**MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY**

**(Autonomous Institution – UGC, Govt. of India)**

Recognized under 2(f) and 12 (B) of UGC ACT 1956

(Affiliated to JNTUH, Hyderabad, Approved by AICTE - Accredited by NBA & NAAC –'A' Grade - ISO 9001:2015 Certified)Maisammaguda, Dhulapally (Post Via. Hakimpet), Secunderabad – 500100, Telangana State, India

**Department of Computer Science and Engineering**

# COMPUTER ORGANIZATION AND OPERATING SYSTEMS

## LECTURE NOTES

**B.Tech(III Year- I SEM)**

**(2018-2019)**

**Faculty Incharge** **HOD-CSE**

## MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

### Department of Computer Science and Engineering

**Name of the subject**: Computer organization and operating systems

**Subject Code:** R15A0569 **Program:** UG **Branch**: ECE **Year**: III **Semester**: I

**No of Pages:**

| **PREPARED BY:** | **PREPARED BY:** |
|---|---|
| Name: VELICHETY ALEKHYA | Name: S.J.SRUTHI RANI YADAV |
| Design. : Asst. Professor | Design. : Asst. Professor |

HOD PRINCIPAL

# SYLLABUS

**UNIT - I**: Basic Structure of Computers: Computer Types, Functional UNIT, Basic Operational Concepts, Bus, Structures, Software, Performance, Multiprocessors and Multi Computers, Data Representation, Fixed Point Representation, Floating - Point Representation. Register Transfer Language and Micro Operations: Register Transfer Language, Register Transfer Bus and Memory Transfers, Arithmetic Micro Operations, Logic Micro Operations, Shift Micro Operations, Arithmetic Logic Shift Unit, Instruction Codes, Computer Registers Computer Instructions - Instruction Cycle. Memory - Reference Instructions, Input - Output and Interrupt, STACK Organization, Instruction Formats, Addressing Modes, DATA Transfer and Manipulation, Program Control, Reduced Instruction Set Computer.

**UNIT - II**: Micro Programmed Control: Control Memory, Address Sequencing, Micro program Examples, Design of Control Unit, Hard Wired Control, Micro programmed Control. The Memory System: Basic Concepts of Semiconductor RAM Memories, Read-Only Memories, Cache Memories Performance Considerations, Virtual Memories secondary Storage, Introduction to RAID.

**UNIT - III**: Input-Output Organization: Peripheral Devices, Input-Output Interface, Asynchronous Data Transfer Modes, Priority Interrupt, Direct Memory Access, Input-Output Processor (IOP), Serial Communication; Introduction to Peripheral Components, Interconnect (PCI) Bus, Introduction to Standard Serial Communication Protocols like RS232, USB, IEEE1394.

**UNIT - IV**: Operating Systems Overview: Overview of Computer Operating Systems Functions, Protection and Security, Distributed Systems, Special Purpose Systems, Operating Systems Structures Operating System Services and Systems Calls, System Programs, Operating System Generation. Memory Management: Swapping, Contiguous Memory Allocation, Paging, Structure of the Page Table, Segmentation, Virtual Memory, Demand Paging, Page-Replacement Algorithms, Allocation of Frames, Thrashing Case Studies - UNIX, Linux, Windows Principles of Deadlock: System Model, Deadlock Characterization, Deadlock Prevention, Detection and Avoidance, Recovery from Deadlock.

**UNIT - V**: File System Interface: The Concept of a File, Access Methods, Directory Structure, File System Mounting, File Sharing, Protection. File System Implementation: File System Structure, File system Implementation, Directory Implementation, Allocation Methods, Free-Space Management.

**TEXT BOOKS**:

1. Computer Organization - Carl Hamacher, Zvonks Vranesic, SafeaZaky, 5th Edition, McGraw Hill.
2. Computer System Architecture - M. moris mano, 3rd edition, Pearson
3. Operating System Concepts - Abreham Silberchatz, Peter B. Galvin, Greg Gagne, 8th Edition, John Wiley.

**REFERENCE BOOKS**:

1. Computer Organization and Architecture - William Stallings 6th Edition, Pearson
2. Structured Computer Organization - Andrew S. Tanenbaum, 4th Edition, PHI

## DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

### Vision

- To acknowledge quality education and instill high patterns of discipline making the students technologically superior and ethically strong which involves the improvement in the quality of life in human race.

### Mission

- To achieve and impart holistic technical education using the best of infrastructure, outstanding technical and teaching expertise to establish the students into competent and confident engineers.
- Evolving the center of excellence through creative and innovative teaching learning practices for promoting academic achievement to produce internationally accepted competitive and world class professionals.

## PROGRAMME EDUCATIONAL OBJECTIVES (PEOs)

### PEO1 – ANALYTICAL SKILLS

1.To facilitate the graduates with the ability to visualize, gather information, articulate, analyze, solve complex problems, and make decisions. These are essential to address the challenges of complex and computation intensive problems increasing their productivity.

### PEO2 – TECHNICAL SKILLS

2.To facilitate the graduates with the technical skills that prepare them for immediate employment and pursue certification providing a deeper understanding of the technology in advanced areas of computer science and related fields, thus encouraging to pursue higher education and research based on their interest.

### PEO3 – SOFT SKILLS

3.To facilitate the graduates with the soft skills that include fulfilling the mission, setting goals, showing self-confidence by communicating effectively, having a positive attitude, get involved in team-work, being a leader, managing their career and their life.

### PEO4 – PROFESSIONAL ETHICS

4.To facilitate the graduates with the knowledge of professional and ethical responsibilities by paying attention to grooming, being conservative with style, following dress codes, safety codes,and adapting themselves to technological advancements.

## PROGRAM SPECIFIC OUTCOMES (PSOs)

After the completion of the course, B. Tech Computer Science and Engineering, the graduates will have the following Program Specific Outcomes:

1. **Fundamentals and critical knowledge of the Computer System:-** Able to Understand the working principles of the computer System and its components , Apply the knowledge to build, asses, and analyze the software and hardware aspects of it .

2. **The comprehensive and Applicative knowledge of Software Development:** Comprehensive skills of Programming Languages, Software process models, methodologies, and able to plan, develop, test, analyze, and manage the software and hardware intensive systems in heterogeneous platforms individually or working in teams.

3. **Applications of Computing Domain & Research:** Able to use the professional, managerial, interdisciplinary skill set, and domain specific tools in development processes, identify the research gaps, and provide innovative solutions to them.

# UNIT-1

## BASIC STRUCTURE OF COMPUTERS

### Contents:

# Unit- I

## Basic Structure of Computers

Computer Architecture in general covers three aspects of computer design namely: Computer Hardware, Instruction set Architecture and Computer Organization. Computer hardware consists of electronic circuits, displays, magnetic and optical storage media and communication facilities. Instruction set Architecture is programmer visible machine interface such as instruction set, registers, memory organization and exception handling. Two main approaches are mainly CISC (Complex Instruction Set Computer) and RISC (Reduced Instruction Set Computer) Computer Organization includes the high level aspects of a design, such as

memory system, the bus structure and the design of the internal CPU.

### Computer Types

Computer is a fast electronic calculating machine which accepts digital input, processes it according to the internally stored instructions (Programs) and produces the result on the output device. The internal operation of the computer can be as depicted in the figure below:

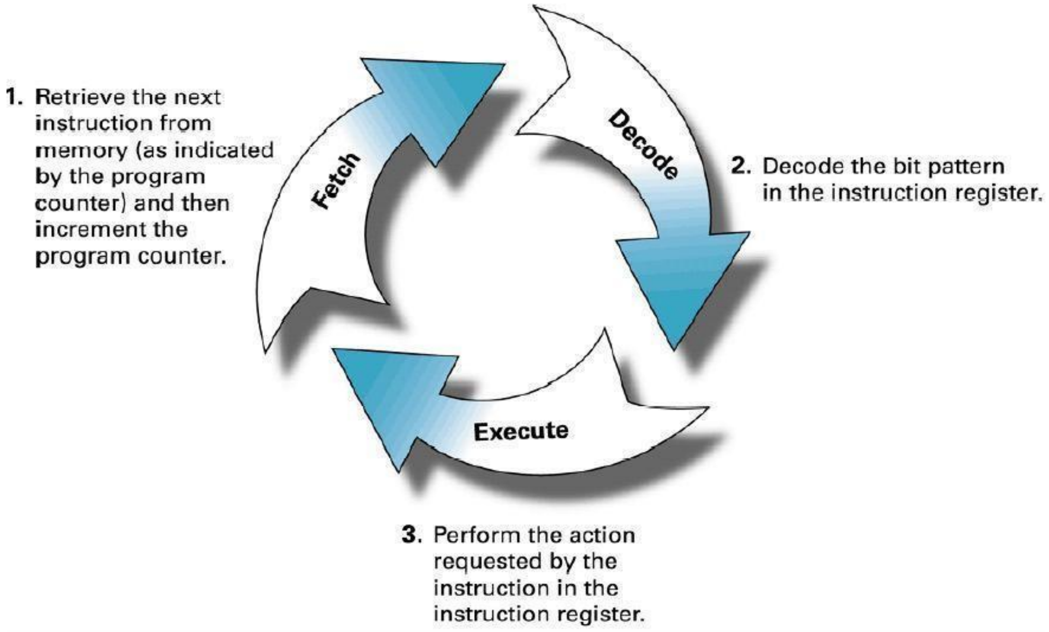

1. Retrieve the next instruction from memory (as indicated by the program counter) and then increment the program counter.
Fetch
Decode
2. Decode the bit pattern in the instruction register.
Execute
3. Perform the action requested by the instruction in the instruction register.

**Figure 1: Fetch, Decode and Execute steps in a Computer System**

The computers can be classified into various categories as given below:

- Micro Computer
- Laptop Computer
- Work Station
- Super Computer
- Main Frame
- Hand Held
- Multi core

**Micro Computer:** A personal computer; designed to meet the computer needs of an individual. Provides access to a wide variety of computing applications, such as word processing, photo editing, e-mail, and internet.

**Laptop Computer:** A portable, compact computer that can run on power supply or a battery unit. All components are integrated as one compact unit. It is generally more expensive than a comparable desktop. It is also called a Notebook.

**Work Station:** Powerful desktop computer designed for specialized tasks. Generally used for tasks that requires a lot of processing speed. Can also be an ordinary personal computer attached to a LAN (local area network).

**Super Computer:** A computer that is considered to be fastest in the world. Used to execute tasks that would take lot of time for other computers. For Ex: Modeling weather systems, genome sequence, etc (Refer site: http://www.top500.org/)

**Main Frame:** Large expensive computer capable of simultaneously processing data for hundreds or thousands of users. Used to store, manage, and process large amounts of data that need to be reliable, secure, and centralized.

**Hand Held:** It is also called a PDA (Personal Digital Assistant). A computer that fits into a pocket, runs on batteries, and is used while holding the unit in your hand. Typically used as an appointment book, address book, calculator and notepad.

**Multi Core:** Have Multiple Cores – parallel computing platforms. Many Cores or computing elements in a single chip. Typical Examples: Sony Play station, Core 2 Duo, i3, i7 etc.

## Functional Units

A computer in its simplest form comprises five functional units namely input unit, output unit memory unit, arithmetic & logic unit and control unit. Figure 2 depicts the functional units of a computer system.

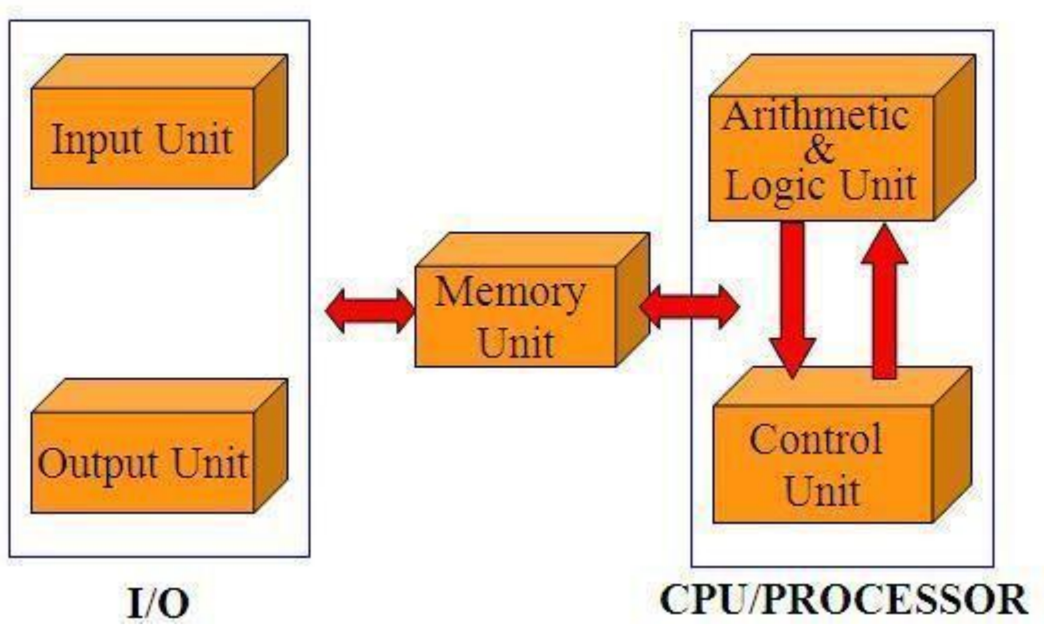


**Figure 2: Basic functional units of a computer**

Let us discuss about each of them in brief:

1. **Input Unit:** Computer accepts encoded information through input unit. The standard input device is a keyboard. Whenever a key is pressed, keyboard controller sends the code to CPU/Memory.

   Examples include Mouse, Joystick, Tracker ball, Light pen, Digitizer, Scanner etc.

2. **Memory Unit:** Memory unit stores the program instructions (Code), data and results of computations etc. Memory unit is classified as:

   Primary /Main Memory

- Secondary /Auxiliary Memory

**Primary memory** is a semiconductor memory that provides access at high speed. Run time program instructions and operands are stored in the main memory. Main memory is classified again as ROM and RAM. ROM holds system programs and firmware routines such as BIOS, POST, I/O Drivers that are essential to manage the hardware of a computer. RAM is termed as Read/Write memory or user memory that holds run time program instruction and data. While primary storage is essential, it is volatile in nature and expensive. Additional requirement of memory could be supplied as auxiliary memory at cheaper cost.

**Secondary memories** are non volatile in nature.

**Arithmetic and logic unit:** ALU consist of necessary logic circuits like adder, comparator etc., to perform operations of addition, multiplication, comparison of two numbers etc.

**Output Unit:** Computer after computation returns the computed results, error messages, etc. via output unit. The standard output device is a video monitor, LCD/TFT monitor. Other output devices are printers, plotters etc.

**Control Unit:** Control unit co-ordinates activities of all units by issuing control signals. Control signals issued by control unit govern the data transfers and then appropriate operations take place. Control unit interprets or decides the operation/action to be performed.

The operations of a computer can be summarized as follows:

A set of instructions called a program reside in the main memory of computer.

The CPU fetches those instructions sequentially one-by-one from the main memory, decodes them and performs the specified operation on associated data operands in ALU.

Processed data and results will be displayed on an output unit.

All activities pertaining to processing and data movement inside the computer machine are governed by control unit.

## Basic Operational Concepts

An Instruction consists of two parts, an Operation code and operand/s as shown below:

OPCODE OPERAND/s

Let us see a typical instruction

ADD LOCA, R0

This instruction is an addition operation. The following are the steps to execute the instruction:

Step 1: Fetch the instruction from main memory into the processor

Step 2: Fetch the operand at location LOCA from main memory into the processor

Step 3: Add the memory operand (i.e. fetched contents of LOCA) to the contents of register R0 Step 4: Store the result (sum) in R0.

The same instruction can be realized using two instructions as Load
LOCA, R1
Add R1, R0
The steps to execute the instructions can be enumerated as below:

Step 1: Fetch the instruction from main memory into the processor Step 2: Fetch the operand at location LOCA from main memory into

the processor Register R1

Step 3: Add the content of Register R1 and the contents of register R0 Step 4: Store the result (sum) in R0.

Figure 3 below shows how the memory and the processor are connected. As shown in the diagram, in addition to the ALU and the control circuitry, the processor contains a number of registers used for several different purposes. The instruction register holds the instruction that is currently being executed. The program counter keeps track of the execution of the program. It contains the memory address of the next instruction to be fetched and executed. There are n general purpose registers R0 to $R_{n-1}$ which can be used by the programmers during writing programs.

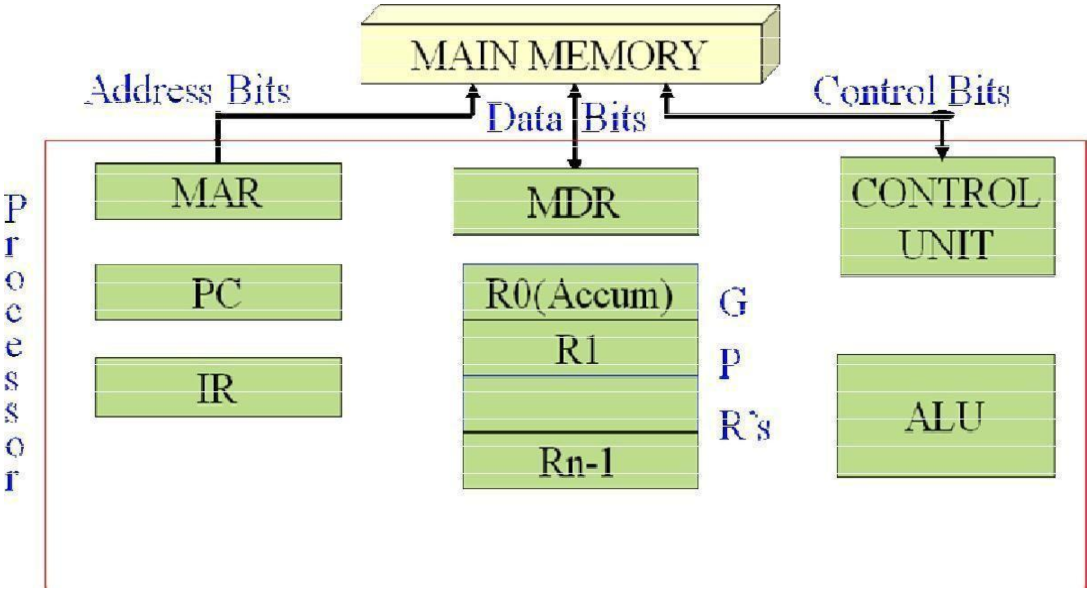


**Figure 3: Connections between the processor and the memory**

The interaction between the processor and the memory and the direction of flow of information is as shown in the diagram below:

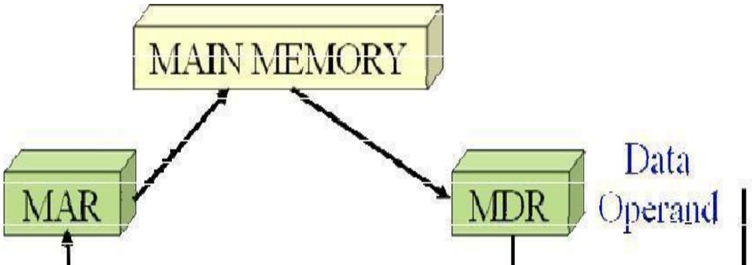


**Figure 4: Interaction between the memory and the ALU**

## BUS STRUCTURES

Group of lines that serve as connecting path for several devices is called a bus (one bit per line). Individual parts must communicate over a communication line or path for exchanging data, address and control information as shown in the diagram below. Printer example – processor to printer. A common approach is to use the concept of buffer registers to hold the content during the transfer.

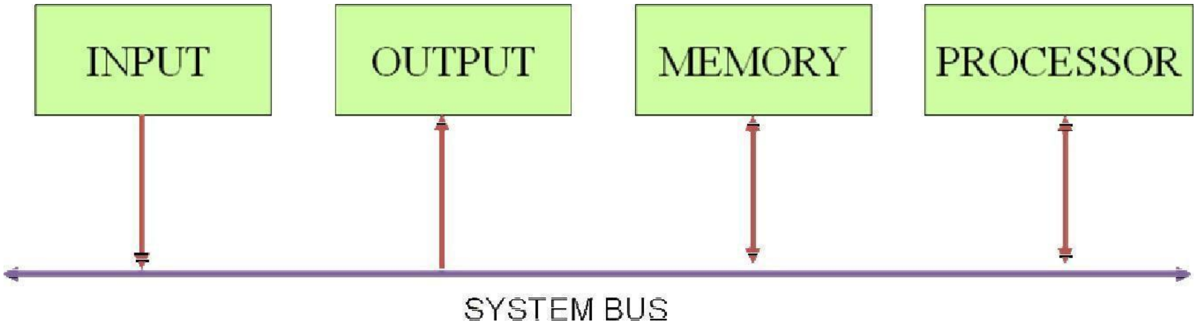


**Figure 5: Single bus structure**

## SOFTWARE

If a user wants to enter and run an application program, he/she needs a System Software. System Software is a collection of programs that are executed as needed to perform functions such as:

Receiving and interpreting user commands

Entering and editing application programs and storing then as files in secondary storage devices

Running standard application programs such as word processors, spread sheets, games etc.

Operating system - is key system software component which helps the user to exploit the below underlying hardware with the programs.

## USER PROGRAM and OS ROUTINE INTERACTION

Let‘s assume computer with 1 processor, 1 disk and 1 printer and application program is in machine code on disk. The various tasks are performed in a coordinated fashion, which is called multitasking. t0, t1 …t5 are the instances of time and the interaction during various instances as given below:

t0: the OS loads the program from the disk to memory t1: program executes
t2: program accesses disk
t3: program executes some more t4: program accesses printer
t5: program terminates

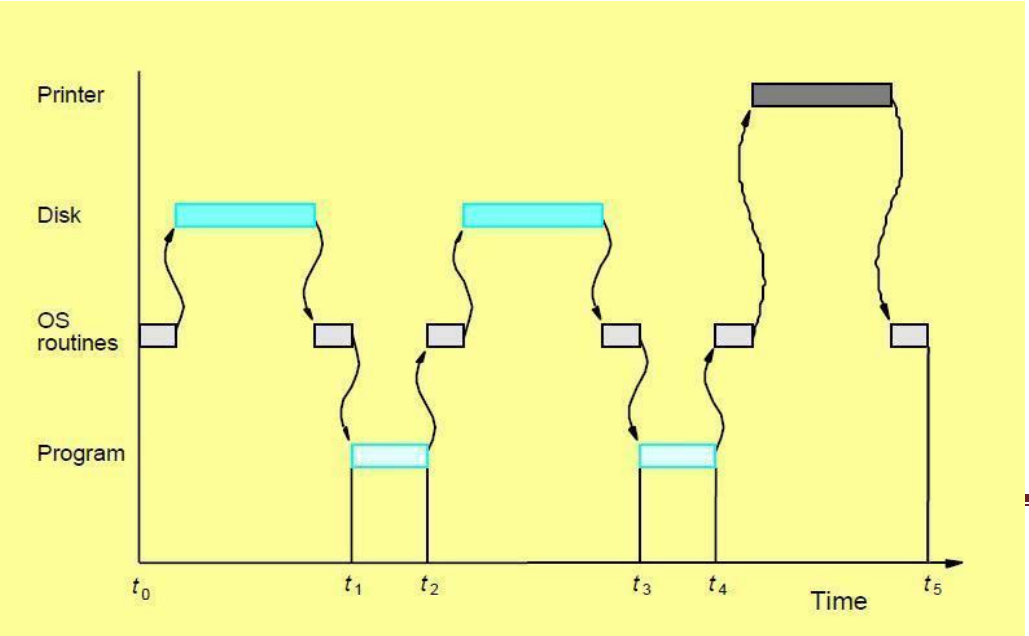


**Figure 6 :User program and OS routine sharing of the processor**

## PERFORMANCE

The total time required to execute a program is the most important measure of performance for a computer. (t0-t5 of earlier example). Compiler, instruction set and hardware architecture, program all have impact on performance.

**Basic Performance Equation:** The basic performance equation is given by

**T = (N * S) / R**

where T=execution time, N=number of instructions, S=average cycles per instruction, R=clock rate in cycles per second

## CACHING

Commonly used data are copied to on-processor memory (cache) to reduce access time.

Small memories can be made with higher speed than large ones. In a computer, we need both.

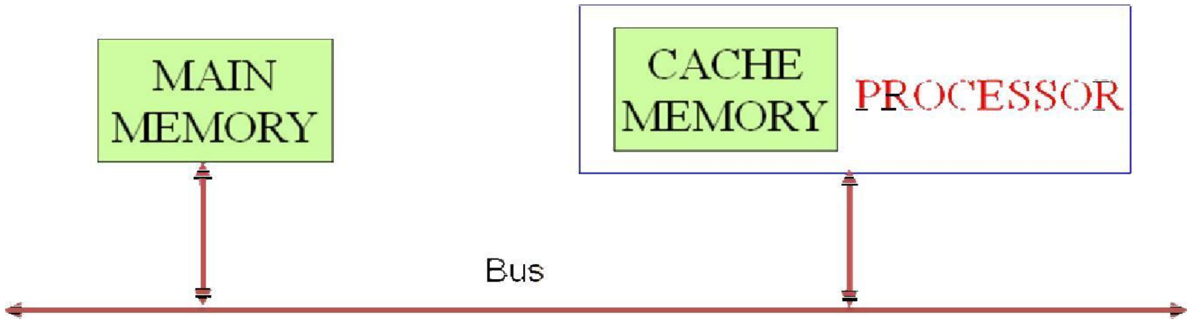


**Figure 7: The processor cache**

## PIPELINING and SUPERSCALR OPERATION

Pipelining: Like a production line, instruction execution overlapped so greater parallelism is achieved.
Superscalar operation: Execute several instructions simultaneously using multiple ALU‘s.

## CISC vs RISC

Reduced instruction set computer
– Large N, small S
Complex instruction set computer
– Small N, large S

## COMPILER

Translates high level language such as C, C++ and Java to machine instructions. Aims to reduce N×S

## PERFORMANCE MEASUREMENT

Benchmark refers to standard task used to measure how well a processor operates. To evaluate the performance of Computers, a non-profit organization known as SPEC-System Performance Evaluation Corporation employs agreed-upon application programs of real world for benchmarks. Accordingly, it gives performance measure for a computer as the time required to execute a given benchmark program. The SPEC rating is computed as follows

$$\text{SPEC Rating} - \frac{\text{Running time on a Reference Computer}}{\text{Running time on a test Computer}}$$

## MULTIPROCESSORS AND MULTICOMPUTERS

**Multicomputer**-- A computer made up of several computers. The term generally refers to an architecture in which each processor has its own memory rather than multiple processors with a shared memory. Distributed computing deals with hardware and software systems containing more than one processing.

Element or storage element, concurrent processes, or multiple programs, running under a loosely or tightly controlled regime. A multicomputer may be considered to be either a loosely coupled NUMA computer or a tightly coupled cluster. Multi computers are commonly used when strong computer power is required in an environment with restricted physical space or electrical power.

Common suppliers include Mercury Computer Systems, CSPI, and SKY Computers. Common uses include 3D medical imaging devices and mobile radar. In distributed computing a program is split up into parts that run simultaneously on multiple computers communicating over a network. Distributed computing is a form of parallel computing, but parallel computing is most commonly used to describe program parts running simultaneously on multiple processors in the same computer. Both types of processing require dividing a program into parts that can run simultaneously, but distributed programs often must deal with heterogeneous environments, network links of varying latencies, and unpredictable failures in the network or the computers.

**Multiprocessor**-- A multiprocessor system is simply a computer that has more than one CPU on its motherboard. If the operating system is built to take advantage of this, it can run different processes (or different threads belonging to the same process) on different CPUs. Multiprocessing is the use of two or more central processing units (CPUs) within a single computer system. The term also refers to the ability of a system to support more than one processor and/or the ability to allocate tasks between them.[1] There are many variations on this basic theme, and the definition of multiprocessing can vary with context, mostly as a function of how CPUs are defined (multiple cores on one die, multiple chips in one package, multiple packages in one system unit, etc.).

## Data Representation

### Introduction

The digital computer is a digital system that performs various computational tasks. The word ***digital*** implies that the information in the computer is represented by variables that take a limited number of discrete values. These values are processed internally by components that can maintain a limited number of discrete states. The decimal digits 0, 1, 2, 9, for example, provide 10 discrete values. The first electronic digital computers, developed in the late 1940s, were used primarily for numerical computations. In this case the discrete elements are the digits. From this application the term ***digital computer*** has emerged. In practice, digital computers function more reliably only if two states are used. Because of the physical restriction of components, and because human logic tends to be binary (i.e. true-or-false, yes-or-no statements), digital components that are constrained to take discrete values are further constrained to take only two values and are said to be ***binary***.

### Objectives:

After studying this unit, the learner will be able to

□ Explain various units of a digital computer

□ Understand different data types

□ Explain fixed and floating point number representation

□ Discuss various binary and error detection codes

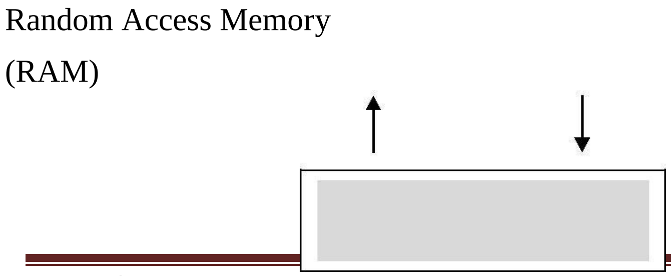
### Digital Computers

Digital computers use the binary number system, which has two digits: 0 and 1. A binary digit is called a ***bit***. Information is represented in digital computers in groups of bits. By using various coding techniques, groups of bits can be made to represent not only binary numbers but also other discrete symbols, such as decimal digits or letters of the alphabet. By the judicious use of binary arrangements and by using various coding techniques, the groups of bits are used to develop complete sets of instructions for performing various types of computations. In contrast to the common decimal numbers that employ the base 10 system, binary numbers use a base 2 system with two digits: 0 and 1.

The decimal equivalent of a binary number can be found by expanding it into a power series with a base of 2. For example, the binary number 1001011 represents a quantity that can be converted to a decimal number by multiplying each bit by the base 2 raised to an integer power as follows:

$1 \text{ x } 2^6 + 0 \text{ x } 2^5 + 0 \text{ x } 2^4 + 1 \text{ x } 2^3 + 0 \text{ x } 2^2 + 1 \text{ x } 2^1 + 1 \text{ x } 2^0 = 75$

The seven bits 1001011 represent a binary number whose decimal equivalent is 75. However, this same group of seven bits represents the letter K when used in conjunction with a binary code for the letters of the alphabet. It may also represent a control code for specifying some decision logic in a particular digital computer. In other words, groups of bits in a digital computer are used to represent many different things. This is similar to the concept that the same letters of an alphabet are used to construct different languages, such as English and French. A computer system is sometimes subdivided into two functional entities: hardware and software. The ***hardware*** of the computer consists of all the electronic components and electromechanical devices that comprise the physical entity of the device. Computer ***software*** consists of the instructions and the data that the computer manipulates to perform various data-processing tasks. A sequence of instructions for the computer is called a ***program***. The data that are manipulated by the program constitute the ***data base***. A computer system is composed of its hardware and the system software available for its use. The ***system software*** of a computer consists of a collection of programs whose purpose is to make more effective use of the computer. The programs included in a systems software package are referred to as the ***operating system***. They are distinguished from application programs written by the user for the purpose of solving particular problems. For example, a high-level language program written by a user to solve particular data-processing needs is an ***application program***, but the compiler that translates the high-level language program to machine language is a ***system program***. The customer who buys a computer system would need, in addition to the hardware, any available software needed for effective operation of the computer. The system software is an indispensable part of a total computer system. Its function is to compensate for the differences that exist between user needs and the capability of the hardware. The hardware of the computer is usually divided into three major parts as shown in Fig. 1.1.



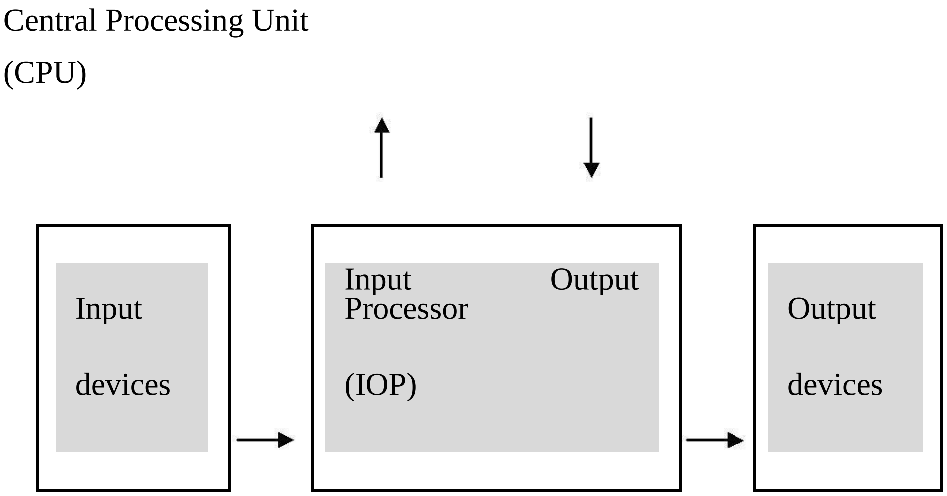


**Fig. 1.1: Block diagram of a digital computer**

The ***Central Processing Unit (CPU)*** contains an ***Arithmetic and Logic Unit (ALU)*** for manipulating data, a number of registers for storing data, and control circuits for fetching and executing instructions. The memory of a computer contains storage for instructions and data. It is called a ***Random Access Memory (RAM)*** because the CPU can access any location in memory at random and retrieve the binary information within a fixed interval of time. The ***Input-Output Processor (IOP)*** contains electronic circuits for communicating and controlling the transfer of information between the computer and the outside world. The input and output devices connected to the computer include keyboards, printers, terminals, magnetic disk drives, and other communication devices. This book provides the basic knowledge necessary to understand the hardware operations of a computer system. The subject is sometimes considered from three different points of view, depending on the interest of the investigator. When dealing with computer hardware it is customary to distinguish between what is referred to as computer organization, computer design, and computer architecture.

***Computer organization*** is concerned with the way the hardware components operate and the way they are connected together to form the computer system. The various components are assumed to be in place and the task is to investigate the organizational structure to verify that the computer parts operate as intended.

***Computer design*** is concerned with the hardware design of the computer. Once the computer specifications are formulated, it is the task of the designer to develop hardware for the system. Computer design is concerned with the determination of what hardware should be used and how the parts should be connected. This aspect of computer hardware is sometimes referred to as ***computer implementation***.

***Computer architecture*** is concerned with the structure and behavior of the computers as seen by the user. It includes the information formats, the instruction set, and techniques for addressing memory. The architectural design of a computer system is concerned with the specifications of the various functional modules, such as processors and memories, and structuring them together into a computer system.

## Data Types

Binary information in digital computers is stored in memory or processor registers. Registers contain either data or control information. Control information is a bit or a group of bits used to specify the sequence of command signals needed for manipulation of the data in other registers. Data are numbers and other binary-coded information that are operated on, to achieve required computational results. The data types found in the registers of digital computers may be classified as being one of the following categories:

(1) numbers used in arithmetic computations, (2) letters of the alphabet used in data processing, and (3) other discrete symbols used for specific purposes. All types of data, except binary numbers, are represented in computer registers in binary-coded form. This is because registers are made up of flip-flops and flip-flops are two-state devices that can store only 1‘s and 0‘s. The binary number system is the most natural system to be used in a digital computer. But sometimes it is convenient to employ different number systems, especially the decimal number system, since it is used by people to perform arithmetic computations.

## Number Systems

A number system of ***base***, or ***radix, r*** is a system that uses distinct symbols for ***r*** digits. Numbers are represented by a string of digit symbols. To determine the quantity that the number represents, it is necessary to multiply each digit by an integer power of ***r*** and then form the sum of all weighted digits. For example, the ***decimal number system*** in everyday use employs the radix 10 system. The 10 symbols are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. The string of digits 724.5 is interpreted to represent the quantity.

$7 \text{ x } 10^2 + 2 \text{ x } 10^1 + 4 \text{ x } 10^0 + 5 \text{ x } 10^{-1}$

that is, 7 hundreds, plus 2 tens, plus 4 units, plus 5 tenths. Every decimal number can be similarly interpreted to find the quantity it represents. To distinguish between different radix numbers, the digits will be enclosed in parentheses and the radix of the number inserted as a subscript. For example, to show the equality between decimal and binary forty-five we will write $(101101)_2 = (45)_{10}$. Besides the decimal and binary number systems, the ***octal*** (radix 8) and ***hexadecimal*** (radix 16) are important in digital computer work. The eight symbols of the octal system are 0, 1, 2, 3, 4, 5, 6, and 7. The 16 symbols of the hexadecimal system are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. The last six symbols are, unfortunately, identical to the letters of the alphabet and can cause confusion at times. However, this is the convention that has been adopted. When used to represent hexadecimal digits, the symbols A, B, C, D, E, F correspond to the decimal numbers 10, 11, 12, 13, 14, 15, respectively.

A number in radix ***r*** can be converted into the familiar decimal system by forming the sum of the weighted digits. For example, octal 736.4 is converted to decimal as follows:

$(736.4)_8 = 7 \text{ x } 8^2 + 3 \text{ x } 8^1 + 6 \text{ x } 8^0 + 4 \text{ x } 8^{-1}$

$= 7 \text{ x } 64 + 3 \text{ x } 8 + 6 \text{ x } 1 + 4 / 8 = (478.5)_{10}$

The equivalent decimal number of hexadecimal F3 is obtained from the following calculation:

$(F3)_{16} = F \text{ x } 16 + 3 = 15 \text{ x } 16 + 3 = (243)_{10}$.

| | | | |
|---|---|---|---|
| **Integer = 41** | | | **Fraction = 0.6875** |
| **41** | | **0.6875** | |
| **20** | **1** | | **x 2** |
| **10** | **0** | **1.3750** | |
| **5** | **0** | | **x 2** |
| **2** | **1** | **0.7500** | |
| **1** | **0** | | **x 2** |
| **0** | **1** | **1.5000** | |

X 2

1.0000

**$(41)_{10} = (101001)_2$** **$(0.6875)_{10} = (0.1011)_2$**

**$(41.6875)_{10} = (101001.1011)_2$**

**Fig. 1.2: Conversion of decimal 41.6875 into binary**

The conversion of decimal 41.6875 into binary is done by first separating the number into its integer part 41 and fraction part 0.6875. The integer part is converted by dividing 41 by ***r*** = 2 to give an integer quotient 20 and a remainder of 1. The quotient is again divided by 2 to give a new quotient and remainder. This process is repeated until the integer quotient becomes 0. The coefficients of the binary number are obtained from the remainders with the first remainder giving the low-order bit of the converted binary number.The fraction part is converted by multiplying it by ***r*** = 2 to give an integer and a fraction. The new fraction (without the integer) is multiplied again by 2 to give a new integer and a new fraction. This process is repeated until the fraction part becomes zero or until the number of digits obtained gives the required accuracy.

**Octal and Hexadecimal Numbers**

The conversion from and to binary, octal, and hexadecimal representation plays an important part in digital computers. Since $2^3 = 8$ and $2^4 = 16$, each octal digit corresponds to three binary digits and each hexadecimal digit corresponds to four binary digits. The conversion from binary to octal is easily accomplished by partitioning the binary number into groups of three bits each starting from the least significant bit (LSB) position. The corresponding octal digit is then assigned to each group of bits and the string of digits so obtained gives the octal equivalent of the binary number. Consider, for example, a 16-bit register. Physically, one may think of the register as composed of 16 binary storage cells, with each cell capable of holding either a 1 or a 0. Suppose that the bit configuration stored in the register is

as shown in Fig.1.3.Since a binary number consists of a string of 1‘s and 0‘s, the 16- bit register can be used to store any binary number from 0 to $2^{16} – 1$. For the particular example shown, the binary number stored in the register is the equivalent of decimal 44899. Starting from the low-order bit, we partition the register into groups of three bits each (the sixteenth bit remains in a group by itself). Each group of three bits is assigned its octal equivalent and placed on the top of the register. The string of octal digits so obtained represents the octal equivalent of the binary number.

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Octal** | 1 | | **2** | | | **7** | | | **5** | | | **4** | | | **3** | |
| **Binary** | **1** | **0** | **1** | **0** | **1** | **1** | **1** | **1** | **0** | **1** | **1** | **0** | **0** | **0** | **1** | **1** |
| **Hexadecimal** | | | **A** | | | | **F** | | | | **6** | | | | **3** | |

**Fig. 1.3: Binary, octal and hexadecimal conversion.**

Conversion from binary to hexadecimal is similar except that the bits are divided into groups of four. The corresponding hexadecimal digit for each group of four bits is written as shown below the register of Fig.1.3. The string of hexadecimal digits so obtained represents the hexadecimal equivalent of the binary number. The corresponding octal digit for each group of three bits is easily remembered after studying the first eight entries listed in Table 1.1. The correspondence between a hexadecimal digit and its equivalent 4-bit code can be found in the first 16 entries Table 1.1 lists a few octal numbers and their representation in registers in binary-coded form. The binary code is obtained by the procedure explained above. Each octal digit is assigned a 3-bit code as specified by the entries of the first eight digits in the table. Similarly, Table 1.2 lists a few hexadecimal numbers and their representation in registers in binary-coded form. Here the binary code is obtained by assigning to each hexadecimal digit the 4-bit code listed in the first 16 entries of the table. Comparing the binary-coded octal and hexadecimal numbers with their binary number equivalent we find that the bit combination in all three representations is exactly the same. For example, decimal 99, when converted to binary, becomes 1100011. The binary-coded octal equivalent of decimal 99 is 143 (001 100 011) and the binary-coded hexadecimal of decimal 99 is 63 (0110 0011). If we neglect the leading zeros in these two binary representations, we find that their bit combination is identical. This should be so because of the straight forward conversion that exists between binary numbers and octal or hexadecimal. The point of all this is that a string of 1s and 0s stored in a register could represent a binary number, but this same string of bits may be interpreted as holding an octal number in binary-coded form (if we divide the bits into groups of three) or as holding a hexadecimal number in binary-coded form (if we divide the bits into groups of four). The registers in a digital computer contain many bits. Specifying the content of registers by their binary values will require a long string of binary digits. It is more convenient to specify content of registers by their octal or hexadecimal equivalent. The number of digits is reduced by one-third in the octal designation and by one-fourth in the hexadecimal designation. For example, the binary number 1111 1111 1111 has 12 digits. It can be expressed in octal as 7777 (four digits) or in hexadecimal as FFF (three digits). Computer manuals invariably choose either the octal or the hexadecimal designation for specifying contents of registers.

**Decimal Representation**

The binary number system is the most natural system for a computer, but people are accustomed to the decimal system. One way to solve this conflict is to convert all input decimal numbers into binary numbers, let the computer perform all arithmetic operations in binary and then convert the binary results back to decimal for the human user to understand. However, it is also possible for the computer to perform arithmetic operations directly with decimal numbers provided they are placed in registers in a coded form. Decimal numbers enter the computer usually as binary-coded alphanumeric characters. These codes, introduced later, may contain from six to eight bits for each decimal digit. When decimal numbers are used for internal arithmetic computations, they are converted into a binary code with four bits per digit.A binary code is a group of n bits that assume up to $2^n$ distinct combination of 1s and 0s with each combination representing one element of the set that is being coded. For example, a set of four elements can be coded by a 2-bit code with each element assigned one of the following bit combinations; 00, 01, 10, or 11. A set of eight elements requires a 3-bit code; a set of 16 elements requires a 4-bit code, and so on. A binary code will have some unassigned bit combinations if the number of elements in the set is not a multiple power of 2. The 10 decimal digits from 0 to 9 form such a set. A binary code that distinguishes among 10 elements must contain at least four bits, but six combinations will remain unassigned. Numerous different codes can be obtained by arranging four bits in 10 distinct combinations. The bit assignment most commonly used for the decimal digits is the straight binary assignment listed in the first 10 entries of Table 1.3. This particular code is called

***Binary Coded Decimal*** **(BCD)**. Other decimal codes are sometimes used and a few of them are given in the section 1.7. It is very important to understand the difference between the conversion of decimal numbers into binary and the binary coding of decimal numbers. For example, when converted to a binary number, the decimal number 99 is represented by the string of bits 1100011, but when represented in BCD, it becomes 1001 1001. The only difference between a decimal number represented by the familiar digit symbols 0, 1, 2, …, 9 and the BCD symbols 0001, 0010, …, 1001 is in the symbols used to represent the digits – the number itself is exactly the same. A few decimal numbers and their representation in BCD are listed in Table 1.3.

| ***Decimal number*** | ***Binary-coded decimal (BCD) number*** |
|---|---|
| 0 | 0000 |
| 1 | 0001 |
| 2 | 0010 |
| 3 | 0011 |
| 4 | 0100 |
| 5 | 0101 |
| 6 | 0110 |
| 7 | 0111 |
| 8 | 1000 |
| 9 | 1001 |
| 10 | 0001 0000 |
| 20 | 0010 0000 |

Code for one decimal digit

| | |
|---|---|
| 50 | 0101 0000 |
| 99 | 1001 1001 |
| 248 | 0010 0100 1000 |

**Table 1.3: Binary Coded Decimal (BCD) Numbers**

**Alphanumeric Representation**

Many applications of digital computers require the handling of data that consist not only of numbers, but also of the letters of the alphabet and certain special characters. An ***alphanumeric character set*** is a set of elements that includes the 10 decimal digits, the 26 letters of the alphabet and a number of special characters, such as $, +, and =. Such a set contains between 32 and 64 elements (if only uppercase letters are included) or between 64 and 128 (if both uppercase and lowercase letters are included). In the first case, the binary code will require six bits and in the second case, seven bits. The standard alphanumeric binary code is the

***American Standard Code for Information Interchange (ASCII),*** which uses seven bits to code 128 characters. The binary code for the uppercase letters, the decimal digits, and a few special characters is listed in Table 1.4. Note that the decimal digits in ASCII can be converted into BCD by removing the three high-order bits, 011.

| ***Character*** | ***Binary code*** | ***Character*** | ***Binary code*** |
|---|---|---|---|
| A | 100 0001 | 0 | 011 0000 |
| B | 100 0010 | 1 | 011 0001 |
| C | 100 0011 | 2 | 011 0010 |
| D | 100 0100 | 3 | 011 0011 |
| E | 100 0101 | 4 | 011 0100 |
| F | 100 0110 | 5 | 011 0101 |
| G | 100 0111 | 6 | 011 0110 |
| H | 100 1000 | 7 | 011 0111 |
| I | 100 1001 | 8 | 011 1000 |
| J | 100 1010 | 9 | 011 1001 |
| K | 100 1011 | | |
| L | 100 1100 | | |
| M | 100 1101 | Space | 010 0000 |

| | | | |
|---|---|---|---|
| N | 100 1110 | . | 010 1110 |
| O | 100 1111 | ( | 010 1000 |
| P | 101 0000 | + | 010 1011 |
| Q | 101 0001 | $ | 010 0100 |
| R | 101 0010 | * | 010 0100 |
| S | 101 0011 | ) | 010 1001 |
| T | 101 0100 | - | 010 1101 |
| U | 101 0101 | / | 010 1111 |
| V | 101 0110 | , | 010 1100 |
| W | 101 0111 | = | 011 1101 |
| X | 101 1000 | | |
| Y | 101 1001 | | |
| Z | 101 1010 | | |

**Table 1.4: American Standard Code for Information Interchange (ASCII)**

## Complements

Complements are used in digital computers for simplifying the subtraction operation and for logical manipulation. There are two types of complements for each base ***r*** system: the ***r's*** complement and the ***(r-1)'s*** complement. When the value of the base ***r*** is substituted in the name, the two types are referred to as the 2‘s and 1‘s complement for binary numbers and the 10‘s and 9‘s complement for decimal numbers.

### (r-1)'s Complement

Given a number ***N*** in base ***r*** having ***n*** digits, the ***(r-1)'s*** complement of ***N*** is defined as $(r^n - 1) - N$. For decimal numbers $r = 10$ and $r - 1 = 9$, so the ***9's*** complement of ***N*** is $(10^n - 1) - N$. Now, $10^n$ represents a number that consists of a single ***1*** followed by ***n 0s***. $10^n - 1$ is a number represented by ***n 9s***. For example, with $n = 4$ we have $10^4 = 10000$ and $10^4 - 1 = 9999$. It follows that the ***9's*** complement of a decimal number is obtained by subtracting each digit from ***9***. For example, the ***9's*** complement of ***546700*** is $999999 - 546700 = 453299$ and the ***9's*** complement of ***12389*** is $99999 - 12389 = 87610$. For binary numbers, $r = 2$ and $r - 1 = 1$, so the 1‘s complement of ***N*** is $(2^n - 1) - N$. Again, $2^n$ is represented by a binary number that consists of a ***1*** followed by ***n 0s***.

$2^n - 1$ is a binary number represented by ***n 1s***. For example, with $n = 4$, we have $2^4 = (10000)_2$ and $2^4 - 1 = (1111)_2$. Thus the ***1's*** complement of a binary number is obtained by subtracting each digit from ***1***. However, the subtraction of a binary digit from ***1*** causes the bit to change from ***0 to 1*** or from ***1 to 0***. Therefore, the ***1's*** complement of a binary number is formed by changing ***1s into 0s*** and ***0s into 1s***. For example, the ***1's*** complement of ***1011001*** is ***0100110*** and the ***1's*** complement of ***0001111*** is ***1110000.*** The ***(r – 1)'s*** complement of octal or hexadecimal numbers are obtained by subtracting each digit from ***7 or F*** (decimal 15) respectively.

### (r ’s) Complement

The ***r ’s*** complement of an ***n*** digit number ***N*** in base ***r*** is defined as $\boldsymbol{r^{n} - N}$ for ***N ≠ 0*** and ***0*** for ***N = 0***. Comparing with the ***(r – 1)’s*** complement, we note that the ***r ’s*** complement is obtained by adding ***1*** to the ***(r – 1)’s*** complement since $\boldsymbol{r^{n} - N = [(r^{n} - 1) - N] + 1}$. Thus the ***10’s*** complement of the decimal ***2389*** is ***7610 + 1 = 7611*** and is obtained by adding ***1*** to the ***9’s*** complement value. The ***2’s*** complement of binary ***101100*** is ***010011 + 1 = 010100*** and is obtained by adding ***1*** to the ***1’s*** complement value. Since $\boldsymbol{10^{n}}$ is a number represented by a ***1*** followed by ***n 0s***, then $\boldsymbol{10^{n} - N}$, which is the ***10’s*** complement of ***N***, can be formed also by leaving all least significant ***0s*** unchanged, subtracting the first non-zero least significant digit from ***10***, and then subtracting all higher significant digits from ***9***. The ***10’s*** complement of ***246700*** is ***753300*** and is obtained by ***leaving the two zeros unchanged***, subtracting ***7 from 10***, and subtracting the ***other three digits from 9***. Similarly, the ***2’s*** complement can be formed by leaving all least significant ***0’s*** and the first ***1*** unchanged, and then replacing ***1s by 0s*** and ***0s by 1s*** in all other higher significant bits. The ***2’s*** complement of ***1101100*** is ***0010100*** and is obtained by ***leaving the two low-order 0s and the first 1 unchanged,*** and then replacing ***1s by 0s*** and ***0s by 1s*** in the other four most significant bits. In the definitions above it was assumed that the numbers do not have a radix point. If the original number ***N*** contains a radix point, it should be removed temporarily to form the ***r’s*** or ***(r-1)’s*** complement. The radix point is then restored to the complemented number in the same relative position. It is also worth mentioning that the complement of the complement restores the number to its original value. The ***r’s*** omplement of ***N*** is $\boldsymbol{r^{n} - N}$. The complement of the complement is $\boldsymbol{r^{n} - (r^{n} - N) = N}$ giving back the original number.

### Subtraction of Unsigned Numbers

The direct method of subtraction taught in elementary schools uses the borrow concept. In this method we borrow a 1 from a higher significant position when the minuend digit is smaller than the corresponding subtrahend digit. This seems to be easiest when people perform subtraction with paper and pencil. When subtraction is implemented with digital hardware, this method is found to be less efficient than the method that uses complements.

The subtraction of two ***n*** digit unsigned numbers ***M – N ( N ≠ 0)*** in base ***r*** can be done as follows:

1. ***Add the minuend M to the r’s complement of the subtrahend N.***

   ***This performs*** $\boldsymbol{M + (r^{n} - N) = M - N + r^{n}}$.
2. ***If M ≥ N, the sum will produce an end carry*** $\boldsymbol{r^{n}}$ ***which is discarded, and what is left is the result M – N.***
3. ***If M < N, the sum does not produce an end carry and is equal to*** $\boldsymbol{r^{n} -}$

***(N – M), which is the r’s complement of (N – M). To obtain the answer in a familiar form, take the r’s complement of the sum and place a negative sign in front.***

Consider, for example, the subtraction 72532 – 13250 = 59282. The 10‘s complement of 13250 is 86750.
Therefore:

| | | |
|---|---|---|
| M | = | 72532 |
| 10‘s complement of N | = | +86750 |
| Sum | = | 159282 |
| Discard end carry $10^{5}$ | = | - 100000 |
| Answer | = | 59282 |

Since we are dealing with unsigned numbers, there is really no way to get an unsigned result for the second example. When working with paper and pencil, we recognize that the answer must be changed to a signed negative number. When subtracting with complements, the negative answer is recognized by the absence of the end carry and the complemented result. Subtraction with complements is done with binary numbers in a similar manner using the same procedure outlined above. Using the two binary numbers X = 1010100 and Y = 1000011, we perform the subtraction X – Y and Y – X using 2‘s complements:

| | | |
|---|---|---|
| X | = | 1010100 |
| 2‘s complement of Y | = | +0111101 |
| Sum | = | 10010001 |
| Discard end carry $2^7$ | = | -10000000 |
| Answer: X – Y | = | 0010001 |

| | | |
|---|---|---|
| Y | = | 1000011 |
| 2‘s complement of X | = | +0101100 |
| Sum | = | 1101111 |

There is no end carry. Answer is negative 0010001 = 2‘s complement of 1101111.

**Fixed-Point Representation**

Positive integers, including zero, can be represented as unsigned numbers. However, to represent negative integers, we need a notation for negative values. In ordinary arithmetic, a negative number is indicated by a minus sign and a positive number by a plus sign. Because of hardware limitations, computers must represent everything with 1s and 0s, including the sign of a number. As a consequence, it is customary to represent the sign with a bit placed in the leftmost position of the number. The convention is to make the sign bit equal to 0 for positive and to 1 for negative. In addition to the sign, a number may have a ***binary (or decimal) point***. The position of the binary point is needed to represent fractions, integers, or mixed integer-fraction numbers. The representation of the binary point in a register is complicated by the fact that it is characterized by a position in the register. There are two ways of specifying the position of the binary point in a register: by giving it a fixed position or by employing a floating-point representation. The fixed-point method assumes that the binary point is always fixed in one position. The two positions most widely used are (1) a binary point in the extreme left of the register to make the stored number a fraction, and (2) a binary point in the extreme right of the register to make the stored number an integer. In either case, the binary point is not actually present, but its presence is assumed from the fact that the number stored in the register is treated as a fraction or as an integer. The floating-point representation uses a second register to store a number that designates the position of the decimal point in the first register. Floating-point representation is discussed further in the next section.

**Integer Representation**

When an integer binary number is positive, the sign is represented by 0 and the magnitude by a positive binary number. When the number is negative, the sign is represented by 1 but the rest of the number may be represented in one of three possible ways:

1. ***Signed-magnitude representation***
2. ***Signed-1’s complement representation***
3. ***Signed-2’s complement representation***

The signed-magnitude representation of a negative number consists of the magnitude and a negative sign. In the other two representations, the negative number is represented in either the 1‘s or 2‘s complement of its positive value. As an example, consider the signed number 14 stored in an 8-bit register. +14 is represented by a sign bit of 0 in the leftmost position followed by the binary equivalent of 14: 00001110. Note that each of the eight bits of the register must have a value and therefore 0‘s must be inserted in the most significant positions following the sign bit. ***Although there is only one way to represent +14, there are three different ways to represent -14 with eight bits.***

| | | |
|---|---|---|
| ***In signed-magnitude representation*** | ***1*** | ***0001110*** |
| ***In signed-1’s complement representation*** | ***1*** | ***1110001*** |
| ***In signed-2’s complement representation*** | ***1*** | ***1110010*** |

The signed-magnitude representation of -14 is obtained from +14 by complementing only the sign bit. The signed-1‘s complement representation of -14 is obtained by complementing all the bits of +14, including the sign bit. The signed-2‘s complement representation is obtained by taking the 2‘s complement of the positive number, including its sign bit. The signed-magnitude system is used in ordinary arithmetic but is awkward when employed in computer arithmetic. Therefore, the signed-complement is normally used. The 1‘s complement imposes difficulties because it has two representations of 0 (+0 and -0). It is seldom used for arithmetic operations except in some older computers. The 1‘s complement is useful as a logical operation since the change of 1 to 0 or 0 to 1 is equivalent to a logical complement operation.The following discussion of signed binary arithmetic deals exclusively with the signed-2‘s complement representation of negative numbers.

### Arithmetic Addition

The addition of two numbers in the signed-magnitude system follows the rules of ordinary arithmetic. If the signs are the same, we add the two magnitudes and give the sum the common sign. If the signs are different, we subtract the smaller magnitude from the larger and give the result the sign of the larger magnitude. For example, (+25) + (-37) = -(37 – 25) = -12 and is done by subtracting the smaller magnitude 25 from the larger magnitude 37 and using the sign of 37 for the sign of the result. This is a process that requires the comparison of the signs and the magnitudes and then performing either addition or subtraction.

| | | | |
|---|---|---|---|
| +6 | 00000110 | -6 | 11111010 |
| +13 | 00001101 | +13 | 00001101 |
| ----- | ------------- | ----- | ------------- |
| +19 | 00010011 | +7 | 00000111 |
| +6 | 00000110 | -6 | 11111010 |
| -13 | 11110011 | -13 | 11110011 |
| ---- | ------------- | ----- | ------------- |
| -7 | 11111001 | -19 | 11101101 |

By contrast, the rule for adding numbers in the signed-2‘s complement system does not require a comparison or subtraction, only addition and complementation. The procedure is very simple and can be stated as follows: ***Add the two numbers, including their sign bits, and discard any carry out of the sign (leftmost) bit position.*** Numerical examples for addition are shown below. ***Note that negative numbers must initially be in 2's complement and that if the sum obtained after the addition is negative, it is in 2's complement form.***

In each of the four cases, the operation performed is always addition, including the sign bits. Any carry out of the sign bit position is discarded, and negative results are automatically in 2‘s complement form. The complement form of representing negative numbers is unfamiliar to people used to the signed-magnitude system. To determine the value of a negative number when in signed-2‘s complement, it is necessary to convert it to a positive number to place it in a more familiar form. For example, the signed binary number 11111001 is negative because the leftmost bit is 1.Its 2‘s complement is 00000111, which is the binary equivalent of +7. We therefore recognize the original negative number to be equal to -7.

**Arithmetic Subtraction**

Subtraction of two signed binary numbers when negative numbers are in 2‘s complement form is very simple and can be stated as follows: ***Take the 2's complement of the subtrahend (including the sign bit) and add it to the minuend (including the sign bit). A carry out of the sign bit position is discarded.***

This procedure stems from the fact that a subtraction operation can be changed to an addition operation if the sign of the subtrahend is changed. This is demonstrated by the following relationship:

(±A) – (+B) = (±A) + (-B)

(±A) – (-B) = (±A) + (+B)

But changing a positive number to a negative number is easily done by taking its 2‘s complement. The reverse is also true because the complement of a negative number in complement form produces the equivalent positive number. Consider the subtraction of (-6) – (-13) = +7. In binary with eight bits this is written as

11111010– 11110011. The subtraction is changed to addition by taking the 2‘s complement of the subtrahend (-13) to give (+13). In binary this is 11111010 + 00001101 = 100000111. Removing the end carry, we obtain the correct answer 00000111 (+7). It is worth noting that binary numbers in the signed-2‘s complement system are added and subtracted by the same basic addition and subtraction rules as unsigned numbers. Therefore, computers need only one common hardware circuit to handle both types of arithmetic. The user or programmer must interpret the results of such addition or subtraction differently depending on whether it is assumed that the numbers are signed or unsigned.

**Overflow**

When two numbers of n digits each are added and the sum occupies n+1 digits, we say that an overflow has occurred. When the addition is performed with paper and pencil, an overflow is not a problem since there is no limit to the width of the page to write down the sum. An overflow is a problem in digital computers because the width of registers is finite. A result that contains n+1 bits cannot be accommodated in a register with a standard length of n bits. For this reason, many computers detect the occurrence of an overflow, and when it occurs, a corresponding flip-flop is set which can then be checked by the user.The detection of an overflow after the addition of two binary numbers depends on whether the numbers are considered to be signed or unsigned. When two unsigned numbers are added, an overflow is detected from the end carry out of the most significant position. In the case of singed numbers, the leftmost bit always represents the sign, and negative numbers are in 2‘s complement form. When two signed numbers are added, the sign bit is treated as part of the number and the end carry does not indicate an overflow.An overflow cannot occur after an addition if one number is positive and the other is negative, since adding a positive number to a negative number produces a result that is smaller than the larger of the two original numbers. An overflow may occur if the two numbers added are both positive or both negative. To see how this can happen, consider the following example. Two signed binary numbers, +70 and +80, are stored in two 8-bit registers. The range of numbers that each register can accommodate is from binary +127 to binary -128. Since the sum of the two numbers is +150, it exceeds the capacity of the 8-bit register. This is true if the numbers are both positive or both negative. The two additions in binary are shown below together with the last two carries.

| | | | | |
|---|---|---|---|---|
| carries: 0 | 1 | | carries: 1 | 0 |
| +70 | 0 | 1000110 | -70 | 1 0111010 |
| +80 | 0 | 1010000 | -80 | 1 0110000 |
| ------- | ---------------- | | ------- | ---------------- |
| +150 | 1 | 0010110 | -150 | 0 1101010 |

Note that the 8-bit result that should have been positive has a negative sign bit and the 8-bit result that should have been negative has a positive sign bit. If, however, the carry out of the sign bit position is taken as the sign bit of the result, the 9-bit answer so obtained will be correct. Since the answer cannot be accommodated within 8 bits, we say that an overflow occurred. An overflow condition can be detected by observing the carry into the sign bit position and the carry out of the sign bit position. If these two carries are not equal, an overflow condition is produced. This is indicated in the examples where the two carries are explicitly shown. If the two carries are applied to an exclusive-OR gate, an overflow will be detected when the output of the gate is equal to 1.

**Decimal Fixed-Point Representation**

The representation of decimal numbers in registers is a function of the binary code used to represent a decimal digit. A 4-bit decimal code requires four flip-flops for each decimal digit. The representation of 4385 in BCD requires 16 flip-flops, four flip-flops for each digit. The number will be represented in a register with 16 flip-flops as follows:

01000011 1000 0101

By representing numbers in decimal we are wasting a considerable amount of storage space since the number of bits needed to store a decimal number in a binary code is greater than the number of bits needed for its equivalent binary representation. Also, the circuits required to perform decimal arithmetic are more complex. However, there are some advantages in the use of decimal representation because computer input and output data are generated by people who use the decimal system. Some applications, such as business data processing, require small amounts of arithmetic computations compared to the amount required for input and output of decimal data. For this reason, some computers and all electronic calculators perform arithmetic operations directly with the decimal data (in a binary code) and thus eliminate the need for conversion into binary and back to decimal. Some computer systems have hardware for arithmetic calculations with both binary and decimal data.The representation of signed decimal numbers in BCD is similar to the representation of signed numbers in binary. We can either use the familiar signed-magnitude system or the signed-complement system. The sign of a decimal number is usually represented with four bits to conform to the 4-bit code of the decimal digits. It is customary to designate a plus with four 0‘s and a minus with the BCD equivalent of 9, which is 1001. The signed-magnitude system is difficult to use with computers. The signed-complement system can be either the 9‘s or the 10‘s complement, but the 10‘s complement is the one most often used. To obtain the 10‘s complement of a BCD number, we first take the 9‘s complement and then add one to the least significant digit. The 9‘s complement is calculated from the subtraction of each digit from 9. The procedures developed for the signed-2‘s complement system apply also to the signed-10‘s complement system for decimal numbers. Addition is done by adding all digits, including the sign digit, and discarding the end carry. Obviously, this assumes that all negative numbers are in 10‘s complement form.

Consider the addition (+375) + (-240) = +135 done in the signed-10‘s complement system.

| | |
|---|---|
| 0 375 | $(0000\ 0011\ 0111\ 0101)_{BCD}$ |

| | |
|---|---|
| +9 760 | $(1001\ 0111\ 0110\ 0000)_{BCD}$ |
| -------- | ---------------------------------<br>- |
| 0 135 | $(0000\ 0001\ 0011\ 0101)_{BCD}$ |

The 9 in the leftmost position of the second number indicates that the number is negative. 9760 is the 10‛s complement of 0240. The two numbers are added and the end carry is discarded to obtain +135. Of course, the decimal numbers inside the computer must be in BCD, including the sign digits. The addition is done with BCD adders. The subtraction of decimal numbers either unsigned or in the signed-10‛s complement system is the same as in the binary case. Take the 10‛s complement of the subtrahend and add it to the minuend. Many computers have special hardware to perform arithmetic calculations directly with decimal numbers in BCD. The user of the computer can specify by programmed instructions that the arithmetic operations be performed with decimal numbers directly without having to convert them to binary.

**Floating-Point Representation**

The floating-point representation of a number has two parts. The first part represents a signed, fixed-point number called the ***mantissa***. The second part designates the position of the decimal (or binary) point and is called the ***exponent***. The fixed-point mantissa may be a fraction or an integer. For example, the decimal number +6132.789 is represented in floating-point with a fraction and an exponent as follows:

| ***Fraction*** | ***Exponent*** |
|---|---|
| +0.6132789 | +04 |

The value of the exponent indicates that the actual position of the decimal point is four positions to the right of the indicated decimal point in the fraction. This representation is equivalent to the scientific notation +0.6132789 x $10^{+4}$. Floating-point is always interpreted to represent a number in the following form:

$$\boldsymbol{m \times r^{e}}$$

Only the mantissa m and the exponent e are physically represented in the register (including their signs). The radix r and the radix-point position of the mantissa are always assumed. The circuits that manipulate the floating-point numbers in registers conform with these two assumptions in order to provide the correct computational results. A floating-point binary number is represented in a similar manner except that it uses base 2 for the exponent. For example, the binary number +1001.11 is represented with an 8-bit fraction and 6-bit exponent as follows:

| ***Fraction*** | ***Exponent*** |
|---|---|
| 01001110 | 000100 |

The fraction has a 0 in the leftmost position to denote positive. The binary point of the fraction follows the sign bit but is not shown in the register. The exponent has the equivalent binary number +4. The floating-point number is equivalent to

$$\boldsymbol{m \times 2^{e} = +(.1001110)_2 \times 2^{+4}}$$

A floating-point number is said to be ***normalized*** if the most significant digit of the mantissa is nonzero. For example, the decimal number 350 is normalized but 00035 is not. Regardless of where the position of the radix point is assumed to be in the mantissa, the number is normalized only if its leftmost digit is nonzero. For example, the 8-bit binary number 00011010 is not normalized because of the three leading 0s. The number can be normalized by shifting it three positions to the left and discarding the leading 0s to obtain 11010000. The three shifts multiply the number by $2^3 = 8$. To keep the same value for the floating-point number, the exponent must be subtracted by 3. Normalized numbers provide the maximum possible precision for the floating-point number. A zero cannot be normalized because it does not have a nonzero digit. It is usually represented in floating-point by all 0s in the mantissa and exponent. Arithmetic operations with floating-point numbers are more complicated than arithmetic operations with fixed-point numbers and their execution takes longer and requires more complex hardware. However, floating-point representation is a must for scientific computations because of the scaling problems involved with fixed-point computations. Many computers and all electronic calculators have the built-in capability of performing floating-point arithmetic operations. Computers that do not have hardware for floating-point computations have a set of subroutines to help the user program scientific problems with floating-point numbers.

**Other Binary Codes**

In previous sections we introduced the most common types of binary-coded data found in digital computers. Other binary codes for decimal numbers and alphanumeric characters are sometimes used. Digital computers also employ other binary codes for special applications. A few additional binary codes encountered in digital computers are presented in this section.

**Gray Code**

Digital systems can process data in discrete form only. Many physical systems supply continuous output data. The data must be converted into digital form before they can be used by a digital computer. Continuous, or analog, information is converted into digital form by means of an analog-to-digital converter. The reflected binary or ***Gray code***, shown in Table 1.5, is sometimes used for the converted digital data.

| ***Binary code*** | ***Decimal equivalent*** | ***Binary code*** | ***Decimal equivalent*** |
|---|---|---|---|
| 0000 | 0 | 1100 | 8 |
| 0001 | 1 | 1101 | 9 |
| 0011 | 2 | 1111 | 10 |
| 0010 | 3 | 1110 | 11 |
| 0110 | 4 | 1010 | 12 |
| 0111 | 5 | 1011 | 13 |
| 0101 | 6 | 1001 | 14 |
| 0100 | 7 | 1000 | 15 |

**Table 1.5: 4-Bit Gray Code**

Gray codes counters are sometimes used to provide the timing sequence that control the operations in a digital system. A Gray code counter is a counter whose flip-flops go through a sequence of states as specified in Table 1.5. Gray code counters remove the ambiguity during the change from one state of the counter to the next because only one bit can change during the state transition.

**Other Decimal Codes**

Binary codes for decimal digits require a minimum of four bits. Numerous different codes can be formulated by arranging four or more bits in 10 distinct possible combinations. A few possibilities are shown in Table 1.6.

| ***Decimal digit*** | ***BCD 8421*** | ***2421*** | ***Excess-3*** | ***Excess-3 gray*** |
|---|---|---|---|---|
| 0 | 0000 | 0000 | 0011 | 0010 |
| | | | | |
| 1 | 0001 | 0001 | 0100 | 0110 |
| | | | | |
| 2 | 0010 | 0010 | 0101 | 0111 |
| | | | | |
| 3 | 0011 | 0011 | 0110 | 0101 |
| | | | | |
| 4 | 0100 | 0100 | 0111 | 0100 |
| | | | | |
| 5 | 0101 | 1011 | 1000 | 1100 |
| | | | | |
| 6 | 0110 | 1100 | 1001 | 1101 |
| | | | | |
| 7 | 0111 | 1101 | 1010 | 1111 |
| | | | | |
| | | | | |
| 8 | 1000 | 1110 | 1011 | 1110 |
| | | | | |
| 9 | 1001 | 1111 | 1100 | 1010 |
| | | | | |
| Unused | 1010 | 0101 | 0000 | 0000 |
| bit | | | | |
| | 1011 | 0110 | 0001 | 0001 |
| combinations | | | | |
| | 1100 | 0111 | 0010 | 0011 |
| | | | | |
| | 1101 | 1000 | 1101 | 1000 |
| | | | | |
| | 1110 | 1001 | 1110 | 1001 |
| | | | | |
| | 1111 | 1010 | 1111 | 1011 |
| | | | | |

**Fig:1.6 Different Binary Codes for the Decimal Digit**

The BCD (binary-coded decimal) has been introduced before. It uses a straight assignment of the binary equivalent of the digit. The six unused bit combinations listed have no meaning when BCD is used, just as the letter H has no meaning when decimal digit symbols are written down. For example, saying that 1001 110 is a decimal number in BCD is like saying that 9H is a decimal number in the conventional symbol designation. Both cases contain an invalid symbol and therefore designate a meaningless number. One disadvantage of using BCD is the difficulty encountered when the 9‘s complement of the number is to be computed. On the other hand, the 9‘s complement is easily obtained with the 2421 and the excess-3 codes listed in Table 1.6. These two codes have a ***self-complementing*** property which means that the 9‘s complement of a decimal number, when represented in one of these codes, is easily obtained by changing 1‘s to 0‘s and 0‘s to 1‘s. The property is useful when arithmetic operations are done in signed-complement representation. The 2421 is an example of a ***weighted code***. In a weighted code, the bits are multiplied by the weights indicated and the sum of the weighted bits gives the decimal digit. For example, the bit combination 1101, when weighted by the respective digits 2421, gives the decimal equivalent of 2 x 1 + 4 x 1 + 2 x 0 + 1 + 1 = 7. The BCD code can be assigned the weights 8421 and for this reason it is sometimes called the 8421 code. The ***excess-3 code*** is a decimal code that has been used in older computers. This is an unweighted code. Its binary code assignment is obtained from the corresponding BCD equivalent binary number after the addition of binary 3 (0011). From Table 1.5 we note that the Gray code is not suited for a decimal code if we were to choose the first 10 entries in the table. This is because the transition from 9 back to 0 involves a change of three bits (from 1101 to 0000). To overcome this difficulty, we choose the 10 numbers starting from the third entry 0010 up to the twelfth entry 1010. Now the transition from 1010 to 0010 involves a change of only one bit. Since the code has been shifted up three numbers, it is called the excess-3 Gray. This code is listed with the other decimal codes in Table 1.6.

**Other Alphanumeric Codes**

The ASCII code (Table 1.4) is the standard code commonly used for the transmission of binary information. Each character is represented by a 7-bit code and usually an eighth bit is inserted for parity. The code consists of 128 characters. Ninety-five characters represent *graphic symbols* that include upper- and lowercase letters, numerals zero to nine, punctuation marks, and special symbols. Twenty-three characters represent format effectors, which are functional characters for controlling the layout of printing or display devices such as carriage return, line feed, horizontal tabulation, and back space.The other 10 characters are used to direct the data communication flow and report its status. Another alphanumeric (sometimes called alphanumeric code used in IBM equipment is the ***EBCDIC (Extended BCD Interchange Code)***. It uses eight bits for each character (and a ninth bit for parity). EBCDIC has the same character symbols as ASCII but the bit assignment to characters is different. When alphanumeric characters are used internally in a computer for data processing (not for transmission purpose) it is more convenient to use a 6-bit code to represent 64-characters. A 6-bit code can specify the 26 uppercase letters of the alphabet, numerals zero to nine, and up to 28 special characters. This set of characters is usually sufficient for data-processing purposes. Using fewer bits to code characters has the advantage of reducing the memory space needed to store large quantities of alphanumeric data.

## Register Transfer and Micro-operations

### Introduction

A digital system is an interconnection of digital hardware modules that accomplish a specific information-processing task. Digital systems vary in size and complexity, from a few integrated circuits to a complex of interconnected and interacting digital computers. Digital system design invariably uses a modular approach. The modules are constructed from such digital components as registers, decoders, arithmetic elements, and control logic. The various modules are interconnected with common data and control paths to form a digital computer system.

### Objectives:

After studying this unit, the learner will be able to

□ Explain Register Transfer Language

□ Understand bus selection techniques

□ Explain arithmetic and logical circuits

□ Understand the operation of arithmetic logic shift unit

### Register Transfer Language

Digital modules are best defined by the registers they contain and the operations that are performed on the data stored in them. The operations executed on data stored in registers are called ***microoperations***. A microoperation is an elementary operation performed on the information stored in one or more registers. The result of the operation may replace the previous binary information of a register or may be transferred to another register. Examples of microoperations are shift, count, clear, and load.

The internal hardware organization of a digital computer is best defined by specifying:

***1. The set of registers it contains and their function.***

2. ***The sequence of microoperations performed on the binary information stored in the registers.***
3. **The control that initiates the sequence of microoperations.**

It is possible to specify the sequence of microoperations in a computer by explaining every operation in words, but this procedure usually involves a lengthy descriptive explanation. It is more convenient to adopt a suitable symbology to describe the sequence of transfers between registers and the various arithmetic and logic microoperations associated with the transform. The use of symbols instead of a narrative explanation provides an organized and concise manner for listing the microoperation sequences in registers and the control functions that initiate them.

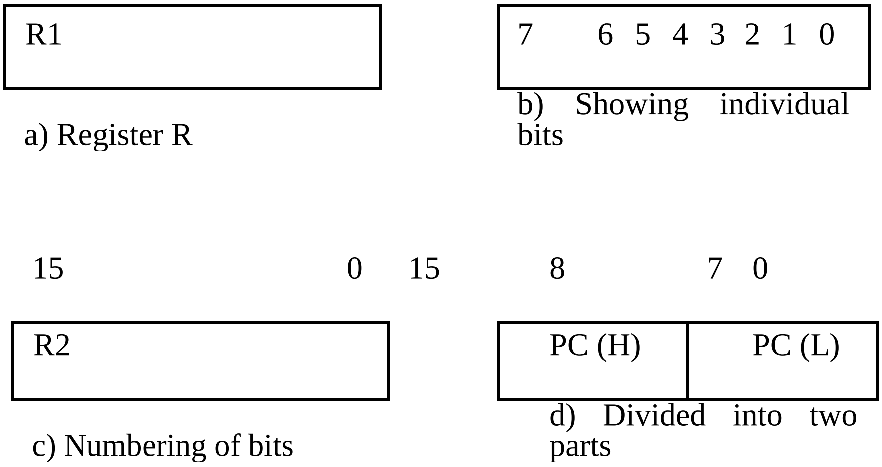
The symbolic notation used to describe the microoperation transfers among registers is called a ***register transfer language***. The term ―register transfer‖ implies the availability of hardware logic circuits that can perform a stated

microoperation and transfer the result of the operation to the same or another register. The word ―language‖ is borrowed from programmers, who apply this term to programming languages. A programming language is a procedure for writing symbols to specify a given computational process. Similarly, a natural language such as English is a system for writing symbols and combining them into words and sentences for the purpose of communication between people. A register transfer language is a system for expressing in symbolic form the microoperation sequences among the registers of a digital module. It is a convenient tool for describing the internal organization of digital computers in concise and precise manner. It can also be used to facilitate the design process of digital systems.

The register transfer language adopted here is believed to be as simple as possible, so it should not take very long to memorize. We will proceed to define symbols for various types of microoperations, and at the same time, describe associated hardware that can implement the stated microoperations. The symbolic designation introduced in this chapter will be utilized in subsequent chapters to specify the register transfers, the microoperations, and the control functions that describe the internal hardware organization of digital computers.

### Register Transfer

Computer registers are designated by capital letters (sometimes followed by numerals) to denote the function of the register. For example, the register that holds an address for the memory unit is usually called a ***memory address register*** and is designated by the name ***MAR***. Other designations for registers are ***PC*** (for ***program counter***), ***IR*** (for ***instruction register***), and ***R1*** (for ***processor register***). The individual flip-flops in an *n-bit* register are numbers in sequence from *0* through *n-1*, starting from *0* in the rightmost position and increasing the numbers toward the left. Fig.2.1 shows the representation of registers in block diagram form. The most common way to represent a register is by a rectangular box with the name of the register inside, as in Fig.2.1 (a). The individual bits can be distinguished as in (b). The numbering of bits in a 16-bit register can be marked on top of the box as shown in (c). A 16-bit register is partitioned into two parts in (d). Bits 0 through 7 are assigned the symbol *L* (for low byte) and bits 8 through 15 are assigned the symbol *H* (for high byte). The name of the 16-bit register is PC. The symbol ***PC (0-7)*** or ***PC (L)*** refers to the low-order byte and ***PC (8-15)*** or ***PC (H)*** to the high-order byte.



a) Register R

b) Showing individual bits

c) Numbering of bits

d) Divided into two parts

**Fig. 2.1: Block diagram of register**

Information transfer from one register to another is designated in symbolic form by means of a replacement operator. The statement

□ **R2 R1**

denotes a transfer of the content of register R1 into register R2. It designates a replacement of the content of R2 by the content of R1. By definition, the content of the source register R1 does not change after the transfer.

A statement that specifies a register transfer implies that circuits are available from the outputs of the source register to the inputs of the destination register and that the designation register has a parallel load capability. Normally, we want the transfer to occur only under a predetermined control condition. This can be shown by means of an *if-then* statement.

□

***If* (P = 1) *then* R**
**(R2 1)**

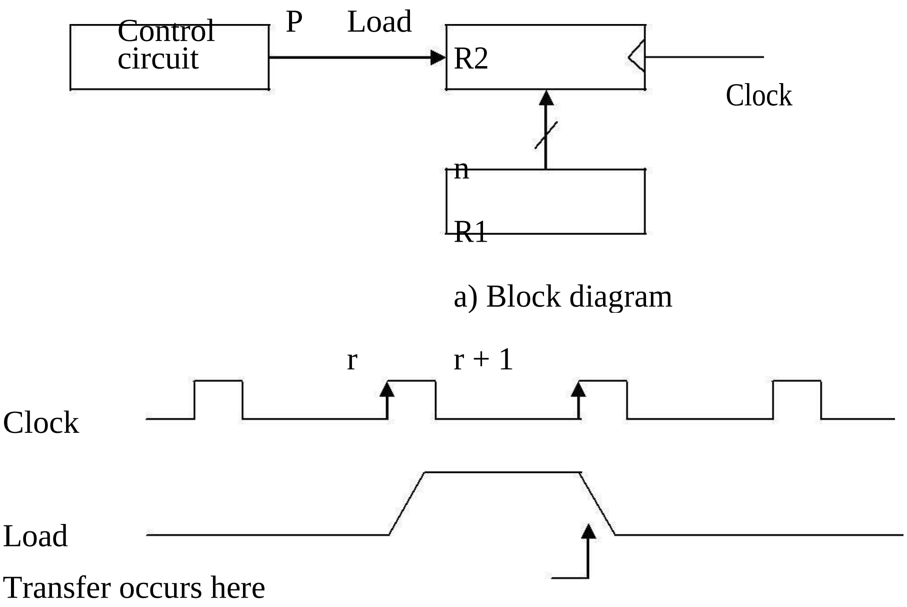
where P is a control signal generated in the control section. It is sometimes convenient to separate the control variables from the register transfer operation by specifying a ***control function***. A control function is a Boolean variable that is equal to 1 or 0. The control function is included in the statement as follows:

□

**P: R2 R1**

The control condition is terminated with a colon. It symbolizes the requirement that the transfer operation be executed by the hardware only if P = 1.

Every statement written in a register transfer notation implies a hardware construction for implementing the transfer. Fig.2.2 shows the block diagram that depicts the transfer from R1 to R2. The *n* outputs of register R1 are connected to the *n* inputs of register R2. The letter *n* will be used to indicate any number of bits for the register. It will be replaced by an actual number when the length of the register is known. Register R2 has a load input that is activated by the control variable P. It is assumed that the control variable is synchronized with the same clock as the one applied to the register. As shown in the timing diagram (b), P is activated in the control section by the rising edge of a clock pulse at time *t*. The next positive transition of the clock at time *t + 1* finds the load input active and the data inputs of R2 are then loaded into the register in parallel. P may go back to 0 at time *t + 1*; otherwise, the transfer will occur with every clock pulse transition while P remains active.



a) Block diagram

b) Timing diagram

**Fig. 2.2: Transfer from R1 to R2 when P = 1**

Note that the clock is not included as a variable in the register transfer statements. It is assumed that all transfers occur during a clock edge transition. Even though the control condition such as P becomes active just after time *t*, the actual transfer does not occur until the register is triggered by the next positive transition of the clock at time *t + 1*.

The basic symbols of the register transfer notation are listed in Table 2.1. Registers are denoted by capital letters, and numerals may follow the letters. Parentheses are used to denote a part of a register by specifying the range of bits or by giving a symbol name to a portion of a register. The arrow denotes a transfer of information and the direction of transfer. A comma is used to separate two or more operations that are executed at the same time. The statement

**T: R2 □ R1, R1 □ R2**

denotes an operation that exchanges the contents of two registers during one common clock pulse provided that T = 1. This simultaneous operation is possible with registers that have edge-triggered flip-flops.

| ***Symbol*** | ***Description*** | ***Examples*** |
|---|---|---|
| Letters | Denotes a register | MAR, R2 |

| | | |
|---|---|---|
| (and numerals) | | |
| Parentheses ( ) | Denotes a part of a register | R2(0-7), R2(L) |
| Arrow | Denotes transfer of information | R<br>2 R1 |
| Comma, | Separates two microoperations | R<br>2 R1, R1R2 |

**Table 2.1: Basic Symbols for Register Transfers**

**Bus and Memory Transfers**

A typical digital computer has many registers, and paths must be provided to transfer information from one register to another. The number of wires will be excessive if separate lines are used between each register and all other registers in the system. A more efficient scheme for transferring information between registers in a multiple-register configuration is a ***common bus system***. A bus structure consists of a set of common lines, one for each bit of a register, through which binary information is transferred one at a time. Control signals determine which register is selected by the bus during each particular register transfer.

One way of constructing a common bus system is with ***multiplexers***. The multiplexers select the source register whose binary information is then placed on the bus. The construction of a bus system for four registers is shown in Fig.2.3. Each register has four bits, numbered 0 through 3. The bus consists of four 4 x 1 multiplexers each having four data inputs, 0 through 3, and two selection inputs, S1 and S0. In order not to complicate the diagram with 16 lines crossing each other, we use labels to show the connections from the outputs of the registers to the inputs of the multiplexers. For example, output 1 of register A is connected to input 0 of MUX 1 because this input is labeled A1. The diagram shows that the bits in the same significant position in each register are connected to the data inputs of one multiplexer to form one line of the bus. Thus MUX 0 multiplexes the four 0 bits of the registers, MUX 1 multiplexes the four 1 bits of the registers, and similarly for the other two bits.

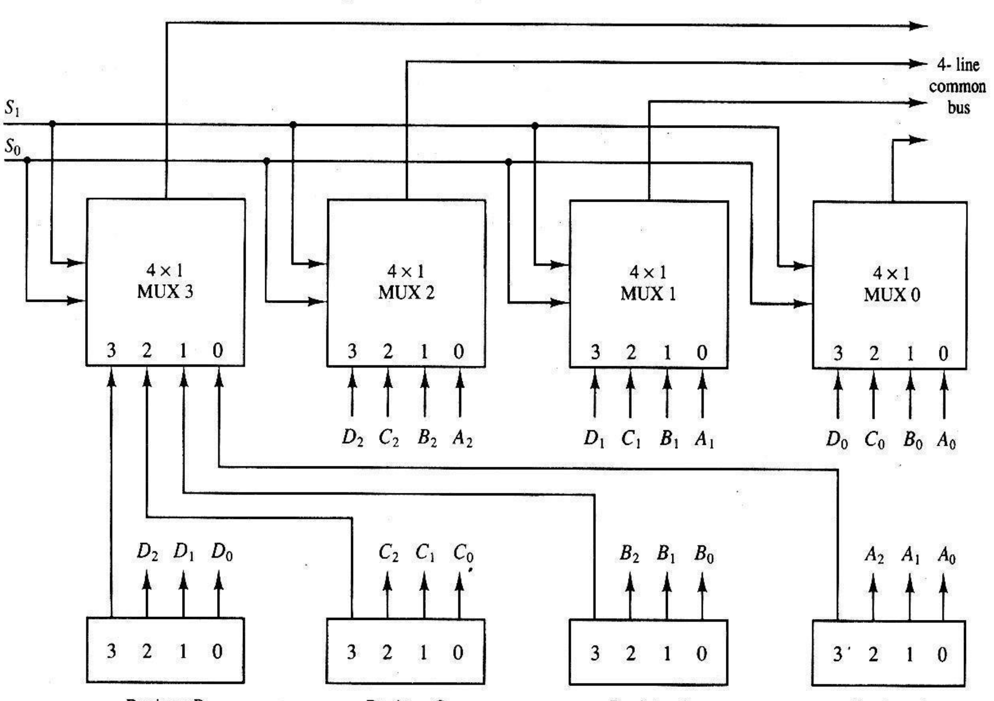


**Fig. 2.3: Bus system for four registers**

The two ***selection lines*** S1 and S0 are connected to the selection inputs of all four multiplexers. The selection lines choose the four bits of one register and transfer them into the four-line common bus. When S1S0 = 00, the 0 data inputs of all four multiplexers are selected and applied to the outputs that form the bus. This causes the bus lines to receive the content of register A since the outputs of this register are connected to the 0 data inputs of the multiplexers. Similarly, register B is selected if S1S0 = 01, and so on. Table 2.2 shows the register that is selected by the bus for each of the four possible binary values of the selection lines.

| ***1*** | ***S0*** | ***Register selected*** |
|---|---|---|
| 0 | 0 | A |
| 0 | 1 | B |
| 1 | 0 | C |
| 1 | 1 | D |

**Table 2.2: Function Table for Bus of Fig. 2.3**

In general, a bus system will multiplex *k* registers of *n* bits each to produce an *n*-line common bus. The number of multiplexers needed to construct the bus is equal to *n*, the number of bits in each register. The size of each multiplexer must be *k x 1* since it multiplexes *k* data lines. For example, a common bus for eight registers of 16 bits each requires 16 multiplexers, one for each line in the bus. Each multiplexer must have eight data input lines and three selection lines to multiplex one significant bit in the eight registers.

The transfer of information from a bus into one of many destination registers can be accomplished by connecting the bus lines to the inputs of all destination registers and activating the load control of the particular destination register selected. The symbolic statement for a bus transfer may mention the bus or its presence may be implied in the statement. When the bus is included in the statement, the register transfer is symbolized as follows:

**BUS □ C, R1 □ BUS**

The content of register C is placed on the bus, and the content of the bus is loaded into register R1 by activating its load control input. If the bus is known to exist in the system, it may be convenient just to show the direct transfer.

**R1 □ C**

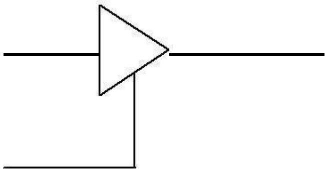
From this statement the designer knows which control signals must be activated to produce the transfer through the bus.

### Three-State Bus Buffers

A bus system can be constructed with ***three-state gates*** instead of multiplexers. A three-state gate is a digital circuit that exhibits three states. Two of the states are signals equivalent to logic 1 and 0 as in a conventional gate. The third state is a ***high-impedance state***. The high-impedance state behaves like an open circuit, which means that the output is disconnected and does not have a logical significance. Three-state gates may perform any conventional logic, such as AND or NAND. However, the one most commonly used in the design of a bus system is the buffer gate.

The graphic symbol of a ***three-state buffer*** gate is shown in Fig.2.4. It is distinguished from a normal buffer by having both a normal input and a control input. The control input determines the output state. When the control input is equal to 1, the output is enabled and the gate behaves like any conventional buffer, with the output equal to the normal input. When the control input is 0, the output is disabled and the gate goes to a high-impedance state, regardless of the value in the normal input. The high-impedance state of a three-state gate provides a special feature not available in other gates. Because of this feature, a large number of three-state gate outputs can be connected with wires to form a common bus line without endangering loading effects.

Normal input A



Output Y=A if C=1

High-impedance if C=0

Control input C

**Fig. 2.4: Graphic symbols for three-state buffer**

The construction of a ***bus system*** with three-state buffers is demonstrated in Fig.2.5. The outputs of four buffers are connected together to form a single bus line. (It must be realized that this type of connection cannot be done with gates that do not have three-state outputs). The control inputs to the buffers determine which of the four normal inputs will communicate with the bus line. No more than one buffer may be in the active state at any given time. The connected buffers must be controlled so that only one three-state buffer has access to the bus line while all other buffers are maintained in a high-impedance state.

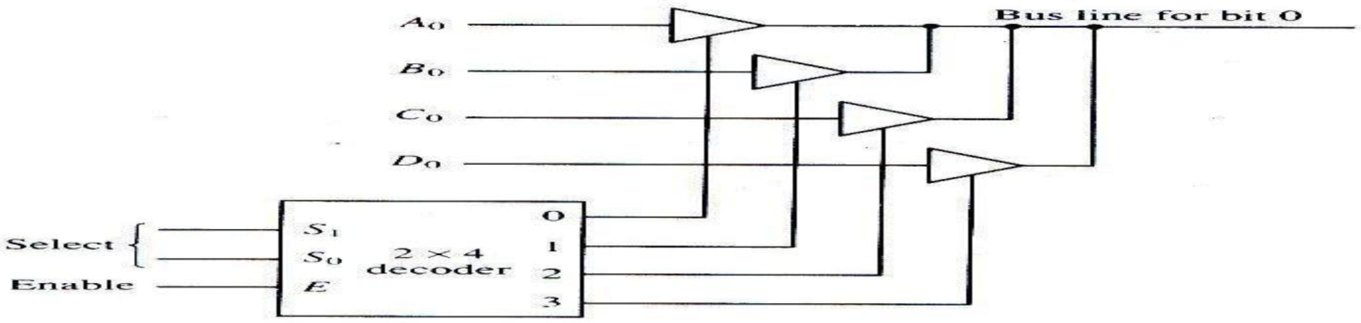


**Fig. 2.5: Bus line with three state-buffers**

One way to ensure that no more than one control input is active at any given time is to use a decoder, as shown in the diagram. When the enable input

**Write: M[AR]R1**

This causes a transfer of information from R1 into the memory word M selected by the address in AR.

**Arithmetic Microoperations**

**A microoperation is an elementary operation performed with the data stored in registers.**

The microoperations most often encountered in digital computers are classified into four categories:

1. **Register transfer microoperations** transfer binary information from one register to another.
2. **Arithmetic microoperations** perform arithmetic operations on numeric data stored in registers.
3. **Logic microoperations** perform bit manipulation operations on non-numeric data stored in registers.
4. **Shift microoperations** perform shift operations on data stored in registers.

The register transfer microoperation was introduced in Sec.2.3. This type of microoperation does not change the information content when the binary information moves from the source register to the destination register. The other three types of microoperations change the information content during the transfer. In this section we introduce a set of arithmetic microoperations. In the next two sections we present the logic and shift microoperations.

The basic arithmetic microoperations are addition, subtraction, increment, decrement, and shift. Arithmetic shifts are explained later in conjunction with the shift microoperations. The arithmetic microoperation defined by the statement

□

**R3R1 + R2**

specifies an add microoperation. It states that the contents of register R1 are added to the contents of register R2 and the sum transferred to register R3. To implement this statement with hardware we need three registers and the digital component that performs the addition operation. The other basic arithmetic microoperations are listed in Table 2.3. Subtraction is most often implemented through complementation and addition. Instead of using the minus operator, we can specify the subtraction by the following statement:

□

**R3R1 + R2 + 1**

R2 is the symbol for the 1's complement of R2. Adding 1 to the 1's complement produces the 2's complement. Adding the contents of R1 to the 2's complement of R2 is equivalent to R1 – R2.

| ***Symbolic designation*** | ***Description*** |
|---|---|
| R3 □ R1 + R2<br>R3 □ R1 – R2 | Contents of R1 plus R2 transferred to R3<br>Contents of R1 minus R2 transferred to R3 |
| R2 □ R2 | Complement the contents of R2 (1's complement) |

| | |
|---|---|
| | |
| R 2 □ $\overline{R2}$ + 1 | 2's complement the contents of R2 (negate) |
| | |
| R 3 □ R1 + $\overline{R2}$ + 1<br>R 1 □ R1 + 1 | R1 plus the 2's complement of R2 (subtraction)<br>Increment the contents of R1 by one |
| R □ | |
| 1 R1 – 1 | Decrement the contents of R1 by one |

**Table 2.3: Arithmetic Microoperations**

The increment and decrement microoperations are symbolized by plus-one and minus-one operations, respectively. These microoperations are implemented with a combinational circuit or with a binary up-down counter.

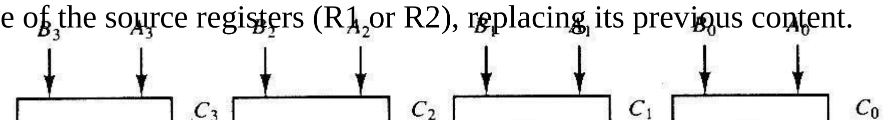
The arithmetic operations of multiply and divide are not listed in Table 2.3. These two operations are valid arithmetic operations but are not included in the basic set of microoperations. The only place where these operations can be considered as microoperations is in a digital system, where they are implemented by means of a combinational circuit. In such a case, the signals that perform these operations propagate through gates, and the result of the operation can be transferred into a destination register by a clock pulse as soon as the output signal propagates through the combinational circuit. In most computers, the multiplication operation is implemented with a sequence of add and shift microoperations. Division is implemented with a sequence of subtract and shift microoperations. To specify the hardware in such a case requires a list of statements that use the basic microoperations of add, subtract and shift.

## Binary Adder

To implement the add microoperation with hardware, we need the registers that hold the data and the digital component that performs the arithmetic addition. The digital circuit that forms the arithmetic sum of two bits and a previous carry is called a full-adder. The digital circuit that generates the arithmetic sum of two binary numbers of any lengths is called a binary adder. The binary adder is constructed with full-adder circuits connected in cascade, with the output carry from one full-adder connected to the input carry of the next full-adder. Fig.2.6 shows the interconnections of four full-adders (FA) to provide a 4-bit binary adder. The augend bits of A and the addend bits of B are designated by subscript numbers from right to left, with subscript 0 denoting the low-order bit. The carries are connected in a chain through the full-adders. The input carry to the binary adder is C0 and the output carry is C4. The S outputs of the full-adders generate the required sum bits.

**Fig. 2.6: 4-bit binary adder**

An n-bit binary adder requires n full-adders. The output carry from each full-adder is connected to the input carry of the next-high-order full-adder. The n data bits for the A inputs come from one register (such as R1), and the n data bits for the B inputs come from another register (such as R2). The sum can be transferred to a third register or to one of the source registers (R1 or R2), replacing its previous content.



## Binary Adder-Subtractor

The subtraction of binary numbers can be done most conveniently by means of complements. Remember that the subtraction, A – B can be done by taking the 2's complement of B and adding it to A. The 2's complement can be obtained by taking the 1's complement and adding one to the least significant pair of bits. The 1's complement can be implemented with inverters and a one can be added to the sum through the input carry.The addition and subtraction operations can be combined into one common circuit by including an exclusive-OR gate with each full-adder. A 4-bit adder-subtractor circuit is shown in Fig.2.7. The mode input M controls the operation. When M = 0 the circuit is an adder and when M = 1 the circuit becomes a subtractor. Each exclusive-OR gate receives input M and one of the inputs of B. When M = 0, we have B □ 0 = B. The full-adders receive the value of B, the input carry is 0, and the circuit performs A plus B. When M = 1, we have B □ 1 = B' and C0 = 1. The B inputs are all complemented and a 1 is added through the input carry. The circuit performs the operation A plus the 2's complement of B.  For unsigned numbers, this gives A – B if A ≥ B or the 2's complement of (B – A) if A < B. For signed numbers, the result is A – B provided that there is no overflow.

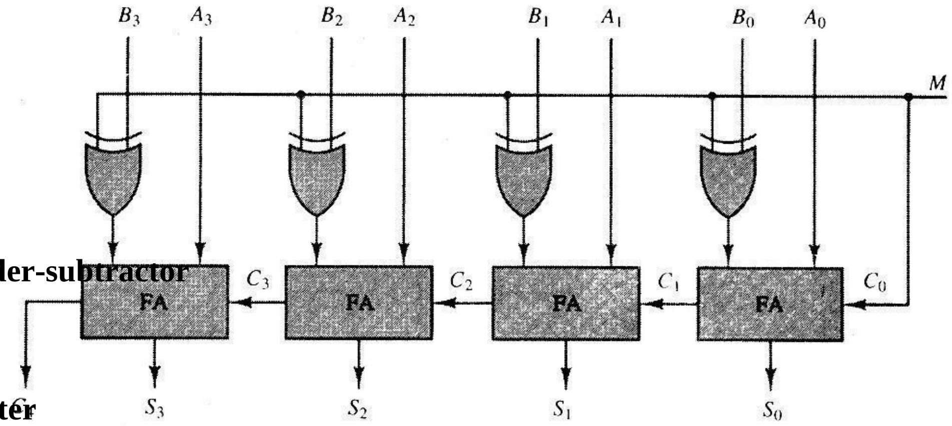


**Fig. 2.7: 4-bit adder-subtractor**

## Binary Incrementer

The increment microoperation adds one to a number in a register. For example, if a 4-bit register has a binary value 0110, it will go to 0111 after it is incremented. This microoperation is easily implemented with a binary counter. Every time the count enable is active, the clock pulse transition increments the content of the register by one. There may be occasions when the increment microoperation must be done with a combinational circuit independent of a particular register. This can be accomplished by means of half-adders connected in cascade.

The diagram of a 4-bit combinational circuit incrementer is shown in Fig.2.8. One of the inputs to the least significant half-adder (HA) is connected to logic-1 and the other input is connected to the least significant bit of the number to be incremented. The output carry from one half-adder is connected to one of the inputs of the next-higher-order half-adder. The circuit receives the four bits from A0 through A3, adds one to it, and generates the incremented output in S0 through S3. The output carry C4 will be 1 only after incrementing binary 1111. This also causes outputs S0 through S3 to go to 0.

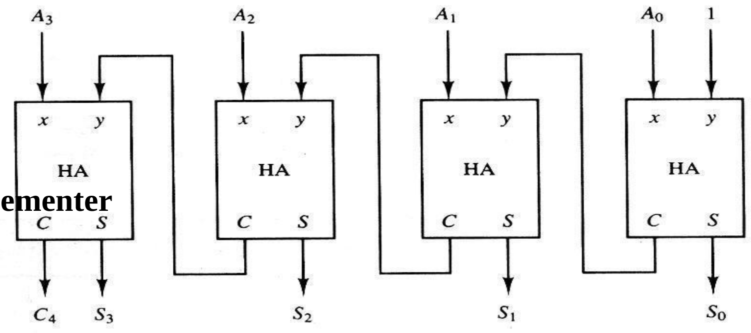


**Fig. 2.8: 4-bit binary incrementer**

The circuit of Fig.2.8 can be extended to an n-bit binary incrementer by extending the diagram to include n half-adders. The least significant bit must have one input connected to logic-1. The other inputs receive the number to be incremented or the carry from the previous stage.

**Arithmetic Circuit**

The arithmetic microoperations listed in Table 2.3 can be implemented in one composite arithmetic circuit. The basic component of an arithmetic circuit is the parallel adder. By controlling the data inputs to the adder, it is possible to obtain different types of arithmetic operations.The diagram of a 4-bit arithmetic circuit is shown in Fig.2.9. It has four full-adder circuits that constitute the 4-bit adder and four multiplexers for choosing different operations. There are two 4-bit inputs A and B and a 4-bit output D. The four inputs from A go directly to the X inputs of the binary adder. Each of the four inputs from B is connected to the data inputs of the multiplexers. The multiplexers data inputs also receive the complement of B. The other two data inputs are connected to logic-0 and logic-1. Logic-0 is a fixed voltage value (0 volts for TTL integrated circuits) and the logic-1 signal can be generated through an inverter whose input is 0. The four multiplexers are controlled by two selection inputs, S1 and S0. The input carry Cin goes to the carry input of the FA in the least significant position. The other carries are connected from one stage to the next.

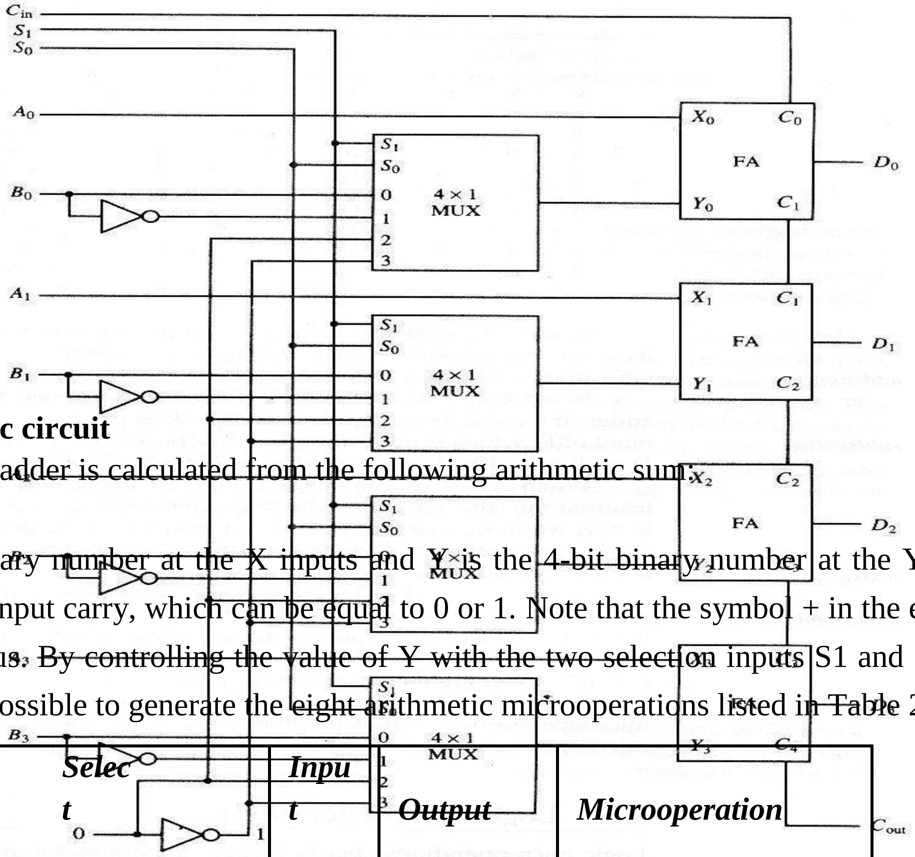


**Fig. 2.9: 4-bit arithmetic circuit**

The output of the binary adder is calculated from the following arithmetic sum:

**D = A + Y + Cin**

where A is the 4-bit binary number at the X inputs and Y is the 4-bit binary number at the Y inputs of the binary adder. Cin is the input carry, which can be equal to 0 or 1. Note that the symbol + in the equation above denotes an arithmetic plus. By controlling the value of Y with the two selection inputs S1 and $S_0$ and making $C_{in}$ equal to 0 or 1, it is possible to generate the eight arithmetic microoperations listed in Table 2.4.

| ***Select*** | | | ***Input*** | ***Output*** | ***Microoperation*** |
|---|---|---|---|---|---|
| ***S*1** | ***S*0** | ***C*in** | ***Y*** | ***D=A+Y+ Cin*** | |
| 0 | 0 | 0 | B | D=A+B | Add |
| 0 | 0 | 1 | B | D=A+B+ | Add with carry |

| | | | | | |
|---|---|---|---|---|---|
| | | | | 1 | |
| 0 | 1 | 0 | $\overline{B}$ | $D=A+\overline{B}$ | Subtract with borrow |
| 0 | 1 | 1 | $\overline{B}$ | $D=A+\overline{B}+1$ | Subtract |
| 1 | 0 | 0 | 0 | D=A | Transfer A |
| 1 | 0 | 1 | 0 | D=A+1 | Increment A |
| 1 | 1 | 0 | 1 | D=A-1 | Decrement A |
| 1 | 1 | 1 | 1 | D=A | Transfer A |

**Table 2.4: Arithmetic Circuit Function Table**

When S1S0 = 00, the value of B is applied to the Y inputs of the adder. If Cin0, the output D = A + B. If Cin = 1, output D = A + B + 1. Both cases perform the add microoperation with or without adding the input carry.

When S1S0 = 01, the complement of B is applied to the Y inputs of the adder. If Cin = 1, then D = A + B + 1. This produces A plus the 2's complement of B, which is equivalent to a subtraction of A – B. When Cin = 0, then D = A + B . This is equivalent to a subtract with borrow, that is, A – B– 1.When S1S0 = 10, the inputs from B are neglected, and instead, all 0's are inserted into the Y inputs. The output becomes D = A + 0 + Cin. This gives D=A when Cin = 0 and D = A + 1 when Cin = 1. In the first case we have a direct transfer from input A to output D. In the second case, the value of A is incremented by 1.

When S1S0 = 11, all 1's are inserted into the Y inputs of the adder to produce the decrement operation D = A – 1 when Cin = 0. This is because a number with all 1's is equal to the 2's complement of 1 (the 2's complement of binary 0001 is 1111). Adding a number A to the 2's complement of 1 produces F = A + 2's complement of 1 = A – 1. When Cin = 1, then D = A – 1 + 1 = A, which causes a direct transfer from input A to output D. Note that the microoperation D = A is generated twice, so there are only seven distinct microoperations in the arithmetic circuit.

**Logic Microoperations**

Logic microoperations specify binary operations for strings of bits stored in registers. These operations consider each bit of the register separately and treat them as binary variables. For example, the exclusive-OR microoperation with the contents of two registers R1 and R2 is symbolized by the statement

**P: R1 □ R1 □ R2**

It specifies a logic microoperation to be executed on the individual bits of the registers provided that the control variable P = 1. As a numerical example, assume that each register has four bits. Let the content of R1 be 1010 and the content of R2 be 1100. The exclusive-OR microoperation stated above symbolizes the following logic computation:

| | |
|---|---|
| **1010** | **Content of R1** |
| **1100** | **Content of R2** |
| **0110** | **Content of R1 after P = 1** |

The content of R1, after the execution of the microoperation, is equal to the bit-by-bit exclusive-OR operation on pairs of bits in R2 and previous values of R1. The logic microoperations are seldom used in scientific computations, but they are very useful for bit manipulation of binary data and for making logical decisions.

Special symbols will be adopted for the logic microoperations OR, AND, and complement, to distinguish them from the corresponding symbols used to express Boolean functions. The symbol V will be used to denote an OR microoperation and the symbol ∧ to denote an AND microoperation. The complement microoperation is the same as the 1's complement and uses a bar on top of the symbol that denotes the register name. By using different symbols, it will be possible to differentiate between a logic microoperation and a control (or Boolean) function. Another reason for adopting two sets of symbols is to be able to distinguish the symbol +, when used to symbolize an arithmetic plus, from a logic OR operation. Although the + symbol has two meanings, it will be possible to distinguish between them by noting where the symbol occurs. When the symbol + occurs in a microoperation, it will denote an arithmetic plus. When it occurs in a control (or Boolean) function, it will denote an OR operation. We will never use it to symbolize an OR microoperation. For example, in the statement

**P + Q: R1R2 + R3, R4R5 V R6**

the + between P and Q is an OR operation between two binary variables of a control function. The + between R2 and R3 specifies an add microoperation. The OR microoperation is designated by the symbol V between registers R5 and R6.

**List of Logic Microoperations**

There are 16 different logic binary variables. They can operations that can be performed with two be determined from all possible truth tables obtained with two binary variables as shown in Table 2.5. In this table, each of the 16 columns F0 through F15 represents a truth table of one possible Boolean function for the two variables x and y. Note that the functions are determined from the 16 binary combinations that can be assigned to F.

| ***x y*** | ***F*0 *F*1 *F*2 *F*3 *F*4 *F*7 *F*8 *F*9 *F*12 *F*14**<br>***F*5 *F*6 *F*10 *F*11 *F*13 *F*15** |
|---|---|
| 0 0 | 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 |
| 0 1 | 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 |
| 1 0 | 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 |

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |

**Table 2.5: Truth Tables for 16 Functions of Two Variables**

The 16 Boolean functions of two variables x and y are expressed in algebraic form in the first column of Table 2.6. The 16 logic microoperations are derived from these functions by replacing variable x by the binary content of register A and variable y by the binary content of register B. It is important to realize that the Boolean functions listed in the first column of Table 2.6 represent a relationship between two binary variables x and y. The logic microoperations listed in the second column represent a relationship between the binary content of two registers A and B. Each bit of the register is treated as a binary variable and the microoperation is performed on the string of bits stored in the registers.

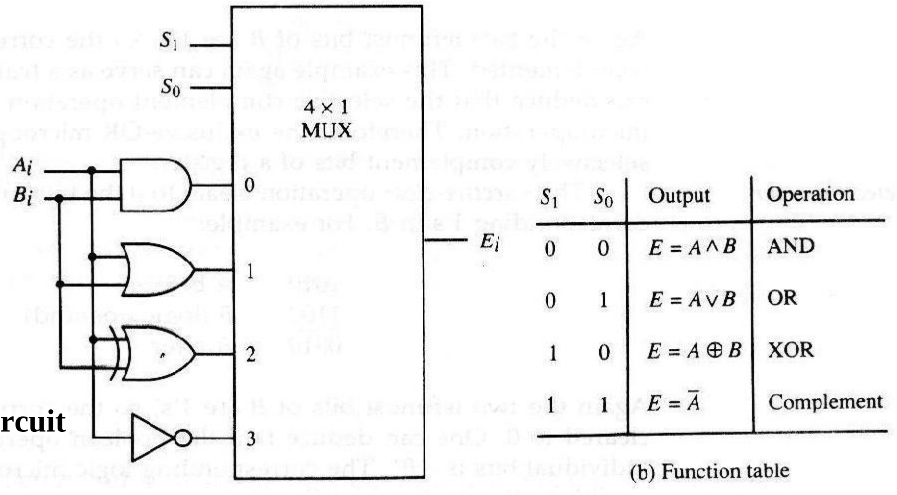
| ***Boolean function*** | ***Microoperation*** | ***Name*** |
|---|---|---|
| F0 = 0 | Ɨ □ 0 | Clear |
| F1 = xy | Ɨ A ∧ B | AND |
| F2 = xy’ | F □ A ∧ $\overline{B}$ | |
| F3 = x | Ɨ A | Transfer A |
| F 4 = x’y | F □ $\overline{A}$ ∧ B | |
| F 5 = y | Ɨ □ B | Transfer B |
| F 6 = x □ y | Ɨ A □ B | Exclusive-OR |
| F7 = x + y | □ FA V B | OR |
| F8 = (x + y)’ | F □ $\overline{AVB}$ | NOR |
| F9 = (x □ y)’ | Ɨ □ $\overline{A □ B}$ | Exclusive-NOR |
| F10 = y’ | F □ $\overline{B}$ | Complement B |
| F11 = x + y’ | F □ A V $\overline{B}$ | |
| F12 = x’ | F □ $\overline{A}$ | Complement A |
| F13 = x’ + y | F □ $\overline{A}$ V B | |

| | | |
|---|---|---|
| F14 = (xy)' | A<br>□<br>F□ B<br>□ | NAND |
| F15 = 1 | I all 1s | Set to all 1s |

**Sixteen Logic Microoperations**

### Hardware Implementation

The hardware implementation of logic microoperations requires that logic gates be inserted for each bit or pair of bits in the registers to perform the required logic function. Although there are 16 logic microoperations, most computers use only four – AND, OR, XOR (exclusive-OR), and complement–from which all others can be derived. shows one stage of a circuit that generates the four basic logic microoperations. It consists of four gates and a multiplexer. Each of the four logic operations is generated through a gate that performs the required logic. The outputs of the gates are applied to the data inputs of the multiplexer. The two selection inputs S1 and S0 choose one of the data inputs of the multiplexer and direct its value to the output. The diagram shows one typical stage with subscript i. For a logic circuit with n bits, the diagram must be repeated n times for I = 0, 1, 2, …, n-1. The selection variables are applied to all stages. The function table in Fig.2.10 (b) lists the logic micro operations obtained for each combination of the selection variables.



| $S_1$ | $S_0$ | Output | Operation |
|---|---|---|---|
| 0 | 0 | $E = A \wedge B$ | AND |
| 0 | 1 | $E = A \vee B$ | OR |
| 1 | 0 | $E = A \oplus B$ | XOR |
| 1 | 1 | $E = \bar{A}$ | Complement |

(b) Function table

(a) Logic diagram

**One stage of logic circuit**

### Some Applications

Logic microoperations are very useful for manipulating individual bits or a portion of a word stored in a register. They can be used to change bit values, delete a group of bits, or insert new bit values into a register. The following examples show how the bits of one register (designated by A) are manipulated by logic microoperations as a function of the bits of another register (designated by B). In a typical application, register A is a processor register and the bits of register B constitute a logic operand extracted from memory and placed in register B.

The selective-set operation sets to 1 the bits in register A where there are corresponding 1's in register B. It does not affect bit positions that have 0's in B. The following numerical example clarifies this operation:

| | |
|---|---|
| **1010** | **A before** |
| **1100** | **B (logic operand)** |

| | |
|---|---|
| **1110** | **A after** |

The two leftmost bits of B are 1’s, so the corresponding bits of A are set to
1. One of these two bits was already set and the other has been changed from 0 to 1. The two bits of A with corresponding 0’s in B remain unchanged. The example above serves as a truth table since it has all four possible combinations of two binary variables. From the truth table we note that the bits of A after the operation are obtained from the logic-OR operation of bits in B and previous values of A. Therefore, the OR micro operation can be used to selectively set bits of a register.

The selective-complement operation complements bits in A where there are corresponding 1’s in B. It does not affect bit positions that have 0’s in B.

For example:

| | |
|---|---|
| **1010** | **A before** |
| **1100** | **B (logic operand)** |
| **0110** | **A after** |

Again the two leftmost bits of B are 1’s, so the corresponding bits of A are complemented. This example again can serve as a truth table from which one can deduce that the selective-complement operation is just an exclusive-OR micro operation. Therefore, the exclusive-OR micro operation can be used to selectively complement bits of a register.

The selective-clear operation clears to 0 the bits in A only where there are corresponding 1’s in B. For example:

| | |
|---|---|
| **1010** | **A before** |
| **1100** | **B (logic operand)** |
| **0010** | **A after** |

Again the two leftmost bits of B are 1’s, so the corresponding bits of A are cleared to 0. One can deduce that the Boolean operation performed on the individual bits is AB’. The corresponding logic micro operation is

□.

**A** **A** □ *B*

The mask operation is similar to the selective-clear operation except that the bits of A are cleared only where there are corresponding 0’s in B. The mask operation is an AND micro operation as seen from the following numerical example:

| | |
|---|---|
| **1010** | **A before** |
| **1100** | **B (logic operand)** |
| **1000** | **A after masking** |

The two rightmost bits of A are cleared because the corresponding bits of B are 0s. The two leftmost bits are left unchanged because the corresponding bits of B are 1s. The mask operation is more convenient to use than the selective-clear operation because most computers provide an AND instruction, and few provide an instruction that executes the micro operation for selective-clear.

The insert operation inserts a new value into a group of bits. This is done by first masking the bits and then O Ring them with the required value. For example, suppose that an A register contains eight bits, 0110, 1010. To replace the four leftmost bits by the value 1001 we first mask the four unwanted bits:

| | |
|---|---|
| **0110 1010** | **A before** |
| **0000 1111** | **B (mask)** |
| **0000 1010** | **A after masking** |

and then insert the new value:

| | |
|---|---|
| **0000 1010** | **A before** |
| **1001 0000** | **B (insert)** |
| **1001 1010** | **A after insertion** |

The mask operation is an AND micro operation and the insert operation is an OR micro operation.

The clear operation compares the words in A and B and produces an all 0s result if the two numbers are equal. This operation is achieved by an exclusive-OR micro operation as shown by the following example:

**10**
**10 A**
**10**
**10 B**
□.
□
**00**
**00 AA B**

When A and B are equal, the two corresponding bits are either both 0 or both 1. In either case the exclusive-OR operation produces a 0. The all-0s result is then checked to determine if the two numbers were equal.

**Shift Micro operations**

Shift micro operations are used for serial transfer of data. They are also used in conjunction with arithmetic, logic, and other data-processing operations. The contents of a register can be shifted to the left or the right. At the same time that the bits are shifted, the first flip-flop receives its binary information from the serial input. During a shift-left operation the serial input transfers a bit into the rightmost position. During a shift-right operation the serial input transfers a bit into the leftmost position. The information transferred through the serial input determines the type of shift. There are three types of shifts: logical, circular, and arithmetic.

A logical shift is one that transfers 0 through the serial input. We will adopt the symbols shl and shr for logical shift-left and shift-right micro operations. For example:

**R1 □ shl R1**

**R2 □ shr R2**

are two micro operations that specify a 1-bit shift to the left of the content of register R1 and a 1-bit shift to the right of the content of register R2. The register symbol must be the same on both sides of the arrow. The bit transferred to the end position through the serial input is assumed to be 0 during a logical shift.

The circular shift (also known as a rotate operation) circulates the bits of the register around the two ends without loss of information. This is accomplished by connecting the serial output of the shift register to its serial input. We will use the symbols cil and cir for the circular shift left and right, respectively. The symbolic notation for the shift micro operations is shown in Table 2.7.

| ***Symbolic designation*** | | ***Description*** |
|---|---|---|
| | □ | |
| R | shl R | Shift-left register R |
| | | |
| | □ | |
| R | shr R | Shift-right register R |
| | □ | |
| R | cil R | Circular shift-left register R |
| | □ | |
| R | cir R | Circular shift-right register R |
| | □ | |
| R | ashl R | Arithmetic shift-left R |
| | □ | |
| R | ashr R | Arithmetic shift-right R |

**Shift Micro-operations**

An arithmetic shift is a micro operation that shifts a signed binary number to the left or right. An arithmetic shift-left multiplies a signed binary number by An arithmetic shift-right divides the number by 2. Arithmetic shifts must leave the sign bit unchanged because the sign of the number remains the same when it is multiplied or divided by 2. The leftmost bit in a register holds the sign bit, and the remaining bits hold the number. The sign bit is 0 for positive and 1 for negative. Negative numbers are in 2's complement form.Fig.2.11 shows a typical register of n bits. Bit Rn-1 in the leftmost position holds the sign bit. Rn-2 is the most significant bit of the number and R0 is the least significant bit. The arithmetic shift-right leaves the sign bit unchanged and shifts the number (including the sign bit) to the right. Thus Rn-1 remains the same, Rn-2 receives the bit from Rn-1, and so on for the other bits in the register. The bit in R0 is lost.

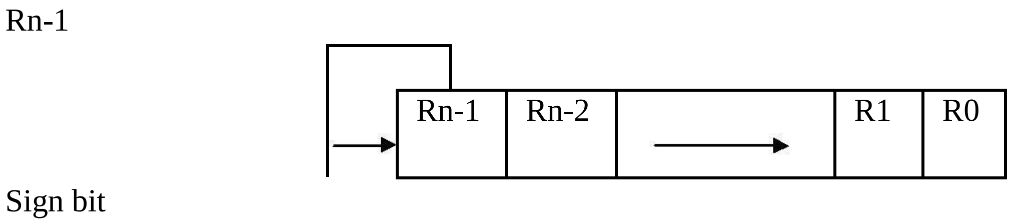


**Fig. 2.11: Arithmetic shift right**

The arithmetic shift-left inserts a 0 into R0, and shifts all other bits to the left. The initial bit of Rn-1 is lost and replaced by the bit from Rn-2. A sign reversal occurs if the bit in Rn-1 changes in value after the shift. This happens if the multiplication by 2 causes an overflow. An overflow occurs after an arithmetic shift left if initially, before the shift, Rn-1 is not equal to Rn-2. An overflow flip-flop Vs can be used to detect an arithmetic shift-left overflow.

**Vs = Rn-1 □ Rn-2**

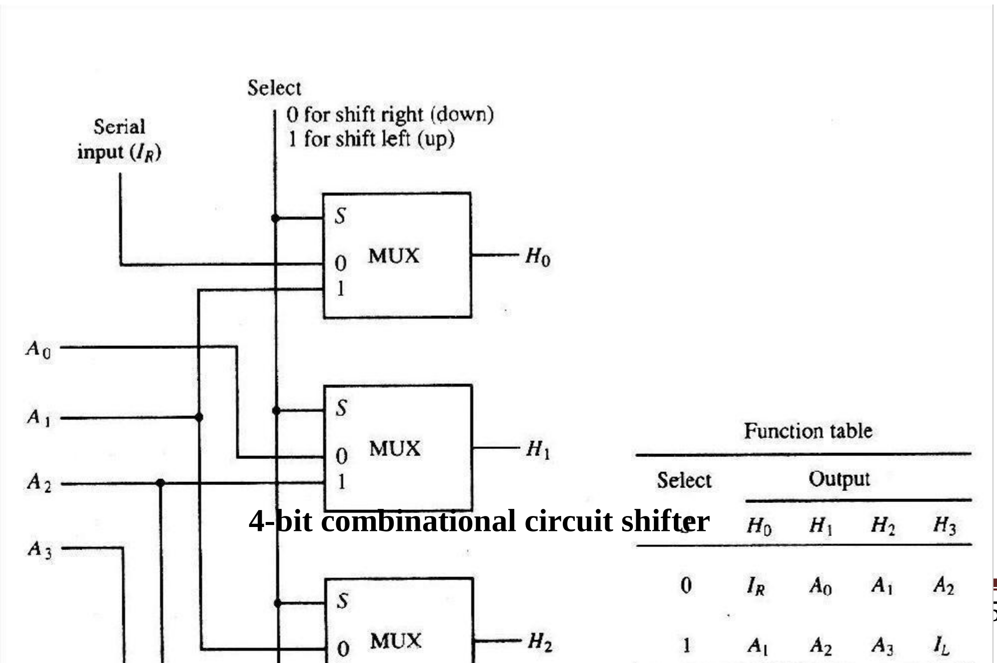
If Vs = 0, there is no overflow, but if Vs = 1, there is an overflow and a sign reversal after the shift. Vs must be transferred into the overflow flip-flop with the same clock pulse that shifts the register.

**Hardware Implementation**

A possible choice for a shift unit would be a bidirectional shift register with parallel load. Information can be transferred to the register in parallel and then shifted to the right or left. In this type of configuration, a clock pulse is needed for loading the data into the register, and another pulse is needed to initiate the shift. In a processor unit with many registers it is more efficient to implement the shift operation with a combinational circuit. In this way the content of a register that has to be shifted is first placed onto a common bus whose output is connected to the combinational shifter, and the shifted number is then loaded back into the register. This requires only one clock pulse for loading the shifted value into the register. A combinational circuit shifter can be constructed with multiplexers as shown in Fig.2.12. The 4-bit shifter has four data inputs. A0 through A3, and four data outputs, H0 through H3. There are two serial inputs, one for shift left (IL) and the other for shift right (IR). When the selection input S=0, the input data are shifted right (down in the diagram). When S=1, the input data are shifted left (up in the diagram).The function table in Fig.2.12 shows which input goes to each output after the shift. A shifter with n data inputs and outputs requires multiplexers. The two serial inputs can be controlled by another multiplexer to provide the three possible types of shifts.

**UNIT 2**

Computer Organization and Architecture



Function table

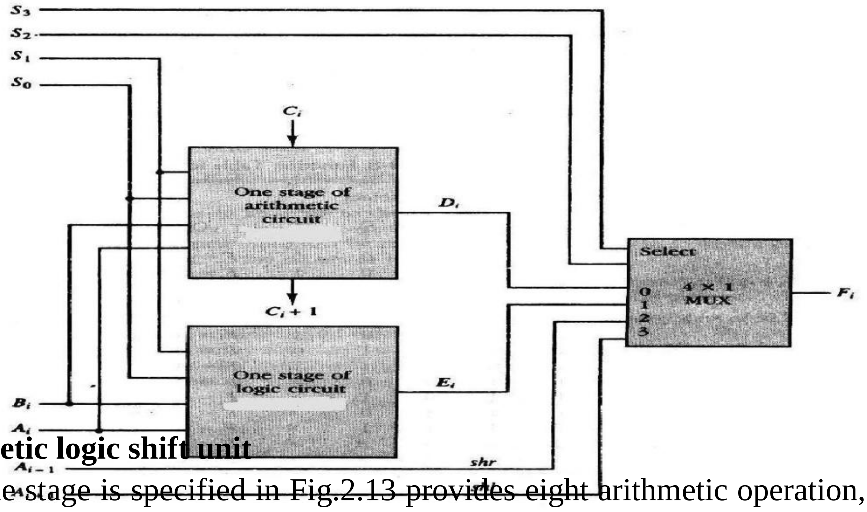
| Select | Output | | | |
|---|---|---|---|---|
| | $H_0$ | $H_1$ | $H_2$ | $H_3$ |
| 0 | $I_R$ | $A_0$ | $A_1$ | $A_2$ |
| 1 | $A_1$ | $A_2$ | $A_3$ | $I_L$ |

**4-bit combinational circuit shifter**

## Arithmetic Logic Shift Unit

Instead of having individual registers performing the microoperations directly, computer systems employ a number of storage registers connected to a common operational unit called an arithmetic logic unit, abbreviated ALU. To perform a microoperation, the contents of specified registers are placed in the inputs of the common ALU. The ALU performs an operation and the result of the operation is then transferred to a destination register.

The ALU is a combinational circuit so that the entire register transfer operation from the source registers through the ALU and into the destination register can be performed during one clock pulse period. The shift microoperations are often performed in a separate unit, but sometimes the shift unit is made part of the overall ALU.The arithmetic, logic, and shift circuits introduced in previous sections can be combined into one ALU with common selection variables. One stage of an arithmetic logic shift unit is shown in Fig.2.13. The subscript I designates a typical stage. Inputs Ai and Bi are applied to both the arithmetic and logic units. A particular microoperation is selected with inputs S1 and S0. A 4 x 1 multiplexer at the output chooses between an arithmetic output in Ei and a logic output in Hi. The data in the multiplexer are selected with inputs S3 and S2. The other two data inputs to the multiplexer receive inputs Ai-1 for the shift-right operation and Ai+1 for the shift-left operation. Note that the diagram shows just one typical stage. The circuit of Fig.2.13 must be repeated n times for an n-bit ALU. The output carry Ci+1 of a given arithmetic stage must be connected to the input carry Ci of the next stage in sequence. The input carry to the first stage is the input carry Cin, which provides a selection variable for the arithmetic operations.



### One stage of arithmetic logic shift unit

The circuit whose one stage is specified in Fig.2.13 provides eight arithmetic operation, four logic operations, and two shift operations. Each operation is selected with the five variables S3, S2, S1, S0, and Cin. The input carry Cin is used for selecting an arithmetic operation only.

Table 2.8 lists the 14 operations of the ALU. The first eight are arithmetic operations and are selected with S3S2=00. The next four are logic operations and are selected with S3S2=01. The input carry has no effect during the logic operations and is marked with don’t-care x’s. The last two operations are shift operations and are selected with S3S2=10 and 11. The other three selection inputs have no effect on the shift.

| ***Operation Select*** | | | | | ***Operation*** | ***Function*** |
|---|---|---|---|---|---|---|
| ***S*3** | ***S*2** | ***S*1** | ***S*0** | ***C*in** | | |
| 0 | 0 | 0 | 0 | 0 | F = A | Transfer A |
| 0 | 0 | 0 | 0 | 1 | F = A + 1 | Increment A |
| 0 | 0 | 0 | 1 | 0 | F = A + B | Addition |
| 0 | 0 | 0 | 1 | 1 | F = A + B + 1 | Add with carry |
| 0 | 0 | 1 | 0 | 0 | $F = A + \overline{B}$ | Subtract with borrow |
| 0 | 0 | 1 | 0 | 1 | $F = A + \overline{B} + 1$ | Subtraction |
| 0 | 0 | 1 | 1 | 0 | F = A – 1 | Decrement A |
| 0 | 0 | 1 | 1 | 1 | F = A | Transfer A |
| 0 | 1 | 0 | 0 | X | F = A ∧ B | AND |
| 0 | 1 | 0 | 1 | X | F = A V B | OR |
| 0 | 1 | 1 | 0 | X | F = A □ B | XOR |
| 0 | 1 | 1 | 1 | X | $F = \overline{A}$ | Complement A |
| 1 | 0 | X | X | X | F = shr A | Shift right A into F |
| 1 | 1 | X | X | X | F = shl A | Shift left A into F |

**: Function Table for Arithmetic Logic Shift Unit**

## INSTRUCTION CODES

- Every different processor type has its own design (different registers, buses, microoperations,

machine instructions, etc)

- Modern processor is a very complex device
  - It containMany registers
  - Multiple arithmetic units, for both integer and floating point calculations
  - The ability to pipeline several consecutive instructions to speed execution

- However, to understand how processors work, we will start with a simplified processor model This is similar to what real processors were like ~25 years ago
- M. Morris Mano introduces a simple processor model he calls the *Basic Computer*
- We will use this to introduce processor organization and the relationship of the RTL model to the higher level computer processor

***Instructions***

COMPUTER INSTRUCTIONS

- **Basic Computer Instruction Format**

**Memory-Reference Instructions** **(OP-code = 000 ~ 110)**

**12**
**15 14 11 0**

| I | Opcode | Address |
|---|---|---|

**Register-Reference Instructions** **(OP-code = 111, I = 0)**

**12**
**15 11 0**

| 1 0 1 1 | Register operation |
|---|---|

**Input-Output Instructions** **(OP-code =111, I = 1)**

**12**
**15 11 0**

| 1 1 1 1 | I/O operation |
|---|---|

## INSTRUCTION CYCLE

- In Basic Computer, a machine instruction is executed in the following cycle:
  1. Fetch an instruction from memory
  2. Decode the instruction
  3. Read the effective address from memory if the instruction has an indirect address
  4. Execute the instruction
- After an instruction is executed, the cycle starts again at step 1, for the next instruction
- *Note*: Every different processor has its own (different) instruction cycle

***Instruction Cycle***

FETCH and DECODE

- **Fetch and Decode** **T0: AR □ PC ($S_0S_1S_2$=010, T0=1)**

**T1: IR □ M [AR], PC □ PC + 1** **(S0S1S2=111, T1=1)**

**T2: D0, . . . , D7 □ Decode IR(12-14), AR □ IR(0-11), I □ IR(15)**

**T1**

**S**

2

S Bu

T0

1 s

S

0

Memory

7

unit

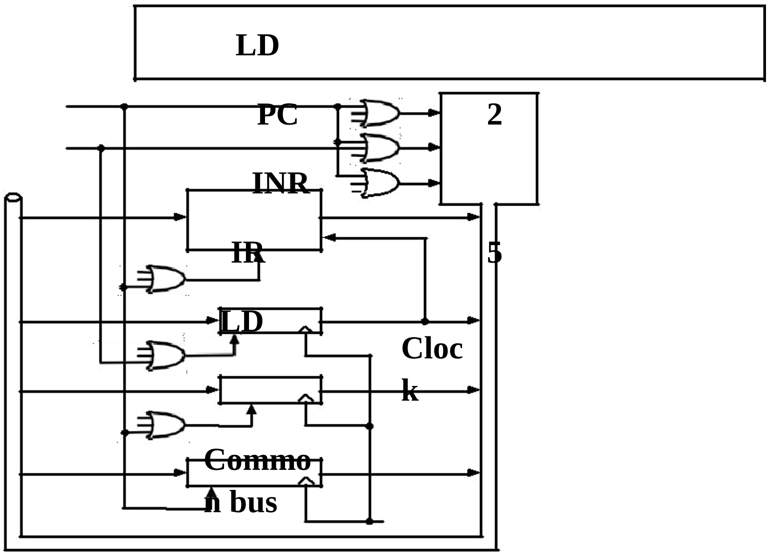
Addres

s

Rea

d

AR 1



*I/O and Interrupt*

INPUT-OUTPUT AND INTERRUPT

**A Terminal with a keyboard and a Printer**

**• Input-Output Configuration**

**Serial**

**Compu ter**

**Input-**

**output**

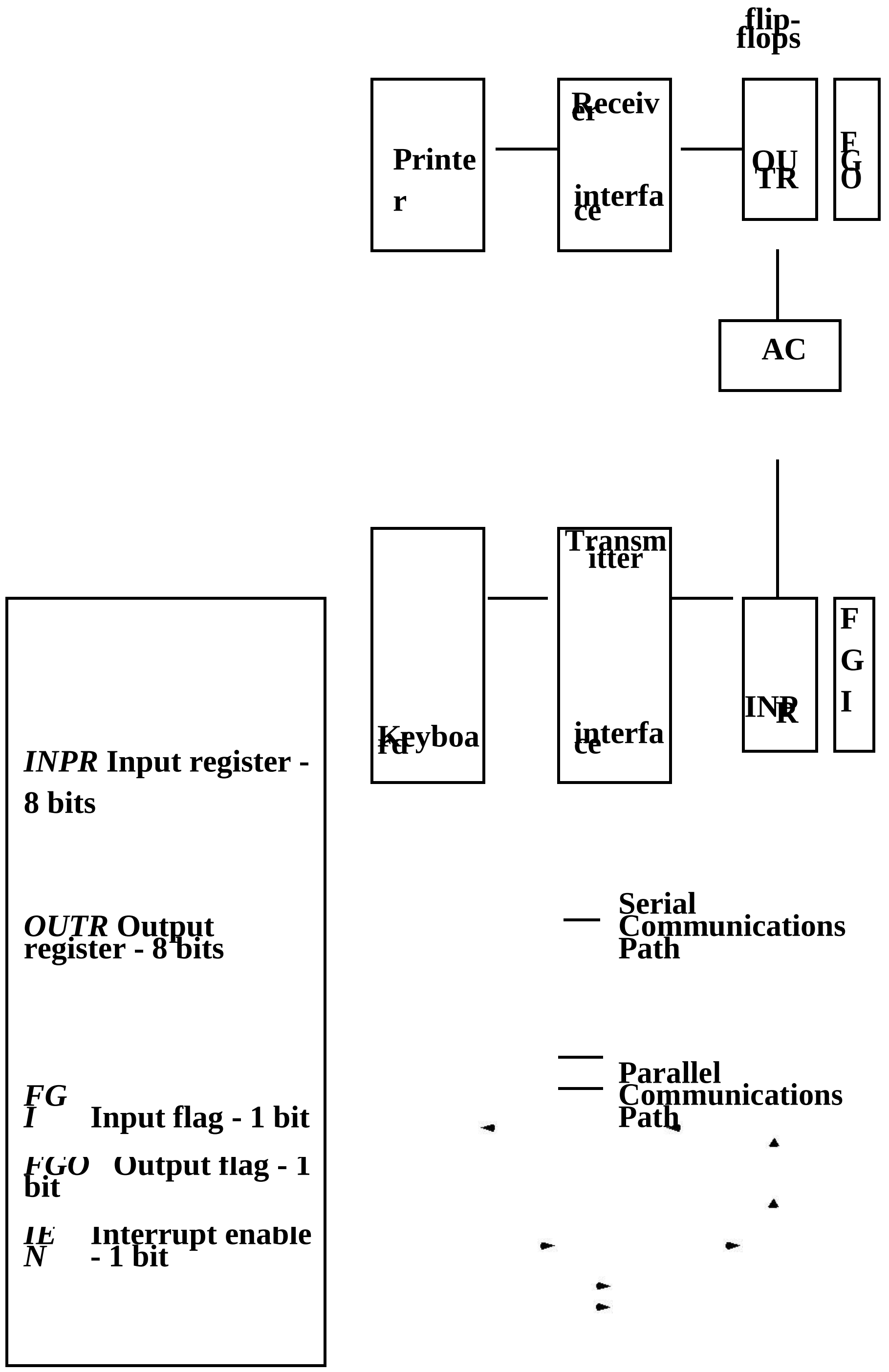
**communi cation**

**register**

**termin al**

**interfa ce**

**s and**


flip-flops
Printer
Receiver interface
OUTR
FGO
AC
Keyboard
Transmitter interface
INPR
FGI
INPR Input register - 8 bits
OUTR Output register - 8 bits
FGI Input flag - 1 bit
FGO Output flag - 1 bit
IEN Interrupt enable - 1 bit
Serial Communications Path
Parallel Communications Path

- The terminal sends and receives serial information
- The serial info. from the keyboard is shifted into INPR
- The serial info. for the printer is stored in the OUTR

- INPR and OUTR communicate with the terminal serially and with the AC in parallel.
- The flags are needed to *synchronize* the timing difference between I/O device and the computer

***MR Instructions***

## MEMORY REFERENCE INSTRUCTIONS

| Symbol | Operation Decoder | Symbolic Description |
|---|---|---|
| AND | $^{D}0$ | AC □ AC □ M[AR] |
| ADD | $^{D}1$ | AC □ AC + M[AR], E □ $C_{out}$ |
| LDA | $^{D}2$ | AC □ M[AR] |
| STA | $D_3$ | M[AR] □ AC |
| BUN | $D_4$ | PC □ AR |
| BSA | $D_5$ | M[AR] □ PC, PC □ AR + 1 |
| ISZ | $D_6$ | M[AR] □ M[AR] + 1, if M[AR] + 1 = 0 then PC □ PC+1 |

- The effective address of the instruction is in AR and was placed there during timing signal $T_2$ when I = 0, or during timing signal $T_3$ when I = 1

- Memory cycle is assumed to be short enough to complete in a CPU cycle
- The execution of MR instruction starts with $T_4$

**AND to AC**

$D_0T_4$:DR □ M[AR] Read operand

$D_0T_5$:AC □ AC □ DR, SC □ 0 AND with AC

**ADD to AC**

$D_1T_4$:DR □ M[AR] Read operand

$D_1T_5$:AC □ AC + DR, E □ $C_{out}$, SC □ 0Add to AC and store carry in E

## ARITHMETIC LOGIC SHIFT UNIT

| S3 | S2 | S1 | S0 | Cin | Operation | Function |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | F = A | Transfer A |
| 0 | 0 | 0 | 0 | 1 | F = A + 1 | Increment A |

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | F = A + B | Addition |
| 0 | 0 | 0 | 1 | 1 | F = A + B + 1 | Add with carry |
| 0 | 0 | 1 | 0 | 0 | F = A + B‘ | Subtract with borrow |
| 0 | 0 | 1 | 0 | 1 | F = A + B‘+ 1 | Subtraction |
| 0 | 0 | 1 | 1 | 0 | F = A - 1 | Decrement A |
| 1 | 0 | X | X | X | F = shr A | Shift right A into F |
| 1 | 1 | X | X | X | F = shl A | Shift left A into F |

***MR Instructions***

## MEMORY REFERENCE INSTRUCTIONS

| **Symbol** | **Operation Decoder** | **Symbolic Description** |
|---|---|---|
| **AND** | $^{D}$**0** | **AC □ AC □ M[AR]** |
| **ADD** | $^{D}$**1** | **AC □ AC + M[AR], E □ $C_{out}$** |
| **LDA** | $^{D}$**2** | **AC □ M[AR]** |
| **STA** | **$D_3$** | **M[AR] □ AC** |
| **BUN** | **$D_4$** | **PC □ AR** |
| **BSA** | **$D_5$** | **M[AR] □ PC, PC □ AR + 1** |
| **ISZ** | **$D_6$** | **M[AR] □ M[AR] + 1, if M[AR] + 1 = 0 then PC □ PC+1** |

- **The effective address of the instruction is in AR and was placed there during timing signal $T_2$ when I = 0, or during timing signal $T_3$ when I = 1**

- **Memory cycle is assumed to be short enough to complete in a CPU cycle**
- **The execution of MR instruction starts with $T_4$**

**AND to AC**

**$D_0T_4$:DR □ M[AR]** **Read operand**

**$D_0T_5$:AC □ AC □ DR, SC □ 0 AND with AC**

**ADD to AC**

**$D_1T_4$:DR □ M[AR] Read operand**

**$D_1T_5$:AC □ AC + DR, E □ $C_{out}$, SC □ 0Add to AC and store carry in E**

MEMORY REFERENCE INSTRUCTIONS

**LDA: Load to AC**

**$D_2T_4$:DR □ M[AR]**

**$D_2T_5$:AC □ DR, SC □ 0**

**STA: Store AC**

**$D_3T_4$:M[AR] □ AC, SC □ 0**

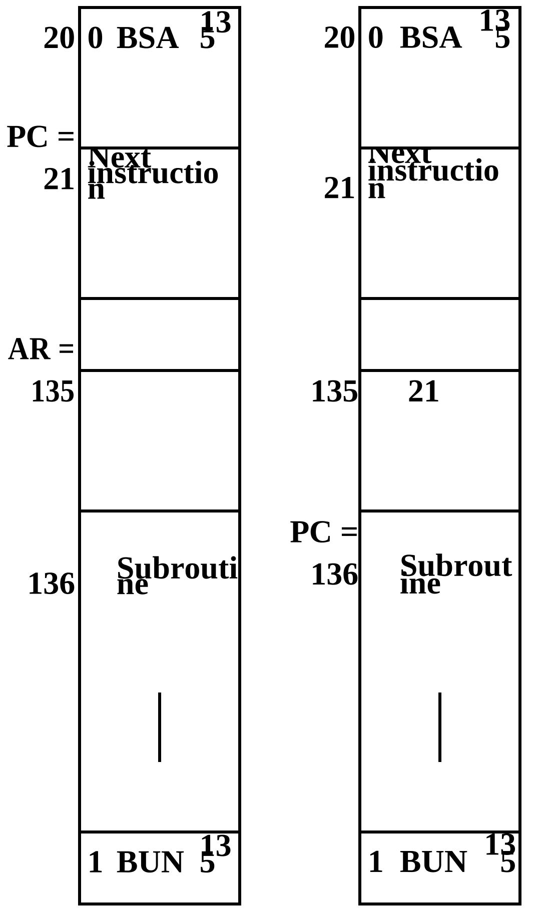
**BUN: Branch Unconditionally**

**$D_4T_4$:PC □ AR, SC □ 0**

**BSA: Branch and Save Return Address**

**M[AR] □ PC, PC □ AR + 1**

**Memory, PC, AR at time T4** **Memory, PC after execution**



**Mem ory**

**Me mor y**

***MR Instructions***

MEMORY REFERENCE

INSTRUCTIONS

**BSA:**

**$D_5T_4$: M[AR] □ PC, AR □ AR + 1**

**$D_5T_5$:PC □ AR, SC □ 0**

**ISZ: Increment and Skip-if-Zero**

**$D_6T_4$:DR □ M[AR]**

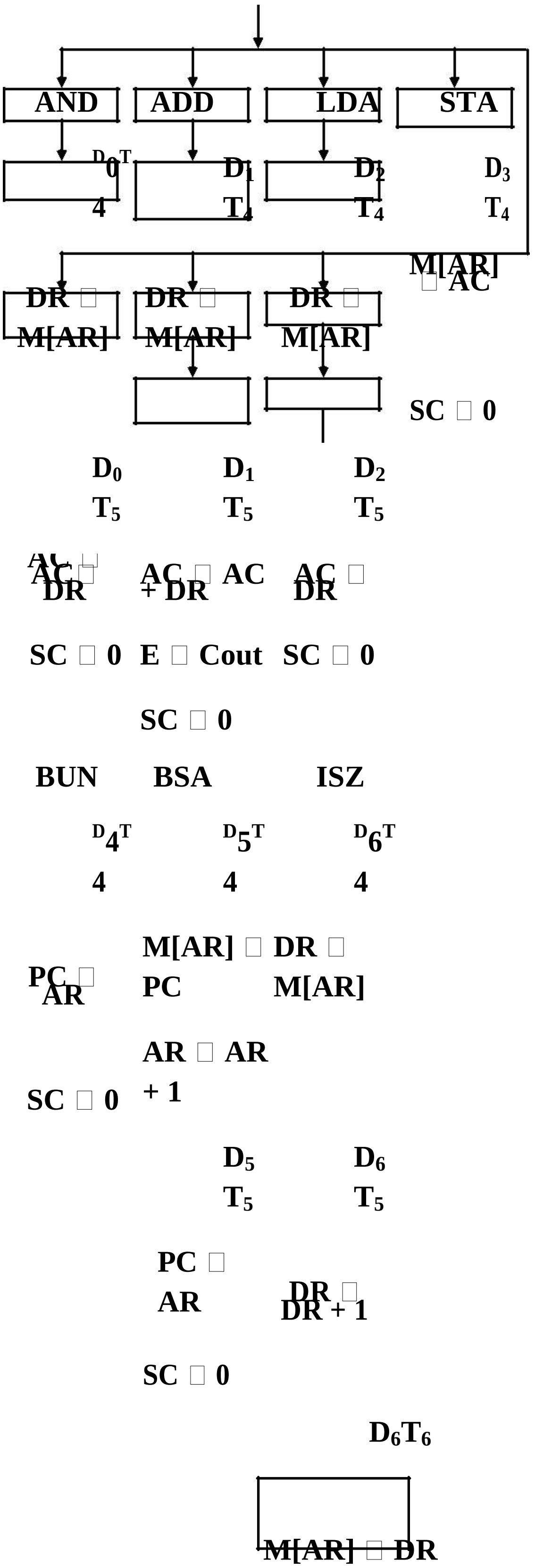
**$D_6T_5$:DR □ DR + 1**

**$D_6T_4$:M[AR] □ DR, if (DR = 0) then (PC □ PC + 1), SC □ 0**

***MR Instructions***

LOWCHART FOR MEMORY REFERENCE INSTRUCTIO

**Memory-reference instruction**



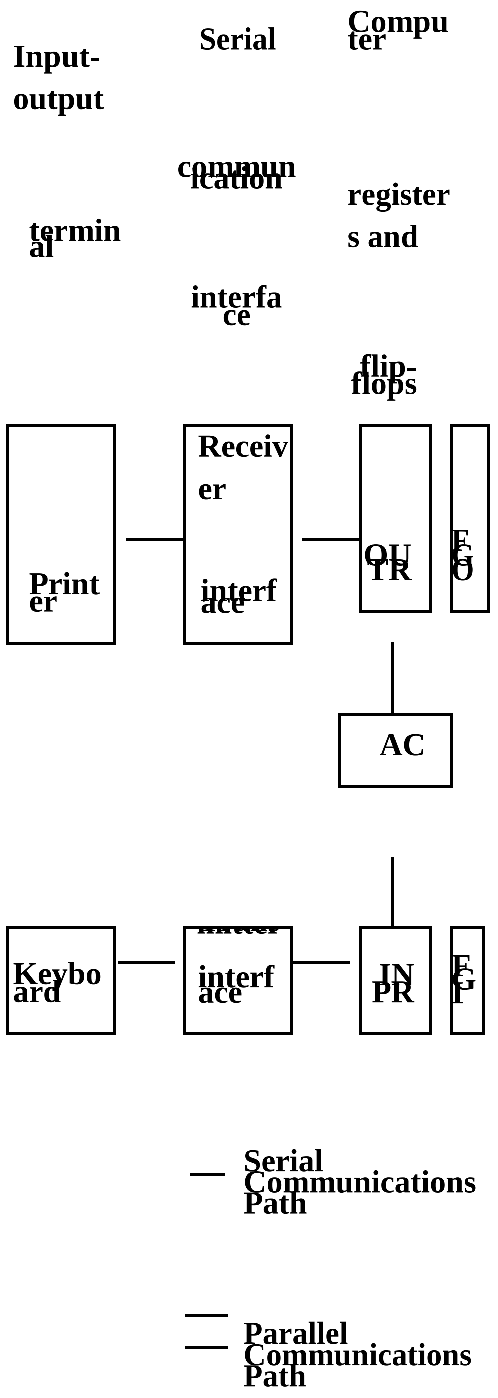
**If (DR = 0)**

**then (PC □ PC + 1)** **SC □ 0**

## INPUT-OUTPUT AND INTERRUPT

**A Terminal with a keyboard and a Printer**

- **Input-Output Configuration**


Input-output terminal
Serial communication interface
Computer registers and flip-flops
Printer
Receiver interface
OUTR
FGO
AC
Keyboard
interface
INPR
FGI
Serial Communications Path
Parallel Communications Path

*INPR* Input register - 8 bits

*OUTR* Output register - 8 bits

*FGI* Input flag - 1 bit

*FGO* Output flag - 1 bit

**IEN Interrupt enable - 1 bit**

- **The terminal sends and receives serial information**
- **The serial info. from the keyboard is shifted into INPR**
- **The serial info. for the printer is stored in the OUTR**

- **INPR and OUTR communicate with the terminal serially and with the AC in parallel.**

- **The flags are needed to *synchronize* the timing difference between I/O device and the computer**

## 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.

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

3. 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

MOVR1,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.

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 registerAs 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 | R ← R1 + M |

| | | |
|---|---|---|
| | | 1 [B] |
| MOV | R2, C | R<br>2 ← M [C] |
| ADD | R2, D | R ← R2 + M<br>2 [D] |
| MUL | R1, R2 | R<br>1 ← 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<br>[D] |
| MUL | T | AC ← AC ∗ M<br>[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:

1 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

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

3 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.

**1 Implied Mode:** In this mode the operands are specified implicitly in the definition of theinstruction. 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.

**2 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.

**3 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.

**4 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 of 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.

**5 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.

**6 Direct Address Mode:** In this mode the effective address is equal to the address part ofthe 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.

**7 Indirect Address Mode:** In this mode the address field of the instruction gives theaddress 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.

**8 Relative Address Mode:** In this mode the content of the program counter is added to theaddress 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.

**9 Indexed Addressing Mode:** In this mode the content of an index register is added to theaddress 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 instructionconverts 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.

**10 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**

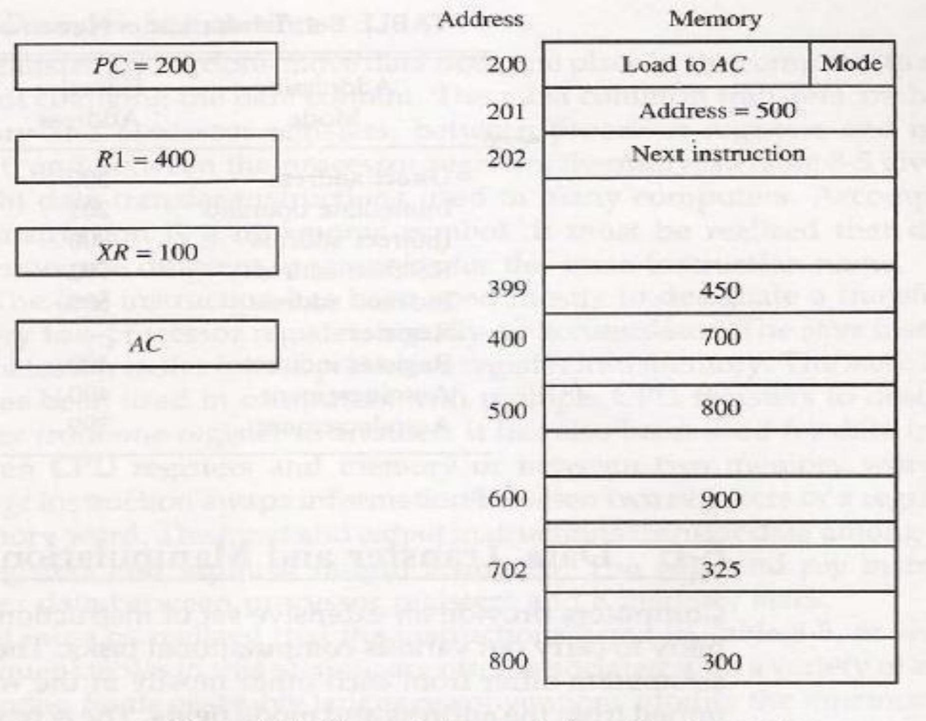


Figure 8-7 Numerical example for addressing modes.

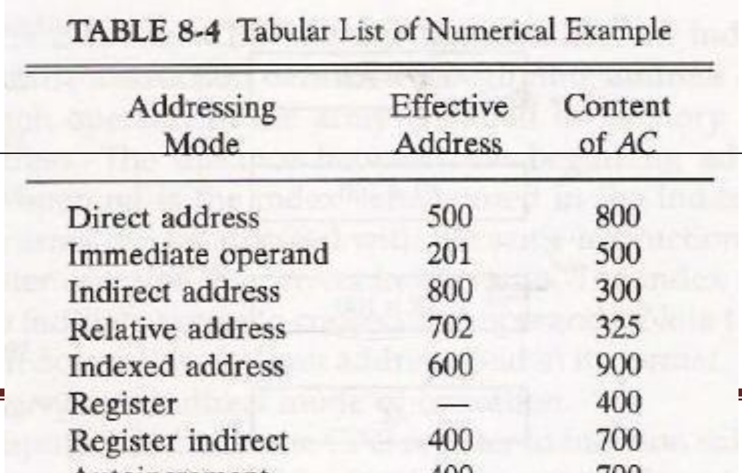


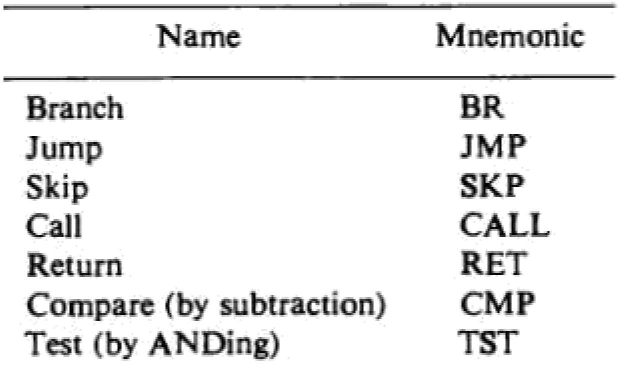
TABLE 8-4 Tabular List of Numerical Example

| 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 |

## 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.

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

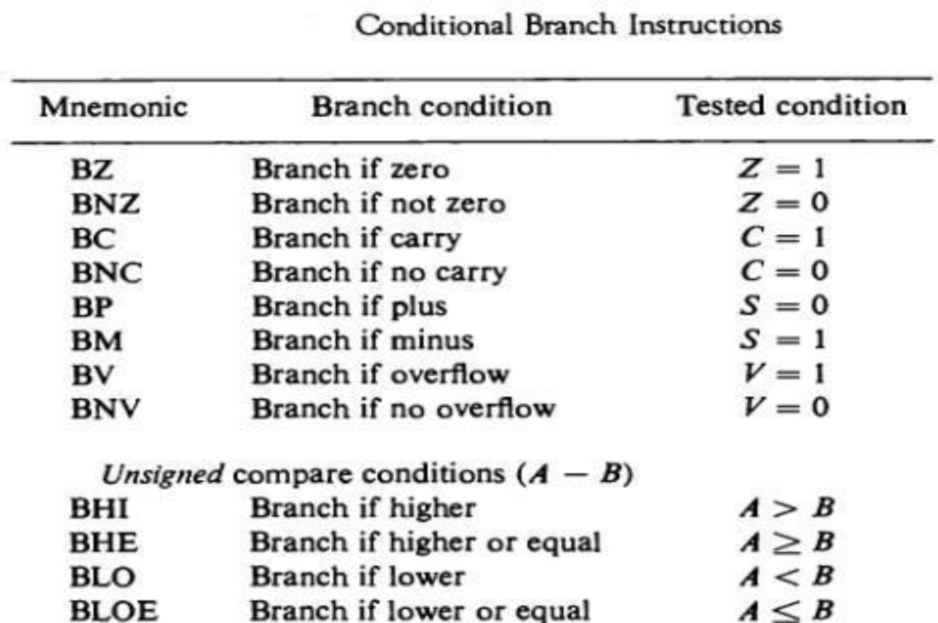
2. Bit S (sign) is set to 1 if the highest-order bit F? is 1. It is set to 0 if the bit is 0.

3. 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.

4. 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.

## 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 | $A \leq B$ |

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:

(1) 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

(2) 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:

(1) 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);

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

(3) 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:

1. The content of the program counter

2. The content of all processor registers

3. 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.

## RISC(REDUCED INSTRUCTION SET COMPUTERS)

**Reduced instruction set computing**, or **RISC** (pronounced /□ rɪsk/), is a CPU design strategy based on the insight that simplified (as opposed to complex) instructions can provide higher performance if this simplicity enables much faster execution of each instruction. A computer based on this strategy is a **reduced instruction set computer** (also **RISC**). There are many proposals for precise definitions, but the term is slowly being replaced by the more descriptive **load-store architecture**. Well known RISC families include, and SPARC .Some aspects attributed to the first RISC-*labeled* designs around 1975 include the observations that the memory-restricted compilers of the time were often unable to take advantage of features intended to facilitate *manual* assembly coding, and that complex addressing modes take many cycles to perform due to the required additional memory accesses. It was argued that such functions would be better performed by sequences of simpler instructions if this could yield implementations small enough to leave room for many registers, reducing the number of slow memory accesses. In these simple designs, most instructions are of uniform length and similar structure, arithmetic operations are restricted to CPU registers and only separate *load* and *store* instructions access memory. These properties enable a better balancing of pipeline stages than before, making RISC pipelines significantly more efficient and allowing

### Typical characteristics of RISC

For any given level of general performance, a RISC chip will typically have far fewer transistors dedicated to the core logic which originally allowed designers to increase the size of the register set and increase internal parallelism.

Other features, which are typically found in RISC architectures are:

- Uniform instruction format, using a single word with the op code in the same bit positions in every instruction, demanding less decoding;

- Identical general purpose registers, allowing any register to be used in any context, simplifying compiler design (although normally there are separate floating point registers);

- Simple addressing modes. Complex addressing performed via sequences of arithmetic and/or load-store operations;

- Fe data types in hardware, some CISCs have byte string instructions, or support complex numbers; this is so far unlikely to be found on a RISC .Exceptions abound, of course, within both CISC and RISC.RISC designs are also more likely to feature a Harvard memory model, where the instruction stream and the data stream are conceptually separated; this means that modifying the memory where code is held might not have any effect on the instructions executed by the processor (because the CPU has a separate instruction and data cache), at least until a special synchronization instruction is issued. On the upside, this allows both caches to be accessed simultaneously, which can often improve performance Many early RISC designs also shared the characteristic of having a branch delay slot.
- A branch delay slot is an instruction space immediately following a jump or branch. The instruction in this space is executed, whether or not the branch is taken (in other words the effect of the branch is delayed).

This instruction keeps the ALU of the CPU busy for the extra time normally needed to perform a branch. Nowadays the branch delay slot is considered an unfortunate side effect of a particular strategy for implementing some RISC designs, and modern RISC designs generally do away with it .

## MICROPROGRAMMED CONTROL

## Control Memory

### Micro program

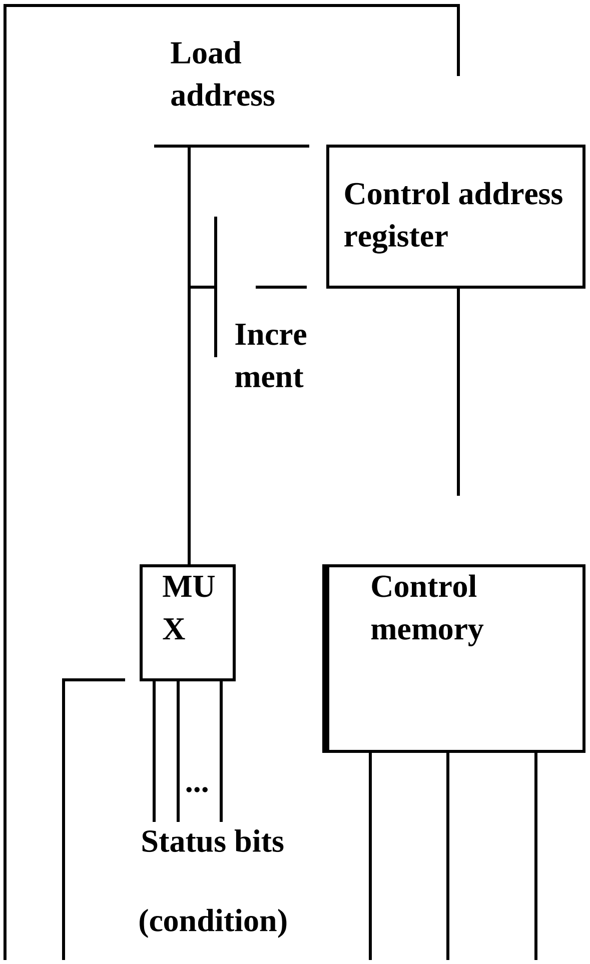
Program stored in memory that generates all the control signals required to execute the instruction set Correctly. Consists of micro instructions .it Contains a control word and a sequencing word Control Word - All the control information required for one clock cycle Sequencing Word - Information needed to decide the next microinstruction address.

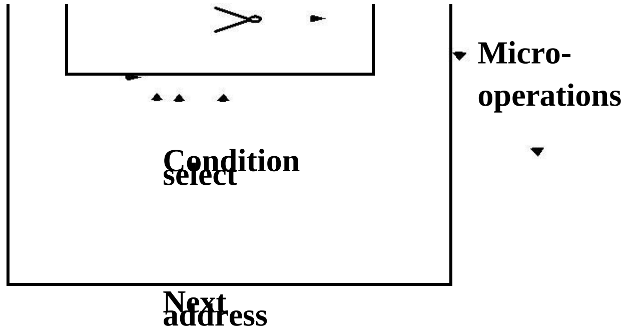
Control Memory(Control Storage: CS) Storage in the micro programmed control unit to store the micro program Writeable Control Memory(Writeable Control Storage :WCS) CS whose contents can be modified, Allows the micro program can be changed, Instruction set can be changed or modified.

### ADDRESS SEQUENSING

A Micro program Control Unit that determines the Microinstruction Address to be executed n the next clock cycle

CONDITIONAL BRANCHING





**Conditional Branch**

**If *Condition* is true, then *Branch* (address from the next address field of the current microinstruction)**

**else *Fall Through***

**Conditions to Test: O(overflow), N(negative),Z(zero), C(carry), etc.**

**Unconditional Branch**

**Fixing the value of one status bit at the input of the multiplexer to 1**

## CONDITIONAL BRANCHING

Unconditional Branch

Fixing the value of one status bit at the input of the multiplexer to 1

Conditional Branch

If *Condition* is true, then *Branch* (address from

the next address field of the current microinstruction)

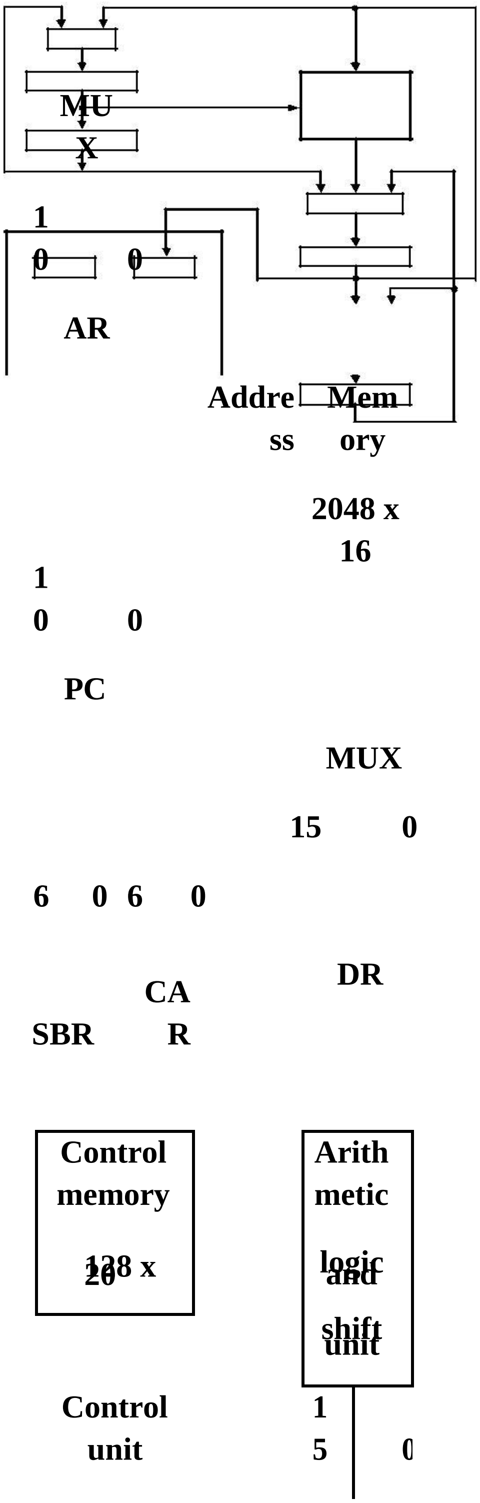
else *Fall Through*

Conditions to Test: O(overflow), N(negative),

Z(zero), C(carry), etc.

## MICROPROGRAM EXAMPLE

**Computer Configuration**



AC

DESIGN OF CONTROL UNIT

- DECODING ALU CONTROL INFORMATION -

**microoperation fields**

**F1** **F2** **F3**

**3 x 8 decoder** **3 x 8 decoder** **3 x 8 decoder**

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

**AND**

**ADD**

**AC**

**Arithmetic**

**logic and**

**DRTAC**

**DR**

**shift unit**

**From PC** **From DR(0-10)**

**Load**

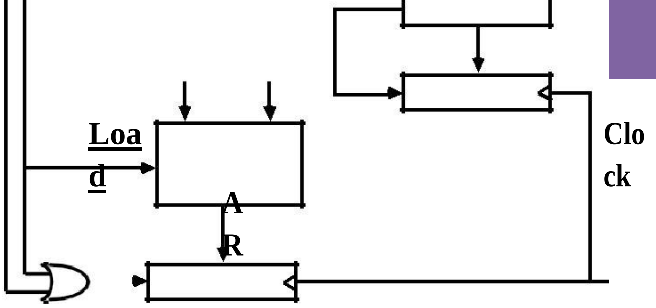
**AC**

**DRTAR**

**0** **1**

**Select**

**Multiplexers**

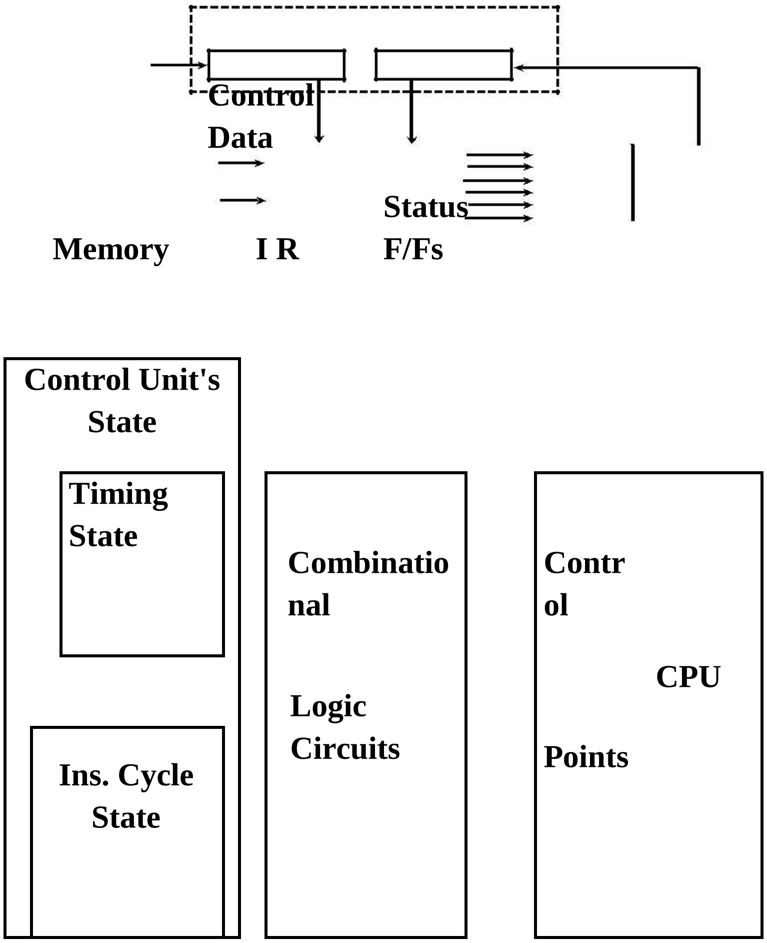


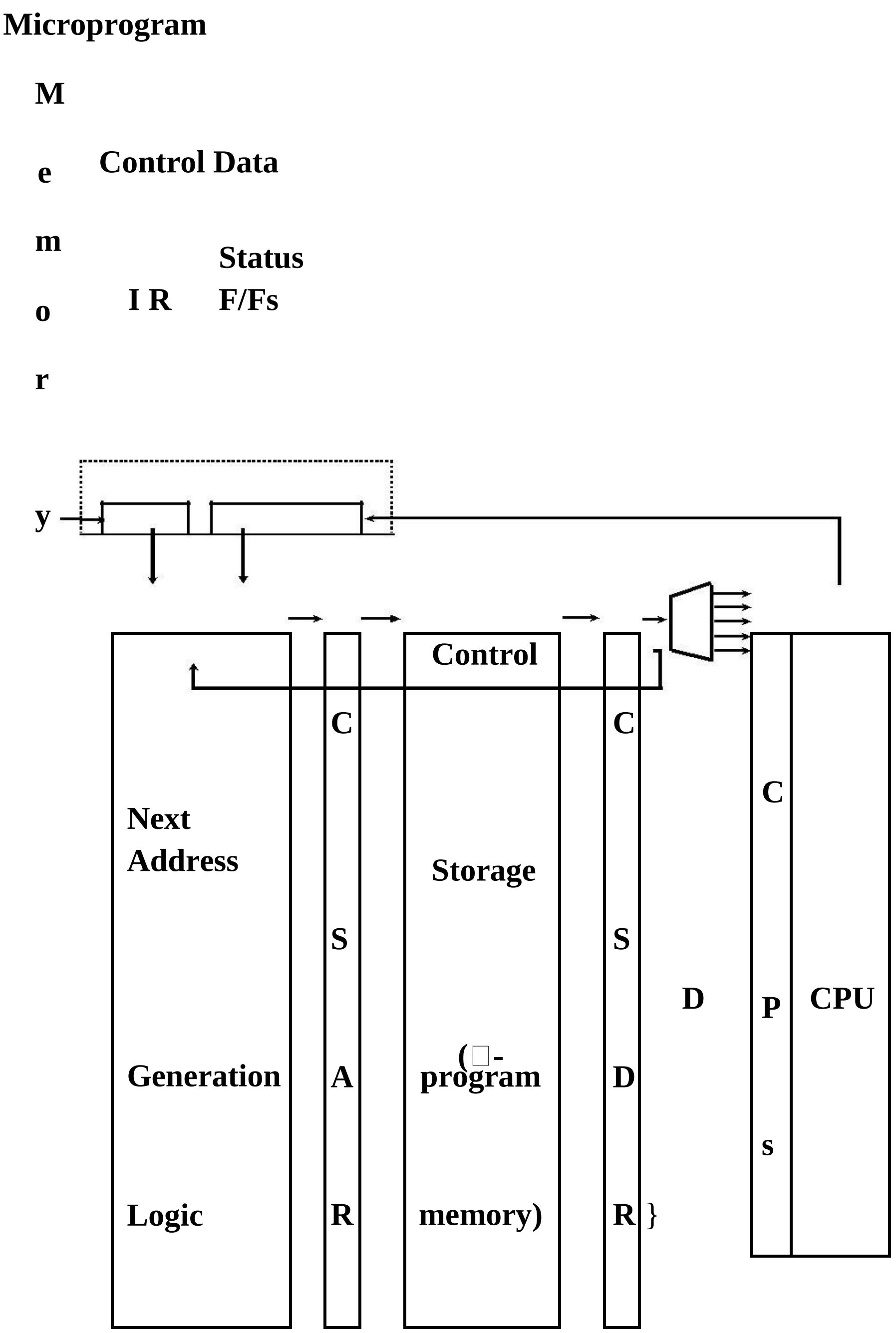
*Implementation of Control Unit*

COMPARISON OF CONTROL UNIT IMPLEMENTATIONS

**Control Unit Implementation**

**Combinational Logic Circuits (Hard-wired)**




Microprogram
Memory
Control Data
I R
Status F/Fs
Next Address Generation Logic
CSAR
Control Storage (□-program memory)
CSDR
}
D
CPs
CPU

## MICROPROGRAMMED CONTROL

### MICROINSTRUCTION FORMAT

- Control Information, Sequencing Information, Constant Information which is useful when feeding into the system.

These information needs to be organized in some way for Efficient use of the microinstruction

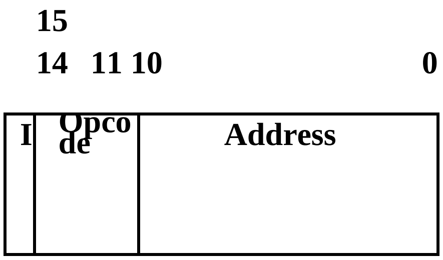
bits Fast decoding

Field Encoding

- Encoding the microinstruction bits
- Encoding slows down the execution speed due to the decoding delay
- Encoding also reduces the flexibility due to the decoding hardware

MACHINE INSTRUCTION FORMAT

**Machine instruction format**



**Sample machine instructions**

| Symbol | OP-code | Description |
|---|---|---|
| ADD | 0000 | AC □ AC + M[EA] |
| BRANCH | 0001 | if (AC < 0) then (PC □ EA) |
| STORE | 0010 | M[EA] □ AC |
| EXCHANGE | 0011 | AC □ M[EA], M[EA] □ AC |

**EA is the effective address**

**Microinstruction Format**

| 3 | 3 | 3 | 2 | 2 | 7 |
|---|---|---|---|---|---|
| F1 | F2 | F3 | CD | BR | AD |

**F1, F2, F3: Microoperation fields**

**CD: Condition for branching**

**BR: Branch field**

**AD: Address field**

## SYMBOLIC MICROINSTRUCTIONS

- Symbols are used in microinstructions as in assembly language
- A symbolic microprogram can be translated into its binary equivalent by a microprogram assembler.

Sample Format

five fields: label; micro-ops; CD; BR; AD

CD: one of {U, I, S, Z}, where U: Unconditional Branch
I: Indirect address bit
S: Sign of AC
Z: Zero value in AC

BR: one of {JMP, CALL, RET, MAP}

AD: one of {Symbolic address, NEXT, empty}

## SYMBOLIC MICROPROGRAM - FETCH ROUTINE

During FETCH, Read an instruction from memory and decode the instruction and update PC
Sequence of microoperations in the fetch cycle:AR □ PC

DR □ M[AR], PC □ PC + 1,AR □ DR(0-10), CAR(2-5) □ DR(11-14), CAR(0,1,6) □ 0

### SYMBOLIC MICROPROGRAM

- Control Storage: 128 20-bit words
- The first 64 words: Routines for the 16 machine instructions
- The last 64 words: Used for other purpose (e.g., fetch routine and other subroutines)
- Mapping: OP-code XXXX into 0XXXX00, the first address for the 16 routines

SYMBOLIC MICROPROGRAM

**Partial Symbolic Microprogram**

| **Label** | **Microops** | **CD** | **BR** | **AD** |
|---|---|---|---|---|
| | **ORG 0** | | | |
| **ADD:** | **NOP** | **I** | **CALL** | **INDRCT** |
| | **READ** | **U** | **JMP** | **NEXT** |
| | **ADD** | **U** | **JMP** | **FETCH** |
| | **ORG 4** | | | |
| **BRANCH:** | **NOP** | **S** | **JMP** | **OVER** |
| | **NOP** | **U** | **JMP** | **FETCH** |
| **OVER:** | **NOP** | **I** | **CALL** | **INDRCT** |
| | **ARTPC** | **U** | **JMP** | **FETCH** |
| | **ORG 8** | | | |

| | | | | |
|---|---|---|---|---|
| STORE: | NOP | I | CALL | INDRCT |
| | ACTDR | U | JMP | NEXT |
| | WRITE | U | JMP | FETCH |
| | ORG 12 | | | |
| EXCHANGE: | NOP | I | CALL | INDRCT |
| | READ | U | JMP | NEXT |
| | ACTDR, DRTAC | U | JMP | NEXT |
| | WRITE | U | JMP | FETCH |
| | ORG 64 | | | |
| FETCH: | PCTAR | U | JMP | NEXT |
| | READ, INCPC | U | JMP | NEXT |
| | DRTAR | U | MAP | |
| INDRCT: | READ | U | JMP | NEXT |
| | DRTAR | U | RET | |

BINARY MICROPROGRAM

| | Address | | Binary Microinstruction | | | | | |
|---|---|---|---|---|---|---|---|---|
| Micro Routine | Decimal | Binary | F1 | F2 | F3 | CD | BR | AD |
| ADD | 0 | 0000000 | 000 | 000 | 000 | 01 | 01 | 1000011 |
| | 1 | 0000001 | 000 | 100 | 000 | 00 | 00 | 0000010 |
| | 2 | 0000010 | 001 | 000 | 000 | 00 | 00 | 1000000 |
| | 3 | 0000011 | 000 | 000 | 000 | 00 | 00 | 1000000 |
| BRANCH | 4 | 0000100 | 000 | 000 | 000 | 10 | 00 | 0000110 |
| | 5 | 000010 | 000 | 000 | 000 | 00 | 00 | 100000 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | 1 | | | | | | 0 |
| | 6 | 0000110 | 000 | 000 | 000 | 01 | 01 | 1000011 |
| | 7 | 0000111 | 000 | 000 | 110 | 00 | 00 | 1000000 |
| STORE | 8 | 0001000 | 000 | 000 | 000 | 01 | 01 | 1000011 |
| | 9 | 0001001 | 000 | 101 | 000 | 00 | 00 | 0001010 |
| | 10 | 0001010 | 111 | 000 | 000 | 00 | 00 | 1000000 |
| | 11 | 0001011 | 000 | 000 | 000 | 00 | 00 | 1000000 |
| EXCHANGE | 12 | 0001100 | 000 | 000 | 000 | 01 | 01 | 1000011 |
| | 13 | 0001101 | 001 | 000 | 000 | 00 | 00 | 0001110 |
| | 14 | 0001110 | 100 | 101 | 000 | 00 | 00 | 0001111 |
| | 15 | 0001111 | 111 | 000 | 000 | 00 | 00 | 1000000 |
| FETCH | 64 | 1000000 | 110 | 000 | 000 | 00 | 00 | 1000001 |
| | 65 | 1000001 | 000 | 100 | 101 | 00 | 00 | 1000010 |
| | 66 | 1000010 | 101 | 000 | 000 | 00 | 11 | 0000000 |
| INDRCT | 67 | 1000011 | 000 | 100 | 000 | 00 | 00 | 1000100 |
| | 68 | 1000100 | 101 | 000 | 000 | 00 | 10 | 0000000 |

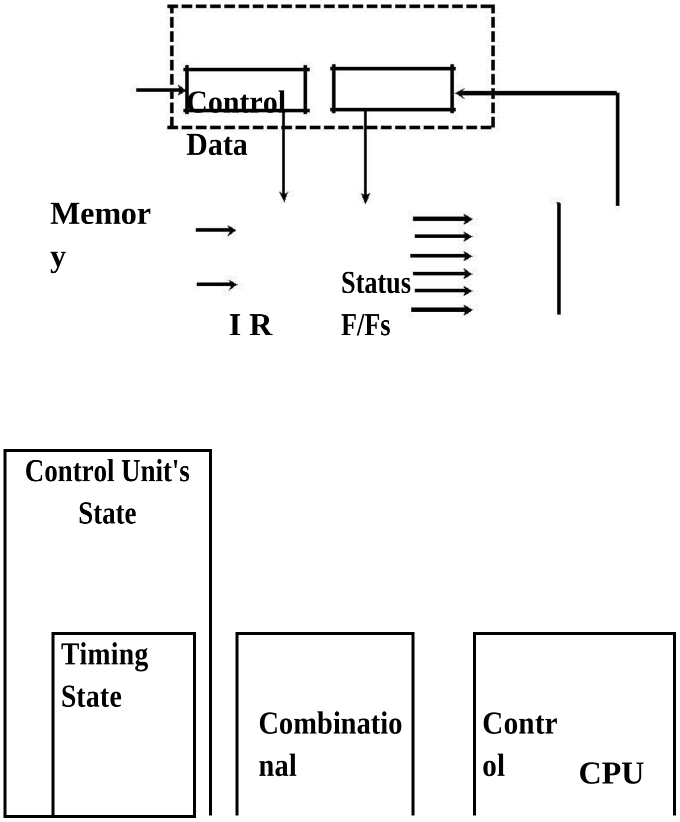
**This microprogram can be implemented using ROM**

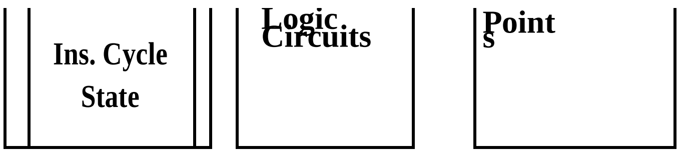
***Implementation of Control Unit***

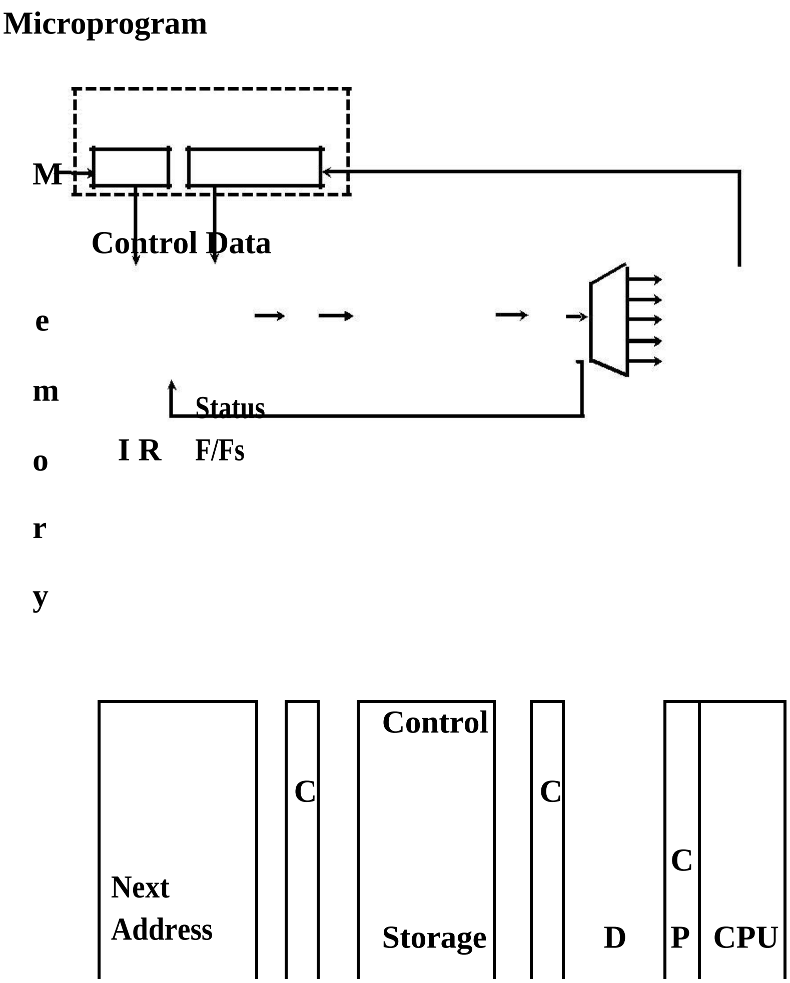
COMPARISON OF CONTROL UNIT IMPLEMENTATIONS

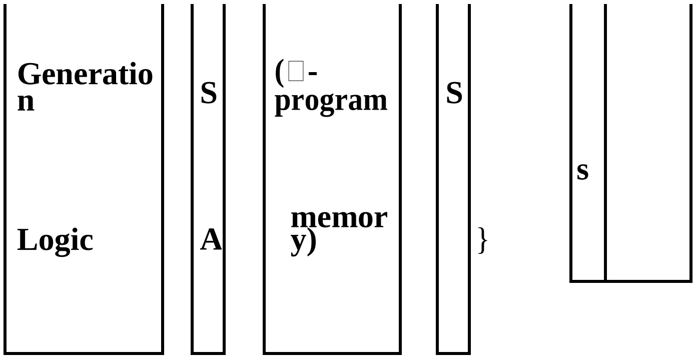
**Control Unit Implementation**

**Combinational Logic Circuits (Hard-wired)**




Ins. Cycle
State
Logic
Circuits
Point
s


Microprogram
M
e
m
o
r
y
Control Data
Status
I R
F/Fs
Next
Address
C
Control
Storage
C
D
C
P
CPU



## NANOSTORAGE AND NANOINSTRUCTION

The decoder circuits in a vertical microprogram storage organization can be replaced by a ROM

=> Two levels of control storage First level - *Control Storage*

Second level - *Nano Storage*

Two-level microprogram First level *Vertical* format Microprogram Second level

-*Horizontal* format Nanoprogram , Interprets the microinstruction fields, thus converts a vertical microinstruction format into a horizontal nanoinstruction format.

## TWO-LEVEL MICROPROGRAMMING - EXAMPLE

Microprogram: 2048 microinstructions of 200 bits each With 1-Level Control Storage: 2048 x 200

= 409,600 bits Assumption:256 distinct microinstructions among 2048
* With 2-Level Control Storage:

Nano Storage: 256 x 200 bits to store 256 distinct nanoinstructions

Control storage: 2048 x 8 bits

To address 256 nano storage locations 8 bits are needed

* Total 1-Level control storage: 409,600 bits

Total 2-Level control storage: 67,584 bits (256 x 200 + 2048 x 8)

# UNIT-2

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 mapping

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

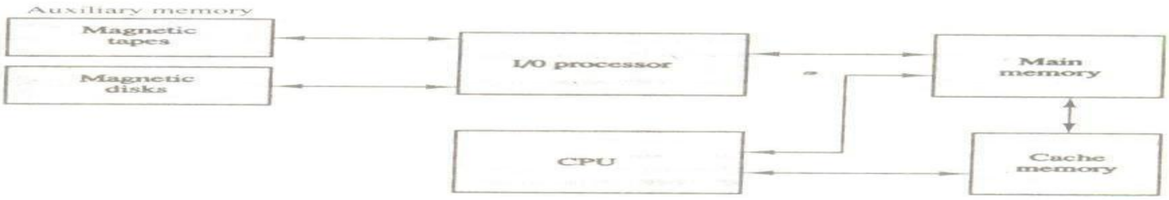


**Figure -** Memory hierarchy in a computer system.

Making programs and data available at a rapid rate, it is possible to increase the performance rate of the computer. 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.

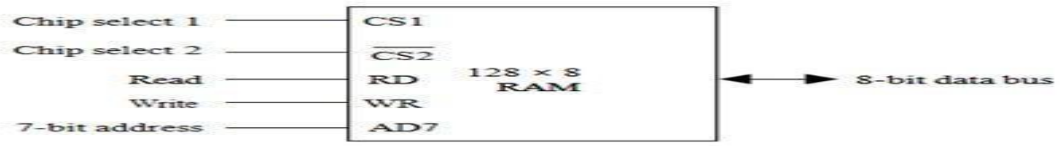
Most of the main memory in a general-purpose computer is made up of RAM integrated circuit chips, but a portion of the memory may be constructed with ROM chips. Originally, RAM was used to refer to a random-access memory, but now it is used to designate a read/write memory to distinguish it from a read-only 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.

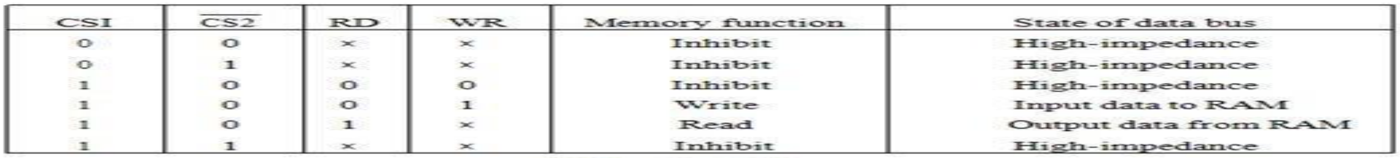
RAM and ROM chips are available in a variety of sizes. If the memory needed for the computer is larger than the capacity of one chip, it is necessary to combine a number of chips to form the required memory size. To demonstrate the chip interconnection, we will show an example of a 1024 × 8 memory constructed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



(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

Fig:Typical Ram

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 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 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, 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

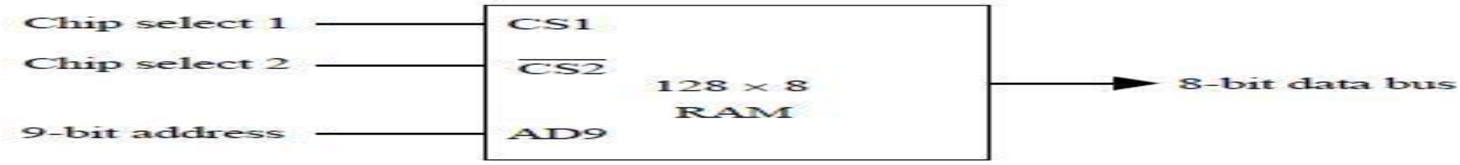
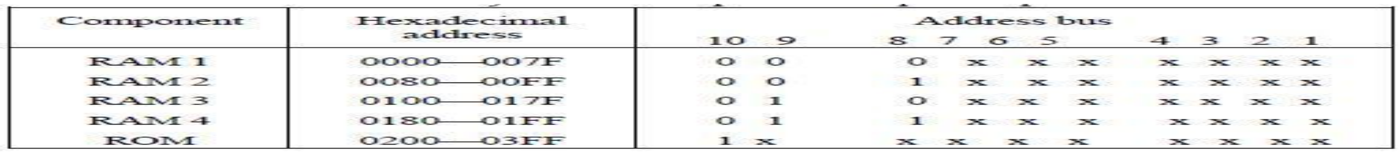


**Figure-**Typical ROM chip.

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 Micro procomputer



| 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.

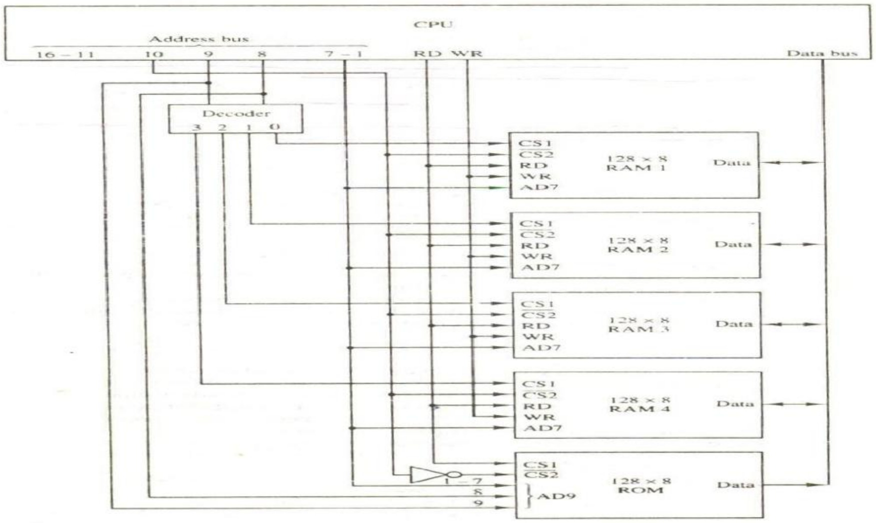


**Figure -**Memory connection to the CPU.

## 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.

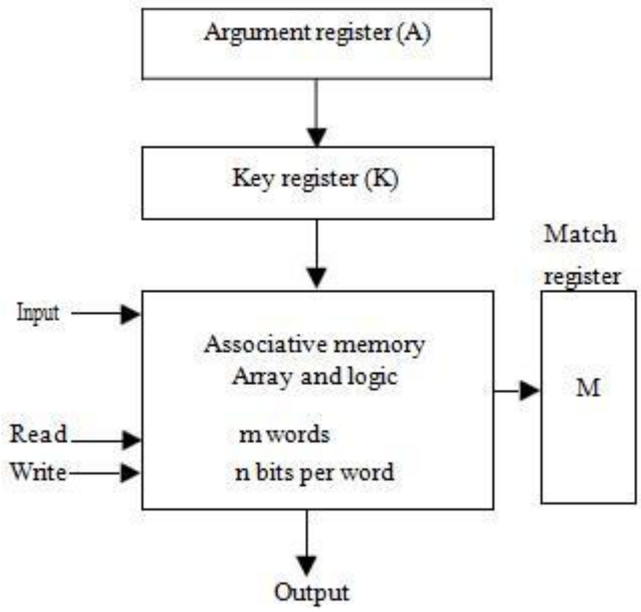


**Figure-** Block diagram of associative memory

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.

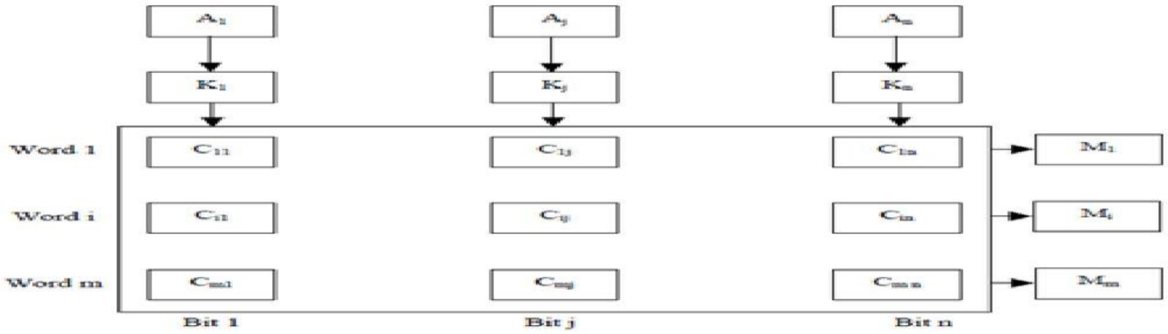


**Figure -**Associative memory of m word, n cells per word

The internal organization of a typical cell $C_{ij}$ is shown in Fig..It consists of 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 .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 $M_i = x_1\, x_2\, x_3 \ldots x_n$ And constitutes the AND operation of all pairs of matched bits in a word.

**Figure -** One cell of associative memory.

We now include the key bit $K_j$ in the comparison logic. The requirement is that if $K_j = 0$, the corresponding bits of $A_j$ and $F_{ij}$ need no comparison. Only when $K_j = 1$ must they be compared. This requirement is achieved by ORing each term with $K_j$' , thus:

$$x_j + K'_j = \begin{cases} x_j & \text{if } K_j = 1 \\ 1 & \text{if } K_j = 0 \end{cases}$$

When $K_j = 1$, we have $K_j' = 0$ and $x_j + 0 = x_j$. When $K_j = 0$, then $K_j' = 1$ $x_j + 1 = 1$. A term $(x_j + K_j')$ 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 $K_j = 1$.

The match logic for word i in an associative memory can now be expressed by the following Boolean function:"

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

Each term in the expression will be equal to 1 if its corresponding $K'_j = 0$. ifK $_j$= 1, the term will be either 0 or 1 depending on the value of $x_j$. A match will occur and $M_i$ will be equal to 1 if all terms are equal to 1. If we substitute the original definition of $x_j$. The Boolean function above can be expressed as follows:

$M_i = \prod (A_jF_{ij} + A'_j F'_j + K'_j)_{j=1}$ 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 nd one OR gate. The inverters for $A_j$ and $K_j$ are needed once for each column and are used for all bits in th column. The output of all OR gates in the cells of the same word go to the input of a common AND gate togenerate the match signal for $M_i$.$M_i$ will be logic 1 if a catch occurs and 0 if no match occurs. Note that if thekey register contains all 0's, output $M_i$ will be a 1 irrespective of the value of A or the word. This occurrene 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 $M_i$ bit is a 1.

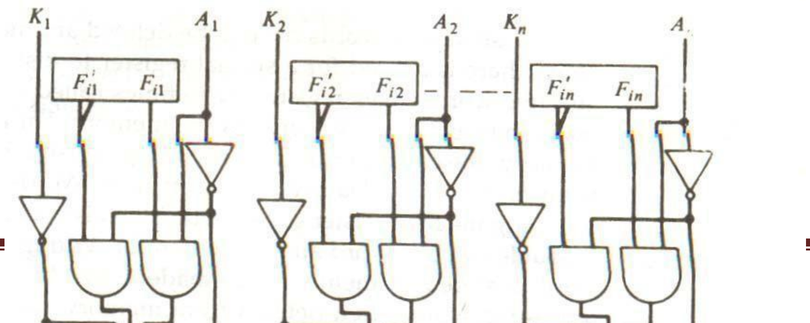


**Figure -** Match logic for one word of associative memory

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 $M_i$ 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:

4. Associative mapping
5. Direct mapping
6. 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.

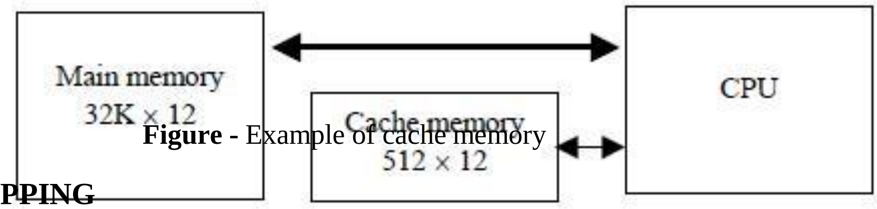
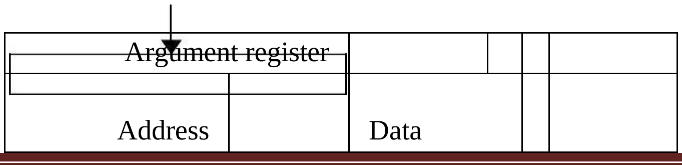


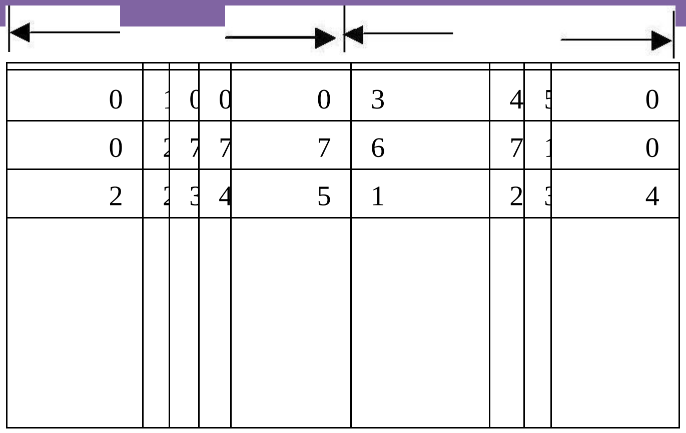
**Figure -** Example of cache memory

## 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

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 |
| | | | | | | | | |

**Figure-**Associative mapping cache (all numbers in octal)

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.

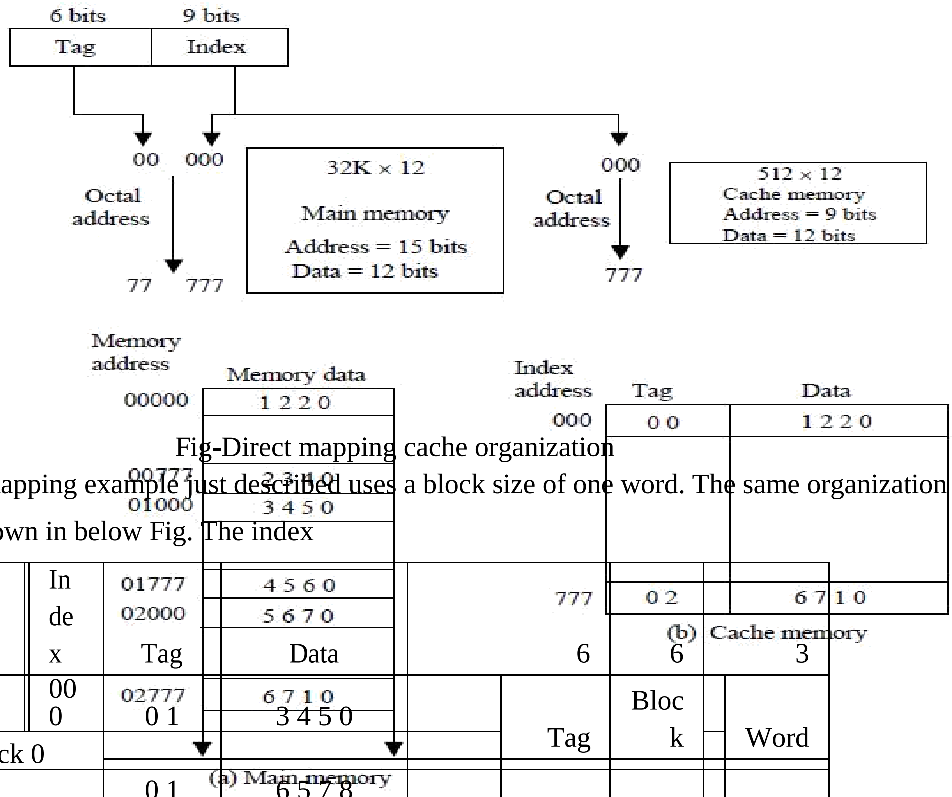


Fig-Direct mapping cache organization

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

| | Index | Tag | Data |
|---|---|---|---|
| | 000 | 0 1 | 3 4 5 0 |
| Block 0 | | | |
| | 007 | 0 1 | 6 5 7 8 |
| | 010 | | |
| Block 1 | | | |
| | 017 | | |
| | 770 | 0 2 | |
| Block 3 | | | |
| | 777 | 0 2 | 6 7 1 0 |

| 6 | 6 | 3 |
|---|---|---|
| Tag | Block | Word |
| | 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 |
|---|---|---|---|---|
| 00 | | | | |
| 0 | 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

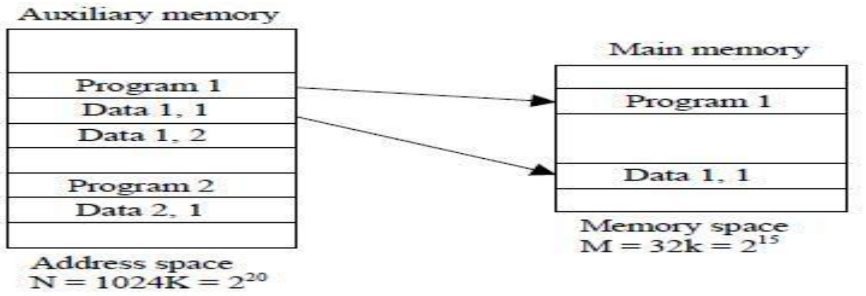
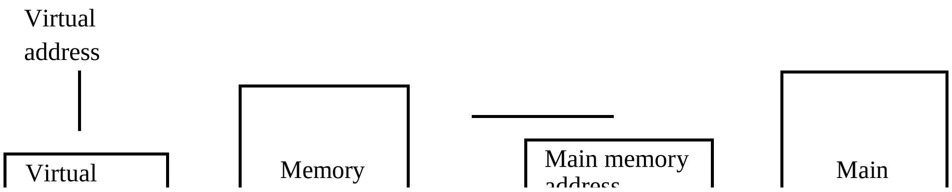


Fig-Relation between address and memory space in a virtual memory system

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



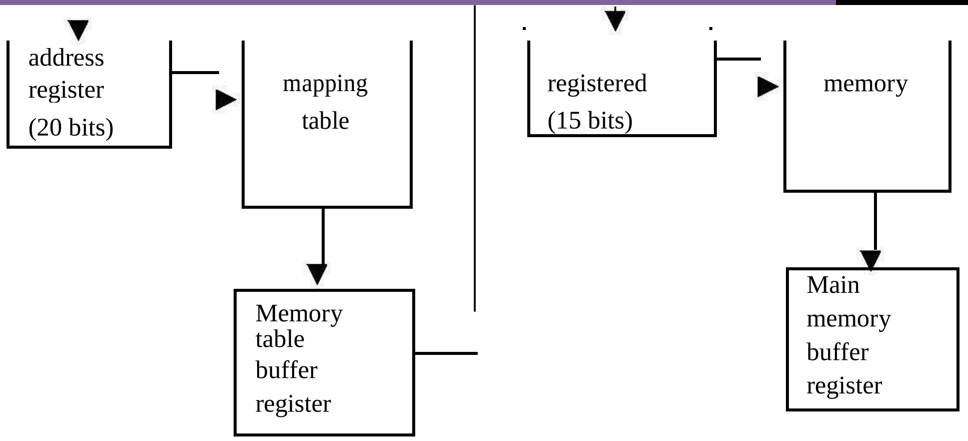


Figure - Memory table for mapping a virtual address.

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.

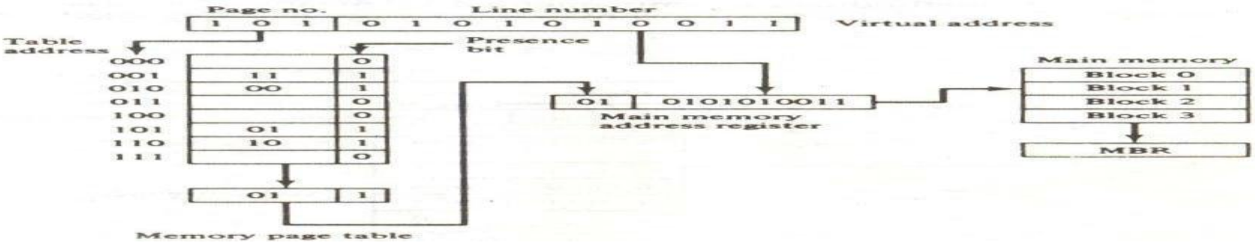


Figure - Memory table in a paged system.

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.

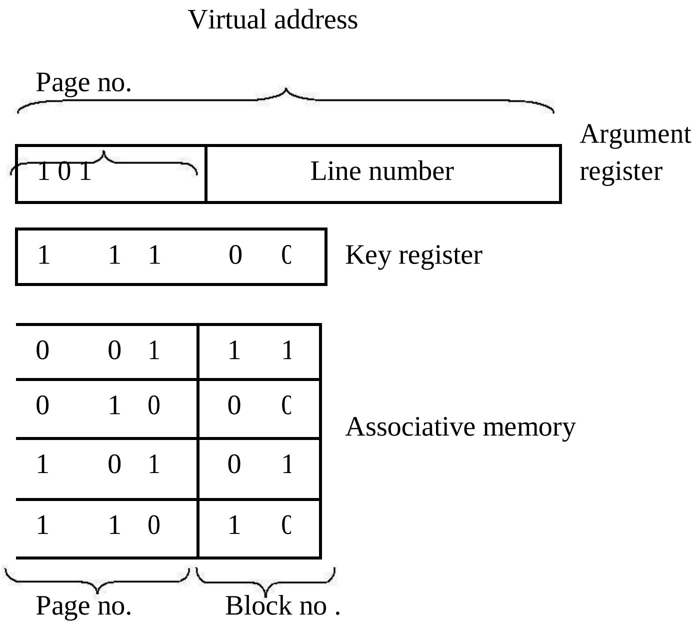


**Figure -**An associative memory page table.

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.

**Redundant Array of Inexpensive Disk(RAID)**

RAID (redundant array of independent disks; originally *redundant array of inexpensive disks*) is a way of storing the same data in different places (thus, redundantly) on multiple hard disks. By placing data on multiple disks, I/O (input/output) operations can overlap in a balanced way, improving performance. Since multiple disks increases the mean time between failures (MTBF), storing data redundantly also increases fault tolerance.

A RAID appears to the operating system to be a single logical hard disk. RAID employs the technique of disk striping, which involves partitioning each drive's storage space into units ranging from a sector (512 bytes) up to several megabytes. The stripes of all the disks are interleaved and addressed in order.In a multi-user system, better performance requires establishing a stripe wide enough to hold the typical or maximum size record. This allows overlapped disk I/O across drives. There are at least nine types of RAID plus a non-redundant array (RAID-0):

α) RAID-0: This technique has striping but no redundancy of data. It offers the best performance but no fault-tolerance.

a) RAID-1: This type is also known as *disk mirroring* and consists of at least two drives that duplicate the storage of data. There is no striping. Read performance is improved since either disk can be read at the same time

b) RAID-2: This type uses striping across disks with some disks storing error checking and correcting (ECC) information. It has no advantage over RAID-3.

c)RAID-3: This type uses striping and dedicates one drive to storing parity information. The embedded error checking (ECC) information is used to detect errors. Data recovery is accomplished by calculating the exclusive OR (XOR) of the information recorded on the other drives.

δ)RAID-4: This type uses large stripes, which means you can read records from any single drive

e)RAID-5: This type includes a rotating parity array, thus addressing the write limitation inRAID-4. Thus, all read and write operations can be overlapped. RAID-5 stores parity information but not redundant data (but parity information can be used to reconstruct data). RAID-5 requires at least three and usually five disks for the array. It's best for multi-user systems in which performance is not critical or which do few write operations.

f)RAID-6: This type is similar to RAID-5 but includes a second parity scheme that is distributed across different drives and thus offers extremely high fault- and drive-failure tolerance.

g)RAID-7: This type includes a real-time embedded operating system as a controller, caching via a high-speed bus, and other characteristics of a stand-alone computer. One vendor offers this system.

h)RAID-10: Combining RAID-0 and RAID-1 is often referred to as RAID-10, which offers higher performance than RAID-1 but at much higher cost. There are two subtypes: In RAID-0+1, data is organized as stripes across multiple disks, and then the striped disk sets are mirrored. In RAID-1+0, the data is mirrored and the mirrors are striped.

i)RAID-50 (or RAID-5+0): This type consists of a series of RAID-5 groups and striped in RAID-0 fashion to improve RAID-5 performance without reducing data protection.

j)RAID-53 (or RAID-5+3): This type uses striping (in RAID-0 style) for RAID-3's virtual disk blocks. This offers higher performance than RAID-3 but at much higher cost.

κ)RAID-S (also known as Parity RAID): This is an alternate, proprietary method for striped parity RAID from EMC Symmetrix that is no longer in use on current equipment. It appears to be similar to RAID-5 with some performance enhancements as well as the enhancements that come from having a high-speed disk cache on the disk.

# UNIT-3

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

4. Interrupt Initiated I/O

5. 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:

1 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.

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

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

4 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.

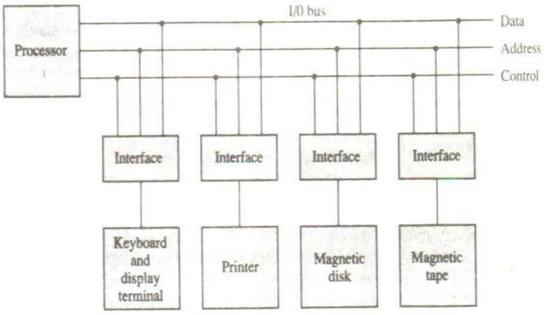


Figure -Connection of I/O bus to input devices.

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:

1. Use two separate buses, one for memory and the other for I/O.
2. Use one common bus for both memory and I/O but have separate control lines for each.
3. 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

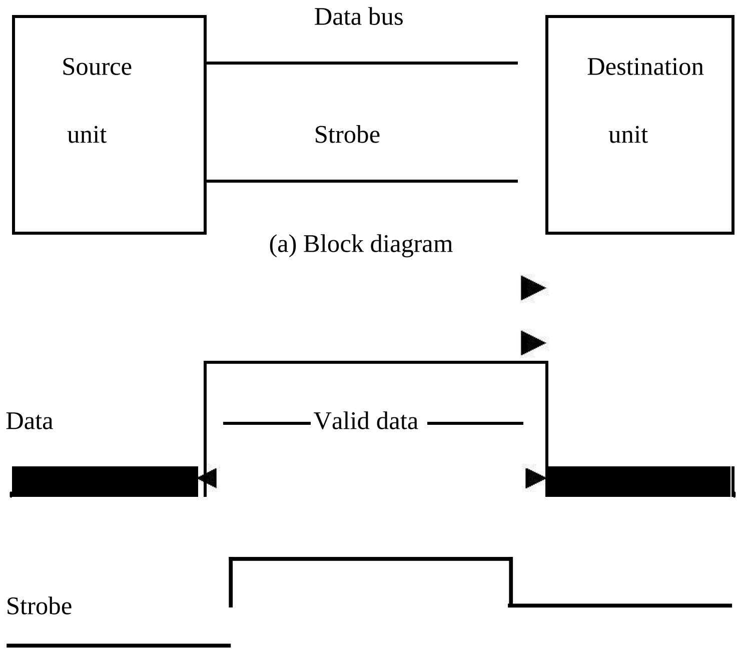
The internal operations in a digital system are synchronized by means of clock pules supplied by a common pulse generator. Clock pulses are applied to all registers within a unit and all data transfers internal registers occur simultaneously during the occurrence of a clock pulse. Two units, such as a CPU and an I/ interface, are designed independently of each other. If the registers in the interface share a common clock with the CU registers, the transfer between the two units is said to be synchronous. In most cases, the internal timing in each unit is inependent 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.



(a) Block diagram

(b) Timing diagram

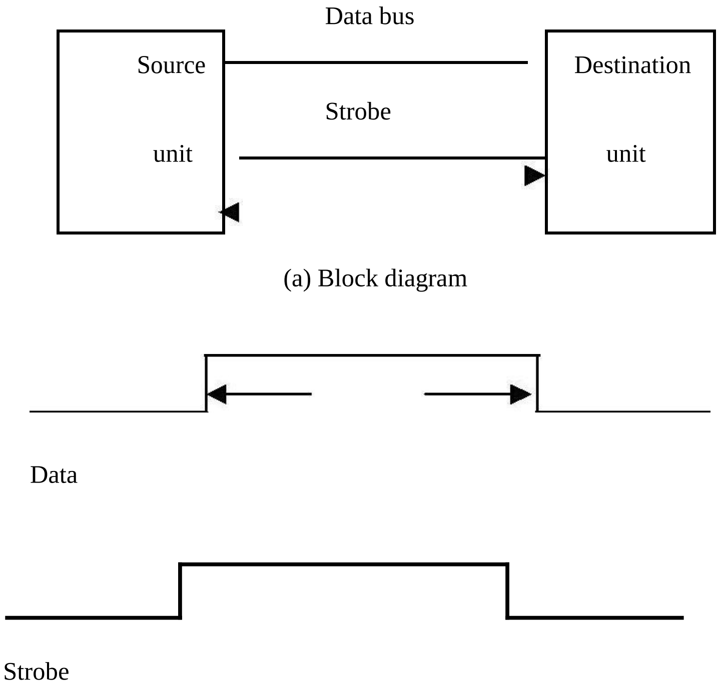
Figure- Source-initiated strobe for data transfer.

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.



(a) Block diagram

(b) 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 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.

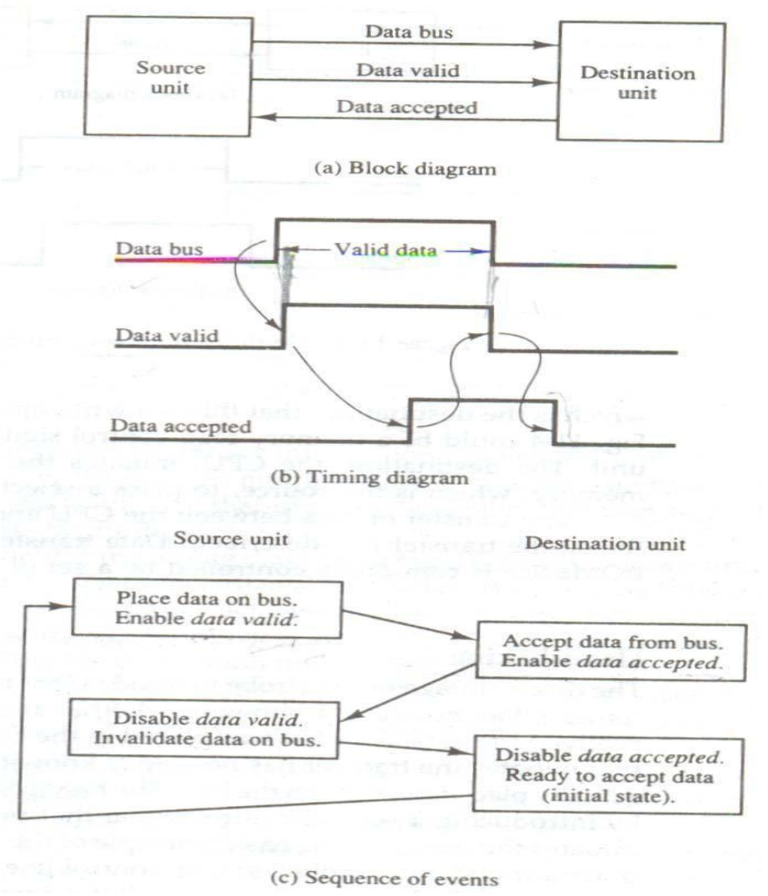

Source unit
Data bus
Data valid
Data accepted
Destination unit
(a) Block diagram
Data bus
Valid data
Data valid
Data accepted
(b) Timing diagram
Source unit
Destination unit
Place data on bus. Enable data valid.
Accept data from bus. Enable data accepted.
Disable data valid. Invalidate data on bus.
Disable data accepted. Ready to accept data (initial state).
(c) Sequence of events

**Figure -**Source-initiated transfer using handshaking.

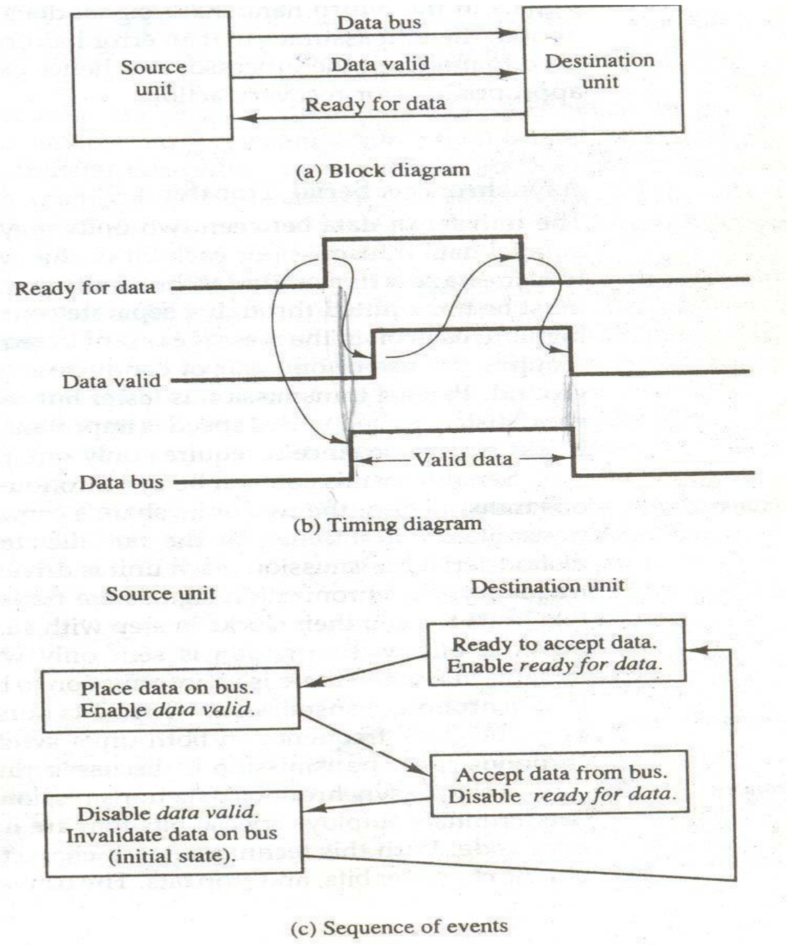

Source unit
Data bus
Data valid
Ready for data
Destination unit
(a) Block diagram
Ready for data
Data valid
Data bus
Valid data
(b) Timing diagram
Source unit
Destination unit
Ready to accept data. Enable ready for data.
Place data on bus. Enable data valid.
Accept data from bus. Disable ready for data.
Disable data valid. Invalidate data on bus (initial state).
(c) Sequence of events

**Figure**-Destination-initiated transfer using handshaking.

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

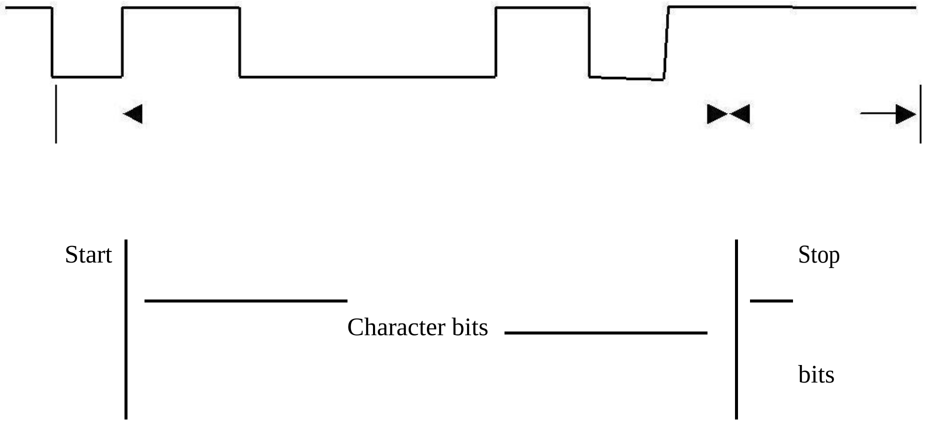


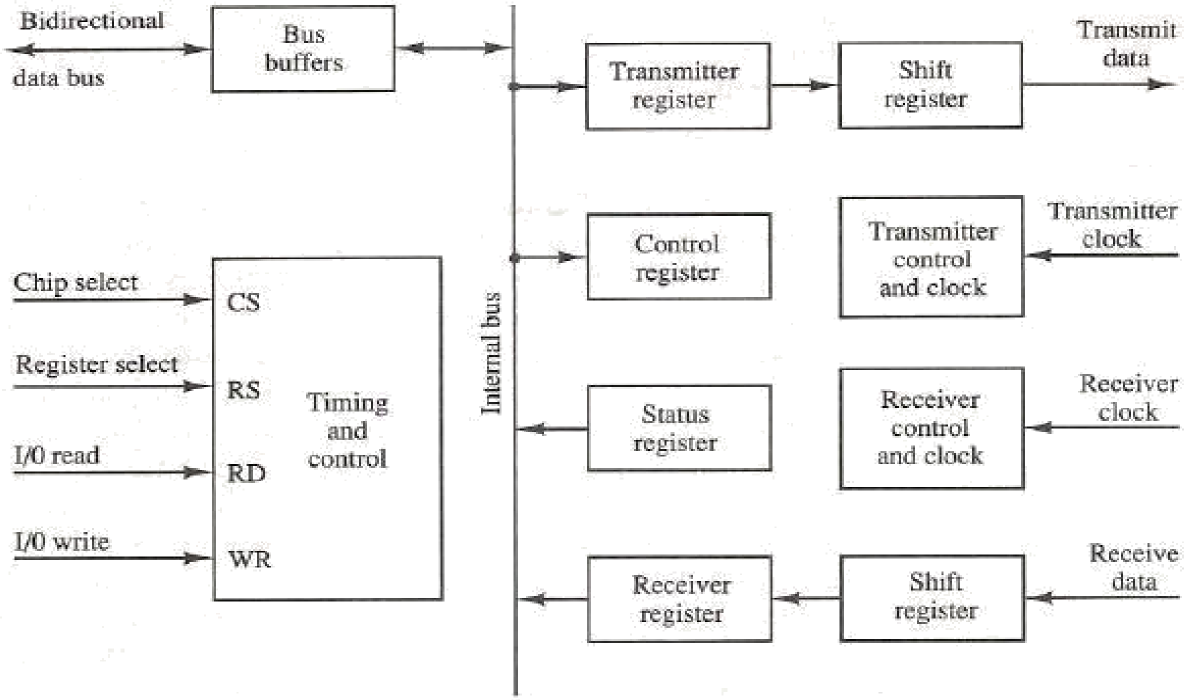
Figure - Asynchronous serial transmission

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:

1. Programmed I/O

2. 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.

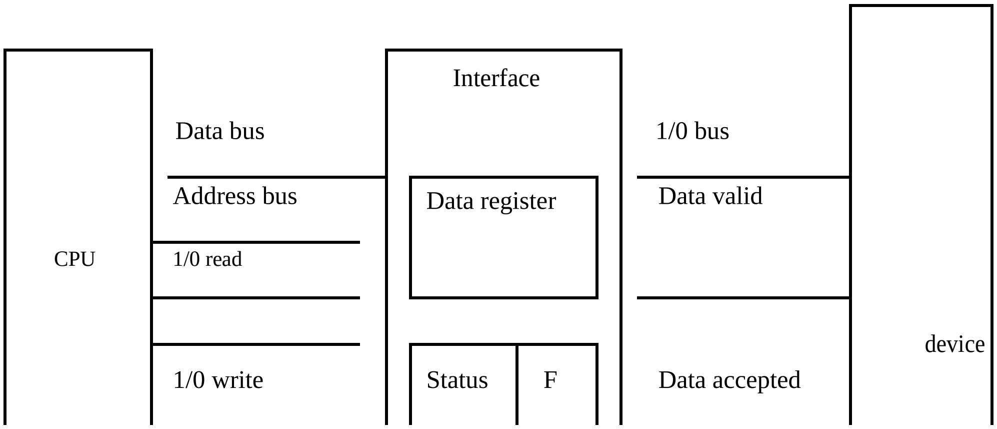
An example of data transfer from an I/O device through an interface into the CPU is shown in below Fig. The device transfers bytes of data one at a time as they are available. When a byte of data is available, the device places it in the I/O bus and enables its data valid line. The interface accepts 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.

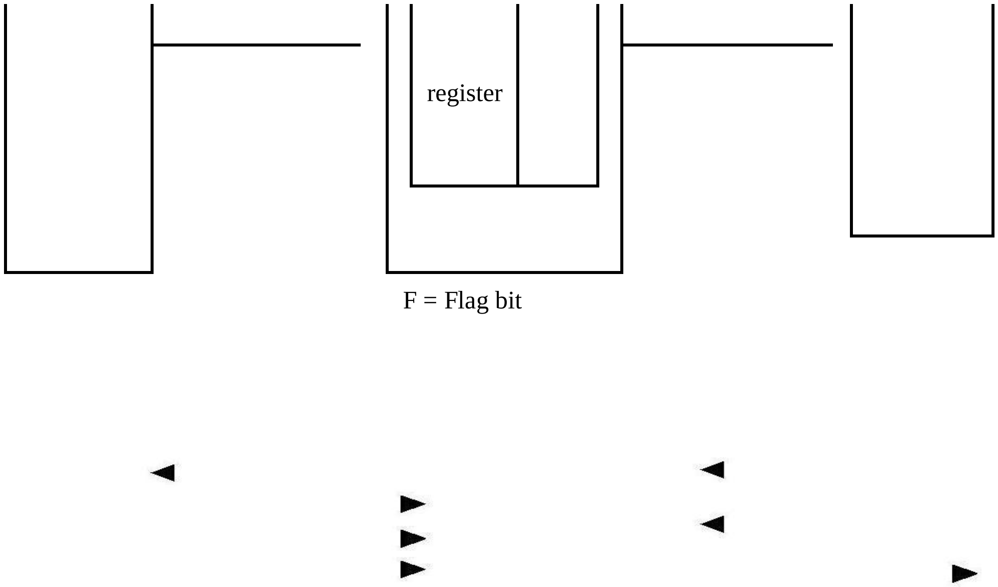
A flowchart of the program that must be written for the CPU is shown in Fig. Flowcharts for CPU program to input data. It is assumed that the device is sending a sequence of bytes that must be stored in memory. The transfer of each byte requires three instructions:

1. Read the status register.
2. Check the status of the flag bit and branch to step 1 if not set or to step 3 if set.
3. 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 theinstructions mentioned in the earlier chapter.

**Figure -**Data transfer form I/O device to CPU





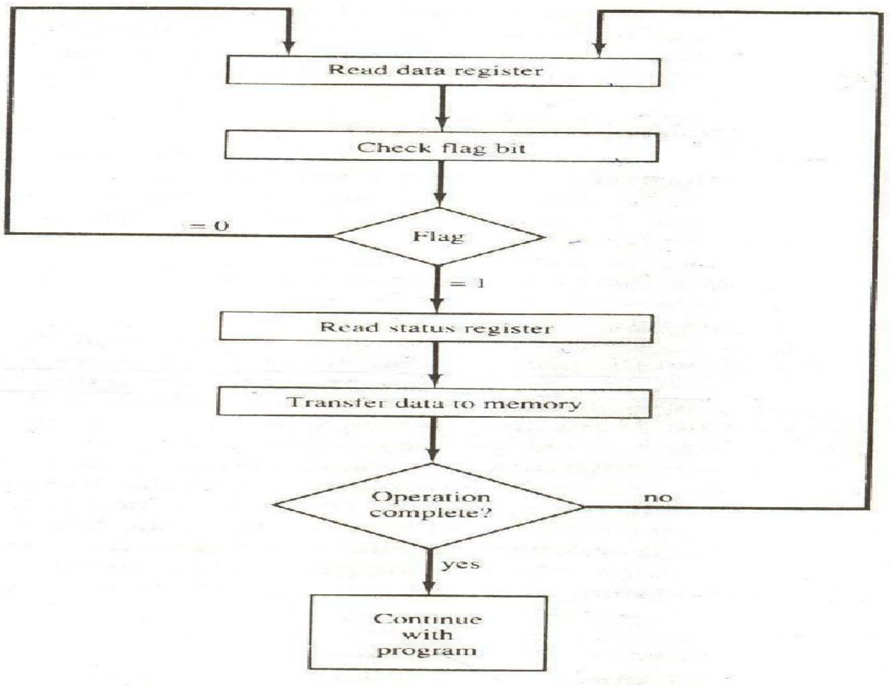


**Figure -**Flowcharts for CPU program to input data

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

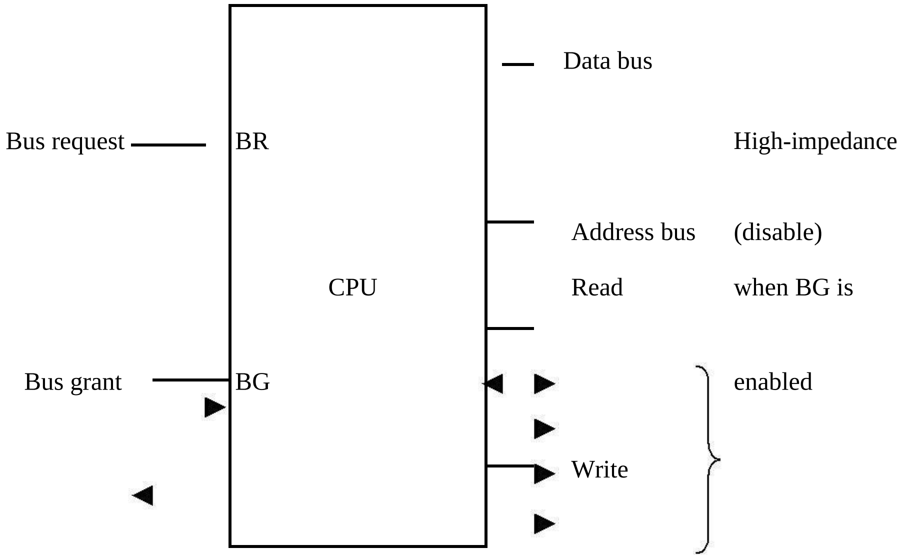


**Figure** -CPU bus signals for DMA transfer.

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.

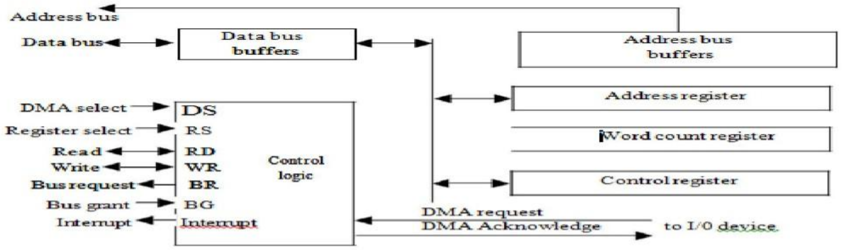


Figure-Block diagram of DMA controller.

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:

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

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

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

4. 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.

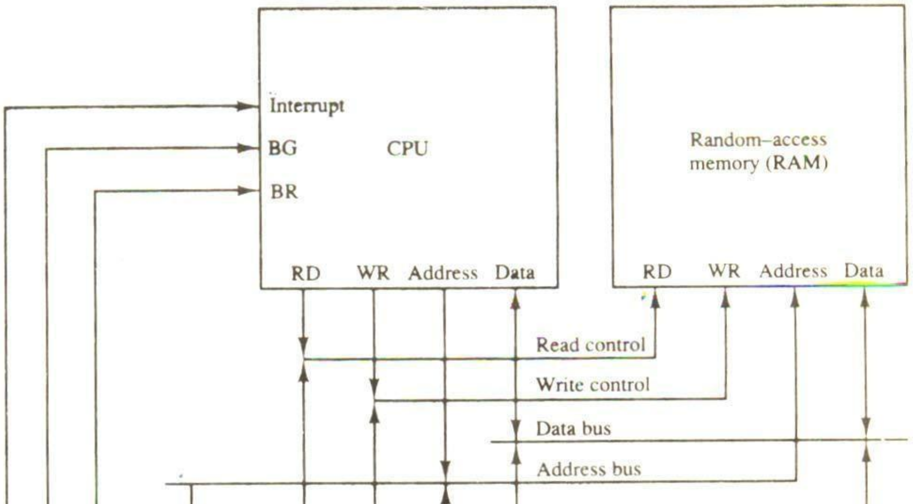


**Figure-**DMA transfer in a computer system.

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.

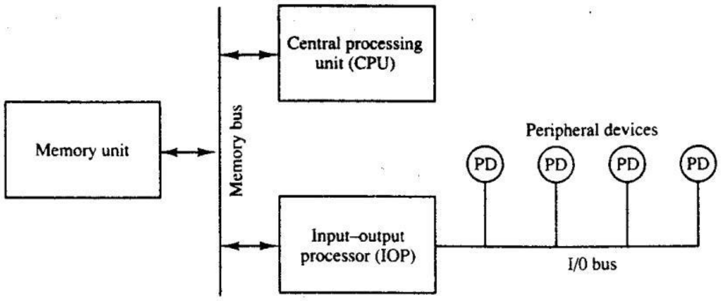


Figure- Block diagram of a computer with I/O processor

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.

## CPU-IOP Communication

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.

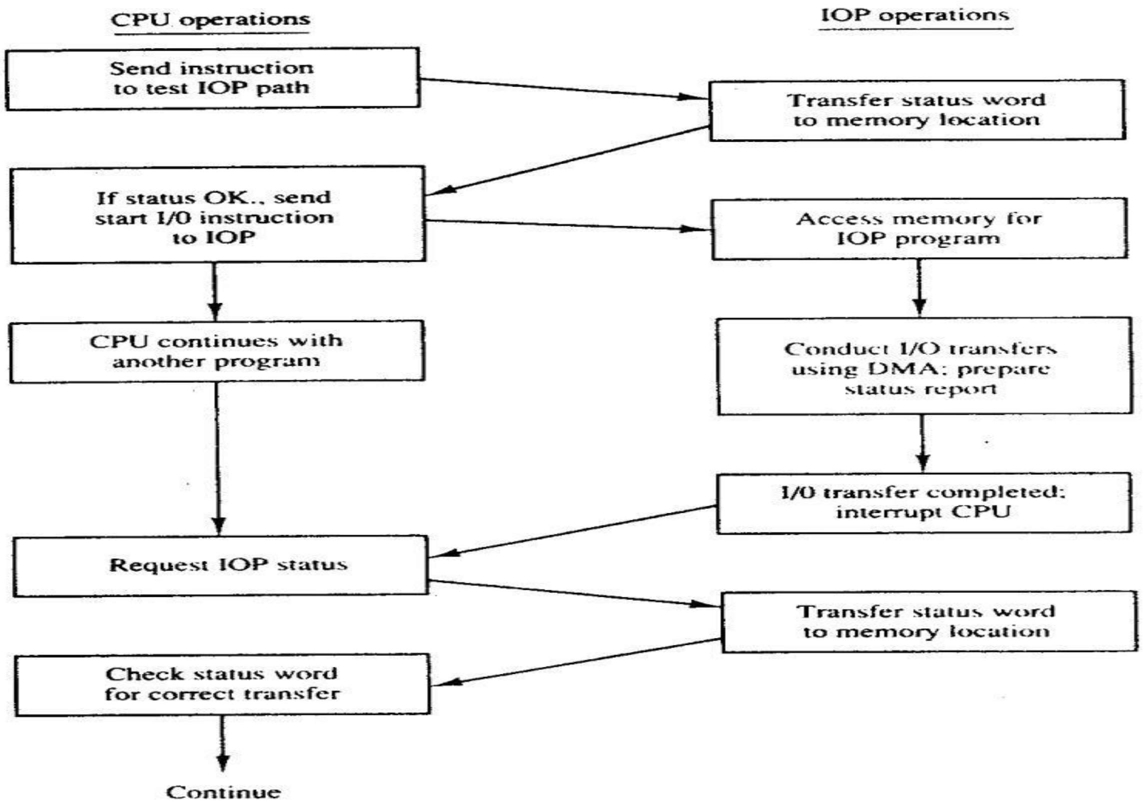

CPU operations
IOP operations
Send instruction to test IOP path
Transfer status word to memory location
If status OK., send start I/0 instruction to IOP
Access memory for IOP program
CPU continues with another program
Conduct I/O transfers using DMA; prepare status report
I/0 transfer completed; interrupt 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.

## INTEL 8089 IOP

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

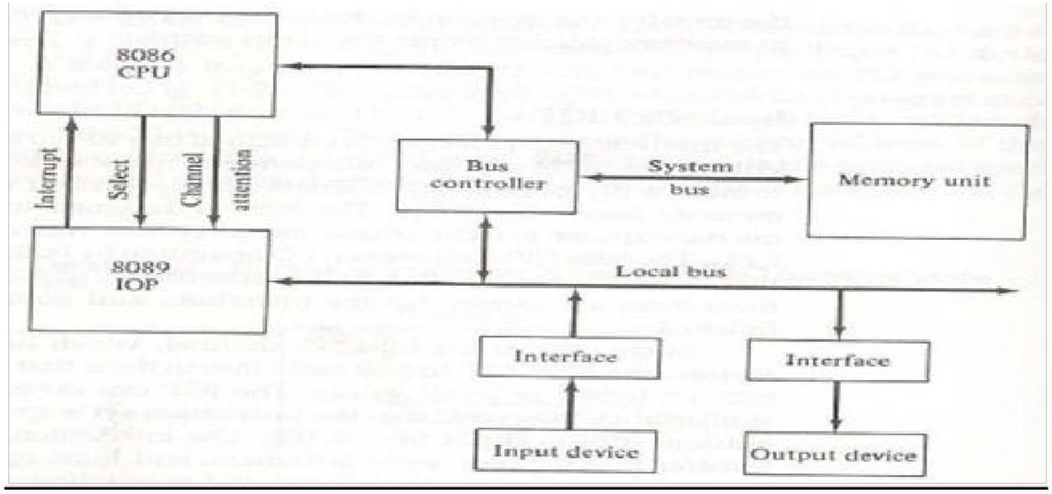


**Fig (a)- Intel 8086/8089 Microcomputer system block diagram.**

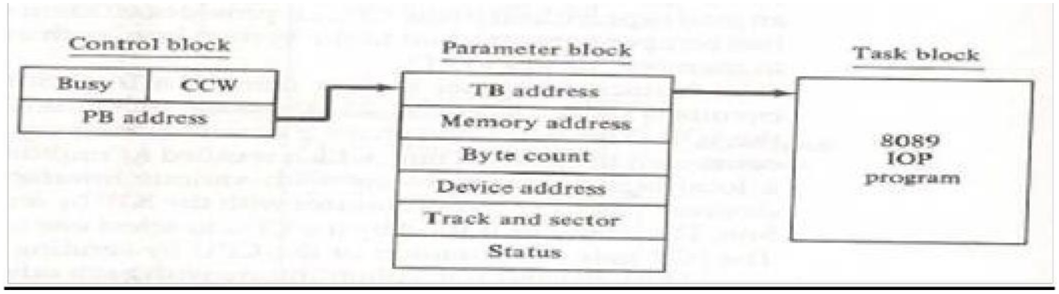


**Fig (b) – Location of information in memory for I/O operations in the Intel8086/8089 microcomputer system.**

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/Odevice. Theparameters 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.

# OVER VIEW OF OPERATING SYSTEM

**What is an Operating System**?

A program that acts as an intermediary between a user of a computer and the

computer hardware

Operating system goals:

Execute user programs and make solving user problems easier  Make the computer system convenient to use

Use the computer hardware in an efficient manner

## Computer System Structure

Computer system can be divided into four components

- Hardware – provides basic computing resources  CPU, memory, I/O devices

Operating system

Controls and coordinates use of hardware among various applications and users

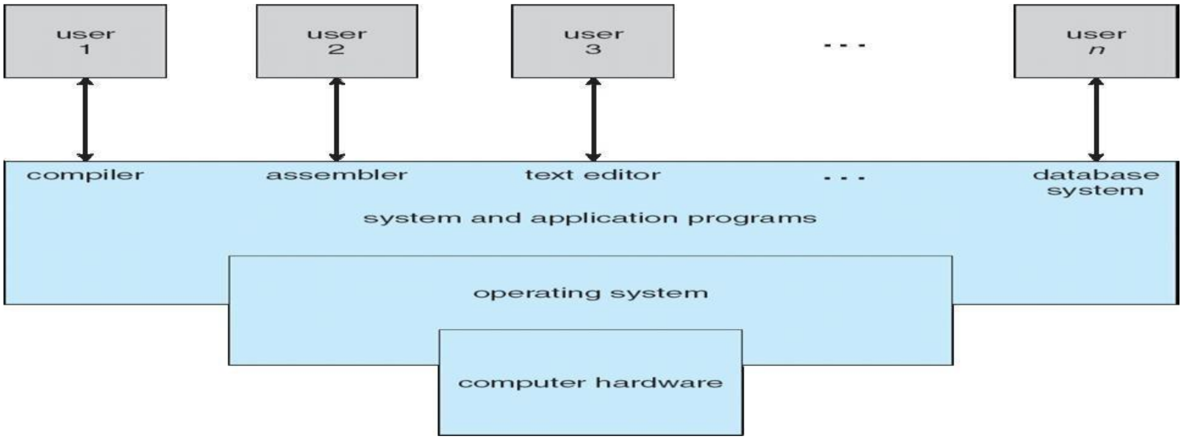
- Application programs – define the ways in which the system resources are used to solve the computing problems of the users

Word processors, compilers, web browsers, database systems, video games

Users

People, machines, other computers

### Four Components of a Computer System



### Operating System Definition

OS is a **resource allocator**

Manages all resources

Decides between conflicting requests for efficient and fair resource use

OS is a **control program**

Controls execution of programs to prevent errors and improper use of the computer  No universally accepted definition

- Everything a vendor ships when you order an operating system‖  is good approximation

But varies wildly

- —The one program running at all times on the computer‖  is the **kernel.** Everything else is either a system program (ships with the operating system) or an application program

## Computer Startup

**bootstrap program** is loaded at power-up or reboot

Typically stored in ROM or EPROM, generally known as **firmware**  Initializes all aspects of system

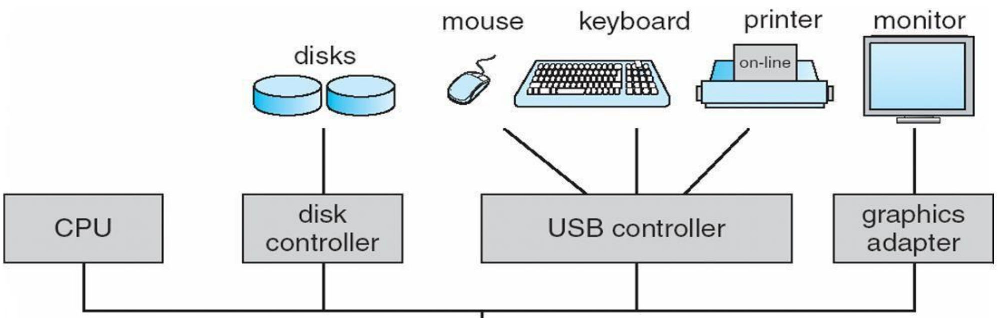
Loads operating system kernel and starts execution

## Computer System Organization

Computer-system operation

One or more CPUs, device controllers connect through common bus providing access to shared memory

Concurrent execution of CPUs and devices competing for memory cycles



## Computer-System Operation

I/O devices and the CPU can execute concurrently

Each device controller is in charge of a particular device type  Each device controller has a local buffer

CPU moves data from/to main memory to/from local buffers  I/O is from the device to local buffer of controller

Device controller informs CPU that it has finished its operation by causing An *interrupt*

## Common Functions of Interrupts

Interrupt transfers control to the interrupt service routine generally, through the **interrupt vector**, which contains the addresses of all the service routines

Interrupt architecture must save the address of the interrupted instruction

Incoming interrupts are *disabled* while another interrupt is being processed to prevent a *lost interrupt*nA *trap* is a software-generated interrupt caused either by an error or a user request

An operating system is **interrupt driven**

## Interrupt Handling

The operating system preserves the state of the CPU by storing registers and the program counter

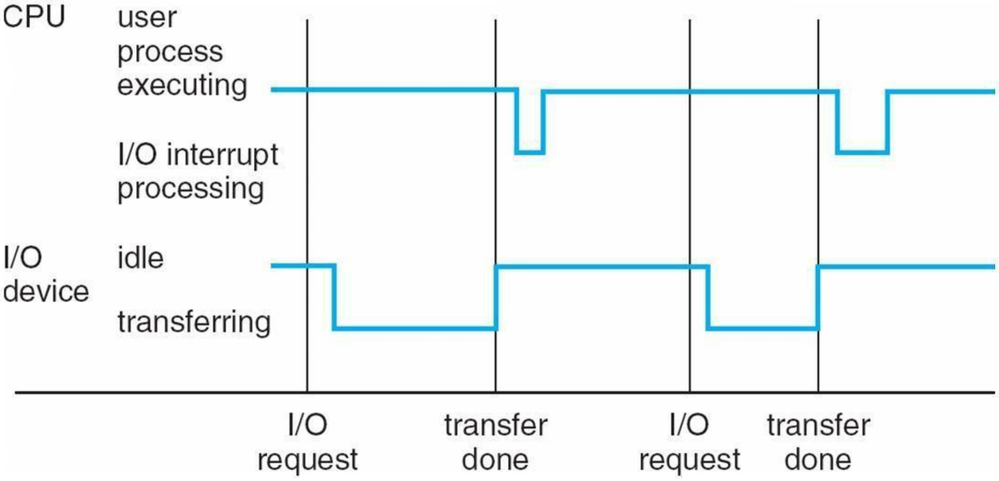
Determines which type of interrupt has occurred:

**polling**

**vectored** interrupt system

Separate segments of code determine what action should be taken for each type of interrupt

## Interrupt Timeline



## I/O Structure

After I/O starts, control returns to user program only upon I/O completion  Wait instruction idles the CPU until the next interrupt

Wait loop (contention for memory access)

At most one I/O request is outstanding at a time, no simultaneous I/O processing After I/O starts, control returns to user program without waiting for I/O completion

- **System call** – request to the operating system to allow user to wait for I/O completion **Device-status table** contains entry for each I/O device indicating its type, address, and

**State** Operating system indexes into I/O device table to determine device status and to modify table entry to include interrupt

### Direct Memory Access Structure

Used for high-speed I/O devices able to transmit information at close to memory speeds

Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention

Only one interrupt is generated per block, rather than the one interrupt per byte

## Storage Structure

- Main memory – only large storage media that the CPU can access directly
- Secondary storage – extension of main memory that provides large nonvolatile storage capacity
- Magnetic disks – rigid metal or glass platters covered with magnetic recording material

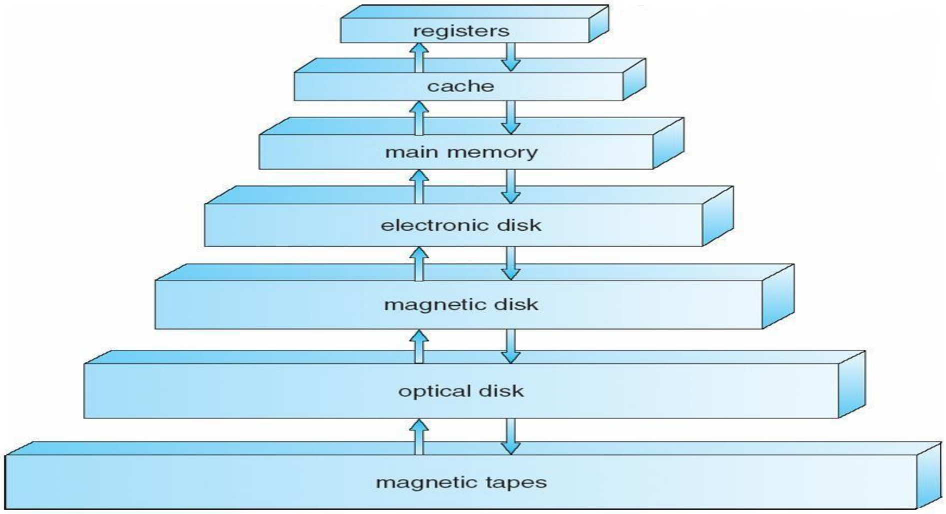
Disk surface is logically divided into **tracks**, which are subdivided into **sectors**

The **disk controller** determines the logical interaction between the device and the computer

## Storage Hierarchy

Storage systems organized in
hierarchy Speed Cost Volatility

**Caching** – copying information into faster storage system; main memory can be viewed as a last *cache* for secondary storage



## Caching

Important principle, performed at many levels in a computer (in hardware, operating system, software)

Information in use copied from slower to faster storage temporarily

Faster storage (cache) checked first to determine if information is there If it is, information used directly from the cache (fast)

If not, data copied to cache and used there
Cache smaller than storage being cached

Cache management important design problem Cache size and replacement policy

## Computer-System Architecture

Most systems use a single general-purpose processor (PDAs through mainframes) Most systems have special-purpose processors as well

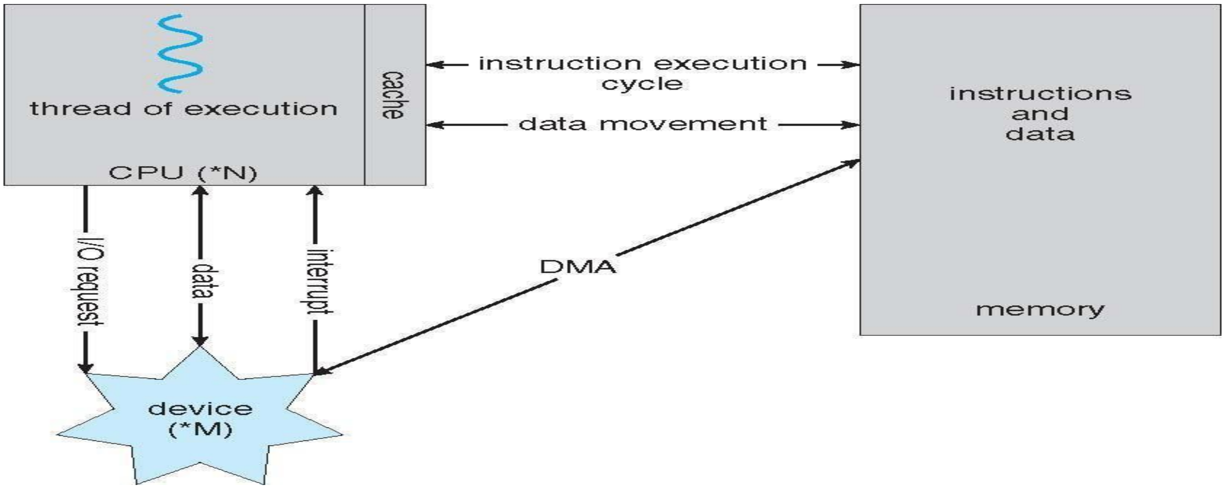
Multiprocessors systems growing in use and importance
Also known as parallel systems, tightly-coupled systems

Advantages include

1. Increased throughput
2. Economy of scale
3. Increased reliability – graceful degradation or fault tolerance

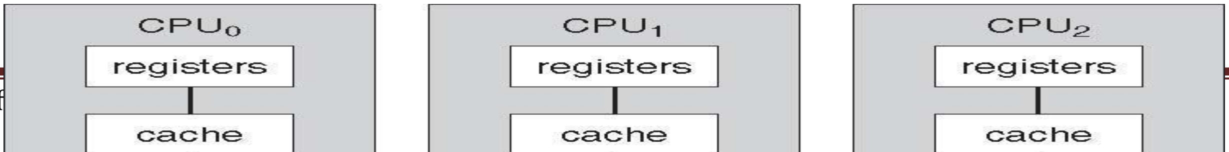
Two types

1. Asymmetric Multiprocessing
2. Symmetric Multiprocessing

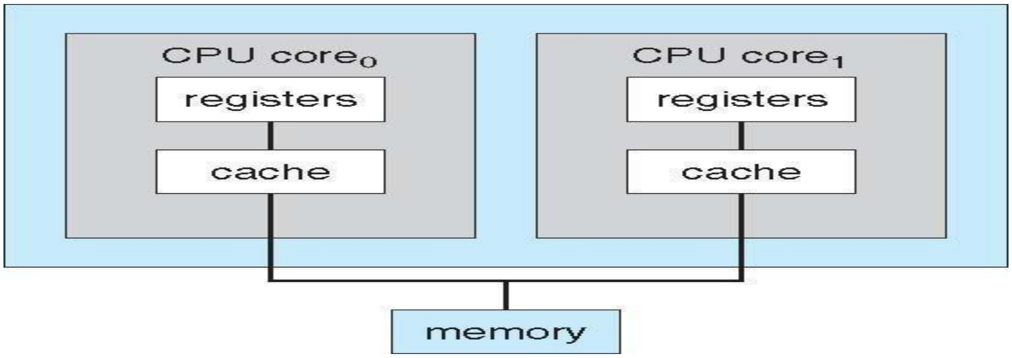


## How a Modern Computer Works

## Symmetric Multiprocessing Architecture



## A Dual-Core Design



## Clustered Systems

Li ke multiprocessor systems, but multiple systems working
together Usually sharing storage via a storage-area network (SAN)

Provides a high-availability service which survives failures
Asymmetric clustering has one machine in hot-standby mode

Symmetric clustering has multiple nodes running applications, monitoring each other

Some clusters are for high-performance computing (HPC)
Applications must be written to use parallelization

## Operating System Structure

**Multiprogramming** needed for efficiency

Single user cannot keep CPU and I/O devices busy at all times

Multiprogramming organizes jobs (code and data) so CPU always has one to
Execute  A subset of total jobs in system is kept in memory

One job selected and run via **job scheduling**

When it has to wait (for I/O for example), OS switches to another job

**Timesharing (multitasking)** is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing

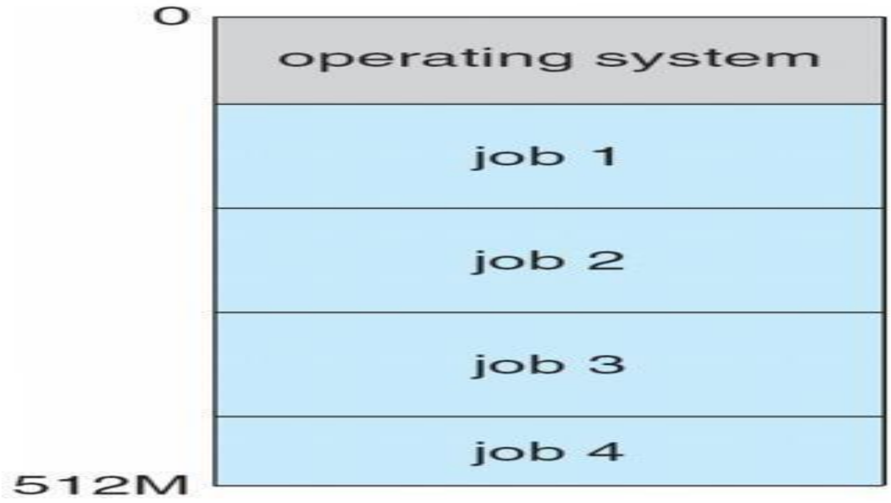
**Response time** should be < 1 second

Each user has at least one program executing in memory [**process** If several jobs ready to run at the same time [ **CPU scheduling**

- If processes don't fit in memory, **swapping** moves them in and out to run

**Virtual memory** allows execution of processes not completely in memory

## Memory Layout for Multiprogrammed System



## Operating-System Operations

Interrupt driven by hardware

Software error or request creates **exception** or **trap**
Division by zero, request for operating system service

Other process problems include infinite loop, processes modifying each Other or the operating system

**Dual-mode** operation allows OS to protect itself and other system components

**User mode** and **kernel mode**

**Mode bit** provided by hardware

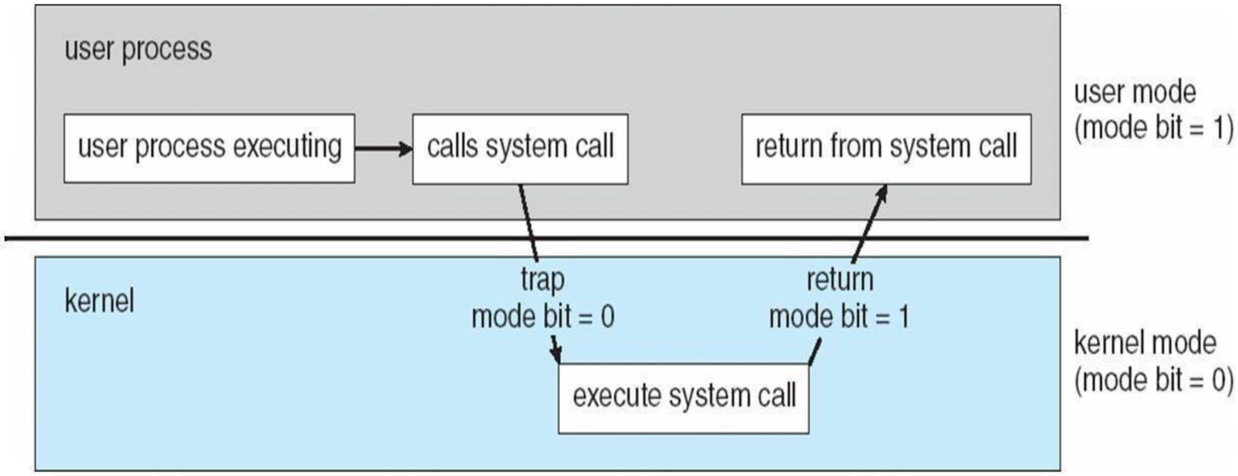
Provides ability to distinguish when system is running user code or kernel code Some instructions designated as **privileged**, only executable in kernel mode System call changes mode to kernel, return from call resets it to user

## Transition from User to Kernel Mode

Timer to prevent infinite loop / process hogging resources  Set interrupt after specific period

Operating system decrements counter
When counter zero generate an interrupt

Set up before scheduling process to regain control or terminate program that exceeds allotted time



# OPERATING SYSTEM FUNCTIONS

## Process Management

A process is a program in execution. It is a unit of work within the system. Program is a *passive entity*, process is an *active entity*.

Process needs resources to accomplish its task CPU, memory, I/O, files

Initialization data

Process termination requires reclaim of any reusable resources

Single-threaded process has one **program counter** specifying location of next instruction to execute

Process executes instructions sequentially, one at a time, until completion Multi-threaded process has one program counter per thread

Typically system has many processes, some user, some operating system running concurrently on one or more CPUs

Concurrency by multiplexing the CPUs among the processes / threads

## Process Management Activities

The operating system is responsible for the following activities in connection with process management:

Creating and deleting both user and system processes Suspending and resuming processes

Providing mechanisms for process synchronization Providing mechanisms for process communication Providing mechanisms for deadlock handling

## Memory Management

All data in memory before and after processing All instructions in memory in order to execute

Memory management determines what is in memory when Optimizing CPU utilization and computer response to users

## Memory management activities

Keeping track of which parts of memory are currently being used and by whom. Deciding which processes (or parts thereof) and data to move into and out of memory  Allocating and deal locating memory space as needed.

## Storage Management

OS provides uniform, logical view of information storage  Abstracts physical properties to logical storage unit – **file.** Each medium is controlled by device (i.e., disk drive, tape drive)

- Varying properties include access speed, capacity, data-transfer rate, access method (sequential or random)

File-System management

Files usually organized into directories

Access control on most systems to determine who can access what

## OS activities include

- Creating and deleting files and directories
- Primitives to manipulate files and dirs
- Mapping files onto secondary storage
- Backup files onto stable (non-volatile) storage media

## Mass-Storage Management

- Usually disks used to store data that does not fit in main memory or data that must be kept for a —long‖  period of time

Proper management is of central importance

Entire speed of computer operation hinges on disk subsystem and its algorithms

## MASS STORAGE activities

Free-space management

Storage allocation

Disk scheduling

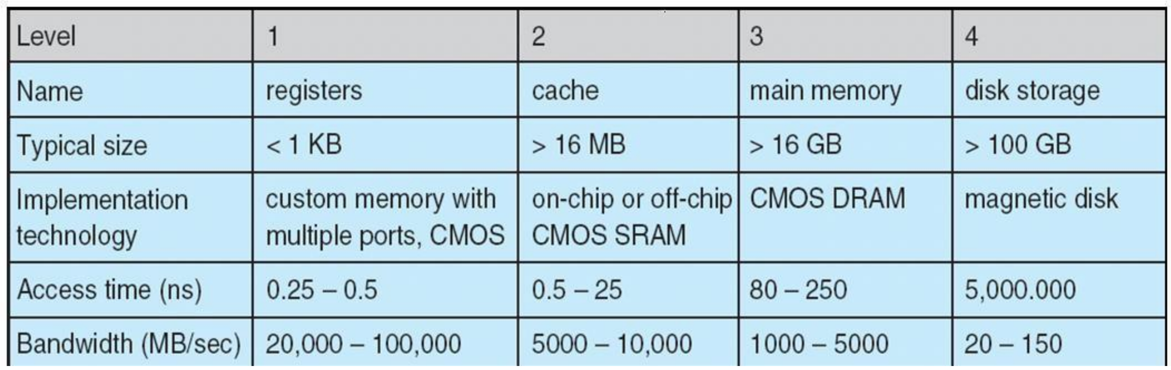
Some storage need not be fast

Tertiary storage includes optical storage, magnetic tape

Still must be managed

Varies between WORM (write-once, read-many-times) and RW (read-write)

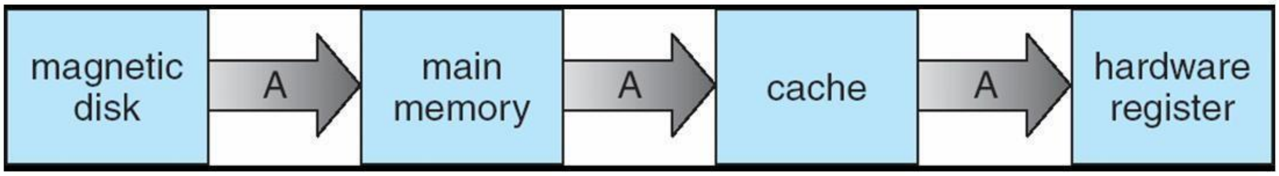
### Performance of Various Levels of Storage



| Level | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Name | registers | cache | main memory | disk storage |
| Typical size | < 1 KB | > 16 MB | > 16 GB | > 100 GB |
| Implementation technology | custom memory with multiple ports, CMOS | on-chip or off-chip CMOS SRAM | CMOS DRAM | magnetic disk |
| Access time (ns) | 0.25 – 0.5 | 0.5 – 25 | 80 – 250 | 5,000.000 |
| Bandwidth (MB/sec) | 20,000 – 100,000 | 5000 – 10,000 | 1000 – 5000 | 20 – 150 |

# Migration of Integer A from Disk to Register

Multitasking environments must be careful to use most recent value, no matter where it is stored in the storage hierarchy



Multiprocessor environment must provide cache coherency in hardware such that all CPUs have the most recent value in their cache. Distributed environment situation even more complex  Several copies of a datum can exist.

### I/O Subsystem

One purpose of OS is to hide peculiarities of hardware devices from the user  I/O subsystem responsible for . Memory management of I/O including buffering (storing data temporarily while it is being transferred), caching (storing parts of data in faster storage for performance), spooling (the overlapping of output of one job with input of other jobs)

General device-driver interface

Drivers for specific hardware devices

## Protection and Security

**Protection** – any mechanism for controlling access of processes or users to resources defined by the OS

**Security** – defense of the system against internal and external attacks

Huge range, including denial-of-service, worms, viruses, identity theft, theft of service Systems generally first distinguish among users, to determine who can do what

User identities (**user IDs**, security IDs) include name and associated number, one per user .User ID then associated with all files, processes of that user to determine access control . Group identifier (**group ID**) allows set of users to be defined and controls managed, then also associated with each process, file .

**Privilege escalation** allows user to change to effective ID with more rights

# DISTRIBUTED SYSTEMS

## Computing Environments

### Traditional computer

Blurring over time  Office environment

PCs connected to a network, terminals attached to mainframe or minicomputers providing batch and timesharing

Now portals allowing networked and remote systems access to same resources

Home networks

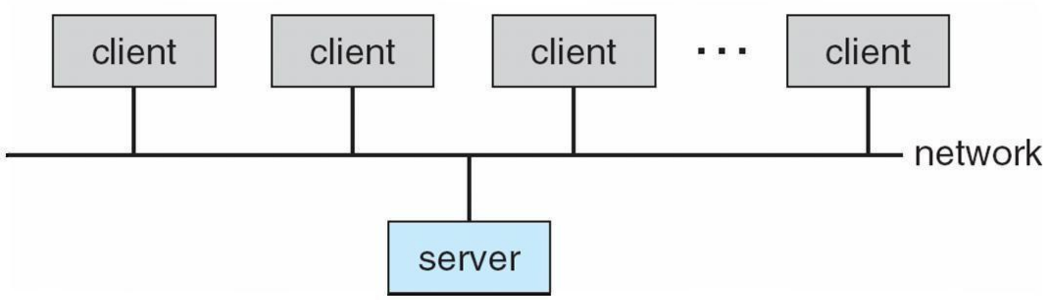
Used to be single system, then modems

Now firewalled, networked

Client-Server Computing

Dumb terminals supplanted by smart PCs

Many systems now **servers**, responding to requests generated by **clients**
**Compute-server** provides an interface to client to request services (i.e. database)
**File-server** provides interface for clients to store and retrieve files



### Peer-to-Peer Computing

Another model of distributed system

P2P does not distinguish clients and servers
Instead all nodes are considered peers

May each act as client, server or
both Node must join P2P network

Registers its service with central
lookup service on network,

or

Broadcast request for service and
respond to requests for service via
**discovery protocol**

Examples include *Napster* and *Gnutella*

## Web-Based Computing

Web has become ubiquitous
PCs most prevalent devices

More devices becoming networked to allow web access

New category of devices to manage web traffic among similar servers: **load balancers**
Use of operating systems like Windows 95, client-side, have evolved into Linux and

Windows XP, which can be clients and servers

## Open-Source Operating Systems

Operating systems made available in source-code format rather than just binary closed-source

Counter to the copy protection and Digital Rights Management (DRM) movement

- Started by Free Software Foundation (FSF), which has —copyleft|| GNU Public License (GPL)

Examples include GNU/Linux, BSD UNIX (including core of Mac OS X), and
Sun Solaris

## Operating System Services

One set of operating-system services provides functions that are helpful to
the user:

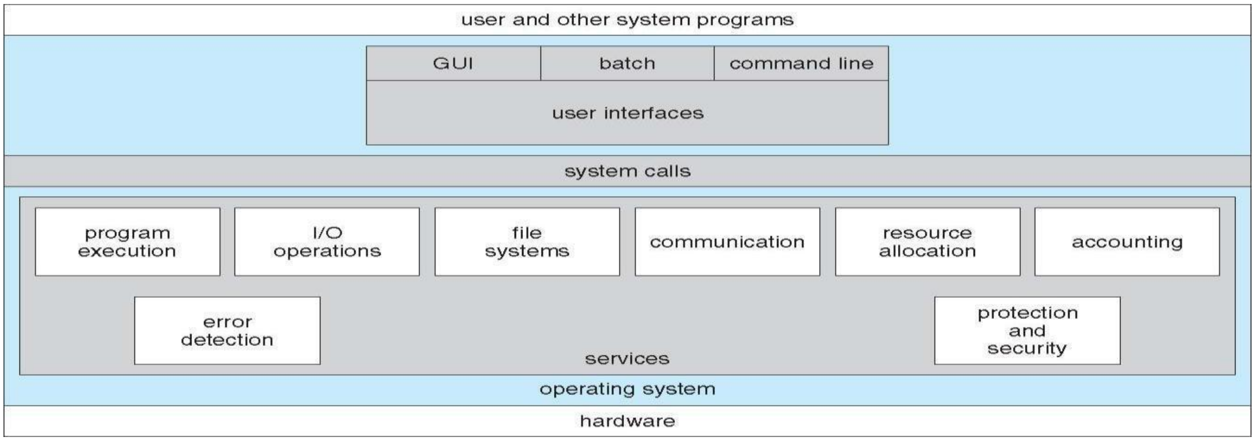
User interface - Almost all operating systems have a user interface (UI)

•Varies between Command-Line (CLI), Graphics User Interface (GUI), Batch

Program execution - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)

I/O operations - A running program may require I/O, which may involve a file or an I/O device .File-system manipulation - The file system is of particular interest. Obviously, programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.

## A View of Operating System Services



## Operating System Services

- One set of operating-system services provides functions that are helpful to the user (Cont):Communications – Processes may exchange information, on the same computer or between computers over a network .Communications may be via shared memory or through message passing (packets moved by the OS)

- Error detection – OS needs to be constantly aware of possible errors

May occur in the CPU and memory hardware, in I/O devices, in user program

For each type of error, OS should take the appropriate action to ensure correct and consistent computing. Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system. Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing

**Resource allocation -** When multiple users or multiple jobs running concurrently, resources must be allocated to each of them

•Many types of resources - Some (such as CPU cycles, main memory, and file storage) may have special allocation code, others (such as I/O devices) may have general request and release code

**Accounting -** To keep track of which users use how much and what kinds of computer resources

**Protection and security -** The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other

**Protection** involves ensuring that all access to system resources is controlled

**Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts . If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

## User Operating System Interface - CLI

Command Line Interface (CLI) or command interpreter allows direct command entry  Sometimes implemented in kernel, sometimes by systems program Sometimes multiple flavors implemented – shells

Primarily fetches a command from user and executes it

Sometimes commands built-in, sometimes just names of programs

- If the latter, adding new features doesn't require shell modification

## User Operating System Interface - GUI

User-friendly desktop metaphor interface Usually mouse, keyboard, and monitor  Icons represent files, programs, actions, etc

Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a folder)

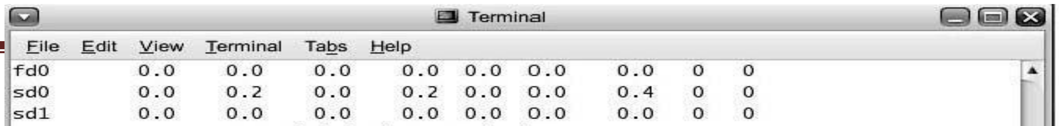
Invented at Xerox PARC

Many systems now include both CLI and GUI interfaces

- Microsoft Windows is GUI with CLI ―command‖ shell
- Apple Mac OS X as ―Aqua‖ GUI interface with UNIX kernel underneath and shells available

Solaris is CLI with optional GUI interfaces (Java Desktop, KDE)

## Bourne Shell Command Interpreter



```
fd0        0.0     0.0     0.0     0.0  0.0  0.0     0.0   0   0
sd0        0.0     0.2     0.0     0.2  0.0  0.0     0.4   0   0
sd1        0.0     0.0     0.0     0.0  0.0  0.0     0.0   0   0
```

### The Mac OS X GUI

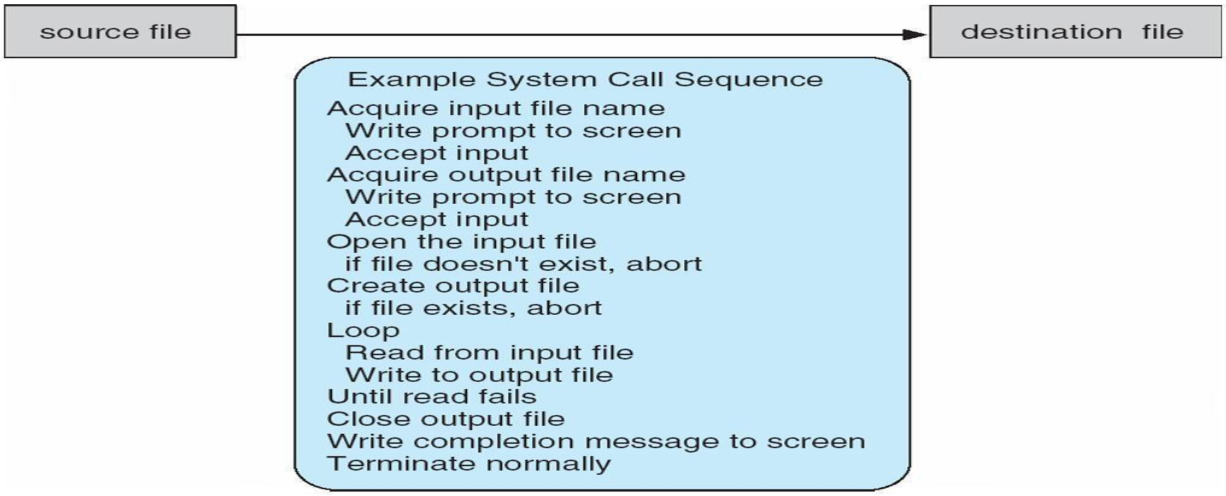
## System Calls

Programming interface to the services provided by the OS  Typically written in a high-level language (C or C++)

Mostly accessed by programs via a high-level Application Program Interface (API) rather than direct system call usenThree most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)

Why use APIs rather than system calls?(Note that the system-call names used throughout this text are generic)

### Example of System Calls

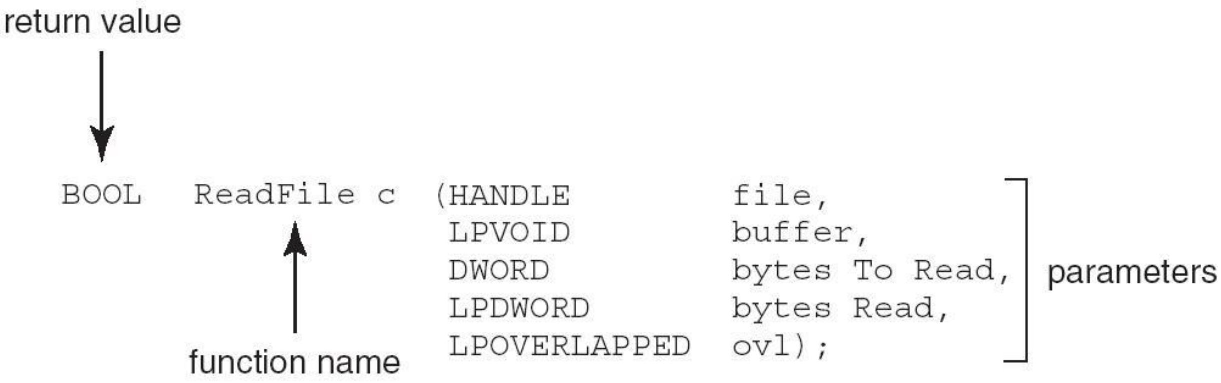


```
Acquire input file name
  Write prompt to screen
  Accept input
Acquire output file name
  Write prompt to screen
  Accept input
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
Loop
  Read from input file
  Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally
```

## Example of Standard API

Consider the ReadFile() function in the

Win32 API—a function for reading from a file



```
BOOL   ReadFile c  (HANDLE        file,
                    LPVOID        buffer,
                    DWORD         bytes To Read,
                    LPDWORD       bytes Read,
                    LPOVERLAPPED  ovl);
```

A description of the parameters passed to ReadFile()

- HANDLE file—the file to be read
- LPVOID buffer—a buffer where the data will be read into and written from
- DWORD bytesToRead—the number of bytes to be read into the buffer
- LPDWORD bytesRead—the number of bytes read during the last read
- LPOVERLAPPED ovl—indicates if overlapped I/O is being used

## System Call Implementation

Typically, a number associated with each system call

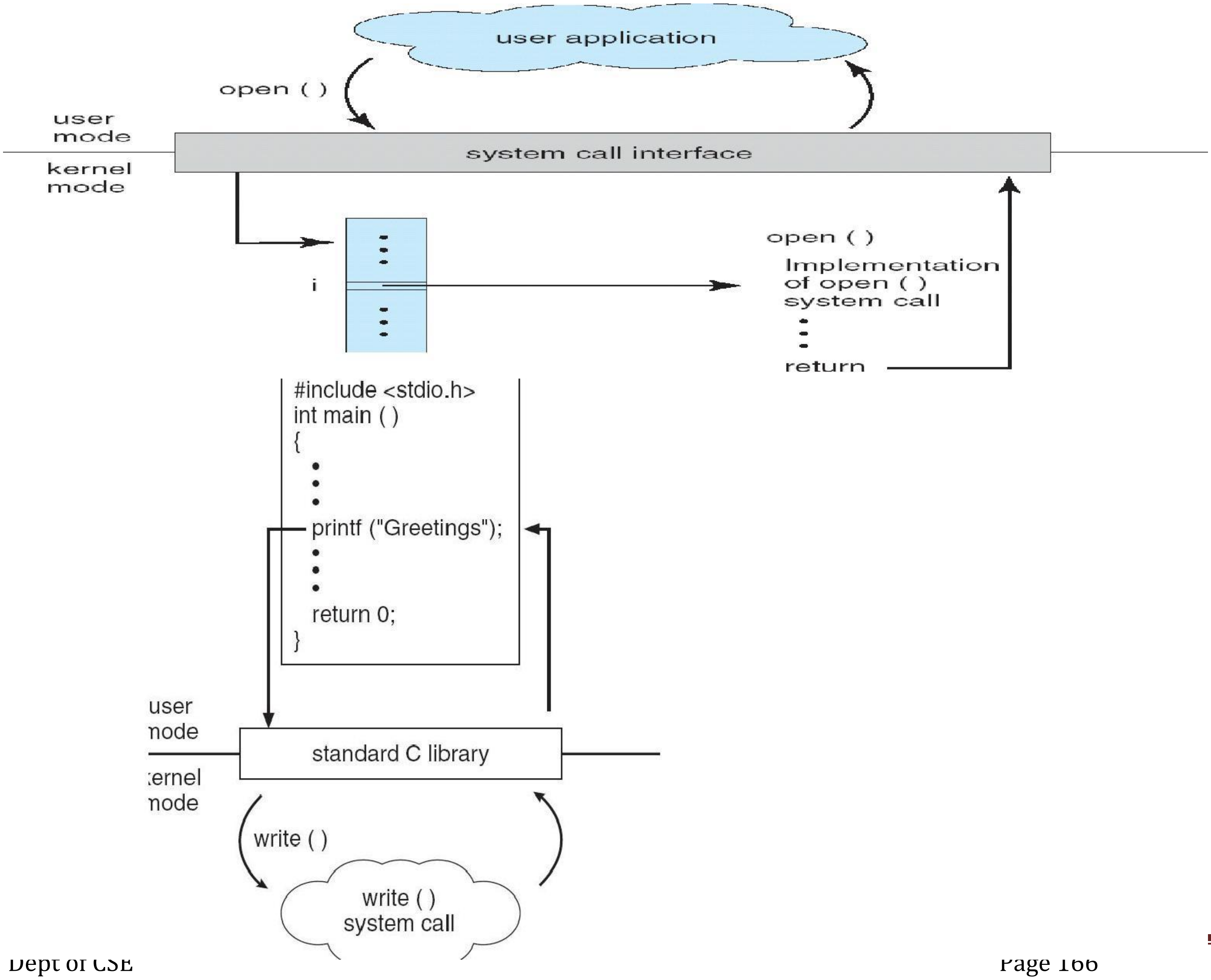
System-call interface maintains a table indexed according to these Numbers

The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values

The caller need know nothing about how the system call is implemented
Just needs to obey API and understand what OS will do as a result call
Most details of OS interface hidden from programmer by API

Managed by run-time support library (set of functions built into libraries included with compiler)

## API – System Call – OS Relationship



```
#include <stdio.h>
int main ( )
{
   •
   •
   •
   printf ("Greetings");
   •
   •
   •
   return 0;
}
```

## System Call Parameter Passing

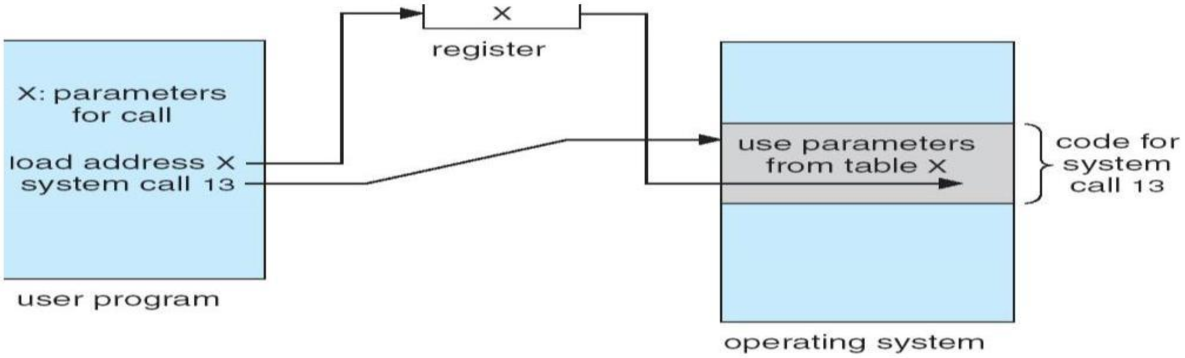
Often, more information is required than simply identity of desired system call  Exact type and amount of information vary according to OS and call

Three general methods used to pass parameters to the OS  Simplest: pass the parameters in *registers* .In some cases, may be more parameters than registers

Parameters stored in a *block,* or table, in memory, and address of block passed as a parameter in a register .This approach taken by Linux and Solaris

Parameters placed, or *pushed,* onto the *stack* by the program and *popped* off the stack by the operating system .Block and stack methods do not limit the number or length of parameters being passed .

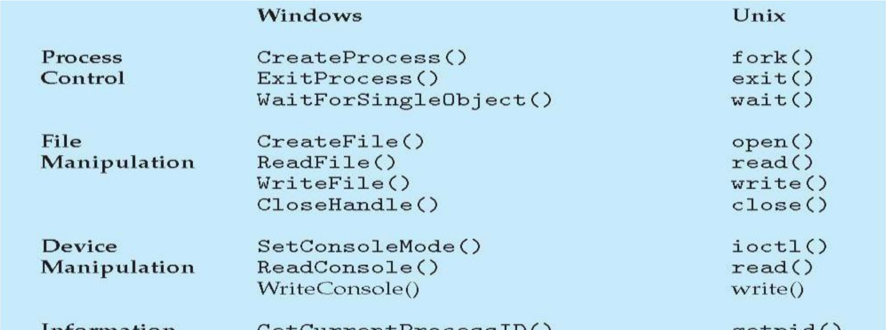
## Parameter Passing via Table



**Types of:** Pr • File De In

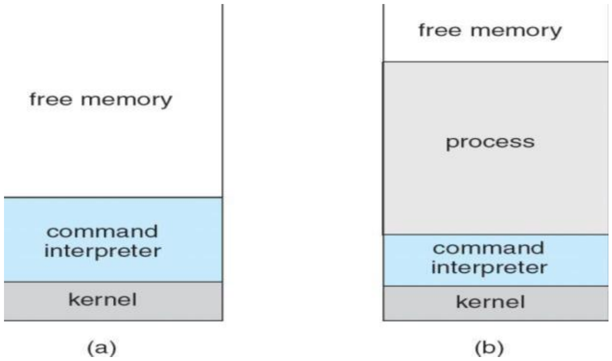
Communications  Protection

## Examples of Windows and Unix System Calls



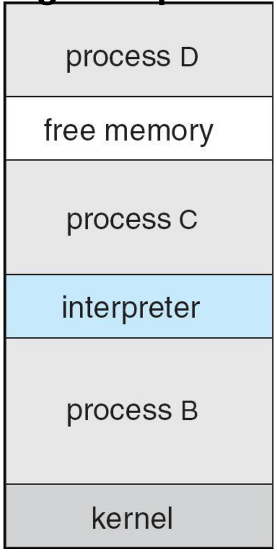
| | Windows | Unix |
|---|---|---|
| Process Control | CreateProcess()<br>ExitProcess()<br>WaitForSingleObject() | fork()<br>exit()<br>wait() |
| File Manipulation | CreateFile()<br>ReadFile()<br>WriteFile()<br>CloseHandle() | open()<br>read()<br>write()<br>close() |
| Device Manipulation | SetConsoleMode()<br>ReadConsole()<br>WriteConsole() | ioctl()<br>read()<br>write() |
| Information | GetCurrentProcessID() | getpid() |

## MS-DOS execution



(a) At system startup (b) running a program

## FreeBSD Running Multiple Programs



## System Programs

System programs provide a convenient environment for program development and execution.

The can be divided into:

File manipulation
Status information
File modification

Programming language support Program loading and execution Communications

Application programs

Most users' view of the operation system is defined by system programs, not the actual system calls

Provide a convenient environment for program development and execution

Some of them are simply user interfaces to system calls; others are considerably more complex

File management - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories

Status information

Some ask the system for info - date, time, amount of available memory, disk space, number of users

Others provide detailed performance, logging, and debugging information

Typically, these programs format and print the output to the terminal or other output devices
Some systems implement a registry - used to store and retrieve configuration information

File modification

Text editors to create and modify files

Special commands to search contents of files or perform transformations of the text
Programming-language support - Compilers, assemblers, debuggers and interpreters

sometimes provided

Program loading and execution- Absolute loaders, relocate able loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language

Communications - Provide the mechanism for creating virtual connections among processes, users, and computer systems

- Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another

## Operating System Design and Implementation

- Design and Implementation of OS not ―solvable‖ , but some approaches have proven successful

Internal structure of different Operating Systems can vary widely

Start by defining goals and specifications

Affected by choice of hardware, type of system

*User* goals and *System* goals

- User goals – operating system should be convenient to use, easy to learn, reliable, safe, and fast
- System goals – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient

Important principle to separate

**Policy:** What will be done?

**Mechanism:** How to do it?

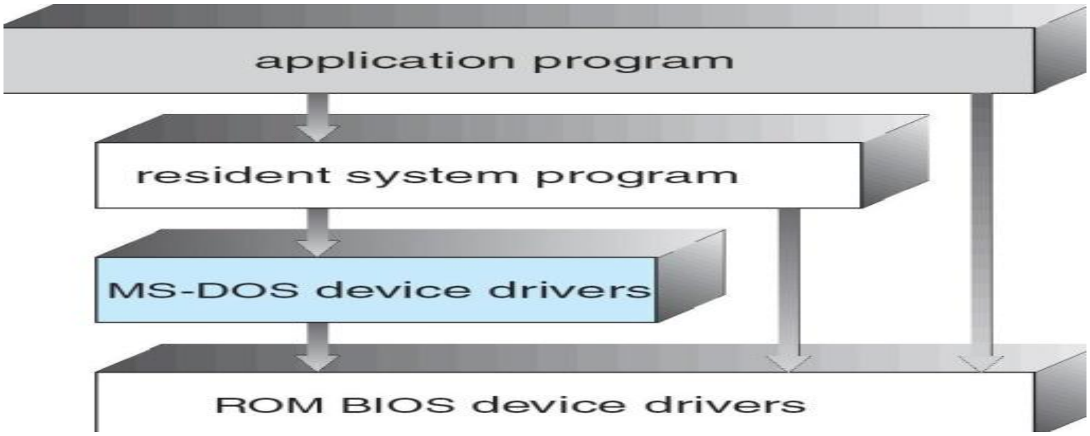
Mechanisms determine how to do something, policies decide what will be done The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later

**Simple Structure** MS-DOS – written to provide the most functionality in the least space

Not divided into modules

Although MS-DOS has some structure, its interfaces and levels of Functionality are not well separated

## MS-DOS Layer Structure

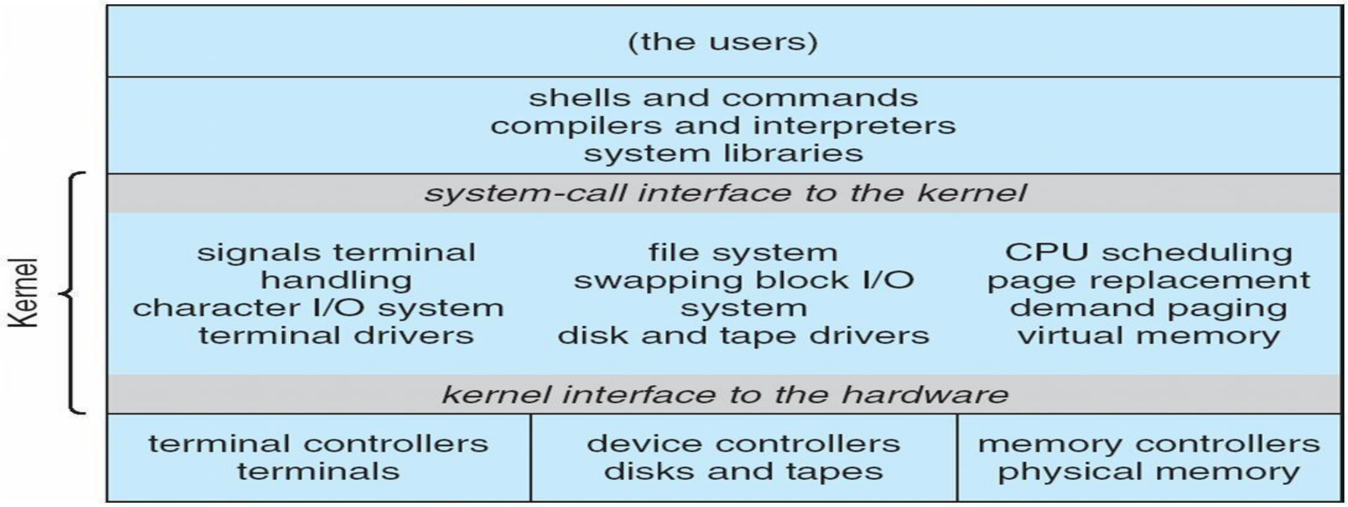


## Layered Approach

The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.

With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers

## Traditional UNIX System Structure



## UNIX

- UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts

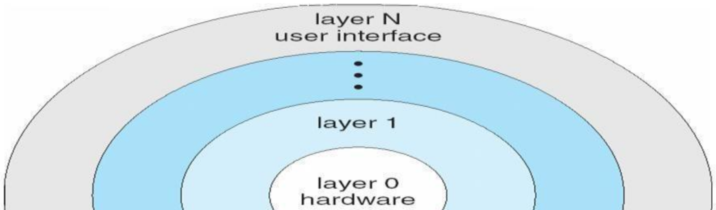
Systems programs

The kernel

Consists of everything below the system-call interface and above the physical hardware

Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level

## Layered Operating System



## Micro kernel System Structure

- Moves as much from the kernel into —*user*‖ space

Communication takes place between user modules using message passing

Benefits: Easier to extend a microkernel

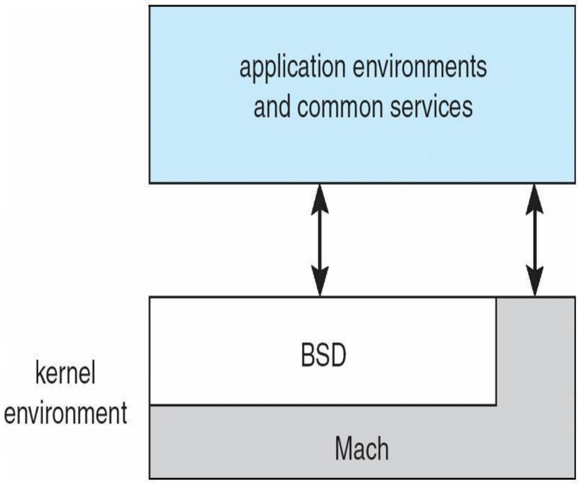
Easier to port the operating system to new architectures

More reliable (less code is running in kernel mode)

More secure

Detriments: Performance overhead of user space to kernel space communication

## Mac OS X Structure



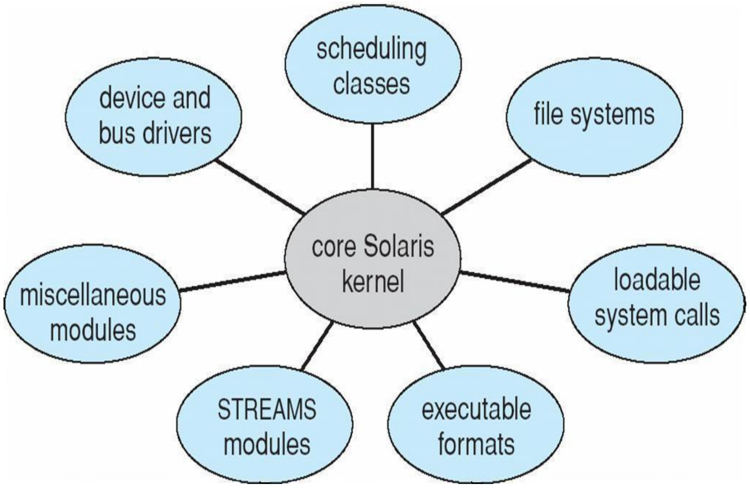
## Modules

Most modern operating systems implement kernel modules  Uses object-oriented approach

Each core component is separate

Each talks to the others over known interfaces
Each is loadable as needed within the kernel
Overall, similar to layers but with more flexible

## Solaris Modular Approach



### Virtual Machines

A virtual machine takes the layered approach to its logical conclusion. It treats hardware and the operating system kernel as though they were all hardware.

A virtual machine provides an interface *identical* to the underlying bare hardwarẹ

The operating system host creates the illusion that a process has its own processor and (virtual memory)̃Each guest provided with a (virtual) copy of underlying computeṛ̃

### Virtual Machines History and Benefits

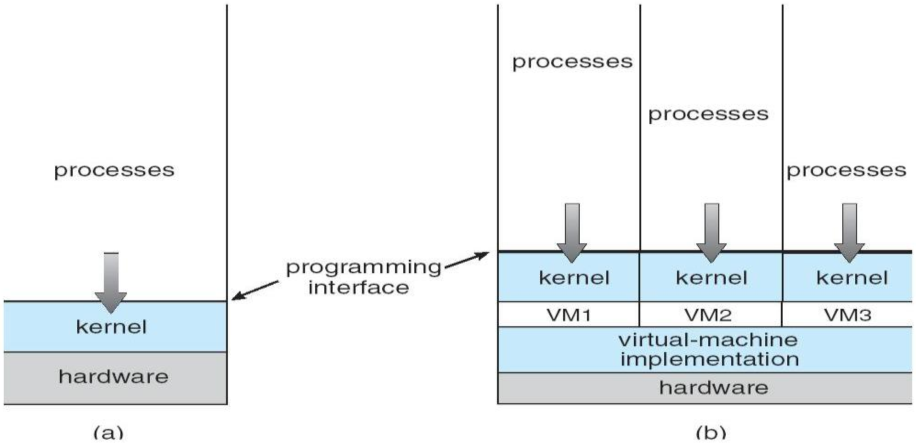
First appeared commercially in IBM mainframes in 1972.

Fundamentally, multiple execution environments (different operating systems) can share the same hardware.

Protect from each other.

Some sharing of file can be permitted, controlled commutate with each other, other physical systems via networking. Useful for development, testing consolidation of many low-resource use systems onto fewer busier systems

Open Virtual Machine Format||, standard format of virtual machines, allows a VM to run within many different virtual machine (host) platforms.

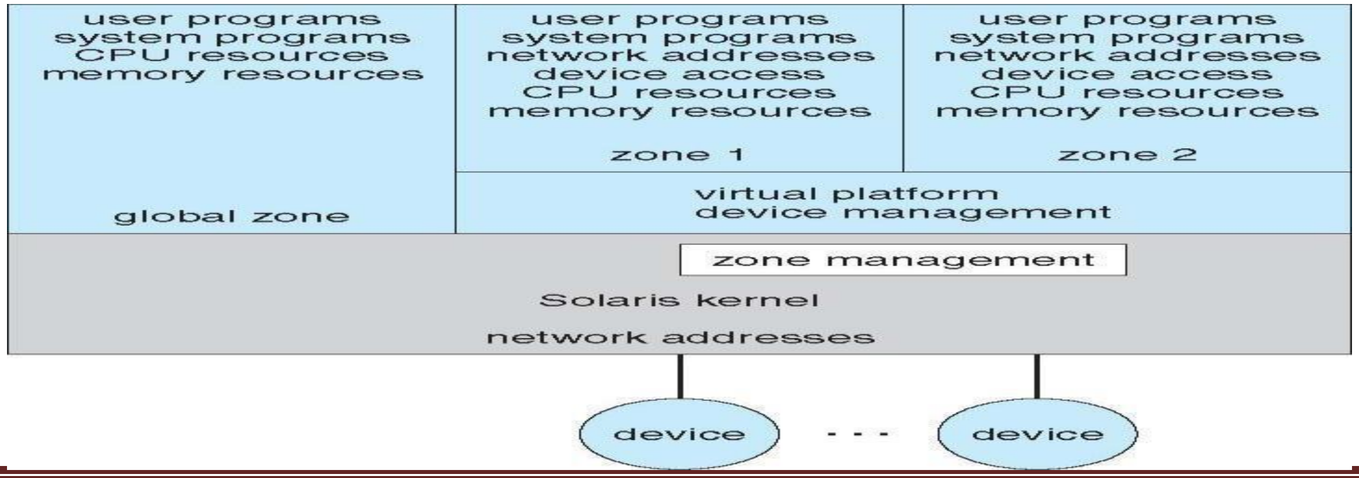


**Para-virtualization**

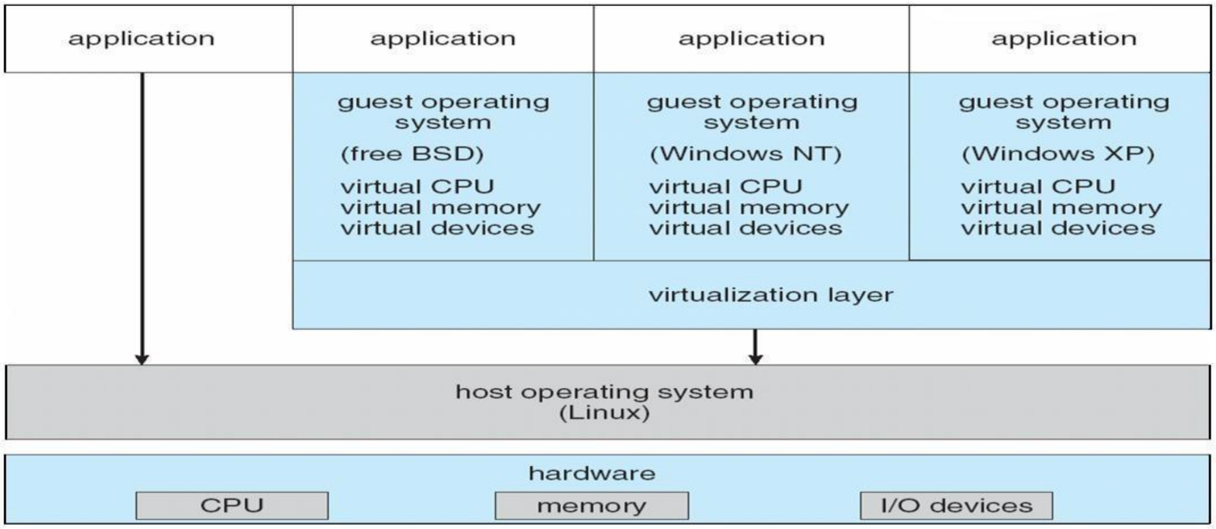
Presents guest with system similar but not identical to hardwarẹ Guest must be modified to run on paravirtualized hardwareḞ

Guest can be an OS, or in the case of Solaris 10 applications running in containerṣ

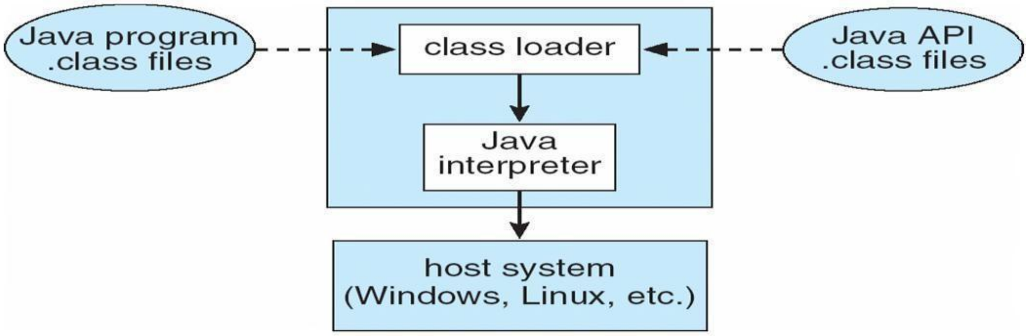
**Solaris 10 with Two Containers**



### VMware Architecture



### The Java Virtual Machine



### Operating-System Debugging

Debugging is finding and fixing errors, or bugs.

O Ses generate log files containing error information.

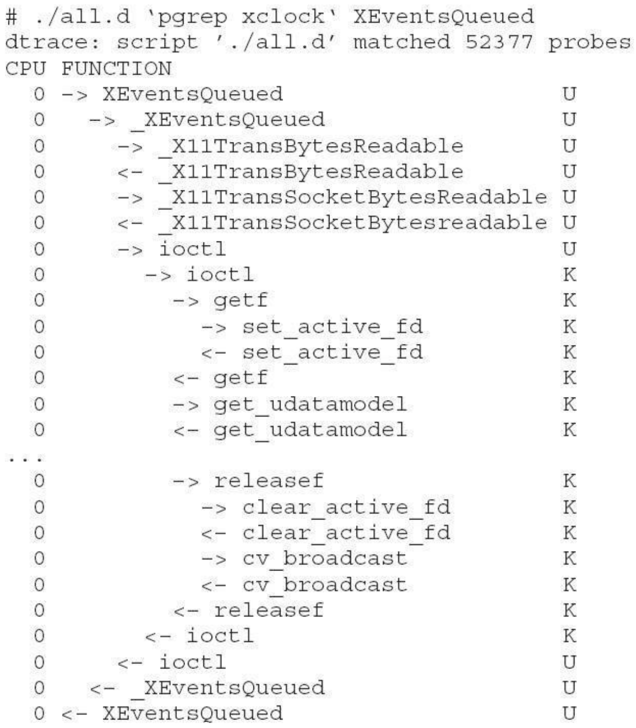
Failure of an application can generate core dump file capturing memory of the process Operating system failure can generate crash dump file containing kernel memory.

Beyond crashes, performance tuning can optimize system performance.

□ Kernighan’s Law: ―Debugging is twice as hard as writing the code in the rst place.

Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it. D Trace tool in Solaris, FreeBSD, Mac OS X allows live instrumentation on production systems. Probes fire when code is executed, capturing state data and sending it to consumers of those probes.

**Solaris 10 dtrace Following System Call**



```
# ./all.d 'pgrep xclock' XEventsQueued
dtrace: script './all.d' matched 52377 probes
CPU FUNCTION
  0 -> XEventsQueued                    U
  0   -> _XEventsQueued                 U
  0     -> _X11TransBytesReadable       U
  0     <- _X11TransBytesReadable       U
  0     -> _X11TransSocketBytesReadable U
  0     <- _X11TransSocketBytesreadable U
  0     -> ioctl                        U
  0       -> ioctl                      K
  0         -> getf                     K
  0           -> set_active_fd          K
  0           <- set_active_fd          K
  0         <- getf                     K
  0         -> get_udatamodel           K
  0         <- get_udatamodel           K
...
  0         -> releasef                 K
  0           -> clear_active_fd        K
  0           <- clear_active_fd        K
  0           -> cv_broadcast           K
  0           <- cv_broadcast           K
  0         <- releasef                 K
  0       <- ioctl                      K
  0     <- ioctl                        U
  0   <- _XEventsQueued                 U
  0 <- XEventsQueued                    U
```

**Operating System Generation**

Operating systems are designed to run on any of a class of machines; the system must be configured for each specific computer site.

SYSGEN program obtains information concerning the specific configuration of the hardware systeṃ

□ *Booting* – starting a computer by loading the kerneḷ

□ *Bootstrap program* – code stored in ROM that is able to locate the kernel, load it into memory, and start its executioṇ

**System Boot**

Operating system must be made available to hardware so hardware can start ịt

□ Small piece of code – **bootstrap loader**, locates the kernel, loads it into memory, and starts it. Sometimes two-step process where **boot block** at fixed location loads bootstrap loadeṛ When power initialized on system, execution starts at a fixed memory location. Firmware used to hold initial boot code

## Memory Management

To provide a detailed description of various ways of organizing memory hardware

To discuss various memory-management techniques, including paging and segmentation

To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging

Program must be brought (from disk) into memory and placed within a process for it to be run

Main memory and registers are only storage CPU can access directly Register access in one CPU clock (or less)

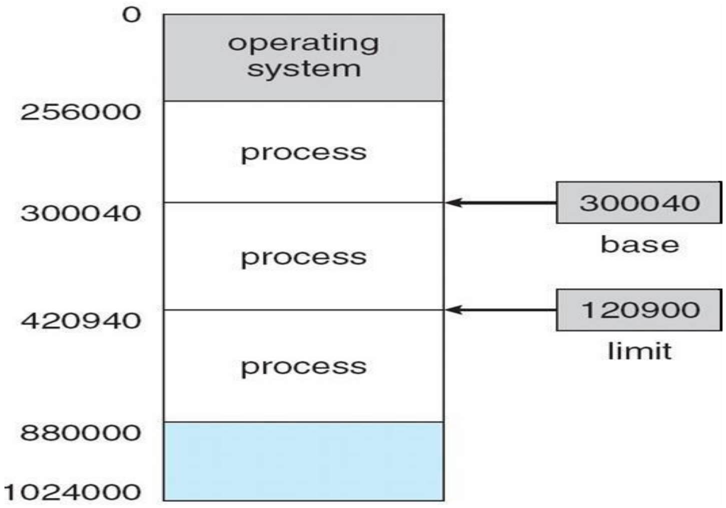
Main memory can take many cycles

**Cache** sits between main memory and CPU registers

Protection of memory required to ensure correct operation

**Base and Limit Registers**

A pair of **base** and **limit** registers define the logical address space



**Binding of Instructions and Data to Memory**

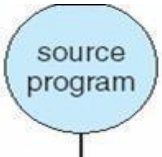
Address binding of instructions and data to memory addresses can happen at three different stages

**Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes

**Load time**: Must generate **relocatable code** if memory location is not known at compile time

**Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., base and limit registers)

**Multistep Processing of a User Program**



**Logical vs. Physical Address Space**

The concept of a logical address space that is bound to a separate. **physical address space** is central to proper memory management

- **Logical address** – generated by the CPU; also referred to as **virtual address.**
- **Physical address** – address seen by the memory unit.

Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme.

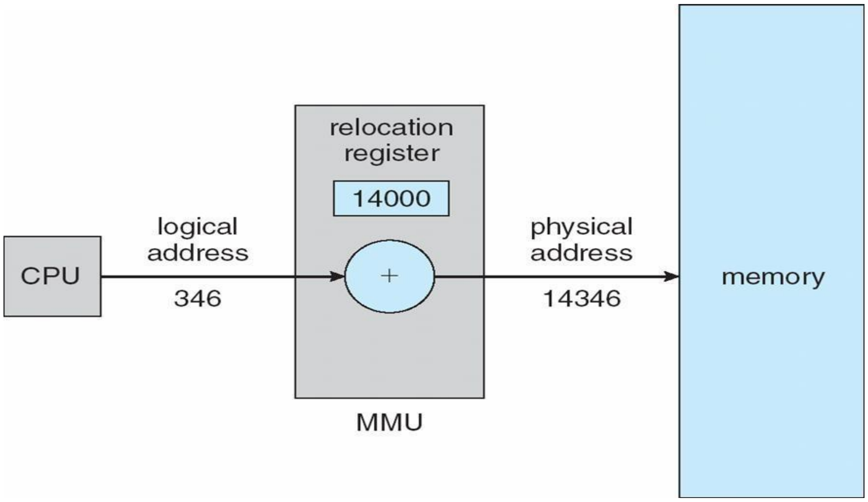
**Memory-Management Unit (MMU)**

Hardware device that maps virtual to physical address.

In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory The user program deals with *logical* addresses; it never sees the *reaḷ*

physical addresses

**Dynamic relocation using a relocation register**



**Dynamic Loading**

Routine is not loaded until it is calleḍ

Better memory-space utilization; unused routine is never loadeḍ

Useful when large amounts of code are needed to handle infrequently occurring caseṣ

No special support from the operating system is required implemented through program desigṇ

**Dynamic Linking**

□ Linking postponed until execution time

□ Small piece of code, *stub*, used to locate the appropriate memory-resident library routine

□ Stub replaces itself with the address of the routine, and executes the routine

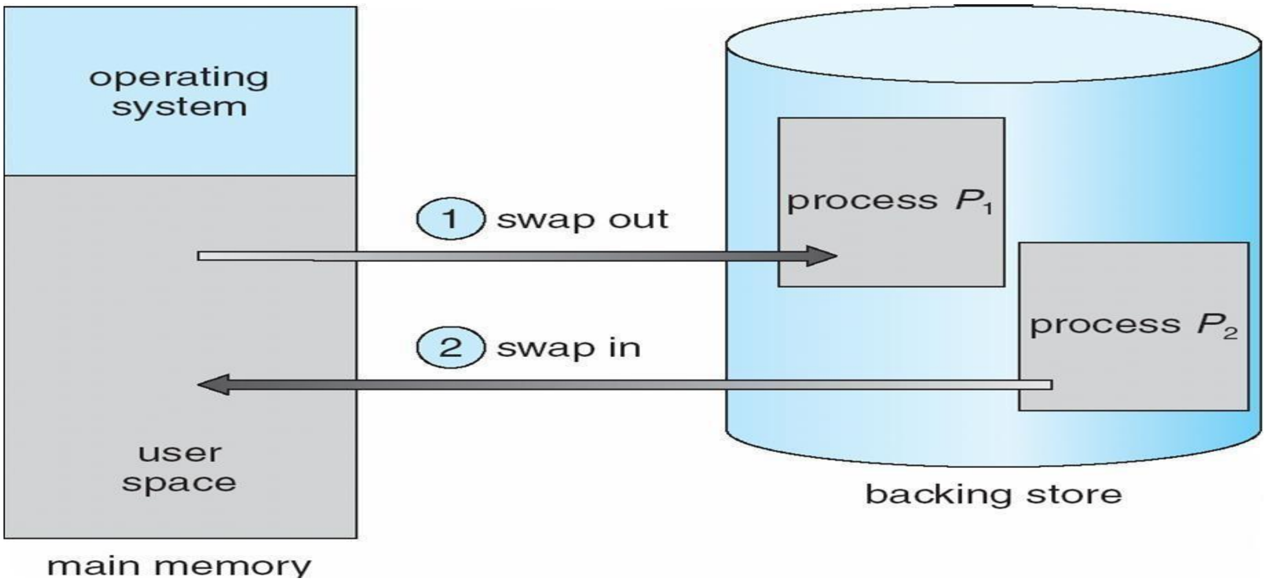
□ Operating system needed to check if routine is in processes' memory address
□ Dynamic linking is particularly useful for libraries

□ System also known as **shared libraries**

**Swapping**

A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued executionn**Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory imagesn**Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executednMajor part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swappednModified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows) System maintains a **ready queue** of ready-to-run processes which have memory images on disk

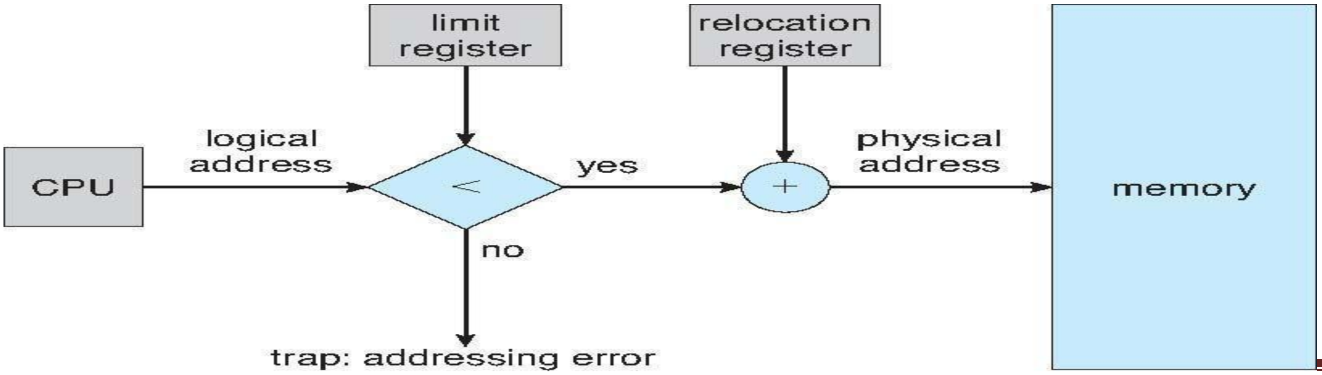
**Schematic View of Swapping**



**Contiguous Allocation**

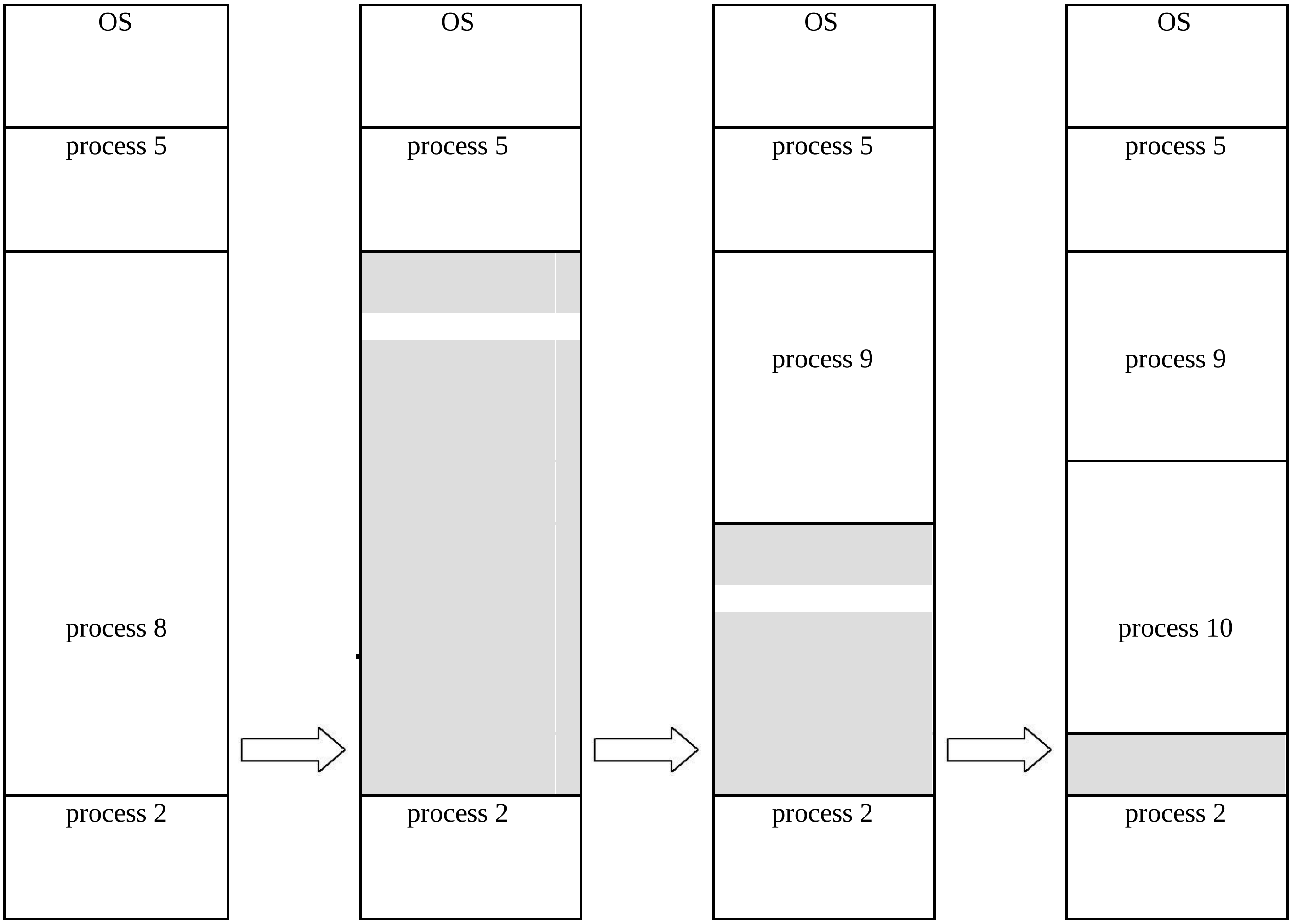
- □ Main memory usually into two partitions:
- □ Resident operating system, usually held in low memory with interrupt vector
- □ User processes then held in high memorynRelocation registers used to protect user processes from each other, and from changing operating-system code and data
- □ Base register contains value of smallest physical address
- □ Limit register contains range of logical addresses – each logical address must be less than the limit register
- □ MMU maps logical address *dynamically*

**Hardware Support for Relocation and Limit Registers**



- Multiple-partition allocation
- Hole – block of available memory; holes of various size are scattered throughout memory
- When a process arrives, it is allocated memory from a hole large enough to accommodate it
- Operating system maintains information about:

a) allocated partitions b) free partitions (hole)



**Dynamic Storage-Allocation Problem**

**First-fit**: Allocate the *first* hole that is big enougḥ

**Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size Produces the smallest leftover holẹ

**Worst-fit**: Allocate the *largest* hole; must also search entire lisṭ Produces the largest leftover holẹ

First-fit and best-fit better than worst-fit in terms of speed and storage utilization

**Fragmentation**

□ **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguouṣ

□ **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being useḍ

Reduce external fragmentation by **compactioṇ**

Shuffle memory contents to place all free memory together in one large bloc ḳ

Compaction is possible *only* if relocation is dynamic, and is done at execution timẹ.

I/O problem Latch job in memory while it is involved in I/O Do I/O only into OS buffers

**Paging**

Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is availablẹ

Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8,192 byteṣ)

Divide logical memory into blocks of same size called **pages**nKeep track of all free frameṣ

To run a program of size ***n*** pages, need to find *n* free frames and load prograṃ

Set up a page table to translate logical to physical addresseṣ Internal fragmentatioṇ

**Address Translation Scheme**

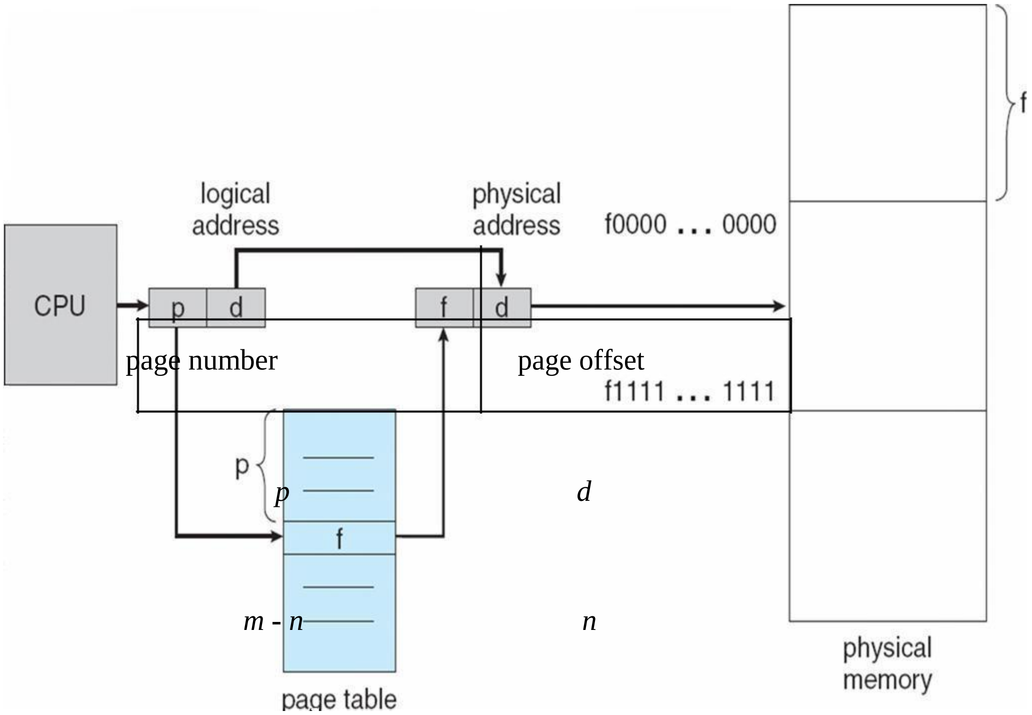
Address generated by CPU is divided into

□ **Page number (*p*)** – used as an index into a *page table* which contains base address of each page in physical memory

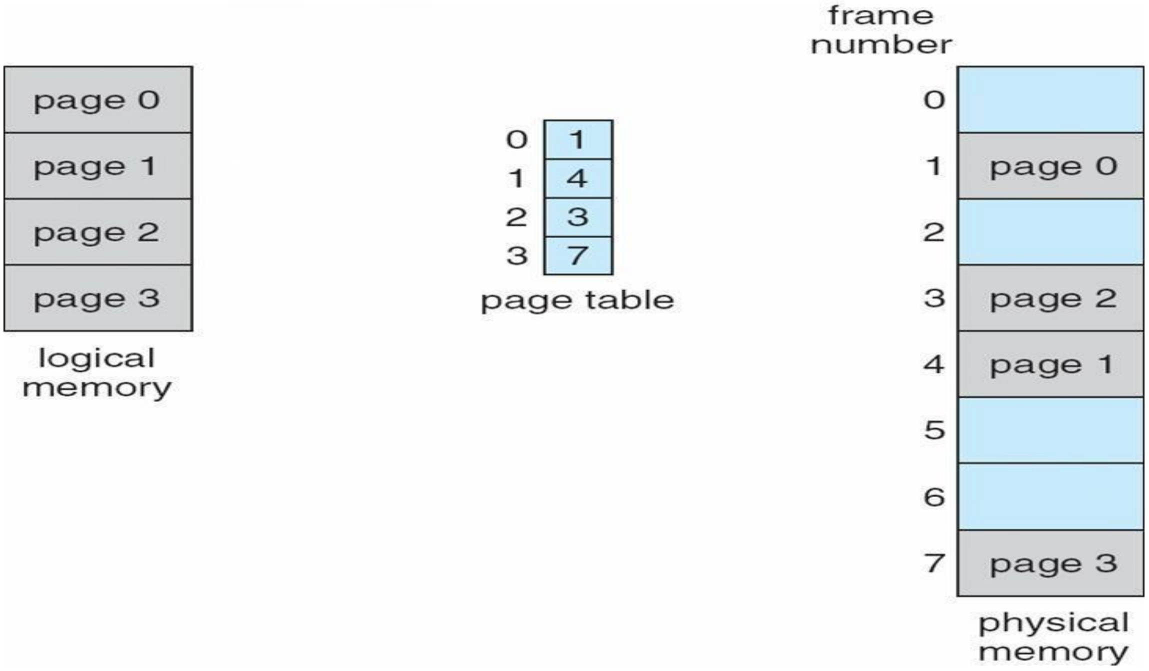
□ **Page offset (d)** – combined with base address to define the physical memory address that is sent to the memory unit

For given logical address space 2*m and page size 2*n

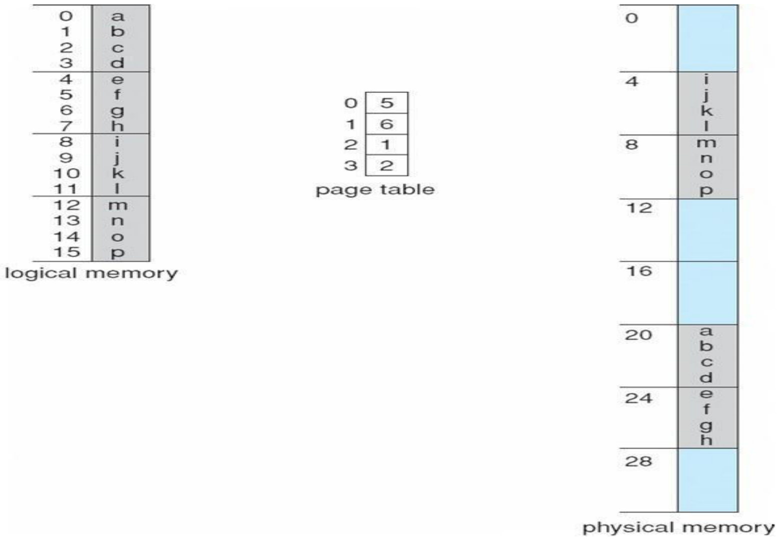
**Paging Hardware**



**Paging Model of Logical and Physical Memory**

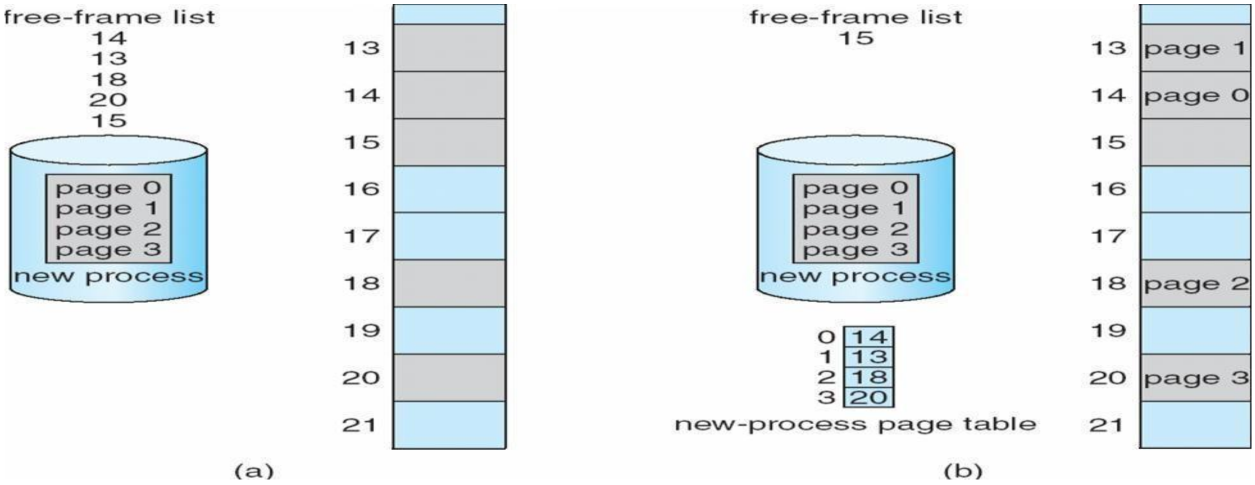


**Paging Example**



32-byte memory and 4-byte pages

**Free Frames**



## Implementation of Page Table

Page table is kept in main memory

**Page-table base register (PTBR)** points to the page table

**Page-table length register (PRLR)** indicates size of the page table

In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.

The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**

□ Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process

## Associative Memory

□ Associative memory – parallel search

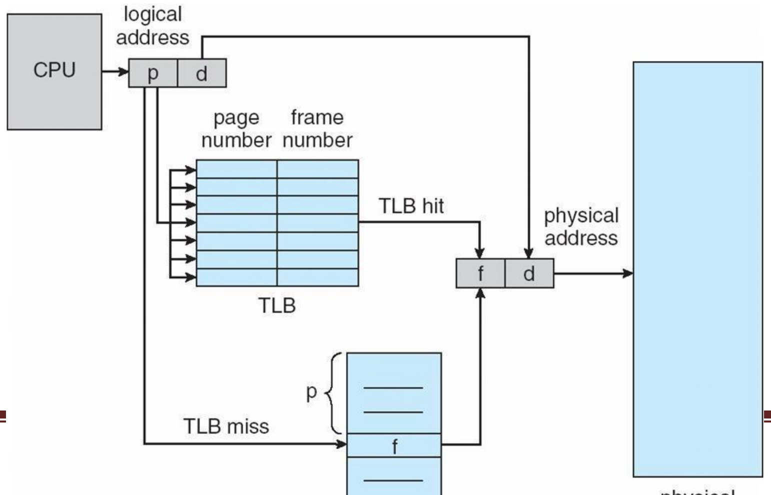
Address translation (p, d)

If p is in associative register, get frame # out

Otherwise get frame # from page table in memory

| | |
|---|---|
| | |
| Page # | |
| | Frame # |
| | |

## Paging Hardware With TLB



**Effective Access Time**

Associative Lookup = e time unit

Assume memory cycle time is 1 microsecond

 Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers

Hit ratio = an **Effective Access Time** (EAT)

$$EAT = (1 + e) a + (2 + e)(1 – a)$$

$$= 2 + e – a$$

**Memory Protection**

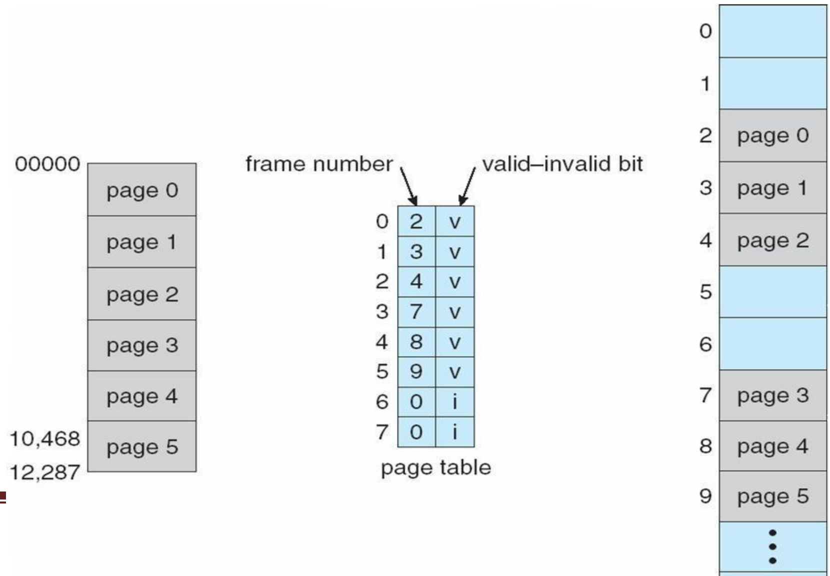
Memory protection implemented by associating protection bit with each frame

**Valid-invalid** bit attached to each entry in the page table:

 —valid|| indicates that the associated page is in the process' logical address space, and is thus a legal page

 —invalid|| indicates that the page is not in the process' logical address space

**Valid (v) or Invalid (i) Bit In A Page Table**



**Shared Pages**

**Shared code**

One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).

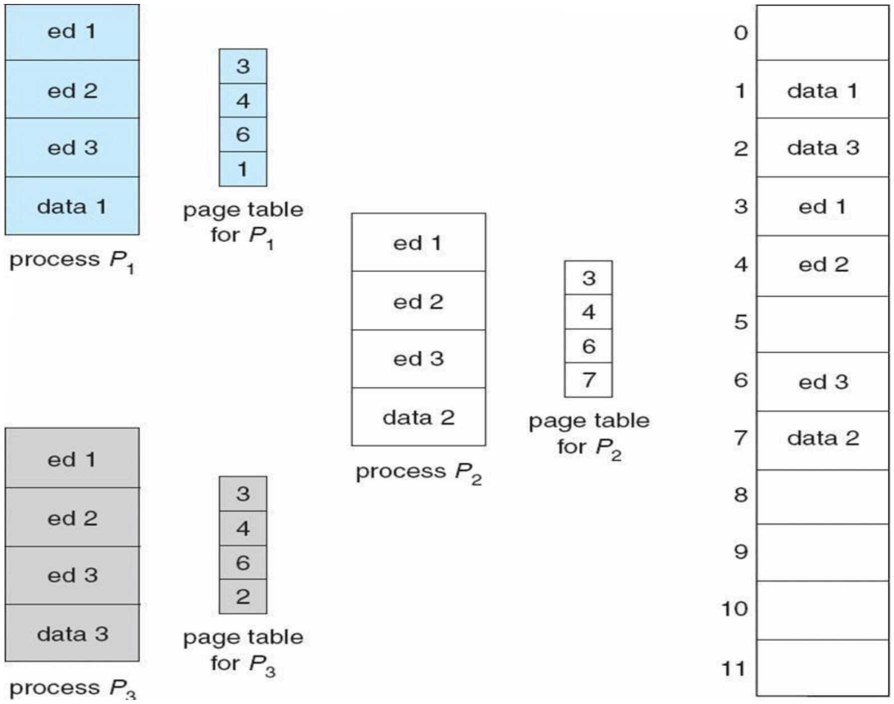
Shared code must appear in same location in the logical address space of all processes

**Private code and data**

Each process keeps a separate copy of the code and data

The pages for the private code and data can appear anywhere in the logical address space

**Shared Pages Example**



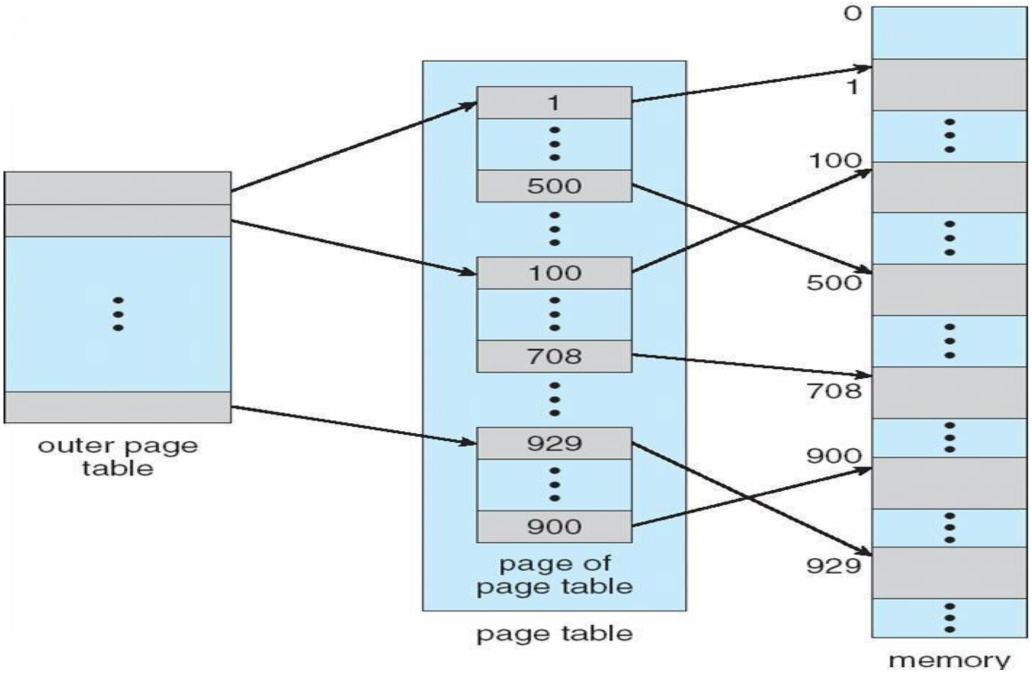
**Structure of the Page Table**

Hierarchical Paging Hashed
Page Tables Inverted Page
Tables

### Hierarchical Page Tables

Break up the logical address space into multiple page tableṣ A simple technique is a two-level page table

### Two-Level Page-Table Scheme



### Two-Level Paging Example

A logical address (on 32-bit machine with 1K page size) is divided intọ: a page number consisting of 22 bitṣ

a page offset consisting of 10 bits

Since the page table is paged, the page number is further divided intọ: a 12-bit page number

a 10-bit page offset

Thus, a logical address is as follows

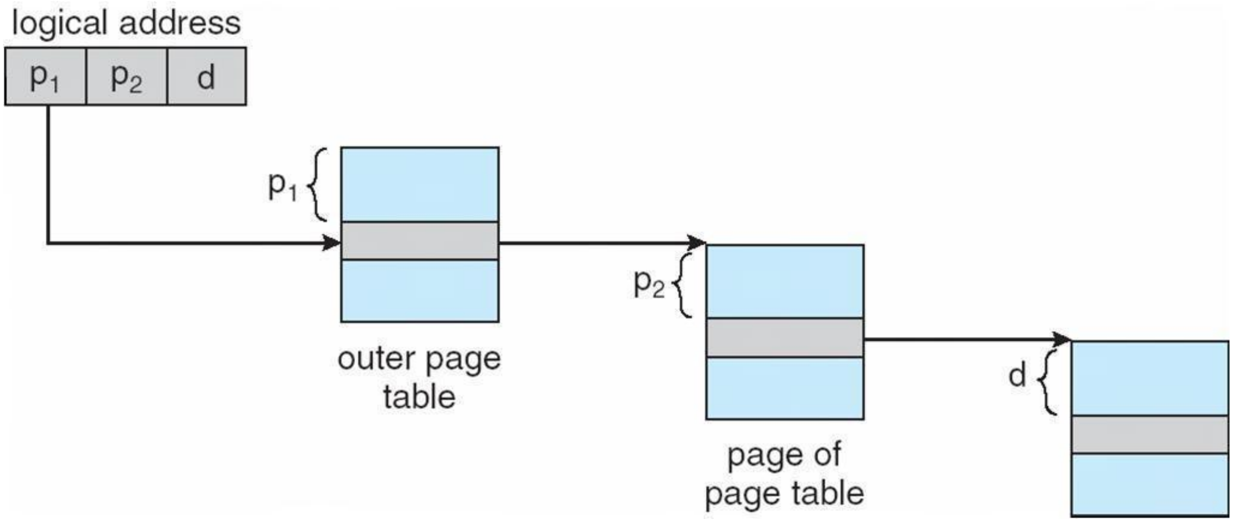
where *pi* is an index into the outer page table, and *p2* is the displacement within the page of the outer page table p

age number page offset

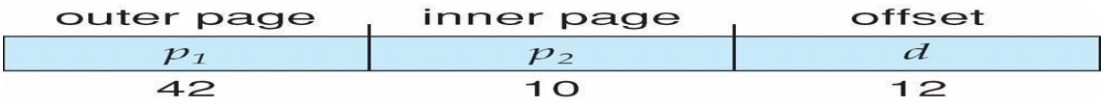
*p*i *p*2 *d*

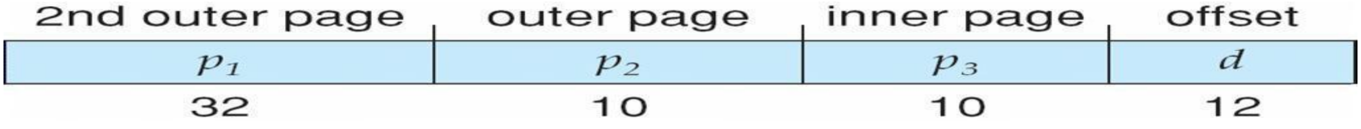
12 10 10

**Address-Translation Scheme**



**Three-level Paging Scheme**





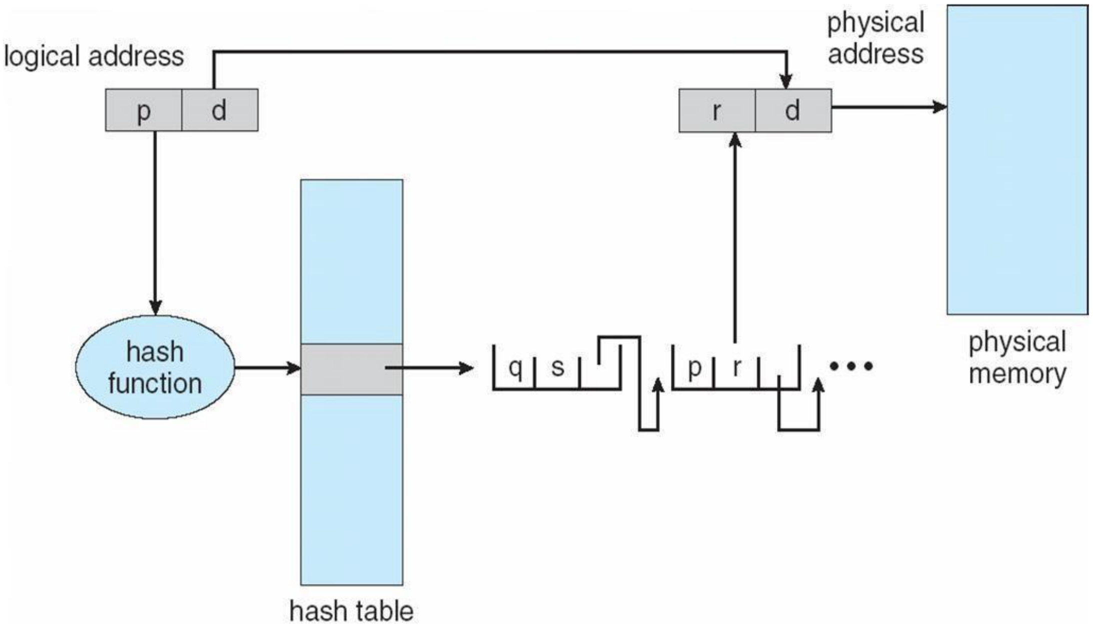
**Hashed Page Tables**

Common in address spaces > 32 bitṣ

The virtual page number is hashed into a page tablẹ

This page table contains a chain of elements hashing to the same locatioṇ Virtual page numbers are compared in this chain searching for a matcḥ If a match is found, the corresponding physical frame is extracted

**Hashed Page Table**



**Inverted Page Table**

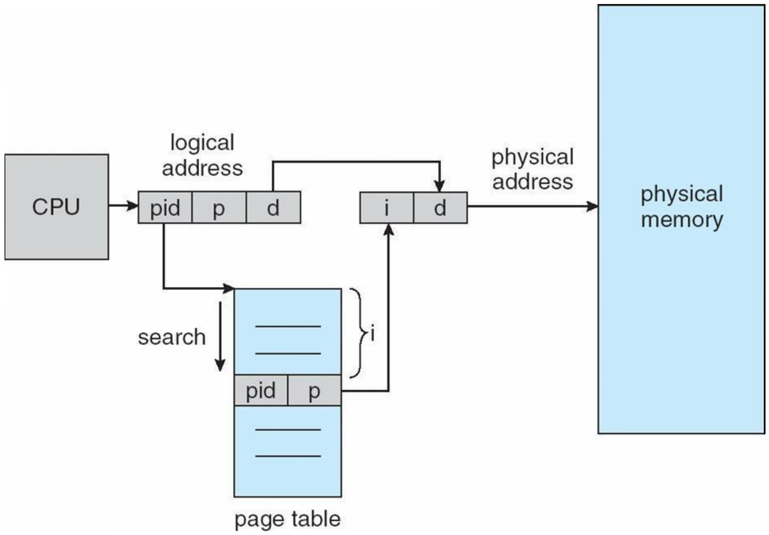
One entry for each real page of memory

Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page

Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs

□ Use hash table to limit the search to one — or at most a few — page-table entries

**Inverted Page Table Architecture**



**Segmentation**

Memory-management scheme that supports user view of memory

A program is a collection of
segmentṣ A segment is a logical unit
such aṣ: main prograṃ

procedure
functioṇ
methoḍ

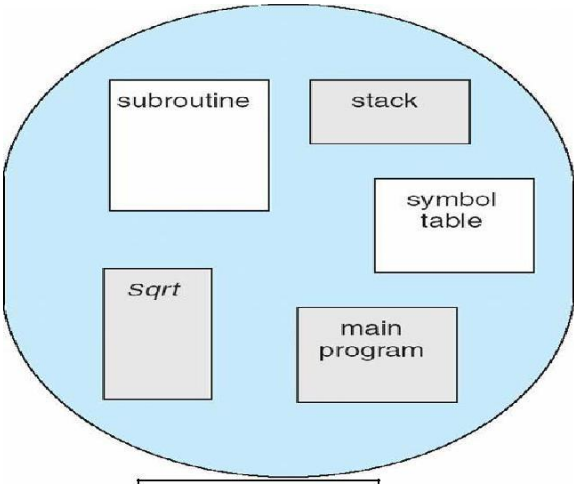
objecṭ

local variables, global
variableṣ common blocḳ

stacḳ

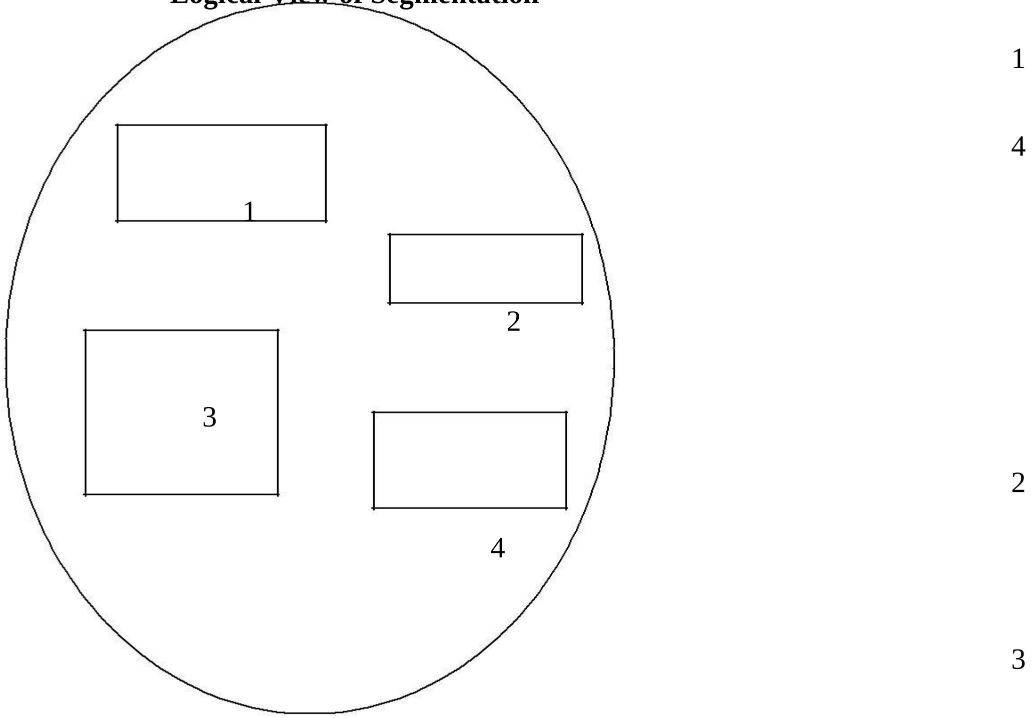
symbol tablẹ

arrayṣ



**User’s View of a Program**

**Logical View of Segmentation**



user space

**Segmentation Architecture**

Logical address consists of a two tuple:

<segment-number, offset>,

□ **Segment table** – maps two-dimensional physical addresses; each table entry has:

□ **base** – contains the starting physical address where the segments reside in memory

□ **limit** – specifies the length of the segment

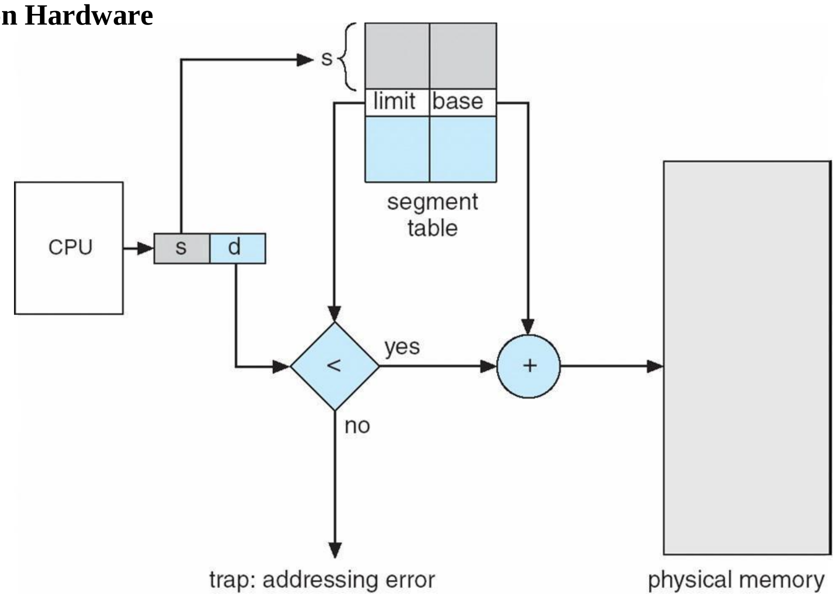
□ **Segment-table base register (STBR)** points to the segment table's location in memory

**Segment-table length register (STLR)** indicates number of segments used by a program;

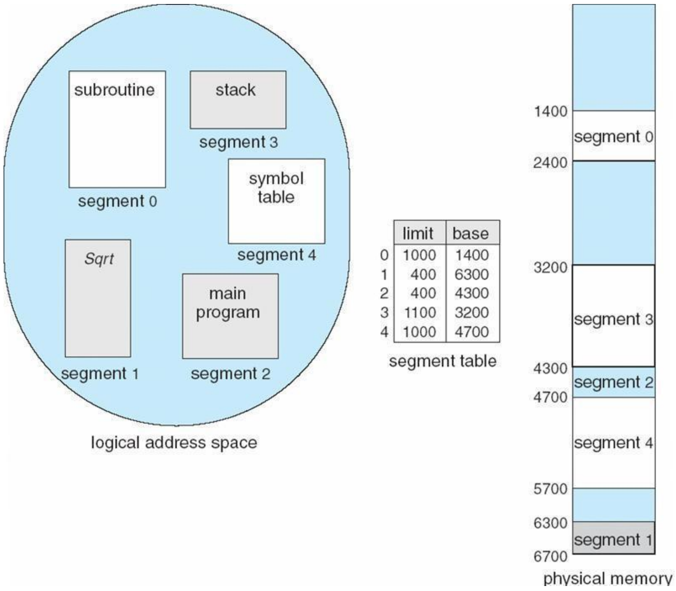
segment number ***s*** is legal if ***s*** < **STLR**

Protection With each entry in segment table associate: □ validation bit = 0 Þ illegal segment read/write/execute privileges Protection bits associated with segments; code sharing occurs at segment level Since segments vary in length, memory allocation is a dynamic storage-allocation problem A segmentation example is shown in the following diagram

### Segmentation Hardware



### Example of Segmentation



|   | limit | base |
|---|---|---|
| 0 | 1000 | 1400 |
| 1 | 400 | 6300 |
| 2 | 400 | 4300 |
| 3 | 1100 | 3200 |
| 4 | 1000 | 4700 |

segment table

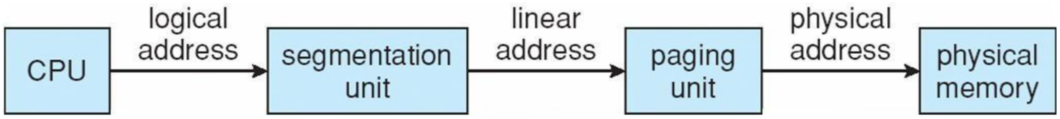
### Example: The Intel Pentium

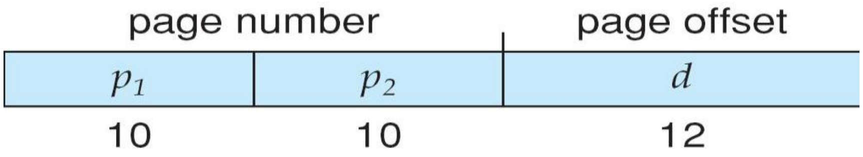
Supports both segmentation and segmentation with paging CPU generates logical address

Given to segmentation unit

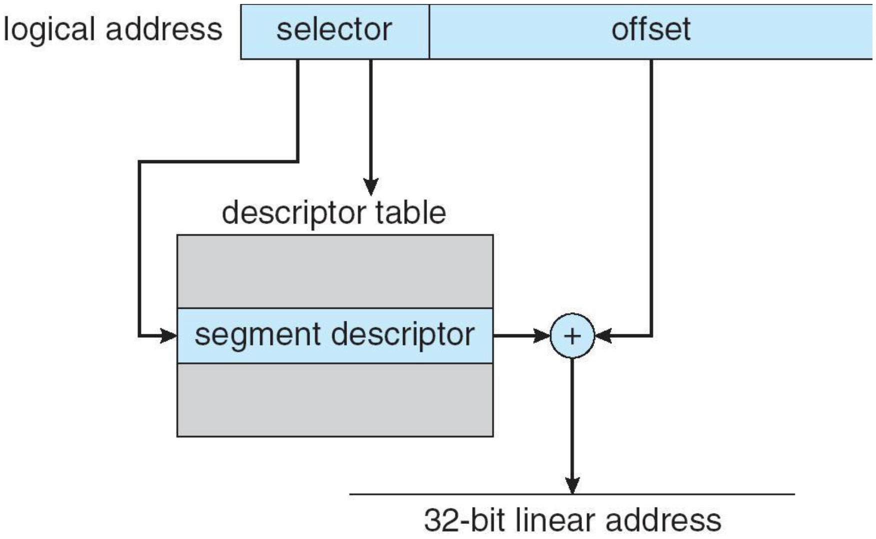
Which produces linear addresses Linear address given to paging unit Which generates physical address in main memory Paging units form equivalent of MMU

**Logical to Physical Address Translation in Pentium**

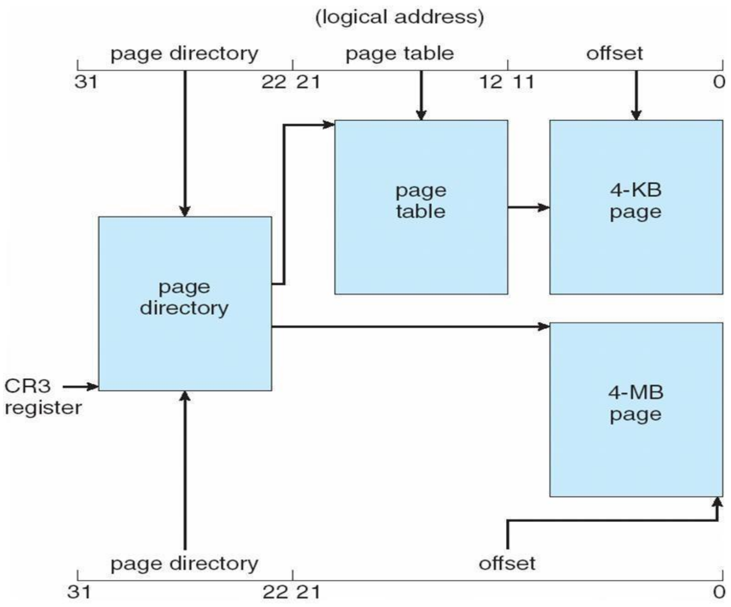




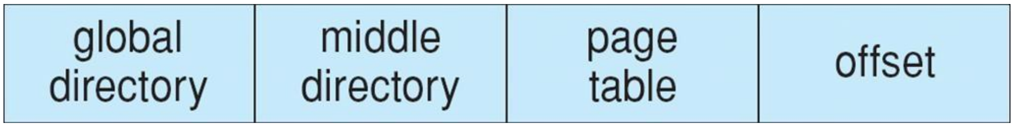
**Intel Pentium Segmentation**



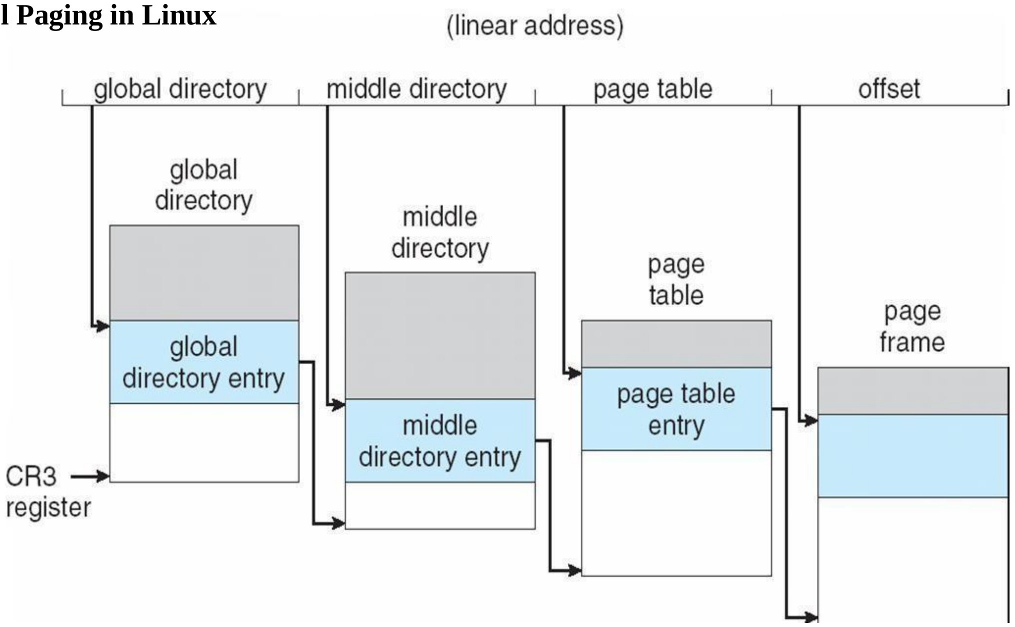
**Pentium Paging Architecture**



**Linear Address in Linux**



**Three-level Paging in Linux**



## PRINCIPLES OF DEADLOCK

Introduction

System Model

Deadlock Characterization Necessary

Deadlock Handling

Deadlock Prevention.

Deadlock Avoidance

Safe State

Resource-Allocation Graph Algorithm

Banker‘s Algorithm
Safety Algorithm

Resource Request Algorithm

Deadlock Detection

Single Instance of a Resource

Multiple Instances of a Resource

Recovery from Deadlock

Summary

Terminal Questions

Answers

**Introduction**

Several processes compete for a finite set of resources in a multi-programmed environment. A process requests for resources that may not be readily available at the time of the request. In such a case the process goes into a wait state. It may so happen that this process may never change state because the requested resources are held by other processes which themselves are waiting for additional resources and hence in a wait state. This situation is called a deadlock. **Deadlock** occurs when we have a set of processes [not necessarily all the processes in the system], each holding some resources, each requesting some resources, and none of them is able to obtain what it needs, i.e. to make progress. We will usually reason in terms of resources R1, R2, .., Rm and processes P1, P2, .., Pn. A process Pi that is waiting for some currently unavailable resource is said to be **blocked**.

Resources can be **preemptable** or **non-preemptable**. A resource is preemptable if it can be taken away from the process that is holding it [we can think that the original holder waits, frozen, until the resource is returned to it]. Memory is an example of a preemptable resource. Of course, one may choose to deal with intrinsically preemptable resources as if they were non-preemptable. In our discussion we only consider non-preemptable resources. Resources can be **reusable** or **consumable**. They are reusable if they can be used again after a process is done using them. Memory, printers, tape drives are examples of reusable resources. Consumable resources are resources that can be used only once. For example a message or a signal or an event. If two processes are waiting for a message and one receives it, then the other process remains waiting. To reason about deadlocks when dealing with consumable resources is extremely difficult. Thus we will restrict our discussion to reusable resources.

**Objectives:**

At the end of this unit, you will be able to understand:

System model for Deadlocks, Deadlock Characterization, Deadlock Handling, Deadlock prevention, how to avoid Deadlocks using Banker‘s Algorithm and Deadlock Detection.

**System Model**

The number of resources in a system is always finite. But the number of competing processes are many. Resources are of several types, each type having identical instances of the resource. Examples for resources could be memory space, CPU time, files, I/O devices and so on. If a system has 2 CPUs that are equivalent, then the resource type CPU time has 2 instances. If they are not equivalent, then each CPU is of a different resource type. Similarly the system may have 2 dot matrix printers and 1 line printer. Here the resource type of dot matrix printer has 2 instances whereas there is a single instance of type line printer. A process requests for resources, uses them if granted and then releases the resources for others to use. It goes without saying that the number of resources requested shall not exceed the total of each type available in the system. If a request for a resource cannot be granted immediately then the process requesting the resource goes into a wait state and joins the wait queue for the resource. A set of processes is in a state of deadlock if every process in the set is in some wait queue of a resource and is waiting for an event (release resource) to occur that can be caused by another process in the set. For example, there are 2 resources, 1 printer and 1 tape drive. Process P1 is allocated tape drive and P2 is allocated printer. Now if P1 requests for printer and P2 for tape drive, a deadlock occurs.

**Deadlock Characterization**

**Necessary Conditions for Deadlock**

A deadlock occurs in a system if the following four conditions hold simultaneously:

□ Mutual exclusion: At least one of the resources is non-sharable, that is, only one process at a time can use the resource.

□ Hold and wait: A process exists that is holding on to at least one resource and waiting for an additional resource held by another process.

□ No preemption: Resources cannot be preempted, that is, a resource is released only by the process that is holding it.

□ Circular wait: There exist a set of processes $P_0$, $P_1$, ….., $P_n$ of waiting processes such that $P_0$ is waiting for a resource held by $P_1$, $P_1$ is waiting for a resource held by $P_2$, ….., $P_{n-1}$ is waiting for a resource held $P_n$ and $P_n$ is in turn waiting for a resource held by $P_0$.

**Resource-Allocation Graph**

Deadlocks can be described by a resource allocation graph. The resource allocation graph is a directed graph consisting of vertices and directed edges. The vertex set is partitioned into two types, a subset representing processes and another subset representing resources. Pictorially, the resources are represented by rectangles with dots within, each dot representing an instance of the resource and circles represent processes. A directed edge from a process to a resource ($P_i$ $R_j$) signifies a request from a process $P_i$ for an instance of the resource $R_j$ and $P_i$ is waiting for $R_j$. A directed edge from a resource to a process ($R_j$ $P_i$) indicates that an instance of the resource $R_j$ has been allotted to process $P_i$. Thus a resource allocation graph consists of vertices which include resources and processes and directed edges which consist of request edges and assignment edges. A request edge is introduced into the graph when a process requests for a resource. This edge is converted into an assignment edge when the resource is granted. When the process releases the resource, the assignment edge is deleted.

Consider the following system:

There are 3 processes $P_1$, $P_2$ and $P_3$.

Resources $R_1$, $R_2$, $R_3$ and $R_4$ have instances 1, 2, 1, and 3 respectively. $P_1$ is holding $R_2$ and waiting for $R_1$.

$P_2$ is holding $R_1$, $R_2$ and is waiting for $R_3$.

$P_3$ is holding $R_3$.

The resource allocation gragh for a system in the above situation is as shown below (Figure 5.1).

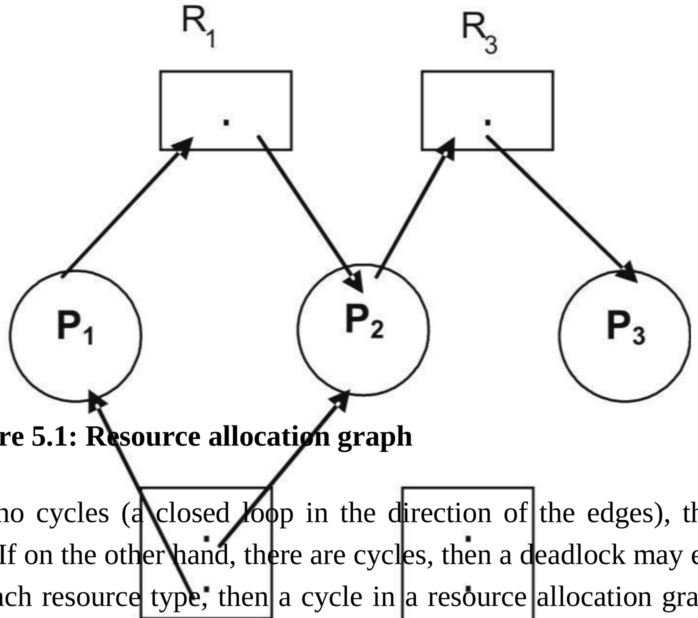


**Figure 5.1: Resource allocation graph**

If a resource allocation graph has no cycles (a closed loop in the direction of the edges), then the system is not in a state of deadlock. If on the other hand, there are cycles, then a deadlock may exist. If there are only single instances of each resource type, then a cycle in a resource allocation graph is a necessary and sufficient condition for existence of a deadlock (Figure 5.2). Here two cycles exist: Processes $P_0$, $P_1$ and $P_3$ are deadlocked and are in a circular wait. $P_2$ is waiting for $R_3$ held by $P_3$. $P_3$ is waiting for $P_1$ or $P_2$ to release $R_2$. So also $P_1$ is waiting for $P_2$ to releaseR3.

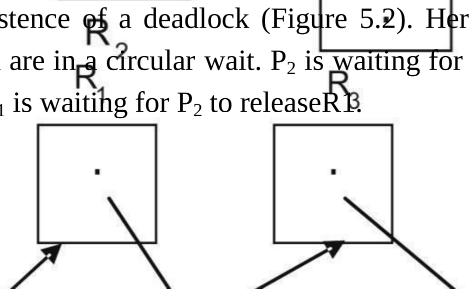


**Figure 5.2: Resource allocation graph with deadlock**

If there are multiple instances of resources types, then a cycle does not necessarily imply a deadlock. Here a cycle is a necessary condition but not a sufficient condition for the existence of a deadlock (Figure 5.3). Here also there is a cycle:

The cycle above does not imply a deadlock because an instance of $R_1$ released by $P_2$ could be assigned to $P_1$ or an instance of $R_2$ released by $P_4$ could be assigned to $P_3$ there by breaking the cycle.

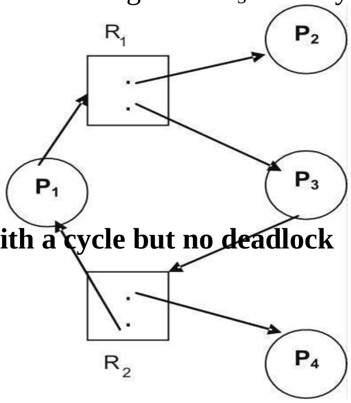


**Figure 5.3: Resource allocation graph with a cycle but no deadlock**

**Deadlock Handling**

Different methods to deal with deadlocks include methods to ensure that the system will never enter into a state of deadlock, methods that allow the system to enter into a deadlock and then recover or to just ignore the problem of deadlocks .To ensure that deadlocks never occur, deadlock prevention / avoidance schemes are used. The four necessary conditions for deadlocks to occur are mutual exclusion, hod and wait, no preemption and circular wait. Deadlock prevention ensures that at least one of the four necessary conditions for deadlocks do not hold. To do this the scheme enforces constraints on requests for resources. Dead-lock avoidance scheme requires the operating system to know in advance, the resources needed by a process for its entire lifetime. Based on this a priori information, the process making a request is either made to wait or not to wait in case the requested resource is not readily available. If none of the above two schemes are used, then deadlocks may occur. In such a case, an algorithm to recover from the state of deadlock is used. If the problem of deadlocks is ignored totally, that is, to say the system does not ensure that a deadlock does not occur and also does not provide for recovery from deadlock and such a situation arises, then there is no way out of the deadlock. Eventually, the system may crash because more and more processes request for resources and enter into deadlock.

**Deadlock Prevention**

The four necessary conditions for deadlocks to occur are mutual exclusion, hold and wait, no preemption and circular wait. If any one of the above four conditions does not hold, then deadlocks will not occur. Thus prevention of deadlock is possible by ensuring that at least one of the four conditions cannot hold. Mutual exclusion: Resources that can be shared are never involved in a deadlock because such resources can always be granted simultaneous access by processes. Hence processes requesting for such a sharable resource will never have to wait. Examples of such resources include read-only files. Mutual exclusion must therefore hold for non-sharable resources. But it is not always possible to prevent deadlocks by denying mutual exclusion condition because some resources are by nature non-sharable, for example printers.

**Hold and wait**: To avoid hold and wait, the system must ensure that a process that requests for a resource does not hold on to another. There can be two approaches to this scheme:

1. a process requests for and gets allocated all the resources it uses before execution begins.
2. a process can request for a resource only when it does not hold on to any other.

Algorithms based on these approaches have poor resource utilization. This is because resources get locked with processes much earlier than they are actually used and hence not available for others to use as in the first approach. The second approach seems applicable only when there is assurance about reusability of data and code on the released resources. The algorithms also suffer from starvation since popular resources may never be freely available.

**No preemption**: This condition states that resources allocated to processes cannot be preempted. To ensure that this condition does not hold, resources could be preempted. When a process requests for a resource, it is allocated the resource if it is available. If it is not, than a check is made to see if the process holding the wanted resource is also waiting for additional resources. If so the wanted resource is preempted from the waiting process and allotted to the requesting process. If both the above is not true, that is, the resource is neither available nor held by a waiting process, then the requesting process waits. During its waiting period, some of its resources could also be preempted in which case the process will be restarted only when all the new and the preempted resources are allocated to it. Another alternative approach could be as follows: If a process requests for a resource which is not available immediately, then all other resources it currently holds are preempted. The process restarts only when the new and the preempted resources are allocated to it as in the previous case. Resources can be preempted only if their current status can be saved so that processes could be restarted later by restoring the previous states. Example CPU memory and main memory. But resources such as printers cannot be preempted, as their states cannot be saved for restoration later. Circular wait: Resource types need to be ordered and processes requesting for resources will do so in an increasing order of enumeration. Each resource type is mapped to a unique integer that allows resources to compare and to find out the precedence order for the resources. Thus F: R-> N is a 1:1 function that maps resources to numbers. For example:

F (tape drive) = 1, F (disk drive) = 5, F (printer) = 10.

To ensure that deadlocks do not occur, each process can request for resources only in an increasing order of these numbers. A process, to start within the very first, instance can request for any resource say $R_i$. Thereafter it can request for a resource $R_j$ if and only if $F(R_j)$ is greater than $F(R_i)$. Alternately, if $F(R_j)$ is less than $F(R_i)$, then $R_j$ can be allocated to the process if and only if the process releases $R_i$. The mapping function F should be so defined that resources get numbers in the usual order of usage.

**Deadlock Avoidance**

Deadlock prevention algorithms ensure that at least one of the four necessary conditions for deadlocks namely mutual exclusion, hold and wait, no preemption and circular wait do not hold. The disadvantage with prevention algorithms is poor resource utilization and thus reduced system throughput. An alternate method is to avoid deadlocks. In this case additional a priori information about the usage of resources by processes is required. This information helps to decide on whether a process should wait for a resource or not. Decision about a request is based on all the resources available, resources allocated to processes, future requests and releases by processes. A deadlock avoidance algorithm requires each process to make known in advance the maximum number of resources of each type that it may need. Also known is the maximum number of resources of each type available. Using both the above a priori knowledge, deadlock avoidance algorithm ensures that a circular wait condition never occurs.

**Safe State**

A system is said to be in a safe state if it can allocate resources upto the maximum available and is not in a state of deadlock. A safe sequence of processes always ensures a safe state. A sequence of processes $< P_1, P_2, \ldots, P_n >$ is safe for the current allocation of resources to processes if resource requests from each $P_i$ can be satisfied from the currently available resources and the resources held by all $P_j$ where $j < i$. If the state is safe then $P_i$ requesting for resources can wait till $P_j$'s have completed. If such a safe sequence does not exist, then the system is in an unsafe state.A safe state is not a deadlock state.

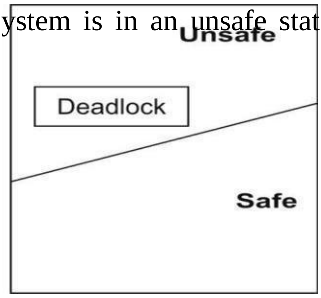


**Figure 5.4: Safe, unsafe and deadlock state spaces**

If a system is in a safe state it can stay away from an unsafe state and thus avoid deadlock. On the other hand, if a system is in an unsafe state, deadlocks cannot be avoided.

**Illustration:** A system has 12 instances of a resource type and 3 processes using these resources. The maximum requirements for the resource by the processes and their current allocation at an instance say t0 is as shown below:

| Process | Maximum | Current |
|---|---|---|
| P0 | 10 | 5 |

| | | |
|---|---|---|
| P1 | 4 | 2 |
| P3 | 9 | 2 |

At the instant $t_0$, the system is in a safe state and one safe sequence is < $P_1$, $P_0$, $P_2$ >.
This is true because of the following facts:
Out of 12 instances of the resource, 9 are currently allocated and 3 are free. $P_1$ needs only 2 more, its maximum being 4, can be allotted 2.
Now only 1 instance of the resource is free.
When $P_1$ terminates, 5 instances of the resource will be free. $P_0$ needs only 5 more, its maximum being 10, can be allotted 5. Now resource is not free.
Once $P_0$ terminates, 10 instances of the resource will be free. $P_3$ needs only 7 more, its maximum being 9, can be allotted 7. Now 3 instances of the resource are free. When $P_3$ terminates, all 12 instances of the resource will be free. Thus the sequence < $P_1$, $P_0$, $P_3$ > is a safe sequence and the system is in a safe state. Let us now consider the following scenario at an instant $t_1$. In addition to the allocation shown in the table above, $P_2$ requests for 1 more instance of the resource and the allocation is made. At the instance $t_1$, a safe sequence cannot be found as shown below:
Out of 12 instances of the resource, 10 are currently allocated and 2 are free.
$P_1$ needs only 2 more, its maximum being 4, can be allotted 2. Now resource is not free
Once $P_1$ terminates, 4 instances of the resource will be free. $P_0$ needs 5 more while $P_2$ needs 6 more.
Since both $P_0$ and $P_2$ cannot be granted resources, they wait. The result is a deadlock
Thus the system has gone from a safe state at time instant $t_0$ into an unsafe state at an instant $t_1$. The extra resource that was granted to $P_2$ at the instant $t_1$ was a mistake. $P_2$ should have waited till other processes finished and released their resources. Since resources available should not be allocated right away as the system may enter an unsafe state, resource utilization is low if deadlock avoidance algorithms are used.

**Resource Allocation Graph Algorithm**

A resource allocation graph could be used to avoid deadlocks. If a resource allocation graph does not have a cycle, then the system is not in deadlock. But if there is a cycle then the system may be in a deadlock. If the resource allocation graph shows only resources that have only a single instance, then a cycle does imply a deadlock. An algorithm for avoiding deadlocks where resources have single instances in a resource allocation graph is as described below.The resource allocation graph has request edges and assignment edges. Let there be another kind of edge called a claim edge. A directed edge $P_i$ $R_j$ indicates that $P_i$ may request for the resource $R_j$ some time later. In a resource allocation graph a li

dashed n
e

represents a claim edge. Later when a process makes an actual request for a resource, the corresponding claim edge is converted to a request edge $P_i$ $R_j$. Similarly when a process releases a resource after use, the assignment edge $R_j$ $P_i$ is reconverted to a claim edge $P_i$ $R_j$. Thus a process must be associated with all its claim edges before it starts executing.If a process Pi requests for a resource $R_j$, then the claim edge $P_i$ $R_j$ is first converted to a request edge $P_i$ $R_j$. The request of $P_i$ can be granted only if the request edge when converted to an assignment edge does not result in a cycle .If no cycle exists, the system is in a safe state and requests can be granted. If not the system is in an unsafe state and hence in a deadlock. In such a case, requests should not be granted. This is illustrated below (Figure 5.5a, 5.5b).

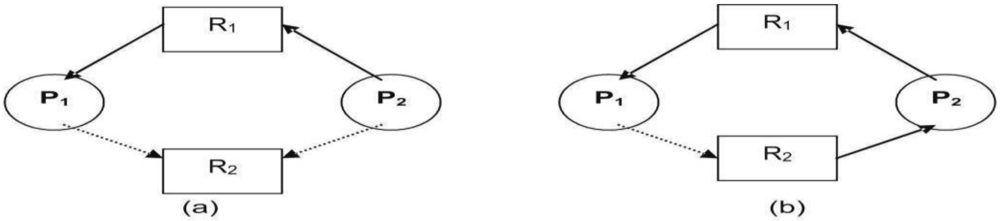


**Figure 5.5: Resource allocation graph showing safe and deadlock states**

Consider the resource allocation graph shown on the left above. Resource $R_2$ is currently free. Allocation of $R_2$ to $P_2$ on request will result in a cycle as shown on the right. Therefore the system will be in an unsafe state. In this situation if $P_1$ requests for $R_2$, then a deadlock occurs

**Banker's Algorithm**

The resource allocation graph algorithm is not applicable where resources have multiple instances. In such a case Banker‘s algorithm is used. A new process entering the system must make known a priori the maximum instances of each resource that it needs subject to the maximum available for each type. As execution proceeds and requests are made, the system checks to see if the allocation of the requested resources ensures a safe state. If so only are the allocations made, else processes must wait for resources.

The following are the data structures maintained to implement the Banker‘s algorithm:

n: Number of processes in the system.

m: Number of resource types in the system.

1)Available: is a vector of length m. Each entry in this vector gives maximum instances of a resource type that are available at the instant.

Available[j] = k means to say there are k instances of the j th resource type R $_j$.

2)Max: is a demand vector of size n x m. It defines the maximum needs of each resource by the process. Max[i][j] = k says the I th process $P_i$ can request for almost k instances of the j th resource type R $_j$.

3)Allocation: is a n x m vector which at any instant defines the number of resources of each type currently allocated to each of the m processes. If Allocation[i][j] = k then I th process $P_i$ is currently holding k instances of the j th resource type $R_j$.

4) Need: is also a n x m vector which gives the remaining needs of the processes. Need[i][j] = k means the ith process $P_i$ still needs k more instances of the j th resource type $R_j$. Thus Need[i][j] = Max[i][j] –Allocation[i][j].

**Safety Algorithm**

Using the above defined data structures, the Banker‘s algorithm to find out if a system is in a safe state or not is described below:

Define a vector Work of length m and a vector Finish of length n.

Initialize Work = Available and Finish[i] = false for i = 1, 2, ., n.

Find an i such that

a. Finish[i] = false and b. Need $_i$ <= Work (Need $_i$ represents the I th row of the vector Need). If such an i does not exist , go to step 5.

Work = Work + Allocation $_i$ Go to step 3.

If finish[i] = true for all i, then the system is in a safe state.

**Resource-Request Algorithm**

Let $\text{Request}_i$ be the vector representing the requests from a process $P_i$. Requesti[j] = k shows that process $P_i$ wants k instances of the resource type $R_j$.

The following is the algorithm to find out if a request by a process can immediately be granted:

1. If $\text{Request}_i$ <= $\text{Need}_i$, go to step 2.

else Error ―request of $P_i$ exceeds $\text{Max}_i$‖ .

2 If $\text{Request}_i$ <= $\text{Available}_i$, go to step 3.

else $P_i$ must wait for resources to be released.

3 An assumed allocation is made as follows: Available = Available – $\text{Request}_i$

$\text{Allocation}_i$ = $\text{Allocation}_i$ + $\text{Request}_i$

$\text{Need}_i$ = $\text{Need}_i$ – $\text{Request}_i$

If the resulting state is safe, then process $P_i$ is allocated the resources and the above changes are made permanent. If the new state is unsafe, then $P_i$ must wait and the old status of the data structures is restored.

**Illustration:** n = 5 < $P_0$, $P_1$, $P_2$, $P_3$, $P_4$ >

M = 3 < A, B, C >

Initially Available = < 10, 5, 7 >

At an instant t0, the data structures have the following values:

| | Allocation | | | Max | | | Available | | | Need | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | [redacted] | | | [redacted] | | | [redacted] | | | [redacted] | |
| | | | | A B | | | | | | A B | |
| | A B C | | | C | | | A B C | | | C | |
| | | | | | | | | | | 7 | |
| Po | 0 1 | | 0 | 7 | 5 | 3 | 3 | 3 | 2 | 4 | 3 |
| $P_1$ | 2 0 | | 0 | 3 | 2 | 2 | | | | 2 | 2 |
| | | | | | | | | | | 6 | |
| P2 | 3 0 | | 2 | 9 | 0 | 2 | | | | 0 | 0 |
| | | | | | | | | | | 0 | |
| P3 | 2 | 1 | 1 | 2 | 2 | 2 | | | | 1 | |
| | | | | | | | | | | 4 | |
| $P_4$ | 0 | 0 | 2 | 4 | 3 | 3 | | | | 3 | |

To find a safe sequence and to prove that the system is in a safe state, use the safety algorithm as follows:

| Step | Work | Finish | Safe sequence |
|---|---|---|---|
| 0 | 3 3 2 | F F F F F | < > |
| 1 | 5 3 2 | F T F F F | < $P_1$ > |
| 2 | 7 4 3 | F T F T F | < $P_1$, $P_3$ > |
| 3 | 7 4 5 | F T F T T | < $P_1$, $P_3$, $P_4$ > |
| 4 | 7 5 5 | T T F T T | < $P_1$, $P_3$, $P_4$, $P_0$ > |
| 5 | 10 5 7 | T T T T T | < $P_1$, $P_3$, $P_4$, $P_0$, $P_2$ > |

Now at an instant $t_1$, $Request_1$ = < 1, 0, 2 >. To actually allocate the requested resources, use the request-resource algorithm as follows:

$Request_1$ < $Need_1$ and $Request_1$ < Available so the request can be considered. If the request is fulfilled, then the new the values in the data structures are as follows:

| | Allocation | | | Max | Available | | | Need | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | [redacted] | | | [redacted] | [redacted] | | | [redacted] | | |
| | | | | | A B | | | A B | | |
| | A B C | | A B C | | C | | | C | | |
| P | | 1 | | | | | | | | |
| 0 | 0 | 0 | 7 | 5 3 | 2 | 3 | 0 | 7 | 4 | 3 |
| | | 0 | | | | | | | | |
| $P_1$ | 3 | 2 | 3 | 2 2 | | | | 0 | 2 | 0 |

| | | | |
|---|---|---|---|
| $P_2$ | 3 0 2 | 9 0 2 | 6 0 0 |
| $P_3$ | 2 1 1 | 2 2 2 | 0 1 1 |
| $P_4$ | 0 0 2 | 4 3 3 | 4 3 1 |

Use the safety algorithm to see if the resulting state is safe:

| Step | Work | Finish | Safe sequence |
|---|---|---|---|
| 0 | 2 3 0 | F F F F F | < > |
| 1 | 5 3 2 | F T F F F | $< P_1 >$ |
| 2 | 7 4 3 | F T F T F | $< P_1, P_3 >$ |
| 3 | 7 4 5 | F T F T T | $< P_1, P_3, P_4 >$ |
| 4 | 7 5 5 | T T F T T | $< P_1, P_3, P_4, P_0 >$ |
| 5 | 10 5 7 | T T T T T | $< P_1, P_3, P_4, P_0, P_2 >$ |

Since the resulting state is safe, request by $P_1$ can be granted.

Now at an instant $t_2$ $Request_4 = < 3, 3, 0 >$. But since $Request_4 >$ Available, the request cannot be granted. Also $Request_0 = < 0, 2, 0>$ at $t_2$ cannot be granted since the resulting state is unsafe as shown below:

| | Allocation | Max | Available | Need |
|---|---|---|---|---|
| | [redacted] | [redacted] | [redacted] | [redacted] |
| | A B C | A B C | A B C | A B C |
| $P_0$ | 0 3 0 | 7 5 3 | 2 1 0 | 2<br>7 3<br>0 |
| $P_1$ | 3 0 2 | 3 2 2 | | 2 0<br>6 |
| $P_2$ | 3 0 2 | 9 0 2 | | 0 0<br>0 |
| $P_3$ | 2 1 1 | 2 2 2 | | 1 1 |
| $P_4$ | 0 0 2 | 4 3 3 | | 4 3<br>1 |

Using the safety algorithm, the resulting state is unsafe since Finish is false for all values of i and we cannot find a safe sequence.

| **Step** | **Work** | **Finish** | **Safe sequence** |
|---|---|---|---|
| 0 | 2 1 0 | F F F F F | < > |

### 5.7 Deadlock Detection

If the system does not ensure that a deadlock cannot be prevented or a deadlock cannot be avoided, then a deadlock may occur. In case a deadlock occurs the system must-

1)Detect the deadlock

2)Recover from the deadlock

**Single Instance of a Resource**

If the system has resources, all of which have only single instances, then a deadlock detection algorithm, which uses a variant of the resource allocation graph, can be used. The graph used in this case is called a wait-for graph. The wait-for graph is a directed graph having vertices and edges. The vertices represent processes and directed edges are present between two processes, one of which is waiting for a resource held by the other. Two edges $P_i$ $R_q$ and $R_q$ $P_j$ in the resource allocation graph are replaced by one edge $P_i$ $P_j$ in the wait-for graph. Thus, the wait-for graph is obtained by removing vertices representing resources and then collapsing the corresponding edges in a resource allocation graph. An Illustration is shown in Figure 5.6.

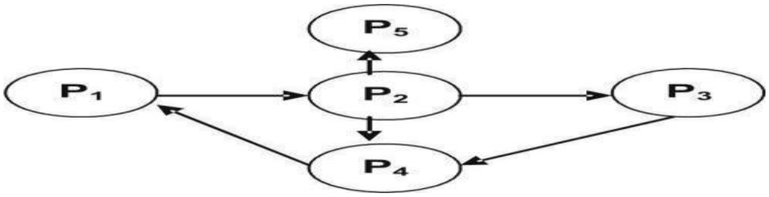


**Figure 5.6: Wait-for graph**

As in the previous case, a cycle in a wait-for graph indicates a deadlock. Therefore, the system maintains a wait-for graph and periodically invokes an algorithm to check for a cycle in the wait-for graph.

**Multiple Instance of a Resource**

A wait-for graph is not applicable for detecting deadlocks where multiple instances of resources exist. This is because there is a situation where a cycle may or may not indicate a deadlock. If this is so then a decision cannot be made. In situations where there are multiple instances of resources, an algorithm similar to Banker‘s algorithm for deadlock avoidance is used.

Data structures used are similar to those used in Banker‘s algorithm and are given below:

□ n: Number of processes in the system.

□ m: Number of resource types in the system.

1) Available: is a vector of length m. Each entry in this vector gives maximum instances of a resource type that are available at the instant.

2) Allocation: is a n x m vector which at any instant defines the number of resources of each type currently allocated to each of the m processes.

3) Request: is also a n x m vector defining the current requests of each process. Request[i][j] = k means the ith process $P_i$ is requesting for k instances of the jth resource type $R_j$.

**ALGORITHM**

Define a vector Work of length m and a vector Finish of length n. 2. Initialize Work = Available and

For I = 1, 2, ….., n
If $\text{Allocation}_i$ != 0

Finish[i] = false

Else

Finish[i] = true

Find an i such that

Finish[i] = false and
$\text{Request}_i$ <= Work

If such an i does not exist , go to step 5.
Work = Work + $\text{Allocation}_i$ Finish[i] = true

Go to step 3.

If finish[i] = true for all i, then the system is not in deadlock.
Else the system is in deadlock with all processes corresponding to Finish[i] = false being deadlocked.
To prove that the system is not deadlocked, use the above algorithm as follows:

| Step | Work | Finish | Safe sequence |
|---|---|---|---|
| 0 | 0 0 0 | F F F F F | < > |
| 1 | 0 1 0 | T F F F F | < $P_0$ > |
| 2 | 3 1 3 | T F T F F | < $P_0$, $P_2$ > |
| 3 | 5 2 4 | T F T T F | < $P_0$, $P_2$, $P_3$ > |
| 4 | 5 2 6 | T F T T T | < $P_0$, $P_2$, $P_3$, $P_4$ > |
| 5 | 7 2 6 | T T T T T | < P0, P2, P3, P4, P1 > |

Now at an instant $t_1$, $\text{Request}_2$ = < 0, 0, 1 > and the new values in the data structures are as follows:

| Allocation | Request | Available |
|---|---|---|
| A B C | A B C | A B C |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **P0** | **0 1 0** | | **0 0**<br>**0** | | | **0** | **0** | **0** |
| **P$_1$** | **2 0** | **0** | **2** | **0** | **2** | | | |
| **P$_2$** | **3 0** | **3** | **0** | **0** | **1** | | | |
| **P$_3$** | **2 1** | **1** | **1** | **0** | **0** | | | |
| **P4** | **0 0** | **2** | **0** | **0** | **2** | | | |

To prove that the system is deadlocked, use the above algorithm as follows:

| **St**<br>**ep** | **Work** | **Finish** | **Safe sequence** |
|---|---|---|---|
| 0 | 0 0 0 | F F F F F | < > |
| 1 | 0 1 0 | T F F F F | < P0 > |

The system is in deadlock with processes P$_1$, P$_2$, P$_3$, and P$_4$ deadlocked.

**Recovery from Deadlock**

Once a deadlock has been detected, it must be broken. Breaking a deadlock may be manual by the operator when informed of the deadlock or automatically by the system. There exist two options for breaking deadlocks:

abort one or more processes to break the circular-wait condition causing deadlock

Preempting resources from one or more processes which are deadlocked.

In the first option, one or more processes involved in deadlock could be terminated to break the deadlock.

Then either abort all processes or abort one process at a time till deadlock is broken. The first case guarantees that the deadlock will be broken. But processes that have executed for a long time will have to restart all over again. The second case is better but has considerable overhead as detection algorithm has to be invoked after terminating every process. Also choosing a process that becomes a victim for termination is based on many factors like priority of the processes length of time each process has executed and how much more it needs for completion type of resources and the instances of each that the processes use need for additional resources to complete nature of the processes, whether iterative or batch. Based on these and many more factors, a process that incurs minimum cost on termination becomes a victim. In the second option some resources are preempted from some processes and given to other processes until the deadlock cycle is broken. Selecting the victim whose resources can be preempted is again based on the minimum cost criteria. Parameters such as number of resources a process is holding and the amount of these resources used thus far by the process are used to select a victim. When resources are preempted, the process holding the resource cannot continue. A simple solution is to abort the process also. Better still is to rollback the process to a safe state to restart later. To determine this safe state, more information about running processes is required which is again an overhead. Also starvation may occur when a victim is selected for preemption; the reason being resources from the same process may again and again be preempted. As a result the process starves for want of resources. Ensuring that a process can be a victim only a finite number of times by having this information as one of the parameters for victim selection could prevent starvation. Prevention, avoidance and detection are the three basic approaches to handle deadlocks. But they do not encompass all the problems encountered. Thus a combined approach of all the three basic approaches is used.

# UNIT V

## FILE SYSTEM INTERFACE

1) **Introduction**
2) **Concept of a File**
3) **File Access methods**
4)**Directory Structure**
5) **Allocation Methods**
6) **Free Space management**
7) **Directory Implementation**

## Introduction

The operating system is a resource manager. Secondary resources like the disk are also to be managed. Information is stored in secondary storage because it costs less, is non-volatile and provides large storage space. Processes access data / information present on secondary storage while in execution. Thus, the operating system has to properly organize data / information in secondary storage for efficient access.

The file system is the most visible part of an operating system. It is a way for on-line storage and access of both data and code of the operating system and the users. It resides on the secondary storage because of the two main characteristics of secondary storage, namely, large storage capacity and non-volatile nature.

## Objectives:

At the end of this unit, you will be able to understand:

The concepts of Files, Different File access methods. Different directory structures, disk space allocation methods, how to manage free space on the disk and implementation of directory.

## Concept of a File

Users use different storage media such as magnetic disks, tapes, optical disks and so on. All these different storage media have their own way of storing information. The operating system provides a uniform logical view of information stored in these different media. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit called a file. These files are then mapped on to physical devices by the operating system during use. The storage devices are usually non-volatile, meaning the contents stored in these devices persist through power failures and system reboots.The concept of a file is extremely general. A file is a collection of related information recorded on the secondary storage. For example, a file containing student information, a file containing employee information, files containing C source code and so on. A file is thus the smallest allotment of logical secondary storage, that is any information to be stored on the secondary storage need to be written on to a file and the file is to be stored. Information in files could be program code or data in numeric, alphanumeric, alphabetic or binary form either formatted or in free form.

A file is therefore a collection of records if it is a data file or a collection of bits / bytes / lines if it is code. Program code stored in files could be source code, object code or executable code whereas data stored in files may consist of plain text, records pertaining to an application, images, sound and so on. Depending on the contents of a file, each file has a pre-defined structure. For example, a file containing text is a collection of characters organized as lines, paragraphs and pages whereas a file containing source code is an organized collection of segments which in turn are organized into declaration and executable statements.

**Attributes of a File**

A file has a name. The file name is a string of characters. For example, test.c, pay.cob, master.dat, os.doc. In addition to a name, a file has certain other attributes. Important attributes among them are:

Type: information on the type of file.

- Location: information is a pointer to a device and the location of the file on that device.
- Size: The current size of the file in bytes.
- Protection: Control information for user access.
- Time, date and user id: Information regarding when the file was created last modified and last used. This information is useful for protection, security and usage monitoring.

All these attributes of files are stored in a centralized place called the directory. The directory is big if the numbers of files are many and also requires permanent storage. It is therefore stored on secondary storage.

**Operations on Files**

A file is an abstract data type. Six basic operations are possible on files. They are:

1. Creating a file: two steps in file creation include space allocation for the file and an entry to be made in the directory to record the name and location of the file.
2. Writing a file: parameters required to write into a file are the name of the file and the contents to be written into it. Given the name of the file the operating system makes a search in the directory to find the location of the file. An updated write pointer eables to write the contents at a proper location in the file.
3. Reading a file: to read information stored in a file the name of the file specified as a parameter is searched by the operating system in the directory to locate the file. An updated read pointer helps read information from a particular location in the file.
4. Repositioning within a file: a file is searched in the directory and a given new value replaces the current file position. No I/O takes place. It is also known as file seek.
5. Deleting a file: The directory is searched for the particular file, If it is found, file space and other resources associated with that file are released and the corresponding directory entry is erased.
6. Truncating a file: file attributes remain the same, but the file has a reduced size because the user deletes information in the file. The end of file pointer is reset.

Other common operations are combinations of these basic operations. They include append, rename and copy. A file on the system is very similar to a manual file. An operation on a file is possible only if the file is open. After performing the operation, the file is closed. All the above basic operations together with the open and close are provided by the operating system as system calls.

**Types of Files**

The operating system recognizes and supports different file types. The most common way of implementing file types is to include the type of the file as part of the file name. The attribute _name' of the file consists of two parts: a name and an extension separated by a period. The extension is the part of a file name that identifies the type of the file. For example, in MS-DOS a file name can be up to eight characters long followed by a period and then a three-character extension. Executable files have a .com / .exe / .bat extension, C source code files have a .c extension, COBOL source code files have a .cob extension and so on. If an operating system can recognize the type of a file then it can operate on the file quite well. For example, an attempt to print an executable file should be aborted since it will produce only garbage. Another use of file types is the capability of the operating system to automatically recompile the latest version of source code to execute the latest modified program.

**Structure of File**

File types are an indication of the internal structure of a file. Some files even need to have a structure that need to be understood by the operating system. For example, the structure of executable files need to be known to the operating system so that it can be loaded in memory and control transferred to the first instruction for execution to begin. Some operating systems also support multiple file structures. Operating system support for multiple file structures makes the operating system more complex. Hence some operating systems support only a minimal number of files structures. A very good example of this type of operating system is the UNIX operating system. UNIX treats each file as a sequence of bytes. It is up to the application program to interpret a file. Here maximum flexibility is present but support from operating system point of view is minimal. Irrespective of any file structure support, every operating system must support at least an executable file structure to load and execute programs. Disk I/O is always in terms of blocks. A block is a physical unit of storage. Usually all blocks are of same size. For example, each block = 512 bytes. Logical records have their own structure that is very rarely an exact multiple of the physical block size. Therefore a number of logical records are packed into one physical block. This helps the operating system to easily locate an offset within a file. For example, as discussed above, UNIX treats files as a sequence of bytes. If each physical block is say 512 bytes, then the operating system packs and unpacks 512 bytes of logical records into physical blocks. File access is always in terms of blocks. The logical size, physical size and packing technique determine the number of logical records that can be packed into one physical block. The mapping is usually done by the operating system. But since the total file size is not always an exact multiple of the block size, the last physical block containing logical records is not full. Some part of this last block is always wasted. On an average half a block is wasted. This is termed internal fragmentation. Larger the physical block size, greater is the internal fragmentation. All file systems do suffer from internal fragmentation. This is the penalty paid for easy file access by the operating system in terms of blocks instead of bits or bytes.

**File Access Methods**

Information is stored in files. Files reside on secondary storage. When this information is to be used, it has to be accessed and brought into primary main memory. Information in files could be accessed in many ways. It is usually dependent on an application. Access methods could be :-

5. Sequential access
6. Direct access
7. Indexed sequential access

**Sequential Access**

In a simple access method, information in a file is accessed sequentially one record after another. To process the i$^{th}$ record all the i-1 records previous to I must be accessed. Sequential access is based on the tape model that is inherently a sequential access device. Sequential access is best suited where most of the records in a file are to be processed. For example, transaction files.

**Direct Access**

Sometimes it is not necessary to process every record in a file. It may not be necessary to process records in the order in which they are present. Information present in a record of a file is to be accessed only if some key value in that record is known. In all such cases, direct access is used. Direct access is based on the disk that is a direct access device and allows random access of any file block. Since a file is a collection of physical blocks, any block and hence the records in that block are accessed. For example, master files. Databases are often of this type since they allow query processing that involves immediate access to large amounts of information. All reservation systems fall into this category. Not all operating systems support direct access files. Usually files are to be defined as sequential or direct at the time of creation and accessed accordingly later. Sequential access of a direct access file is possible but direct access of a sequential file is not.

**Indexed Sequential Access**

This access method is a slight modification of the direct access method. It is in fact a combination of both the sequential access as well as direct access. The main concept is to access a file direct first and then sequentially from that point onwards.

This access method involves maintaining an index. The index is a pointer to a block. To access a record in a file, a direct access of the index is made. The information obtained from this access is used to access the file. For example, the direct access to a file will give the block address and within the block the record is accessed sequentially. Sometimes indexes may be big. So a hierarchy of indexes are built in which one direct access of an index leads to info to access another index directly and so on till the actual file is accessed sequentially for the particular record. The main advantage in this type of access is that both direct and sequential access of files is possible.

### Directory Structure

Files systems are very large. Files have to be organized. Usually a two level organization is done:

= The file system is divided into partitions. In Default there is at least one partition. Partitions are nothing but virtual disks with each partition considered as a separate storage device.

= Each partition has information about the files in it. This information is nothing but a table of contents. It is known as a directory.

The directory maintains information about the name, location, size and type of all files in the partition. A directory has a logical structure. This is dependent on many factors including operations that are to be performed on the directory like search for file/s, create a file, delete a file, list a directory, rename a file and traverse a file system. For example, the dir, del, ren commands in MS-DOS.

### Single-Level Directory

This is a simple directory structure that is very easy to support. All files reside in one and the same directory (Figure 8.1).

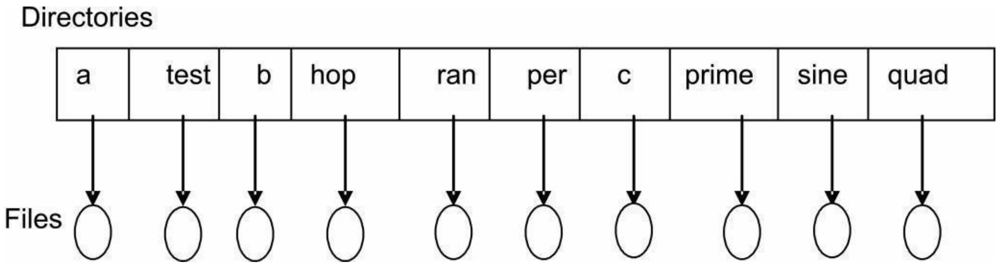


**Figure 8.1: Single-level directory structure**

A single-level directory has limitations as the number of files and users increase. Since there is only one directory to list all the files, no two files can have the same name that is, file names must be unique in order to identify one file from another. Even with one user, it is difficult to maintain files with unique names when the number of files becomes large.

### Two-Level Directory

The main limitation of single-level directory is to have unique file names by different users. One solution to the problem could be to create separate directories for each user. A two-level directory structure has one directory exclusively for each user. The directory structure of each user is similar in structure and maintains file information about files present in that directory only. The operating system has one master directory for a partition. This directory has entries for each of the user directories (Figure 8.2).Files with same names exists across user directories but not in the same user directory. File maintenance is easy. Users are isolated from one another. But when users work in a group and each wants to access files in another users directory, it may not be possible. Access to a file is through user name and file name. This is known as a path. Thus a path uniquely defines a file. For example, in MS-DOS if _C‘ is the partition then C:\USER1\TEST, C:\USER2\TEST, C:\USER3\C is all files in user directories. Files could be created, deleted, searched and renamed in the user directories only.

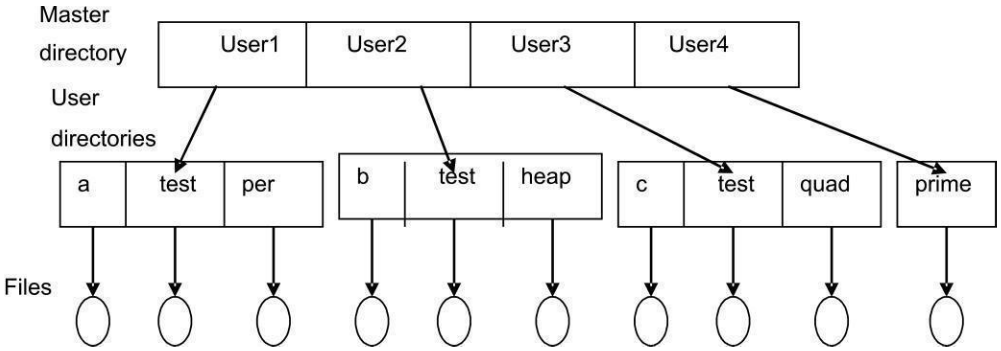


**Figure 8.2: Two-level directory structure**

### Tree-Structured Directories

A two-level directory is a tree of height two with the master file directory at the root having user directories as descendants that in turn have the files themselves as descendants(Figure 8.3). This generalization allows users to organize files within user directories into sub directories. Every file has a unique path. Here the path is from the root through all the sub directories to the specific file. Usually the user has a current directory. User created sub directories could be traversed.

Files are usually accessed by giving their path names.Path names could be either absolute or relative. Absolute path names begin with the root and give the complete path down to the file. Relative path names begin with the current directory. Allowing users to define sub directories allows for organizing user files based on topics. A directory is treated as yet another file in the directory, higher up in the hierarchy. To delete a directory it must be empty. Two options exist: delete all files and then delete the directory or delete all entries in the directory when the directory is deleted. Deletion may be a recursive process since directory to be deleted may contain sub directories.

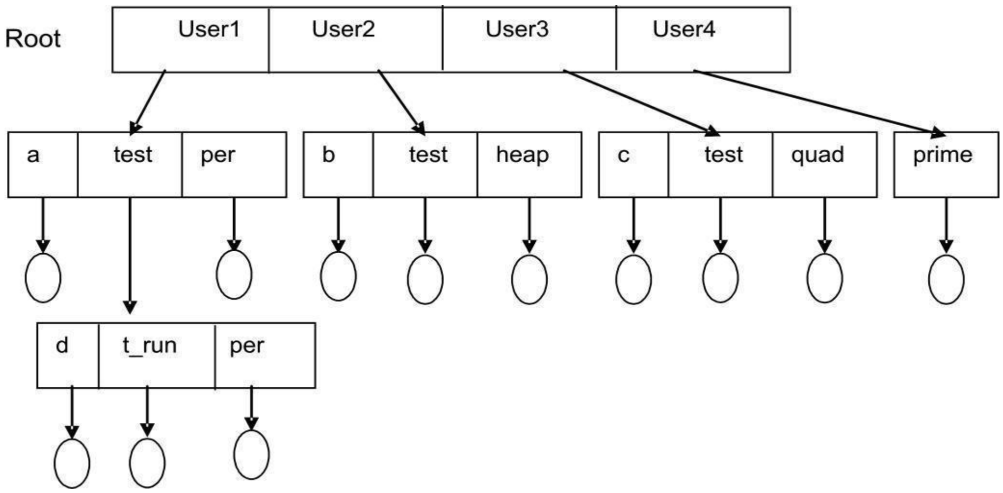


**Figure 8.3: Tree-structured directory structure**

## Allocation Methods

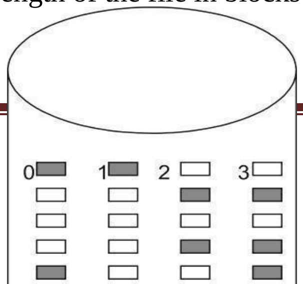
Allocation of disk space to files is a problem that looks at how effectively disk space is utilized and quickly files can be accessed. The three major methods of disk space allocation are:

1) Contiguous allocation
2) Linked allocation
3) Indexed allocation

## Contiguous Allocation

Contiguous allocation requires a file to occupy contiguous blocks on the disk. Because of this constraint disk access time is reduced, as disk head movement is usually restricted to only one track. Number of seeks for accessing contiguously allocated files is minimal and so also seek times. A file that is _n‘ blocks long starting at a location _b‘ on the disk occupies blocks b, b+1, b+2, ….., b+(n-1).

The directory entry for each contiguously allocated file gives the address of the starting block and the length of the file in blocks as illustrated below (Figure 8.4).



**Directory:**

| File | Start | Length |
|---|---|---|
| a.c | 0 | 2 |
| per | 14 | 3 |
| hop | 19 | 6 |
| b | 28 | 4 |
| f | 6 | 2 |

**Figure 8.4: Contiguous allocation**

Accessing a contiguously allocated file is easy. Both sequential and random access of a file is possible. If a sequential access of a file is made then the next block after the current is accessed, whereas if a direct access is made then a direct block address to the i$^{th}$ block is calculated as b+i where b is the starting block address. A major disadvantage with contiguous allocation is to find contiguous space enough for the file. From a set of free blocks, a first-fit or best-fit strategy is adopted to find _n‘ contiguous holes for a file of size _n‘. But these algorithms suffer from external fragmentation. As disk space is allocated and released, a single large hole of disk space is fragmented into smaller holes. Sometimes the total size of all the holes put together is larger than the size of the file size that is to be allocated space. But the file cannot be allocated space because there is no contiguous hole of size equal to that of the file. This is when external fragmentation has occurred. Compaction of disk space is a solution to external fragmentation. But it has a very large overhead. Another problem with contiguous allocation is to determine the space needed for a file. The file is a dynamic entity that grows and shrinks. If allocated space is just enough (a best-fit allocation strategy is adopted) and if the file grows, there may not be space on either side of the file to expand. The solution to this problem is to again reallocate the file into a bigger space and release the existing space. Another solution that could be possible if the file size is known in advance is to make an allocation for the known file size. But in this case there is always a possibility of a large amount of internal fragmentation because initially the file may not occupy the entire space and also grow very slowly.

**Linked Allocation**

Linked allocation overcomes all problems of contiguous allocation. A file is allocated blocks of physical storage in any order. A file is thus a list of blocks that are linked together. The directory contains the address of the starting block and the ending block of the file. The first block contains a pointer to the second, the second a pointer to the third and so on till the last block (Figure 8.5) Initially a block is allocated to a file, with the directory having this block as the start and end. As the file grows, additional blocks are allocated with the current block containing a pointer to the next and the end block being updated in the directory. This allocation method does not suffer from external fragmentation because any free block can satisfy a request. Hence there is no need for compaction. moreover a file can grow and shrink without problems of allocation.

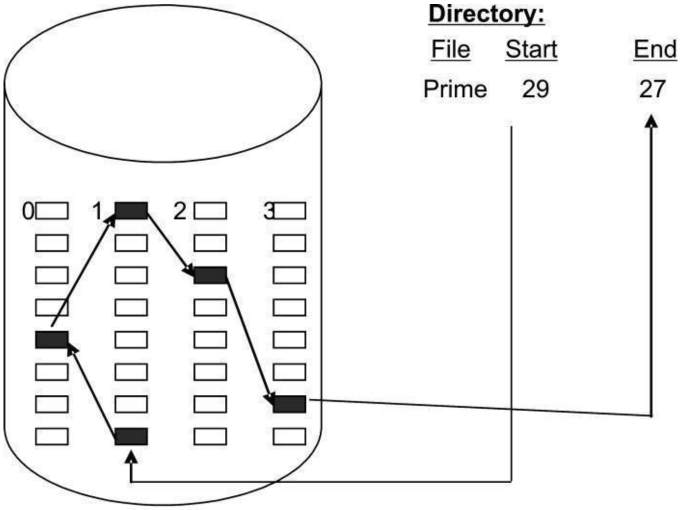


**Figure 8.5: Linked allocation**

Linked allocation has some disadvantages. Random access of files is not possible. To access the i$^{th}$ block access begins at the beginning of the file and follows the pointers in all the blocks till the i$^{th}$ block is accessed. Therefore access is always sequential. Also some space in all the allocated blocks is used for storing pointers.

This is clearly an overhead as a fixed percentage from every block is wasted. This problem is overcome by allocating blocks in clusters that are nothing but groups of blocks. But this tends to increase internal fragmentation. Another problem in this allocation scheme is that of scattered pointers. If for any reason a pointer is lost, then the file after that block is inaccessible. A doubly linked block structure may solve the problem at the cost of additional pointers to be maintained.MS-DOS uses a variation of the linked allocation called a file allocation table (FAT). The FAT resides on the disk and contains entry for each disk block and is indexed by block number. The directory contains the starting block address of the file. This block in the FAT has a pointer to the next block and so on till the last block (Figure 8.6). Random access of files is possible because the FAT can be scanned for a direct block address.

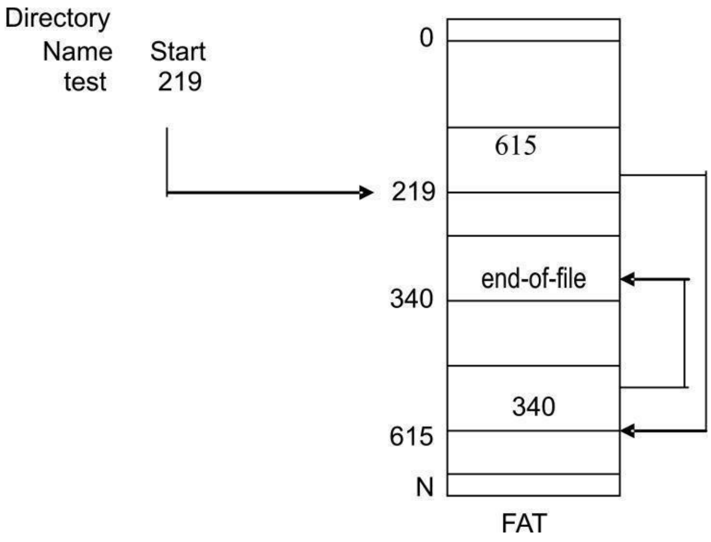


**Figure 8.6: File allocation table**

### Indexed Allocation

Problems of external fragmentation and size declaration present in contiguous allocation are overcome in linked allocation. But in the absence of FAT, linked allocation does not support random access of files since pointers hidden in blocks need to be accessed sequentially. Indexed allocation solves this problem by bringing all pointers together into an index block. This also solves the problem of scattered pointers in linked allocation. Each file has an index block. The address of this index block finds an entry in the directory and contains only block addresses in the order in which they are allocated to the file. The $i^{th}$ address in the index block is the $i^{th}$ block of the file (Figure

8.7). Here both sequential and direct access of a file is possible. Also it does not suffer from external fragmentation.

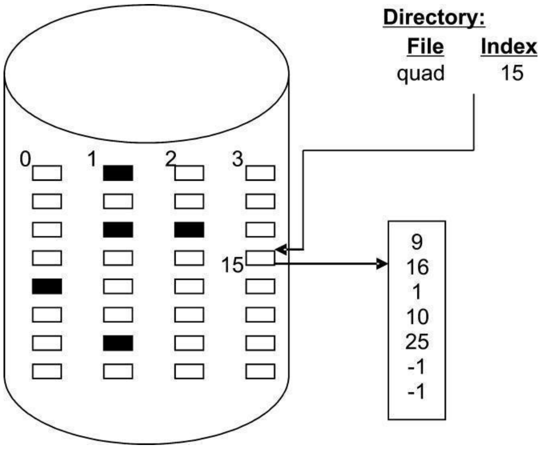


**Figure 8.7: Indexed Allocation**

Indexed allocation does suffer from wasted block space. Pointer overhead is more in indexed allocation than in linked allocation. Every file needs an index block. Then what should be the size of the index block? If it is too big, space is wasted. If it is too small, large files cannot be stored. More than one index blocks are linked so that large files can be stored. Multilevel index blocks are also used. A combined scheme having direct index blocks as well as linked index blocks has been implemented in the UNIX operating system.

**Performance Comparison**

All the three allocation methods differ in storage efficiency and block access time. Contiguous allocation requires only one disk access to get a block, whether it be thenext block (sequential) or the $i^{th}$ block (direct). In the case of linked allocation, the address of the next block is available in the current block being accessed and so is very much suited for sequential access. Hence direct access files could use contiguous allocation and sequential access files could use linked allocation. But if this is fixed then the type of access on a file needs to be declared at the time of file creation. Thus a sequential access file will be linked and cannot support direct access. On the other hand a direct access file will have contiguous allocation and can also support sequential access, the constraint in this case is making known the file length at the time of file creation. The operating system will then have to support algorithms and data structures for both allocation methods. Conversion of one file type to another needs a copy operation to the desired file type. Some systems support both contiguous and linked allocation. Initially all files have contiguous allocation. As they grow a switch to indexed allocation takes place. If on an average files are small, than contiguous file allocation is advantageous and provides good performance.

## Free Space Management

The disk is a scarce resource. Also disk space can be reused. Free space present on the disk is maintained by the operating system. Physical blocks that are free are listed in a free-space list. When a file is created or a file grows, requests for blocks of disk space are checked in the free-space list and then allocated. The list is updated accordingly. Similarly, freed blocks are added to the free-space list. The free-space list could be implemented in many ways as follows:

### Bit Vector

A bit map or a bit vector is a very common way of implementing a free-space list. This vector _n‘ number of bits where _n‘ is the total number of available disk blocks. A free block has its corresponding bit set (1) in the bit vector whereas an allocated block has its bit reset (0).

Illustration: If blocks 2, 4, 5, 9, 10, 12, 15, 18, 20, 22, 23, 24, 25, 29 are free and the rest are allocated, then a free-space list implemented as a bit vector would look as shown below:

00101100011010010010101111000100000………

The advantage of this approach is that it is very simple to implement and efficient to access. If only one free block is needed then a search for the first _1‘ in the vector is necessary. If a contiguous allocation for _b‘ blocks is required, then a contiguous run of _b‘ number of 1‘s is searched. And if the first-fit scheme is used then the first such run is chosen and the best of such runs is chosen if best-fit scheme is used. Bit vectors are inefficient if they are not in memory. Also the size of the vector has to be updated if the size of the disk changes.

### Linked List

All free blocks are linked together. The free-space list head contains the address of the first free block. This block in turn contains the address of the next free block and so on. But this scheme works well for linked allocation. If contiguous allocation is used then to search for _b‘ contiguous free blocks calls for traversal of the free-space list which is not efficient. The FAT in MS-DOS builds in free block accounting into the allocation data structure itself where free blocks have an entry say –1 in the FAT.

### Grouping

Another approach is to store _n‘ free block addresses in the first free block. Here (n-blocks are actually free. The last nth address is the address of a block that contains the next set of free block addresses. This method has the advantage that a large number of free block addresses are available at a single place unlike in the previous linked approach where free block addresses are scattered.

**Counting**

If contiguous allocation is used and a file has freed its disk space then a contiguous set of _n‘ blocks is free. Instead of storing the addresses of all these _n‘ blocks in the free-space list, only the starting free block address and a count of the number of blocks free from that address can be stored. This is exactly what is done in this scheme where each entry in the free-space list is a disk address followed by a count.

## Directory Implementation

The two main methods of implementing a directory are:

**1) Linear list**

**2) Hash table**

**Linear List**

A linear list of file names with pointers to the data blocks is one way to implement a directory. A linear search is necessary to find a particular file. The method is simple but the search is time consuming. To create a file, a linear search is made to look for the existence of a file with the same file name and if no such file is found the new file created is added to the directory at the end. To delete a file, a linear search for the file name is made and if found allocated space is released. Every time making a linear search consumes time and increases access time that is not desirable since a directory information is frequently used. A sorted list allows for a binary search that is time efficient compared to the linear search. But maintaining a sorted list is an overhead especially because of file creations and deletions.

**Hash table**

Another data structure for directory implementation is the hash table. A linear list is used to store directory entries. A hash table takes a value computed from the file name and returns a pointer to the file name in the linear list. Thus search time is greatly reduced. Insertions are prone to collisions that are resolved. The main problem is the hash function that is dependent on the hash table size. A solution to the problem is to allow for chained overflow with each hash entry being a linked list. Directory lookups in a hash table are faster than in a linear list.