D.N.R.COLLEGE(AUTONOMOUS)::BHIMAVARAM

I MCA

I SEMSETER

E-CONTENT

COMPUTER ORGANIZATION

Presented by
K.VENKATESH

MCA-20104 COMPUTER ORGANIZATION

Instruction: 4 Periods/week	Time: 3 Hours	Credits: 4
Internal: 25 Marks	External: 75 Marks	Total: 100 Marks

UNIT-I

Digital Logic Circuits:

Digital Computers, Logic Gates, Boolean Algebra, Map Simplification, Combinational Circuit, Flip-flops Sequential Circuits.

Digital Components:

Integrated Circuits, Decoders, Multiplexes, Registers, Shift Registers, Counters, Memory Unit.

UNIT-II

Data Representation:

Data Types, Complements, Fixed-point Representation, Floating point Representation.

Register Transfer and Micro Operations:

Register Transfer Language, Register Transfer, Bus and Memory Transfer, Arithmetic Micro Operations, Assembly language Instructions, 8085 Microprocessor Instruction Set,8085 Architecture.

UNIT-III

Basic Computer Organization and Design:

Instruction Codes, Computer Register, Computer Instructions, Timing and Control, Instruction Cycle, Memory Reference Instructions, Input-Output, Interrupt.

Central Processing Unit:

Introduction, General Register Organization, Stack Organization, Instruction formats, addressing modes.

UNIT-IV

Input /Output Organization:

Peripherals Devices, I/O Interface, Asynchronous Data Transfer, Mode of Transfer, Priority Interrupt, Direct memory access, Input – Output Processor(IOP).

Memory Organization:

Memory Hierarchy, Main memory, Auxiliary Memory, Associate Memory, Cache Memory and Virtual Memory.

Text Books:
1.Computer System Architecture, M.Morris Mano, Prentice Hall of India Pvt.ltd. Third Edition, Sept. 2008.
2. B. Ram, "Fundamentals of Microprocessors and Microcomputers", Dhanpat Rai Publications.
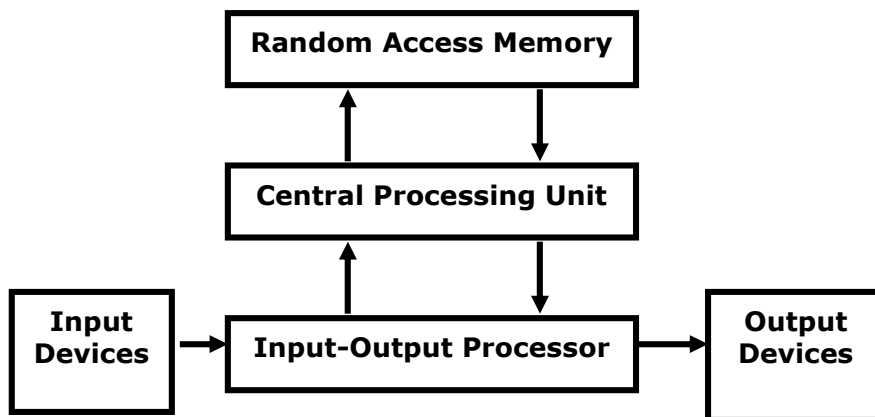**Reference Books:**
1.Computer Architecture and Organization, William Stallings, PHI Pvt. Ltd. Eastern Economy Edition, Sixth Edition, 2003.
2.Computer System Architecture John P. Hayes.

UNIT 1: DIGITAL LOGICAL CIRCUITS

What is Digital Computer? OR Explain the block diagram of digital computers.

- Digital computer is a digital system that performs various computational tasks.
- The word "DIGITAL" implies that the information in the computer is represented by digits.
- 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 the groups of bit.
- By using various coding techniques, groups of bits can be made to represent not only binary numbers but also other symbols, letters of alphabets and decimal digit.
- Bits are grouped together as *bytes* and *words* to form some type of representation within the computer.
- A sequence of instructions for the computer is known as *program*.

Block diagram of a digital computer

```
                  ┌──────────────────────────┐
                  │  Random Access Memory     │
                  └──────────────────────────┘
                       ↑            │
                       │            ↓
                  ┌──────────────────────────┐
                  │  Central Processing Unit  │
                  └──────────────────────────┘
                       ↑            │
                       │            ↓
┌──────────┐     ┌──────────────────────────┐     ┌──────────┐
│  Input   │ →   │  Input-Output Processor   │ →   │  Output  │
│ Devices  │     └──────────────────────────┘     │ Devices  │
└──────────┘                                       └──────────┘
```

- The hardware of the computer is usually divided into three major parts.
- The Central processing Unit (CPU) contains an arithmetic and logic unit 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 and output processor contains electronic circuit for communication and controlling the transfer of information between the computer and the outside world.
- The input and output device connected to the computer include keyboards, printers, terminals, magnetic disk drives and other communication devices.

What is Gates? Explain the Logic Gates in brief.

- Binary information is represented in digital computers using electrical signals.
- These signals can be represented by voltage to specify one of two possible states.
- The two states represent a binary variable that can be equal to 1 or 0.
- The manipulation of binary information in a computer is done using logic circuits called *gates*.

- Gates are blocks of hardware that produce signals of binary 1 or 0 when input logic requirements are satisfied.
- There are various types of logic gates are commonly used in digital computer.
- Each gate has a different graphic symbols and operation.
- The input-output relationship of binary variables for each gate can be represented in tabular form by Truth-Table.
- There are three types of gates:
  - Basic / Fundamental Gates (AND, OR, NOT)
  - Universal Gates (NAND, NOR)
  - Exclusive Gates (EX-OR, EX-NOR)

LOGICAL GATES

Basic / Fundamental Gates	Universal Gates	Exclusive Gates
(AND, OR, NOT)	(NAND, NOR)	(EX-OR, EX-NOR)
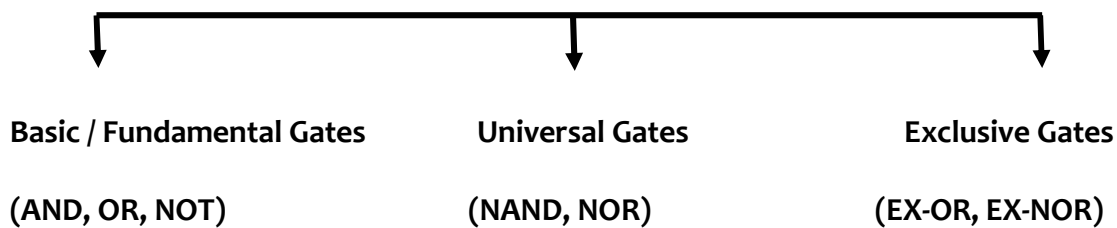
Basic Gates

AND Gate:
- In this type of gate output is high only when all its inputs are high.
- If any single input is law then the output will remain low.
- So it is said that in AND gate the output is only high when the input is also high.

SYMBOL:

$$C = A * B$$

TRUTH-TABLE:

INPUT		OUTPUT
A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

OR Gate:
- In this type of gate if any input signal is high then the output will be high.
- The output is only low only when all the inputs are low

$$C = A + B$$

TRUTH-TABLE:

INPUT		OUTPUT
A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

NOT Gate:
- This type of gate is also known as "Inverter".
- It is a gate that contains only one input and only one output.
- The output is always opposite than the input signals.

SYMBOL:

TRUTH-TABLE:

INPUT	OUTPUT
A	NOT A (A')
0	1
1	0

Universal Gates

NAND and NOR gates are known as universal gates because we can construct any gate using NAND & NOR gate.

NOR Gate:
- The NOR gate is the complement of the OR gate.
- As shown in the truth table that the output of NOR gate is exactly opposite than the output of OR gate.
- This means that the output will be high when all the input is low.

SYMBOL:

$$C = \overline{A + B}$$

TRUTH-TABLE:

INPUT		OUTPUT
A	B	A NOR B
0	0	1
0	1	0
1	0	0
1	1	0

NAND Gate:

- The NAND gate is an AND gate followed by NOT gate.
- As shown in the truth table that the output of NAND gate is exactly opposite than the output of AND gate.
- This means that the output will be high when all the input is high.

SYMBOL:

$$C = \overline{A \cdot B}$$

TRUTH-TABLE:

INPUT		OUTPUT
A	B	A NAND B
0	0	1
0	1	1
1	0	1
1	1	0

Exclusive Gates

EX-OR Gate:

- This gate is produces high output whenever the two inputs are at opposite level.
- The EX-OR gate is the gate that produces high output for Odd number of high inputs.

SYMBOL:

$$C = A \oplus B$$

TRUTH-TABLE:

INPUT		OUTPUT
A	B	A EX-OR B
0	0	0
0	1	1
1	0	1
1	1	0

EX-NOR Gate:

- This gate is produces high output whenever the two inputs are at same level.
- The EX-OR gate is the gate that produces high output for Even number of high inputs.
- The truth table shows that output of this gate is exactly opposite of EX-OR gate.

SYMBOL:

$$C = \overline{A \oplus B}$$

TRUTH-TABLE:

INPUT		OUTPUT
A	B	A EX-NOR B
0	0	1
0	1	0
1	0	0
1	1	1

Write a note on Boolean Algebra

- In 1854 George Boole introduced a systemic treatment of logic and developed for this purpose an algebric system called Boolean Algebra.
- Boolean Algebra is an algebra that deals with binary variables and logic operations.
- The variables are designated by letters such as A,B, X ,Y etc.
- The three basic operations are AND, OR and complement.
- A Boolean function can be expressed with binary variable, the logic operation symbols, parentheses (rounded bracket) and equal to (=) sign.
- The result of a Boolean function is either 0 or 1.

- A Boolean function can be represented by either:

    a. Truth tables
    b. Logic diagrams
    c. Algebraic expression

- For example: F=x+y'z
    - F=1 only if x is 1 or if both y' and z=1.
    - If y'(complement of y)=1 means that y=0 so we can say that F=1 only when x=1,y=0,z=1.
    - So we can say that function F equal to 1 for those combination where x=1 or yz=01
- A Boolean function can be transformed form algebraic expression into a logic diagram composed of AND,OR and NOT gates.
- **Truth table and logic diagram For above example :**

x	Y	z	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1
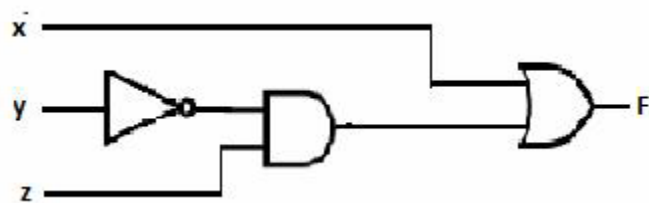
Boolean Operations

There are three basic logical operations:

- AND: This operation is represented by a dot or by the absence of an operator. For example, x.y = z or xy = z is read "x AND y is equal to z." The logical operation AND is interpreted to mean that z=1 if and only if x=1 and y=1; otherwise z=0.
- OR: This operation is represented by a plus sign. For example, x+y = z is read "x OR y is equal to z", meaning that z=1 if x=1 or if y=1 or if both x=1 and y=1. If both x=0 and y=0, then z=0.
- NOT: This operation is represented by a prime (sometimes by a bar). For example, $x'=z$ (or $\bar{x} = z$) is read "x not is equal to z", meaning that z is complement of x. In other words, if x=1, then z=0, but if
- x=0, then z=1.

Basic Identities of Boolean Algebra

Postulates and theorems of Boolean Algebra

(a) $x + 0 = x$ (b) $x \cdot 1 = x$

(a) $x + x' = 1$ (b) $x \cdot x' = 0$

(a) $x + x = x$ (b) $x \cdot x = x$

(a) $x + 1 = 1$ (b) $x \cdot 0 = 0$

$(x')' = x$

(a) $x + y = y + x$ (b) $xy = yx$

(a) $x + (y + z) = (x + y) + z$ (b) $x(yz) = (xy)z$

(a) $x(y + z) = xy + xz$ (b) $x + yz = (x + y)(x + z)$

(a) $(x + y)' = x'y'$ (b) $(xy)' = x' + y'$

(a) $x + xy = x$ (b) $x(x + y) = x$

Write a note on DeMorgan's theorem

- It is developed greater mathematician and logician named De-Morgan.
- He developed two theorems which makes the complement of questions and product(**incomplete line).**
- It is very important in dealing with NOR and NAND gates.
- The 2 most important theorems of De-Morgan are as follows:
  1) **The complement of sum equal to the product of the complement.**
  $$\overline{x + y} = \overline{x} \cdot \overline{y}$$

  2) **The complement of product equal to the sum of the complement.**
  $$\overline{x \cdot y} = \overline{x} + \overline{y}$$
- We can prove the theorems with the help of truth table.

- **THEOREM 1:** $\overline{x + y} = \overline{x} \cdot \overline{y}$

x	y	x'	y'	(x+y)	(x+y)'	x'. y'
0	0	1	1	0	1	1
0	1	1	0	1	0	0
1	0	0	1	1	0	0
1	1	0	0	1	0	0

- Last 2 columns gives same output so LHS=RHS

THEOREM 2: $\overline{x \cdot y} = \overline{x} + \overline{y}$

x	y	x'	y'	(x.y)	(x.y)'	x' + y'
0	0	1	1	0	1	1
0	1	1	0	0	1	1
1	0	0	1	0	1	1
1	1	0	0	1	0	0

- Last 2 columns gives same output so LHS=RHS

Write a note on K- Maps.

- The Karnaugh map is also referred as Veitch Diagrams, KV maps or K-maps.
- K-map is a method to minimizes the Boolean function.
- K-map provides a simple and straight forward method to minimizing Boolean expression.
- With the help of K-map we can simplified Boolean expression up to 4 and 6 variables.
- K-map diagram represents squares and each square represents 1 minterm.
- In K-map values of the variables are written in binary form & the logic function can be expressed in one of the following form
    - **SUM OF PRODUCTS (SOP)**
    - **PRODUCT OF SUM (POS)**
- A K-map for n variables is made up of **2^n** squares and each squares designed a product term of Boolean expression.
- For product terms which are present in expression, 1s are written in correspondence squares and 0 will be written in blank square.
- **For example: K-map for 2 variables:**

- F =xy' + x'y

k- map for 3 variables

x \ yz	00	01	11	10
0	x'y'z'	x'y'z	x'yz	x'yz'
1	xy'z'	xy'z	xyz	xyz'

k-map for 4 variables

wx \ yz	00	01	11	10
00	w'x'y'z'	w'x'y'z	w'x'yz	w'x'yz'
01	w'xy'z'	w'xy'z	w'xyz	w'xyz'
11	wxy'z'	wxy'z	wxyz	wxyz'
10	wx'y'z'	wx'y'z	wx'yz	wx'yz'

- **RULES FOR K- MAP:**
- Each cell with 1 must be included in at list 1 group.
- Try to form the largest possible groups.
- Try to end up with as few groups as possible.
- Groups may be in sizes that are powered of 2.
- Groups may be square or rectangular only.
- Groups may be horizontal or vertical but not diagonal.
- Groups may wrap around the table.
- Groups may overleap.
- The larger a group is, the more redundant inputs there are:
  - Group of 1 has no redundant input.
  - Group of 2 known as pair has 1 redundant input.
  - Group of 4 known as quad has 2 redundant input.
  - Group of 8 known as octet has 3 redundant input.

Sum-of-Products Simplification

- A Boolean function represented by a truth table is plotted into the map by inserting 1's into those squares where the function is 1.
- Boolean functions can then be simplified by identifying adjacent squares in the Karnaugh map that contain a 1.
- A square is considered adjacent to another square if it is next to, above, or below it. In addition, squares at the extreme ends of the same horizontal row are also considered

adjacent. The same applies to the top and bottom squares of a column. The objective to identify adjacent squares containing 1's and group them together.

- Groups must contain a number of squares that is an integral power of 2.
- Groups of combined adjacent squares may share one or more squares with one or more groups.
- Each group of squares represents an algebraic term, and the **OR** of those terms gives the simplified algebraic expression for the function.
- To find the most simplified algebraic expression, the goal of map simplification is to identify the least number of groups with the largest number of members.

We will simplify the Boolean function.
$F(A,B,C) = \Sigma(3,4,6,7)$

Map for F(A,B,C) = Σ(3,4,6,7)
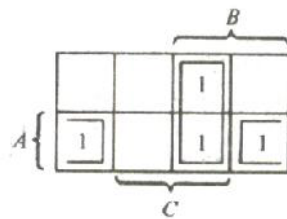
- The three variable maps for this function is shown in the figure 2.4
- There are four squares marked with 1's, one for each minterm that produces 1 for the function. These squares belong to minterm 3,4,6,7 and are recognized from the figure b.
- Two adjacent squares are combined in the third column. This column belongs to both B and C produces the term BC.
- The remaining two squares with 1's in the two corner of the second row are adjacent and belong to row columns of C', so they produce the term AC'.
- The simplified expression for the function is the or of the two terms:
  F = BC + AC'

The second example simplifies the following Boolean function:
  $F(A,B,C) = \Sigma(0,2,4,5,6)$

- The five minterms are marked with 1's in the corresponding squares of the three variable maps.
- The four squares in the first and the fourth columns are adjacent and represent the term C'.
- The remaining square marked with a 1 belongs to minterm 5 and can be combined with the square of minterm 4 to produce the term AB'.
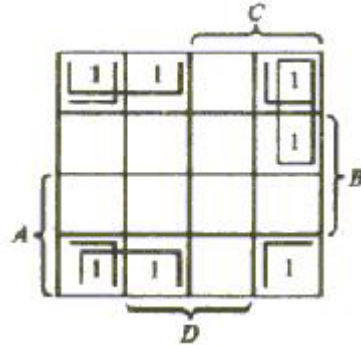
The simplified function is
  F = C'+AB'
Map for F(A,B,C) = Σ(0,2,4,5,6)
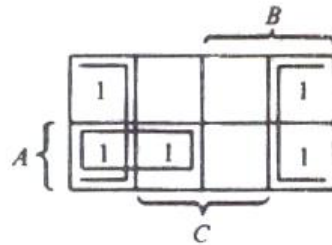
Figure 2.6 Map for F(A,B,C,D) = Σ(0,1,2,6,8,9,10)

- The area in the map covered by this four variable consists of the squares marked with 1's in fig 1.10. The function contains 1's in the four corners that when taken as groups give the term B'D'. This is possible because these four squares are adjacent when the map is considered with the top and bottom or left and right edges touching.
- The two 1's on the bottom row are combined with the two 1's on the left of the bottom row to give the term B'C'.
- The remaining 1 in the square of minterm 6 is combined with the minterm 2 to give the term A'CD'.

The simplified function is:
F = B'D' + B'C' + A'CD'

Product-of-Sums Simplification
- Another method for simplifying Boolean expressions can be to represent the function as a product of sums.
- This approach is similar to the Sum-of-Products simplification, but identifying adjacent squares containing 0's instead of 1's forms the groups of adjacent squares.
- Then, instead of representing the function as a sum of products, the function is represented as a product of sums.

    Examples
    F(A,B,C,D) = Σ (0,1,2,5,8,9,10)

The 1's marked in the map of figure 2.7 represents the minterms that produces a 1 for the function.
The squares marked with 0's represent the minterm not included in F and therefore denote the complement of F.
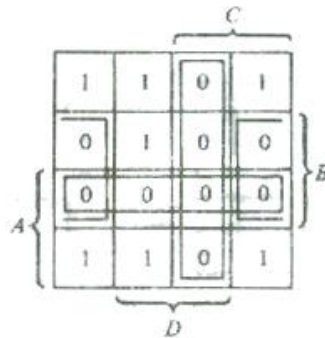Combining the squares with 1's gives the simplified function in sum-of-products form:
            F = B'D +B'C'+A'C'D

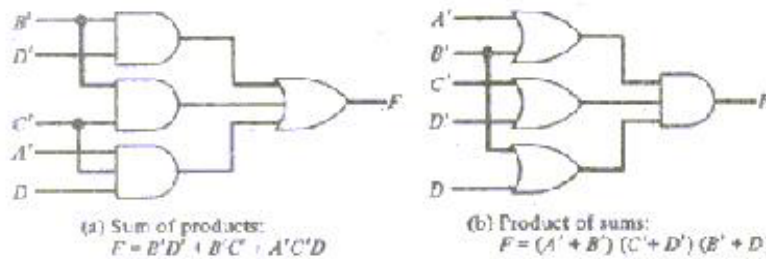If the squares marked with 0's are combined as shown in the diagram, we obtain the simplified complement function:

$$F'=(A'+B')(C'D')(B'+D)$$

Figure 2.7 Map for F(A,B,C,D) = Σ (0,1,2,5,8,9,10)

The logic diagram of the two simplified expression are shown in fig 2.8

Logic Diagram with AND and OR gates

(a) Sum of products:
$F = B'D' + B'C' + A'C'D$

(b) Product of sums:
$F = (A'+B')(C'+D')(B'+D)$

- The sum of product expression us implemented in fig 2.8(a) with a group of of AND gates, one for each AND term.
- The output of the AND gates are connected to the inputs of a single OR gate. The same function is implemented in fig 2.8(b) in product of sum form with a group of OR gates, one for each OR term, the outputs of the OR gates are connected to the inputs of a single And gate.
- In each case it is assumed that the input variable are directly available in their complement, so inverter are not included.

Write a note on Combinational Circuits

- A combinational circuit is the circuit where more than 1 circuit is designed into single component.
- It has N no of inputs and M no of outputs.
- It is basically used to design digital applications and it transforms the data into the digital manner.
- A combinational circuit is a connected arrangement of logic gates with a set of inputs and outputs.

- At any given time, the binary values of the outputs are a function of the binary values of the inputs.
- The design of a combinational circuit starts from a verbal outline of the problem and ends in a logic circuit diagram. The procedure involves the following steps:

    1. The problem is stated.
    2. The input and output variables are assigned letter symbols.
    3. The truth table that defines the relationship between inputs and outputs is derived.
    4. The simplified Boolean functions for each output are obtained.
    5. The logic diagram is drawn.
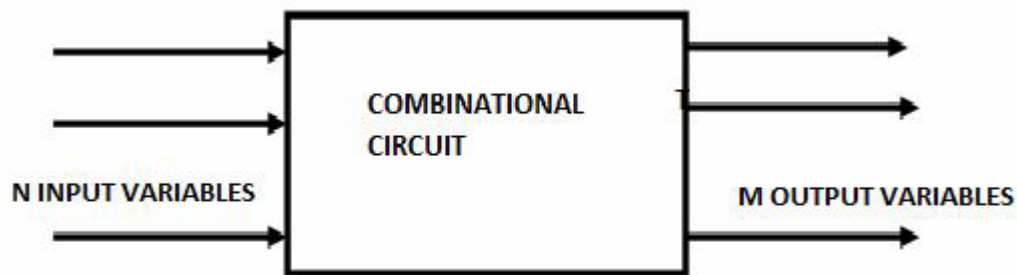
Arithmatic circuits:

- It is made of different arithmetic operators. There will be addition, substraction, division, modules and any other arithmetic operations.

Half-Adder

- Half-Adder is a part of combinational circuit.
- It is basically designed for arithmetic addition.
- It is most basic digital arithmetic circuit.
- Performs the addition of two binary digits.
- The input variables of a half-adder are called the *augend* and the *addend*.
- The output variables of a half-adder are called the *sum* and the *carry*.

| INPUT		OUTPUT	
X	Y	SUM X+Y	CARRY
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1
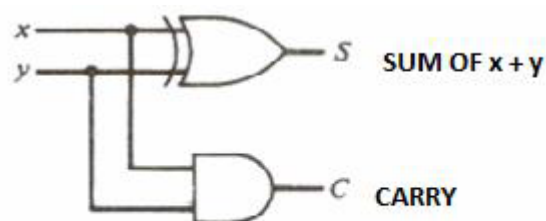
(b) Logic diagram

Full-Adder

- A full-adder performs the addition of three binary digits.
- Two half-adders can be combined to for a full-adder..
- Although a full adder has three inputs, it still only has two outputs since the largest number is 1+1+1 = 3, and 3 can be represented by two bits.

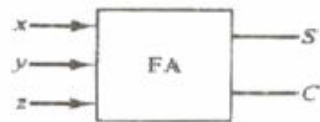## TRUTH TABLE OF FULL - ADDER

INPUT			OUTPUT	
X	Y	Z	SUM OF X+Y+Z	CARRY
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

LOGIC DIAGRAM OF FULL - ADDER
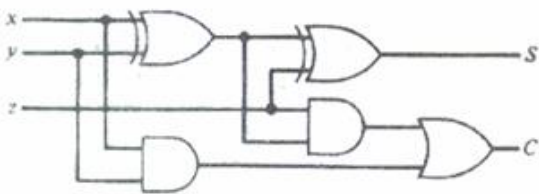
BLOCK DIAGRAM OF FULL ADDER

K-MAP FOR FULL ADDER

WHAT IS THE DIFFERENCE BETWEEN HALF ADDER AND FULL ADDER?

Half adder	Full adder
The most basic digital arithmetic circuit.	A full-adder performs the addition of three binary digits.
Performs the addition of two binary digits.	It is used for multi bit additions.
Output is sum of two signals.	Output is sum of three signals.
There are two input and two output terminal.	There are three input and two output terminal.
From full adder half adder cant not be built	Two full adder makes one full adder
On EX-OR gate and one AND gate are used.	Two EX-OR, two AND and one OR gate is used.

WHAT IS THE DIFFERENCE BETWEEN COMBINATIONAL CIRCUIT AND SEQUENCIAL CIRCUIT?

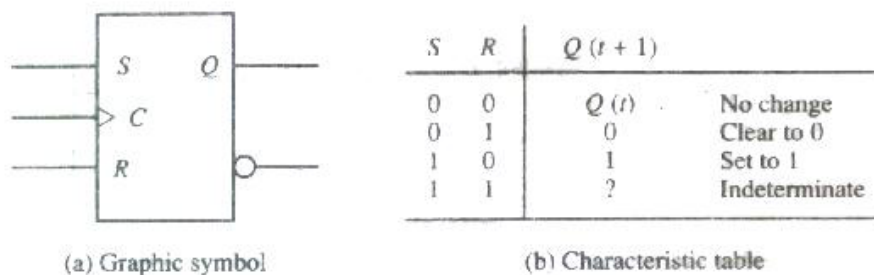| COMBINATIONAL CIRCUIT	SEQUENTIAL CIRCUIT
It is a digital logic circuit whose output depends on the present inputs.	It is a digital logic circuit whose output depends on the present inputs as well as previous inputs.
It can describe by the output values.	It can describe by the output values as well as state values.
It contains no memory element.	It contains at least one memory element.
It is easy to design and understand.	It is difficult to design and understand.
It is faster in speed.	It is slower in speed.
It is expensive in cost.	It is less expensive in cost.
Examples of combinational circuit are half adder and full adder.	Examples of sequential circuit are flip-flops like RS, Clocked RS, D and JK.
A combinational circuit is a connected arrangement of logic gates with a set of inputs and outputs.	However, if a circuit uses both gates and flip-flops, it is called a sequential circuit.
At any given time, the binary values of the outputs are a function of the binary values of the inputs.	Hence, a sequential circuit is an interconnection of flip-flops and gates.
The design of a combinational circuit starts from a verbal outline of the problem and ends in a logic circuit diagram.	If we think of a sequential circuit as some black box that, when provided with some external input, produces some external output

What is Flip-flops

- A Flip-flop is a binary cell capable of storing one bit of information.
- It has two outputs, one for the normal value and one for the complement value of the bit stored in it.
- Flip-flops are storage elements utilized in synchronous sequential circuits.
- Synchronous sequential circuits employ signals that effect storage elements only at discrete instances of time.
- A timing device called a clock pulse generator that produces a periodic train of clock pulses achieves synchronization.

- Values maintained in the storage elements can only change when the clock pulses.
- Hence, a flip-flop maintains a binary state until directed by a clock pulse to switch states.
- The difference in the types of flip flops is in the number of inputs and the manner in which the inputs affect the binary state.
- Flip-flops can be described by a *characteristic table* which permutates all possible inputs (just like a truth table).
- The characteristic table of a flip-flop describes all possible outputs (called the *next state*) at time $Q(t+1)$ over all possible inputs and the *present state* at time $Q(t)$.
- The most common types of flip flops are:
  - SR Flip-Flop
  - D Flip-Flop
  - JK Flip-Flop
  - T Flip-Flop

SR Flip-Flop

Figure SR Flip-Flop

S	R	$Q(t+1)$	
0	0	$Q(t)$	No change
0	1	0	Clear to 0
1	0	1	Set to 1
1	1	?	Indeterminate

(a) Graphic symbol   (b) Characteristic table

Inputs:
- S (for set)
- R (for reset)
- C (for clock)

Outputs:
- Q
- Q'

The operation of the SR flip-flop is as follow.

- If there is no signal at the clock input C, the output of the circuit cannot change irrespective of the values at inputs S and R.
- Only when the clock signals changes from 0 to 1 can the output be affected according to the values in inputs S and R
- If S =1 and R = 0 when C changes when C changes from 0 to 1 output Q is set to 1. If S = 0 and R =1 when C changes from 0 to 1.
- If both S and R are 0 during the clock transition, output does not change.
- When both S and R are equal to 1, the output is unpredictable and may go to either 0 or 1, depending on internal timing that occur within the circuit.

D Flip-Flop

D Flip-flop
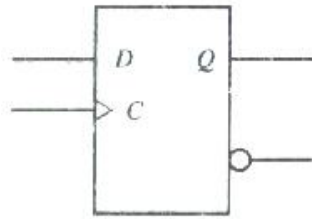
	D	Q (t + 1)	
(a) Graphic symbol	0	0	Clear to 0
	1	1	Set to 1

(a) Graphic symbol   (b) Characteristic table

Inputs:
- D (for data)
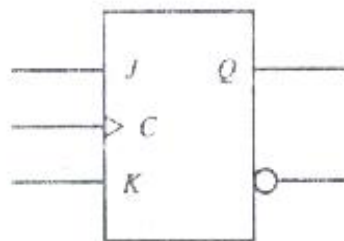- C (for clock)

Outputs:
- Q
- Q'

The operation of the D flip-flop is as follow.

- The D Flip-Flop can be converted from SR Flip-Flop by inserting an inverter between S and R and assigning the symbol D to the single input.
- The D input is sampled during the occurrence of a clock transition from 0 to 1.
- If D=1, the output of the flip-flop goes to the 1 state, but if D=0, the output of the flip-flop goes to the 0 state.
- The next state Q(t+1) is determined from the D input. The relationship can be expressed by a characteristic equation:

$$Q(t+1) = D$$

- D Flip-Flop has the advantage of having only one input (excluding), but the disadvantage that its characteristic table does not have a "no change" condition $Q(t+1) = Q(t)$.

JK Flip-Flop

Jk Flip-Flop

J	K	Q (t + 1)	
0	0	Q (t)	No change
0	1	0	Clear to 0
1	0	1	Set to 1
1	1	Q' (t)	Complement

(a) Graphic symbol   (b) Characteristic table
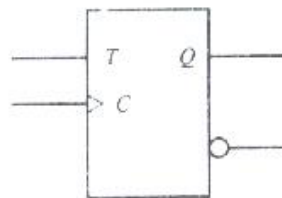
Inputs:
- J
- K
- C (for clock)

Outputs:
- Q
- Q'

The operation of the JK flip-flop is as follow.

- A JK Flip-Flop is a refinement of the SR flip-flop in that the indeterminate condition of the SR type is defined in the JK type.
- Inputs J and K behave like inputs S and R to set and clear the flip-flop, respectively.
- When inputs J and K are both equal to 1, a clock transition switches the outputs of the flip-flop to their complement state.
- Instead of the indeterminate condition of the SR flip-flop, the JK flip-flop has a complement condition $Q(t+1) = Q'(t)$ when both J and K are equal to 1.

T Flip-Flop

T Flip-Flop

T	$Q(t+1)$	
0	$Q(t)$	No change
1	$Q'(t)$	Complement

(a) Graphic symbol   (b) Characteristic table

Inputs:
- T (for toggle)
- C (for clock)

Outputs:
- Q
- Q'

The operation of the T flip-flop is as follow.

- Most flip-flops are edge-triggered flip-flops, which means that the transition occurs at a specific level of the clock pulse.
- A positive-edge transition occurs on the rising edge of the clock signal.
- A negative-edge transition occurs on the falling edge of the clock signal.
- Another type of flip-flop is called a master-slave flip-flop that is basically two flip-flops in series.
- Flip-flops can also include special input terminals for setting or clearing the flip-flop *asynchronously*. These inputs are usually called *preset* and *clear* and are useful for initialing the flip-flops before *clocked* operations are initiated.

Flip-Flop Excitation Tables

- During the design of sequential circuits, the required transition from present state to next state is known.
- What the designer needs to know is what input conditions must exist to implement the required transition.
- This requires the use of flip-flop excitation tables.

Excitation Tables

SR Flip-Flop Excitation Table			
Q(t)	Q(t+1)	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

JK Flip-Flop Excitation Table			
Q(t)	Q(t+1)	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

T Flip-Flop Excitation Table		
Q(t)	Q(t+1)	T
0	0	0
0	1	1
1	0	1
1	1	0

Sequential Circuits

- When a circuit contains just gates, it is called a *combinational* circuit. However, if a circuit uses both gates and flip-flops, it is called a *sequential* circuit. Hence, a sequential circuit is an interconnection of flip-flops and gates.
- If we think of a sequential circuit as some *black box* that, when provided with some *external* input, produces some *external* output, a typical sequential circuit would function as follows:
- The *external* inputs constitute *some* of the inputs to the combinational circuit. The *internal* outputs of the combinational circuit are the *internal* inputs to the flip-flops.
- The *internal* outputs of the flip-flops constitute the remaining inputs to the combinational circuit. The *external* outputs are some combination of the outputs from the combinational circuit and flip-flops. The behavior of a sequential circuit is determined from the inputs, the outputs, and the state of the flip-flops. Both the outputs and the next state are determined by the inputs and the present state.
- A state diagram can represent the information in a state table graphically, where states are represented by circles (vertices) and transitions on specific input is represented by the *labels* on the directed lines (edges) connecting the circles.

Design Procedure

- Formulate behavior of circuit using a state diagram.
- Determine # of flip-flops needed (equal to # bits in circles).
- Determine # inputs (specified on edges of diagram).
- Create state table, assigning letters to flip-flips, input, and output variables.*
- For each row, list the next state as specified by the state diagram.
- Select flip-flop type to be used in circuit.
- Extend state table into an excitation table by including columns for each input of each flip-flop.
- Using excitation table and present state-to-next state transitions, formulate input conditions for flip-flops.
- Construct truth table for combinational circuit using present-state and input columns of excitation table (for inputs) and flip-flop inputs (for outputs).
- Use map simplification of truth table to obtain flip-flop input equations.**
- Determine *external* outputs of sequential circuit (flip-flop outputs and potentially combinational circuit outputs).
- Draw logic diagram as follows:

- Draw flip-flops and label all their inputs and outputs.
- Draw combinational circuit from the Boolean expressions given by the flip-flop input equations.
- Connect outputs of flip-flops to inputs in the combinational circuit.
- Connect outputs of combinational circuit to flip-flop inputs.

For m flip-flops and n inputs, the state table will consist of m columns for the present state, n columns for the inputs, and m columns for the next state. The number of rows in the table will be up to $2m+n$, one row for each binary combination of present state and inputs.

** Each flip-flop input equation specifies a logic diagram whose output must be connected to one of the flip-flop inputs.

NAND and NOR Implementation

A sum-of-products expression can be implemented with NAND and NOR gates as shown in the figure 2.9

Figure 2.9 Logic Diagram with NAND and NOR gates

(a) With NAND gates   (b) With NOR gates
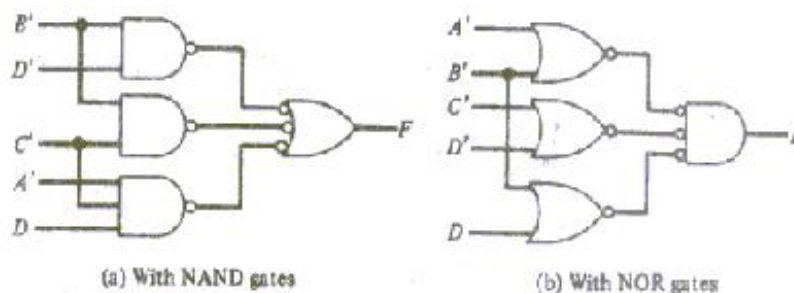
Don't Care Conditions
- In k-map each cell represents a minterm or maxterm and the 0's and 1's in k map represents the minterm that make the function equal to either 0 or 1.

- But in some occasion, it doesn't matter whether a function produces a 0 or 1 for a given minterm.
- When this condition occurs, an X is used in the map to represent the *don't care* condition.
- The minterm that may produce either 0 or 1 for function are said to be Don't Care and marked as x in map.
- This don't care condition are used to further simplify the Boolean expression.
- Don't care condition is the condition where any single square or map will appear as x n it is not necessary to write into Boolean expression.

Example
F(w,x,y,z)= $\sum$(0,1) + d(4,5,14)

| yz				
wx	00	01	11	10
00	1	1		
01	x	x		
11				x
10				

- So the ans is **W'Y'**

UNIT 2: DIGITAL COMPONENTS

What is Decoder?

- Discrete quantities of information are represented in digital computers with binary codes.
- A binary code of n bits is capable of representing up to 2n distinct elements of the coded information.
- A decoder is a combinational circuit that converts binary information from the n coded inputs to a maximum of 2n unique outputs.
- If the n-bit coded information has unused bit combinations, the decoder may have less than 2n outputs.
- The decoders presented in this section are called n-to-m-line decoders, where m <= 2n. Their purpose is to generate the 2n (or fewer) binary combinations of the n input variables. A decoder has n inputs and m outputs and is also referred to as an n x m decoder.
- The logic diagram of a 3-to-8-line decoder is shown bellow.
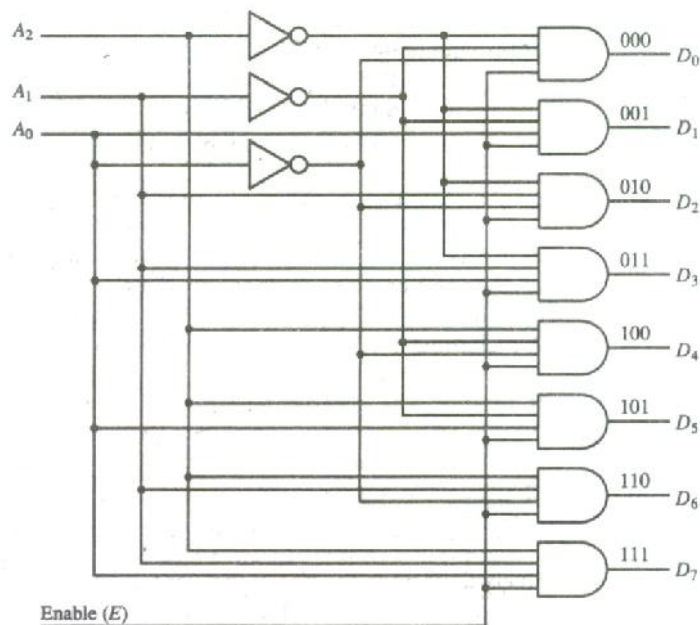
- The three data inputs. A0, A1, and A2, are decoded into eight outputs, each output representing one of the combinations of the three binary input variables.
- The three inverters provide the complement of the inputs, and each of the eight AND gates generates one of the binary combination.
- A particular application of this decoder is a binary-to-octal conversion. The input variables represent a binary number and the outputs represent the eight digits of the octal number system.
- However, a 3-to-8-line decoder can be used for decoding any 3-bit code to provide eight outputs, one for each combination of the binary code.
- Commercial decoders include one or more enable inputs to control the operation of the circuit. The decoder of the Figure has one enable input, E.
- The decoder is enabled when E is equal to 1 and disabled when E is equal to 0. The operation of the decoder can be clarified using the truth table listed in Table.
- When the enable input E is equal to 0, all the outputs are equal to 0 regardless of the values of the other three data inputs.

- The three x's in the table designate don't-care conditions. When the enable input is equal to 1, the decoder operates in a normal fashion.
- For each possible input combination, there are seven outputs that are equal to 0 and only one that is equal to 1.
- The output variable whose value is equal to 1 represents the octal number equivalent of the binary number that is available in the input data lines.

Truth Table for 3-to-8-line Decoder

Enable	Inputs			Outputs							
E	A_2	A_1	A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	x	x	x	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	0	0
1.	1	0	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0

- **NAND Gate Decoder**

- Some decoders are constructed with NAND instead of AND gates. Since a NAND gate produces the AND operation with an inverted output, it becomes more economical to generate the decoder outputs in their complement form.
- A 2-to-4-line decoder with an enable input constructed with NAND gates is shown in Figure.
- The circuit operates with complemented outputs and a complemented enable input E. The decoder is enabled when E is equal to 0. As indicated by the truth table, only one output is equal to 0 at any given time; the other three outputs are equal to 1.
- The output whose value is equal to 0 represents the equivalent binary number in inputs Ai and Ao.
- The circuit is disabled when E is equal to 1, regardless of the values of the other two inputs.
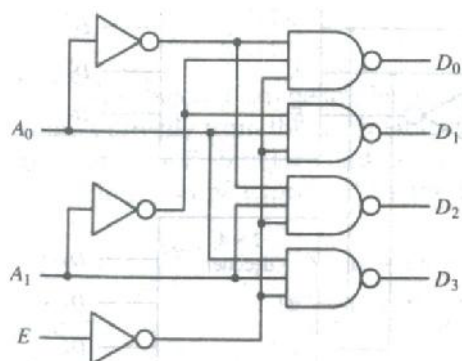
| E	A_1	A_0	D_0	D_1	D_2	D_3
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0
1	x	x	1	1	1	1

(a) Logic diagram   (b) Truth table

- **Decoder Expansion**

- A technique called decoder expansion can be utilized to construct larger decoders out of smaller ones.

- For example, two 2-to-4-line decoders can be combined to construct a 3-to-8-line decoder. Figure below shows 3-8-line decoder constructed with two 2x4 decoders.

3X8 decoder constructed with two 2X4 decoders
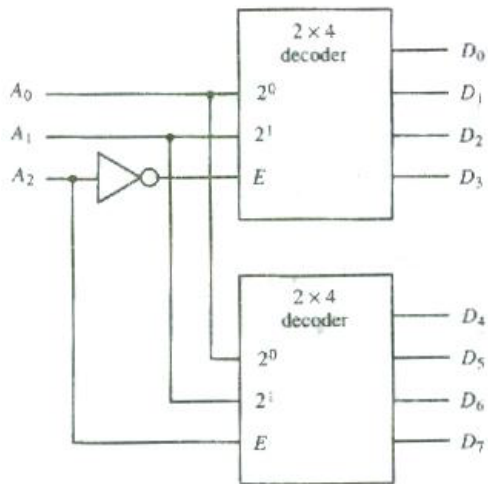
- The above given Figure shows how the decoders with enable inputs can be connected to form a larger decoder.
- As you can see that there are two 2-to-4-line decoders are combined to achieve a 3-to-8-line decoder.
- The two least significant bits of the input are connected to both decoders.
- The most significant bit is connected to the enable input of one decoder and through an inverter to the enable input of the other decoder.
- It is assumed that each decoder is enabled when its E input is equal to 1. When E is equal to 0, the decoder is disabled and all its outputs are in the 0 level. When $A_2 = 0$, the upper decoder is enabled and the lower is disabled.
- The lower decoder outputs become inactive with all outputs at 0. The outputs of the upper decoder generate outputs D_0 through D_3, depending on the values of A_1 and A_0 (while $A_2 = 0$).
- When $A_2 = 1$, the lower decoder is enabled and the upper is disabled. The lower decoder output generates the binary equivalent D_4, through D_7 since these binary numbers have a 1 in the A_2 position.

What is Encoder?

- An encoder is a digital circuit that performs the inverse operation of a decoder. An encoder has 2^n (or less) input lines and n output lines.
- The output lines generate the binary code corresponding to the input value. An example of an encoder is the octal-to-binary encoder, whose truth table is given below.

Inputs								outputs		
D7	D6	D5	D4	D3	D2	D1	D0	A2	A1	A0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

- The encoder can be implemented with OR gates whose inputs are determined directly from the truth table.
- Output A0 =1 if the input octal digit is 1 or 3 or 5 or 7. Similar conditions apply for other two outputs.

    These conditions can be expressed by the following Boolean functions :
    $$A0 = D1 + D3 + D5 + D7$$
    $$A1 = D2 + D4 + D6 + D7$$
    $$A2 = D4 + D5 + D6 + D7$$

    The encoder can be implemented with three OR gates.

Octal-to-binary Encoder
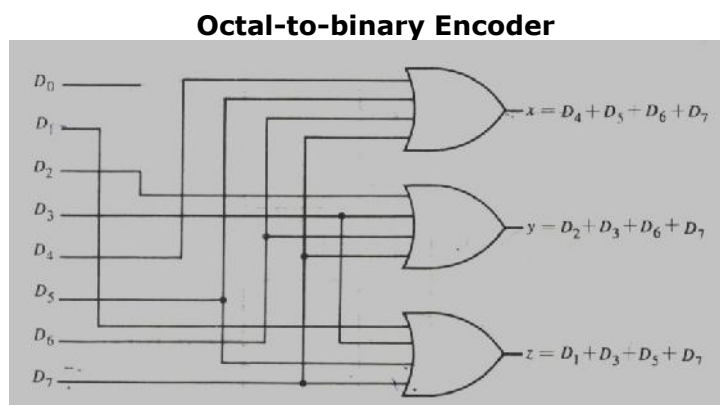
Write a note on Multiplexer

- A multiplexer is a combinational circuit that receives binary information from one of 2n input data lines and directs it to a single output line.
- The selection of a particular input data line for the output is determined by a set of selection inputs. A 2n-to-1 multiplexer has 2n input data lines and n input selection lines whose bit combinations determine which input data are selected for the output.
- The 4-to-1 line multiplexer has six inputs and one output. A truth table describing the circuit needs 64 rows since six input variables can have 26 binary combinations. This is

an extremely long table and will not be shown here. A more convenient way to describe the operation of multiplexers is by means of a function table.

- The function table for the multiplexer is shown in table.
- The table demonstrates the relationship between the four data inputs and the single output as a function of the selection inputs S_1 and S_0.

Function table for 4-to-1 line multiplexer

Select		Output
S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

4-to-1 line Multiplexer
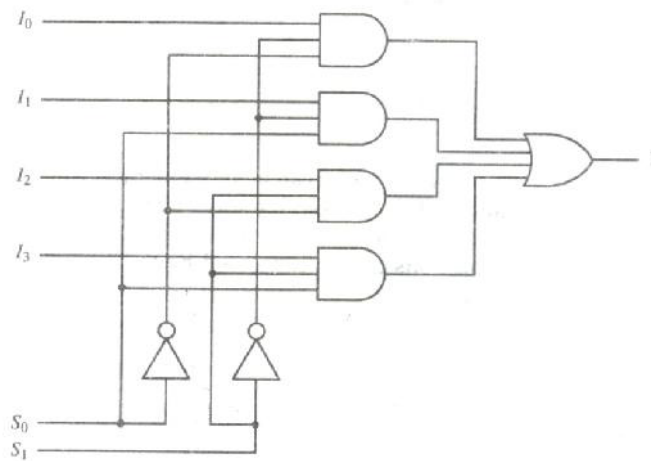
Write a note on DeMultiplexer

- A decoder with an enable input can function as a demultiplexer.
- A demultiplexer is a circuit that receives information on a single line and transmits this information on one of $2n$ possible output lines.
- The selection of a specific output line is controlled by the bit values of n selection lines. The decoder of figure a can function as a demultiplexer if the E line is taken as a data input line and lines A and B are taken as the selection lines.
- The single input variable E has a path to all four outputs, but the input information is directed to only one of the output lines, as specified by the binary value of the two selection lines A and B.
- For example, if the selection lines AB = 10; output D2 will be the same as the input value E, while all other outputs are maintained at 1.
- Because decoder and demultiplexer operations are obtained from the same circuit, a decoder with an enable input is referred to as a decoder / demultiplexer.
- It is the enable input that makes the circuit a demultiplexer.

Block Diagram for DeMultiplexer
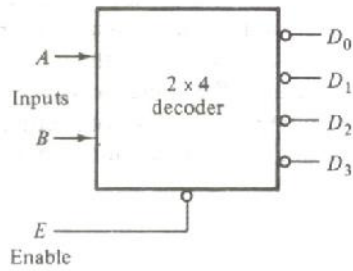
(a) Decoder with enable   (b) Demultiplexer
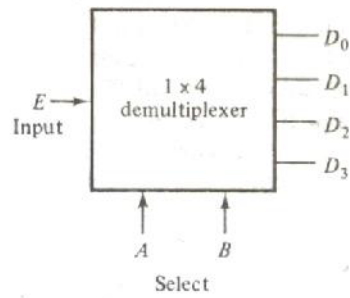
Truth Table Of Demultiplexer

Input					Output	
D	S0	S1	F0	F1	F2	F3
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

Explain MUX in detail.

- MUX is combinational circuit that is used to direct one out of 2n input data Lines to A single output line.
- It is also known as data selector because it selects one of many inputs And directs it to the output.
- The selection of particular input data line is controlled by as Set of selection inputs.
- Normally there are 2n input data lines and n input selection lines.
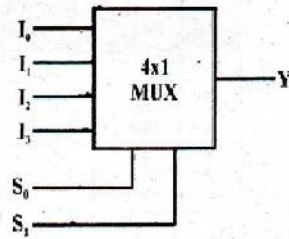
Fig. 3.4 : Block Diagram of 4 x 1 MUX
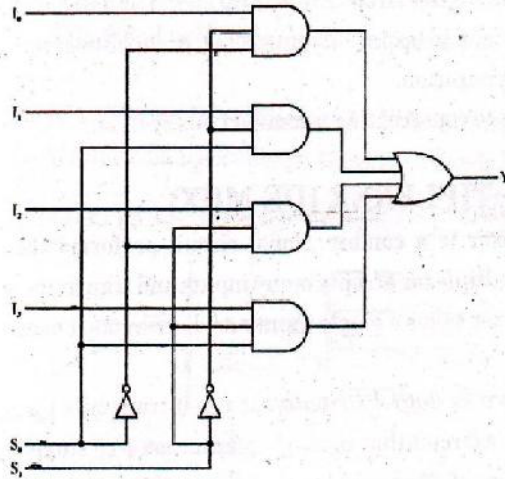
- The Block diagram of 4 to 1 line MUX is shown in fig. and logic circuit is displayed in fig.
- Each of the 4 data input I0 through I3 is applied to one input of AND gate.
- The two selection inputs s1 and s0 are decoded To select a particular AND gate.
- The output of the AND gates are applied to a single OR gate to provide the single output.

simplification of Multiplexer.

- o It is used to connecting two or more sources to a single destination among computer units.
- o It is used in digital circuits to control signal and in data routing.
- o It is also useful in operation sequencing.
- o It is useful to constructing a common bus system.

What are Registers? Explain the types of registers.

- A *register* is a group of flip-flops capable of storing one bit of information.
- An *n*-bit register has a group of *n* flip-flops and is capable of storing any binary information of *n* bits.
- In addition to flip-flops, registers can have combinational gates that perform certain data-processing tasks. The gates control how and when new information is transferred into the registers.
- The transfer of new information into a register is referred to as a *register load*. If the loading occurs simultaneously at a common clock pulse transition, we say that the load is done in *parallel*.
- The *load input* in a register determines the action to be taken with each clock pulse.

- When the load input is 1, the data from the input lines is transferred into the register's flip-flops. When the load input is 0, the data inputs are *inhibited* and the flip-flop maintains its present state.

A 4-bit register is shown in the figure below. A clock transition applied to the C inputs of the register will load all four inputs I_0 through I_3 in parallel.

Figure 5.1 4-bit register
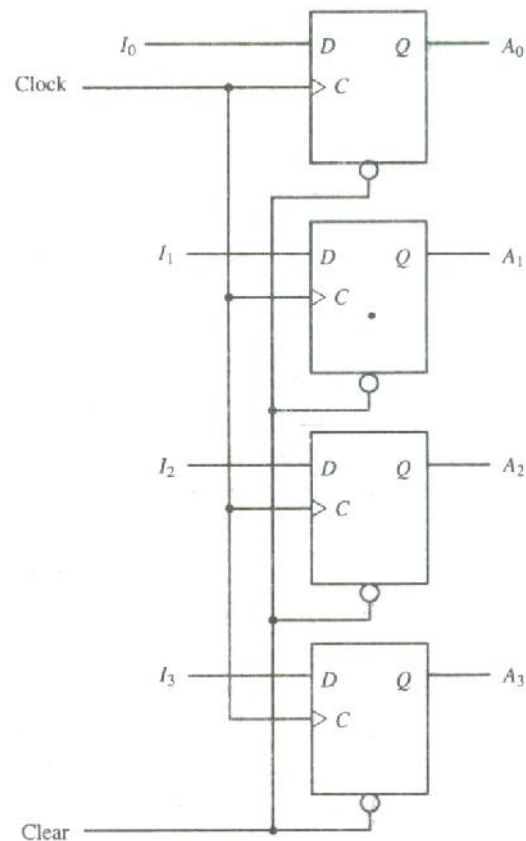
Shift Registers

- A *register* capable of shifting its binary information in one or both directions is called a *shift register*.
- Shift registers are constructed by connecting flip-flops in *cascade*, where the output of one flip-flop is connected to the input of the next flip-flop.
- All flip-flops receive common clock pulses that initiate the shift from one stage to the next.
- A *serial input* shift register has a single external input (called the serial input) entering an outermost flip-flop. Each remaining flip-flop uses the output of the previous flip-flop as its input, with the last flip-flop producing the external output (called the serial output).
- A register capable of shifting in one direction is called a *unidirectional* shift register.
- A register that can shift in both directions is called a *bi-directional* shift register.
- The most general shift register has the following capabilities:
  - An input for clock pulses to synchronize all operations.
  - A shift-right operation and a serial input line associated with the shift-right.
  - A shift-left operation and a serial input line associated with the shift-left.
  - A parallel load operation and *n* input lines associated with the parallel transfer.

- N parallel output lines.
- A control state that leaves the information in the register unchanged even though clock pulses are applied continuously.
- A mode control to determine which type of register operation to perform.

The simplest possible shift register is one that uses only flip-flops, as shown in the figure below.

4-bit shift register
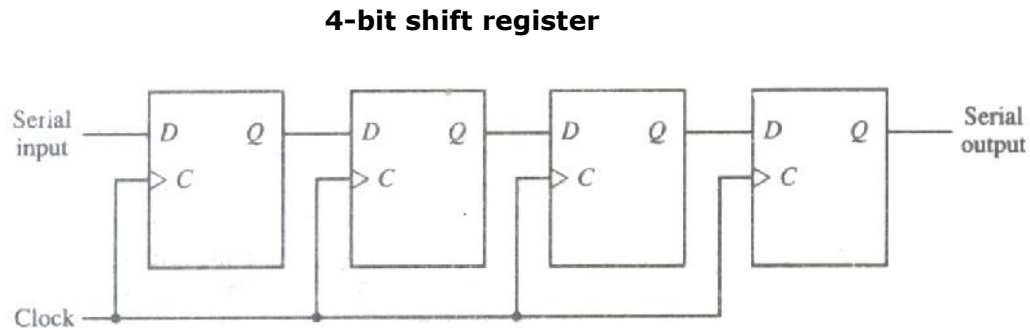
Bidirectional Shift Register with parallel load

- A register that can shift in both directions is called a bi-directional shift register.
- A 4-bit bidirectional shift register with parallel load is shown in figure below. Each stage consists of a D flip-flop and a 4X1 MUX.
- The two selection inputs S_1 and S_0 select one of the MUX data inputs for the D flip-flop. The selection lines control the mode of operation of the register according to the function table shown in table below.
- When the mode control $S_1S_0 = 00$, data input 0 of each MUX is selected.
- This condition forms a path from the output of each flip-flop into the input of the same flip-flop.
- The next clock transition refers into each flip-flop the binary value it held previously, and no change of state occurs. When $S_1S_0 = 01$, the terminal marked 1 in each MUX has a path to the D input of the corresponding flip-flop.
- This causes a shift-right operation, with the serial input data transferred into flip-flop A_0 and the content of each flip-flop A_{i-1} transferred into flip-flop A_i for i=1,2,3. When $S_1S_0 = 10$ a shift-left operations results, with the other serial input data going into flip-flop A_3 and the content of flip-flop A_{i+1} transferred into flip-flop A_i for I=0,1,2. When $S_1S_0 = 11$, the binary information from each input I_0 through I_3 is transferred into the corresponding flip-flop, resulting in a parallel load operation.
- In the diagram, the shift-right operation shifts the contents of the register in the down direction while the shift left operation causes the contents of the register to shift in the upward direction.

Bidirectional Shift register with parallel load
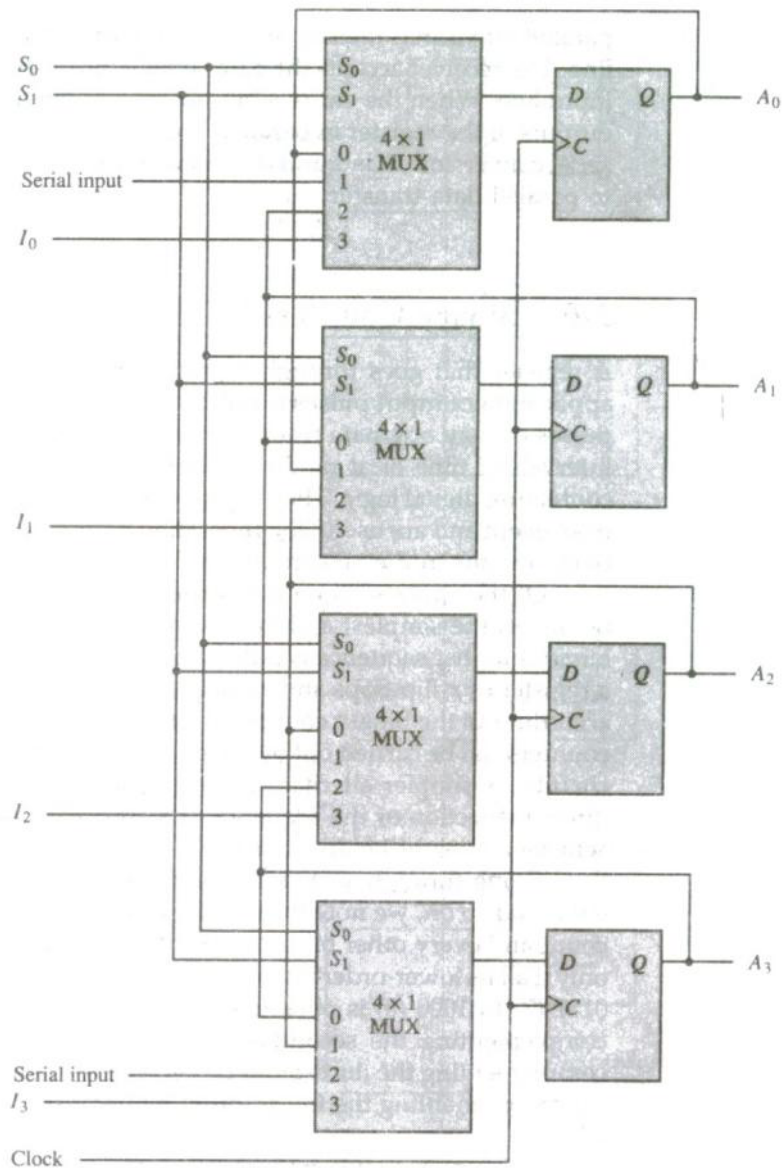
Application of Shift Registers

- Shift registers are often used to interface digital systems situated remotely from each other. For example, suppose that it is necessary to transmit an n-bit quantity between two points.
- If the distance between the source and the destination is too far, it will be expensive to use n lines to transmit the n bits in parallel.
- It may be more economical to use a single line and transmit the information serially one bit at a time.
- The transmitter loads the n-bit data in parallel into a shift register and then transmits the data from the serial output line.
- The receiver accepts the data serially into a shift register through its serial input line. When the entire n bits are accumulated they can be taken from the outputs of the register in parallel.
- Thus the transmitter performs a parallel-to-serial conversion of data and the receiver converts the incoming serial data back to parallel data transfer.

UNIT-3

Instruction Cycle

A program residing in the memory unit of the computer consists of a sequence of instructions. The program is executed in the computer by going through a cycle for each instruction. Each instruction cycle in turn is subdivided into a sequence of sub cycles or phases.

In the basic computer each instruction cycle consists of the following phases:

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.

Upon the completion of step 4, the control goes back to step 1 to fetch, decode, and execute the next instruction. This process continues indefinitely unless a HALT instruction is encountered.

Fetch phase

- Initially, the program counter PC is loaded with the address of the first instruction in the program.
- The sequence counter SC is cleared to 0, providing a decoded timing signal To.
- After each clock pulse, SC is incremented by one, so that the timing signals go through a sequence T0, T1, T2, and so on.
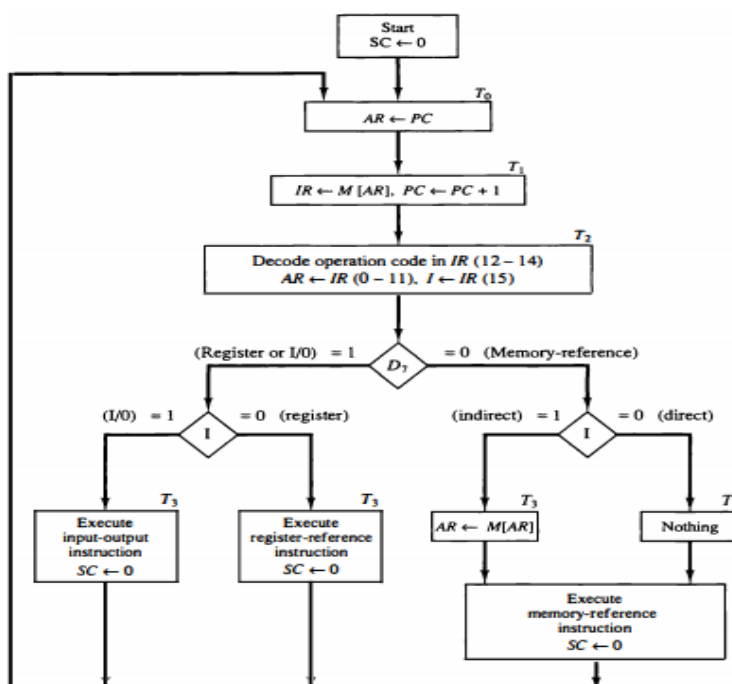- During T_0 we have to transfer the address from PC to AR.

Figure Flowchart for instruction cycle (initial configuration).

T1: IR <-M[AR], PC <- PC + 1

In timing signal T_1 the instruction read from memory is placed in IR,At the same time, PC is incremented by One to prepare it for the address of the next instruction in the program.

Decodephase:

T_2: D_0………….D_7 ⟵ Decode IR(12-14)

AR ⟵ IR(0-11),I ⟵ IR(15)

The operation code in IR is decoded the indirect bit is transferred to flipflop I and address part of the instruction is transferred to AR.

Decisionphase:

During time T_3,the control unit determines the type of instruction that was just read from memory.

D_7 → Decision making

If 0 → then memory reference instruction

If 1 → the I/o or register reference instruction

If D_7=0 then opcode code have any value from 000 to 110

D_7=0 memory reference instruction

If D_7=0 and I=1

memory reference instruction with an indirect address. It is then necessary to read the effective address from memory

$$AR \leftarrow M[AR]$$

If D_7=0 and I=0

 not necessary to do anything since effective address is already in AR.

If D_7=0 and I=0 →Register reference instruction

If D_7=1 and I=1→Input and output instruction

$$D_7' I T_3: \quad AR \leftarrow M[AR]$$
$$D_7' I' T_3: \quad Nothing$$
$$D_7 I' T_3: \quad \text{Execute a register-reference instruction}$$
$$D_7 I T_3: \quad \text{Execute an input–output instruction}$$

2

Memory reference instruction

When $D_7=0$ and $I=0$

i.e D_0 to D_6 are opcode

$D_0 \rightarrow$ AND to AC

There is no direct path from bus into accumulator, first the logic receive info from the data register and then transfer. first the contents will be stored in data register and then AND operation will be performed between the bits of the accumulator and that particular data.

$$D0T4: DR \leftarrow M[AR]$$

$$D0T5: AC \leftarrow AC \wedge DR, SC \leftarrow 0$$

$D_1 \rightarrow$ ADD to AC

This instruction adds the content of the memory word specified by the effective address to the value of AC. The sum is transferred into AC and the output carry C,., is transferred to the E (extended accumulator) flip-flop. The microoperations

$$D_1T_4: \quad DR \leftarrow M[AR]$$
$$D_1T_5: \quad AC \leftarrow AC + DR, \quad E \leftarrow C_{out}, \quad SC \leftarrow 0$$

$D_2 \rightarrow$ LDA: Load to AC

This instruction transfers the memory word specified by the effective address to AC. The rnicrooperations needed to execute this instruction are

$$D_2T_4: \quad DR \leftarrow M[AR]$$
$$D_2T_5: \quad AC \leftarrow DR, \quad SC \leftarrow 0$$

Looking back at the bus system shown in Fig. 5-4 we note that there is no direct path from the bus into AC. The adder and logic circuit receive information from DR which can be transferred into AC. Therefore, it is necessary to read the memory word into DR first and then transfer the content of DR into AC. The reason for not connecting the bus to the inputs of AC is the delay encountered in the adder and logic circuit. It is assumed that the time it takes to read from memory and transfer the word through the bus as well as the adder and logic circuit is more than the time of one clock cycle. By not connecting the bus to the inputs of AC we can maintain one clock cycle per microoperation.

$D_3 \rightarrow$ STA: Store AC

This instruction stores the content of AC into the memory word specified by the effective address. Since the output of AC is applied to the bus and the data input of memory is connected to the bus, we can execute this instruction with one microoperation:

$$D_3T_4: \quad M[AR] \leftarrow AC, \quad SC \leftarrow 0$$

D4→BUN: Branch Unconditionally

This instruction transfers the program to the instruction specified by the effective address. Remember that PC holds the address of the instruction to be read from memory in the next instruction cycle. PC is incremented at time T1 to prepare it for the address of the next instruction in the program sequence. The BUN instruction allows the programmer to specify an instruction out of sequence and we say that the program branches (or jumps) unconditionally. The instruction is executed with one microoperation:

$$D_4T_4: \quad PC \leftarrow AR, \quad SC \leftarrow 0$$

The effective address from AR is transferred through the common bus to PC . Resetting SC to 0 transfers control to T0• The next instruction is then fetched and executed from the memory address given by the new value in PC.

D5→BSA: Branch and Save Return Address

This instruction is useful for branching to a portion of the program called a subroutine or procedure. When executed, the BSA instruction stores the address of the next instruction in sequence (which is available in PC) into a memory location specified by the effective address. The effective address plus one is then transferred to PC to serve as the address of the first instruction in the subroutine. This operation was specified in Table 5-4 with the following register transfer:

$$M[AR] \leftarrow PC, \quad PC \leftarrow AR + 1$$

A numerical example that demonstrates how this instruction is used with a subroutine is shown in Fig. 5-10. The BSA instruction is assumed to be in memory at address 20. The I bit is 0 and the address part of the instruction has the binary equivalent of 135. After the fetch and decode phases, PC contains 21, which is the address of the next instruction in the program (referred to as the return address). AR holds the effective address 135. This is shown in part (a) of the figure. The BSA instruction performs the following numerical operation:

$$M[135] \leftarrow 21, \quad PC \leftarrow 135 + 1 = 136$$

The result of this operation is shown in part (b) of the figure. The return address 21 is stored in memory location 135 and control continues with the subroutine program starting from address 136. The return to the original program (at address 21) is accomplished by means of an indirect BUN instruction placed at the end of the subroutine. When this instruction is executed, control goes to the indirect phase to read the effective address at location 135, where it finds the previously saved address 21. When the BUN instruction is executed, the effective address 21 is transferred to PC . The next instruction cycle finds PC with the value 21, so control continues to execute the

4

instruction at the return address. The BSA instruction performs the function usually referred to as a subroutine call. The indirect BUN instruction at the end of the subroutine performs the function referred to as a subroutine return. In most commercial computers, the return address associated with a subroutine is stored in either a processor

Figure 5-10   Example of BSA instruction execution.

(a) Memory, *PC*, and *AR* at time T_4

(b) Memory and *PC* after execution
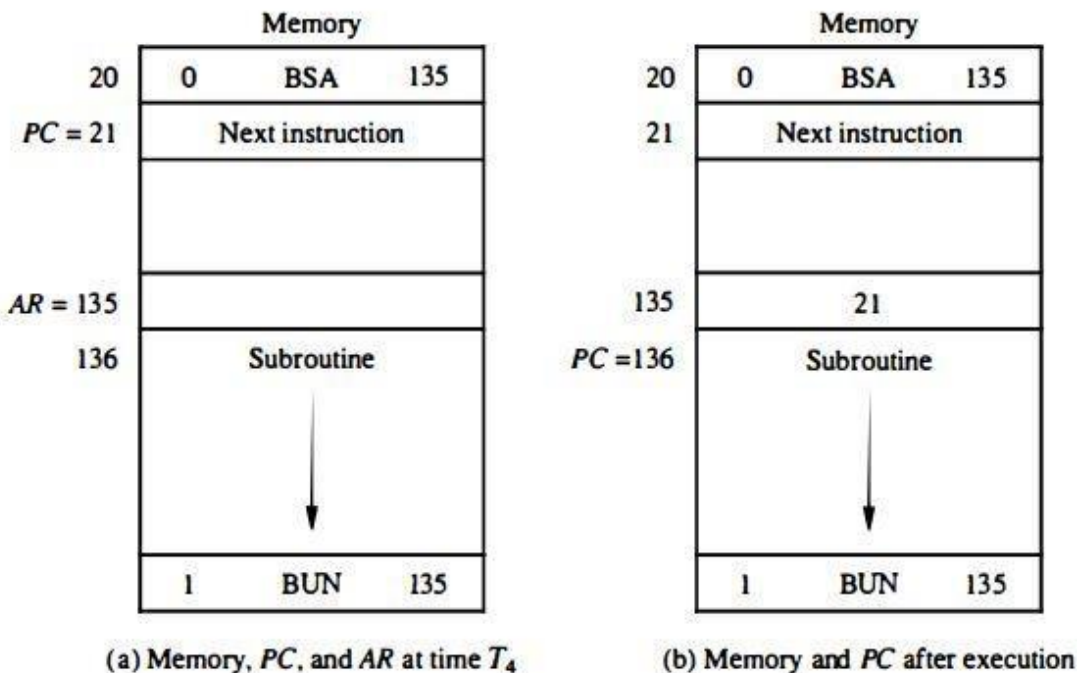
register or in a portion of memory called a stack. This is discussed in more detail in Sec. 8-7. It is not possible to perform the operation of the BSA instruction in one clock cycle when we use the bus system of the basic computer. To use the memory and the bus properly, the BSA instruction must be executed With a sequence of two microoperations:

$$D_5T_4: \quad M[AR] \leftarrow PC, \quad AR \leftarrow AR + 1$$
$$D_5T_5: \quad PC \leftarrow AR, \quad SC \leftarrow 0$$

Timing signal T4 initiates a memory write operation, places the content of PC onto the bus, and enables the INR input of AR . The memory write operation is completed and AR is incremented by the time the next clock transition occurs. The bus is used at T5 to transfer the content of AR to PC .

ISZ: Increment and Skip if Zero
This instruction increments the word specified by the effective address, and if the incremented value is equal to 0, PC is incremented by 1. The programmer usually stores a negative number (in 2's complement) in the memory word. As this negative number is repeatedly incremented by one, it eventually reaches the value of zero. At that time PC is incremented by one in order to skip the next instruction in the program. Since it is not possible to increment a word inside the memory, it is necessary to read the word into DR, increment DR, and store the word back into memory. This is done with the following sequence of microoperations:

5

D_6T_4: $DR \leftarrow M[AR]$
D_6T_5: $DR \leftarrow DR + 1$
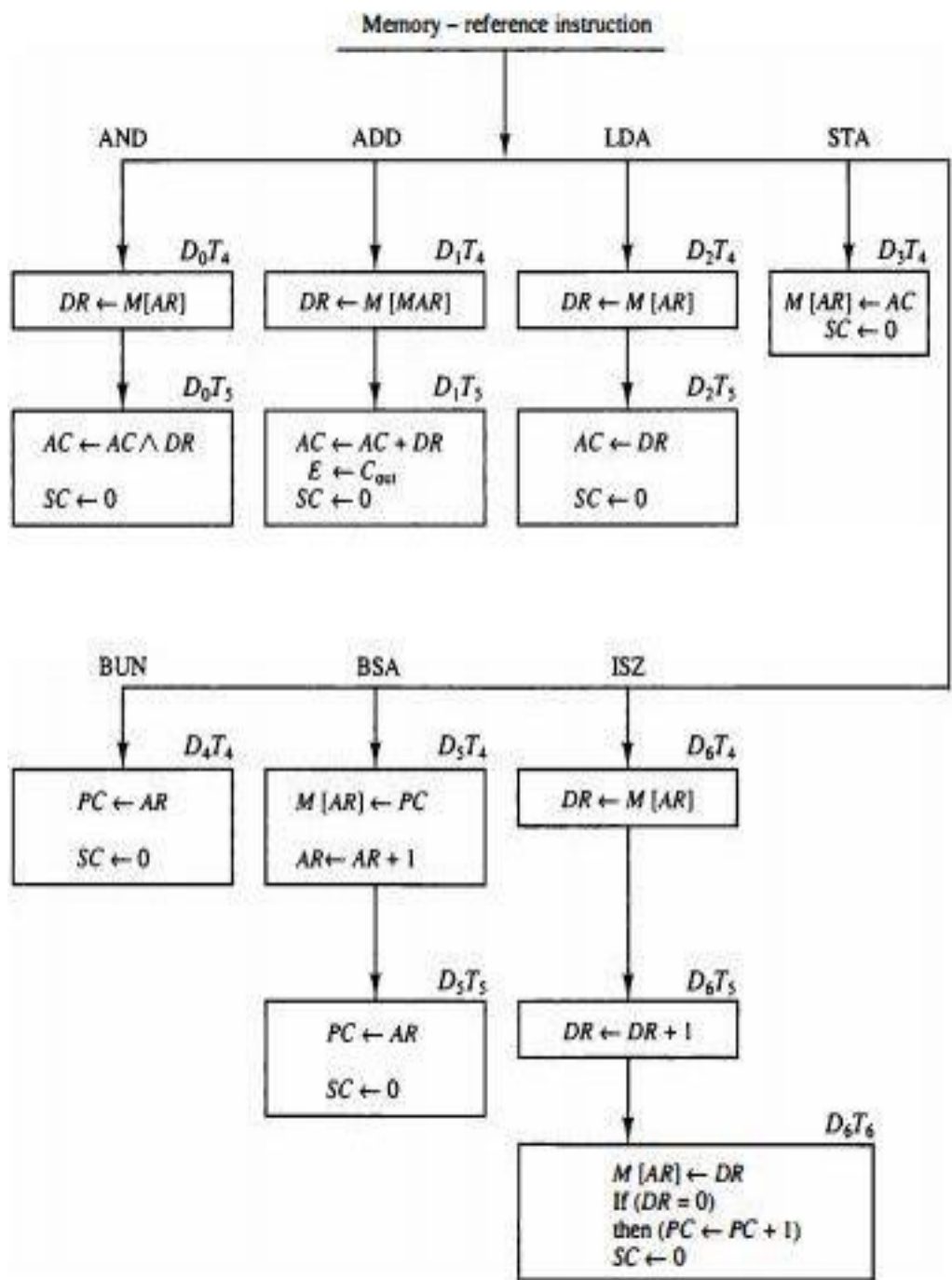D_6T_6: $M[AR] \leftarrow DR$,   if $(DR = 0)$ then $(PC \leftarrow PC + 1)$, $SC \leftarrow 0$

Memory – reference instruction

AND	ADD	LDA	STA
D_0T_4	D_1T_4	D_2T_4	D_3T_4
$DR \leftarrow M[AR]$	$DR \leftarrow M[MAR]$	$DR \leftarrow M[AR]$	$M[AR] \leftarrow AC$ $SC \leftarrow 0$
D_0T_5	D_1T_5	D_2T_5	
$AC \leftarrow AC \wedge DR$ $SC \leftarrow 0$	$AC \leftarrow AC + DR$ $E \leftarrow C_{out}$ $SC \leftarrow 0$	$AC \leftarrow DR$ $SC \leftarrow 0$	

BUN	BSA	ISZ
D_4T_4	D_5T_4	D_6T_4
$PC \leftarrow AR$ $SC \leftarrow 0$	$M[AR] \leftarrow PC$ $AR \leftarrow AR + 1$	$DR \leftarrow M[AR]$
	D_5T_5	D_6T_5
	$PC \leftarrow AR$ $SC \leftarrow 0$	$DR \leftarrow DR + 1$
		D_6T_6
		$M[AR] \leftarrow DR$ If $(DR = 0)$ then $(PC \leftarrow PC + 1)$ $SC \leftarrow 0$

Figure 5-11 Flowchart for memory-reference instructions.

6

Input-Output and Interrupt

Input-Output Configuration:
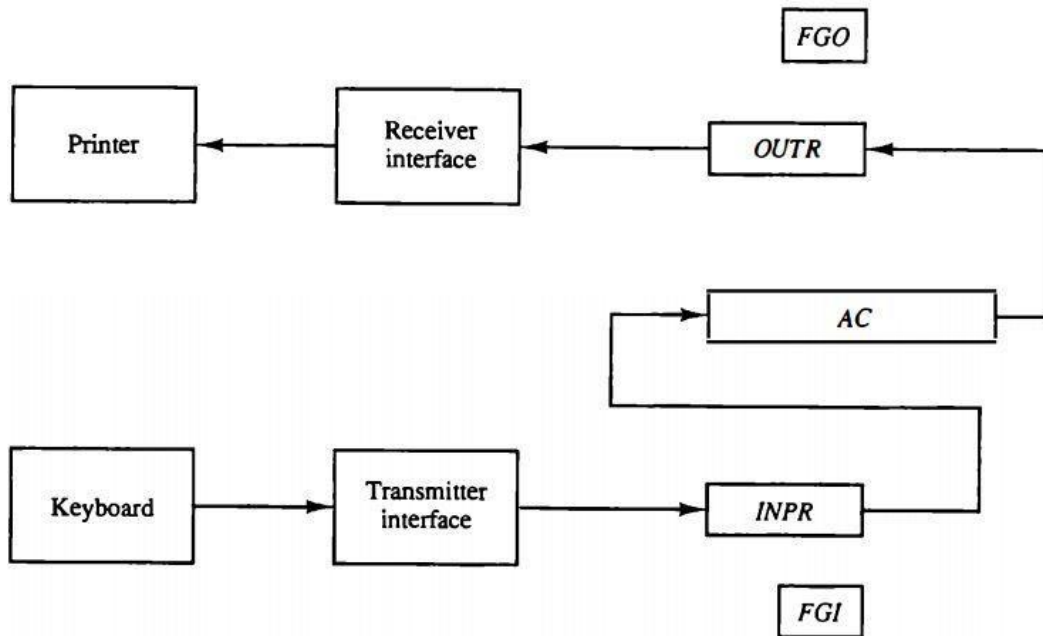
How input and output devices communicate, with each other using certain interfaces, flipflops and registers.

Printer , keyboard→ input-output terminals

Receiver interface, transmitter interface→ serial common interface

OUTR, INPR → COMPUTER REGISTERS

FGO, FGI→ FLIPFLOPS

The INPR and OUTR are eight bits.

The 1bit Input flag FGI is a control flip-flop.The flag bit is set to 1 when new information is available in the input device and is cleared to 0 when the information is accepted by the computer. If flag set to 1 the computer checks the flag bit i.e information from AC to OUTR and FGO is cleared to 0.

Input output instruction have an operation code 1111

$D_7=1$ and $I=1$

These instructions are executed with the clock transition T_3

$D_7=1$ I $T_3=P$ (common to all input-output instruction)

TABLE 5-5 Input-Output Instructions

$D_7 IT_3 = p$ (common to all input–output instructions)
$IR(i) = B_i$ [bit in $IR(6–11)$ that specifies the instruction]

	p:	$SC \leftarrow 0$	Clear SC
INP	pB_{11}:	$AC(0–7) \leftarrow INPR, \quad FGI \leftarrow 0$	Input character
OUT	pB_{10}:	$OUTR \leftarrow AC(0–7), \quad FGO \leftarrow 0$	Output character
SKI	pB_9:	If $(FGI = 1)$ then $(PC \leftarrow PC + 1)$	Skip on input flag
SKO	pB_8:	If $(FGO = 1)$ then $(PC \leftarrow PC + 1)$	Skip on output flag
ION	pB_7:	$IEN \leftarrow 1$	Interrupt enable on
IOF	pB_6:	$IEN \leftarrow 0$	Interrupt enable off

Program Interrupt

During an instruction cycle how an interrupt is handled.
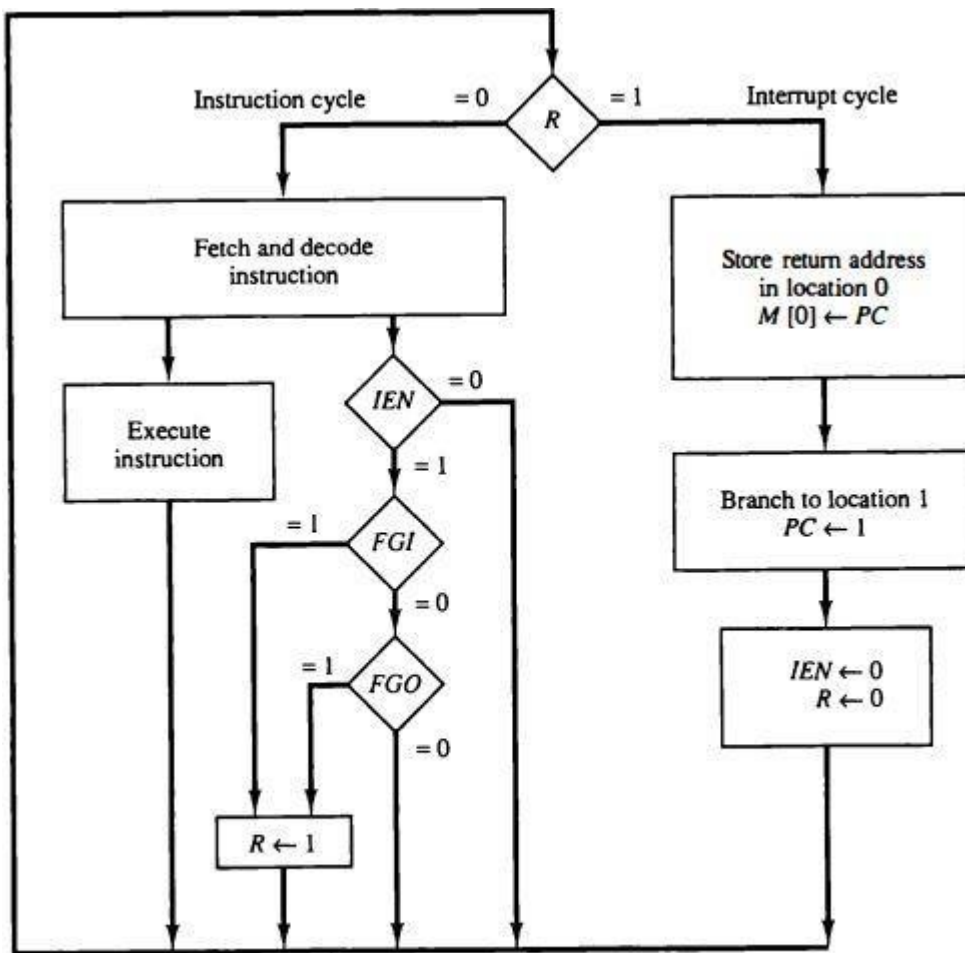
Figure 5-13 Flowchart for interrupt cycle.

8

IEN-Interrupt enable flipflop

R-Interrupt flipflop

FGO-1 bit output flag

FGI-1 bit input flag

- The value of R determines whether it enters the interrupt cycle or instruction cycle.
- If 0 the normal instruction cycle executed i.e fetch and decode instruction.

During execution phase there is another interrupt IEN-interrupt enable flipflop.

If IEN=0 means no interrupt and normal execution takes place.

If IEN=1 then it is checked for the input or output interrupt.

If FGO and FG1 both are 0 then no interrupts and normal execution takes place.

If FGI and FGO are one then R=1 has to be executed.

If R=1 then interrupt cycle is executed. first store return address then branch to location and then both value IEN$\leftarrow$0,R$\leftarrow$0 set to zero

Central processing unit

Introduction

The part of the computer that performs the bulk of data-processing operations is called the central processing unit and is referred to as the CPU.

The CPU is made up of three major parts, as shown in Fig. 8-I
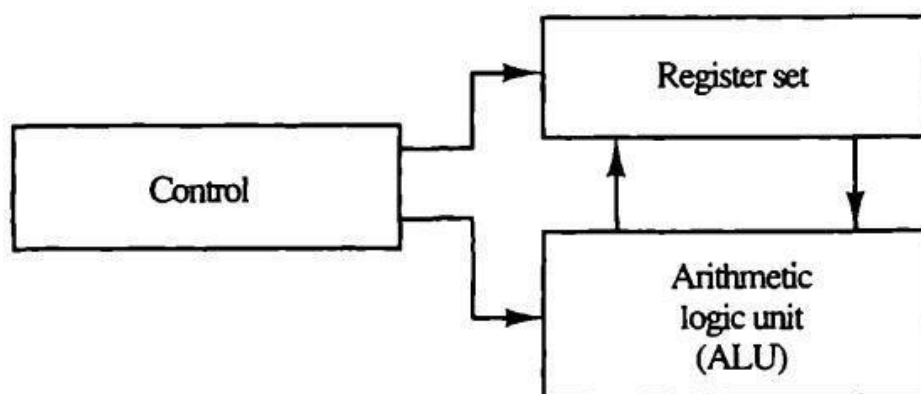
Figure 8-1   Major components of CPU.

- The register set stores intermediate data used during the execution of the instructions. The arithmetic logic unit (ALU) performs the required microoperations for executing the instructions.
- The control unit supervises the transfer of information among the registers and instructs the ALU as to which operation to perform.

General Register Organization

- It is shown that memory locations are needed for storing pointers, counters, return addresses, temporary results, and partial products during multiplication.
- Having to refer to memory locations for such applications is time consuming because memory access is the most time-consuming operation in a computer.
- It is more convenient and more efficient to store these intermediate values in processor registers.
- When a large number of registers are included in the CPU, it is most efficient to connect them through a common bus system.
- The registers communicate with each other not only for direct data transfers, but also while performing various microoperations. Hence it is necessary to provide a common unit that can perform all the arithmetic, logic, and shift microoperations in the processor.

A bus organization for seven CPU registers is shown in Fig. 8-2. The output of each register is connected to two multiplexers (MUX) to form the two buses A and B.

- The selection lines in each multiplexer select one register or the input data for the particular bus.
- The A and B buses form the inputs to a common arithmetic logic unit (ALU).
- The operation selected in the ALU determines the arithmetic or logic microoperation that is to be performed.
- The result of the microoperation is available for output data and also goes into the inputs of all the registers.
- The register that receives the information from the output bus is selected by a decoder.
- The decoder activates one of the register load inputs, thus providing a transfer path between the data in the output bus and the inputs of the selected destination register.
- The control unit that operates the CPU bus system directs the information flow through the registers and ALU by selecting the various components in the system.

For example, to perform the operation the control must provide binary selection variables to the following selector inputs:

$$R1 \leftarrow R2 + R3$$

1. MUX A selector (SELA): to place the content of R2 into bus A.

2. MUX B selector (SELB): to place the content o f R 3 into bus B.

3. ALU operation selector (OPR): to provide the arithmetic addition A+ B.

4. Decoder destination selector (SELD): to transfer the content of the output bus into R1.

tim
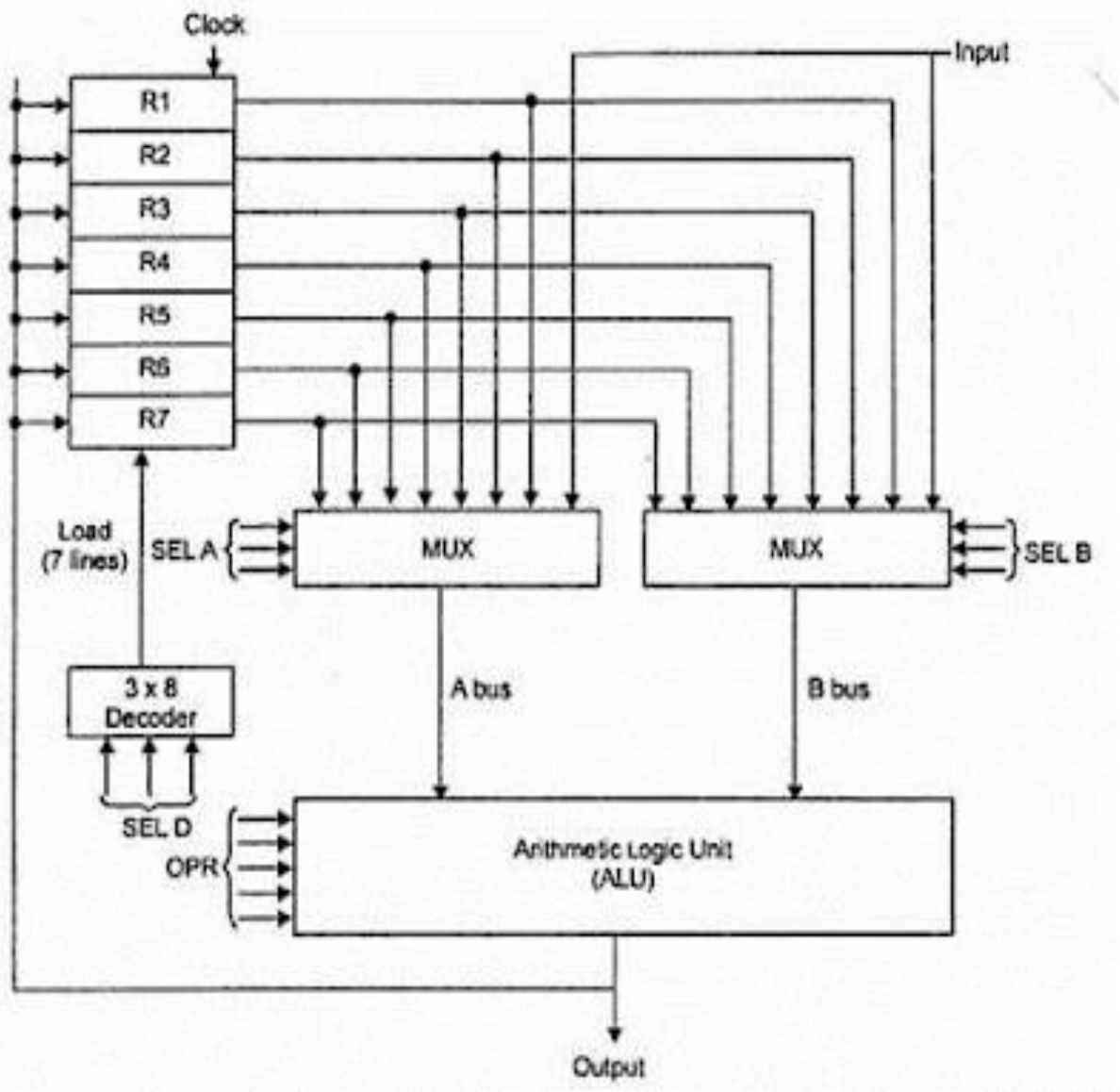
Control Word

There are 14 binary selection inputs in the unit, and their combined value specifies a control word. The 14-bit control word is defined in Fig. 8-2(b).

(b) Control word

It consists of four fields. Three fields contain three bits each, and one field has five bits. The three bits of SELA select a source register for the A input of the ALU. The three bits of SELB select a register for the B input of the ALU. The three bits of SELD select a destination register using the decoder and its seven load outputs. The five bits of OPR select one of the operations in the ALU. The 14-bit control word when applied to the selection inputs specify a particular microoperation.

The encoding of the register selections is specified in Table 8-1.

TABLE 8-1 Encoding of Register Selection Fields

Binary Code	SELA	SELB	SELD
000	Input	Input	None
001	R1	R1	R1
010	R2	R2	R2
011	R3	R3	R3
100	R4	R4	R4
101	R5	R5	R5
110	R6	R6	R6
111	R7	R7	R7

12

The encoding of the ALU operations for the CPU is taken from Sec. 4-7 and is specified in Table 8-2. The OPR field has five bits and each operation is designated with a symbolic name.

TABLE 8-2 Encoding of ALU Operations

OPR Select	Operation	Symbol
00000	Transfer A	TSFA
00001	Increment A	INCA
00010	Add $A + B$	ADD
00101	Subtract $A - B$	SUB
00110	Decrement A	DECA
01000	AND A and B	AND
01010	OR A and B	OR
01100	XOR A and B	XOR
01110	Complement A	COMA
10000	Shift right A	SHRA
11000	Shift left A	SHLA

Examples of Microoperations

A control word of 14 bits is needed to specify a micro operation in the CPU. The control word for a given microoperation can be derived from the selection variables.

For example, the subtract micro operation given by the statement R1 <-R2 - R3 specifies R2 for the A input of the ALU, R3 for the B input of the ALU, R1 for the destination register, and an ALU operation to subtract A - B.

Thus the control word is specified by the four fields and the corresponding binary value for each field is obtained from the encoding listed in Tables 8-1 and 8-2.

The binary control word for the subtract micro operation is 010 011 001 00101 and is obtained as follows:

Field:	SELA	SELB	SELD	OPR
Symbol:	R2	R3	R1	SUB
Control word:	010	011	001	00101

13

Stack Organization

A useful feature that is included in the CPU of most computers is a stack or last-in, first-out (LIFO) list. A stack is a storage device that stores information in such a manner that the item stored last is the first item retrieved.

The stack in digital computers is essentially a memory unit with an address register that can count only (after an initial value is loaded into it).

The register that holds the address for the stack is called a stack pointer (SP) because its value always points at the top item in the stack.

The two operations of a stack are the insertion and deletion of items. However, nothing is pushed or popped in a computer stack. These operations are simulated by incrementing or decrementing the stack pointer register.

Register Stack A stack can be placed in a portion of a large memory or it can be organized as a collection of a finite number of memory words or registers. Figure 3 shows the organization of a 64-word register stack. The stack pointer register SP contains a binary number whose value is equal to the address of the word that is currently on top of the stack. Three items are placed in the stack: A, B, and C, in that order. Item C is on top of the stack so that the content of SP is now 3.
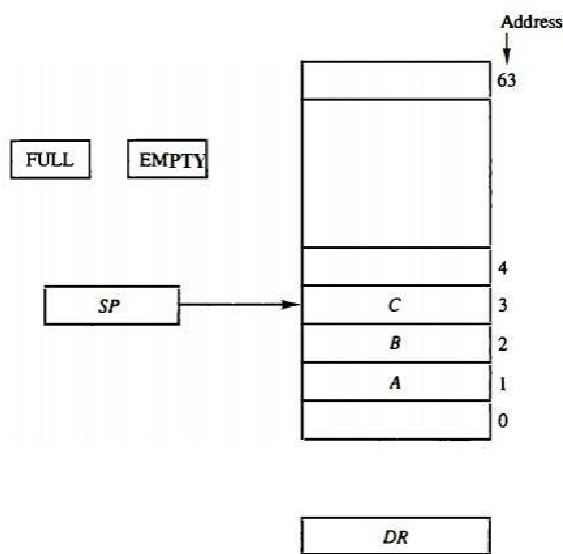
Figure 8-3 Block diagram of a 64-word stack.

- **To remove the top item**, the stack is popped by reading the memory word at address 3 and decrementing the content of SP

- **Item B is now on top of the stack since SP holds address 2**. To insert a new item, the stack is pushed by incrementing SP and writing a word in the next-higher

location in the stack. Note that item C has been read out but not physically removed.

o **This does not matter because when the stack is pushed**, a new item is written in its place. In a 64-word stack, the stack pointer contains 6 bits because $2^6 = 64$.

o **Since SP has only six bits**, it cannot exceed a number greater than 63 (111111 in binary). When63 is incrementedby 1, the resultis 0 since $111111 + 1 = 1000000$ in binary, but SP can accommodate only the six least significant bits.

o **Similarly, when 000000 is decremented by 1**, the result is 111111. The one-bit register FULL is set to 1 when the stack is full, and the one-bit register EMPTY is set to 1 when the stack is empty of items. DR is the data register that holds the binary data to be written into or read out of the stack.

o **Initially**, SP is cleared to 0, EMPTY is set to 1, and FULL is cleared to 0, so that SP points to the word at address 0 and the stack is marked empty and not full. If the stack is not full (if FULL = 0), a new item is inserted with a push operation.

o **The push operation** is implemented with the following sequence of microoperations;

- SP ← SP + 1 Increment stack pointer
- M[SP] ← DR Write item on top of the stack
- If (SP = 0) then (FULL ←1) Check if stack is full
- EMPTY ← 0 Mark the stack not empty

- **The stack pointer** is incremented so that it points to the address of the next-higher word. A memory write operation inserts the word from DR into the top of the stack. Note that SP holds the address of the top of the stack and that M[SP] denotes the memory word specified by the address presently available in SP.
- **The first item stored** in the stack is at address L The last item is stored at address 0.

- **If SP reaches 0**, the stack is full of items, so FULL is set to L This condition is reached if the top item prior to the last push was in location 63 and, after incrementing SP, the last item is stored in location 0.

- **Once an item is stored** in location 0, there are no more empty registers in the stack. If an item is written in the stack, obviously the stack cannot be empty, so EMPTY is cleared to 0.

```
DR ← M[SP] Read item from the top of stack
SP ← SP - 1  Decrement stack pointer
```

```
If (SP = 0) then (EMPTY ← 1)   Check if stack is empty

FULL ← 0     Mark the stack not full
```

A new item is deleted from the stack if the stack is not empty (if EMPTY = 0). The pop operation consists of the following sequence of microoperations:

The top item is read from the stack into DR. The stack pointer is then decremented. If its value reaches zero, the stack is empty, so EMPTY is set to 1.

This condition is reached if the item read was in location 1. Once this item is read out, SP is decremented and reaches the value 0, which is the initial value of SP. Note that if a pop operation reads the item from location 0 and then SP is decremented, SP changes to 111111, which is equivalent to decimal 63.

In this configuration, the word in address 0 receives the last item in the stack. Note also that an erroneous operation will result if the stack is pushed when FULL = 1 or popped when EMPTY = 1.

Memory Stack

A stack can exist as a stand-alone unit as in Fig. 3 or can be implemented in a random-access memory attached to a CPU. The implementation of a stack in the CPU is done by assigning a portion of memory to a stack operation and using a processor register as a stack pointer.

Figure 4 shows a portion of computer memory partitioned into three segments: program, data, and stack. The program counter PC points at the address of the next instruction in the program. The address register AR points at an array of data
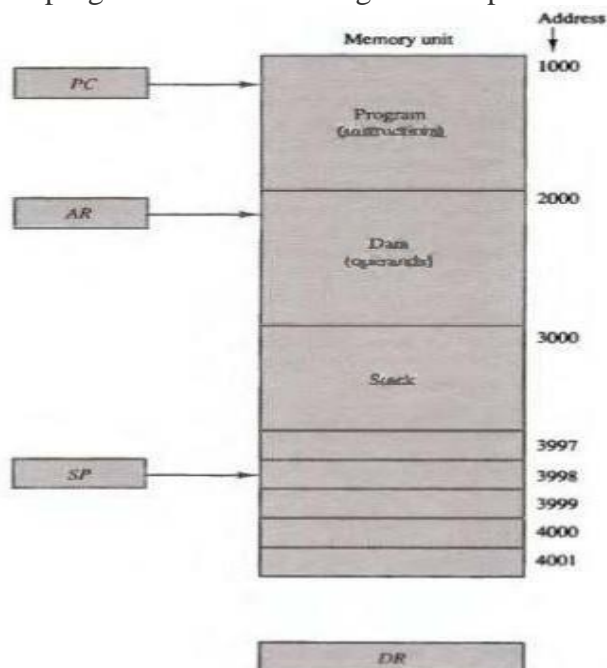
Figure 8-4 Computer memory with program, data, and stack segments.

The stack pointer SP points at the top of the stack. The three registers are connected to a common address bus, and either one can provide an address for memory.

16

PC is used during the fetch phase to read an instruction. AR is used during the execute phase to read an operand.

SP is used to push or pop items into or the stack. As shown in Fig. 4, the initial value of SP is 4001 and the stack grows with decreasing addresses.

Thus the first item stored in the stack is at address 4000, the second item is stored at address 3999, and the last address that can be used for the stack Is 3000.

No provisions are available for stack limjt checks.

We assume that the items in the stack communicate with a data register DR . A new item is inserted with the push operation as follows

$$SP \leftarrow SP - 1$$
$$M[SP] \leftarrow DR$$

The stack pointer is decremented so that it points at the address of the next word. A memory write operation inserts the word from DR into the top of the stack. A new item is deleted with a pop operation as follows:

$$DR \leftarrow M[SP]$$
$$SP \leftarrow SP + 1$$

The top item is read from the stack into DR. The stack pointer is then incremented to point at the next item in the stack.

Most computers do not provide hardware to check for stack overflow (full stack) or underflow (empty stack).

The stack limits can be checked by using two processor registers: one to hold the upper limit (3000 in this case), and the other to hold the lower limit (4001 in this case).

After a push operation, SP is compared with the upper-limit register and after a pop operation, SP is compared with the lower-limit register.

The two microoperations needed for either the push or pop are (1) an access to memory through SP, and (2) updating SP. Which of the two microoperations is done first and whether SP is updated by incrementing or decrementing depends on the organization of the stack.

In Fig. 4 the stack grows by decreasing the memory address. The stack may be constructed to grow by increasing the memory address as in Fig. 3.

In such a case, SP is incremented for the push operation and decremented for the pop operation. A stack may be constructed so that SP points at the next empty location above the top of the stack.

In this case the sequence of microoperations must be interchanged. A stack pointer is loaded with an initial value. This initial value must be the bottom address of an assigned

stack in memory. Henceforth, SP is automatically decremented or incremented with every push or pop operation.

The advantage of a memory stack is that the CPU can refer to it without having to specify an address, since the address is always available and automatically updated in the stack pointer.

Notations

Infix notation

A+B

Prefix notation or Polish notation

+AB

Postfix notation or Reverse polish notation

AB+

Evaluation of Arithmetic Expressions

The procedure consists of first converting the arithmetic expression into its equivalent reverse Polish notation. The operands are pushed into the stack in the order in which they appear. The initiation of an operation depends on whether we have a calculator or a computer. In a calculator, the operators are entered through the keyboard. In a computer, they must be initiated by instructions that contain an operation field (no address field is required). The following rnicrooperations are executed with the stack when an operation is entered in a calculator or issued by the control in a computer: (1) the two topmost operands in the stack are used for the operation, and (2) the stack is popped and the result of the operation replaces the lower operand. By pushing the operands into the stack continuously and performing the operations as defined above, the expression is evaluated in the proper order and the final result remains on top of the stack. The following numerical example may clarify this procedure. Consider the arithmetic expression

(3 * 4) + (5 * 6) In reverse Polish notation, it is expressed as 34•56•+ Now consider the stack operations shown in Fig. 8-5. Each box represents one stack operation and the arrow always points to the top of the stack. Scanning the expression from left to right, we encounter two operands. First the number 3 is pushed into the stack, then the number 4. The next symbol is the multiplication operator • . This causes a multiplication of the two topmost items in the stack. The stack is then popped and the product is placed on top of the stack, replacing the two original operands. Next we encounter the two operands 5 and 6, so they are pushed into the stack. The stack operation that results from the next • replaces these two numbers by their product. The last operation causes an arithmetic addition of the two topmost numbers in the stack to produce the final result of 42. Scientific calculators that employ an internal stack require that the user

convert the arithmetic expressions into reverse Polish notation. Computers that use a stack-organized CPU provide a system program to perform the

Figure 8-5 Stack operations to evaluate 3 · 4 + 5 · 6.
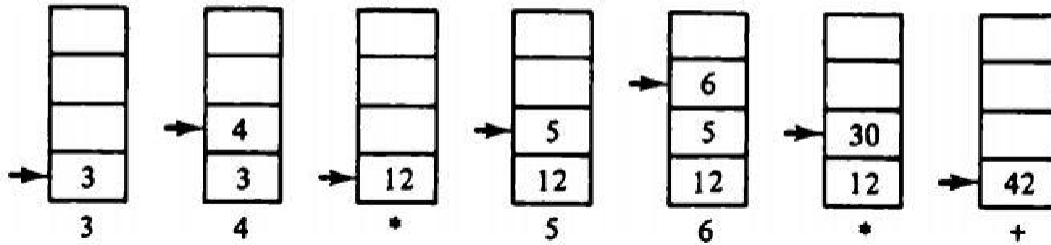
conversion for the user. Most compilers, irrespective of their CPU organization, convert all arithmetic expressions into Polish notation anyway because this is the most efficient method for translating arithmetic expressions into machine language instructions. So in essence, a stack-organized CPU may be more efficient in some applications than a CPU without a stack.

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 register.

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

19

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.

Fields:

Mode field(S): determine how address field is to be interpreted to get effective address or the operand.

Opcode field: Specify the operations to be performed

Address field: designate memory address(es) or processor registers

Mode field	Opcode field	Address field

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 $X = (A + B) \cdot (C + D)$ is shown below

```
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
```

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

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 $X = (A + B) \cdot (C + D)$ is as follows:

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

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

One-address instructions

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 register and assume that the AC contains the result of all operations. The program to evaluate $X = (A + B) \cdot (C + D)$ is

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

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

Zero-Address Instructions

A stack-organized computer does not use an address field for the instructions ADD and MUL. The PUSH and POP instructions, however, need an address field to specify the operand that communicates with the stack. The following program shows how $X = (A + B) \cdot (C + D)$ 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

- Specifies a rule for interpreting or modifying the address field of the instruction
- Effective address is the address of the operand for any computational instruction.
- Variety of addressing modes
  - To give programming flexibility to the user
  - To use bits in the address fields of the instruction effectively

Mode field	Opcode	Address

1.Implied Mode:

In this mode the operands are specified implicitly in the definition of the instruction.

Effective address is always equal to accumulator

EA=AC or EA=stack[SP]

2.Immediate Mode:

In this mode the operand is specified in the instruction itself.

Fast to acquire an operand

3.Register Mode:

Address of the instruction is the register itself.

Effective address is always equal to the register field of instruction register.

Symbolically EA= IR(R)

(IR(R) Register field of IR)

4.Register Indirect Mode:

The instruction here specifies the register which contains the address of the operand

Symbolically EA= IR(R) [X]

EA= effective address IR=instruction register R=register X=operand

5. Autoincrement or Autodecrement Mode:

Same as the register indirect, but the value in the register is incremented or decremented by 1(after or before the execution of the instruction)

6. Relative Addressing Mode:

The content of program counter are added to the address part of the instruction to find the Effective address.

EA=PC+IR(address)

7. Direct Addressing Mode:

Instruction specifies the address which can directly be accessed from the main memory

Symbolically EA=IR(address) where IR(address) address field of IR

8. Indirect direct addressing mode:

The address field of an instruction specifies the address of a memory location that contains the address of the operand

Symbolically EA=M[IR[X]]

9. Indexed Addressing Mode:

In the mode the content of index register (X) is added to the address part of instruction to find the effective address.

EA=IR(address + X) where X is a special cpu register containing index value

10. Base Register Addressing Mode:

In this mode, the content of base register is added to the address part of instruction to find the effective address

EA = IR(Address)+ Base register

Where base register holds a base address

UNIT-IV

INPUT-OUTPUT ORGANIZATION

Peripheral Devices:

The Input / output organization of computer depends upon the size of computer and the peripherals connected to it. The I/O Subsystem of the computer, provides an efficient mode of communication between the central system and the outside environment

The most common input output devices are:

      i) Monitor

      ii) Keyboard

      iii) Mouse

      iv) Printer

      v) Magnetic tapes

   The devices that are under the direct control of the computer are said to be <u>connected online.</u>

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 <u>special communication links</u> for interfacing them with the central processing unit.

   The purpose of communication link is to <u>resolve the differences that exist between the central computer and each peripheral.</u>

The Major Differences are:-

1. Peripherals are electromechnical and electromagnetic devices and CPU and memory are electronic devices. Therefore, a conversion of signal values may be needed.

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

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

4. The operating modes of peripherals are different from each other and 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 supervises and synchronizes all input and out transfers

- These components are called Interface Units because they interface between the processor bus and the peripheral devices.

I/O BUS and Interface Module

It defines the typical link between the processor and several peripherals.

The I/O Bus consists of data lines, address lines and control lines.

The I/O bus from the processor is attached to all peripherals interface.

To communicate with a particular device, the processor places a device address on address lines.

Each Interface decodes the address and control received from the I/O bus, interprets them for peripherals and provides signals for the peripheral controller.

It is also synchronizes the data flow and supervises the transfer between peripheral and processor.

Each peripheral has its own controller.

For example, the printer controller controls the paper motion, the print timing

The control lines are referred as I/O command. The commands are as following:

Control command- A control command is issued to activate the peripheral and to inform it what to do.

Status command- A status command is used to test various status conditions in the interface and the peripheral.

Data Output command- A data output command causes the interface to respond by transferring data from the bus into one of its registers.

Data Input command- The data input command is the opposite of the data output.

In this case the interface receives on item of data from the peripheral and places it in its buffer register. I/O Versus Memory Bus
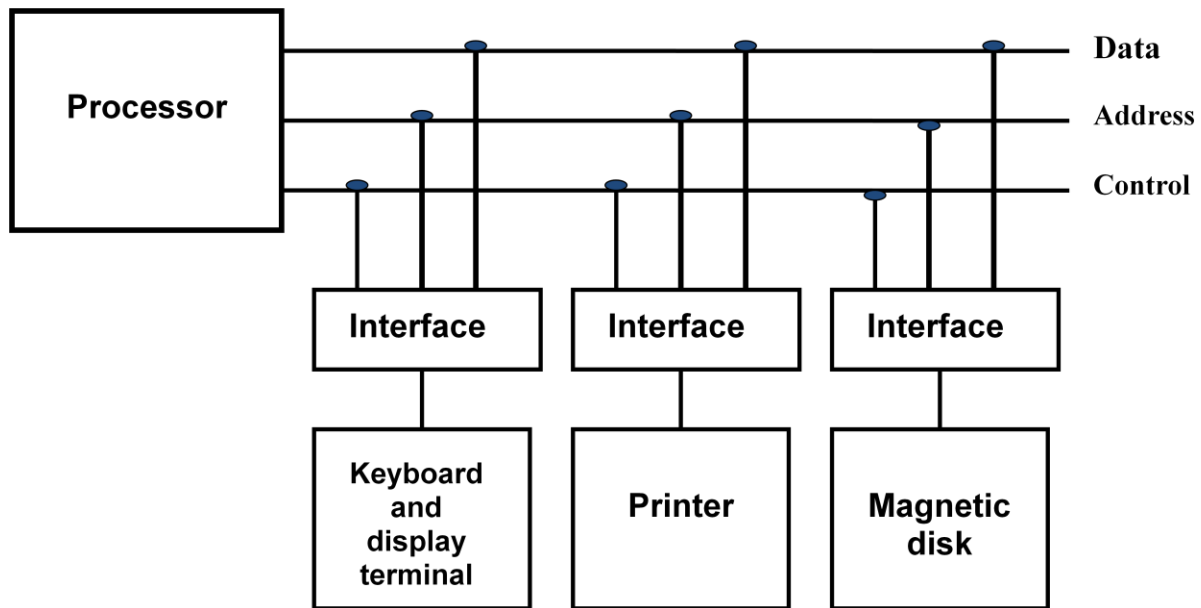
Connection of I/O bus to input-output devices

 To communicate 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 3 ways that computer buses can be used to communicate with memory and I/O:

  i.   Use two Separate buses , one for memory and other for I/O.

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

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

I/O Processor

   In the first method, the computer has independent sets of data, address and control buses one for accessing memory and other for I/O. This is done in computers that provides a separate I/O processor (IOP). The purpose of IOP is to provide an independent pathway for the transfer of information between external device and internal memory.

Asynchronous Data Transfer :

This Scheme is used when speed of I/O devices do not match with microprocessor, and timing characteristics of I/O devices is not predictable. In this method, process initiates the device and check its status. As a result, CPU has to wait till I/O device is ready to transfer data. When device is ready CPU issues instruction for I/O transfer. In this method two types of techniques are used based on signals before data transfer.
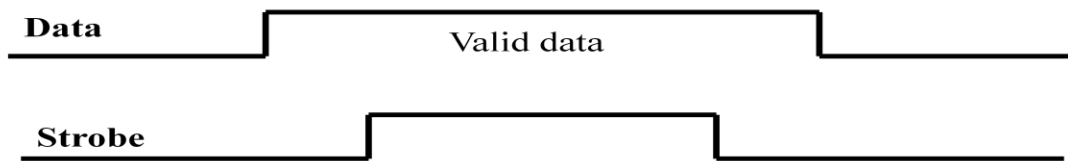
  i. Strobe Control

  ii. Handshaking

Strobe Signal :

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.

Data Transfer Initiated by Source Unit:
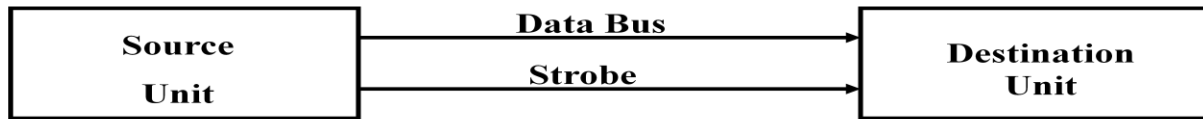
(a)   **Block Diagram**

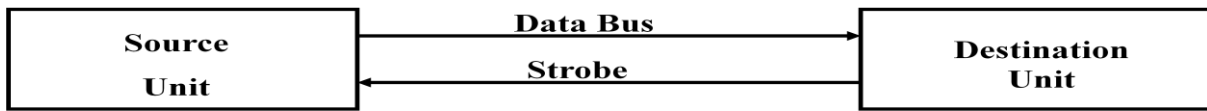(b)   **Timing Diagram**

Source-Initiated strobe for Data Transfer

In the block diagram fig. (a), the data bus carries the binary information from source to 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.

The timing diagram fig. (b) the source unit first places the data on the data bus. The information on the data bus and strobe signal remain in the active state to allow the destination unit to receive the data.

Data Transfer Initiated by Destination Unit:

   In this method, the destination unit activates the strobe pulse, to informing the source to provide the data. The source will respond 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. When accepted the destination unit then disables the strobe and the source unit removes the data from the bus.

(a)   **Block Diagram**
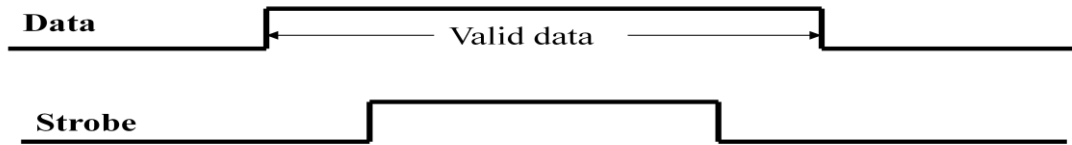
(b)   **Timing Diagram**

Destination–Initiated strobe for Data Transfer

Disadvantage of Strobe Signal :

The disadvantage of the strobe method is that, the source unit initiates the transfer has no way of knowing whether the destination unit has actually received the data item that was places 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 bus. The Handshaking method solves this problem.

Handshaking:

The handshaking method solves the problem of strobe method by introducing a second control signal that provides a reply to the unit that initiates the transfer.

Principle of Handshaking:

The basic principle of the two-wire handshaking method of data transfer is as follow:

One control line is in the same direction as the data flows in the bus from the source to destination. It is used by source unit to inform the destination unit whether there a valid 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 the data. The sequence of control during the transfer depends on the unit that initiates the transfer.

Source Initiated Transfer using Handshaking:

The sequence of events shows 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 accepted* signal and the system goes into its initial state.
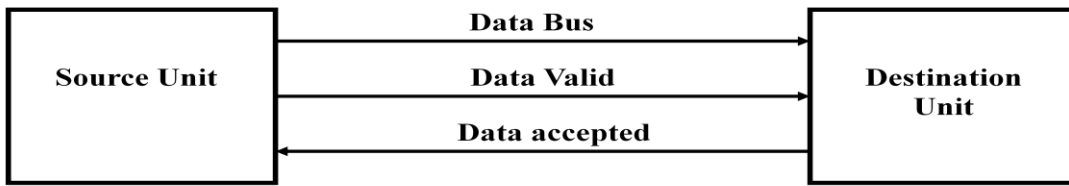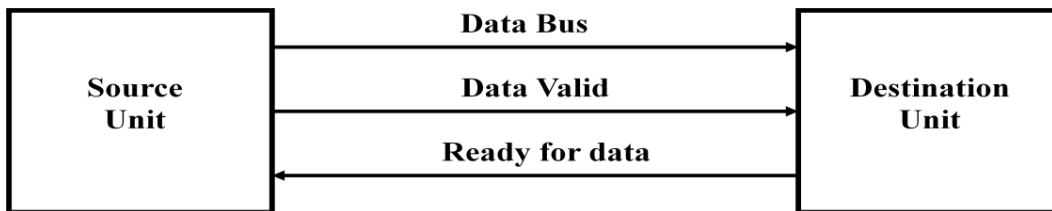
(a) Block Diagram
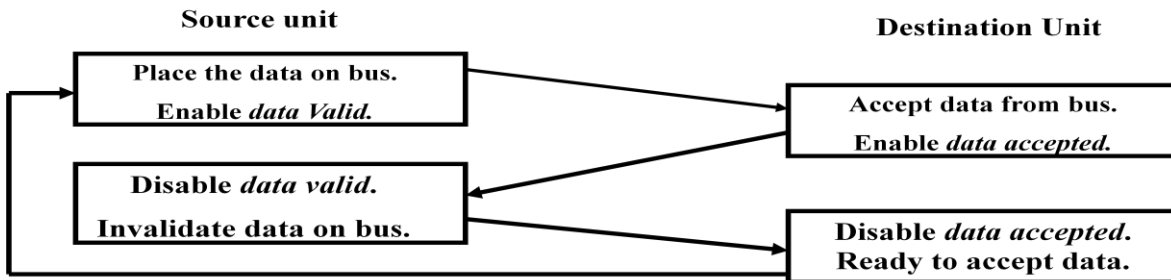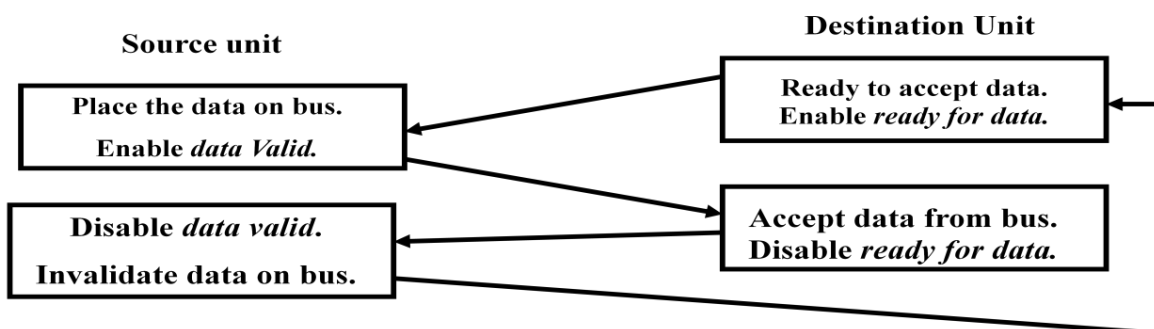

(b) Sequence of events

Destination Initiated Transfer Using Handshaking:

The name of the signal generated by the destination unit has been changed to *ready for data* to reflects 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.

The only difference between the Source Initiated and the Destination Initiated transfer is in their choice of Initial sate.

(a) Block Diagram

(b) Sequence of events

Destination-Initiated transfer using Handshaking

6

Advantage of the Handshaking method:

➢ The Handshaking scheme provides degree of flexibility and reliability because the successful completion of data transfer relies on active participation by both units.

➢ If any of one unit is faulty, the data transfer will not be completed. Such an error can be detected by means of a *Timeout mechanism* which provides an alarm if the data is not completed within time.

Asynchronous Serial Transmission:

The transfer of data between two units is serial or parallel. In parallel data transmission, n bit in the message must be transmitted through n separate conductor path. In serial transmission, each bit in the message is sent in sequence one at a time.

Parallel transmission is faster but it requires many wires. It is used for short distances and where speed is important. Serial transmission is slower but is less expensive.

In Asynchronous serial transfer, each bit of message is sent a sequence at a time, and binary information is transferred only when it is available. When there is no information to be transferred, line remains idle.

In this technique each character consists of three points :

        i. Start bit

        ii. Character bit

        iii. Stop bit

i.   Start Bit- First bit, called start bit is always zero and used to indicate the beginning character.

ii.   Stop Bit-   Last bit, called stop bit is always one and used to indicate end of characters. Stop bit is always in the 1- state and frame the end of the characters to signify the idle or wait state.

iii.   Character Bit- Bits in between the start bit and the stop bit are known as character bits. The character bits always follow the start bit.
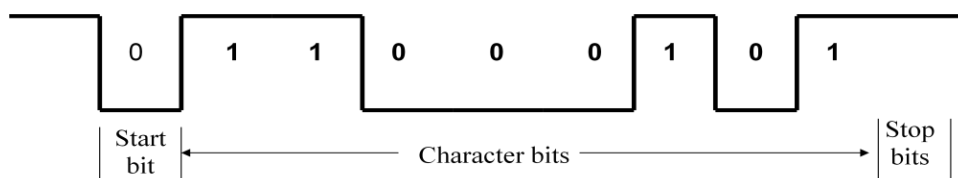
Asynchronous Serial Transmission

Serial Transmission of Asynchronous is done by two ways:

a) Asynchronous Communication Interface

b) First In First out Buffer

Asynchronous Communication Interface:

It works as both a receiver and a transmitter. Its operation is initialized by CPU by sending a byte to the control register.

The transmitter register accepts a data byte from CPU through the data bus and transferred to a shift register for serial transmission.

The receive portion receives information into another shift register, and when a complete data byte is received it is transferred to receiver register.

CPU can select the receiver register to read the byte through the data bus. Data in the status register is used for input and output flags.

First In First Out Buffer (FIFO):

A First In First Out (FIFO) Buffer is a memory unit that stores information in such a manner that the first item is in the item first out. A FIFO buffer comes with separate input and output terminals. The important feature of this buffer is that it can input data and output data at two different rates.

When placed between two units, the FIFO can accept data from the source unit at one rate, rate of transfer and deliver the data to the destination unit at another rate.

If the source is faster than the destination, the FIFO is useful for source data arrive in bursts that fills out the buffer. FIFO is useful in some applications when data are transferred asynchronously.

Modes of Data Transfer :

Transfer of data is required between CPU and peripherals or memory or sometimes between any two devices or units of your computer system. To transfer a data from one unit to another one should be sure that both units have proper connection and at the time of data transfer the receiving unit is not busy. This data transfer with the computer is Internal Operation.

All the internal operations in a digital system are synchronized by means of clock pulses supplied by a common clock pulse Generator. The data transfer can be

i. Synchronous or

ii. Asynchronous

When both the transmitting and receiving units use same clock pulse then such a data transfer is called Synchronous process. On the other hand, if the there is not concept of clock pulses

and the sender operates at different moment than the receiver then such a data transfer is called Asynchronous data transfer.

The data transfer can be handled by various modes. some of the modes use CPU as an intermediate path, others transfer the data directly to and from the memory unit and this can be handled by 3 following ways:

i. Programmed I/O

ii. Interrupt-Initiated I/O

iii. Direct Memory Access (DMA)

Programmed I/O Mode:

In this mode of data transfer the operations are the results in I/O instructions which is a part of computer program. Each data transfer is initiated by a instruction in the program. Normally the transfer is from a CPU register to peripheral device or vice-versa.

Once the data is initiated the CPU starts monitoring the interface to see when next transfer can made. The instructions of the program keep close tabs on everything that takes place in the interface unit and the I/O devices.

Figure   Data transfer from I/O device to CPU.
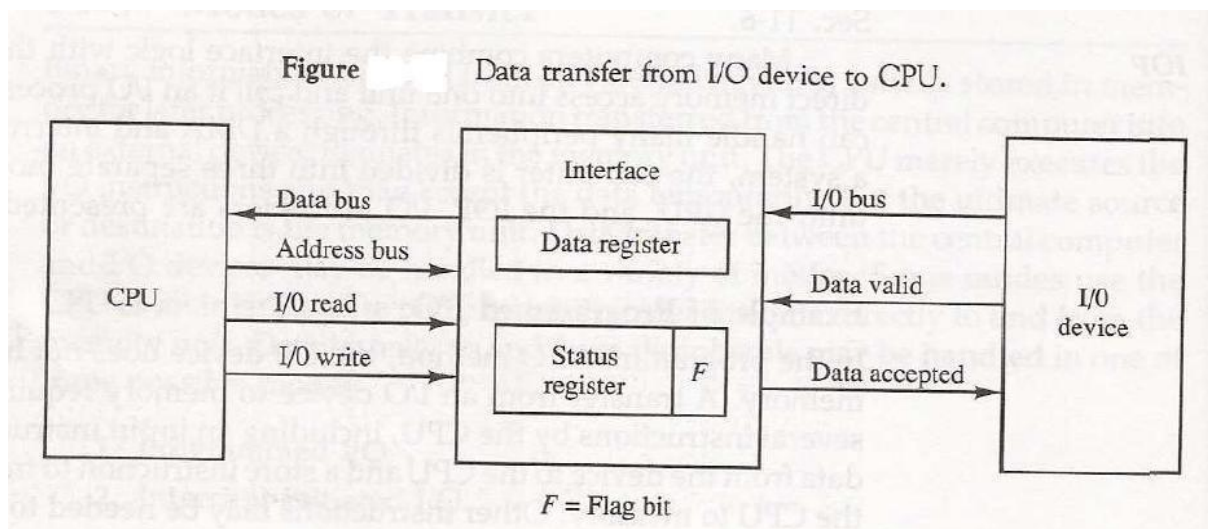
F = Flag bit

● The transfer of data 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.

In this technique CPU is responsible for executing data from the memory for output and storing data in memory for executing of Programmed I/O as shown in Flowchart-:
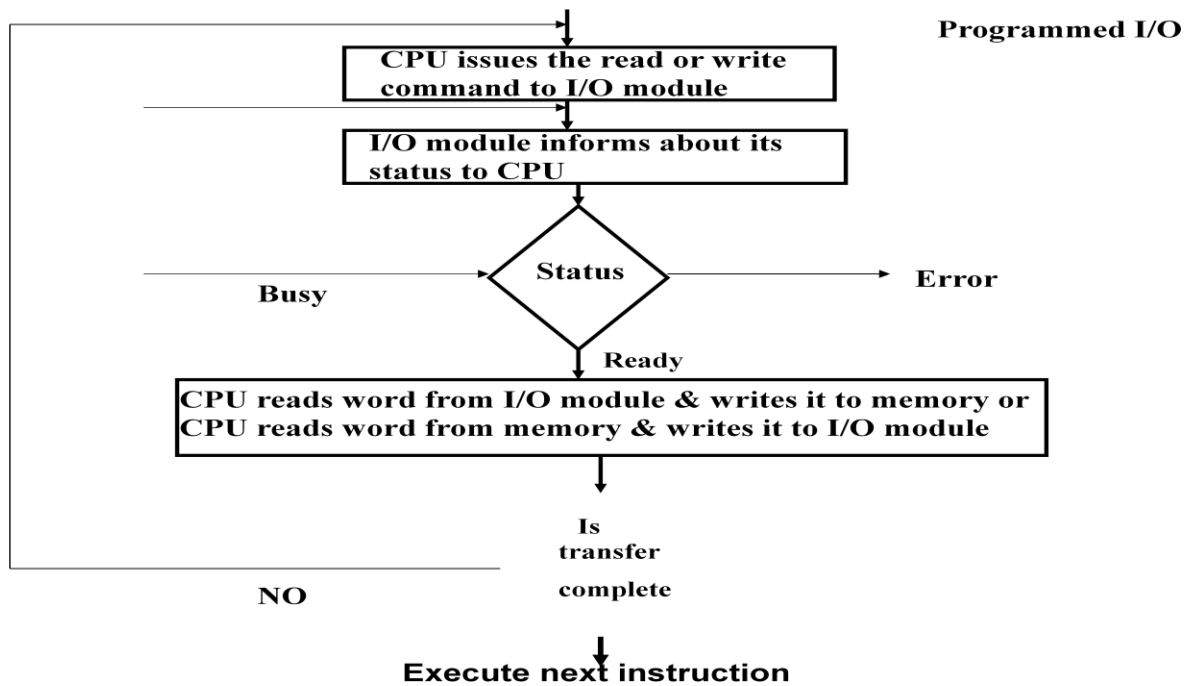
Drawback of the Programmed I/O :

The main drawback of the Program Initiated I/O was that the CPU has to monitor the units all the times when the program is executing. Thus 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 and the CPU time is wasted a lot in keeping an eye to the executing of program.

To remove this problem an Interrupt facility and special commands are used.

Interrupt-Initiated I/O :

In this method an interrupt facility an interrupt command is used to inform the device about the start and end of transfer. In the meantime the CPU executes other program. When the interface determines that the device is ready for data transfer it generates an Interrupt Request and sends it to the computer.

When the CPU receives such an signal, it temporarily stops the execution of the program and branches to a service program to process the I/O transfer and after completing it returns back to task, what it was originally performing.

- In this type of IO, computer does not check the flag. It continue to perform its task.

- Whenever any device wants the attention, it sends the interrupt signal to the CPU.

- CPU then deviates from what it was doing, store the return address from PC and branch to the address of the subroutine.

- There are two ways of choosing the branch address:

    - Vectored Interrupt

    - Non-vectored Interrupt

- In vectored interrupt the source that interrupt the CPU provides the branch information. This information is called interrupt vectored.

- In non-vectored interrupt, the branch address is assigned to the fixed address in the memory.

Priority Interrupt:

- There are number of IO devices attached to the computer.

- They are all capable of generating the interrupt.

- When the interrupt is generated from more than one device, priority interrupt system is used to determine which device is to be serviced first.

- Devices with high speed transfer are given higher priority and slow devices are given lower priority.

- Establishing the priority can be done in two ways:

    - Using Software

    - Using Hardware

- A pooling procedure is used to identify highest priority in software means.

Polling Procedure :

- There is one common branch address for all interrupts.

- Branch address contain the code that polls the interrupt sources in sequence. The highest priority is tested first.

- The particular service routine of the highest priority device is served.

- The disadvantage is that time required to poll them can exceed the time to serve them in large number of IO devices.

Using Hardware:

- Hardware priority system function as an overall manager.

- It accepts interrupt request and determine the priorities.

- To speed up the operation each interrupting devices has its own interrupt vector.

- No polling is required, all decision are established by hardware priority interrupt unit.

- It can be established by serial or parallel connection of interrupt lines.

**Serial or Daisy Chaining Priority:**

- Device with highest priority is placed first.

- Device that wants the attention send the interrupt request to the CPU.

- CPU then sends the INTACK signal which is applied to PI(priority in) of the first device.

- If it had requested the attention, it place its VAD(vector address) on the bus. And it block the signal by placing 0 in PO(priority out)

- If not it pass the signal to next device through PO(priority out) by placing 1.

- This process is continued until appropriate device is found.

- The device whose PI is 1 and PO is 0 is the device that send the interrupt request.

Figure   Daisy-chain priority interrupt.
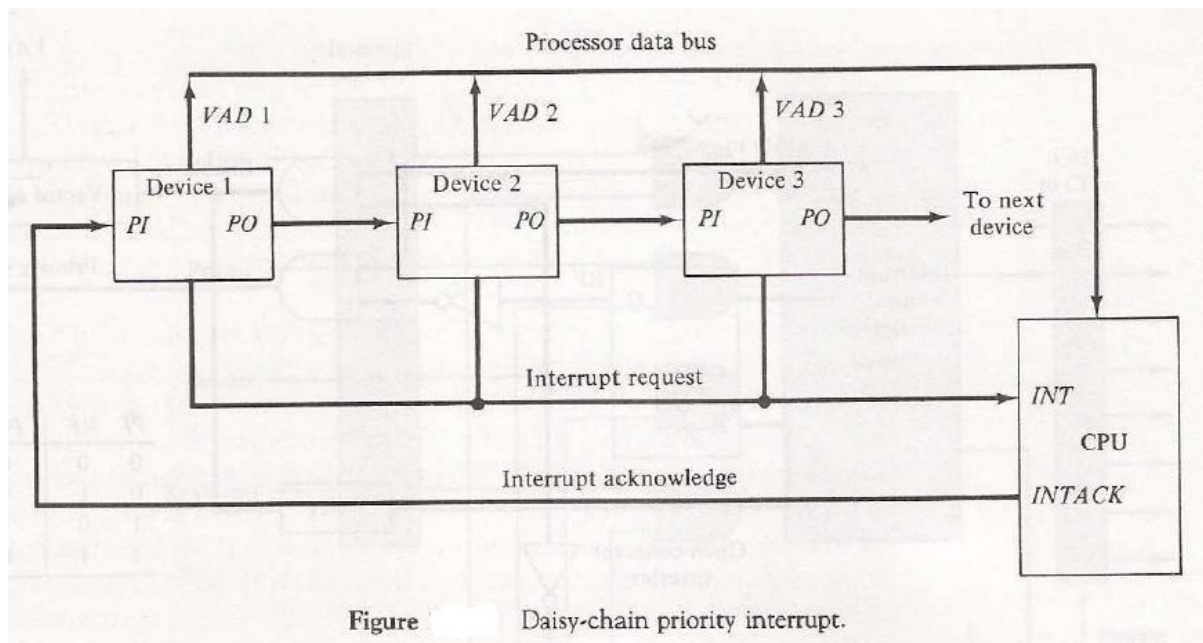
Parallel Priority Interrupt :

- It consist of interrupt register whose bits are set separately by the interrupting devices.

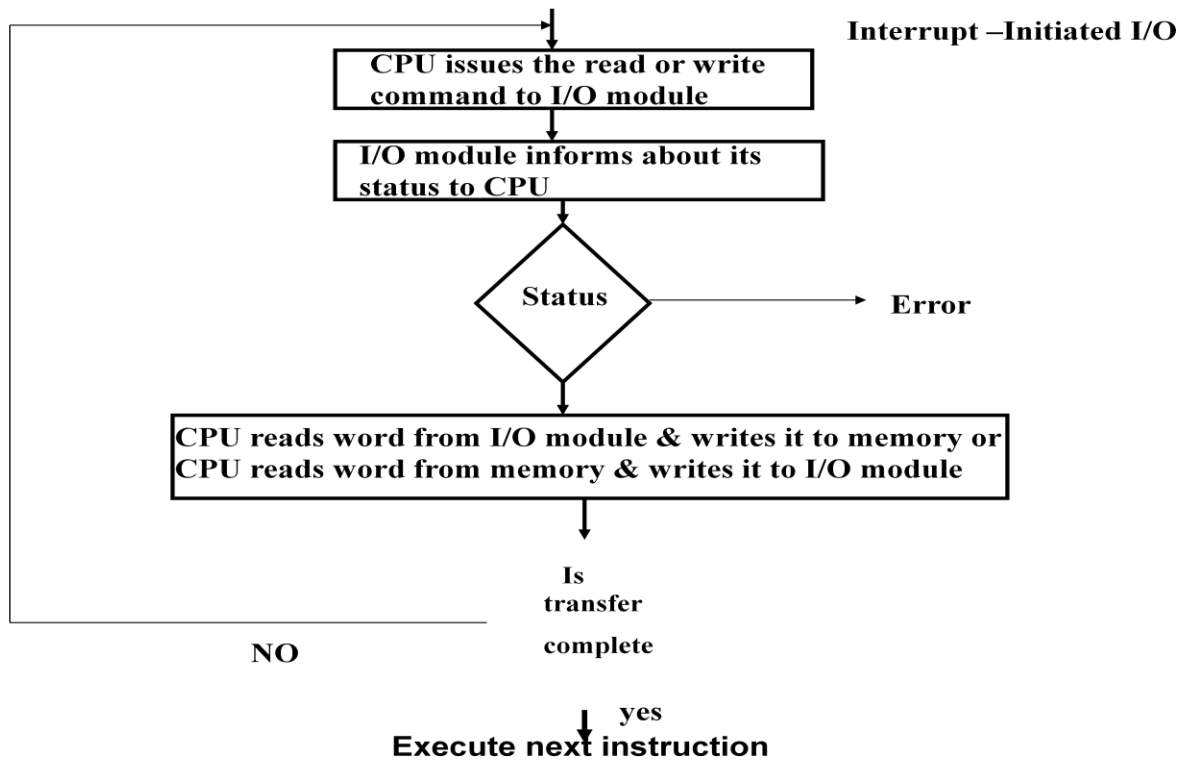- Priority is established according to the position of the bits in the register.

12

- Mask register is used to provide facility for the higher priority devices to interrupt when lower priority device is being serviced or disable all lower priority devices when higher is being serviced.

- Corresponding interrupt bit and mask bit are ANDed and applied to priority encoder.

- Priority encoder generates two bits of vector address.

- Another output from it sets IST(interrupt status flip flop).
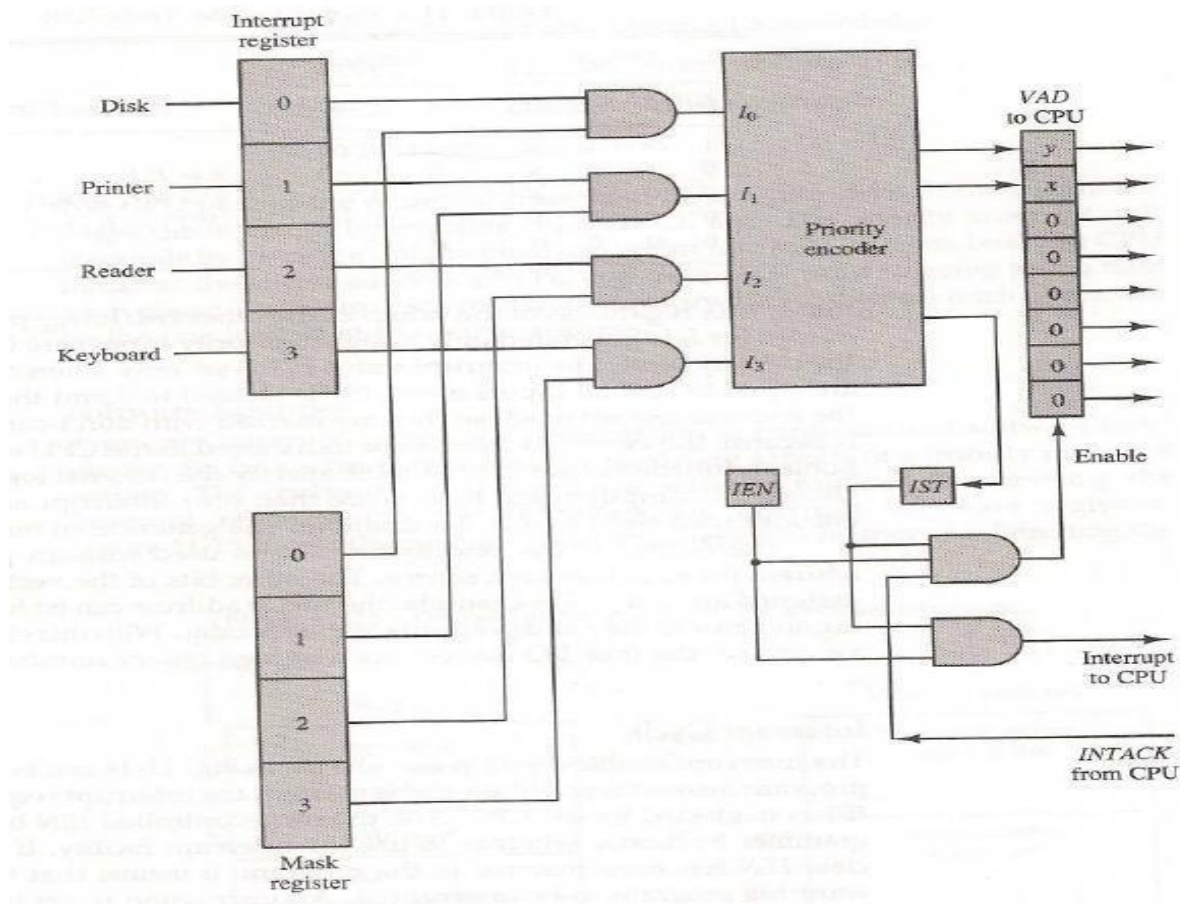
Priority Encoder Truth Table

Inputs				Outputs			Boolean functions
I_0	I_1	I_2	I_3	x	y	IST	
1	×	×	×	0	0	1	
0	1	×	×	0	1	1	$x = I_0'I_1'$
0	0	1	×	1	0	1	$y = I_0'I_1 + I_0'I_2'$
0	0	0	1	1	1	1	$(IST) = I_0 + I_1 + I_2 + I_3$
0	0	0	0	×	×	0	

The Execution process of Interrupt–Initiated I/O is represented in the flowchart:

```
        ┌──────────────────────────────┐
        │ CPU issues the read or write │
        │ command to I/O module        │
        └──────────────────────────────┘
        ┌──────────────────────────────┐
        │ I/O module informs about its │
        │ status to CPU                │
        └──────────────────────────────┘
                      ◇ Status ──────────→ Error
        ┌──────────────────────────────────────────────┐
        │ CPU reads word from I/O module & writes it to memory or │
        │ CPU reads word from memory & writes it to I/O module    │
        └──────────────────────────────────────────────┘
                      Is
                      transfer
        NO            complete
                      │ yes
        Execute next instruction
```

Direct Memory Access (DMA):

   In the Direct Memory Access (DMA) the interface transfer the data into and out of the memory unit through the memory bus. 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 the 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 microprocessor is to disable the buses through special control signals such as:

- Bus Request (BR)

- Bus Grant (BG)

   These two control signals in the CPU that facilitates the DMA transfer. The *Bus Request (BR)* input is used by the *DMA controller* to request the CPU. When this input is active, the CPU terminates the execution of the current instruction and places the address bus, data bus

14

and read write lines into a *high Impedance state.* High Impedance state means that the output is disconnected.
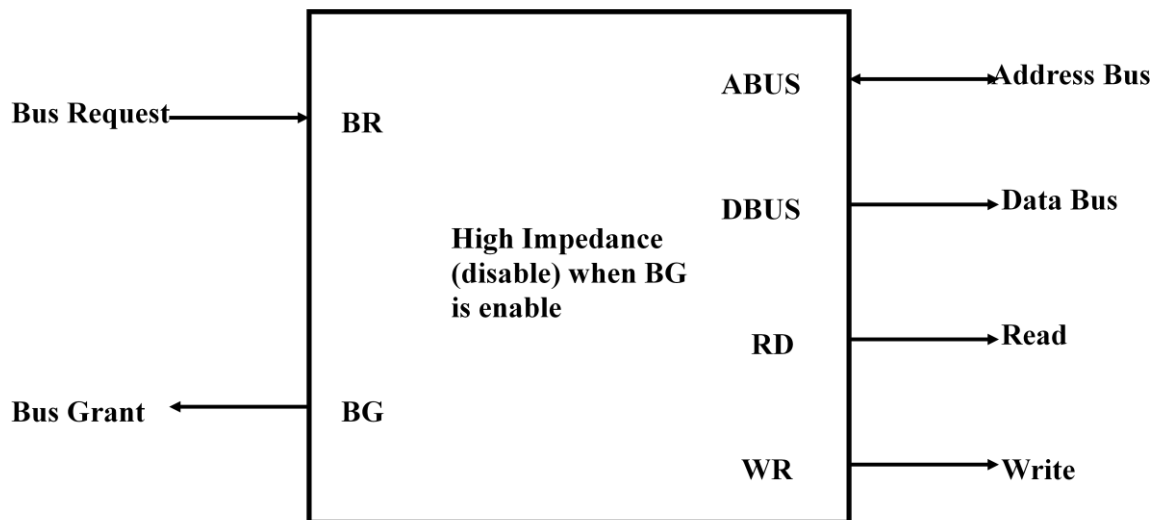
CPU bus Signals for DMA Transfer

The CPU activates the *Bus Grant (BG)* output to inform the external DMA that the Bus Request (BR) can now take control of the buses to conduct memory transfer without processor.

When the DMA terminates the transfer, it disables the *Bus Request (BR)* line. The CPU disables the *Bus Grant (BG)*, takes control of the buses and return to its normal operation.

The transfer can be made in several ways that are:

i. DMA Burst

ii. Cycle Stealing

i) DMA Burst :- In DMA Burst transfer, a block sequence consisting of a number of memory words is transferred in continuous burst while the DMA controller is master of the memory buses.

ii) Cycle Stealing :- Cycle stealing allows the DMA controller to transfer one data word at a time, after which it must returns control of the buses to the CPU.

DMA Controller:

The DMA controller needs the usual circuits of an interface to communicate with the CPU and I/O device. The DMA controller has three registers:

i. Address Register

ii. Word Count Register

iii. Control Register

i. Address Register :- Address Register contains an address to specify the desired location in memory.

ii. Word Count Register :- WC holds the number of words to be transferred. The register is incre/decre by one after each word transfer and internally tested for zero.

i. Control Register :- Control Register specifies the mode of transfer

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 DMA can communicate directly with the memory by specifying an address in the address bus and activating the RD or WR control.
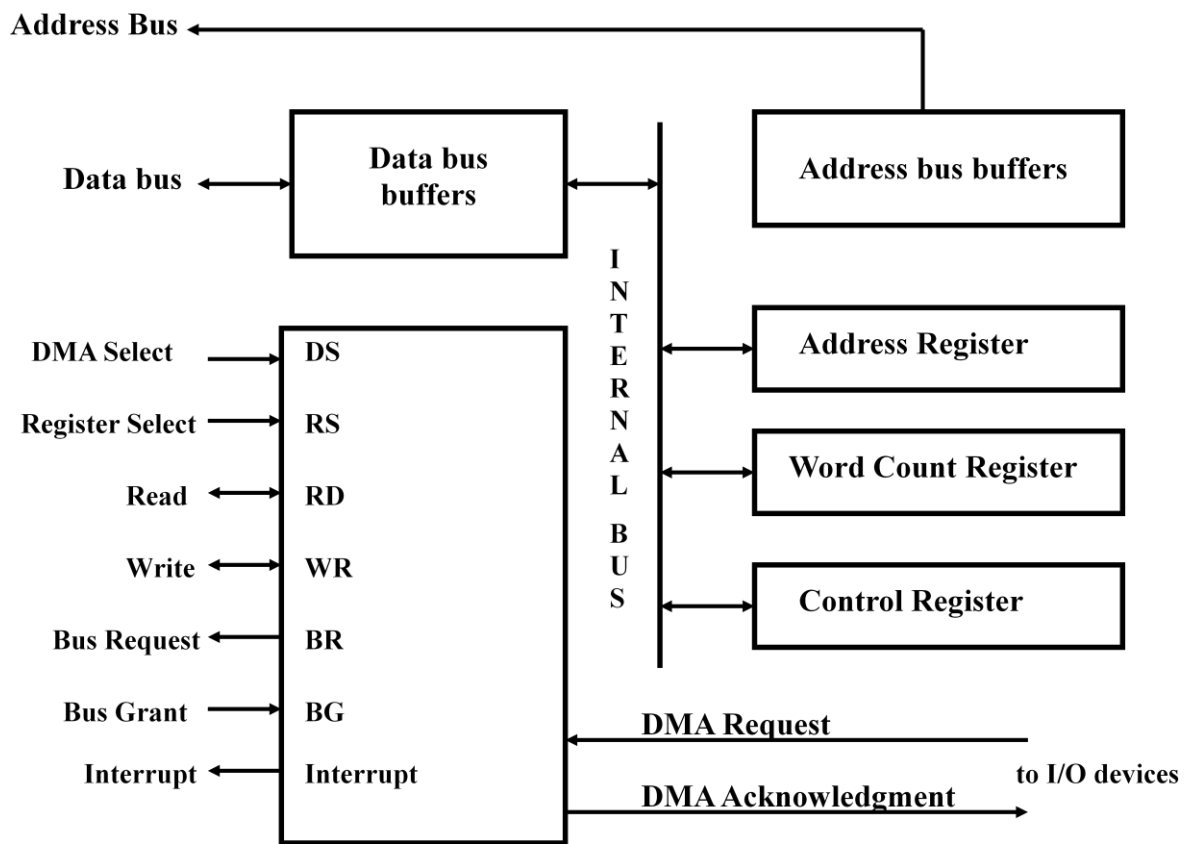
Block Diagram of DMA Controller

DMA Transfer:

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 transfer between the peripheral and the memory.

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 are output lines from the DMA controller to the random access memory to specify the read or write operation of data.

Summary :
- Interface is the point where a connection is made between two different parts of a system.
- The strobe control method of Asynchronous data transfer employs a single control line to time each transfer.
- The handshaking method solves the problem of strobe method by introducing a second control signal that provides a reply to the unit that initiates the transfer.
- Programmed I/O mode of data transfer the operations are the results in I/O instructions which is a part of computer program.
- In the Interrupt Initiated I/O method an interrupt facility an interrupt command is used to inform the device about the start and end of transfer.
- In the Direct Memory Access (DMA) the interface transfer the data into and out of the memory unit through the memory bus.

Input-Output Processor:

- It is a processor with direct memory access capability that communicates with IO devices.

- IOP is similar to CPU except that it is designed to handle the details of IO operation.

- Unlike DMA which is initialized by CPU, IOP can fetch and execute its own instructions.

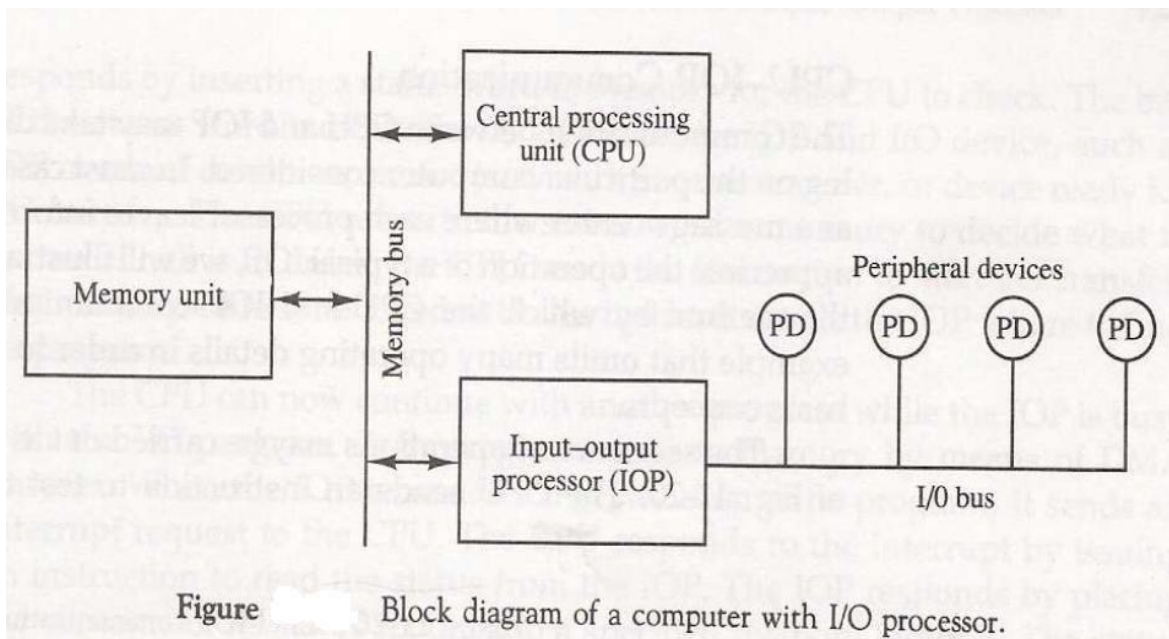- IOP instruction are specially designed to handle IO operation.

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

- Memory occupies the central position and can communicate with each processor by DMA.

- CPU is responsible for processing data.

- IOP provides the path for transfer of data between various peripheral devices and memory.

- Data formats of peripherals differ from CPU and memory. IOP maintain such problems.

- Data are transfer from IOP to memory by stealing one memory cycle.

- Instructions that are read from memory by IOP are called commands to distinguish them from instructions that are read by the CPU.

Figure 11-20 CPU-IOP communication.
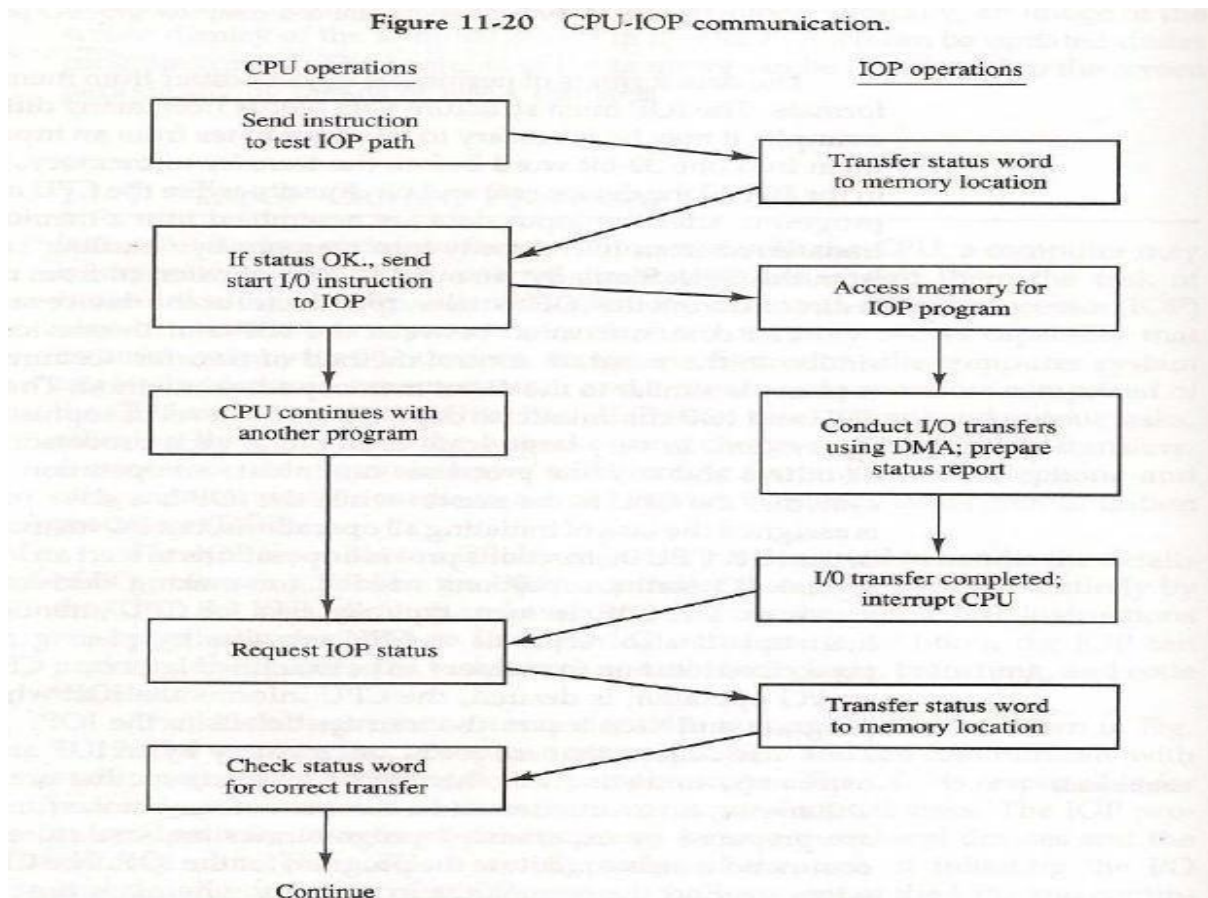
Instruction *that are read from memory by an IOP*

>> Distinguish from instructions that are read by the CPU

>> Commands are prepared by experienced programmers and are stored in memory

>> Command word = IOP program

Memory Organization:

Memory Hierarchy

A memory unit is an essential component in any digital computer since it is needed for storing programs and data.

Typically, a memory unit can be classified into two categories:

1. The memory unit that establishes direct communication with the CPU is called **Main Memory**. The main memory is often referred to as RAM (Random Access Memory).

2. The memory units that provide backup storage are called **Auxiliary Memory**. For instance, magnetic disks and magnetic tapes are the most commonly used auxiliary memories.

Apart from the basic classifications of a memory unit, the memory hierarchy consists all of the storage devices available in a computer system ranging from the slow but high-capacity auxiliary memory to relatively faster main memory.

The following image illustrates the components in a typical memory hierarchy.

Memory Hierarchy in a Computer System:
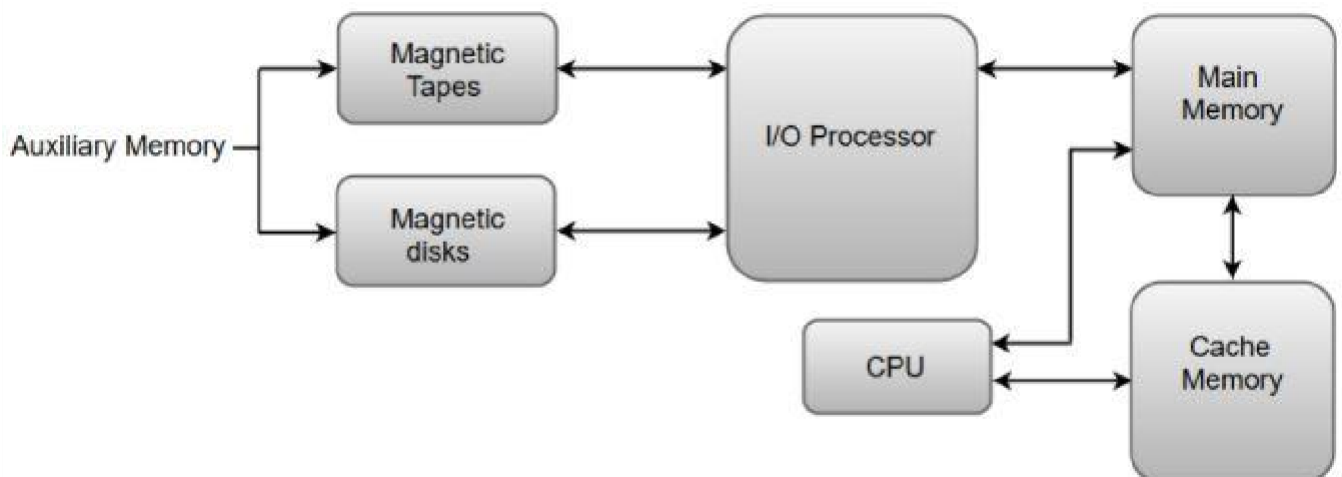
Main Memory

The main memory acts as the central storage unit in a computer system. It is a relatively large and fast memory which is used to store programs and data during the run time operations.

The primary technology used for the main memory is based on semiconductor integrated circuits. The integrated circuits for the main memory are classified into two major units.

1. RAM (Random Access Memory) integrated circuit chips

2. ROM (Read Only Memory) integrated circuit chips

RAM integrated circuit chips

The RAM integrated circuit chips are further classified into two possible operating modes, **static** and **dynamic**.

The primary compositions of a static RAM are flip-flops that store the binary information. The nature of the stored information is volatile, i.e. it remains valid as long as power is applied to the system. The static RAM is easy to use and takes less time performing read and write operations as compared to dynamic RAM.

The dynamic RAM exhibits the binary information in the form of electric charges that are applied to capacitors. The capacitors are integrated inside the chip by MOS transistors. The dynamic RAM consumes less power and provides large storage capacity in a single memory chip.

RAM chips are available in a variety of sizes and are used as per the system requirement. The following block diagram demonstrates the chip interconnection in a 128 * 8 RAM chip.

Typical RAM chip:
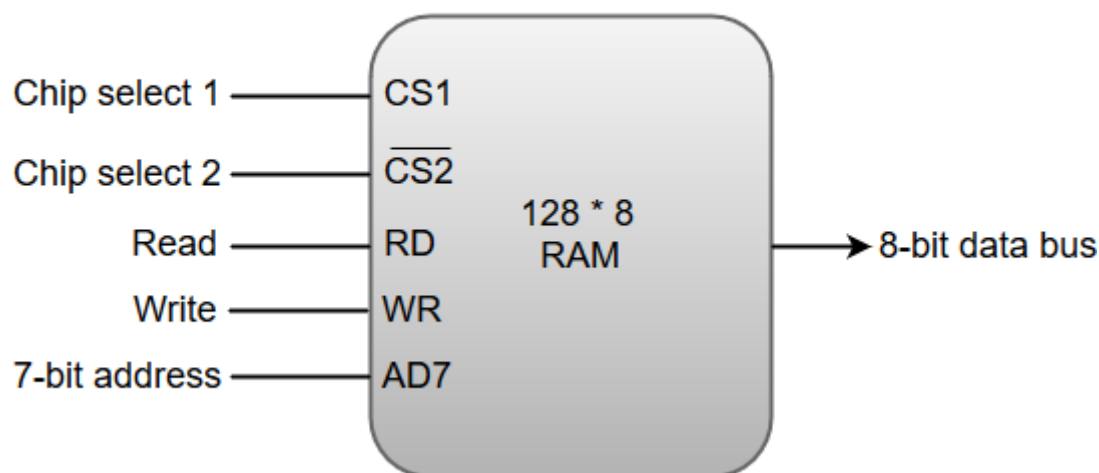
- o A 128 * 8 RAM chip has a memory capacity of 128 words of eight bits (one byte) per word. This requires a 7-bit address and an 8-bit bidirectional data bus.
- o The 8-bit bidirectional data bus allows the transfer of data either from memory to CPU during a **read** operation or from CPU to memory during a **write** operation.
- o The **read** and **write** inputs specify the memory operation, and the two chip select (CS) control inputs are for enabling the chip only when the microprocessor selects it.
- o The bidirectional data bus is constructed using **three-state buffers**.
- o The output generated by three-state buffers can be placed in one of the three possible states which include a signal equivalent to logic 1, a signal equal to logic 0, or a high-impedance state.

The following function table specifies the operations of a 128 * 8 RAM chip.

CS1	$\overline{CS2}$	RD	WR	Memory function	State of data bus
0	0	x	x	Inhibit	High-impedance
0	1	x	x	Inhibit	High-impedance
1	0	0	0	Inhibit	High-impedance
1	0	0	1	Write	Input data to RAM
1	0	1	x	Read	Output data to RAM
1	1	x	x	Inhibit	High-impedance

From the functional table, we can conclude that the unit is in operation only when CS1 = 1 and $\boxed{\overline{CS2}}$ = 0. The bar on top of the second select variable indicates that this input is enabled when it is equal to 0.

ROM integrated circuit

The primary component of the main memory is RAM integrated circuit chips, but a portion of memory may be constructed with ROM chips.

A ROM memory is used for keeping programs and data that are permanently resident in the computer.Apart from the permanent storage of data, the ROM portion of main memory is needed for storing an initial program called a **bootstrap loader**. The primary function of the **bootstrap loader** program is to start the computer software operating when power is turned on.

ROM chips are also available in a variety of sizes and are also used as per the system requirement. The following block diagram demonstrates the chip interconnection in a 512 * 8 ROM chip.

Typical ROM chip:
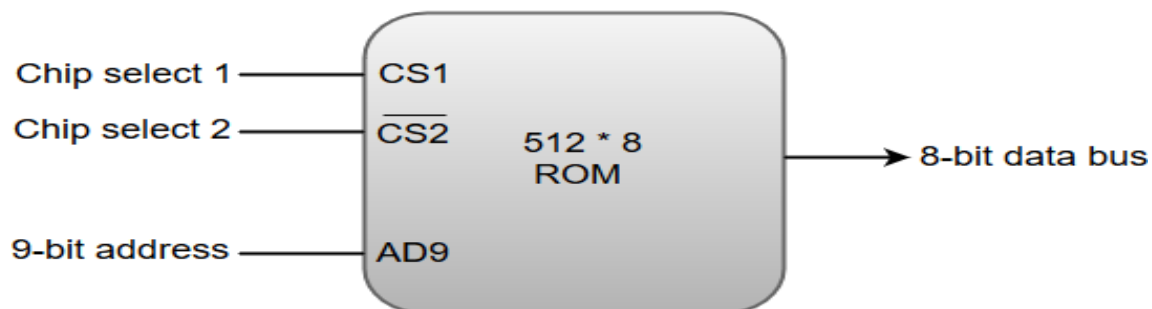
- A ROM chip has a similar organization as a RAM chip. However, a ROM can only perform read operation; the data bus can only operate in an output mode.
- The 9-bit address lines in the ROM chip specify any one of the 512 bytes stored in it.

- o The value for chip select 1 and chip select 2 must be 1 and 0 for the unit to operate. Otherwise, the data bus is said to be in a high-impedance state.

Auxiliary Memory

An Auxiliary memory is known as the lowest-cost, highest-capacity and slowest-access storage in a computer system. It is where programs and data are kept for long-term storage or when not in immediate use. The most common examples of auxiliary memories are magnetic tapes and magnetic disks.

Magnetic Disks

A magnetic disk is a type of memory constructed using a circular plate of metal or plastic coated with magnetized materials. Usually, both sides of the disks are used to carry out read/write operations. However, several disks may be stacked on one spindle with read/write head available on each surface.

The following image shows the structural representation for a magnetic disk.

Magnetic disks
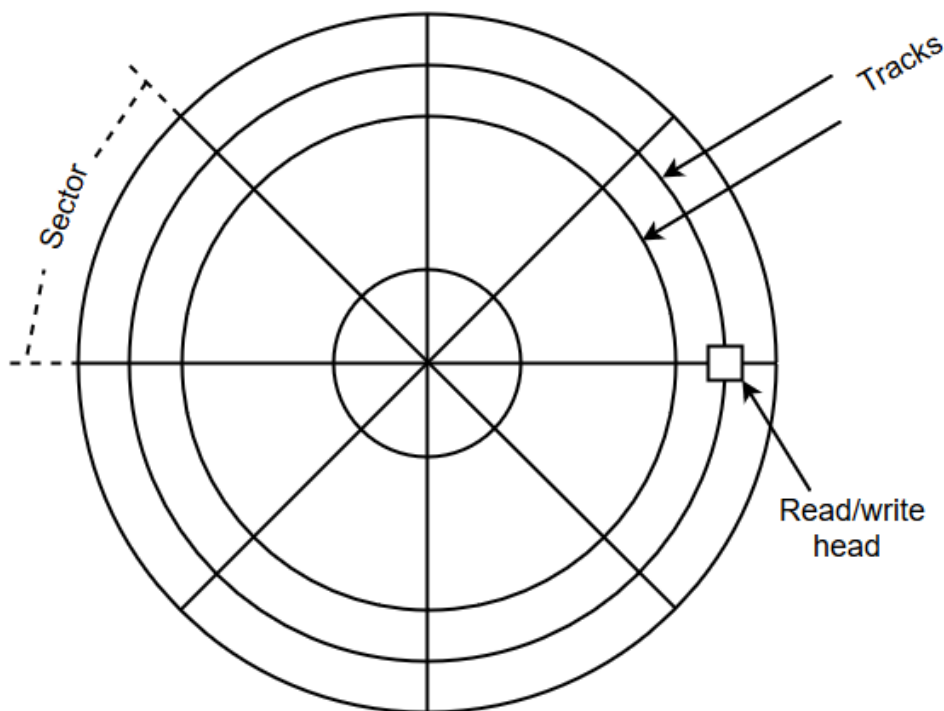
- o The memory bits are stored in the magnetized surface in spots along the concentric circles called tracks.
- o The concentric circles (tracks) are commonly divided into sections called sectors.

Magnetic Tape

Magnetic tape is a storage medium that allows data archiving, collection, and backup for different kinds of data. The magnetic tape is constructed using a plastic strip coated with a magnetic recording medium.

The bits are recorded as magnetic spots on the tape along several tracks. Usually, seven or nine bits are recorded simultaneously to form a character together with a parity bit.

Magnetic tape units can be halted, started to move forward or in reverse, or can be rewound. However, they cannot be started or stopped fast enough between individual characters. For this reason, information is recorded in blocks referred to as records.
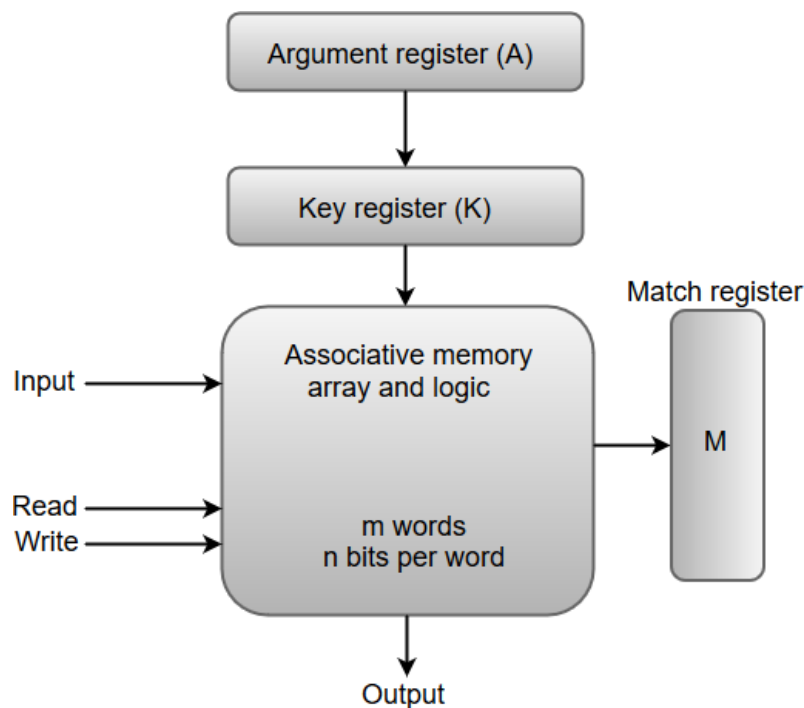
Associative Memory

An associative memory can be considered as a memory unit whose stored data can be identified for access by the content of the data itself rather than by an address or memory location.

Associative memory is often referred to as **Content Addressable Memory (CAM)**.

When a write operation is performed on associative memory, no address or memory location is given to the word. The memory itself is capable of finding an empty unused location to store the word.

On the other hand, when the word is to be read from an associative memory, the content of the word, or part of the word, is specified. The words which match the specified content are located by the memory and are marked for reading.

The following diagram shows the block representation of an Associative memory.

From the block diagram, we can say that an associative memory consists of a memory array and logic for 'm' words with 'n' bits per word.

The functional registers like the argument register **A** and key register **K** each have **n** bits, one for each bit of a word. The match register **M** consists of **m** bits, one for each memory word.

The words which are kept in the memory are compared in parallel with the content of the argument register.

The key register (K) provides a mask for choosing a particular field or key in the argument word. If the key register contains a binary value of all 1's, then the entire argument is compared with each memory word. 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 for identifying a piece of information which specifies how the reference to memory is made.

The following diagram can represent the relation between the memory array and the external registers in an associative memory.

**Associative memory of m word, n cells per word:**
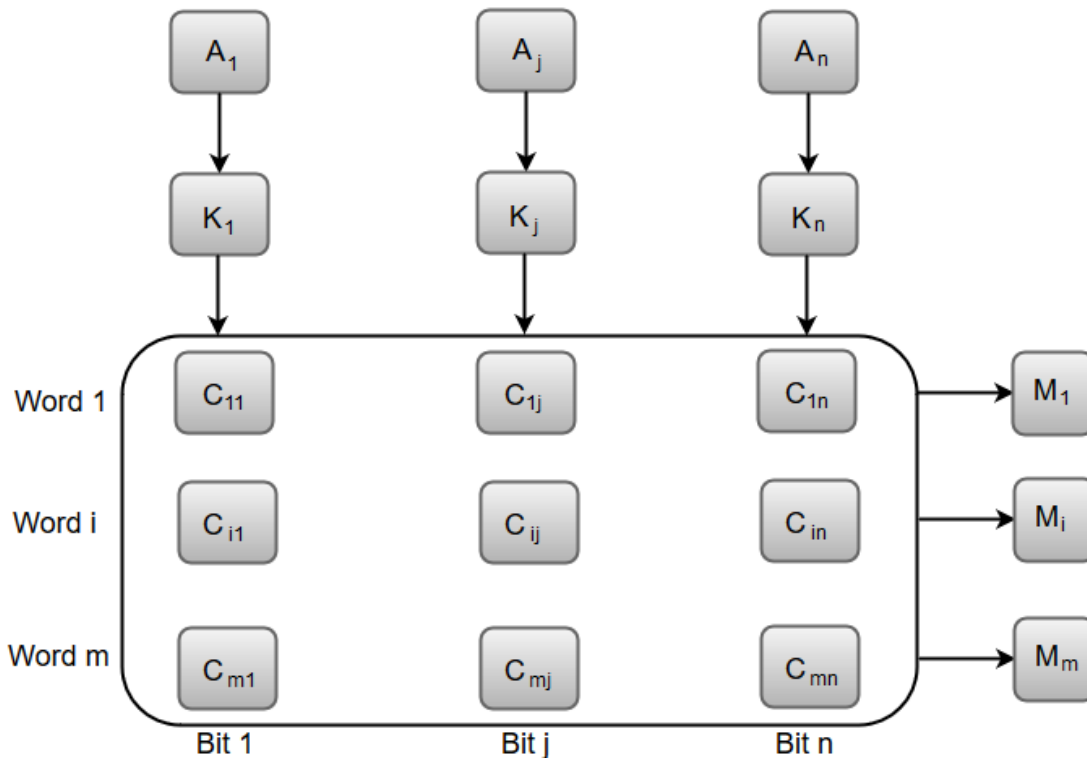
The cells present inside the memory array are marked by the letter C with two subscripts. The first subscript gives the word number and the second specifies the bit position in the word. For instance, the cell C_{ij} is the cell for bit **j** in word **i**.

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 process is done for all columns **j** = 1, 2, 3....., 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.

Cache Memory

The data or contents of the main memory that are used frequently by CPU are stored in the cache memory so that the processor can easily access that data in a shorter time. Whenever the CPU needs to access memory, it first checks the cache memory. If the data is not found in cache memory, then the CPU moves into the main memory.Cache memory is placed between the CPU and the main memory. The block diagram for a cache memory can be represented as:
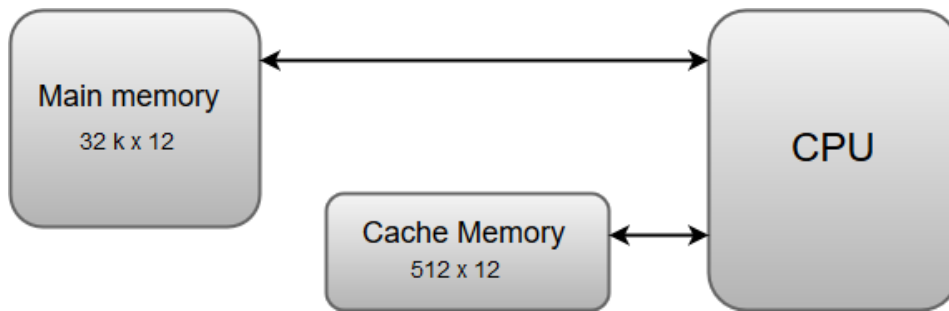
The cache is the fastest component in the memory hierarchy and approaches the speed of CPU components.

The basic operation of a cache memory 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 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.

- The performance of the 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 the 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.