ORGANIZATION

The block diagram of an associative memory is shown in below Fig. It consists of a memory array and logic from words with n bits per word. The argument register A and key register K each have n bits, one for each bit of a word. The match register M has m bits, one for each memory word. Each word in memory is compared in parallel with the content of the argument register. The words that match the bits of the argument register set a corresponding bit in the match register. After the matching process, those bits in the match register that have been set indicate the fact that their corresponding words have been matched. Reading is accomplished by a sequential access to memory for those words whose corresponding bits in the match register have been set.

The key register provides a mask for choosing a particular field or key in the argument word. The entire argument is compared with each memory word if the key register contains all 1's. Otherwise, only those bits in the argument that have 1's in their corresponding position of the key register are compared. Thus the key provides a mask or identifying piece of information which specifies how the reference to memory is made.

Figure- Block diagram of associative memory



To illustrate with a numerical example, suppose that the argument register A and the key register K have the bit configuration shown below. Only the three leftmost bits of A are compared with memory words because K has 1's in these positions.

A	**101**	**111100**	
K	**111**	**000000**	
Word 1	**100**	**111100**	**no match**
Word 2	**101**	**000001**	**match**

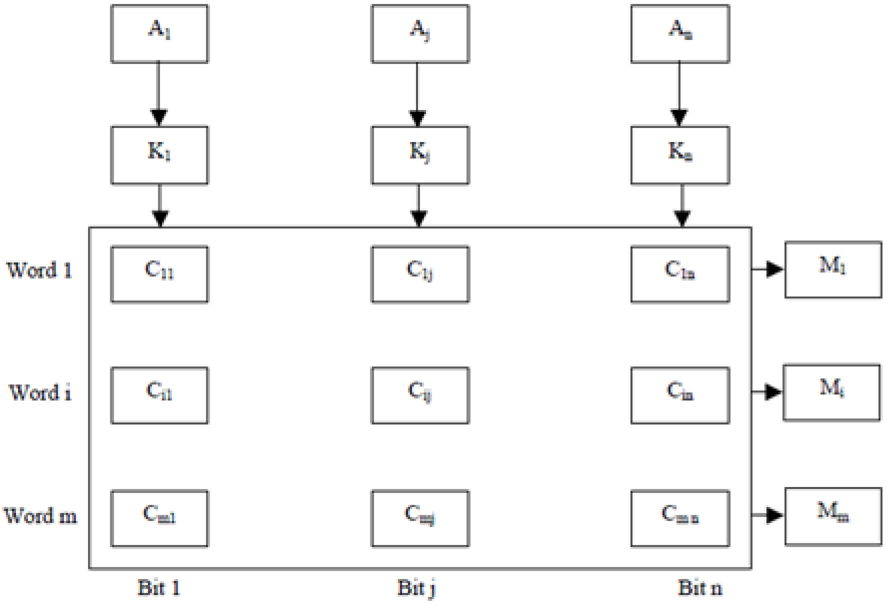
Word 2 matches the unmasked argument field because the three leftmost bits of the argument and the word are equal.

The relation between the memory array and external registers in an associative memory is shown in below Fig. The cells in the array are marked by the letter C with two subscripts. The first subscript gives the

word number and the second specifies the bit position in the word. Thus cell C_{ij} is the cell for bit j in word i. A

bit A_j in the argument register is compared with all the bits in column j of the array provided that $K_j = 1$. This is done for all columns j = 1, 2,…,n. If a match occurs between all the unmasked bits of the argument and the bits in word i, the corresponding bit M_i in the match register is set to 1. If one or more unmasked bits of the argument and the word do not match, M_i is cleared to 0.

Figure -Associative memory of m word, n cells per word



The internal organization of a typical cell C_{ij} is shown in Fig.. It consists of a flip-Flop storage element F_{ij} and the circuits for reading, writing, and matching the cell. The input bit is transferred into the storage cell during a write operation. The bit stored is read out during a read operation. The match logic compares the content of the storage cell with the corresponding unmasked bit of the argument and provides an
output for the decision logic that sets the bit in M_i.

MATCH LOGIC

The match logic for each word can be derived from the comparison algorithm for two binary numbers. First, we neglect the key bits and compare the argument in A with the bits stored in the cells of the words. Word
i is equal to the argument in A if $A_j = F_{ij}$ for j = 1, 2,…, n. Two bits are equal if they are both 1 or both 0. The equality of two bits can be expressed logically by the Boolean function

$x_j = A_jF_{ij} + A'_j F'_{ij}$

Where $x_j = 1$ if the pair of bits in position j are equal; otherwise, $x_j = 0$.

For a word i to be equal to the argument in a we must have all x_j variables equal to 1. This is the condition for setting the corresponding match bit M_i to 1. The Boolean function for this condition is $M_i = x_1\ x_2\ x_3 \dots x_n$

And constitutes the AND operation of all pairs of matched bits in a word. **Figure** One cell of associative memory.

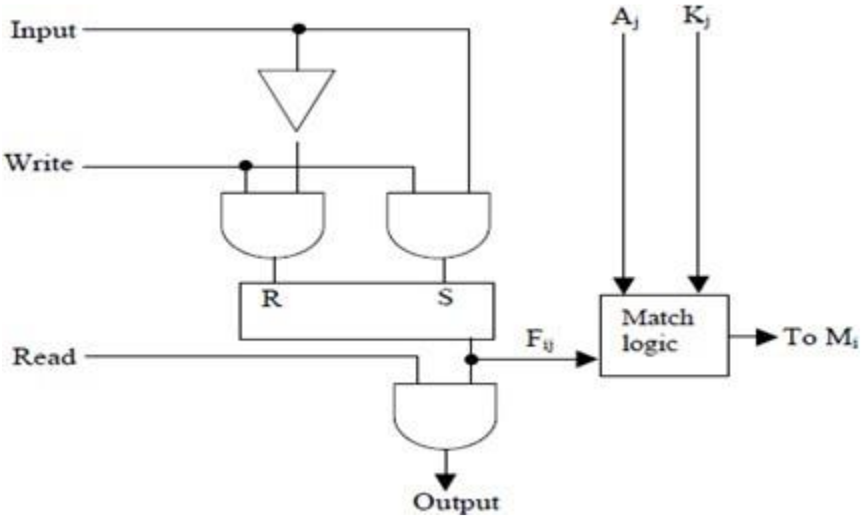


Figure - One cell of associative memory.

We now include the key bit Kj in the comparison logic. The requirement is that if Kj = 0, the corresponding bits of Aj and Fij need no comparison. Only when Kj = 1 must they be compared. This requirement is achieved by ORing each term with Kj’ , thus:

x
xj + K’j =
if K =1
j j
1 if K = 0
j

When K j = 1, we have K j ’ = 0 and xj + 0 = xj. When Kj = 0, then Kj’ = 1 xj + 1 = 1. A term (xj + Kj’) will be in the 1 state if its pair of bitsis not compared. This is necessary because each term is ANDed with all other terms so that an output of 1 will have no effect. The comparison of the bits

has an effect only when Kj = 1. The match logic for word i in an associative memory can now be expressed by the following Boolean function:

$M_i = (x_1 + K')(x_2 + K')(x_3 + K') \ldots (x_n + K')$

Each term in the expression will be equal to 1 if its corresponding K 'j = 0. ifKj = 1, the term will be either 0 or 1 depending on the value of xj. A match will occur and Mi will be equal to 1 if all terms are equal to 1.

If we substitute the original definition of xj. The Boolean function above can be expressed as follows:

$M_i = \prod (A_jF_{ij} + A'F' + K')$
j=1

Where ∏ is a product symbol designating the AND operation of all n terms. We need m such functions, one for each word i = 1, 2, 3, …., m.

The circuit for catching one word is shown in below Fig. Each cell requires two AND gates and one OR gate. The inverters for Aj and Kj are needed once for each column and are used for all bits in the column. The output of all OR gates in the cells of the same word go to the input of a common AND gate to generate the match signal for Mi.Mi will be logic 1 if a catch occurs and 0 if no match occurs. Note that if the key register

contains all 0's, output Mi will be a 1 irrespective of the value of A or the word. This occurrence must be avoided during normal operation.

READ OPERATION

If more than one word in memory matches the unmasked argument field, all the matched words will have 1's in the corresponding bit position of the catch register. It is then necessary to scan the bits of the match register on eat a time. The matched words are read in sequence by applying a read signal to each word line

whose corresponding Mi bit is a 1.

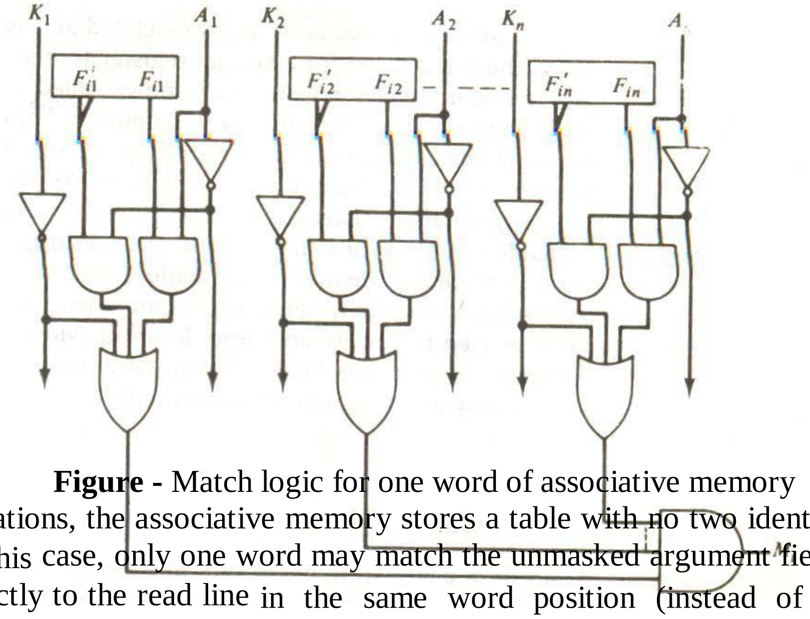


Figure - Match logic for one word of associative memory

In most applications, the associative memory stores a table with no two identical items under a given key. In this case, only one word may match the unmasked argument field. By connecting output Mi directly to the read line in the same word position (instead of the M register), the content of the matched word will be presented

automatically at the output lines and no special read command signal is needed. Furthermore, if we exclude

words having zero content, an all-zero output will indicate that no match occurred and that the searched item is not available in memory.

WRITE OPERATION

An associative memory must have a write capability for storing the information to be searched. Writing in an associative memory can take different forms, depending on the application. If the entire memory is loaded with new information at once prior to a search operation then the writing can be done by addressing each location in sequence. This will make the device a random-access memory for writing and a content addressable memory for reading. The advantage here is that the address for input can be decoded as in a random-access memory. Thus instead of having m address lines, one for each word in memory, the number of address lines can be reduced by the decoder to d lines, where $m = 2^d$.

If unwanted words have to be deleted and new words inserted one at a time, there is a need for a special register to distinguish between active and inactive words. This register, sometimes called a tag register, would have as many bits as there are words in the memory. For every active word stored in memory, the corresponding bit in the tag register is set to 1. A word is deleted from memory by clearing its tag bit to 0. Words are stored in memory by scanning the tag register until the first 0 bit is encountered. This gives the first available inactive word and a position for writing a new word. After the new word is stored in memory it is made active by setting its tag bit to 1. An unwanted word when deleted from memory can be cleared to all 0's if this value is used to

specify an empty location. Moreover, the words that have a tag bit of 0 must be masked (together with the K_j bits) with the argument word so that only active words are compared.

CACHE MEMORY

Analysis of a large number of typical programs has shown that the references, to memory at any given interval of time tend to be confined within a few localized areas in memory. The phenomenon is known as the property of locality of reference. The reason for this property may be understood considering that a typical computer program flows in a straight-line fashion with program loops and subroutine calls encountered frequently. When a program loop is executed, the CPU repeatedly refers to the set of instructions in memory that constitute the loop. Every time a given subroutine is called, its set of instructions is fetched from memory. Thus loops and subroutines tend to localize the references to memory for fetching instructions. To a lesser degree, memory references to data also tend to be localized. Table-lookup procedures repeatedly refer to that portion in memory where the table is stored. Iterative procedures refer to common memory locations and array of numbers are confined within a local portion of memory. The result of all these observations is the locality of reference property, which states that over a short interval of time, the addresses generated by a typical program refer to a few localized areas of memory repeatedly, while the remainder of memory is accessed relatively frequently.

If the active portions of the program and data are placed in a fast small memory, the average memory access time can be reduced, thus reducing the total execution time of the program. Such a fast small memory is referred to as a cache memory. It is placed between the CPU and main memory as illustrated in below Fig. The cache memory access time is less than the access time of main memory by a factor of 5 to 10. The cache is the fastest component in the memory hierarchy and approaches the speed of CPU components.

The fundamental idea of cache organization is that by keeping the most frequently accessed instructions and data in the fast cache memory, the average memory access time will approach the access time of the cache. Although the cache is only a small fraction of the size of main memory, a large fraction of memory requests will be found in the fast cache memory because of the locality of reference property of programs. The basic operation of the cache is as follows. When the CPU needs to access memory, the cache is examined. If the word is found in the cache, it is read from the fast memory. If the word addressed by the CPU is not found in the cache, the main memory is accessed to read the word. A block of words containing the one just accessed is then transferred from main memory to cache memory. The block size may vary from one word (the one just accessed) to about 16 words adjacent to the one just accessed. In this manner, some data are transferred to cache so that future references to memory find the required words in the fast cache memory.

The performance of cache memory is frequently measured in terms of a quantity called hit ratio. When the CPU refers to memory and finds the word in cache, it is said to produce a hit. If the word is not found in cache, it is in main memory and it counts as a miss. The ratio of the number of hits divided by the total CPU references to memory (hits plus misses) is the hit ratio. The hit ratio is best measured experimentally by running representative programs in the computer and measuring the number of hits and misses during a given interval of time. Hit ratios of 0.9 and higher have been reported. This high ratio verifies the validity of the locality of reference property.

The average memory access time of a computer system can be improved considerably by use of a cache. If the hit ratio is high enough so that most of the time the CPU accesses the cache instead of main memory, the average access time is closer to the access time of the fast cache memory. For example, a computer with cache access time of 100 ns, a main memory access time of 1000 ns, and a hit ratio of 0.9 produces an average access time of 200 ns. This is a considerable improvement over a similar computer without a cache memory, whose access time is 1000 ns.

The basic characteristic of cache memory is its fast access time. Therefore, very little or no time must be wasted when searching for words in the cache. The transformation of data from main memory to cache memory is referred to as a mapping process. Three types of mapping procedures are of practical interest when considering the organization of cache memory:

Associative mapping

Direct mapping

Set-associative mapping

To helping the discussion of these three mapping procedures we will use a specific example of a memory organization as shown in below Fig. The main memory can store 32K words of 12 bits each. The cache is capable of storing 512 of these words at any given time. For every word stored in cache, there is a duplicate copy in main memory.The CPU communicates with both memories. It first sends a 15-bit address to cache. If there is a hit, the CPU accepts the 12 -bit data from cache. If there is a miss, the CPU reads the word from main memory and the word is then transferred to cache.

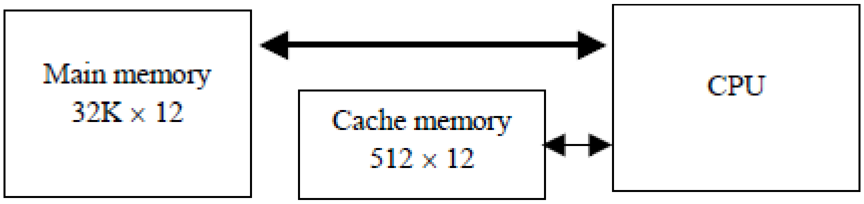
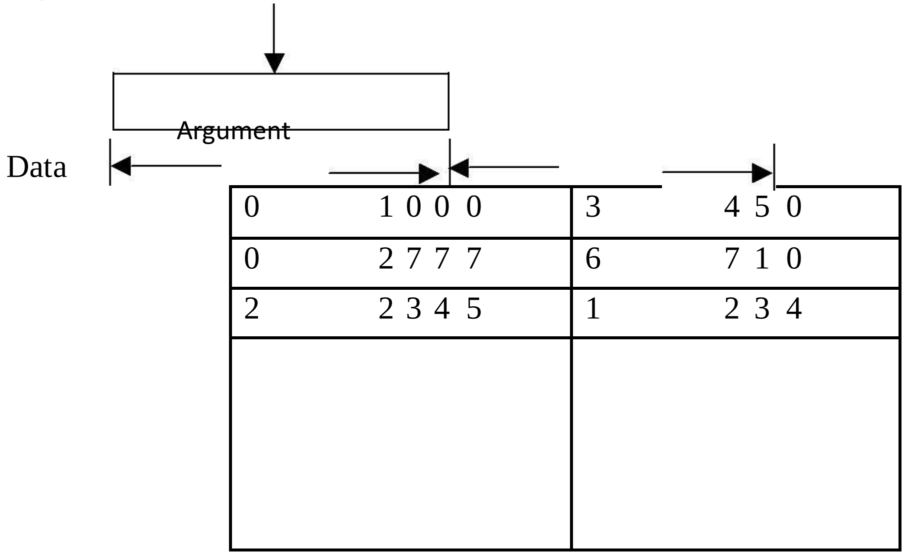


Figure - Example of cache memory

ASSOCIATIVE MAPPING

The fasters and most flexible cache organization use an associative memory. This organization is illustrated in below Fig. The associative memory stores both the address and content (data) of the memory word. This permits any location in cache to store any word from main memory. The diagram shows three words presently stored in the cache. The address value of 15 bits is shown as a five-digit octal number and its corresponding 12-bit word is shown as a four-digit octal number. A CPU address of 15 bits is placed in the argument register and the associative memory is searched for a matching address. If the address is found, the corresponding 12-bit data is read

Figure-Associative mapping cache (all numbers in octal) CPU address (15 bits)



0 1 0 0 0	3 4 5 0
0 2 7 7 7	6 7 1 0
2 2 3 4 5	1 2 3 4

And sent to the CPU. If no match occurs, the main memory is accessed for the word. The address-data pair is then transferred to the associative cache memory. If the cache is full, an address−data pair must be displaced to make room for a pair that is needed and not presently in the cache. The decision as to what pair is replaced is determined from the replacement algorithm that the designer chooses for the cache. A simple procedure is to replace cells of the cache in round-robin order whenever a new word is requested from main memory. This constitutes a first-in first-out (FIFO) replacement policy.

DIRECT MAPPING

Associative memories are expensive compared to random-access memories because of the added logic associated with each cell. The possibility of using a random-access memory for the cache is investigated in Fig. The CPU address of 15 bits is divided into two fields. The nine least significant bits constitute the index field and the remaining six bits from the tag and the index bits. The number of bits in the index field is equal to the number of address bits required to access the cache memory.

In the general case, there are 2^k words in cache memory and 2^n words in main memory. The n-bit

memory address is divided into two fields: k bits for the index field and n − k bits for the tag field. The direct mapping cache organization uses the n-bit address to access the main memory and the k-bit index to access the cache. The internal organization of the words in the cache memory is as shown in Fig. (b). Each word in cache consists of the data word and its associated tag. When a new word is first brought into the cache, the tag bits are stored alongside the data bits. When the CPU generates a memory request, the index field is used for the address to access the cache.

The tag field of the CPU address is compared with the tag in the word read from the cache. If the two tags match, there is a hit and the desired data word is in cache. If the two tags match, there is a hit and the desired data word is in cache. If there is no match, there is a miss and the required word is read from main memory. It is then stored in the cache together with the new tag, replacing the previous value. The disadvantage of direct mapping is that the hit ratio can droop considerably if two or more words whose addresses have the same index but different tags are accessed repeatedly. However, this possibility is minimized by the fact that such words are relatively far apart in the address range (multiples of 512 locations in this example).

To see how the direct-mapping organization operates, consider the numerical example shown in Fig. The word at address zero is presently stored in the cache (index = 000, tag = 00, data = 1220). Suppose that the CPU now wants to access the word at address 02000. The index address is 000, so it is sued to access the cache. The two tags are then compared. The cache tag is 00 but the address tag is 02, which does not produce a match. Therefore, the main memory is accessed and the data word 5670 is transferred to the CPU. The cache word at index address 000 is then replaced with a tag of 02 and data of 5670.

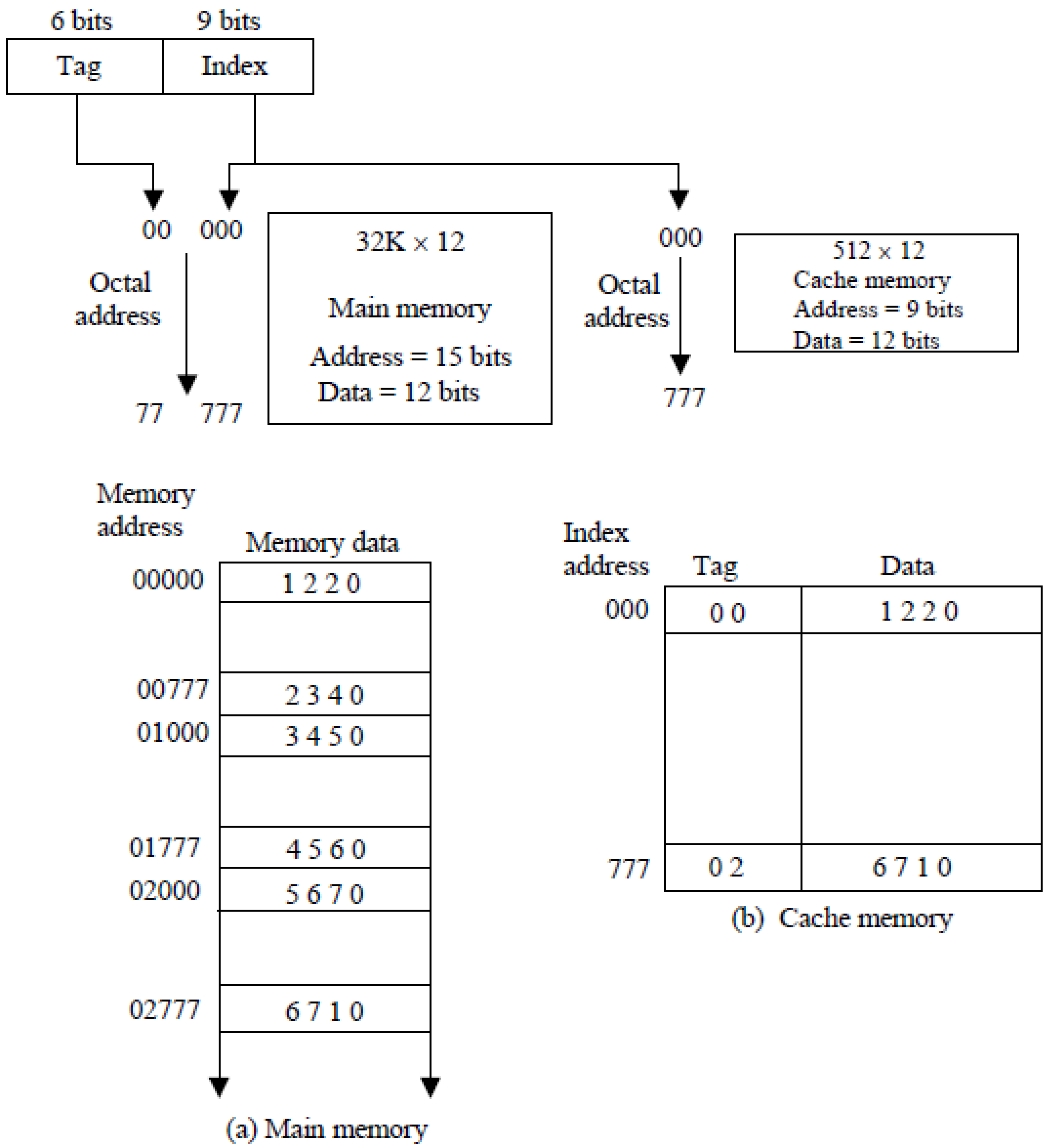


Fig-Direct mapping cache organization

The direct-mapping example just described uses a block size of one word. The same organization but using a block size of 8 words is shown in below Fig. The index Field is now divided into two parts: the block field and the word field. In a 512-word cache there are 64 block of 8 words each, since 64 × 8 = 512. The block number is specified with a 6-bit field and the word within the block is specified with a 3-bit field. The tag field stored within the cache is common to all eight words of the same block. Every time a miss occurs, an entire block of eight words must be transferred from main memory to cache memory. Although this takes extra time, the hit ratio will most likely improve with a larger block size because of the sequential nature of computer programs.

SET-ASSOCIATIVE MAPPING

It was mentioned previously that the disadvantage of direct mapping is that two words with the same index in their address but with different tag values cannot reside in cache memory at the same time. A third type of cache organization, called set-associative mapping, is an improvement over the direct-mapping organization in that each word of cache can store two or more words of memory under the same index address. Each data word is stored together with its tag and the number of tag-data items in one word of cache is said to form a set. An example of a set-associative cache organization for a set size of two is shown in Fig. Each index address refers to two data words and their associated tags. Each tag requires six bits and each data word has 12 bits, so the word length is 2(6 + 12) = 36 bits. An index address of nine bits can accommodate 512 words. Thus the size of cache memory is 512 × 36. It can accommodate 1024 words of main memory since each word of cache contains two data words. In general, a set-associative cache of set size k will accommodate k words of main memory in each word of cache.

Index Tag Data Tag Data 000

0 1	3 4 5 0	0 2	5 6 7 0
777	6 7 1 0	0 0	2 3 4 0

Figure- Two-way set-associative mapping cache.

The octal numbers listed in above Fig. are with reference to the main memory content illustrated in Fig.(a). The words stored at addresses 01000 and 02000 of main memory are stored in cache memory at index address 000. Similarly, the words at addresses 02777 and 00777 are stored in cache at index address 777. When the CPU generates a memory request, the index value of the address is used to access the cache. The tag field of the CPU address is then compared with both tags in the cache to determine if a catch occurs. The comparison logic is done by an associative search of the tags in the set similar to an associative memory search: thus the name “set-associative”. The hit ratio will improve as the set size increases because more words with the same index but different tag can reside in cache. However, an increase in the set size increases the number of bit s in words of cache and requires more complex comparison logic.

When a miss occurs in a set-associative cache and the set is full, it is necessary to replace one of the tag- data items with a new value. The most common replacement algorithms used are: random replacement, first-in, first out (FIFO), and least recently used (LRU). With the random replacement policy the control chooses one tag-data item for replacement at random. The FIFO procedure selects for replacement the item that has been in the set the longest. The LRU algorithm selects for replacement the item that has been least recently used by the CPU. Both FIFO and LRU can be implemented by adding a few extra bits in each word of cache.

WRITING INTO CACHE

An important aspect of cache organization is concerned with memory write requests. When the CPU finds a word in cache during read operation, the main memory is not involved in the transfer. However, if the operation is a write, there are two ways that the system can proceed.

The simplest and most commonly used procedure is to up data main memory with every memory write operation, with cache memory being updated in parallel if it contains the word at the specified address. This is called the write-through method. This method has the advantage that main memory always contains the same data as the cache,. This characteristic is important in systems with direct memory access transfers. It ensures that the data residing in main memory are valid at tall times so that an I/O device communicating through DMA would receive the most recent updated data.

The second procedure is called the write-back method. In this method only the cache location is updated during a write operation. The location is then marked by a flag so that later when the words are removed from the cache it is copied into main memory. The reason for the write-back method is that during the time a word resides in the cache, it may be updated several times; however, as long as the word remains in the cache, it does not matter whether the copy in main memory is out of date, since requests from the word are filled from the cache. It is only when the word is displaced from the cache that an accurate copy need be rewritten into main memory. Analytical results indicate that the number of memory writes in a typical program ranges between 10 and 30 percent of the total references to memory.

CACHE INITIALIZATION

One more aspect of cache organization that must be taken into consideration is the problem of initialization. The cache is initialized when power is applied to the computer or when the main memory is loaded with a complete set of programs from auxiliary memory. After initialization the cache is considered to be empty, built in effect it contains some non-valid data. It is customary to include with each word in cache a valid bit to indicate whether or not the word contains valid data.

The cache is initialized by clearing all the valid bits to 0. The valid bit of a particular cache word is set to 1 the first time this word is loaded from main memory and stays set unless the cache has to be initialized again. The introduction of the valid bit means that a word in cache is not replaced by another word unless the valid bit is set to 1 and a mismatch of tags occurs. If the valid bit happens to be 0, the new word automatically replaces the invalid data. Thus the initialization condition has the effect of forcing misses from the cache until it fills with valid data.

VIRTUAL MEMORY

In a memory hierarchy system, programs and data are brought into main memory as they are needed by the CPU. Virtual memory is a concept used in some large computer systems that permit the user to construct programs as though a large memory space were available, equal to the totality of auxiliary memory. Each address that is referenced by the CPU goes through an address mapping from the so-called virtual address to a physical address in main memory. Virtual memory is used to give programmers the illusion that they have a very large memory at their disposal, even though the computer actually has a relatively small main memory. A virtual memory system provides a mechanism for translating program-generated addresses into correct main memory locations. This is done dynamically, while programs are being executed in the CPU. The translation or mapping is handled automatically by the hardware by means of a mapping table.

ADDRESS SPACE AND MEMORY SPACE

An address used by a programmer will be called a virtual address, and the set of such addresses the address space. An address in main memory is called a location or physical address. The set of such locations is called the memory space. Thus the address space is the set of addresses generated by programs as they reference instructions and data; the memory space consists of the actual main memory locations directly addressable for processing. In most computers the address and memory spaces are identical. The address space is allowed to be larger than the memory space in computers with virtual memory.

As an illustration, consider a computer with a main -memory capacity of 32K words (K = 1024). Fifteen bits are needed to specify a physical address in memory since 32K = 2^{15}. Suppose that the computer has

available auxiliary memory for storing 2^{20} = 1024K words. Thus auxiliary memory has a capacity for storing information equivalent to the capacity of 32 main memories. Denoting the address space by N and the memory space by M, we then have for this example N = 1024K and M = 32K.

In a multiprogramming computer system, programs and data are transferred to and from auxiliary memory and main memory based on demands imposed by the CPU. Suppose that program 1 is currently being executed in the CPU. Program 1 and a portion of its associated data re moved from auxiliary memory into main memory as shown in Fig. Portions of programs and data need not be in contiguous locations in memory since information is being moved in and out, and empty spaces may be available in scattered locations in memory.

In a virtual memory system, programmers are told that they have the total address space at their disposal. Moreover, the address field of the instruction code has a sufficient number of bits to specify all virtual addresses. In our example, the address field of an instruction code will consist of 20 bits but physical memory addresses must be specified with only 15 bits. Thus CPU will reference instructions and data with a 20-bit address, but the information at this address must be taken from physical memory because access to auxiliary storage for individual words will be prohibitively long. (Remember

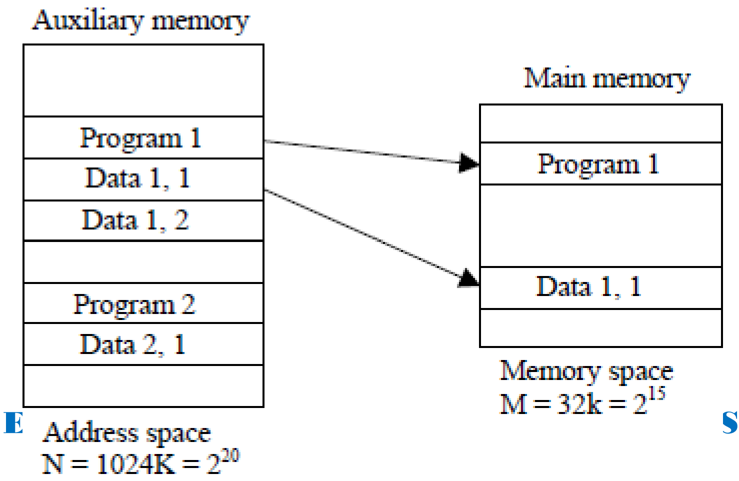
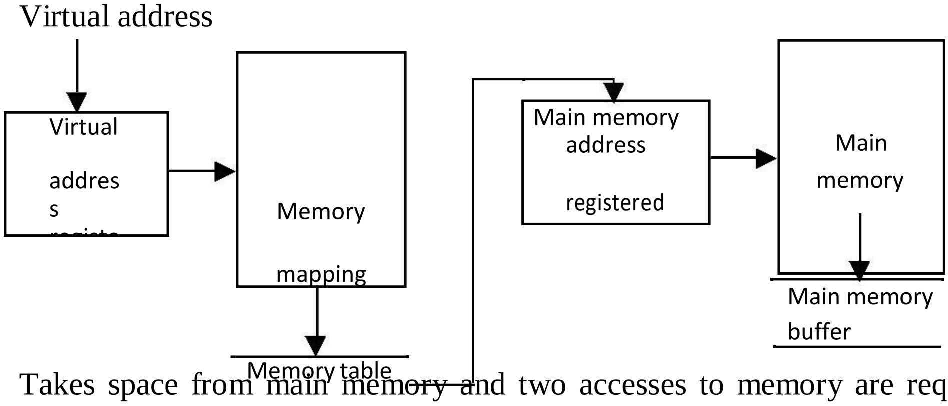


Fig-Relation between address and memory space in a virtual memory system

That for efficient transfers, auxiliary storage moves an entire record to the main memory). A table is then needed, as shown in Fig, to map a virtual address of 20 bits to a physical address of 15 bits. The mapping is a dynamic operation, which means that every address is translated immediately as a word is referenced by CPU.

The mapping table may be stored in a separate memory as shown in Fig. or in main memory. In the first case, an additional memory unit is required as well as one extra memory access time. In the second case, the table

Figure - Memory table for mapping a virtual address.



Takes space from main memory and two accesses to memory are required with the program running at half speed. A third alternative is to use an associative memory as explained below.

ADDRESS MAPPING USING PAGES

The table implementation of the address mapping is simplified if the information in the address space and the memory space are each divided into groups of fixed size. The physical memory is broken down into groups of equal size called blocks, which may range from 64 to 4096 words each. The term page refers to groups of address space of the same size. For example, if a page or block consists of 1K words, then, using the previous example, address space is divided into 1024 pages and main memory is divided into 32 blocks. Although both a page and a block are split into groups of 1K words, a page refers to the organization of address space, while a block refers to the organization of memory space. The programs are also considered to be split into pages. Portions of programs are moved from auxiliary memory to main memory in records equal to the size of a page. The term “page frame” is sometimes used to denote a block.

Consider a computer with an address space of 8K and a memory space of 4K. If we split each into groups of 1K words we obtain eight pages and four blocks as shown in Fig. At any given time, up to four pages of address space may reside in main memory in any one of the four blocks.

The mapping from address space to memory space is facilitated if each virtual address is considered to

be represented by two numbers: a page number address and a line within the page. In a computer with 2^p words per page, p bits are used to specify a line address and the remaining

high-order bits of the virtual address specify
the page number. In the example of Fig, a virtual address has 13 bits. Since each page consists of 2^{10} = 1024

words, the high-order three bits of a virtual address will specify one of the eight pages and the low-order 10 bits give the line address within the page. Note that the line address in address space and memory space is the same; the only mapping required is from a page number to a block number.

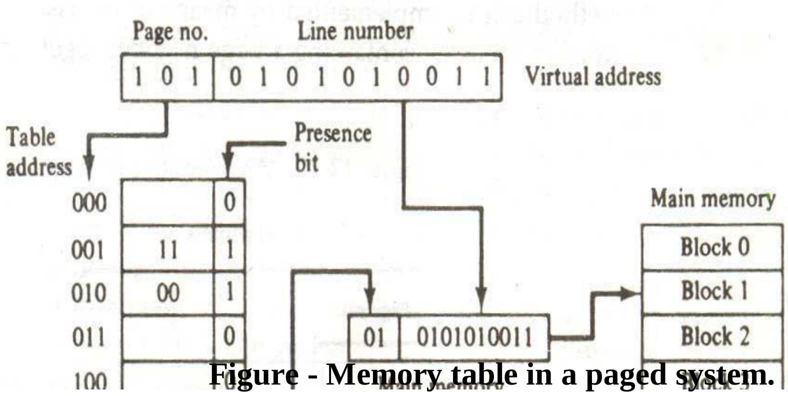


Figure - Memory table in a paged system.

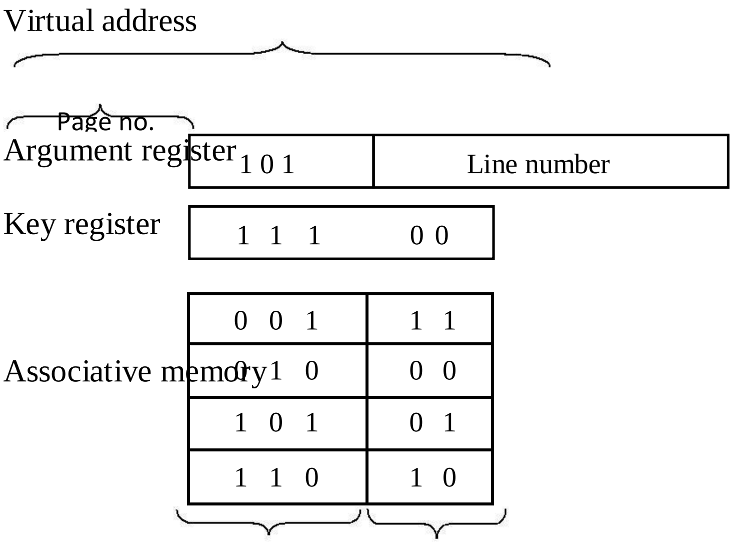
The word to the main memory buffer register ready to be used by the CPU. If the presence bit in the word read from the page table is 0, it signifies that the content of the word referenced by the virtual address does not reside in main memory. A call to the operating system is then generated to fetch the required page from auxiliary memory and place it into main memory before resuming computation.

ASSOCIATIVE MEMORY PAGE TABLE

A random-access memory page table is inefficient with respect to storage utilization. In the example of below Fig. we observe that eight words of memory are needed, one for each page, but at least four words will always be marked empty because main memory cannot accommodate more than four blocks. In general, system with n pages and m blocks would require a memory-page table of n locations of which up to m blocks will be marked with block numbers and all others will be empty. As a second numerical example, consider an address space of 1024K words and memory space of 32K words. If each page or block contains 1K words, the number of pages is 1024 and the number of blocks 32. The capacity of the memory-page table must be 1024 words and only 32 locations may have a presence bit equal to 1. At any given time, at least 992 locations will be empty and not in use.

A more efficient way to organize the page table would be to construct it with a number of words equal to the number of blocks in main memory. In this way the size of the memory is reduced and each location is fully utilized. This method can be implemented by means of an associative memory with each word in memory containing a page number together with its corresponding block number The page field in each word is compared with the page number in the virtual address. If a match occurs, the word is read from memory and its corresponding block number is extracted.

Figure -An associative memory page table.



Page no. Block no .Consider again the case of eight pages and four blocks as in the example of Fig. We replace the random access memory-page table with an associative memory of four words as shown in Fig. Each entry in the associative memory array consists of two fields. The first three bits specify a field from storing the page number. The last two bits constitute a field for storing the block number. The virtual address is placed in the argument register. The page number bits in the argument are compared with all page numbers in the page field of the associative memory. If the page number is found, the 5-bit word is read out from memory. The corresponding block number, being in the same word, is transferred to the main memory address register. If no match occurs, a call to the operating system is generated to bring the required page from auxiliary memory.

PAGE REPLACEMENT

A virtual memory system is a combination of hardware and software techniques. The memory management software system handles all the software operations for the efficient utilization of memory space. It must decide (1) which page in main memory ought to be removed to make room for a new page, (2) when a new page is to be transferred from auxiliary memory to main memory, and (3) where the page is to be placed in main memory. The hardware mapping mechanism and the memory management software together constitute the architecture of a virtual memory.

When a program starts execution, one or more pages are transferred into main memory and the page table is set to indicate their position. The program is executed from main memory until it attempts to reference a page that is still in auxiliary memory. This condition is called page fault. When page fault occurs, the execution of the present program is suspended until the required page is brought into main memory. Since loading a page from auxiliary memory to main memory is basically an I/O operation, the operating system assigns this task to the I/O processor. In the meantime, controls transferred to the next program in memory that is waiting to be processed in the CPU. Later, when the memory block has been assigned and the transfer completed, the original program can resume its operation.

When a page fault occurs in a virtual memory system, it signifies that the page referenced by the CPU is not in main memory. A new page is then transferred from auxiliary memory to main memory. If main memory is full, it would be necessary to remove a page from a memory block to make room for the new page. The policy for choosing pages to remove is determined from the replacement algorithm that is used. The goal of a replacement policy is to try to remove the page least likely to be referenced in the immediate future.

Two of the most common replacement algorithms used are the first-in first-out (FIFO) and the least recently used (LRU). The FIFO algorithm selects for replacement the page the has been in memory the longest time. Each time a page is loaded into memory, its identification number is pushed into a FIFO stack. FIFO will be full whenever memory has no more empty blocks. When a new page must be loaded, the page least recently brought in is removed. The page to be removed is easily determined because its identification number is at the top of the FIFO stack. The FIFO replacement policy has the advantage of being easy to implement. It has the disadvantages that under certain circum-stances pages are removed and loaded form memory too frequently.

The LRU policy is more difficult to implement but has been more attractive on the assumption that the least recently used page is a better candidate for removal than the least recently loaded pages in FIFO. The LRU algorithm can be implemented by associating a counter with every page that is in main memory. When a page is referenced, its associated counter is set to zero. At fixed intervals of time, the counters associated with all pages presently in memory are incremented by 1. The least recently used page is the page with the highest count. The counters are often called aging registers, as their count indicates their age, that is, how long ago their associated pages have been reference.

UNIT-IV

8086 CPU Pin Diagram: Special functions of general purpose registers, Segment register, concept of pipelining, 8086 Flag register, Addressing modes of 8086.

Introduction to processor:

A processor is the logic circuitry that responds to and processes the basic instructions that drive a computer.

The term processor has generally replaced the term central processing unit (CPU). The processor in a personal computer or embedded in small devices is often called a microprocessor.

The **processor** (**CPU**, for Central Processing Unit) is the computer's brain. It allows the processing of numeric data, meaning information entered in binary form, and the execution of instructions stored in memory.

Evolution of Microprocessor:

A microprocessor is used as the CPU in a microcomputer. There are now many different microprocessors available.

Microprocessor is a program-controlled device, which fetches the instructions from memory, decodes and executes the instructions. Most Micro Processor are single- chip devices.

Microprocessor is a backbone of computer system. which is called CPU

Microprocessor speed depends on the processing speed depends on DATA BUS WIDTH.

A common way of categorizing microprocessors is by the no. of bits that their ALU can
Work with at a time

The address bus is unidirectional because the address information is always given by the Micro Processor to address a memory location of an input / output devices.

The data bus is Bi-directional because the same bus is used for transfer of data between Micro Processor and memory or input / output devices in both the direction.

It has limitations on the size of data. Most Microprocessor does not support floating- point operations.

Microprocessor contain ROM chip because it contain instructions to execute data.

What is the primary & secondary storage device? - In primary storage device the
Storage capacity is limited. It has a volatile memory. In secondary storage device the storage capacity is larger. It is a nonvolatile memory.

Primary devices are: RAM (Read / Write memory, High Speed, Volatile Memory) / ROM (Read only memory, Low Speed, Non Voliate Memory)

Secondary devices are: Floppy disc / Hard disk

Compiler:

Compiler is used to translate the high-level language program into machine code at a time. It doesn’t require special instruction to store in a memory, it stores automatically. The Execution time is less compared to Interpreter.

1.4-bit Microprocessor:

The first **microprocessor** (Intel 4004) was invented in 1971. It was a 4-bit calculation device with a speed of 108 kHz. Since then, microprocessor power has grown exponentially. So what exactly are these little pieces of silicone that run our computers(" Common Operating Machine Particularly Used For Trade Education And Research ")

- It has 3200 PMOS transistors.
- It is a 4-bit device used in calculator.

2.8-Bit microprocessor:

In 1972, Intel came out with the 8008 which is 8-bit.

In 1974, Intel announced the 8080 followed by 8085 is a 8-bit processor Because 8085 processor has 8 bit ALU (Arithmetic Logic Review). Similarly 8086 processor has 16 bit ALU. This had a larger instruction set then 8080. used NMOS transistors, so it operated much faster than the 8008.

The 8080 is referred to as a “Second generation Microprocessor”

Limitations of 8 Bit microprocessor:

- Low speed of execution
- Low memory addressing capability
- Limited number of general purpose registers
- Less power full instruction set

Examples for 4/ 8 / 16 / 32 bit Microprocessors:

4-Bit processor – 4004/4040

8-bit Processor - 8085 / Z80 / 6800

c) 16-bit Processor - 8086 / 68000 / Z8000

d) 32-bit Processor - 80386 / 80486

What are 1st / 2nd / 3rd / 4th generation processor?

1. The processor made of PMOS technology is called 1st generation processor, and it is made up of 4 bits
2. The processor made of NMOS technology is called 2nd generation processor, and
3. it is made up of 8 bits
4. The processor made of CMOS technology is called 3rd generation processor, and it is made up of 16 bits
5. The processor made of HCMOS technology is called 4th generation processor,
6. and it is made up of 32 bits (**HCMOS** : High-density n- type Complementary Metal Oxide Silicon field effect transistor)

8086 OVERVIEW

8086 Microprocessor is an enhanced version of 8085Microprocessor that was designed by Intel in 1976. It is a 16-bit Microprocessor having 20 address lines and16 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, i.e. 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.

The most prominent features of a 8086 microprocessor are as follows −

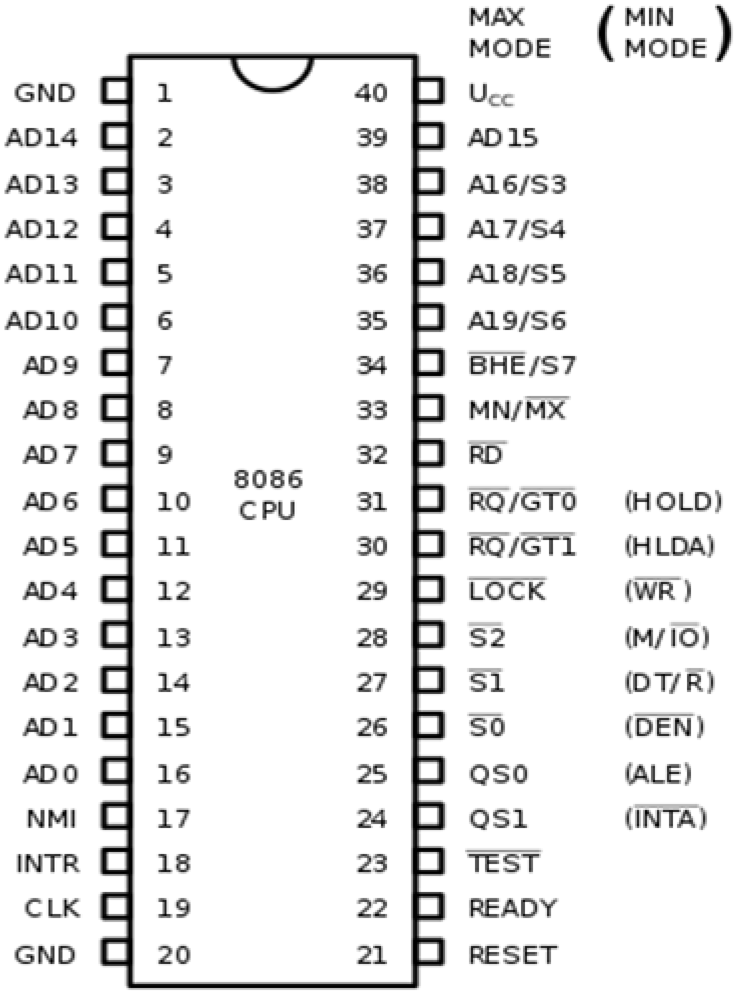
- It has an instruction queue, which is capable of storing six instruction bytes from the memory resulting in faster processing.
- It was the first 16-bit processor having 16-bit ALU, 16-bit registers, internal data bus, and 16-bit external data bus resulting in faster processing.
- It is available in 3 versions based on the frequency of operation −
  - 8086 → 5MHz
  - 8086-2 → 8MHz
  - (c)8086-1 → 10 MHz
- It uses two stages of pipelining, i.e. Fetch Stage and Execute Stage, which improves performance.
- Fetch stage can pre fetch up to 6 bytes of instructions and stores them in the queue.
- Execute stage executes these instructions.
- It has 256 vectored interrupts.
- It consists of 29,000 transistors.

Difference between 8085 and 8086 Microprocessor

085 Microprocessor	**086 Microprocessor**
is an 8-bit microprocessor.	is a 16-bit microprocessor.
has a 16-bit address line.	has a 20-bit address line.
has a 8-bit data bus.	has a 16-bit data bus.
he memory capacity is 64 KB.	he memory capacity is 1 MB.
he Clock speed of this icroprocessor is 3 MHz.	he Clock speed of this microprocessor varies between 5, 8 d 10 MHz for different versions.
has five flags.	has nine flags.
085 microprocessor does not support emory segmentation.	086 microprocessor supports memory segmentation.
does not support pipelining.	supports pipelining.
is accumulator based processor.	is general purpose register based processor.
has no minimum or maximum mode.	has minimum and maximum modes.

ı 8085, only one processor is used.	ı 8086, more than one processor is used. An additional xternal processor can also be employed.
contains less number of transistors ompare to 8086 microprocessor. It ontains about 6500 transistor.	contains more number of transistors compare to 8085 icroprocessor. It contains about 29000 in size.
he cost of 8085 is low.	he cost of 8086 is high.

8086 CPU Pin Diagram



Power supply and frequency signals

It uses 5V DC supply at V_{CC} pin 40, and uses ground at V_{SS} pin 1 and 20 for its operation.

Clock signal

Clock signal is provided through Pin-19. It provides timing to the processor for operations. Its frequency is different for different versions, i.e. 5MHz, 8MHz and 10MHz.

Address/data bus

AD0-AD15. These are 16 address/data bus. AD0-AD7 carries low order byte data and AD8AD15 carries higher order byte data. During the first clock cycle, it carries 16-bit address and after that it carries 16-bit data.

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.

S7/BHE

BHE stands for Bus High Enable. It is available at pin 34 and used to indicate the transfer of data using data bus D8-D15. This signal is low during the first clock cycle, thereafter it is active.

Read(*RD*)

It is available at pin 32 and is used to read signal for Read operation.

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.

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.

INTR

It is available at pin 18. It is an interrupt request signal, which is sampled during the last clock cycle of each instruction to determine if the processor considered this as an interrupt or not.

NMI

It stands for non-maskable interrupt and is available at pin 17. It is an edge triggered input, which causes an interrupt request to the microprocessor.

TEST

This signal is like wait state and is available at pin 23. When this signal is high, then the processor has to wait for IDLE state, else the execution continues.

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-aversa.

INTA

It is an interrupt acknowledgement signal and id available at pin 24. When the microprocessor receives this signal, it acknowledges the interrupt.

ALE

It stands for address enable latch and is available at pin 25. A positive pulse is generated each time the processor begins any operation. This signal indicates the availability of a valid address on the address/data lines.

DEN

It stands for Data Enable and is available at pin 26. It is used to enable Transreceiver 8286. The transreceiver is a device used to separate data from the address/data bus.

DT/R

It stands for Data Transmit/Receive signal and is available at pin 27. It decides the direction of data flow through the transreceiver. When it is high, data is transmitted out and vice-a-versa.

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 indicates the memory operation. It is available at pin 28.

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.

HLDA

It stands for Hold Acknowledgement signal and is available at pin 30. This signal acknowledges the HOLD signal.

HOLD

This signal indicates to the processor that external devices are requesting to access the address/data buses. It is available at pin 31.

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

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 & 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

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.

RQ/GT$_1$ and RQ/GT$_0$

These are the Request/Grant signals used by the other processors requesting the CPU to release the system bus. When the signal is received by CPU, then it sends acknowledgment. RQ/GT$_0$ has a higher priority than RQ/GT$_1$.

Operating Modes of 8086

There are two operating modes of operation for Intel 8086, namely the **minimum mode** and the **maximum mode**.

When only one 8086 CPU is to be used in a microprocessor system, the 8086 is used in the **Minimum mode** of operation.

In a multiprocessor system 8086 operates in the **Maximum mode**.

Pin Description for Minimum Mode

In this minimum mode of operation, the pin MN/$\boxed{\text{MX}}$ is connected to 5V D.C. supply i.e. MN/$\boxed{\text{MX}}$ = VCC.

The description about the pins from 24 to 31 for the minimum mode is as follows:

$\boxed{\text{INTA}}$ (Output): Pin number 24 interrupts acknowledgement. On receiving interrupt signal, the processor issues an interrupt acknowledgment signal. It is active LOW.

ALE (Output): Pin no. 25. Address latch enable. It goes HIGH during T1. The microprocessor 8086 sends this signal to latch the address into the Intel 8282/8283 latch.

$\boxed{\text{DEN}}$ (Output): Pin no. 26. Data Enable. When Intel 8287/8286 octal bus transceiver is used this signal. It is active LOW.

DT/$\boxed{\text{R}}$ (output): Pin No. 27 data Transmit/Receives. When Intel 8287/8286 octal bus transceiver is used this signal controls the direction of data flow through the transceiver. When it is HIGH, data is sent out. When it is LOW, data is received.
M/$\boxed{\text{IO}}$ (Output): Pin no. 28, Memory or I/O access. When this signal is HIGH, the CPU wants to access memory. When this signal is LOW, the CPU wants to access I/O device.
$\boxed{\text{WR}}$ (Output): Pin no. 29, Write. When this signal is LOW, the CPU performs memory or I/O write operation.
HLDA (Output): Pin no. 30, Hold Acknowledgment. It is sent by the processor when it receives HOLD signal. It is active HIGH signal. When HOLD is removed HLDA goes LOW.
HOLD (Input): Pin no. 31, Hold. When another device in microcomputer system wants to use the address and data bus, it sends HOLD request to CPU through this pin. It is an active HIGH signal.

Pin Description for Maximum Mode

In the maximum mode of operation, the pin MN/¯MX is made LOW. It is grounded. The description about the pins from 24 to 31 is as follows:
QS1, QS0 (Output): Pin numbers 24, 25, Instruction Queue Status. Logics are given below:

QS1	QS0	Operation
0 0	00	No operation
0 0	11	1st byte of opcode from queue.
11	00	Empty the queue
11	11	Subsequent byte from queue

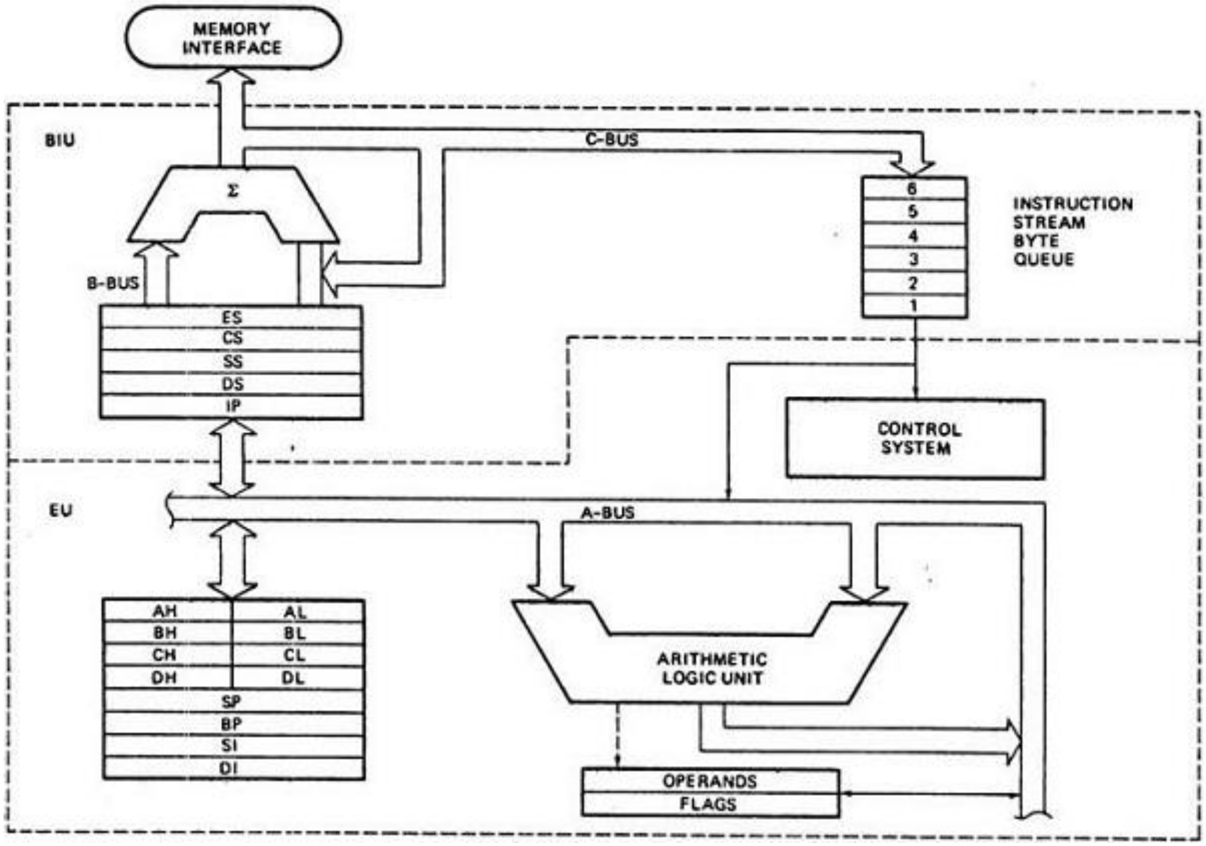
$\boxed{\text{S0}}$, $\boxed{\text{S1}}$, $\boxed{\text{S2}}$ (Output): Pin numbers 26, 27, 28 Status Signals. These signals are connected to the bus controller of Intel 8288. This bus controller generates memory and I/O access control signals. Logics for status signal are given below:

$\boxed{\text{S2}}$	$\boxed{\text{S1}}$	$\boxed{\text{S0}}$	Operation
00	00	00	Interrupt acknowledgement

00	00	11	Read data from I/O port
00	11	00	Write data from I/O port
00	11	11	Halt
11	00	00	Opcode fetch
11	00	11	Memory read
11	11	00	Memory write
1	1	1	Passive state

LOCK (Output): Pin no. 29. It is an active LOW signal. When this signal is LOW, all interrupts are masked and no HOLD request is granted. In a multiprocessor system all other processors are informed through this signal that they should not ask the CPU for relinquishing the bus control.
RG/GT1, RQ/GT0 (Bidirectional): Pin numbers 30, 31, Local Bus Priority Control. Other processors ask the CPU by these lines to release the local bus.
In the maximum mode of operation signals WR, ALE, DEN, DT/R etc. are not available directly from the processor. These signals are available from the controller 8288.

Architecture of 8086



8086 Microprocessor is divided into two functional units, i.e., **EU** (Execution Unit) and **BIU** (Bus Interface Unit).

EU (Execution Unit)

Execution unit gives instructions to BIU stating from where to fetch the data and then decode and execute those instructions. Its function is to control operations on data using the instruction decoder & ALU. EU has no direct connection with system buses as shown in the above figure, it performs operations over data through BIU.
Let us now discuss the functional parts of 8086 microprocessors.

ALU

It handles all arithmetic and logical operations, like +, −, ×, /, OR, AND, NOT operations.

Flag Register

It is a 16-bit register that behaves like a flip-flop, i.e. it changes its status according to the result stored in the accumulator. It has 9 flags and they are divided into 2 groups − Conditional Flags and Control Flags.

Conditional Flags

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

- **Carry flag −** This flag indicates an overflow condition for arithmetic operations.
- **Auxiliary flag −** When an operation is performed at ALU, it results in a carry/barrow from lower nibble (i.e. D0 – D3) to upper nibble (i.e. D4 – D7), then this flag is set, i.e. carry given by D3 bit to D4 is AF flag. The processor uses this flag to perform binary to BCD conversion.
- **Parity flag −** This flag is used to indicate the parity of the result, i.e. when the lower order 8-bits of the result contains even number of 1's, then the Parity Flag is set. For odd number of 1's, the Parity Flag is reset.
- **Zero flag −** This flag is set to 1 when the result of arithmetic or logical operation is zero else it is set to 0.
- **Sign flag −** 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.
- **Overflow flag −** This flag represents the result when the system capacity is exceeded.

Control Flags

Control flags controls the operations of the execution unit. Following is the list of control flags −

- **Trap flag −** 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.
- **Interrupt flag −** It is an interrupt enable/disable flag, i.e. used to allow/prohibit the interruption of a program. It is set to 1 for interrupt enabled condition and set to 0 for interrupt disabled condition.
- **Direction flag −** It is used in string operation. As the name suggests when it is set then string bytes are accessed from the higher memory address to the lower memory address and vice-a-versa.

General purpose register

There are 8 general purpose registers, i.e., AH, AL, BH, BL, CH, CL, DH, and DL. These registers can be used individually to store 8-bit data and can be used in pairs to store 16bit data. The valid register pairs are AH and AL, BH and BL, CH and CL, and DH and DL. It is referred to the AX, BX, CX, and DX respectively.

- **AX register −** It is also known as accumulator register. It is used to store operands for arithmetic operations.
- **BX register −** It is used as a base register. It is used to store the starting base address of the memory area within the data segment.
- **CX register −** It is referred to as counter. It is used in loop instruction to store the loop counter.
- **DX register −** This register is used to hold I/O port address for I/O instruction.

Stack pointer register

It is a 16-bit register, which holds the address from the start of the segment to the memory location, where a word was most recently stored on the stack.

BIU (Bus Interface Unit)

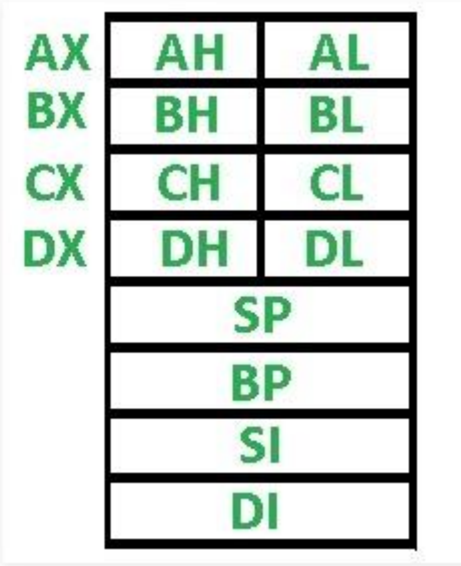
BIU takes care of all data and addresses transfers on the buses for the EU like sending addresses, fetching instructions from the memory, reading data from the ports and the memory as well as writing data to the ports and the memory. EU has no direction connection with System Buses so this is possible with the BIU. EU and BIU are connected with the Internal Bus.

It has the following functional parts −

- **Instruction queue −** BIU contains the instruction queue. BIU gets up to 6 bytes of next instructions and stores them in the instruction queue. When EU executes instructions and is ready for its next instruction, then it simply reads the instruction from this instruction queue resulting in increased execution speed.
- Fetching the next instruction while the current instruction executes is called **pipelining**.
- **Segment register −** BIU has 4 segment buses, i.e. CS, DS, SS& ES. It holds the addresses of instructions and data in memory, which are used by the processor to access memory locations. It also contains 1 pointer register IP, which holds the address of the next instruction to executed by the EU.
  - **CS −** It stands for Code Segment. It is used for addressing a memory location in the code segment of the memory, where the executable program is stored.
  - **DS −** It stands for Data Segment. It consists of data used by the program and is accessed in the data segment by an offset address or the content of other register that holds the offset address.
  - **SS −** It stands for Stack Segment. It handles memory to store data and addresses during execution.
  - **ES −** It stands for Extra Segment. ES is additional data segment, which is used by the string to hold the extra destination data.
- **Instruction pointer −** It is a 16-bit register used to hold the address of the next instruction to be executed.

Special functions of general purpose registers

he general purpose registers are used to store temporary data in the time of different operations in microprocessor. 8086 has eight general purpose registers.



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

The description of these general purpose registers

Register	Function
AX	This is the accumulator. It is 16-bit registers, but it is divided into two 8-bit registers. These registers are AH and AL. AX generally used for arithmetic or logical instructions, but it is not mandatory in 8086.
BX	BX is another register pair consisting of BH and BL. This register is used to store the offset values.
CX	CX is generally used as control register. It has two parts CH and CL. For different looping and counting purposes these are used.
DX	DX is data register. The two parts are DH and DL. This register can be used in Multiplication, Input/output addressing etc.
SP	This is the stack pointer. The stack pointer points the top most element of the stack. For empty stack SP will be at position FFFEH.
BP	BP is another 16-bit register. This is base pointer register. This register is primary used in accessing the parameters passed by the stack. It’s offset address relatives to stack segment.
SI	This is Source Index register. This is used to point the source in some string related operations. Its offset is relative to data segment.
DI	This is destination index register. This is used to point destination in some string related operations. Its offset is relative to extra segment.

Segmentation is the process in which the main memory of the computer is divided into different segments and each segment has its own base address. It is basically used to enhance the speed of execution of the computer system, so that processor is able to fetch and execute the data from the memory easily and fast.

Need for Segmentation –

The Bus Interface Unit (BIU) contains four 16 bit special purpose registers (mentioned below) called as Segment Registers.

Code segment register (CS): is used for addressing memory location in the code segment of the memory, where the executable program is stored.

Data segment register (DS): points to the data segment of the memory where the data is stored.

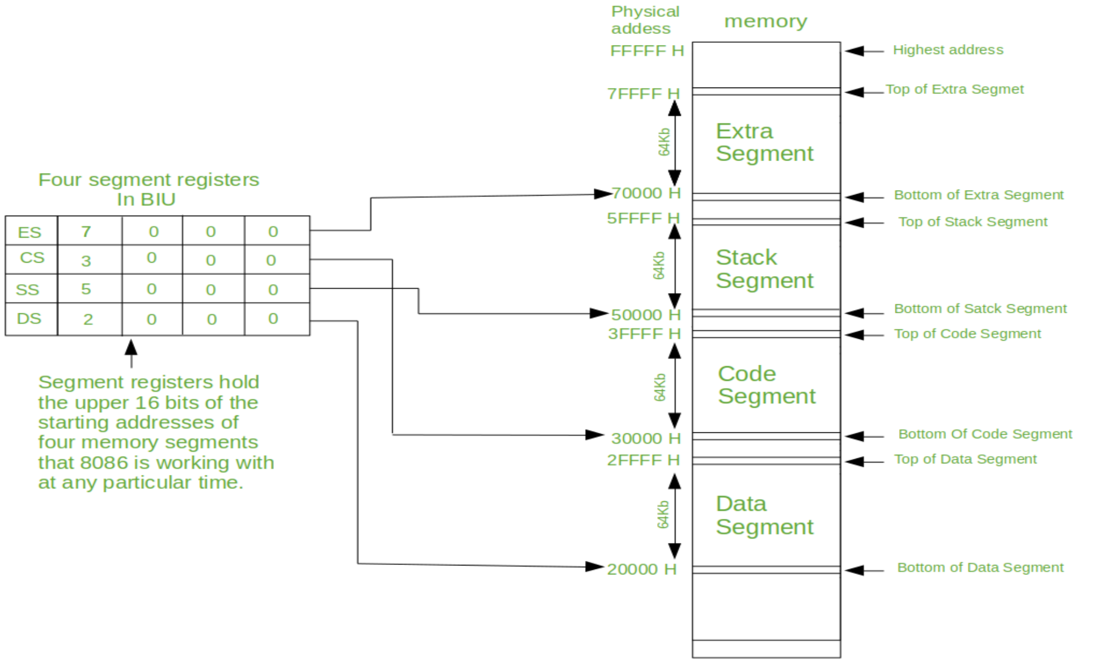
Extra Segment Register (ES): also refers to a segment in the memory which is another data segment in the memory.

Stack Segment Register (SS): is used for addressing stack segment of the memory. The stack segment is that segment of memory which is used to store stack data.

The number of address lines in 8086 is 20, 8086 BIU will send 20bit address, so as to access one of the 1MB memory locations. The four segment registers actually contain the upper 16 bits of the starting addresses of the four memory segments of 64 KB each with which the 8086 is working at that instant of time. A segment is a logical unit of memory that may be up to 64 kilobytes long. Each segment is made up of contiguous memory locations. It is independent, separately addressable unit. Starting address will always be changing. It will not be fixed.

Note that the 8086 does not work the whole 1MB memory at any given time. However it works only with four 64KB segments within the whole 1MB memor

Bellow is the one way of positioning four 64 kilobyte segments within the 1M byte memory space of an 8086.



ES	7	0	0	0
CS	3	0	0	0
SS	5	0	0	0
DS	2	0	0	0

Segment registers hold the upper 16 bits of the starting addresses of four memory segments that 8086 is working with at any particular time.

Types Of Segmentation –

Overlapping Segment – A segment starts at a particular address and its maximum size can go up to 64kilobytes. But if another segment starts along this 64kilobytes location of the first segment, then the two are said to be *Overlapping Segment*.

Non-Overlapped Segment – A segment starts at a particular address and its maximum size can go up to 64kilobytes. But if another segment starts before this 64kilobytes location of the first segment, then the two segments are said to be *Non-Overlapped Segment*.

Advantages of the Segmentation The main advantages of segmentation are as follows:

It provides a powerful memory management mechanism.

Data related or stack related operations can be performed in different segments.

Code related operation can be done in separate code segments.

It allows to processes to easily share data.

It allows to extend the address ability of the processor, i.e. segmentation allows the use of 16 bit registers to give an addressing capability of 1 Megabytes. Without segmentation, it would require 20 bit registers.

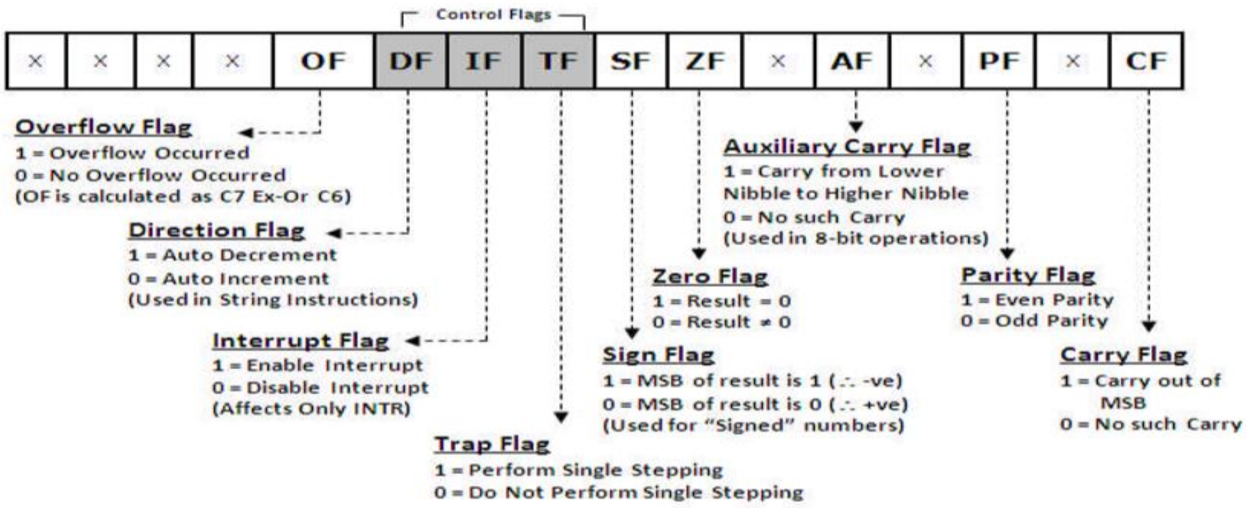
It is possible to enhance the memory size of code data or stack segments beyond 64 KB by allotting more than one segment for each area.

concept of pipelining

The process of fetching the next instruction when the present instruction is being executed is called as pipelining. Pipelining has become possible due to the use of queue.BIU (Bus Interfacing Unit) fills in the queue until the entire queue is full.BIU restarts filling in the queue when at least two locations of queue are vacant. Advantages of pipelining: The execution unit always reads the next instruction byte from the queue in BIU. This is faster than sending out an address to the memory and waiting for the next instruction byte to come. In short pipelining eliminates the waiting time of EU and speeds up the processing. -The 8086 BIU will not initiate a fetch unless and until there are two empty bytes in its queue. 8086 BIU normally obtains two instruction bytes per fetch.

8086 Flag register

The Flag register is a Special Purpose Register. Depending upon the value of result after any arithmetic and logical operation the flag bits become set (1) or reset (0)



We can divide the flag bits into two sections. The Status Flags, and the Control Flags.

Status Flags

In 8086 there are 6 different flags which are set or reset after 8-bit or 16-bit operations. These flags and their functions are listed below.

Flag Bit	Function

Flag Bit	Function
S	After any operation if the MSB is 1, then it indicates that the number is negative. And this flag is set to 1
Z	If the total register is zero, then only the Z flag is set
AC	When some arithmetic operations generates carry after the lower half and sends it to upper half, the AC will be 1
P	This is even parity flag. When result has even number of 1, it will be set to 1, otherwise 0 for odd number of 1s
CY	This is carry bit. If some operations are generating carry after the operation this flag is set to 1
O	The overflow flag is set to 1 when the result of a signed operation is too large to fit.

Control Flags

In 8086 there are 3 different flags which are used to enable or disable some basic operations of the microprocessor. These flags and their functions are listed below.

Flag Bit	Function
D	This is directional flag. This is used in string related operations. D = 1, then the string will be accessed from higher memory address to lower memory address, and if D = 0, it will do the reverse.
I	This is interrupt flag. If I = 1, then MPU will recognize the interrupts from peripherals. For I = 0, the interrupts will be ignored
T	This trap flag is used for on-chip debugging. When T = 1, it will work in a single step mode. After each instruction, one internal interrupt is generated. It helps to execute some program instruction by instruction.

Addressing Modes.

The way of specifying data to be operated by an instruction is known as **addressing modes**. This specifies that the given data is an immediate data or an address. It also specifies whether the given operand is register or register pair.

Types of addressing modes:

Register mode – In this type of addressing mode both the operands are registers.

Example:

```
MOV AX, BX
XOR AX, DX
ADD AL, BL
```

Immediate mode – In this type of addressing mode the source operand is a 8 bit or 16 bit data. Destination operand can never be immediate data.

Example:

```
MOV AX, 2000
MOV CL, 0A
ADD AL, 45
AND AX, 0000
```

Displacement or direct mode – In this type of addressing mode the effective address is directly given in the instruction as displacement.

Example:

```
MOV AX, [DISP]
MOV AX, [0500]
```

Register indirect mode – In this addressing mode the effective address is in SI, DI or BX.

Example:

```
MOV AX, [DI]
ADD AL, [BX]
MOV AX, [SI]
```

Based indexed mode – In this the effective address is sum of base register and index register.

```
Base register: BX, BP
Index register: SI, DI
```

The physical memory address is calculated according to the base register. Example:

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

Indexed mode – In this type of addressing mode the effective address is sum of index register and displacement.

Example:

```
MOV AX, [SI+2000]
MOV AL, [DI+3000]
```

Based mode – In this the effective address is the sum of base register and displacement.

Example:

```
MOV AL, [BP+ 0100]
```

Based indexed displacement mode – In this type of addressing mode the effective address is the sum of index register, base register and displacement.

Example:

```
MOV AL, [SI+BP+2000]
```

String mode – This addressing mode is related to string instructions. In this the value of SI and DI are auto incremented and decremented depending upon the value of directional flag.

Example:

```
MOVS B
MOVS W
```

Input/Output mode – This addressing mode is related with input output operations.

Example:

```
IN A, 45
OUT A, 50
```

Relative mode –

In this the effective address is calculated with reference to instruction pointer.

Example:

```
JNZ 8 bit address
IP=IP+8 bit address
```

UNIT-V

8086-Instruction formats: assembly Language Programs involving branch & Call instructions, sorting, evaluation of arithmetic expressions.

8086 INSTRUCTION SET

The 8086 microprocessor supports 8 types of instructions −

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

Let us now discuss these instruction sets in detail.

Data Transfer Instructions

These instructions are used to transfer the data from the source operand to the destination operand. Following are the list of instructions under this group −

Instruction to transfer a word

MOV − Used to copy the byte or word from the provided source to the provided destination.
PPUSH − Used to put a word at the top of the stack.
POP − Used to get a word from the top of the stack to the provided location.
PUSHA − Used to put all the registers into the stack.
POPA − Used to get words from the stack to all registers.
XCHG − Used to exchange the data from two locations.
XLAT − Used to translate a byte in AL using a table in the memory.

Instructions for input and output port transfer

IN − Used to read a byte or word from the provided port to the accumulator.
OUT − Used to send out a byte or word from the accumulator to the provided port.

Instructions to transfer the address

LEA − Used to load the address of operand into the provided register.
LDS − Used to load DS register and other provided register from the memory
LES − Used to load ES register and other provided register from the memory.

Instructions to transfer flag registers

LAHF − Used to load AH with the low byte of the flag register.
SAHF − Used to store AH register to low byte of the flag register.
PUSHF − Used to copy the flag register at the top of the stack.
POPF − Used to copy a word at the top of the stack to the flag register.

Arithmetic Instructions

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

Following is the list of instructions under this group −

Instructions to perform addition

ADD − Used to add the provided byte to byte/word to word.

ADC − Used to add with carry.

INC − Used to increment the provided byte/word by 1.

AAA − Used to adjust ASCII after addition.

DAA − Used to adjust the decimal after the addition/subtraction operation.

Instructions to perform subtraction

SUB − Used to subtract the byte from byte/word from word.

SBB − Used to perform subtraction with borrow.

DEC − Used to decrement the provided byte/word by 1.

NPG − Used to negate each bit of the provided byte/word and add 1/2's complement.

CMP − Used to compare 2 provided byte/word.

AAS − Used to adjust ASCII codes after subtraction.

DAS − Used to adjust decimal after subtraction.

Instruction to perform multiplication

MUL − Used to multiply unsigned byte by byte/word by word.

IMUL − Used to multiply signed byte by byte/word by word.

AAM − Used to adjust ASCII codes after multiplication.

Instructions to perform division

DIV − Used to divide the unsigned word by byte or unsigned double word by word.

IDIV − Used to divide the signed word by byte or signed double word by word.

AAD − Used to adjust ASCII codes after division.

CBW − Used to fill the upper byte of the word with the copies of sign bit of the lower byte.

CWD − Used to fill the upper word of the double word with the sign bit of the lower word.

Bit Manipulation Instructions

These instructions are used to perform operations where data bits are involved, i.e. operations like logical, shift, etc.

Following is the list of instructions under this group −

Instructions to perform logical operation

NOT − Used to invert each bit of a byte or word.

AND − Used for adding each bit in a byte/word with the corresponding bit in another byte/word.

OR − Used to multiply each bit in a byte/word with the corresponding bit in another byte/word.

XOR − Used to perform Exclusive-OR operation over each bit in a byte/word with the corresponding bit in another byte/word.

TEST − Used to add operands to update flags, without affecting operands.

Instructions to perform shift operations

SHL/SAL − Used to shift bits of a byte/word towards left and put zero(S) in LSBs.

SHR − Used to shift bits of a byte/word towards the right and put zero(S) in MSBs.

SAR − Used to shift bits of a byte/word towards the right and copy the old MSB into the new MSB.

Instructions to perform rotate operations

ROL − Used to rotate bits of byte/word towards the left, i.e. MSB to LSB and to Carry Flag

[CF].
ROR − Used to rotate bits of byte/word towards the right, i.e. LSB to MSB and to Carry Flag [CF].
RCR − Used to rotate bits of byte/word towards the right, i.e. LSB to CF and CF to MSB.
RCL − Used to rotate bits of byte/word towards the left, i.e. MSB to CF and CF to LSB.

String Instructions

String is a group of bytes/words and their memory is always allocated in a sequential order.
Following is the list of instructions under this group −
REP − Used to repeat the given instruction till CX ≠ 0.
REPE/REPZ − Used to repeat the given instruction until CX = 0 or zero flag ZF = 1.
REPNE/REPNZ − Used to repeat the given instruction until CX = 0 or zero flag ZF = 1.
MOVS/MOVSB/MOVSW − Used to move the byte/word from one string to another.
COMS/COMPSB/COMPSW − Used to compare two string bytes/words.
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.
SCAS/SCASB/SCASW − Used to scan a string and compare its byte with a byte in AL or string word with a word in AX.
LODS/LODSB/LODSW − Used to store the string byte into AL or string word into AX.

Program Execution Transfer Instructions (Branch and Loop Instructions)

These instructions are used to transfer/branch the instructions during an execution. It includes the following instructions −
Instructions to transfer the instruction during an execution without any condition −
CALL − Used to call a procedure and save their return address to the stack.
RET − Used to return from the procedure to the main program.
JMP − Used to jump to the provided address to proceed to the next instruction.
Instructions to transfer the instruction during an execution with some conditions −
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

Processor Control Instructions

These instructions are used to control the processor action by setting/resetting the flag values. Following are the instructions under this group −

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

Iteration Control 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
JCXZ − Used to jump to the provided address if 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

Evaluation Of Arithmetic Expressions

The stack organization is very effective in evaluating arithmetic expressions. Expressions are usually represented in what is known as **Infix notation**, in which each operator is written between two operands (i.e., A + B). With this notation, we must distinguish between (A + B)*C and A + (B * C) by using either parentheses or some operator-precedence convention. Thus, the order of operators and operands in an arithmetic expression does not uniquely determine the order in which the operations are to be performed.

Polish notation (prefix notation) –

It refers to the notation in which the operator is placed before its two operands . Here no parentheses are required, i.e.,
+AB

Reverse Polish notation(postfix notation) –

It refers to the analogous notation in which the operator is placed after its two operands. Again, no parentheses is required in Reverse Polish notation, i.e.,
Stack organized computers are better suited for post-fix notation then the traditional infix notation. Thus the infix notation must be converted to the post-fix notation. The conversion from infix notation to post-fix notation must take into consideration the operational hierarchy.

There are 3 levels of precedence for 5 binary operators as given below:
Highest: Exponentiation (^)
Next highest: Multiplication (*) and division (/)
Lowest: Addition (+) and Subtraction (-)

For example –

Infix notation: (A-B)*[C/(D+E)+F]

Post-fix notation: AB- CDE +/F +*

Here, we first perform the arithmetic inside the parentheses (A-B) and (D+E). The division of C/(D+E) must done prior to the addition with F. After that multiply the two terms inside the parentheses and bracket.

Now we need to calculate the value of these arithmetic operations by using stack.

The procedure for getting the result is:

Convert the expression in Reverse Polish notation(post-fix notation).

Push the operands into the stack in the order they are appear.

When any operator encounter then pop two topmost operands for executing the operation.

After execution push the result obtained into the stack.

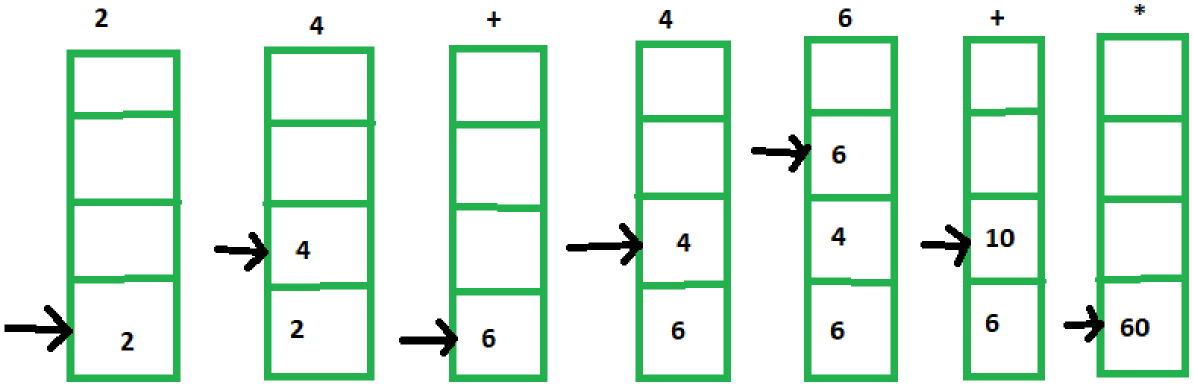
After the complete execution of expression the final result remains on the top of the stack.

For example –

Infix notation: (2+4) * (4+6)

Post-fix notation: 2 4 + 4 6 + *

Result: 60



Stack operations to evaluate (2+4)*(4+6)

EXTRA TOPICS

PROGRAMMER'S MODEL OF 8086

The programming model for a microprocessor shows the various internal registers that are accessible to the programmer.

The Following Figure is a model for the 8086. In general, each register has a special function.

In the programming model there are

1. 4 General Purpose registers(Data Registers)
2. 4 Segment registers
3. 2 Pointer registers
4. 2 Index registers
5. 1 Instruction Pointer register
6. 1 Flag register

General purpose registers:

AX Register (Accumulator): This is accumulator register. It gets used in arithmetic, logic and data transfer instructions. In manipulation and division, one of the numbers involved must be in AX or AL.

BX Register (Base Register): This is base register. BX register is an address register. It usually contain a data pointer used for based, based indexed or register indirect addressing.

CX Register (Counter register): This is Count register. This serves as a loop counter. Program loop constructions are facilitated by it. Count register can also be used as a counter in string manipulation and shift/rotate instruction.

DX Register (Data Register): This is data register. Data register can be used as a port number in I/O operations. It is also used in multiplication and division.

Segement Registers:

There are four segment registers in Intel 8086:

1. Code Segment Register (CS),
2. Data Segment Register (DS),
3. Stack Segment Register (SS),
4. Extra Segment Register (ES).

A segment register points to the starting address of a memory segment. Maximum capacity of a segment may be up to 64 KB.

Code segment Register(CS):- It is a 16-bit register containing the starting address of 64 KB segment. The processor uses CS segment for all accesses to instructions referenced by instruction pointer (IP) register.

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

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

Extra segment Register (ES):- It is a 16-bit register containing address of 64KB segment, usually with program data. By default, the processor assumes that the DI register references the ES segment in string manipulation instructions. It is possible to change default segments used by general and index registers by prefixing instructions with a CS, SS,DS or ES prefix.

Pointer Registers:

SP Register (Stack Pointer): This is stack pointer register pointing to program stack. It is used in conjunction with SS for accessing the stack segment.

BP Register (Base Pointer): This is base pointer register pointing to data in stack segment. Unlike

SP, we can use BP to access data in the other segments.

Index Registers:

SI Register (Source Index): This is used to point to memory locations in the data segment addressed by DS. By incrementing the contents of SI one can easily access consecutive memory locations.

DI Register (Destination Index): This register performs the same function as SI. There is a class of instructions called string operations, that use DI to access the memory locations addressed by ES.

Instruction Pointer: The Instruction Pointer (IP) points to the address of the next instruction to be executed. Its content is automatically incremented when the execution of a program proceeds further. The contents of the IP and Code Segment Register are used to compute the memory address of the instruction code to be fetched. This is done during the Fetch Cycle.

Flag Register: Status Flags determines the current state of the accumulator. They are modified automatically by CPU after mathematical operations. This allows to determine the type of the result. 8086 has 16-bit status register. It is also called Flag Register or Program Status Word (PSW). There are nine status flags and seven bit positions remain unused.

8086 has 16 flag registers among which 9 are active. The purpose of the FLAGS register is to indicate the status of the processor. It does this by setting the individual bits called flags. There are two kinds of FLAGS;

Status FLAGS and Control FLAGS. Status FLAGS reflect the result of an operation executed by the processor. The control FLAGS enable or disable certain operations of the processor.

8086 ASSEMBLER DIRECTIVES

SEGMENT

The SEGMENT directive is used to indicate the start of a logical segment. Preceding the SEGMENT directive is the name you want to give the segment. For example, the statement CODE SEGMENT indicates to the assembler the start of a logical segment called CODE. The SEGMENT and ENDS directive are used to “bracket” a logical segment containing code of data.

Additional terms are often added to a SEGMENT directive statement to indicate some special way in which we want the assembler to treat the segment. The statement CODE SEGMENT WORD tells the assembler that we want the content of this segment located on the next available word (even address) when segments ate combined and given absolute addresses. Without this WORD addition, the segment will be located on the next available paragraph (16-byte) address, which might waste as much as 15 bytes of memory. The statement CODE SEGMENT PUBLIC tells the assembler that the segment may be put together with other segments named CODE from other assembly modules when the modules are linked together.

ENDS (END SEGMENT)

This directive is used with the name of a segment to indicate the end of that logical segment.

 CODE SEGMENT Start of logical segment containing code instruction statements

CODE ENDS End of segment named CODE

END (END PROCEDURE)

The END directive is put after the last statement of a program to tell the assembler that this is the end of the program module. The assembler will ignore any statements after an END directive, so you should make sure to use only one END directive at the very end of your program module. A carriage return is required after the END directive.

ASSUME

The ASSUME directive is used tell the assembler the name of the logical segment it should use for a specified segment. The statement ASSUME CS: CODE, for example, tells the assembler that the instructions for a program are in a logical segment named CODE. The statement ASSUME DS: DATA tells the assembler that for any program instruction, which refers to the data segment, it should use the logical segment called DATA.

DB (DEFINE BYTE)

The DB directive is used to declare a byte type variable, or a set aside one or more storage locations of type byte in memory.

 PRICES DB 49H, 98H, 29H Declare array of 3 bytes named PRICE and initialize them

with specified values.

 NAMES DB “THOMAS” Declare array of 6 bytes and initialize with ASCII codes for the letters in THOMAS.

 TEMP DB 100 DUP (?) Set aside 100 bytes of storage in memory and give it the name TEMP. But leave the 100 bytes un-initialized.

 PRESSURE DB 20H DUP (0) Set aside 20H bytes of storage in memory, give it the name PRESSURE and put 0 in all 20H locations.

DD (DEFINE DOUBLE WORD)

The DD directive is used to declare a variable of type double word or to reserve memory locations, which can be accessed as type double word. The statement ARRAY DD 25629261H, for example, will define a double word named ARRAY and initialize the double word with the specified value when the program is loaded into memory to be run. The low word, 9261H, will be put in memory at a lower address than the high word.

DQ (DEFINE QUADWORD)

The DQ directive is used to tell the assembler to declare a variable 4 words in length or to reserve 4 words of storage in memory. The statement BIG_NUMBER DQ 243598740192A92BH, for example, will declare a variable named BIG_NUMBER and initialize the 4 words set aside with the specified number when the program is loaded into memory to be run.

DT (DEFINE TEN BYTES)

The DT directive is used to tell the assembler to declare a variable, which is 10 bytes in length or to reserve 10 bytes of storage in memory. The statement PACKED_BCD DT 11223344556677889900 will declare an array named PACKED_BCD, which is 10 bytes in length. It will initialize the 10 bytes with the values 11, 22, 33, 44, 55, 66, 77, 88, 99, and 00 when the program is loaded into memory to be run. The statement RESULT DT 20H DUP (0) will declare an array of 20H blocks of 10 bytes each and initialize all 320 bytes to 00 when the program is loaded into memory to be run.

DW (DEFINE WORD)

The DW directive is used to tell the assembler to define a variable of type word or to reserve storage locations of type word in memory. The statement MULTIPLIER DW 437AH, for example, declares a variable of type word named MULTIPLIER, and initialized with the value 437AH when the program is loaded into memory to be run.

 WORDS DW 1234H, 3456H Declare an array of 2 words and initialize them with the specified values.

 STORAGE DW 100 DUP (0) Reserve an array of 100 words of memory and initialize all 100 words with 0000. Array is named as STORAGE.

 STORAGE DW 100 DUP (?) Reserve 100 word of storage in memory and give it the name

STORAGE, but leave the words un-initialized.

EQU (EQUATE)

EQU is used to give a name to some value or symbol. Each time the assembler finds the given name in the program, it replaces the name with the value or symbol you equated with that name. Suppose, for example, you write the statement FACTOR EQU 03H at the start of your program, and later in the program you write the instruction statement ADD AL, FACTOR. When the assembler codes this

instruction statement, it will code it as if you had written the instruction ADD AL, 03H.
□ CONTROL EQU 11000110 B Replacement
MOV AL, CONTROL Assignment
□ DECIMAL_ADJUST EQU DAA Create clearer mnemonic for DAA
ADD AL, BL Add BCD numbers
DECIMAL_ADJUST Keep result in BCD format Page 28

LENGTH

LENGTH is an operator, which tells the assembler to determine the number of elements in some named data item, such as a string or an array. When the assembler reads the statement MOV CX, LENGTH STRING1, for example, will determine the number of elements in STRING1 and load it into CX. If the string was declared as a string of bytes, LENGTH will produce the number of bytes in the string. If the string was declared as a word string, LENGTH will produce the number of words in the string.

OFFSET

OFFSET is an operator, which tells the assembler to determine the offset or displacement of a named data item (variable), a procedure from the start of the segment, which contains it. When the assembler reads the statement MOV BX, OFFSET PRICES, for example, it will determine the offset of the variable PRICES from the start of the segment in which PRICES is defined and will load this value into BX.

PTR (POINTER)

The PTR operator is used to assign a specific type to a variable or a label. It is necessary to do this in any instruction where the type of the operand is not clear. When the assembler reads the instruction INC [BX], for example, it will not know whether to increment the byte pointed to by BX. We use the PTR operator to clarify how we want the assembler to code the instruction. The statement INC BYTE PTR [BX] tells the assembler that we want to increment the byte pointed to by BX. The statement INC WORD PTR [BX] tells the assembler that we want to increment the word pointed to by BX. The PTR operator assigns the type specified before PTR to the variable specified after PTR.
We can also use the PTR operator to clarify our intentions when we use indirect Jump instructions. The statement JMP [BX], for example, does not tell the assembler whether to code the instruction for a near jump. If we want to do a near jump, we write the instruction as JMP WORD PTR [BX]. If we want to do a far jump, we write the instruction as JMP DWORD PTR [BX].

EVEN (ALIGN ON EVEN MEMORY ADDRESS)

As an assembler assembles a section of data declaration or instruction statements, it uses a location counter to keep track of how many bytes it is from the start of a segment at any time. The EVEN directive tells the assembler to increment the location counter to the next even address, if it is not already at an even address. A NOP instruction will be inserted in the location incremented over.
□ DATA SEGMENT
SALES DB 9 DUP (?) Location counter will point to 0009 after this instruction.
EVEN Increment location counter to 000AH
INVENTORY DW 100 DUP (0) Array of 100 words starting on even address for quicker read
DATA ENDS

PROC (PROCEDURE)

The PROC directive is used to identify the start of a procedure. The PROC directive follows a name you give the procedure. After the PROC directive, the term *near* or the term *far* is used to specify the type of the procedure. The statement DIVIDE PROC FAR, for example, identifies the start of a procedure named DIVIDE and tells the assembler that the procedure is far (in a segment with different name from the one that contains the instructions which calls the procedure). The PROC directive is used with the ENDP directive to “bracket” a procedure. Page 29

ENDP (END PROCEDURE)

The directive is used along with the name of the procedure to indicate the end of a procedure to the assembler. The directive, together with the procedure directive, PROC, is used to “bracket” a procedure.

SQUARE_ROOT PROC Start of procedure.

SQUARE_ROOT ENDP End of procedure.

ORG (ORIGIN)

As an assembler assembles a section of a data declarations or instruction statements, it uses a location counter to keep track of how many bytes it is from the start of a segment at any time. The location counter is automatically set to 0000 when assembler starts reading a segment. The ORG directive allows you to set the location counter to a desired value at any point in the program. The statement ORG 2000H tells the assembler to set the location counter to 2000H, for example.

A “$” it often used to symbolically represent the current value of the location counter, the $ actually represents the next available byte location where the assembler can put a data or code byte. The $ is often used in ORG statements to tell the assembler to make some change in the location counter relative to its current value. The statement ORG $ + 100 tells the assembler increment the value of the location counter by 100 from its current value.

NAME

The NAME directive is used to give a specific name to each assembly module when programs consisting of several modules are written.

LABEL

As an assembler assembles a section of a data declarations or instruction statements, it uses a location counter to be keep track of how many bytes it is from the start of a segment at any time. The LABEL directive is used to give a name to the current value in the location counter. The LABEL directive must be followed by a term that specifics the type you want to associate with that name. If the label is going to be used as the destination for a jump or a call, then the label must be specified as type *near* or type *far*. If the label is going to be used to reference a data item, then the label must be specified as type byte, type word, or type double word. Here’s how we use the LABEL directive for a jump address.

ENTRY_POINT LABEL FAR Can jump to here from another segment

NEXT: MOV AL, BL Can not do a far jump directly to a label with a colon

The following example shows how we use the label directive for a data reference.

STACK_SEG SEGMENT STACK

DW 100 DUP (0) Set aside 100 words for stack

STACK_TOP LABEL WORD Give name to next location after last word in stack

STACK_SEG ENDS

To initialize stack pointer, use MOV SP, OFFSET STACK_TOP.

EXTRN

The EXTRN directive is used to tell the assembler that the name or labels following the directive are in some other assembly module. For example, if you want to call a procedure, which in a program module assembled at a different time from that which contains the CALL instruction, you must tell the assembler

that the procedure is external. The assembler will then put this information in the object code file so that the linker can connect the two modules together. For a reference to externally named variable, you must specify the type of the variable, as in the statement EXTRN DIVISOR: WORD. The statement EXTRN DIVIDE: FAR tells the assembler that DIVIDE is a label of type FAR in another assembler module. Name or labels referred to as external in one module must be declared public with the PUBLIC directive in the module in which they are defined.

PROCEDURE SEGMENT
EXTRN DIVIDE: FAR Found in segment PROCEDURES
PROCEDURE ENDS

PUBLIC

Large program are usually written as several separate modules. Each module is individually assembled, tested, and debugged. When all the modules are working correctly, their object code files are linked together to form the complete program. In order for the modules to link together correctly, any variable name or label referred to in other modules must be declared PUBLIC in the module in which it is defined. The PUBLIC directive is used to tell the assembler that a specified name or label will be accessed from other modules. An example is the statement PUBLIC DIVISOR, DIVIDEND, which makes the two variables DIVISOR and DIVIDEND available to other assembly modules.

SHORT

The SHORT operator is used to tell the assembler that only a 1 byte displacement is needed to code a jump instruction in the program. The destination must in the range of –128 bytes to +127 bytes from the address of the instruction after the jump. The statement JMP SHORT NEARBY_LABEL is an example of the use of SHORT.

TYPE

The TYPE operator tells the assembler to determine the type of a specified variable. The assembler actually determines the number of bytes in the type of the variable. For a byte-type variable, the assembler will give a value of 1, for a word-type variable, the assembler will give a value of 2, and for a double word-type variable, it will give a value of 4. It can be used in instruction such as ADD BX, TYPE-WORD-ARRAY, where we want to increment BX to point to the next word in an array of words.

GLOBAL (DECLARE SYMBOLS AS PUBLIC OR EXTRN)

The GLOBAL directive can be used in place of a PUBLIC directive or in place of an EXTRN directive. For a name or symbol defined in the current assembly module, the GLOBAL directive is used to make the symbol available to other modules. The statement GLOBAL DIVISOR, for example, makes the variable DIVISOR public so that it can be accessed from other assembly modules.

INCLUDE (INCLUDE SOURCE CODE FROM FILE)

This directive is used to tell the assembler to insert a block of source code from the named file into the current source module.

Prepare well
All The Best