SREENIVASA INSTITUTE OF TECHNOLOGY AND MANAGEMENT STUDIES.
(AUTONOMOUS)
MCA DEPARTMENT

LECTURE NOTES

COMPUTER ORGANIZATION AND ARCHITECTURE

UNIT-1

Digital Logic Circuits and Digital Components

The digital computer is a digital system that performs various computational tasks. The word digital implies that the information in the computer is represented by variables that take a limited number of discrete values. Digital computers use the binary number system, which has two digits: bit 0 and 1. A binary digit is called a bit. Information is represented in digital computers in groups of bits. By using various coding techniques, groups of bits can be made to represent not only binary numbers but also other discrete symbols, such as decimal digits or letters of the alphabets.

A computer system is sometimes subdivided into two functional entities: hardware and software. The hardware of the computer consists of all the electronic components and electromechanical devices that comprise the physical entity of the device. Computer software consists of the instructions and data that the computer manipulates to perform various data-processing tasks. A sequence of instructions for the computer is called a program. The data that are manipulated by the program constitute the data base.

The hardware of the computer is usually divided into three major parts as shown below, 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 (IOP) contains electronic circuits for communicating and controlling the transfer of information between the computer and the outside world. The input and output devices connected to the computer include keyboards, printers, terminals, magnetic disk drives, and other communication devices.

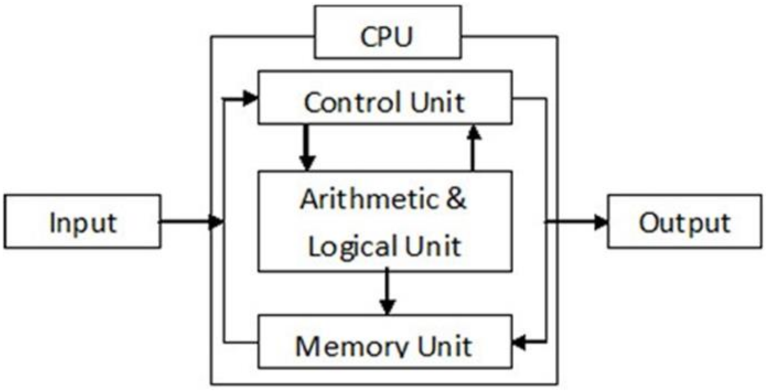


Fig. Block Diagram of Computer

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

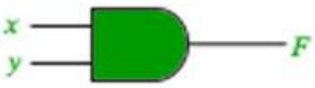
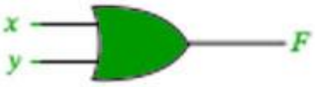
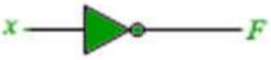
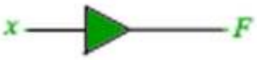
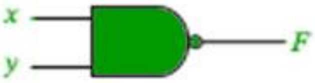
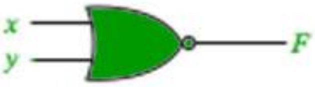
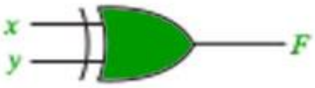
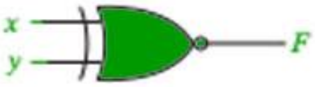
Computer design is concerned with the hardware design of the computer. Once the computer specifications are formulated, it is the task of the designer to develop hardware for the system. Computer design is concerned with the determination of what hardware should be used and how the parts should be connected.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

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

1.1 Logic Gates

Binary information is represented in digital computers by physical quantities called signals . Electrical signals such as voltages exist throughout the computer in either one of two recognizable states. The two states represent a binary variable that can be equal to 1 or 0. For example, a particular digital computer may employ a signal of 3 volts to represent binary 1 and 0.5 volt to represent binary 0. The input terminals of digital circuits accept binary signals of 3 and 0.5 volts and the circuits respond at the output terminals with signals of 3 and 0.5 volts to represent binary input and output corresponding to 1 and 0,respectively.

The manipulation of binary information is done by logic circuits called gates. Gates are blocks of hardware that produce signals of binary 1 or 0 when input logic requirements are satisfied. A variety of logic gates are commonly used in digital computer systems. Each gate has a distinct graphic symbol and its operation can be described by means of an algebraic expression. The input-output relationship of the binary variables for each gate can be represented in tabular form by a truth table.

Name	Graphic symbol	Algebraic function	Truth table
AND		$F = x \cdot y$	x y | F 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1
OR		$F = x + y$	x y | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1
Inverter		$F = x'$	x | F 0 | 1 1 | 0
Buffer		$F = x$	x | F 0 | 0 1 | 1
NAND		$F = (xy)'$	x y | F 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0
NOR		$F = (x + y)'$	x y | F 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0
Exclusive-OR (XOR)		$F = xy' + x'y$ $= x \oplus y$	x y | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 0
Exclusive-NOR or equivalence		$F = xy + x'y'$ $= (x \oplus y)'$	x y | F 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 1

1.2 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, andy. The three basic logic operations are AND, OR, and complement. A Boolean function can be expressed algebraically with binary variables, the logic operation symbols, parentheses, and equal sign. For a given value of the variables, the Boolean function can be either 1 or 0. Consider, for example, the Boolean function

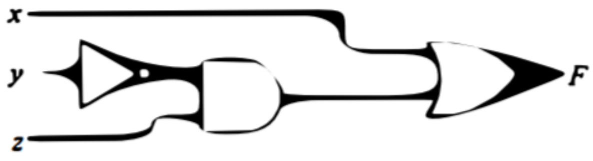
F = x + y'z

The function F is equal to 1 if x is 1 or if both y' and z are equal to I; F is equal to 0 otherwise. But saying that y' = 1 is equivalent to saying that y = 0 since y' is the complement of y. Therefore, we may say that F is equal to 1 if x = 1 or if yz = 01. The relationship between a function and its binary variables can be represented in a truth table. To represent a function in a truth table we need a list of the 2' combinations of then binary variables.

In the following fig (a) there are eight possible distinct combinations for assigning bits to the three variables x, y, and z. The function F is equal to 1 for those combinations where x = 1 or yz = 01; it is equal to 0 for all other combinations.

x	*y*	*z*	*F*
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

(a) Truth table



(b) Logic diagram

A Boolean function can be transformed from an algebraic expression into a logic diagram composed of AND, OR, and inverter gates. The logic diagram for F is shown in Fig. (b). There is an inverter for input y to generate its complement y'. There is an AND gate for the term y'z, and an OR gate is used to combine the two terms. In a logic diagram, the variables of the function are taken to be the inputs of the circuit, and the variable symbol of the function is taken as the output of the circuit. The purpose of Boolean algebra is to facilitate the analysis and design of digital circuits. It provides a convenient tool to:

1. Express in algebraic form a truth table relationship between binary variables.
2. Express in algebraic form the input-output relationship of logic diagrams.
3. Find simpler circuits for the same function.

A Boolean function specified by a truth table can be expressed algebraically in many different ways. By manipulating a Boolean expression according to Boolean algebra rules, one may obtain a simpler expression that will require fewer gates. To see how this is done, we must first study the manipulative capabilities of Boolean algebra.

1.2.1 Rule in Boolean Algebra

Following are the important rules used in Boolean algebra.

- Variable used can have only two values. Binary 1 for HIGH and Binary 0 for LOW.
- Complement of a variable is represented by an overbar. Thus, complement of variable B is represented as B. Thus if B = 0 then B=1 and B = 1 then B= 0.
- OR-ing of the variables is represented by a plus (+) sign between them. For example OR-ing of A, B, C is represented as A + B + C.
- Logical AND-ing of the two or more variable is represented by writing a dot between them such as A.B.C. Sometimes the dot may be omitted like ABC.

1.2.2 Laws of Boolean Algebra

Boolean Algebra Laws are used to simplify boolean expressions.

Basic Boolean Algebra Laws

1. Idempotent Law
    - A * A = A
    - A + A = A
2. Associative Law
    - (A * B) * C = A * (B * C)
    - (A + B) + C = A + (B + C)
3. Commutative Law
    - A * B = B * A
    - A + B = B + A
4. Distributive Law
    - A * (B + C) = A * B + A * C
    - A + (B * C) = (A + B) * (A + C)
5. Identity Law
    - A * 0 = 0   A * 1 = A
    - A + 1 = 1   A + 0 = A
6. Complement Law
    - A * ~A = 0
    - A + ~A = 1

7. Involution Law
    - ~(~A) = A
8. DeMorgan’s Law
    - ~(A * B) = ~A + ~B
    - ~(A + B) = ~A * ~B

Redundancy Laws

9. Absorption
    - A + (A * B) = A
    - A * (A + B) = A
10.
    - (A * B) + (A * ~B) = A
    - (A + B) * (A + ~B) = A
11.
    - A + (~A * B) = A + B
    - A * (~A + B) = A * B

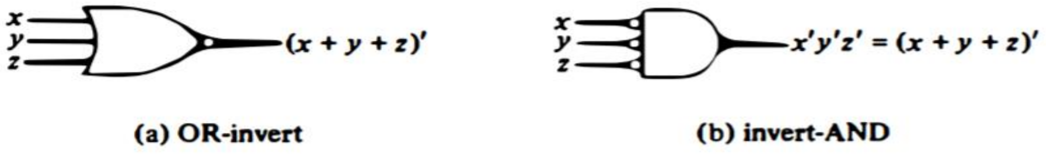
The purpose of Boolean algebra is to analysis and design of digital circuits, it provides a convenient tool to

A Boolean function specified by a truth table can be expressed algebraically in many different ways. By manipulating a Boolean expression according to Boolean algebra rules, one may obtain a simpler expression that will require fewer gates. To see how this is done, we must first study the manipulative capabilities of Boolean algebra.

1.2.3 DeMorgan's theorem

DeMorgan' s theorem is very important in dealing with NOR and NAND gates. It states that a NOR gate that performs the (x + y)' function is equivalent to the function x'y'. Similarly, a NAND function can be expressed by either (xy)' or (x' + y'). For this reason the NOR and NAND gates have two distinct graphic symbols, as shown in Figs. 1-4 and 1-5. Instead of representing a NOR gate with an OR graphic symbol followed by a circle, we can represent it by an AND graphic symbol preceded by circles in all inputs. The invert-AND symbol for the NOR gate follows from DeMorgan's theorem and from the convention that small circles denote complementation. Similarly, the NAND gate has two distinct symbols, as shown in Fig. 1-5.

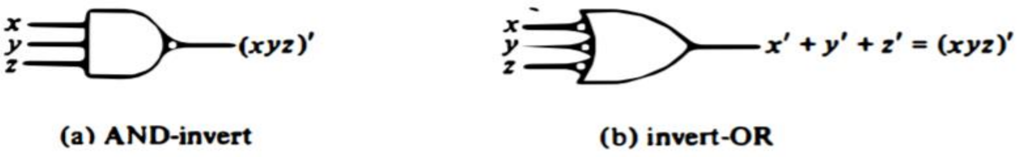
Figure 1-4 Two graphic symbols for NOR gate.



(a) OR-invert

(b) invert-AND

Figure 1-5 Two graphic symbols for NAND gate.



(a) AND-invert

(b) invert-OR

1.2.4 Complement of a Function F

The complement of a function F when expressed in a truth table is obtained by interchanging l's and D's in the values of F in the truth table. When the function is expressed in algebraic form, the complement of the function can be derived by means of DeMorgan's theorem. The general form of DeMorgan's theorem can be expressed as follows:

$$(X1+ X2 + X3 + \cdot \cdot \cdot + Xn)' = X1' X2' X3' \cdots Xn'$$

$$(X1 X2X3 \cdot \cdot \cdot Xn)' = X1' + X2' + X3' + \cdots + Xn'$$

As an example, consider the following expression and its complement:

$$F = AB + C'D' + B'D \quad F' = (A' + B')(C + D)(B + D')$$

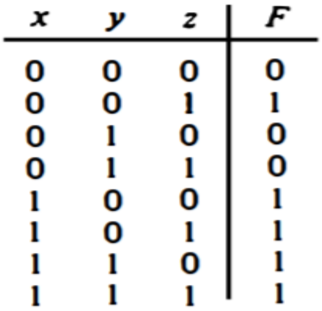
1.3 Map Simplification

The complexity of the logic diagram that implements a Boolean function is related directly to the complexity of the algebraic expression from which the function is implemented. The truth table representation of a function is unique, but the function can appear in many different forms when expressed algebraically. The expression may be simplified using the basic relations of Boolean algebra. However, this procedure is sometimes difficult because it lacks specific rules for predicting each succeeding step in the manipulative process. The map method provides a simple, straightforward procedure for simplifying Boolean expressions. This method may be regarded as a pictorial arrangement of the truth table which allows an easy interpretation for choosing the minimum number of terms needed to express the function algebraically. The map method is also known as the **Karnaugh- Map or K-Map**.

Each combination of the variables in a truth table is called a minterm. When expressed in a truth table a function of n variables will have 2" minterms, equivalent to the 2" binary numbers obtained from n bits. A Boolean function is equal to 1 for some minterms and to 0 for others. The information contained in a truth table may be expressed in compact form by listing the decimal equivalent of those minterms that produce a 1 for the function.

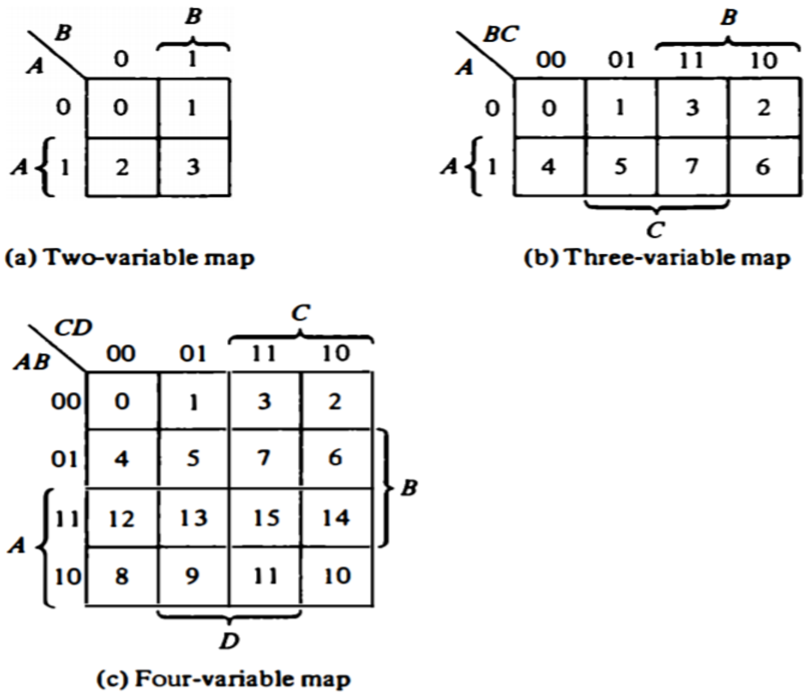
For example the below truth table can be expressed as

F(x, y, z) = ∑(1, 4, 5, 6, 7)



x	y	z	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

The maps for functions of two, three, and four variables are shown in Fig. 1-7. The number of squares in a map of n variables is 2". The 2" minterms are listed by an equivalent decimal number for easy reference. The minterm numbers are assigned in an orderly arrangement such that adjacent squares represent minterms that differ by only one variable. The variable names are listed across both sides of the diagonal line in the corner of the map. The O's and 1' s marked along each row and each column designate the value of the variables.



(a) Two-variable map

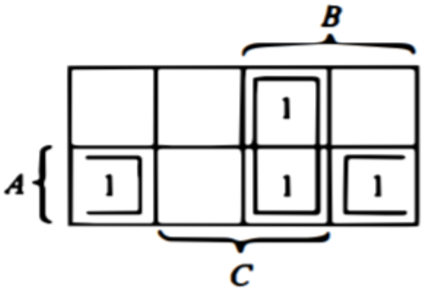
(b) Three-variable map

(c) Four-variable map

1.3.1 Sum-of-products form

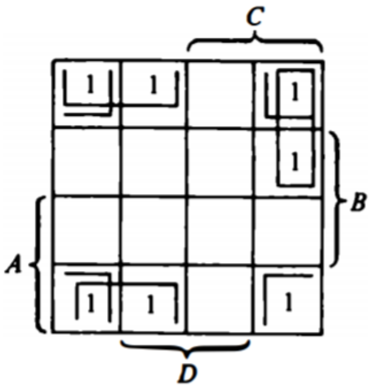
A Boolean function represented by a truth table is plotted into the map by inserting 1's in those squares where the function is I. The squares containing 1's are combined in groups of adjacent squares. These groups must contain a number of squares that is an integral power of 2. Groups of combined adjacent 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. The following examples show the use of the map for simplifying Boolean functions. In the first example we will simplify the Boolean function

F(A, B, C) =∑ (3, 4, 6, 7)



The simplified algebraic expression for the function is the OR of the two terms: F = BC + AC'

four-variable map. F(A, B, C, D) =∑ (0, 1, 2, 6, 8, 9, 10)

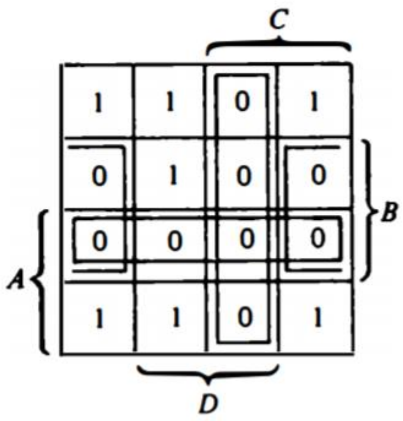


the simplified function is : F = B 'D' + B 'C' + A'CD’

1.3.2 Product of Sums Simplification

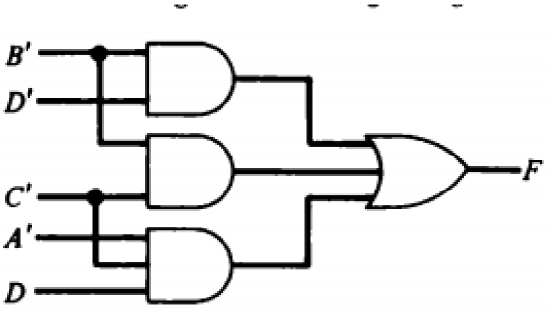
If the squares marked with 0' s are combined, as shown in the diagram, we obtain the simplified complemented function: F' = AB +CD + BD' Taking the complement of F', we obtain the simplified function in product-of sums form: F = (A' + B ')(C' + D')(B ' + D)

Map for $F(A, B, C, D) = \Sigma(0,1,2,5,8,9,10)$.

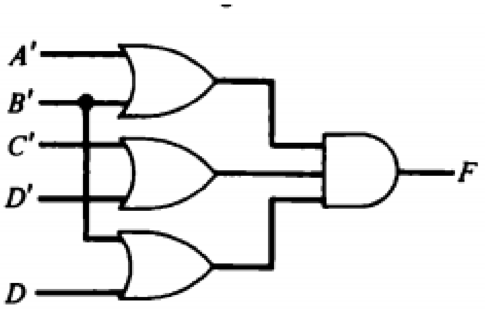


1	1	0	1
0	1	0	0
0	0	0	0
1	1	0	1

The Logic diagram for the **Sum-of–Products** and **Product-of-Sums** is shown in the following diagram.

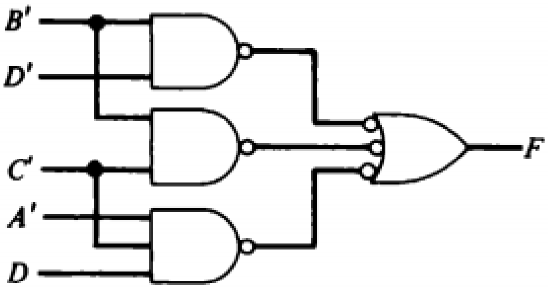


(a) Sum of products:
$F = B'D' + B'C' + A'C'D$

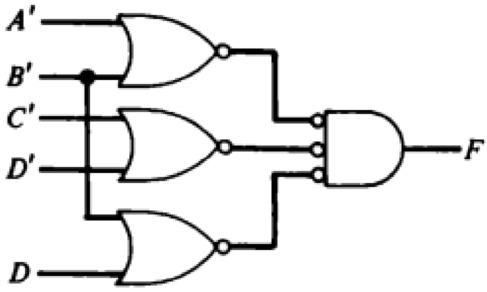


(b) Product of sums:
$F = (A' + B')\,(C' + D')\,(B' + D)$

Logic diagram using NAND and NOR gates are



(a) With NAND gates



(b) With NOR gates

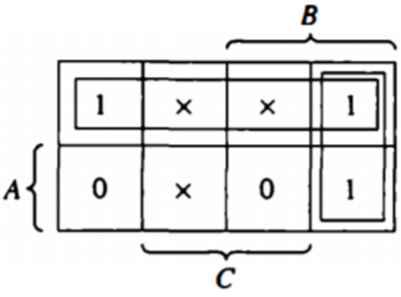
1.3.3 Don't-Care Conditions

The 1' s and 0's in the map represent the min terms that make the function equal to 1 or 0. There are occasions when it does not matter if the function produces 0 or 1 for a given minterm. Since the function may be either 0 or 1, we say that we don't care what the function output is to be for this min term. Min terms that may produce either 0 or 1 for the function are said to be don't-care conditions and are marked with an x in the map. These don't-care conditions can be used to provide further simplification of the algebraic expression.

As an example, consider the following Boolean function together with the don't-care minterms:

F(A, B, C) = ∑(0, 2, 6)

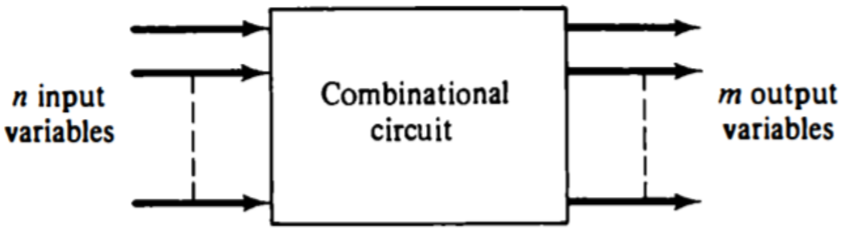
d(A, B, C) = ∑ (1, 3, 5)



The simplified expression is F = A' + BC'

1.4 Combinational Circuits

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 combination of the inputs. A block diagram of a combinational circuit is shown in the following Figure. The n binary input variables come from an external source, the m binary output variables go to an external destination, and in between there is an interconnection of logic gates.



A combinational circuit can be described by a truth table showing the binary relationship between the n input variables and the m output variables. The truth table lists the corresponding output binary values for each of the 2" input combinations. A combinational circuit can also be specified with m Boolean functions, one for each output variable. Each output function is expressed in terms of the n input variables.

The design of combinational circuits starts from the verbal outline of the problem and ends in a logic circuit diagram. The procedure involves the following steps:

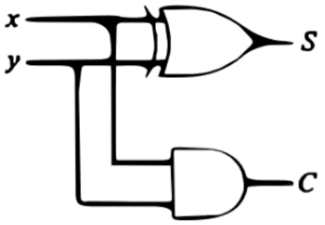
1. The problem is stated.
2. The input and output variables are assigned letter symbols.
3. The truth table that defines the relationship between inputs and outputs is derived.
4. The simplified Boolean functions for each output are obtained.
5. The logic diagram is drawn.

1.4.1 Half-Adder

The most basic digital arithmetic circuit is the addition of two binary digits. A combinational circuit that performs the arithmetic addition of two bits is called a half-adder. One that performs the addition of three bits (two significant bits and a previous carry) is called a full-adder. The name of the former stems from the fact that two half-adders are needed to implement a full-adder. The input variables of a half-adder are called the augend and addend bits. The output variables the sum and carry. It is necessary to specify two output variables because the sum of 1 + 1 is binary 10, which has two digits. We assign symbols x and y to the two input variables, and S (for sum) and C (for carry) to the two output variables. The truth table for the half-adder is shown in Fig. l-16(a). The C output is 0 unless both inputs are I. The S output represents the least significant bit of the sum. The Boolean functions for the two outputs can be obtained directly from the truth table:

x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

(a) Truth table



(b) Logic diagram

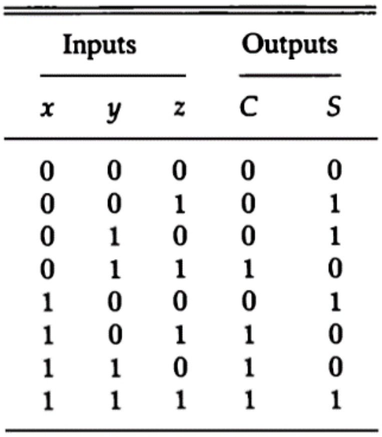
$$S = x'y + xy' = x \oplus y$$

$$C = xy$$

The logic diagram is shown in Fig. l-16(b). It consists of an exclusive-OR gate and an AND gate.

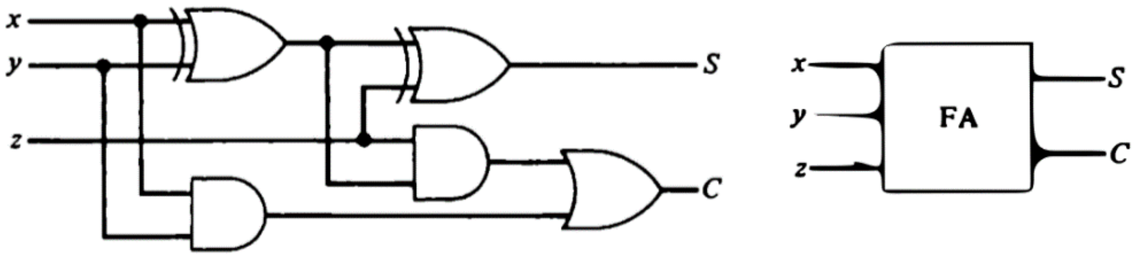
1.4.2 Full-Adder

A full-adder is a combinational circuit that forms the arithmetic sum of three input bits. It consists of three inputs and two outputs. Two of the input variables, denoted by x and y, represent the two significant bits to be added. The third input, z, represents the carry from the previous lower significant position. Two outputs are necessary because the arithmetic sum of three binary digits ranges in value from 0 to 3, and binary 2 or 3 needs two digits. The two outputs are designated by the symbols S (for sum) and C (for carry). The binary variable S gives the value of the least significant bit of the sum. The binary variable C gives the output carry. The truth table of the full-adder is shown in the following Table.



Inputs			Outputs	
x	*y*	*z*	*C*	*S*
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

The eight rows under the input variables designate all possible combinations that the binary variables may have. The value of the output variables are determined from the arithmetic sum of the input bits. When all input bits are 0, the output is 0. The S output is equal to 1 when only one input is equal to 1 or when all three inputs are equal to I. The C output has a carry of 1 if two or three inputs are equal to I.



(a) Logic diagram (b) Block diagram

The logic diagram of the full-adder is drawn in the above figure. Note that the fulladder circuit consists of two half-adders and an OR gate. Realizing that x'y + xy' = xa1y and including the expression for output S, we obtain the two Boolean expressions for the full-adder:

$$\mathbf{S = x \oplus y \oplus z}$$

$$\mathbf{C = xy + (x \oplus y)z}$$

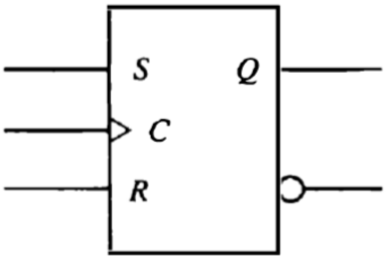
1.5 Flip-Flops

The digital circuits considered thus far have been combinational, where the outputs at any given time are entirely dependent on the inputs that are present at that time. Although every digital system is likely to have a combinational circuit, most systems encountered in practice also include storage elements, which require that the system be described in terms of sequential circuits. The most common type of sequential circuit is the synchronous type.

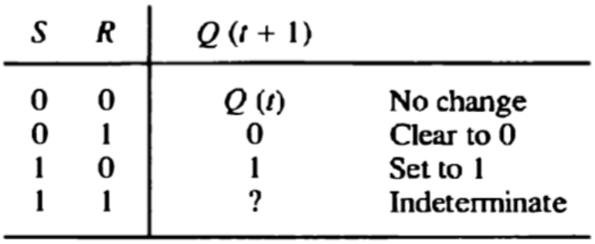
Synchronization is achieved by a timing device called a clock pulse generator that produces a periodic train of clock pulses. The storage elements employed in clocked sequential circuits are called flip-flops. A flip-flop is a binary cell capable of storing one bit of information. It has two outputs, one for the normal value and one for the complement value of the bit stored in it. A flip-flop maintains a binary state until directed by a clock pulse to switch states. The difference among various types of flip-flops is in the number of inputs they possess and in the manner in which the inputs affect the binary state. The most common types of flip-flops are presented below.

1.5.1. SR Flip-Flop

The graphic symbol of the SR flip-flop is shown in Figure. It has three inputs, labelled S (for set), R (for reset), and C (for clock). It has an output Q and sometimes the flip-flop has a complemented output, which is indicated with a small circle at the other output terminal. There is an arrowhead-shaped symbol in front of the letter C to designate a dynamic input. The dynamic indicator symbol denotes the fact that the flip-flop responds to a positive transition (from 0 to 1) of the input clock signal.



(a) Graphic symbol



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

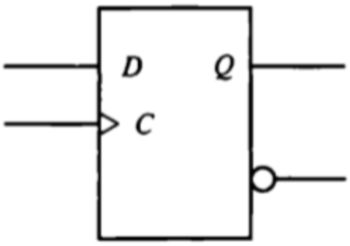
(b) Characteristic table

The operation of the SR flip-flop is as follows. 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 signal 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 from 0 to 1, output Q is set to 1. If S = 0 and R = 1 when C changes from 0 to 1, output Q is cleared to 0. If both S and R are 0 during the clock transition, the output does not change. When both 5 and R are equal to 1, the output is unpredictable and may go to either 0 or 1, depending on internal timing delays that occur within the circuit.

The characteristic table shown above summarizes the operation of the SR flip-flop in tabular formThe S and R columns give the binary values of the two inputs. Q(t) is the binary state of the Q output at a given time (referred to as present state). Q(t + 1) is the binary state of the Q output after the occurrence of a clock transition (referred to as next state).

1.5.2. D Flip-Flop

The D (data) flip-flop is a slight modification of the SR flip-flop. An SR flip-flop is converted to a D 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 graphic symbol and characteristic table of the D flip-flop are shown in Fig. 1-20. From the characteristic table we note that 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



(a) Graphic symbol

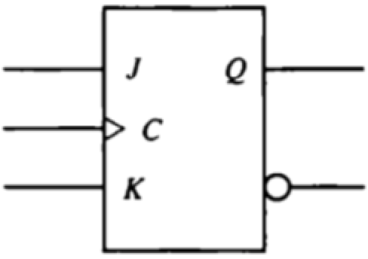
D	$Q(t+1)$	
0	0	Clear to 0
1	1	Set to 1

(b) Characteristic table

Although a D flip-flop has the advantage of having only one input (excluding C), it has the disadvantage that its characteristic table does not have a "no change" condition Q(t + 1) = Q(t). The "no change" condition can be accomplished either by disabling the clock signal or by feeding the output back into the input, so that clock pulses keep the state of the flip-flop unchanged.

1.5.3 JK- Flip-Flop

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. The graphic symbol and characteristic table of the JK flip-flop are shown in following figure. The J input is equivalent to the S (set) input of the SR flip-flop, and the K input is equivalent to the R (clear) input. Instead of the indeterminate condition, the JK flip-flop has a complement condition Q(t + 1) = Q'(t) when both J and K are equal to 1.



(a) Graphic symbol

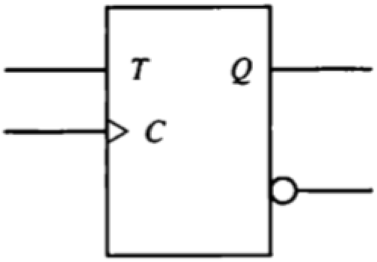
J	K	$Q(t+1)$	
0	0	$Q(t)$	No change
0	1	0	Clear to 0
1	0	1	Set to 1
1	1	$Q'(t)$	Complement

(b) Characteristic table

1.5.4 T-Flip-Flop

Another type of flip-flop found in textbooks is the T (toggle) flip-flop. This flip-flop, shown in Fig. 1-22, is obtained from a JK type when inputs J and K are connected to provide a single input designated by T. The T flip-floptherefore has only two conditions. When T = 0 (J = K = 0) a clock transition does not change the state of the flip-flop. When T = 1 (J = K = 1) a clock transition complements the state of the flip-flop. These conditions can be expressed by a characteristic equation:

$$Q(t + 1) = Q(t) \oplus T$$



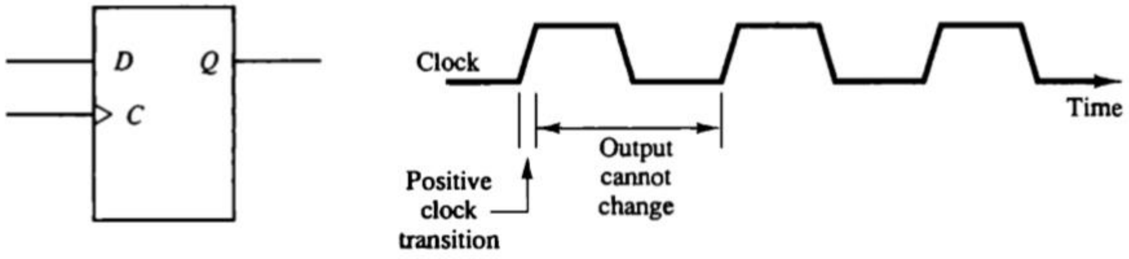
(a) Graphic symbol

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

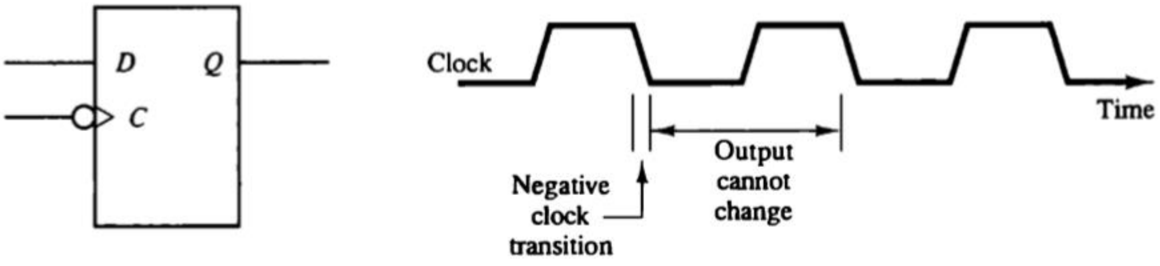
(b) Characteristic table

1.5.5 Edge-Triggered Flip-Flops

The most common type of flip-flop used to synchronize the state change during a clock pulse transition is the edge-triggered flip-flop. In this type of flip-flop, output transitions occur at a specific level of the clock pulse. When the pulse input level exceeds this threshold level, the inputs are locked out so that the flip-flop is unresponsive to further changes in inputs until the clock pulse returns to 0 and another pulse occurs. Some edge-triggered flip-flops cause a transition on the rising edge of the clock signal (positive-edge transition), and others cause a transition on the falling edge (negative-edge transition).



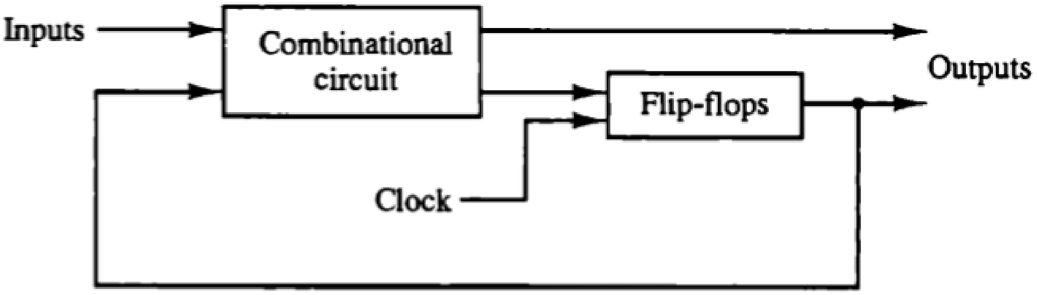
(a) Positive-edge-triggered D flip-flop.



(b) Negative-edge-triggered D flip-flop.

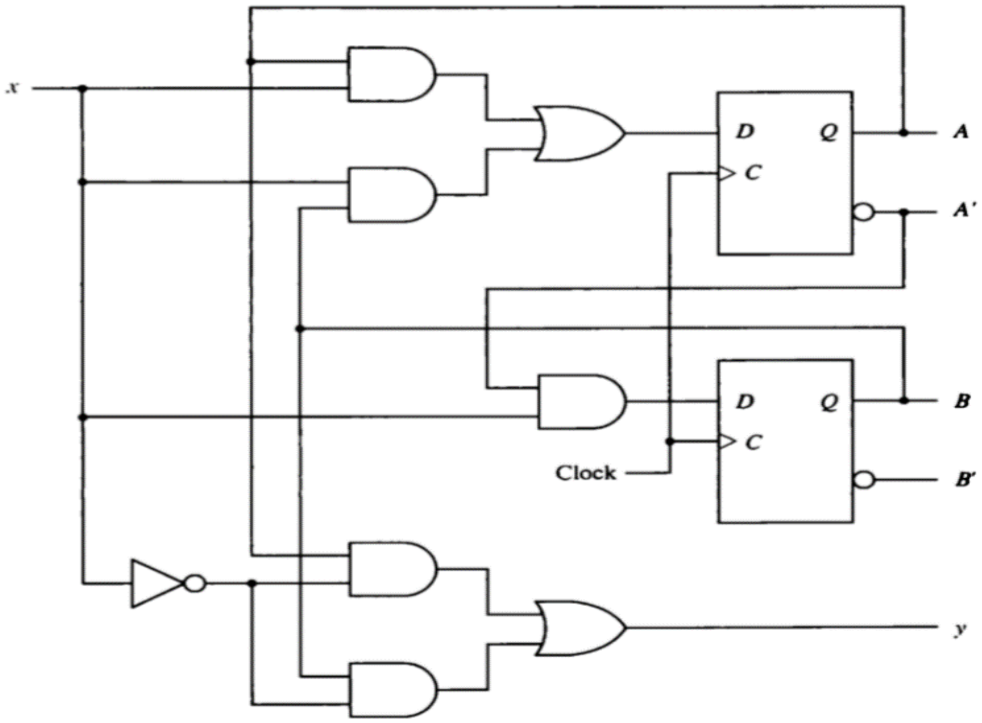
1.6 Sequential Circuits

A sequential circuit is an interconnection of flip-flops and gates. The gates by themselves constitute a combinational circuit, but when included with the flip-flops, the overall circuit is classified as a sequential circuit. The block diagram of a clocked sequential circuit is shown in following figure. It consists of a combinational circuit and a number of clocked flip-flops. In general, any number or type of flip-flops may be included. As shown in the diagram,



The combinational circuit block receives binary signals from external inputs and from the outputs of flip-flops. The outputs of the combinational circuit go to external outputs and to inputs of flip-flops. The gates in the combinational circuit determine the binary value to be stored in the flip-flops after each clock transition. The outputs of flip-flops, in turn, are applied to the combinational circuit inputs and determine the circuit's behavior. This process demonstrates that the external outputs of a sequential circuit are functions of both external inputs and the present state of the flip-flops. Moreover, the next state of flip-flops is also a function of their present state and external inputs. Thus a sequential circuit is specified by a time sequence of external inputs, external outputs, and internal flip-flop binary states.

An example of a sequential circuit is shown in Figure



DIGITAL COMPONENTS

1.7 Integrated Circuits

Digital circuits are constructed with integrated circuits. An integrated circuit (abbreviated IC) is a small silicon semiconductor crystal called chip containing the electronic components for the digital gates. The various gates are in interconnected inside the chip to form the required circuit. The chip is mounted on a ceramic or plastic container, and connections are welded by thin gold wines to external pins to form the integrated circuit. The number of pins may range from 14 in a small IC package to 100 or more in a larger package. Each IC has a numeric designation printed on the surface of the package for identification.

As the technology of ICs has improved, the number of gates that can be put in a single chip has increased considerably.

Small-Scale integration (SSI) devices contain several independent gates in a single package. The inputs and outputs of the gates are connected directly to the pins in the package. The number of gates is usually less than 10 and is limited by the number of pins available in the IC.

Medium-scale integration (MSI) devices have a complexity of approximately 10 to 200 gates in a single package. They usually perform specific elementary digital functions such as decoders, adders, and registers.

Large-scale integration (LSI) devices contain between 200 and a few thousand gates in a single package. They include digital systems, such as processors, memory chips, and programmable modules. **Very-large-scale integration (VLSI)** devices contain thousands of gates within a single package. Examples are large memory arrays and complex microcomputer chips. Because of their small size and low cost, VLSI devices have revolutionized the computer system design technology, giving designers the capability to create structures that previously were not economical.

Digital integrated circuits are classified not only by their logic operation but also by digital logic family. Each logic family has its own basic electronic circuit upon which more complex digital circuits and functions are developed. The basic circuit in each technology is either a NAND, a NOR, or an inverter gate.Many different logic families of integrated circuits have been introduced commercially. The following are the most popular.

- TTL- Transistor-transistor logic
- ECL - Emitter-coupled logic
- MOS - Metal-oxide semiconductor
- CMOS - Complementary metal-oxide semiconductor

The **transistor-transistor logic** family was an evolution of a previous technology that used diodes and transistors for the basic NAND gate. This technology was called DTL, for "diode-transistor logic." Later the diodes were replaced by transistors to improve the circuit operation and the name of the logic family was changed to "transistor-transistor logic." This is the reason for mentioning the word "transistor" twice.

The **emitter-coupled logic** (ECL) family provides the highest-speed digital circuits in integrated form. ECL is used in systems such as supercomputers and signal processors where high speed is essential

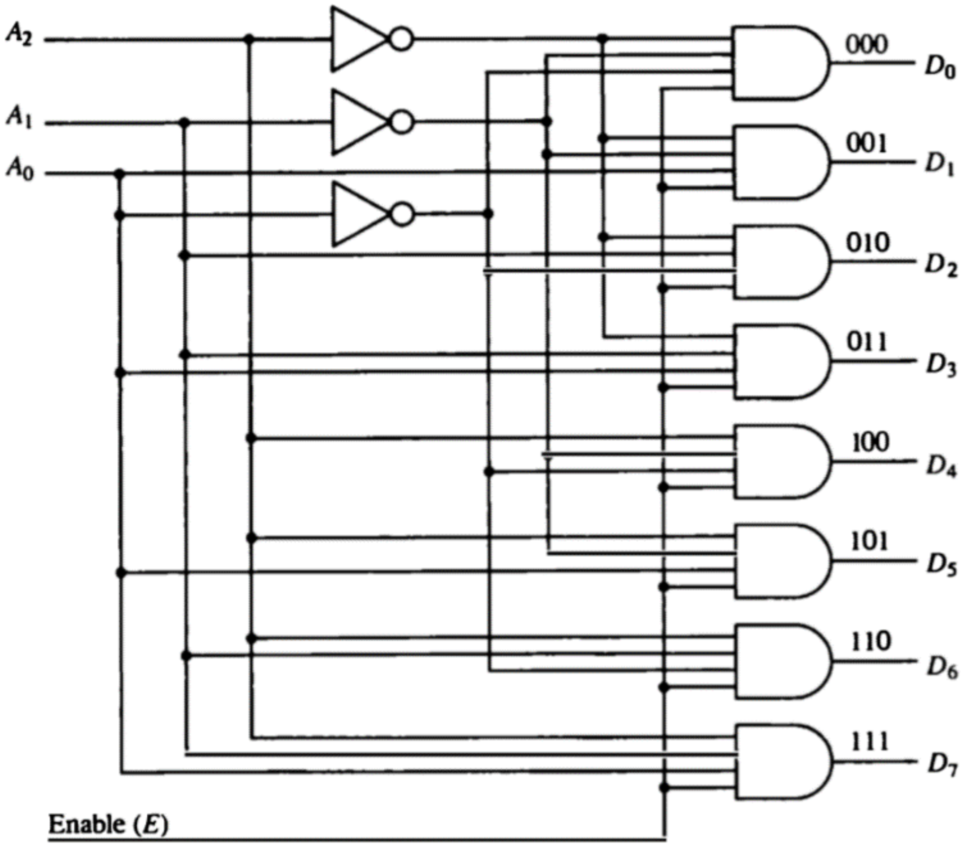
The **metal-oxide semiconductor** (MOS) is a unipolar transistor that depends on the flow of only one type of carrier, which may be electrons (n-channel) or holes (p-channel). This is in contrast to the bipolar transistor used in TIL and ECL gates, where both carriers exist during normal operation.

The **complementary MOS** (CMOS) technology uses PMOS and NMOS transistors connected in a complementary fashion in all circuits. The most important advantages of CMOS over bipolar are the high packing density of circuits, a simpler processing technique during fabrication, and a more economical operation because of low power consumption.

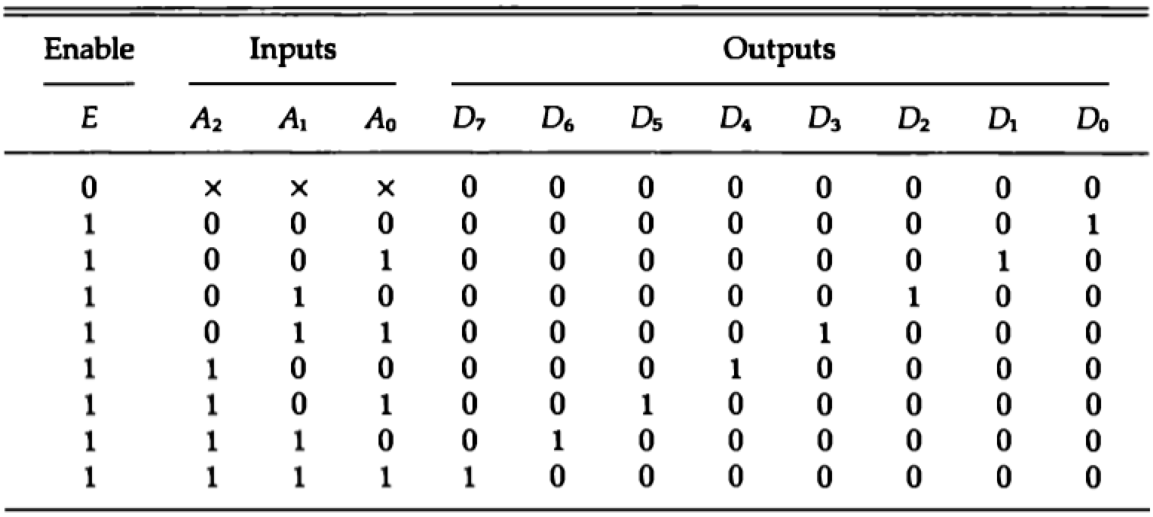
1.8 Decoders

Discrete quantities of information are represented in digital computers with binary codes. A binary code of n bits is capable of representing up to 2^n 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 2^n unique outputs. If the n-bit coded information has unused bit combinations, the decoder may have less than 2^n outputs. The decoders presented in this section are called n-to-m-line decoders, where $m <=2^n$. Their purpose is to generate the 2^n (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 in Figure.

The three data inputs, A_0, A_1, and A_n are decoded into eight outputs, each outputrepresenting 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.



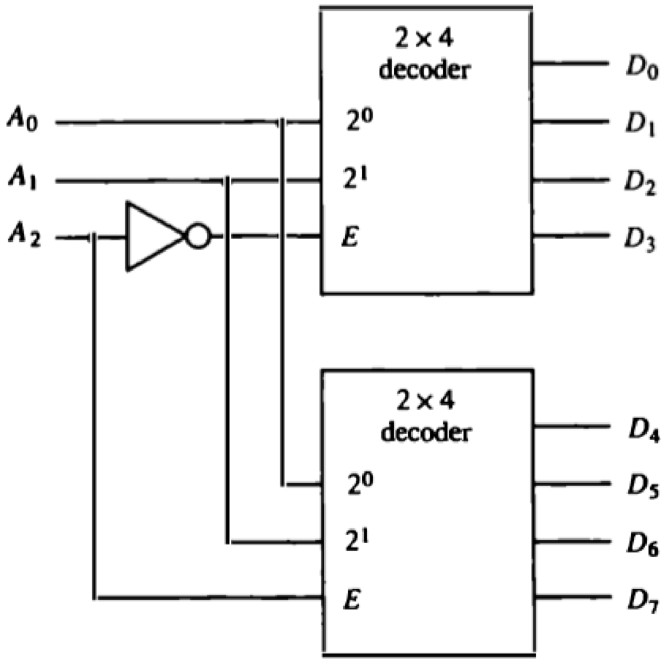
Commercial decoders include one or more enable inputs to control the operation of the circuit. The decoder of the above Figure has one enable input, E. The decoder is enabled when E is equal to 1 and disabled when E is equal to 0. The operation of the decoder can be clarified using the truth table listed in the following 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.



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	×	×	×	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

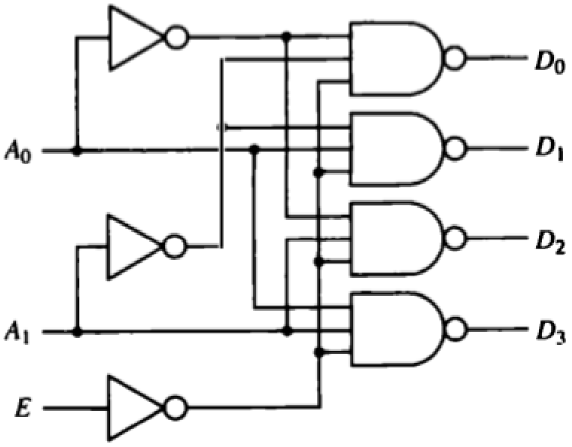
1.8.1 Decoder Expansion

There are occasions when a certain-size decoder is needed but only smaller sizes are available. When this occurs it is possible to combine two or more decoders with enable inputs to form a larger decoder. Thus if a 6-to-64-line decoder is needed, it is possible to construct it with four 4-to-16-line decoders. The following figure shows how decoders with enable inputs can be connected to form a larger decoder. 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 Olevel. When A, = 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 Do through D3, depending on the values of A1 and A0 (while A, = 0). When A, = 1, the lower decoder is enabled and the upper is disabled. The lower decoder output generates the binary equivalent D4 through D, since these binary numbers have a 1 in the A, position. The example demonstrates the usefulness of the enable input in decoders or any other combinational logic component. Enable inputs are a convenient feature for interconnecting two or more circuits for the purpose of expanding the digital component into a similar function but with more inputs and outputs.

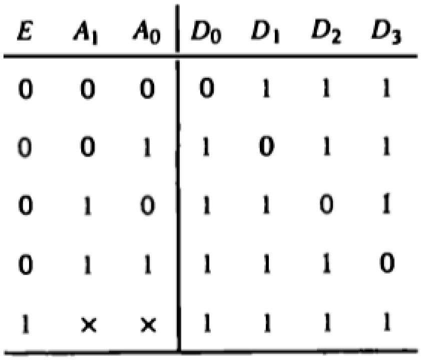


1.8.2 NAND Gate Decoder

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 A1 and A,. The circuit is disabled when E is equal to 1, regardless of the values of the other two inputs.



(a) Logic diagram



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	×	×	1	1	1	1

(b) Truth table

When the circuit is disabled, none of the outputs are selected and all outputs are equal to 1. In general, a decoder may operate with complemented or uncomplemented outputs. The enable input may be activated with a 0 or with a 1 signal level.

1.9 Encoders

An encoder is a digital circuit that performs the inverse operation of a decoder. An encoder has 2" (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 in the following Table. It has eight inputs, one for each of the octal digits, and three outputs that generate the corresponding binary number. It is assumed that only one input has a value of 1 at any given time; otherwise, the circuit has no meaning.

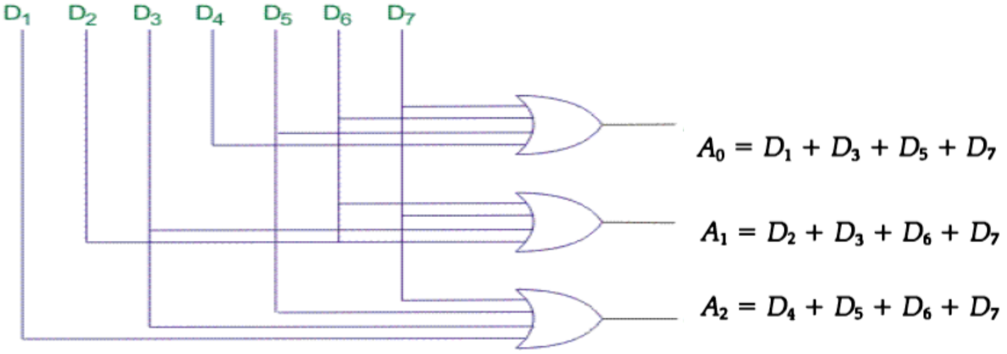
The encoder can be implemented with OR gates whose inputs are determined directly from the truth table. Output A_0 = 1 if the input octal digit is 1 or 3 or 5 or 7. Similar conditions apply for the other two outputs. These conditions can be expressed by the following Boolean functions:

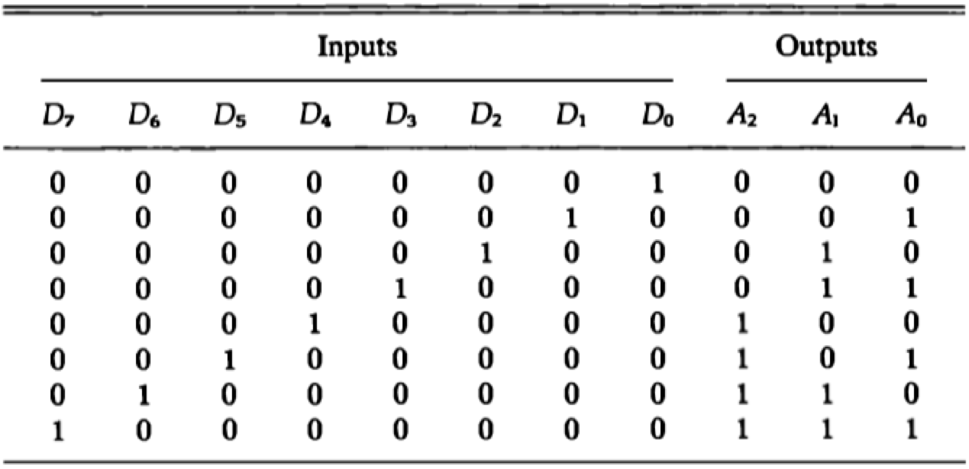
$A_0= D_1 + D_3+D_5+D_7$

$A_1 = D_2 + D_3 + D_6 + D_7$

$A_2 = D_4 + D_5 + D_6 + D_7$

The encoder can be implemented with three OR gates.


D_1
D_2
D_3
D_4
D_5
D_6
D_7
$A_0 = D_1 + D_3 + D_5 + D_7$
$A_1 = D_2 + D_3 + D_6 + D_7$
$A_2 = D_4 + D_5 + D_6 + D_7$



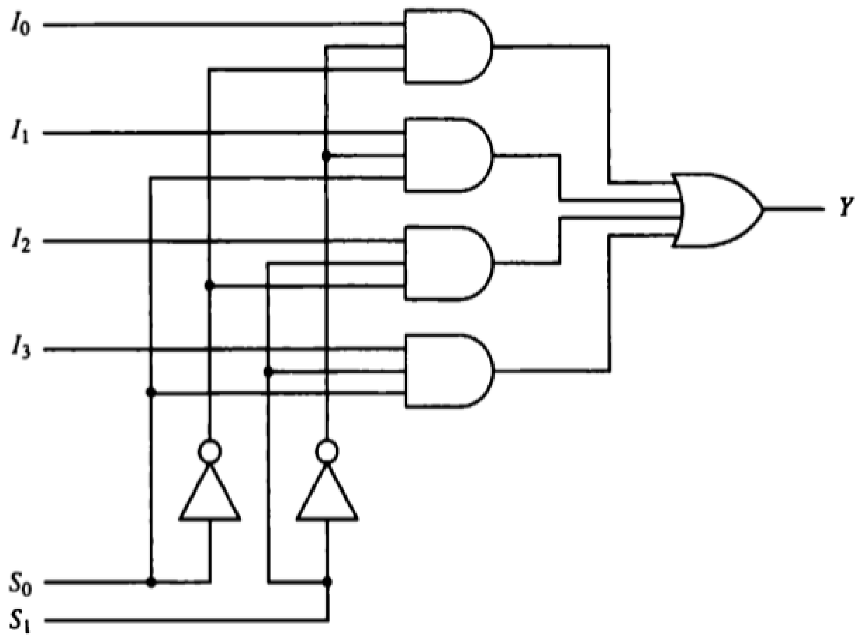
Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

1.10 Multiplexers

A multiplexer is a combinational circuit that receives binary information from one of 2^n 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 2^n-to-1 multiplexer has 2^n input data lines and n input selection lines whose bit combinations determine which input data are selected for the output. A4-to-1-line multiplexer is shown in the following Figure. Each of the four data inputs I_0 through I_1 is applied to one input of an AND gate.

The two selection inputs S_1 and S_0 are decoded to select a particular AND gate. The outputs of the AND gates are applied to a single OR gate to provide the single output. To demonstrate the circuit operation, consider the case when S_1S_0 = 10. The AND gate associated with input I_1 has two of its inputs equal to 1. The third input of the gate is connected to I_2. The other three AND gates have at least one input equal to 0, which makes their outputs equal to 0. The OR gate output is now equal to the value of I_2, thus providing a path from the selected input to the output.

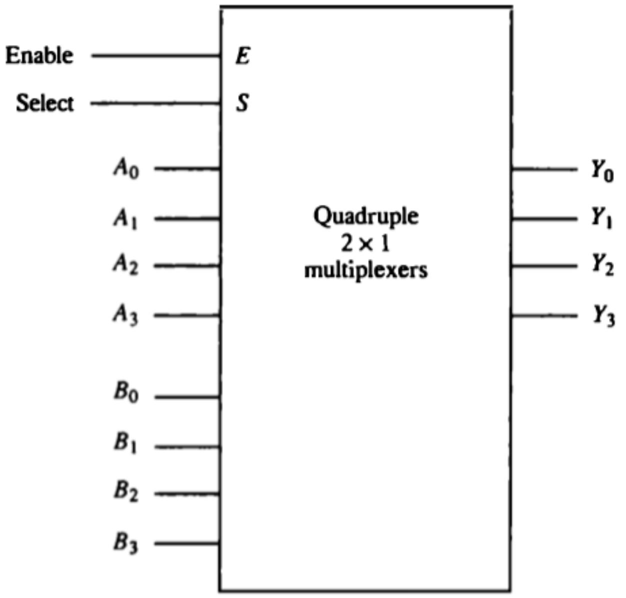
The 4-to-1 line multiplexer of Figure has six inputs and one output. A truth table describing the circuit needs 64 rows since six input variables can have 2^6 binary combinations. A more convenient way to describe the operation of multiplexers is by means of a function table. The function table for the multiplexer is shown in the following 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.When the selection inputs are equal to 00, output Y is equal to input $I_{0.}$ When the selection inputs are equal to 01, input11 has a path to output Y, and similarly for the other two combinations. The multiplexer is also called a data selector, since it selects one of many data inputs and steers the binary information to the output.



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

The AND gates and inverters in the multiplexer resemble a decoder circuit, and indeed they decode the input selection lines. In general, a 2"-to-1line multiplexer is constructed from an n-to-2" decoder by adding to it 2" input lines, one from each data input. The size of the multiplexer is specified by the number 2" of its data inputs and the single output. It is then implied that it also contains n input selection lines. The multiplexer is often abbreviated as MUX. As in decoders, multiplexers may have an enable input to control the operation of the unit. When the enable input is in the inactive state, the outputs are disabled, and when it is in the active state, the circuit functions as a normal multiplexer. The enable input is useful for expanding two or more multiplexers to a multiplexer with a larger number of inputs. In some cases two or more multiplexers are enclosed within a single integrated circuit package. The selection and the enable inputs in multiple-unit construction are usually common to all multiplexers.



(a) Block diagram

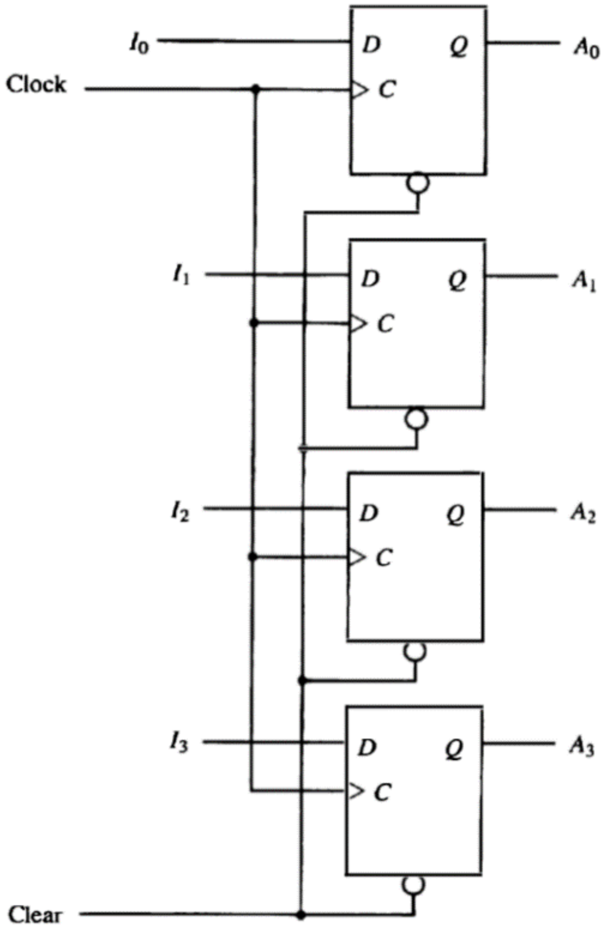
E	*S*	*Y*
0	×	All 0's
1	0	*A*
1	1	*B*

(b) Function table

1.11 Registers

A register is a group of flip-flops with each flip-flop 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 the flip-flops, a register may have combinational gates that perform certain data-processing tasks. In its broadest definition, a register consists of a group of flip-flops and gates that effect their transition. The flip-flops hold the binary information and the gates control when and how new information is transferred into the register.

The simplest register is one that consists only of flip-flops, with no external gates. The following figure shows such a register constructed with four D flip-flops. The common clock input triggers all flip-flops on the rising edge of each pulse, and the binary data available at the four inputs are transferred into the 4-bit register. The four outputs can be sampled at any time to obtain the binary information stored in the register. The clear input goes to a special terminal in each flip-flop. When this input goes to 0, all flip-flops are reset asynchronously. The clear input is useful for clearing the register to all 0's prior to its clocked operation. The clear input must be maintained at logic 1 during normal clocked operation. Note that the clock signal enables the D input but that the clear input is independent of the clock. The transfer of new information into a register is referred to as loading the register.



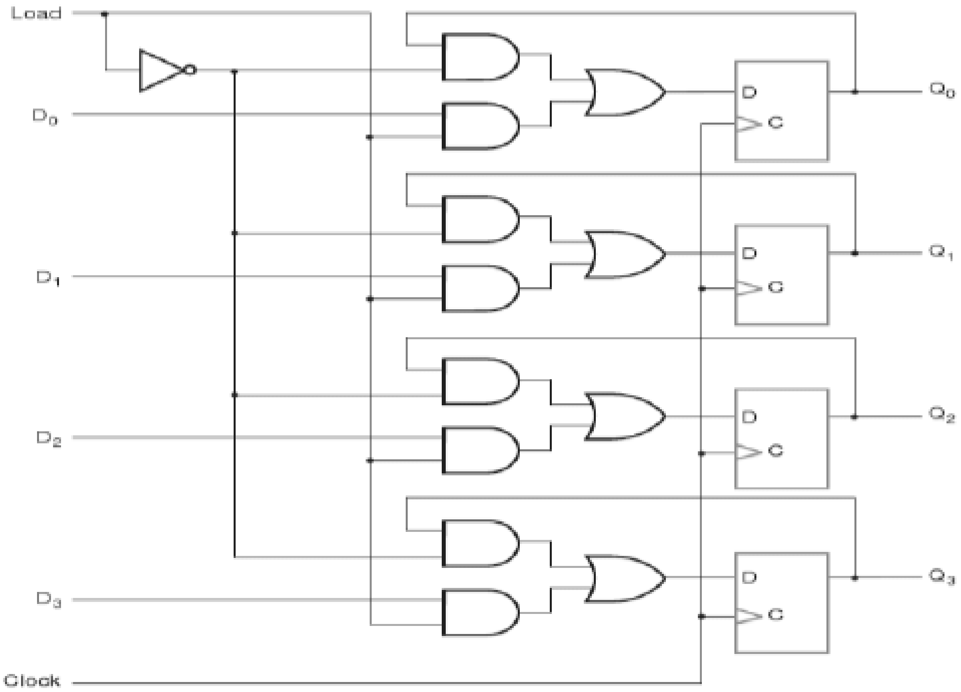
If all the bits of the register are loaded simultaneously with a commonclock pulse transition, we say that the loading is done in parallel. A clock transition applied to the C inputs of the register of Figure will load all four inputs I_0 through I_3 in parallel. In this configuration, the clock must be inhibited from the circuit if the content of the register must be left unchanged.

1.11.1 Register with Parallel Load

Most digital systems have a master clock generator that supplies a continuous train of clock pulses. The clock pulses are applied to all flip-flops and registers in the system. The master clock acts like a pump that supplies a constant beat to all parts of the system. A separate control signal must be used to decide which specific clock pulse will have an effect on a particular register. A 4-bit register with a load control input that is directed through gates and into the D inputs is shown in the folowing Figure.

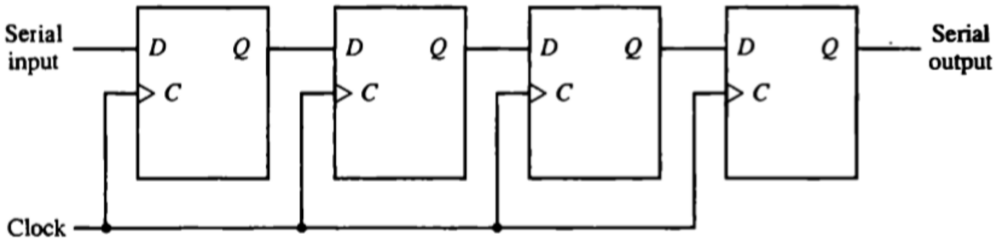
The C inputs receive clock pulses at all times. The buffer gate in the dock input reduces the power requirement. With each clock pulse, the D input determines the next state of the output. To leave the output unchanged, it is necessary to make the D input equal to the present value of the output. Note that the clock pulses are applied to the C inputs at all times. The load input determines whether the next pulse will accept new information or leave the information in the register intact.

The transfer of information from the inputs into the register is done simultaneously with all four bits during a single pulse transition.



1.11.2 Shift Registers

A register capable of shifting its binary information in one or both directions is called a shift register. The logical configuration of a shift register consists of a chain of flip-flops in cascade, with the output of one flip-flop 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. The simplest possible shift register is one that uses only flip-flops, as shown in Figure. The output of a given flip-flop is connected to the D input of the flip-flop at its right. The clock is common to all flip-flops. The serial input determines what goes into the leftmost position during the shift. The serial output is taken from the output of the rightmost flip-flop. Sometimes it is necessary to control the shift so that it occurs with certain clock pulses but not with others. This can be done by inhibiting the clock from the input of the register if we do not want it to shift.



1.11.3 Bidirectional Shift Register with Parallel Load

A register capable of shifting in one direction only is called a unidirectional shift register. A register that can shift in both directions is called a bidirectional shift register. The most general shift register has all the capabilities listed below. Others may have some of these capabilities, with at least one shift operation.

1. An input for clock pulses to synchronize all operations.

2. A shift-right operation and a serial input line associated with the shiftright.

3. A shift-left operation and a serial input line associated with the shift-left.

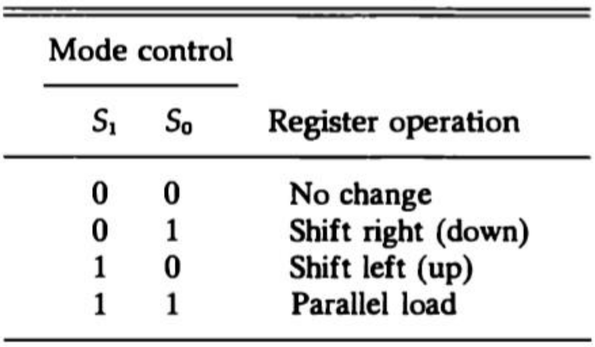
4. A parallel load operation and n input lines associated with the parallel transfer.

5. n parallel output lines.

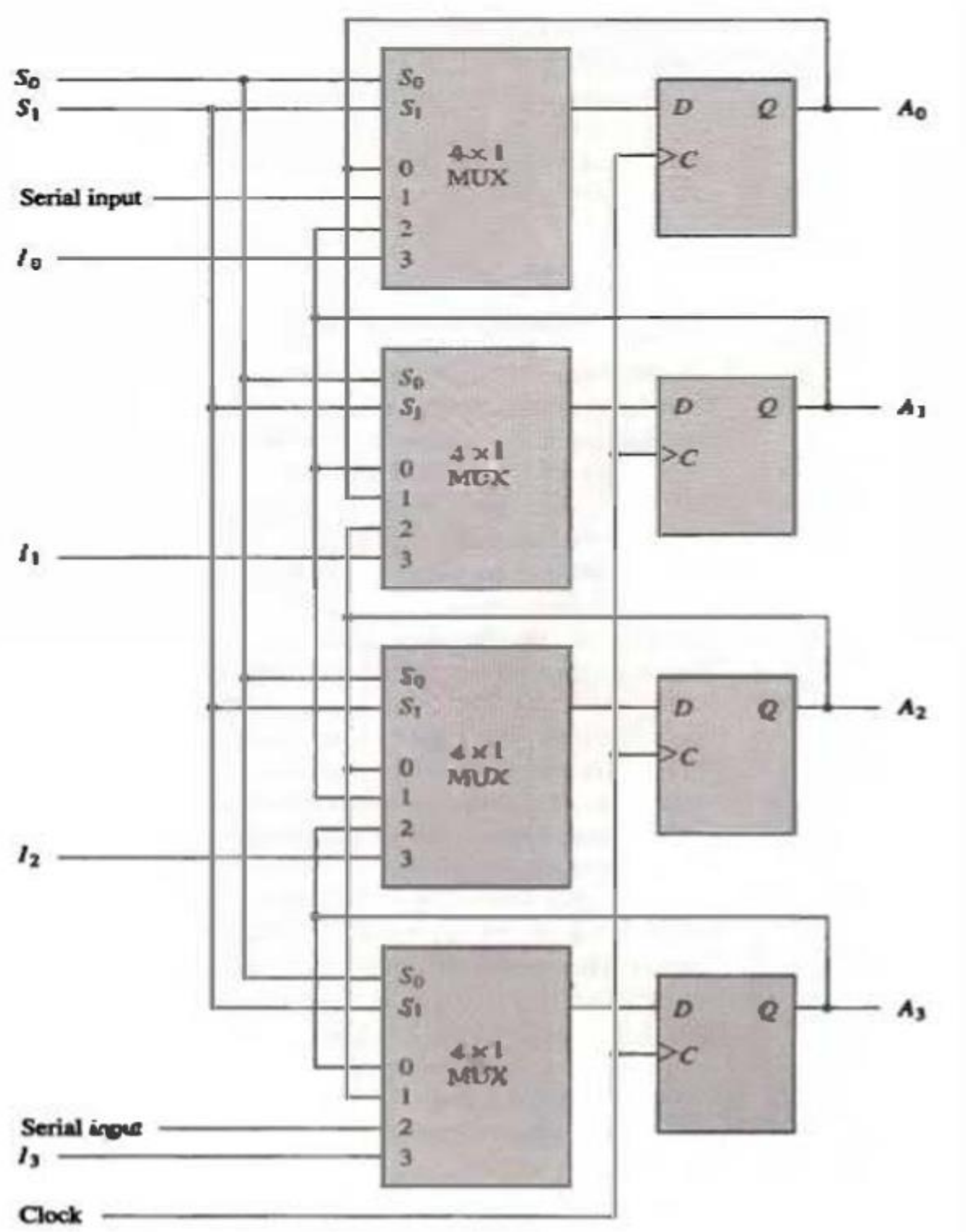
6. A control state that leaves the information in the register unchanged even though clock pulses are applied continuously.

A 4-bit bidirectional shift register with parallel load is shown in the following Figure. Each stage consists of a D flip-flop and a 4 x 1 multiplexer. The two selection inputs S_1 and S_0 select one of the multiplexer 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 the following Table. When the mode control S_1S_0 = 00, data input 0 of each multiplexer 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 transfers 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 multiplexer 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 operation results, with the other serial input data going into flip-flop A, 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. Note that the way the diagram is drawn, 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.



Mode control		
S_1	S_0	Register operation
0	0	No change
0	1	Shift right (down)
1	0	Shift left (up)
1	1	Parallel load



Shift registers are often used to interface digital systems situated remotely from each other. For example, suppose that it is necessary to transmit ann-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 inparallel 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-toserial conversion of data and the receiver converts the incoming serial data back to parallel data transfer.

1.12 Binary Counters

A register that goes through a predetermined sequence of states upon the application of input pulses is called a counter. The input pulses may be clock pulses or may originate from an external source. They may occur at uniform intervals of time or at random. Counters are found in almost all equipment containing digital logic. They are used for counting the number of occurrences of an event and are useful for generating timing signals to control the sequence of operations in digital computers. Of the various sequences a counter may follow, the straight binary sequence is the simplest and most straightforward. A counter that follows the binary number sequence is called a binary counter. An n-bit binary counter is a register of n flip-flops and associated gates that follows a sequence of states according to the binary count of n bits, from 0 to 2^n - 1.

Going through a sequence of binary numbers such as 0000, 0001, 0010, 0011, and so on, we note that the lower-order bit is complemented after every count and every other bit is complemented from one count to the next if and only if all its lower-order bits are equal to 1. For example, the binary count from 0111 (7) to 1000 (8) is obtained by

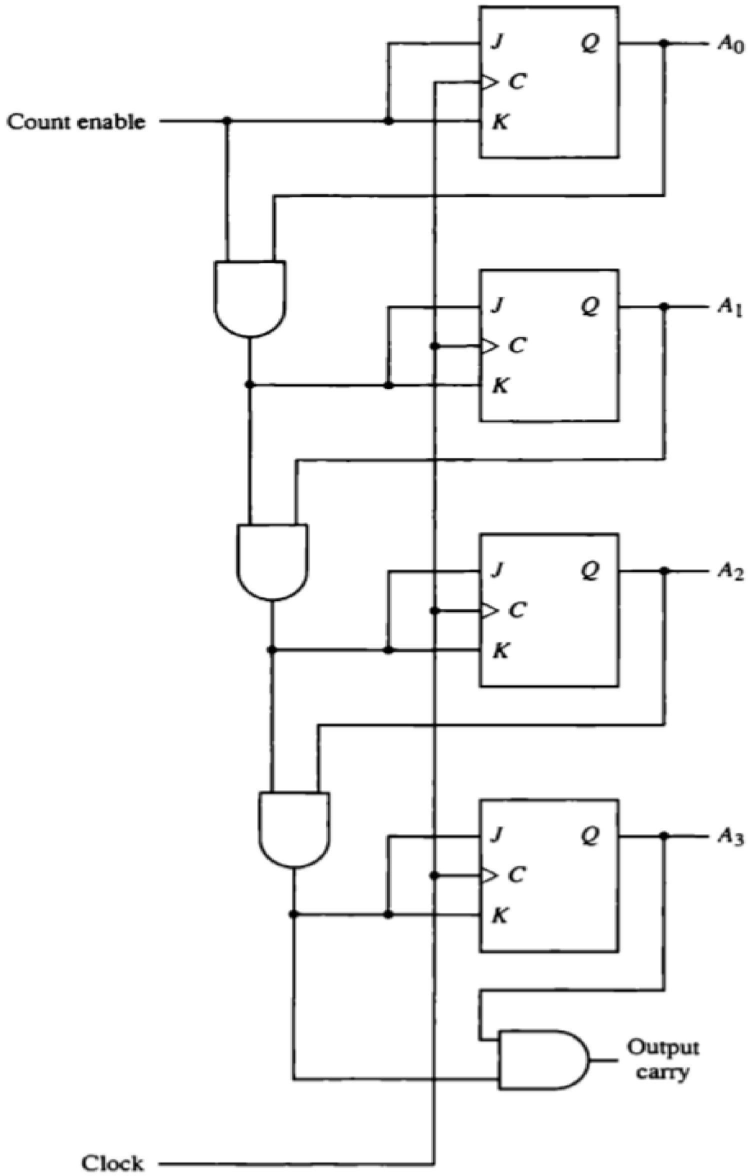
(a) complementing the low-order bit,

(b) complementing the second-order bit because the first bit of 0111 is 1,

(c) complementing the third-order bit because the first two bits of 0111 are 1' s, and

(d) complementing the fourth-order bit because the first three bits of 0111 are all1's.

Synchronous binary counters have a regular pattern, as can be seen from the 4-bit binary counter shown in Figure. The C inputs of all flip-flops receive the common clock. If the count enable is 0, all J and K inputs are maintainedat 0 and the output of the counter does not change. The first stage A_0 is complemented when the counter is enabled and the clock goes through a positive transition. Each of the other three flip-flops are complemented when all previous least significant flip-flops are equal to 1 and the count is enabled. The chain of AND gates generate the required logic for the J and K inputs. Theoutput carry can be used to extend the counter to more stages, with each stage having an additional flip-flop and an AND gate.



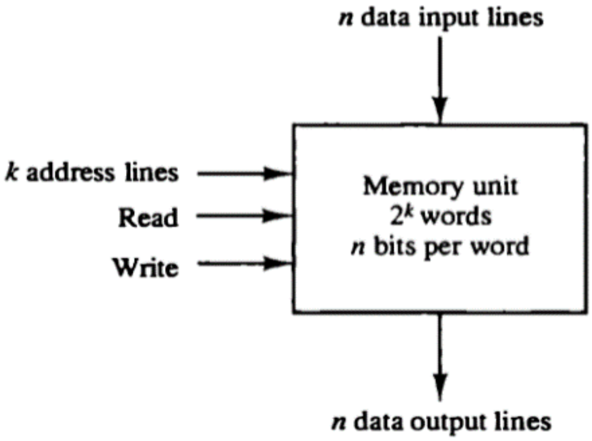
1.13 Memory Unit

A memory unit is a collection of storage cells together with associated circuits needed to transfer information in and out of storage. The memory stores binary information in groups of bits called words. A word in memory is an entity ofbits that move in and out of storage as a unit. A memory word is a group of l's and 0's and may represent a number, an instruction code, one or more alphanumeric characters, or any other binary-coded information. A group of eight bits is called a byte. Most computer memories use words whose number of bits is a multiple of 8. Thus a 16-bit word contains two bytes, and a 32-bit word is made up of four bytes. The capacity of memories in commercial computers is usually stated as the total number of bytes that can be stored.

Two major types of memories are used in computer systems: randomaccess memory (RAM) and read-only memory (ROM).

1.13.1 Random-Access Memory

In random-access memory (RAM) the memory cells can be accessed for information transfer from any desired random location. That is, the process of locating a word in memory is the same and requires an equal amount of time no matter where the cells are located physically in memory: thus the name "random access." Communication between a memory and its environment is achieved through data input and output lines, address selection lines, and control lines that specify the direction of transfer. A block diagram of a RAM unit is shown in the following figure. The n data input lines provide the information to be stored in memory, and the n data output lines supply the information coming out of memory. The k address lines provide a binary number of k bits that specify a particular word chosen among the 2' available inside the memory. The two control inputs specify the direction of transfer desired. The two operations that a random-access memory can perform are the write and read operations. The write signal specifies a transfer-in operation and the read signal specifies a transfer-out operation. On accepting one of these control signals, the internal circuits inside the memory provide the desired function.



The steps that must be taken for the purpose of transferring a new word to be stored into memory are as follows:

1. Apply the binary address of the desired word into the address lines.
2. Apply the data bits that must be stored in memory into the data input lines.
3. Activate the write input.

The memory unit will then take the bits presently available in the input data lines and store them in the word specified by the address lines. The steps that must be taken for the purpose of transferring a stored word out of memory are as follows:

1. Apply the binary address of the desired word into the address lines.

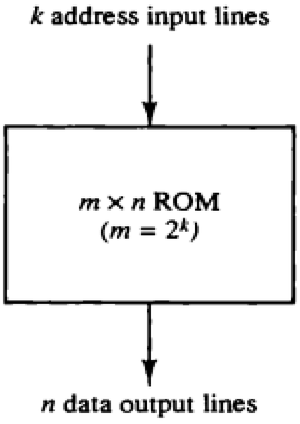
2. Activate the read input.

The memory unit will then take the bits from the word that has been selected by the address and apply them into the output data lines. The content of the selected word does not change after reading.

1.13.2 Read-Only Memory

As the name implies, a read-only memory (ROM) is a memory unit that performs the read operation only; it does not have a write capability. This implies that the binary information stored in a ROM is made permanent during the hardware production of the unit and cannot be altered by writing different words into it. Whereas a RAM is a general-purpose device whose contents can be altered during the computational process, a ROM is restricted to reading words that are permanently stored within the unit. The binary information to be stored, specified by the designer, is then embedded in the unit to form the required interconnection pattern. ROMs come with special internal electronic fuses that can be "programmed" for a specific configuration. Once the pattern is established, it stays within the unit even when power is turned off and on again.

ROMs find a wide range of applications in the design of digital systems. Basically, a ROM generates an input---output relation specified by a truth table. As such, it can implement any combinational circuit with k inputs and n outputs. When employed in a computer system as a memory unit, the ROM is used for storing fixed programs that are not to be altered and for tables of constants that are not subject to change. ROM is also employed in the design of control units for digital computers. As such, they are used to store coded information that represents the sequence of internal control variables needed for enabling the various operations in the computer. A control unit that utilizes a ROM to store binary control information is called a microprogrammed control unit.



1.13.3 Types of ROMs

The required paths in a ROM may be programmed in three different ways.

1. **Mask programming,** is done by the semiconductor company during the last fabrication process of the unit. The procedure for fabricating a ROM requires that the customer fill out the truth table that he or she wishes the ROM to satisfy. The truth table may be submitted in a special form provided by the manufacturer or in a specified format on a computer output medium. The manufacturer makes the corresponding mask for the paths to produce the l's and O's according to

the customer's truth table. This procedure is costly because the vendor charges the customer a special fee for custom masking the particular ROM. For this reason, mask programming is economical only if a large quantity of the same ROM configuration is to be ordered.

2. **Programmable read-only memory :** For small quantities it is more economical to use a second type of ROM called a **programmable read-only memory** or PROM. When ordered. PROM units contain all the fuses intact, giving all 1' s in the bits of the stored words. The fuses in the PROM are blown by application of current pulses through the output terminals for each address. A blown fuse defines a binary 0 state, and an intact fuse gives a binary 1 state. This allows users to program PROMs in their own laboratories to achieve the desired relationship between input addresses and stored words. Special instruments called PROM programmers are available commercially to facilitate this procedure. In any case, all procedures for programming ROMs are hardware procedures even though the word "programming" is used. The hardware procedure for programming ROMs or PROMs is irreversible, and once programmed, the fixed pattern is permanent and cannot be altered. Once a bit pattern has been established, the unit must be discarded if the bit pattern is to be changed.

3. **Erasable PROM** or EPROM. : A third type of ROM available is called **erasable PROM** or EPROM. The EPROM can be restructured to the initial value even though its fuses have been blown previously. When the EPROM is placed under a special ultraviolet light for a given period of time, the shortwave radiation discharges the internal gates that serve as fuses. After erasure, the EPROM returns to its initial state and can be reprogrammed to a new set of words. Certain PROMs can be erased with electrical signals instead of ultraviolet light. These PROMs are called **electrically erasable PROM** or EEPROM.

SREENIVASA INSTITUTE OF TECHNOLOGY AND MANAGEMENT STUDIES.
(AUTONOMOUS)
MCA DEPARTMENT

LECTURE NOTES

COMPUTER ORGANIZATION AND ARCHITECTURE

UNIT-2

Data Representation, Basic Computer Organization

Data Types

The data types found in the registers of digital computers may be classified as being one of the following categories:

(1) Numbers used in arithmetic computations,

(2) Letters of the alphabet used in data processing. and

(3) Other discrete symbols used for specific purposes.

All types of data, except binary numbers, are represented in computer registers in binary coded form. This is because registers are made up of flip-flops and flip-flops are two-state devises that can store only l's and 0's. The binary number system is the most natural system to use in a digital computer. But sometimes it is convenient to employ different number systems, especially the decimal number system, since it is used by people to perform arithmetic computations.

Number Systems

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

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

that is, 7 hundreds, plus 2 tens, plus 4 units, plus 5 tenths. Every decimal number can be similarly interpreted to find the quantity it represents. The binary number system uses the radix 2. The two digit symbols used are 0 and 1. The string of digits 101101 is interpreted to represent the quantity

$$1 \text{ x}2^5+0\text{x}2^4+1\text{x}2^3+1\text{x}2^2+0\text{x}2^1+1\text{x}2^0=45$$

To distinguish between different radix numbers, the digits will be enclosed in parentheses and the radix of the number inserted as a subscript. For example, to show the equality between decimal and binary forty-five we will write $(101101)_2 = (45)_{10}$.

Besides the decimal and binary number systems, the octal (radix 8) and hexadecimal (radix 16) are important in digital computer work. The eight symbols of the octal system are 0, 1, 2, 3, 4, 5, 6, and 7. The 16 symbols of the hexadecimal system are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. The last six symbols are, unfortunately, identical to the letters of the alphabet and can cause confusion at times. However, this is the convention that has been adopted. When used to represent hexadecimal digits, the symbols A, B, C, D, E, F correspond to the decimal numbers 10, 11, 12, 13, 14, 15, respectively. A number in radix r can be converted to the familiar decimal system by forming the sum of the weighted digits. For example, octal 736.4 is converted to decimal as follows:

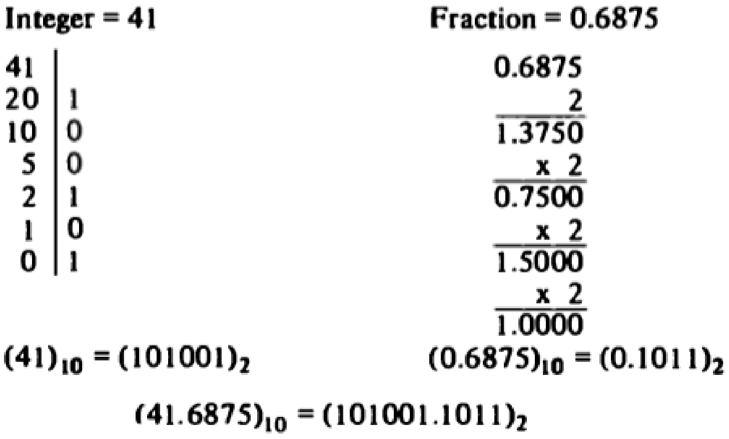
$$(736.4)_8 = 7 \text{ X } 8^2 + 3 \text{ X } 8^1 + 6 \text{ X } 8^0 + 4 \text{ X } 8^{-1} = 7 \text{ X } 64 + 3 \text{ X } 8 + 6 \text{ X } 1 + 4/8 = (478.5)_{10}$$

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

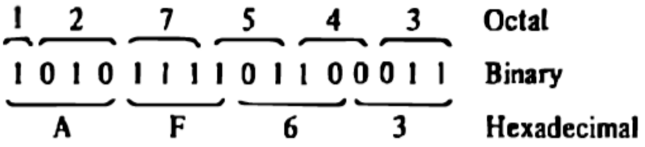
$$(F3)_{16} = F \text{ X } 16 + 3 = 15 \text{ X } 16 + 3 = (243)_{10}$$

Octal and Hexadecimal Numbers

The conversion from Decimal to Binary, Octal, and Hexadecimal representation plays an important part in digital computers. Since $2^3 = 8$ and $2^4 = 16$, each octal digit corresponds to three binary digits and each hexadecimal digit corresponds to four binary digits. The conversion from binary to octal is easily accomplished by partitioning the binary number into groups of three bits each. The corresponding octal digit is then assigned to each group of bits and the string of digits so obtained gives the octal equivalent of the binary number. Consider, for example, a 16-bit register.



Conversion of decimal 41.6875 into binary.



Binary, octal, and hexadecimal conversion

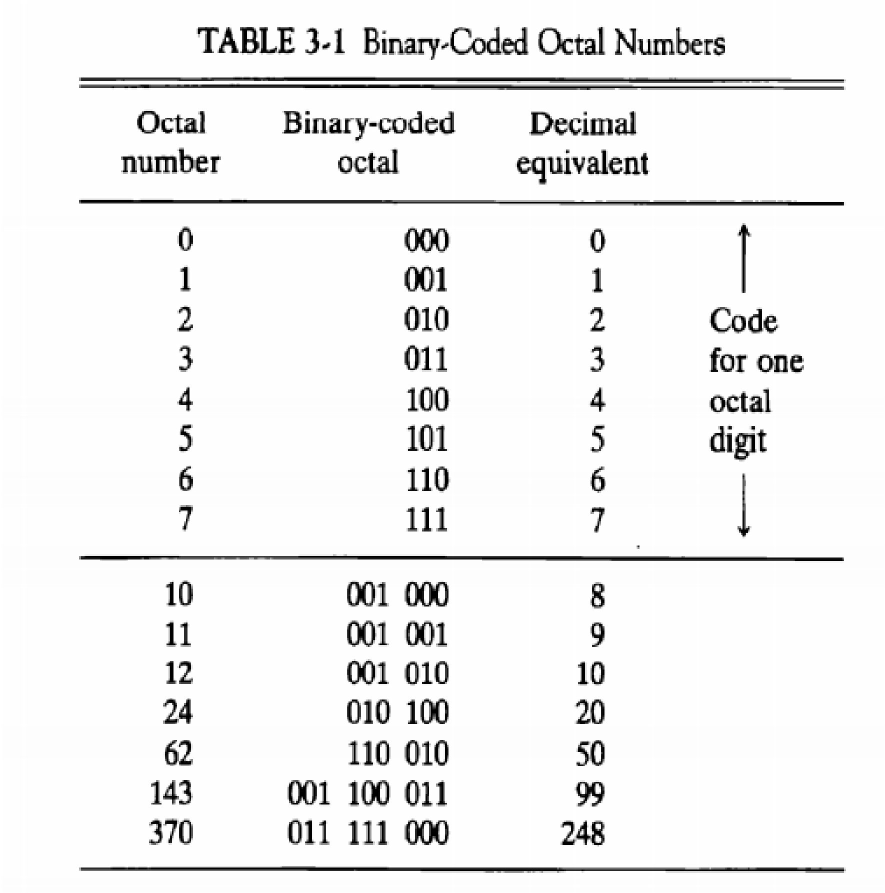
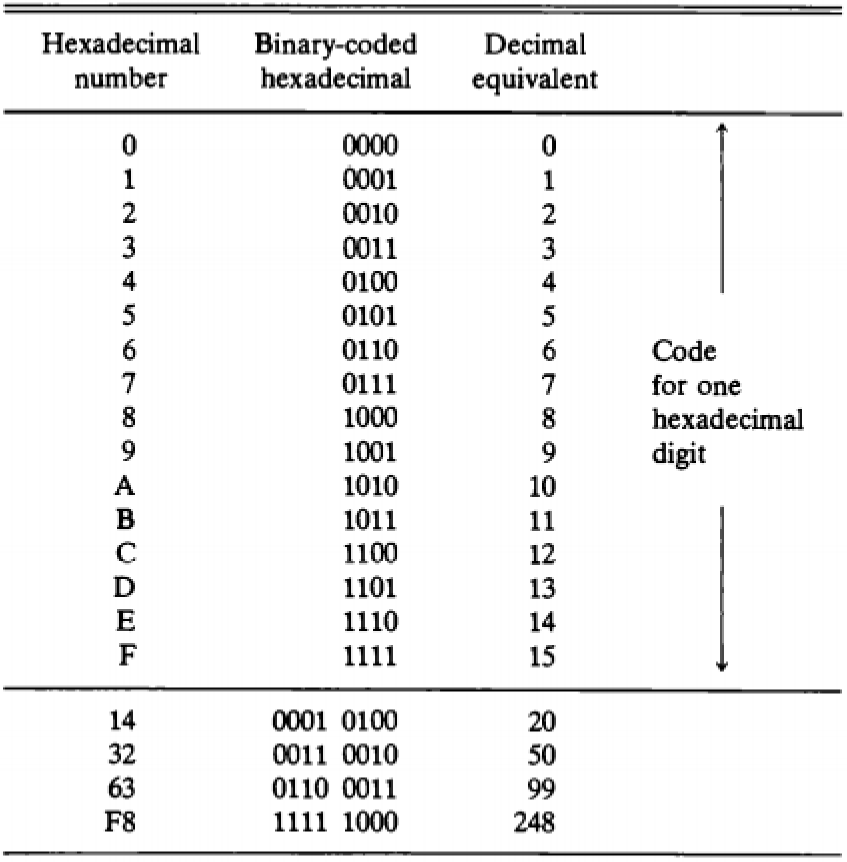


TABLE 3-1 Binary-Coded Octal Numbers

Octal number	Binary-coded octal	Decimal equivalent	
0	000	0	↑ Code for one octal digit ↓
1	001	1	
2	010	2	
3	011	3	
4	100	4	
5	101	5	
6	110	6	
7	111	7	
10	001 000	8	
11	001 001	9	
12	001 010	10	
24	010 100	20	
62	110 010	50	
143	001 100 011	99	
370	011 111 000	248	

TABLE 3-2 Binary-Coded Hexadecimal Numbers



Hexadecimal number	Binary-coded hexadecimal	Decimal equivalent	
0	0000	0	Code for one hexadecimal digit
1	0001	1	
2	0010	2	
3	0011	3	
4	0100	4	
5	0101	5	
6	0110	6	
7	0111	7	
8	1000	8	
9	1001	9	
A	1010	10	
B	1011	11	
C	1100	12	
D	1101	13	
E	1110	14	
F	1111	15	
14	0001 0100	20	
32	0011 0010	50	
63	0110 0011	99	
F8	1111 1000	248	

Decimal Representation

Binary Number System is the most natural system for a computer, but the people are accustomed to the decimal system. One way to solve this conflict is to convert all input decimal numbers into binary numbers, let the computer perform all the operations in binary and convert the result into decimal. It is also possible for the computer to perform all arithmetic operations directly with decimal numbers provided they are placed in the registers in a coded form.

It is important to understand the difference between the conversion of decimal numbers into binary and the binary coding of decimal numbers. For example when converted into binary number, the decimal number 99 is represented as 1100011. But when represented in BCD it is

1001 1001

Few decimal numbers and their representation in BCD are as follows

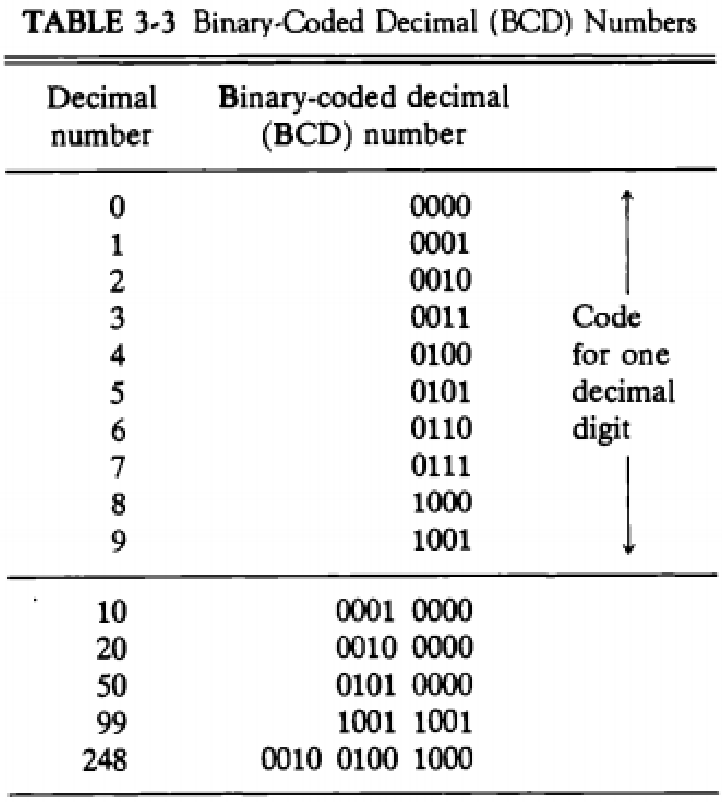


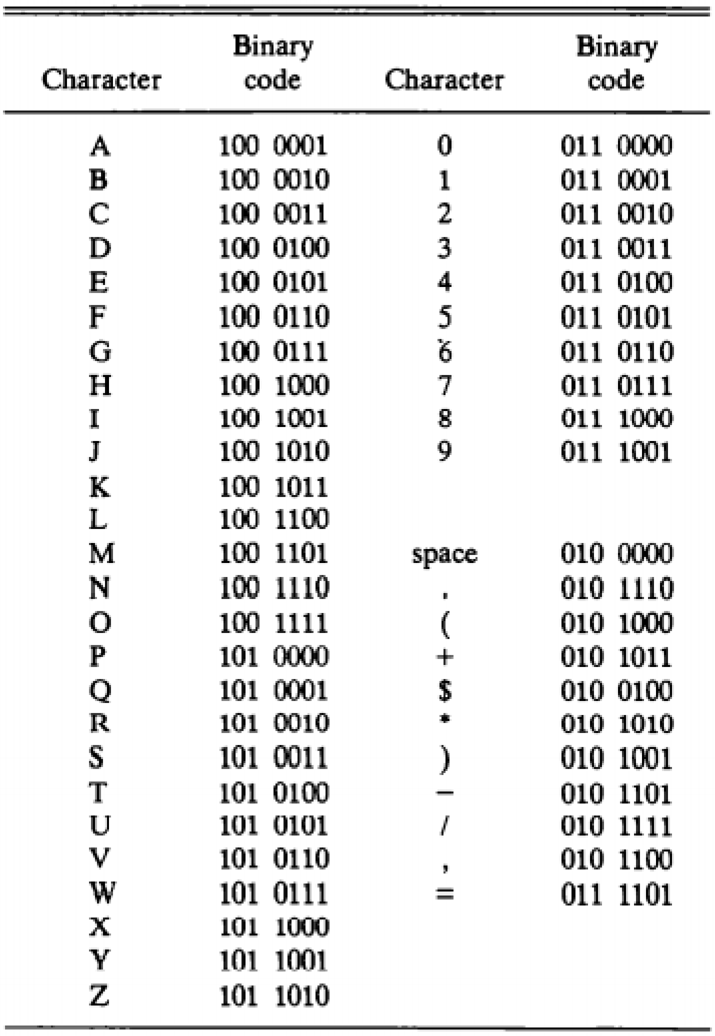
TABLE 3-3 Binary-Coded Decimal (BCD) Numbers

Decimal number	Binary-coded decimal (BCD) number	
0	0000	↑
1	0001	
2	0010	
3	0011	Code
4	0100	for one
5	0101	decimal
6	0110	digit
7	0111	
8	1000	
9	1001	↓
10	0001 0000	
20	0010 0000	
50	0101 0000	
99	1001 1001	
248	0010 0100 1000	

Alphanumeric Representation

Many applications of digital computers require handling of data that consists of not only of numbers but also letters and special characters. The standard alphanumeric binary code is ASCII (American Standard Code for Information Interchange) which uses 7 bits to code 128 characters.

TABLE 3-4 American Standard Code for Information Interchange (ASCII)



Character	Binary code	Character	Binary code
A	100 0001	0	011 0000
B	100 0010	1	011 0001
C	100 0011	2	011 0010
D	100 0100	3	011 0011
E	100 0101	4	011 0100
F	100 0110	5	011 0101
G	100 0111	6	011 0110
H	100 1000	7	011 0111
I	100 1001	8	011 1000
J	100 1010	9	011 1001
K	100 1011		
L	100 1100		
M	100 1101	space	010 0000
N	100 1110	.	010 1110
O	100 1111	(	010 1000
P	101 0000	+	010 1011
Q	101 0001	$	010 0100
R	101 0010	*	010 1010
S	101 0011	)	010 1001
T	101 0100	−	010 1101
U	101 0101	/	010 1111
V	101 0110	,	010 1100
W	101 0111	=	011 1101
X	101 1000		
Y	101 1001		
Z	101 1010		

Complements

Complements are used in digital computers for simplifying the subtraction operation and for logical manipulation. There are two types of complements for each base r system: the r's complement and the (r - l)'s complement.

9' s complement ,10's complement

l's complement ,2's complement

When the value of the base r is substituted in the name, the two types are referred to as the 2's and 1's complement for binary numbers and the 10's and 9's complement for decimal numbers.

(r - l)'s Complement

Given a number N in base r having n digits, the (r - 1)'s complement of N is defined as (r' - 1) - N.

For decimal numbers r = 10 and r - 1 = 9, so the 9's complement of N is

(10' - 1) - N.

Now, 10' represents a number that consists of a single 1 followed by n 0's. 10' - 1 is a number represented by n 9's.

For example, with n = 4 we have 10^4 = 10000 and 10^4 - 1 = 9999. It follows that the 9' s complement of a decimal number is obtained by subtracting each digit from 9.

For example, the 9's complement of 546700 is

999999 - 546700 = 453299

and the 9's complement of I2389 is 99999 - 12389 = 876I0.

For binary numbers, r = 2 and r - 1 = 1, so the 1's complement of N is

(2' - 1) - N.

Again, 2' is represented by a binary number that consists of a 1 followed by n 0's. 2' - 1 is a binary number represented by n 1's.

For example, with n = 4, we have 2^4 = (10000), and 2^4 - 1 = $(1111)_2$. Thus the 1's complement of a binary number is obtained by subtracting each digit from 1. However, the subtraction of a binary digit from 1 causes the bit to change from 0 to 1 or from 1 to 0. Therefore, the 1's complement of a binary number is formed by changing 1's into 0's and 0's into 1's. For example, the 1's complement of I011001 is 0100110 and the 1' s complement of 0001111 is 1110000. The (r - 1)'s complement of octal or hexadecimal numbers are obtained by subtracting each digit from 7 or F (decimal 15) respectively.

(r's) Complement

The r's complement of an n-digit number N in base r is defined as r' - N for $N \neq 0$ and 0 for N = D. Comparing with the (r - 1)'s complement, we note that the r's complement is obtained by adding 1 to the (r - 1)'s complement since

r' - N = [(r' - 1) - N] + 1.

Thus the 10's complement of the decimal 2389 is 7610 + 1 = 7611 and is obtained by adding 1 to the 9' s complement value.

The 2's complement of binary 101100 is

010011 + 1 = 010100

and is obtained by adding 1 to the 1's complement value. Since 10' is a number represented by a 1 followed by n 0's, then 10' - N, which is the 10's complement of N, can be formed also be leaving all least significant 0's unchanged, subtracting the first nonzero least significant digit from 10, and then subtracting all higher significant digits from 9.

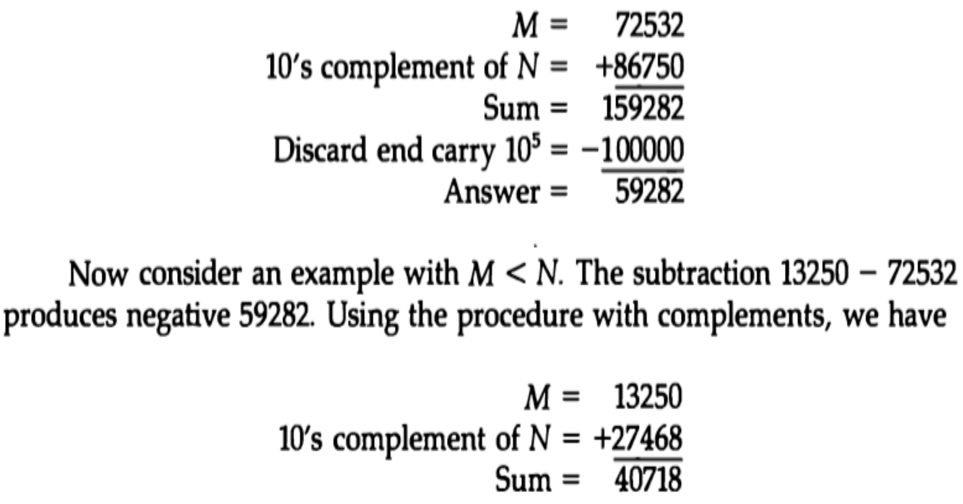
The 10's complement of 246700 is 753300 and is obtained by leaving the two zeros unchanged, subtracting 7 from 10, and subtracting the other three digits from 9. Similarly, the 2's complement can be formed by leaving all least significant 0's and the first 1 unchanged, and then replacing 1's by 0's and 0's by 1's in all other higher significant bits. The 2's complement of 1101100 is 0010100 and is obtained by leaving the two low-order 0's and the first 1 unchanged, and then replacing 1's by 0's and 0's by 1's in the other four most significant bits.

Subtraction of Unsigned Numbers

The direct method of subtraction taught in elementary schools uses the borrow concept. In this method we borrow a 1 from a higher significant position when the minuend digit is smaller than the corresponding subtrahend digit. This seems to be easiest when people perform subtraction with paper and pencil. When subtraction is implemented with digital hardware, this method is found to be less efficient than the method that uses complements. The subtraction of two n-digit unsigned numbers M - N (N ≠ 0) in base r can be done as follows:

1. Add the minuend M to the r's complement of the subtrahend N.
   This performs M + (r' - N) = M-N + r'.
2. If M > N, the sum will produce an end carry r' which is discarded, and what is left is the result M - N.
3. If M < N, the sum does not produce an end carry and is equal to r' - (N - M), which is the r's complement of (N - M).

To obtain the answer in a familiar form, take the r' s complement of the sum and place a negative sign in front. Consider, for example, the subtraction 72532 - 13250 = 59282. The l0's complement of 13250 is 86750. Therefore:



$$\begin{array}{rr} M = & 72532 \\ \text{10's complement of } N = & +\underline{86750} \\ \text{Sum} = & 159282 \\ \text{Discard end carry } 10^5 = & -\underline{100000} \\ \text{Answer} = & 59282 \end{array}$$

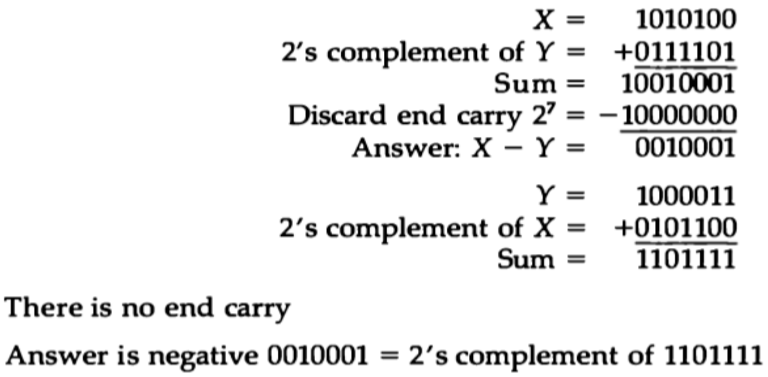
Now consider an example with $M < N$. The subtraction 13250 − 72532 produces negative 59282. Using the procedure with complements, we have

$$\begin{array}{rr} M = & 13250 \\ \text{10's complement of } N = & +\underline{27468} \\ \text{Sum} = & 40718 \end{array}$$

There is no end carry

Answer is negative 59282 = 10's complement of 40718

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



$$\begin{array}{rr}
X = & 1010100 \\
\text{2's complement of } Y = & +\underline{0111101} \\
\text{Sum} = & 10010001 \\
\text{Discard end carry } 2^7 = & -\underline{10000000} \\
\text{Answer: } X - Y = & 0010001
\end{array}$$

$$\begin{array}{rr}
Y = & 1000011 \\
\text{2's complement of } X = & +\underline{0101100} \\
\text{Sum} = & 1101111
\end{array}$$

There is no end carry

Answer is negative 0010001 = 2's complement of 1101111

Fixed-Point Representation

Positive integers, including zero, can be represented as unsigned numbers. However, to represent negative integers, we need a notation for negative values. In ordinary arithmetic, a negative number is indicated by a minus sign and a positive number by a plus sign. Because of hardware limitations, computers must represent everything with 1's and 0's, including the sign of a number. As a consequence, it is customary to represent the sign with a bit placed in the leftmost position of the number. The convention is to make the sign bit equal to 0 for positive and to 1 for negative. In addition to the sign, a number may have a binary (or decimal) point. The position of the binary point is needed to represent fractions, integers, or mixed integer-fraction numbers. The representation of the binary point in a register is complicated by the fact that it is characterized by a position in the register. There are two ways of specifying the position of the binary point in a register: by giving it a fixed position or by employing a floating-point representation.

The fixed-point method assumes that the binary point is always fixed in one position. The two positions most widely used are

(1) a binary point in the extreme left of the register to make the stored number a fraction, and

(2) a binary point in the extreme right of the register to make the stored number an integer.

In either case, the binary point is not actually present, but its presence is assumed from the fact that the number stored in the register is treated as a fraction or as an integer. The floating-point representation uses a second register to store a number that designates the position of the decimal point in the first register. Floating-point representation is discussed further in the next section.

Integer Representation

When an integer binary number is positive the sign is represented with 0, when the number is negative the sign is represented by 1 but the rest of the number is represented in one of three possible ways:

1. Signed Magnitude representation.
2. Signed 1's Complement representation.
3. Signed 2's Complement representation.

Signed Magnitude representation of negative number consists of the magnitude and negative number. In the other two representations, the negative number is represented in either 1's complement or 2's complement of its positive value.

For example,

+ 14 is represented in the 8 bit register as 00001110

There are three different ways to represent **-14**

In Signed Magnitude representation: 1 0001110
In Signed 1's Complement representation : 1 1110001
In Signed 2's Complement representation : 1 1110010

The Signed Magnitude representation of **-14 is** obtained from **+14** by complementing signed bit.

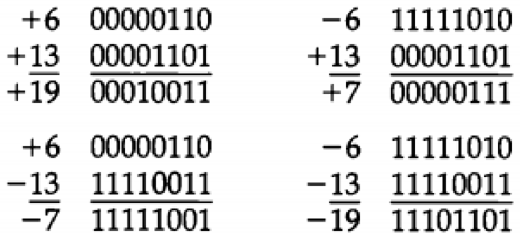
Signed magnitude method is employed in ordinary arithmetic but it is not suitable for computer arithmetic. Signed 1's complement method has some difficulties, it has two representation for 0 (+0 and -0). The signed binary arithmetic entirely deals with 2's complement.

Arithmetic Addition

Addition of two numbers in signed magnitude system follows the rules of ordinary arithmetic. For example

(+32) + (- 45) = - (45 -32) = - 13

This process requires comparison of signs and magnitudes and then performing addition or subtraction. In Signed 2's complement method, no comparison and no subtraction requires only addition and complementation.



```
 +6   00000110      −6   11111010
+13   00001101     +13   00001101
+19   00010011      +7   00000111

 +6   00000110      −6   11111010
−13   11110011     −13   11110011
 −7   11111001     −19   11101101
```

In each of the above four cases, the operation performed is only addition. Any carry out of the sign bit position is discarded and negative results are automatically in 2's complement form.

Arithmetic Subtraction

Subtraction of two signed binary numbers is as follows: Take the 2's complement of subtrahend and add it to the minuend, a carry out of the sign bit position is discarded. Therefore computers need only one common circuit to handle both types of operations.

Consider the subtraction (- 6) – (- 13) = +7

11111010 – 11110011 = 11111010 + 00001101 = 100000111

Discard carry, which is equivalent to +7

Overflow

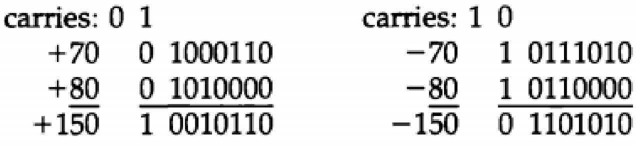
When two digits of length n digits, each are added and the sum occupies (n+1) digits, we say that an overflow occurred. An overflow is a problem in digital computers, because the length of the register is fixed. The result of (n+1) bits can't be accommodated in a register with a standard length of n bits.

Many computers detect the occurrence of the overflow and when it occurs, a corresponding Flip-flop is set, which can then be used for other processing.

The detection of overflow after the addition of two binary numbers depends on whether the numbers are considered as signed or unsigned.

- When the two numbers added are unsigned, an overflow is detected from the end carry out of the most significant digit.
- When the numbers are signed, the left most bit is represented for sign and negative numbers are in 2'complemented form. When they are added, the sign bit is considered as part of the number and the end carry does not indicate an overflow.

An overflow cannot occur if one number is positive and one number is negative. Since adding a positive number to negative number produces a result that is smaller than the larger of the two umbers. An overflow occur when two numbers both are positive or negative.



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

If two carries are applied to EX-OR gate, an over flow can be detected, when it gives 1 as output.

Floating Point Representation

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

Mantissa	**Exponent**
+0.6132789	+04

The value of the exponent indicates that the actual position of the decimal point if four positions to the right of the indicated decimal point in the fraction. This representation is equivalent to the scientific notation**:** **0.6132789 X 10^{+4}**

i.e in the format $m\ x\ r^e$

Only the mantissa and exponent are physically represented in the register. The radix and radix point position of mantissa are always assumed. Floating point binary number is represented in a similar manner except that it uses base 2 for the exponent. For example a binary number +1001.11 is represented with 8 bit mantissa and 6 bit exponent is as follows:

Mantissa	**Exponent**
01001110	000100

Left most 0 represent positive number, this is equivalent to:

$$\mathbf{+(.1001110)_2\ X\ 2^{+4}}$$

Normalization

A Floating point number is said to be normalized if the most significant digit of the mantissa is non zero. For example the number 350 is normalized but 00035 is not. Regardless of where the radix point is assumed to be in the mantissa, the number is normalized only when its left most digit is non zero.

For example the number 00010110 is not normalized, this number can be normalized by shifting the three position left and discarding the leading zeroes to obtain 10110000. The three shift multiply the number with 2^3. Normalized numbers provide the maximum possible precision for the floating point numbers.

Other Binary Codes

Some of the commonly used Binary Codes. The following is the list:

- 8421 Codes
- 2421 Codes
- 5211 Codes
- Excess-3 Codes
- Gray Codes

In the above list, the first three i.e. 8421, 2421 and 5211 are Weighted codes while the other two are non-weighted binary codes.

8421 Code or BCD Code

The decimal numbers 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 can be expressed in Binary numbers as shown below. All these binary numbers again expressed in the last column by expanding into 4 bits. As per the weighted binary digits, the 4 Bit binary numbers can be expressed according to their place value from left to right as 8421 ($2^3\ 2^2\ 2^1\ 2^0 = 8421$).

DECIMAL NUMBER	BINARY NUMBER	4 BIT EXPRESSION(8421)
0	0	0000
1	1	0001
2	10	0010
3	11	0011
4	100	0100
5	101	0101
6	110	0110
7	111	0111
8	1000	1000
9	1001	1001

As per the above expression all the decimal numbers written in the 4 Bit binary code in the form of 8421 and this is called as 8421 Code and also as Binary coded decimal BCD.

As this is a straight code, any Decimal number can be expressed easily because the weights of the positions are straight for easy conversion into this 8421 code.

There are other forms of codes which are not so popular but rather confusing. They are 2421 code, 5211 code, reflective code, sequential code, non- weighted coded, excess-3 code and Grey code. They are having their own importance for some of the exclusive applications and may be useful for some of the typical applications.

2421 Code

This code also a 4 bit application code where the binary weights carry 2, 4, 2, 1 from left to right.

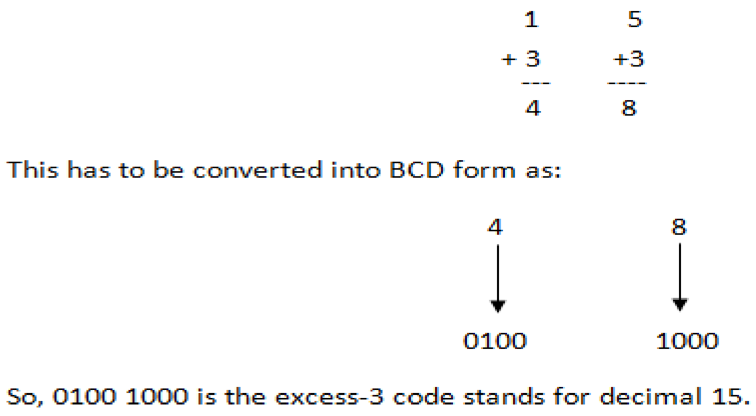
DECIMAL NUMBER	BINARY NUMBER	2421 CODE
0	0	0000

DECIMAL NUMBER	BINARY NUMBER	2421 CODE
1	1	0001
2	10	0010
3	11	0011
4	100	0100
5	101	1011
6	110	1100
7	111	1101
8	1000	1110
9	1001	1111

Excess-3 Code

As mentioned above, some of the codes will not follow the binary weights, Excee-3 code is an example of it and it is an important 4 bit code. The excess – 3 code of a decimal number is achieved by adding the number 3 to the 8421 code.

For example to convert 15 to an excess-3 code, first 3 to be added to each digit as shown below.



```
   1      5
 + 3     +3
  ---    ----
   4      8
```

This has to be converted into BCD form as:

```
  4            8
  ↓            ↓
0100         1000
```

So, 0100 1000 is the excess-3 code stands for decimal 15.

BACK TO TOP

Excess -3 Code Examples

1. Find the excess-3 code of $(237.75)_{10}$
2. Find the decimal number of excess-3 number 110010100011.01110101.

Sol:

1) The excess-3 code for $(237)_{10}$ is obtained by adding 3 to all the digits individually, that is 2, 3 an

d 7 will become 5, 6 and 10 respectively. These 5, 6 and 10 decimals have to be converted into binary form and the result is 010101101010.

The excess-3 code for $(.75)_{10}$ is obtained by replacing 7 and 5 with 10 and 8 respectively by adding 3 to each digit. That is, the excess-3 code for $(.75)_{10}$ is.10101000.

Combining the results of the integral and fractional parts, the excess-3 code for $(237.75)_{10}$ is 010101101010.10101000.

2) The excess-3 code is 110010100011.01110101

By separating 4 bits as group the equivalent excess-3 code is given as 1100 1010 0011.0111 0101.

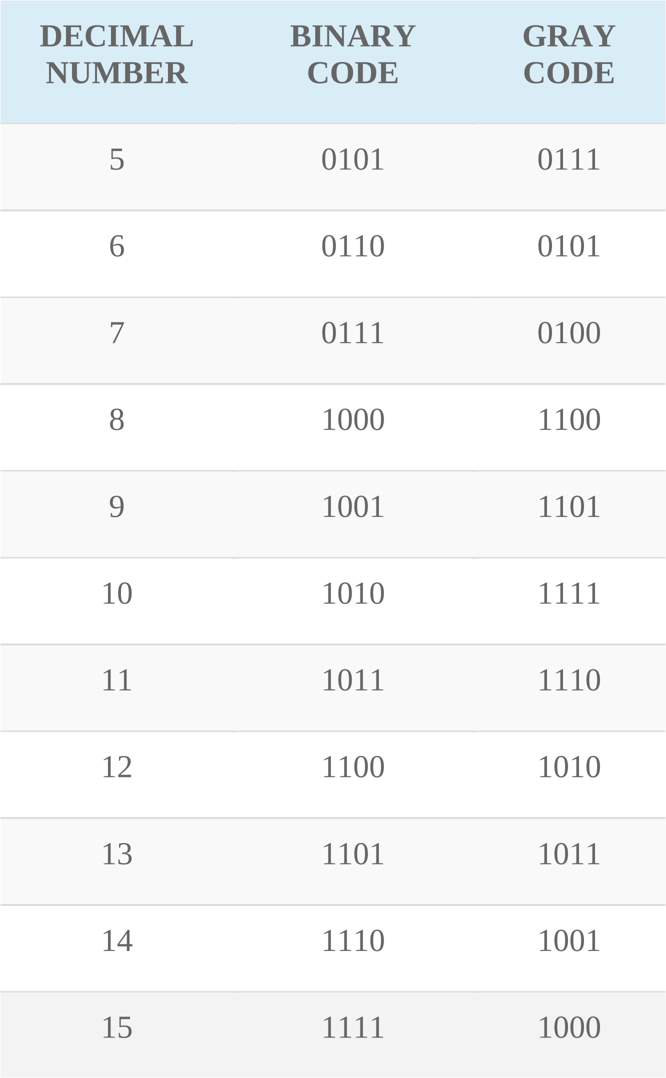
Subtracting 0011 from each four-bit group, we obtain the new number as: 1001 0111 0000.0100 0010.

Therefore, the decimal equivalent is $(970.42)_{10}$.

Gray Code

The gray code is the code where one bit will be differed to the preceding number. For example, decimal numbers 13 and 14 are represented by gray code numbers 1011 and 1001, these numbers differ only in single position that is the second position from the right. In the same way first position on the left changes for 7 and 8 which are 0100 and 1100 and this is also called Unit-distance code. The gray code has very special place in digital electronics.

DECIMAL NUMBER	BINARY CODE	GRAY CODE
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110



DECIMAL NUMBER	BINARY CODE	GRAY CODE
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

EBCDIC Code

EBCDIC stands for Extended Binary Coded Decimal Interchange Code. IBM invented this code to extend the Binary Coded Decimal which existed at that time. All the IBM computers and peripherals use this code. It is an 8-bit code and therefore can accommodate 256 characters. Below are given some characters of **EBCDIC code** to get familiar with it.

Error Detection Codes

The binary information is transferred from one location to another location through some communication medium. The external noise can change bits from 1 to 0 or 0 to 1.This changes in values are called errors. For efficient data transfer, there should be an error detection and correction codes. An error detection code is a binary code that detects digital errors during transmission. A famous error detection code is a Parity Bit method.

Parity Bit Method :

A parity bit is an extra bit included in binary message to make total number of 1's either odd or even. Parity word denotes number of 1's in a binary string. There are two parity system- even and odd. In even parity system 1 is appended to binary string it there is an odd number of 1's in string otherwise 0 is appended to make total even number of 1's.

In odd parity system, 1 is appended to binary string if there is even a number of 1's to make an odd number of 1's. The receiver knows that whether sender is an odd parity generator or even parity generator. Suppose if sender is an odd parity generator then there must be an odd number of 1's in received binary string. If an error occurs to a single bit that is either bit is changed to 1 to 0 or O to 1, received binary bit will have an even number of 1's which will indicate an error.

The limitation of this method is that only error in a single bit would be identified.

Message (XYZ)	P(Odd)	P(Even)
000	1	0
001	0	1
010	0	1
011	1	0
100	0	1
101	1	0
110	1	0
111	0	1

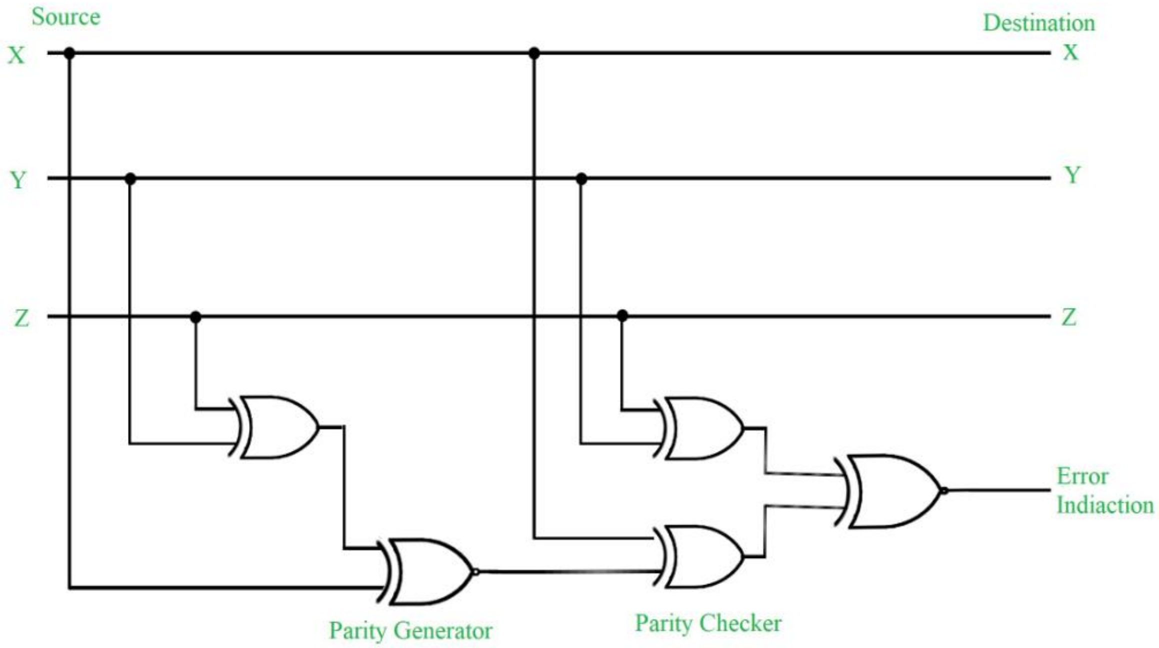


Fig: Parity Checker

Basic Computer Organization

Instruction code

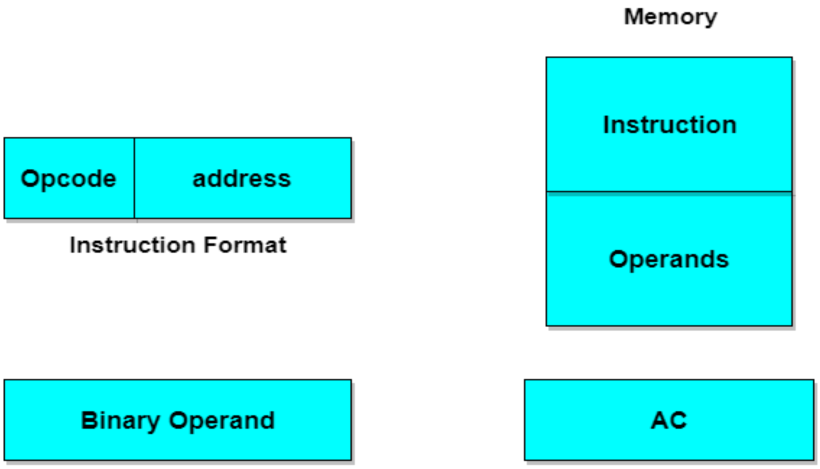
A **Program**, as we all know, is a set of instructions that specify the operations, operands, and the sequence by which processing has to occur. An **instruction code** is a group of bits that tells the computer to perform a specific operation part.

The operation code of an instruction is a group of bits that define operations such as add, subtract, multiply, shift and compliment. The number of bits required for the operation code depends upon the total number of operations available on the computer. The operation code must consist of at least **n bits** for a given **2^n** operations. The operation part of an instruction code specifies the operation to be performed.

The operation must be performed on the data stored in registers. An instruction code therefore specifies not only operations to be performed but also the registers where the operands(data) will be found as well as the registers where the result has to be stored.

Stored Program Organization

The simplest way to organize a computer is to have **Processor Register** and instruction code with two parts. The first part specifies the operation to be performed and second specifies an address. The memory address tells where the operand in memory will be found. Instructions are stored in one section of memory and data in another.



Computers with a single processor register is known as **Accumulator (AC)**. The operation is performed with the memory operand and the content of AC.

Addressing Modes

The operation field of an instruction specifies the operation to be performed. This operation will be executed on some data which is stored in computer registers or the main memory. The way any operand is selected during the program execution is dependent on the addressing mode of the instruction.

Types of Addressing Modes

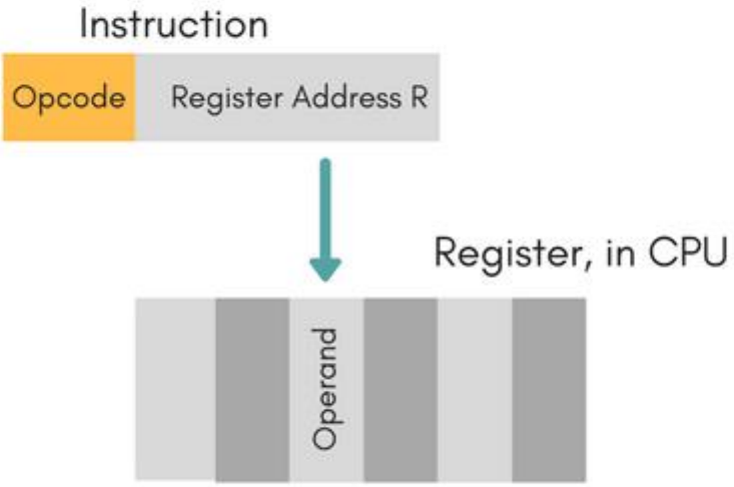
Below we have discussed different types of addressing modes one by one:

1. Immediate Mode

In this mode, the operand is specified in the instruction itself. An immediate mode instruction has an operand field rather than the address field.

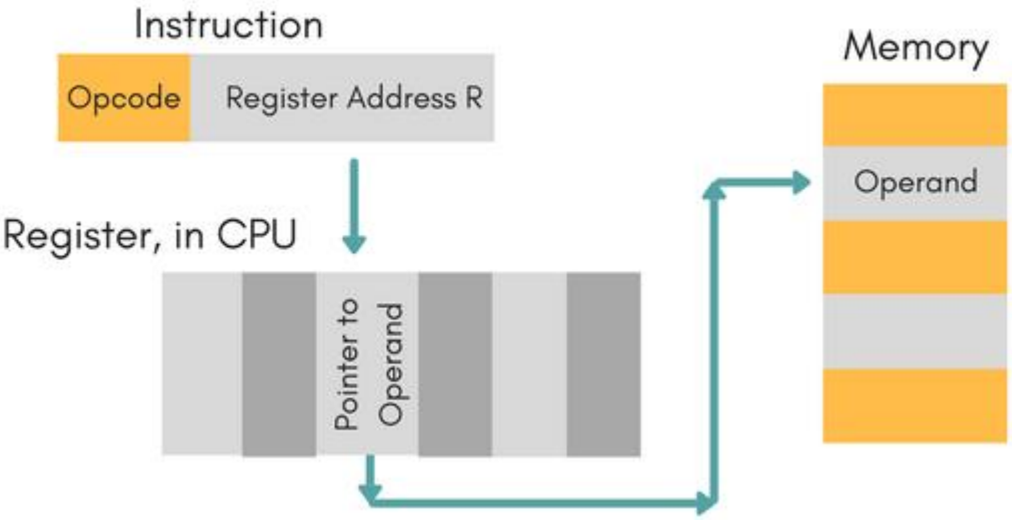
2. Register Mode

In this mode the operand is stored in the register and this register is present in CPU. The instruction has the address of the Register where the operand is stored.



3. Register Indirect Mode

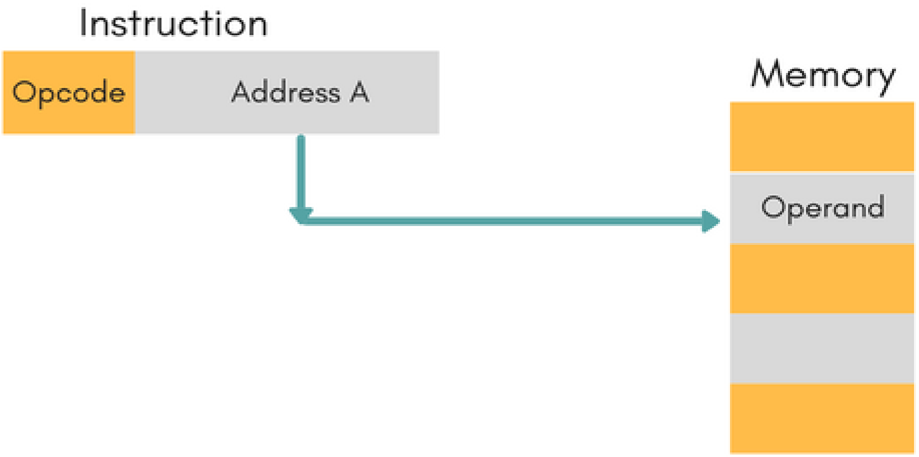
In this mode, the instruction specifies the register whose contents give us the address of operand which is in memory. Thus, the register contains the address of operand rather than the operand itself.



4. Direct Addressing Mode

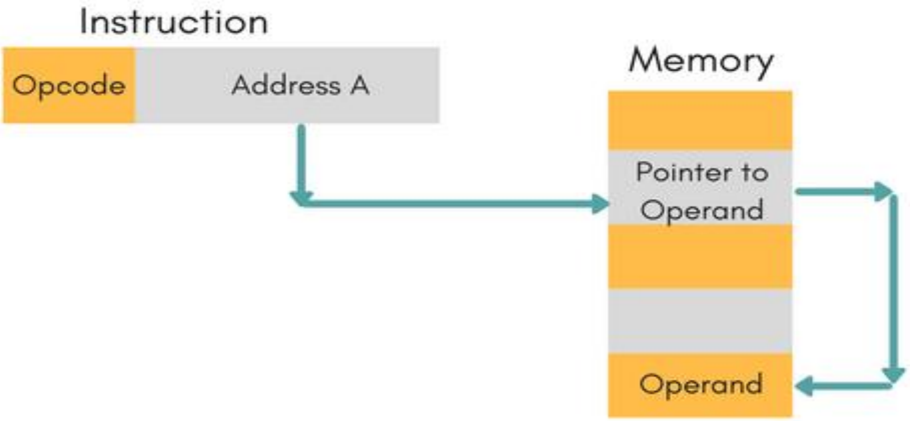
In this mode, effective address of operand is present in instruction itself.

- Single memory reference to access data.
- No additional calculations to find the effective address of the operand.



5. Indirect Addressing Mode

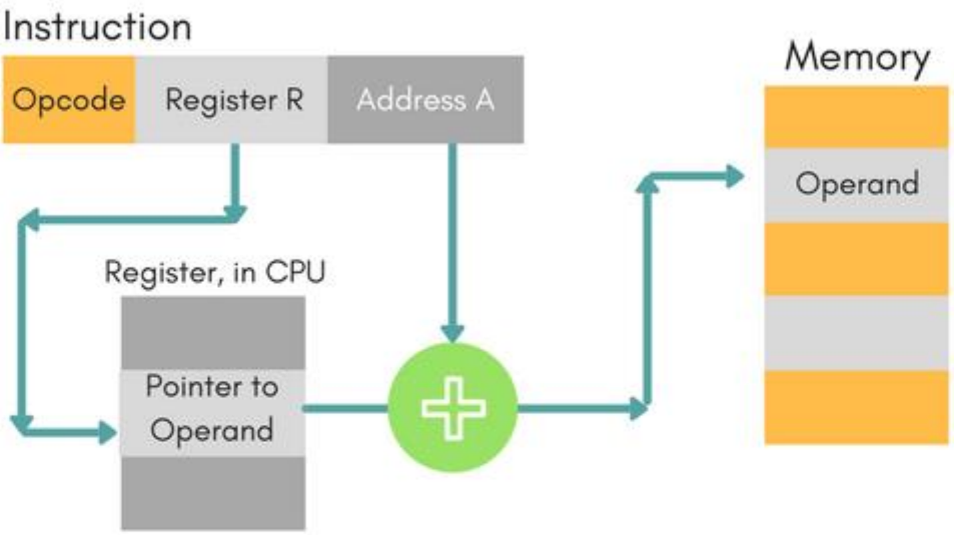
In this, the address field of instruction gives the address where the effective address is stored in memory. This slows down the execution, as this includes multiple memory lookups to find the operand.



6. Displacement Addressing Mode

In this the contents of the indexed register is added to the Address part of the instruction, to obtain the effective address of operand.

EA = A + (R), In this the address field holds two values, A(which is the base value) and R(that holds the displacement), or vice versa.



7. Relative Addressing Mode

It is a version of Displacement addressing mode.

In this the contents of PC(Program Counter) is added to address part of instruction to obtain the effective address.

EA = A + (PC), where EA is effective address and PC is program counter.

The operand is A cells away from the current cell(the one pointed to by PC)

8. Base Register Addressing Mode

It is again a version of Displacement addressing mode. This can be defined as EA = A + (R), where A is displacement and R holds pointer to base address.

9. Stack Addressing Mode

In this mode, operand is at the top of the stack. For example: ADD, this instruction will *POP* top two items from the stack, add them, and will then *PUSH* the result to the top of the stack.

Computer Registers

Registers are a type of computer memory used to quickly accept, store, and transfer data and instructions that are being used immediately by the CPU. The registers used by the CPU are often termed as Processor registers.

A processor register may hold an instruction, a storage address, or any data (such as bit sequence or individual characters).

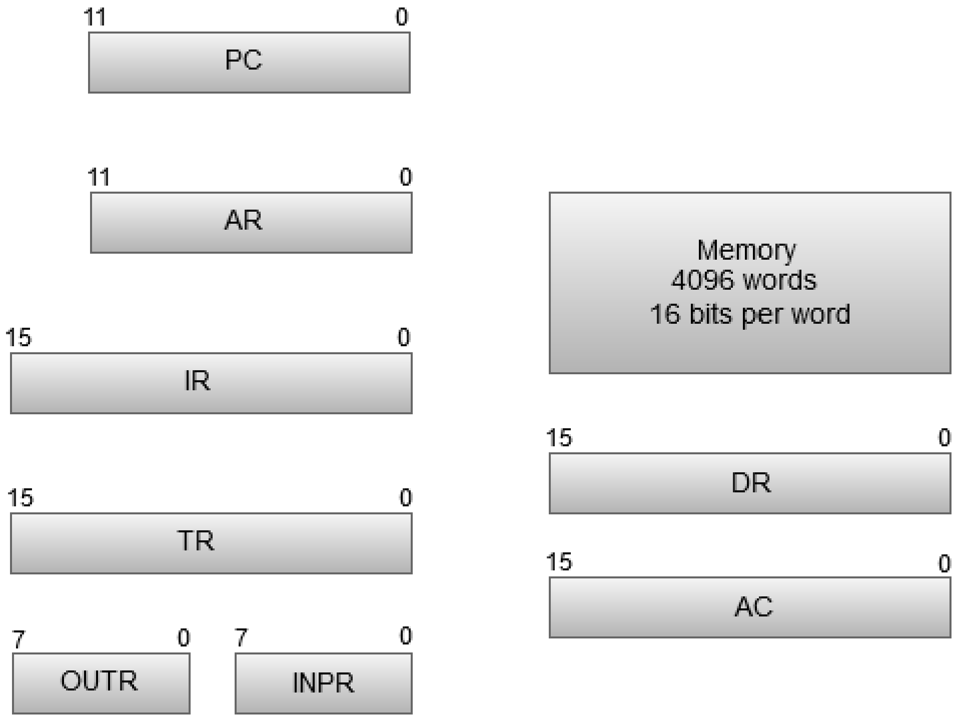
The computer needs processor registers for manipulating data and a register for holding a memory address. The register holding the memory location is used to calculate the address of the next instruction after the execution of the current instruction is completed.

Following is the list of some of the most common registers used in a basic computer:

Register	Symbol	Number of bits	Function
Data register	DR	16	Holds memory operand
Address register	AR	12	Holds address for the memory
Accumulator	AC	16	Processor register
Instruction register	IR	16	Holds instruction code
Program counter	PC	12	Holds address of the instruction
Temporary register	TR	16	Holds temporary data
Input register	INPR	8	Carries input character
Output register	OUTR	8	Carries output character

The following image shows the register and memory configuration for a basic computer.

Register and Memory Configuration of a basic computer:



- The Memory unit has a capacity of 4096 words, and each word contains 16 bits.
- The Data Register (DR) contains 16 bits which hold the operand read from the memory location.
- The Memory Address Register (MAR) contains 12 bits which hold the address for the memory location.
- The Program Counter (PC) also contains 12 bits which hold the address of the next instruction to be read from memory after the current instruction is executed.
- The Accumulator (AC) register is a general purpose processing register.
- The instruction read from memory is placed in the Instruction register (IR).
- The Temporary Register (TR) is used for holding the temporary data during the processing.
- The Input Registers (IR) holds the input characters given by the user.
- The Output Registers (OR) holds the output after processing the input data.

Common Bus System

The basic computer has 8 registers, a memory unit and a control unit. Paths must be provided to transfer data from one register to another. An efficient method for transferring data in a system is to use a **Common Bus System**. The output of registers and memory are connected to the common bus.

Paths must be provided to transfer information from one register to another and between memory and registers. The number of wires will be excessive if connections are made between the outputs of each register and the inputs of the other registers. A more efficient scheme for transferring information in a system with many registers is to use a common bus. We can construct a bus system using multiplexers or three-state buffer gates. We just have to connect the registers and memory of the basic computer to a common bus system.

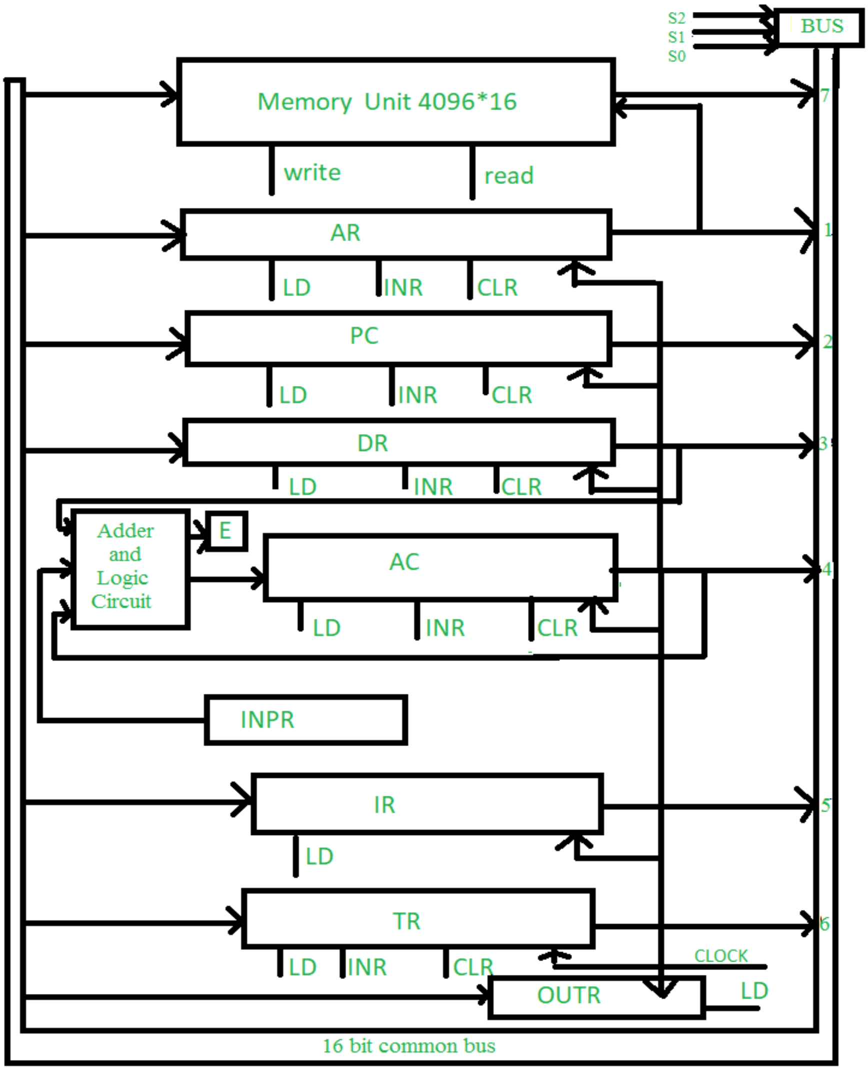
The outputs of seven registers and memory are connected to the common bus. The specific output that is selected for the bus lines at any given time is determined from the binary value of the selection variables S2, S1, and S0. The number along each output shows the decimal equivalent of the required binary selection. For example, the number along the output of DR is 3. The 16-bit outputs of DR are placed on the bus lines when S2S1S0 = 011 since this is the binary value of decimal 3.

The lines from the common bus are connected to the inputs of each register and the data inputs of the memory. The particular register whose LD (load) input is enabled receives the data from the bus during the next clock pulse transition. The memory receives the contents of the bus when its write input is activated. The memory places its 16-bit output onto the bus when the read input is activated and S2S1S0 = 111.

Four registers, DR, AC, IR, and TR, have 16 bits each. Two registers, AR and PC, have 12 bits each since they hold a memory address. When the contents of AR or PC are applied to the 16-bit common bus, the four most significant bits are set to O's. When AR or PC receive information from the bus, only the 12 least significant bits are transferred into the register.

The input register INPR and the output register **OUTR** have 8 bits each and communicate with the eight least significant bits in the bus. INPR is connected to provide information to the bus but OUTR can only receive information from the bus. This is because INPR receives a character from an input device which is then transferred to AC.

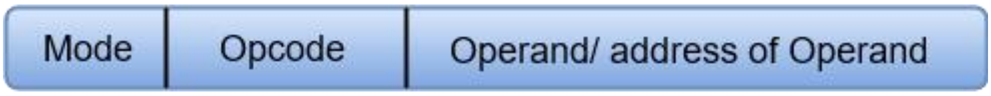
The 16 inputs of AC come from an adder and logic circuit. This circuit has three sets of inputs. One set of 16-bit inputs come from the outputs of AC . They are used to implement register rnlcrooperations such as complement AC and shift AC . Another set of 16-bit inputs come from the data register DR. The inputs from DR and AC are used for arithmetic and logic microoperations, such as add DR to AC or AND DR to AC . The result of an addition is transferred to AC and the end carry-out of the addition is transferred to flip-flop E (Extended AC bit). A third set of 8-bit inputs come from the input register INPR.



Computer Instructions

Computer instructions are a set of machine language instructions that a particular processor understands and executes. A computer performs tasks on the basis of the instruction provided. An instruction comprises of groups called fields. These fields include:

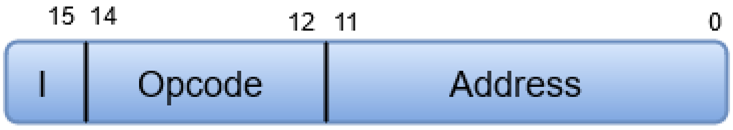
- The Operation code (Opcode) field which specifies the operation to be performed.
- The Address field which contains the location of the operand, i.e., register or memory location.
- The Mode field which specifies how the operand will be located.



A basic computer has three instruction code formats which are:

1. Memory - reference instruction
2. Register - reference instruction
3. Input-Output instruction

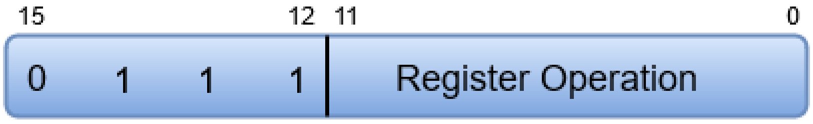
Memory - reference instruction



(Opcode = 000 through 110)

In Memory-reference instruction, 12 bits of memory is used to specify an address and one bit to specify the addressing mode 'I'.

Register - reference instruction

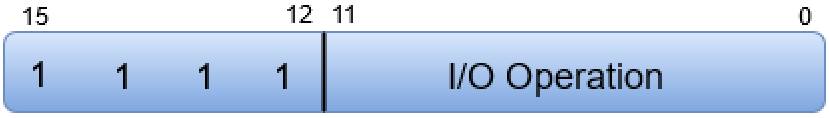


(Opcode = 111, I = 0)

The Register-reference instructions are represented by the Opcode 111 with a 0 in the leftmost bit (bit 15) of the instruction.

A Register-reference instruction specifies an operation on or a test of the AC (Accumulator) register.

Input-Output instruction



(Opcode = 111, I = 1)

Just like the Register-reference instruction, an Input-Output instruction does not need a reference to memory and is recognized by the operation code 111 with a 1 in the leftmost bit of the instruction. The remaining 12 bits are used to specify the type of the input-output operation or test performed.

Instruction Set Completeness

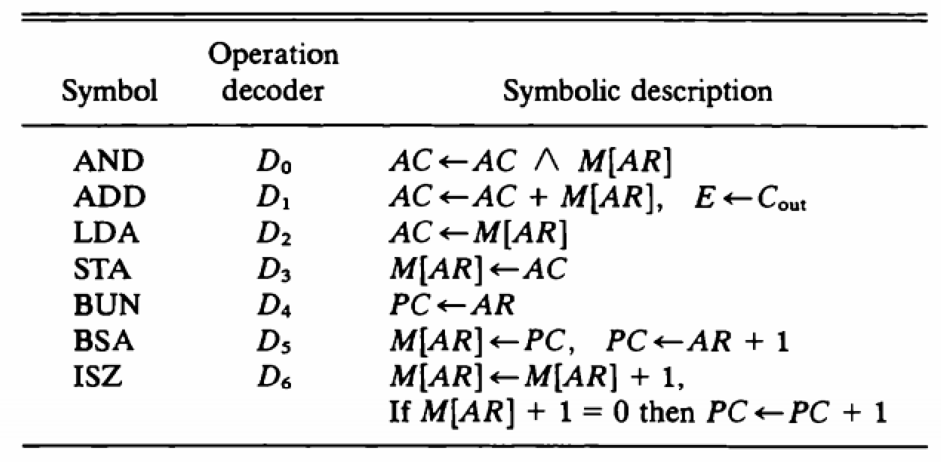
A set of instructions is said to be complete if the computer includes a sufficient number of instructions in each of the following categories:

- Arithmetic, logical and shift instructions
- A set of instructions for moving information to and from memory and processor registers.
- Instructions which controls the program together with instructions that check status conditions.
- Input and Output instructions

In order to specify the micro operations needed for the execution of each operation, it is necessary that the function intended to perform be defined precisely.

Memory- Reference Instructions

In order to specify the micro operations needed for the execution of each operation, it is necessary that the function intended to perform be defined precisely. The decoded output D_i for i=0,1,2,3,4,5,6 from the operation decoder that belongs to each instruction is included in the following table.



Symbol	Operation decoder	Symbolic description
AND	D_0	$AC \leftarrow AC \wedge M[AR]$
ADD	D_1	$AC \leftarrow AC + M[AR], \quad E \leftarrow C_{out}$
LDA	D_2	$AC \leftarrow M[AR]$
STA	D_3	$M[AR] \leftarrow AC$
BUN	D_4	$PC \leftarrow AR$
BSA	D_5	$M[AR] \leftarrow PC, \quad PC \leftarrow AR + 1$
ISZ	D_6	$M[AR] \leftarrow M[AR] + 1,$ If $M[AR] + 1 = 0$ then $PC \leftarrow PC + 1$

AND to AC

This is an instaruction that perform the AND logic operation on pairs of bits in AC and the memory word specified by the effective address. The result of the operation is transferred to AC . The micro operations that execute this instruction are:

D0T4:DR<-M[AR]
D0T5:AC<-AC/\DR, SC<---0

The control function for this instruction uses the operation decoder D0 since this output of the decoder is active when the instruction has an AND operation whose binary code value 000. Two timing signals are needed to execute the instruction. The clock transition associated with timing signal T4 transfers the operand from memory into DR . The clock transition associated with the next timing signal T5 transfers to AC the result of the AND logic operation between the contents of DR and AC. The same clock transition clears SC to 0, transferring control to timing signal T0 to start a new instruction cycle.

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_{out} is transferred to the E (extended accumulator) flip-flop. The rnicrooperations needed to execute this instruction are

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

Same Two timing signals, T, and T5, are used again but with operation decoder D1 instead of D0, which was used for the AND instruction. After the instruction is fetched from memory and decoded, only one output of the operation decoder will be active, and that output determines the sequence of microoperations that the control follows during the execution of a memory-reference instruction.

LDA: Load to AC

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

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

Looking back at the bus system shown in Figure 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.

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:

D3T4: M [AR] ← AC, SC ← 0

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:

D4T4: PC ← AR, SC ← 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 .

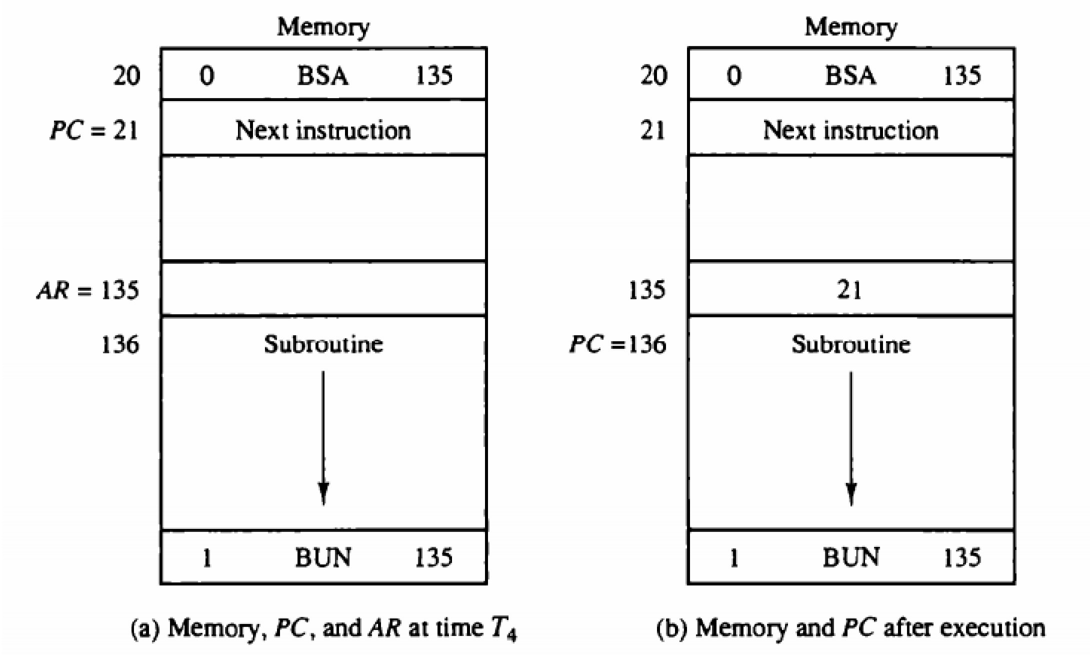
BSA: Branch and Save Return Address

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] <-- PC, PC <-- AR + I

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] <-- 21, PC <-- 135 + 1 = 136



(a) Memory, PC, and AR at time T_4 (b) Memory and PC after execution

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 instruction at the return address.

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:

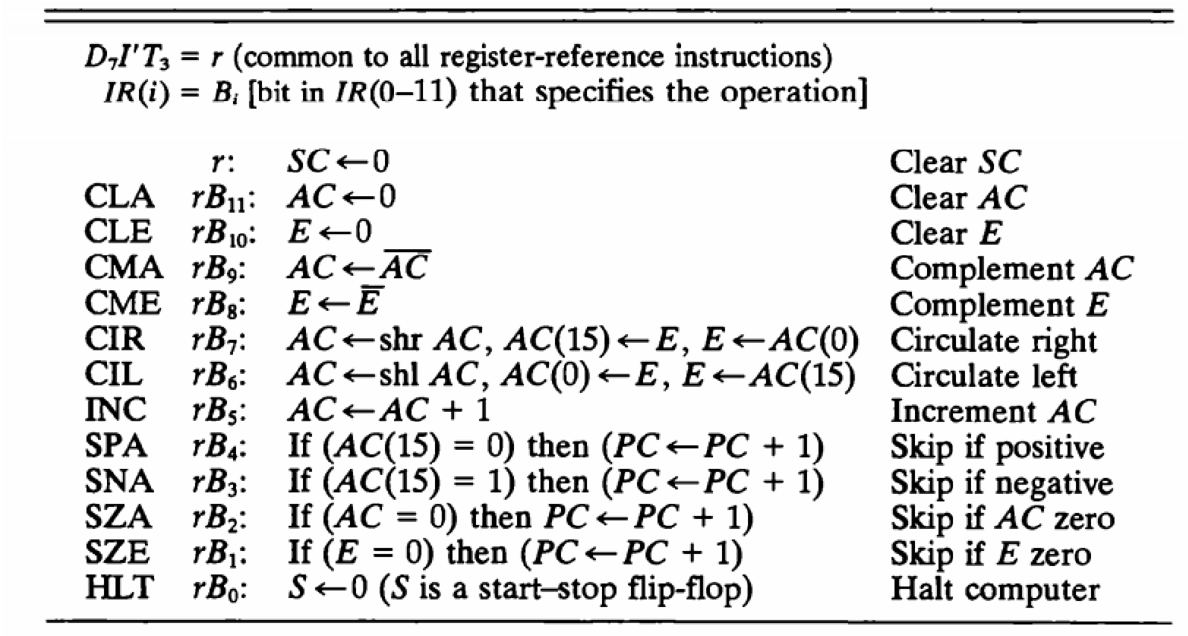
D6T4: DR<--M[AR]
D6T5: DR<--DR+1
D,T,: M[AR]<--DR, if (DR = 0) then (PC ← PC + 1), SC ← 0

Register – Reference Instructions

Register-reference instructions are recognized by the control when 07 = 1 and I = 0. These instructions use bits 0 through 11 of the instruction code to specify one of 12 instructions. These 12 bits are available in IR(0-11). They were also transferred to AR during time T2•

The control functions and microoperations for the register-reference instructions are. listed in the following Table. These instructions are executed with the clock transition associated with timing variable T3. Each control function needs the Boolean relation D7I'T3, which we designate for convenience by the symbol r. The control function is distinguished by one of the bits in IR(0-11). By assigning the symbol B, to bit i of IR, all control functions can be simply denoted by rB;.

For example, the instruction CLA has the hexadecimal code 7800 (see Table 5-2), which gives the binary equivalent 011I 1000 0000 0000. The first bit is a zero and is equivalent to I'. The next three bits constitute the operation code and are recognized from decoder output D7. Bit 11 in IR is I and is recognized from 811. The control function that initiates the rnicrooperation for this instruction is D7I'T3B11 = rB11. The execution of a register-reference instruction is completed at time T3. The sequence counter SC is cleared to 0 and the control goes back to fetch the next instruction with timing signal T0.



$D_7I'T_3 = r$ (common to all register-reference instructions)
$IR(i) = B_i$ [bit in $IR(0–11)$ that specifies the operation]

	r:	$SC \leftarrow 0$	Clear SC
CLA	rB_{11}:	$AC \leftarrow 0$	Clear AC
CLE	rB_{10}:	$E \leftarrow 0$	Clear E
CMA	rB_9:	$AC \leftarrow \overline{AC}$	Complement AC
CME	rB_8:	$E \leftarrow \overline{E}$	Complement E
CIR	rB_7:	$AC \leftarrow \text{shr } AC,\ AC(15) \leftarrow E,\ E \leftarrow AC(0)$	Circulate right
CIL	rB_6:	$AC \leftarrow \text{shl } AC,\ AC(0) \leftarrow E,\ E \leftarrow AC(15)$	Circulate left
INC	rB_5:	$AC \leftarrow AC + 1$	Increment AC
SPA	rB_4:	If $(AC(15) = 0)$ then $(PC \leftarrow PC + 1)$	Skip if positive
SNA	rB_3:	If $(AC(15) = 1)$ then $(PC \leftarrow PC + 1)$	Skip if negative
SZA	rB_2:	If $(AC = 0)$ then $PC \leftarrow PC + 1)$	Skip if AC zero
SZE	rB_1:	If $(E = 0)$ then $(PC \leftarrow PC + 1)$	Skip if E zero
HLT	rB_0:	$S \leftarrow 0$ (S is a start–stop flip-flop)	Halt computer

Timing and Control

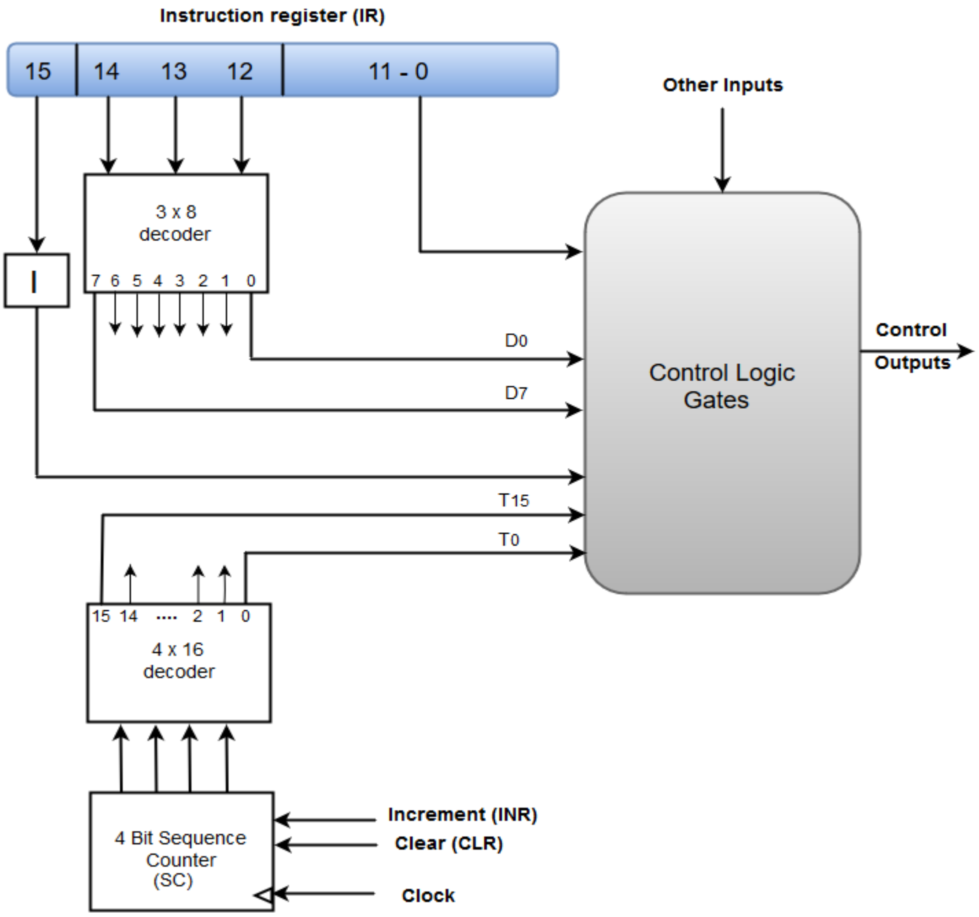
The timing for all registers in the basic computer is controlled by a master clock generator. The clock pulses are applied to all flip-flops and registers in the system, including the flip-flops and registers in the control unit. The clock pulses do not change the state of a register unless the register is enabled by a control signal. The control signals are generated in the control unit and provide control inputs for the multiplexers in the common bus, control inputs in processor registers, and microoperations for the accumulator.

There are two major types of control organization:

1. hardwired control and
2. microprogrammed control.

In the hardwired organization, the control logic is implemented with gates, flip-flops, decoders, and other digital circuits. It has the advantage that it can be optimized to produce a fast mode of operation. In the microprogrammed organization, the control information is stored in a control memory. The control memory is programmed to initiate the required sequence of microoperations. A hardwired control, as the name implies, requires changes in the wiring among the various components if the design has to be modified or changed.

Control Unit of a Basic Computer:



In the microprogrammed control, any required changes or modifications can be done by updating the microprogram in control memory.

Instruction Cycle

A program residing in the memory unit of a computer consists of a sequence of instructions. These instructions are executed by the processor by going through a cycle for each instruction.

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

1. Fetch instruction from memory.
2. Decode the instruction.
3. Read the effective address from memory.
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 and Decode

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 T0. After each clock pulse, SC is incremented by one, so that the timing signals go through a sequence T0, T1, T2, and so on. The microoperations for the fetch and decode phases can be specified by the following register transfer statements.

T0: AR ← PC

T1: IR ← M[AR], PC ← PC + 1

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

Since only AR is connected to the address inputs of memory, it is necessary to transfer the address from PC to AR during the clock transition associated with timing signal T0. The instruction read from memory is then placed in the instruction register IR with the clock transition associated with timing signal T1.

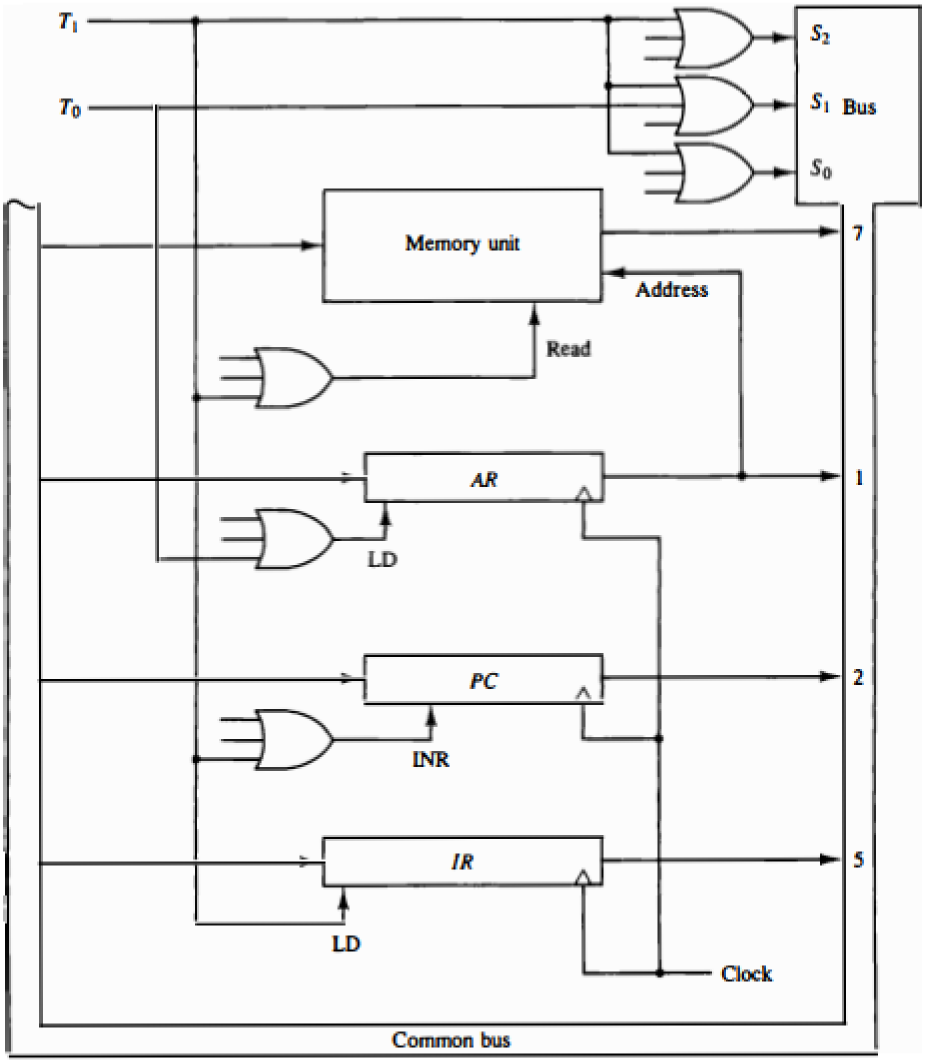


Figure Register transfers for the fetch phase.

At the same time, PC is incremented by one to prepare it for the address of the next instruction in the program. At time T2, the operation code in IR is decoded, the indirect bit is transferred to flip-flop I, and the address part of the instruction is transferred to AR. Note that SC is incremented after each clock pulse to produce the sequence T0, T1, and T2. Figure above shows how the first two register transfer statements are implemented in the bus system. To provide the data path for the transfer of PC to AR we must apply timing signal T0 to achieve the following connection:

1. Place the content of PC onto the bus by making the bus selection inputs

$S_2S_1S_0$ equal to 010.

2. Transfer the content of the bus to AR by enabling the LD input of AR .

The next clock transition initiates the transfer from PC to AR since T0 = 1. In order to implement the second statement : T1: IR ← M[AR], PC ← PC + 1 it is necessary to use timing signal T1 to provide the following connections in the bus system.

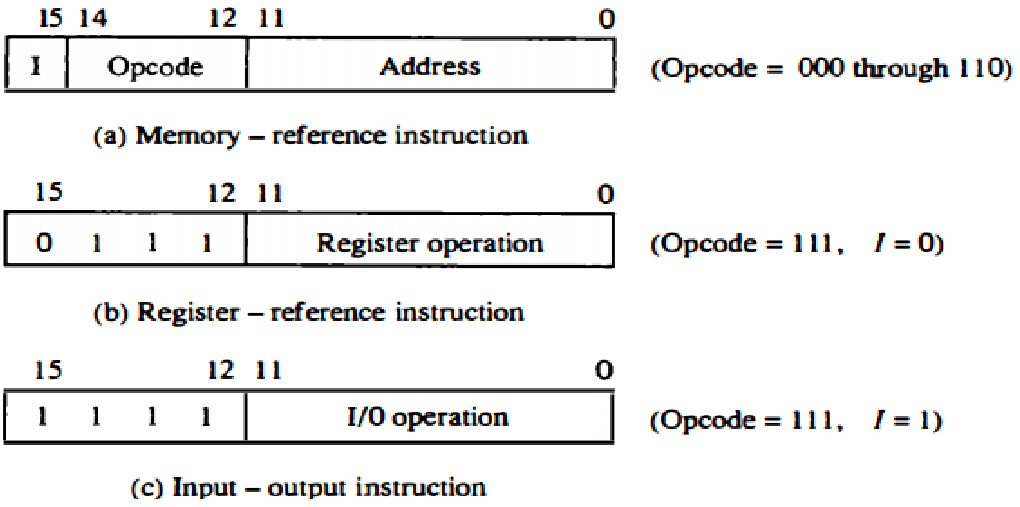
1. Enable the read input of memory.
2. Place the content of memory onto the bus by making $S_2S_1S_0 = 111$.
3. Transfer the content of the bus to IR by enabling the LD input of IR.
4. Increment PC by enabling the INR input of PC.

The next clock transition initiates the read and increment operations since T1 = 1. Figure above duplicates a portion of the bus system and shows how T0 and T1 are connected to the control inputs of the registers, the memory, and the bus selection inputs. Multiple input OR gates are included in the diagram because there are other control functions that will initiate similar operations.

Determine the Type of Instruction

The timing signal that is active after the decoding is T_3. During time T_3, the control unit determines the type of instruction that was just read from memory. The flowchart of Fig. below presents an initial configuration for the instruction cycle and shows how the control determines the instruction type after the decoding. The three possible instruction types available in the basic computer are specified in Fig. on basic computer formats.

Figure Basic computer instruction formats.

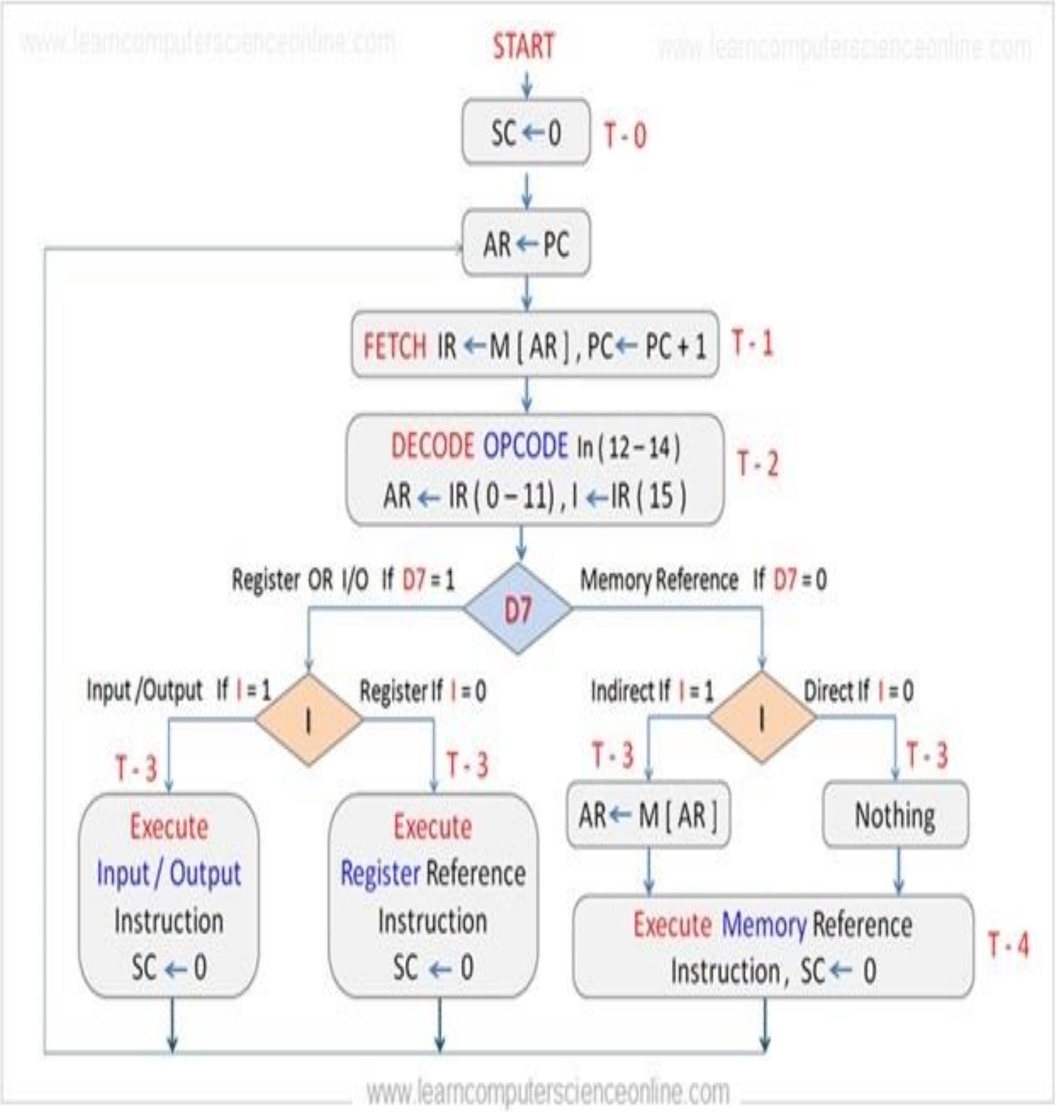


(a) Memory – reference instruction

(b) Register – reference instruction

(c) Input – output instruction

Decoder output D_7 is equal to 1 if the operation code is equal to binary 111. From Fig. on basic computer formats we determine that if $D_7 = 1$, the instruction must be a register-reference or input-output type.


www.learncomputerscienceonline.com
START
SC ← 0
T - 0
AR ← PC
FETCH IR ← M [AR] , PC ← PC + 1
T - 1
DECODE OPCODE In (12 – 14)
AR ← IR (0 – 11) , I ← IR (15)
T - 2
Register OR I/O If D7 = 1
D7
Memory Reference If D7 = 0
Input /Output If I = 1
I
Register If I = 0
Indirect If I = 1
I
Direct If I = 0
T - 3
T - 3
T - 3
T - 3
Execute Input / Output Instruction SC ← 0
Execute Register Reference Instruction SC ← 0
AR ← M [AR]
Nothing
Execute Memory Reference Instruction, SC ← 0
T - 4
www.learncomputerscienceonline.com

If $D_7 = 0$, the operation code must be one of the other seven values 000 through 110, specifying a memory-reference instruction. Control then inspects the value of the first bit of the instruction, which is now available in flip-flop I. If D7 = 0 and I = 1, we have a memory reference instruction with an indirect address.

It is then necessary to read the effective address from memory. The microoperation for the indirect address condition can be symbolized by the register transfer statement : AR ← M[AR]. Initially, AR holds the address part of the instruction. This address is used during the memory read operation. The word at the address given by AR is read from memory and placed on the common bus. The LD input of AR is then enabled to receive the indirect address that resided in the 12 least significant bits of the memory word.

The three instruction types are subdivided into four separate paths. The selected operation is activated with the clock transition associated with timing signal T_3. This can be symbolized as follows:

$D'_7 IT_3$: AR ← M[AR]

$D'_7 I'T_3$: Nothing

$D_7 I'T_3$: Execute a register-reference instruction

D_7IT_3: Execute an input-output instruction

When a memory-reference instruction with I = 0 is encountered, it is not necessary to do anything since the effective address is already in AR. However, the sequence counter SC must be incremented when $D'_7T_3 = 1$, so that the execution of the memory-reference instruction can be continued with timing variable $T_{4.}$

A register-reference or input-output instruction can be executed with the clock associated with timing signal T_3. After the instruction is executed, SC is cleared to 0 and control returns to the fetch phase with $T_0 = 1$. Note that the sequence counter SC is either incremented or cleared to 0 with every positive clock transition. We will adopt the convention that if SC is incremented, we will not write the statement SC ← SC + 1, but it will be implied that the control goes to the next timing signal in sequence. When SC is to be cleared, we will include the statement SC ← 0. The register transfers needed for the execution of the register-reference instructions are presented in this section.

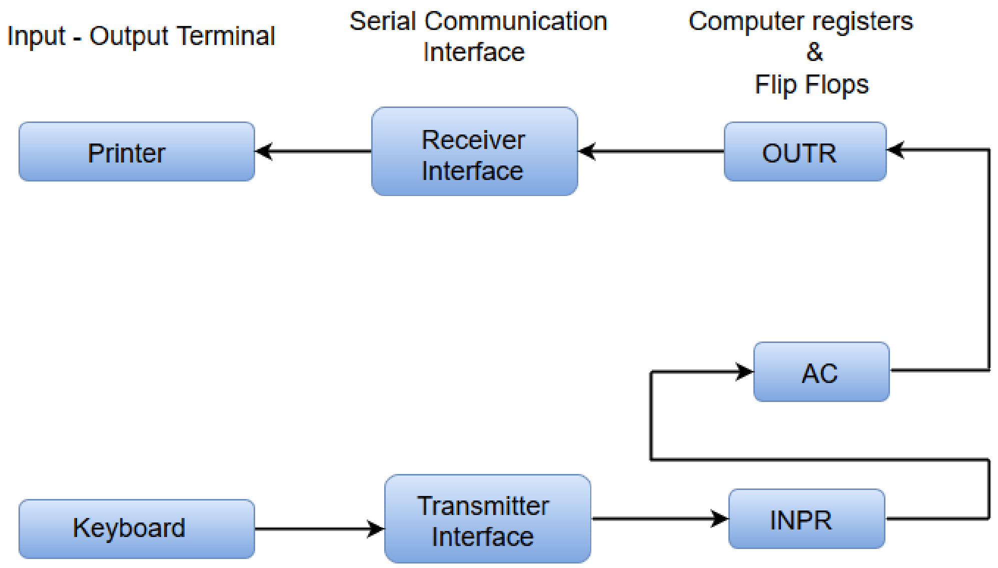
Input-Output and Interrupt

A computer can serve no useful purpose unless it communicates with the external environment. Instructions and data stored in memory must come from some input device. Computational results must be transmitted to the user through some output device. Commercial computers include many types of input and output devices. To demonstrate the most basic requirements for input and output communication, we will use as an illustration a terminal unit with a keyboard and printer.

Input-Output Configuration

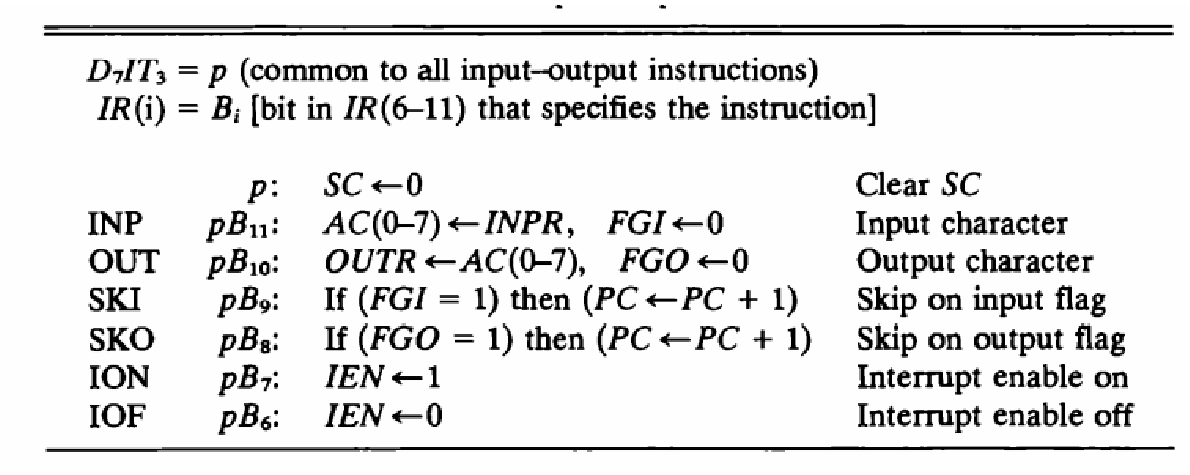
The terminal sends and receives serial information. Each quantity of information has eight bits of an alphanumeric code. The serial information from the keyboard is shifted into the input register INPR. The serial information for the printer is stored in the output register OUTR. These two registers communicate with a communication interface serially and with the AC in parallel. The input-output configuration is shown in Figure. The transmitter interface receives serial information from the keyboard and transmits it to INPR. The receiver interface receives information from OUTR and sends it to the printer serially.

Input - Output Configuration:



The input register INPR consists of eight bits and holds an alphanumeric input information. The 1-bit 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. The flag is needed to synchronize the timing rate difference between the input device and the computer. The process of information transfer is as follows. Initially, the input flag FGI is cleared to 0. When a key is struck in the keyboard, an 8-bit alphanumeric code is shifted into INPR and the input flag FGI is set to 1. As long as the flag is set, the information in INPR cannot be changed by striking another key. The computer checks the flag bit; if it is 1, the information from INPR is transferred in parallel into AC and FGI is cleared to 0. Once the flag is cleared, new information can be shifted into INPR by striking another key.

Input-Output Reference Instructions



$D_7IT_3 = p$ (common to all input–output instructions)
$IR(i) = B_i$ [bit in $IR(6–11)$ that specifies the instruction]

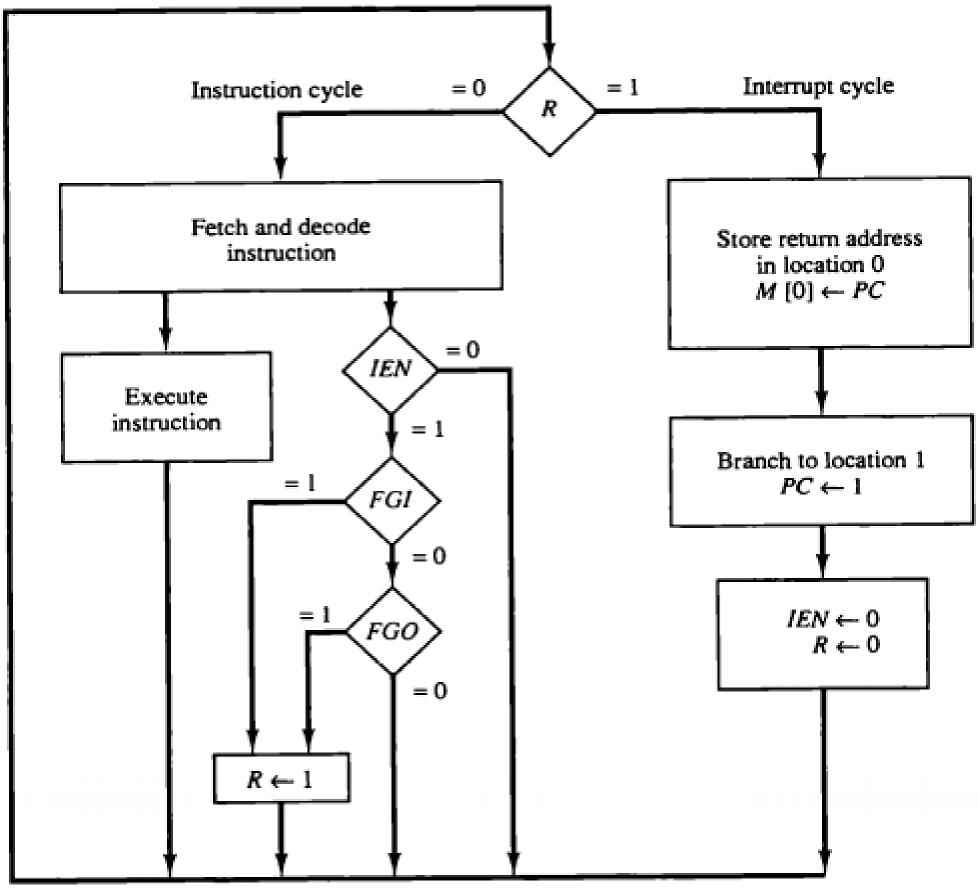
	p:	$SC \leftarrow 0$	Clear SC
INP	pB_{11}:	$AC(0–7) \leftarrow INPR, \quad FGI \leftarrow 0$	Input character
OUT	pB_{10}:	$OUTR \leftarrow AC(0–7), \quad FGO \leftarrow 0$	Output character
SKI	pB_9:	If $(FGI = 1)$ then $(PC \leftarrow PC + 1)$	Skip on input flag
SKO	pB_8:	If $(FGO = 1)$ then $(PC \leftarrow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

Program Interrupt

The computer keeps checking the flag bit, and when it finds it set, it initiates an information transfer. The difference of information flow rate between the computer and that of the input—output device makes this type of transfer inefficient. An alternative to the programmed controlled procedure is to let the external device inform the computer when it is ready for the transfer. In the meantime the computer can be busy with other tasks. This type of transfer uses the interrupt facility. While the computer is running a program, it does not check the flags. When a flag is set, the computer is momentarily interrupted from the current program. The computer deviates momentarily from what it is doing to perform of the input or output transfer. It then returns to the current program to continue what it was doing before the interrupt.

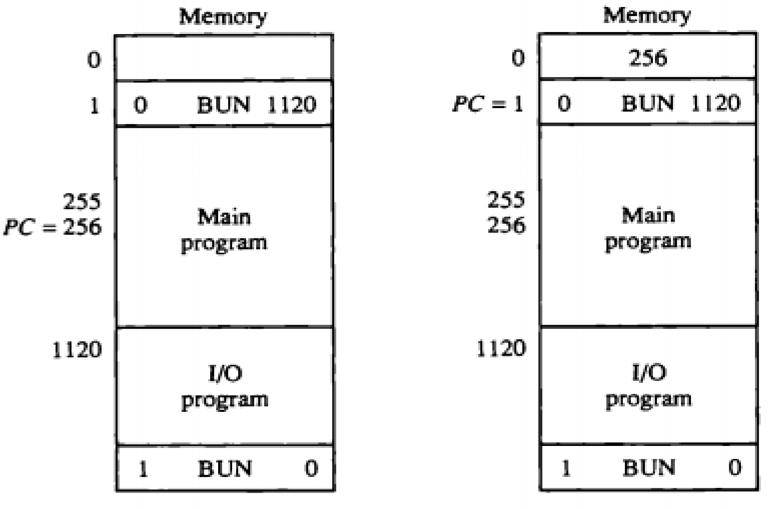
The interrupt enable flip-flop IEN can be set and cleared with two instructions. When IEN is cleared to 0 (with the IOF instruction), the flags cannot interrupt the computer. When IEN is set to (with the ION instruction), the computer can be interrupted. The way that the interrupt is handled by the computer can be explained by means of the flowchart.

An interrupt flip-flop R is included in the computer. When R = 0, the computer goes through an instruction cycle. During the execute phase of the instruction cycle IEN is checked by the control. If it is 0, it indicates that the programmer does not want to use the interrupt,so control continues with the next instruction cycle. If IEN is 1, control checks the flag bits. If both flags are 0, it indicates that neither the input nor the output registers are ready for transfer of information. In this case, control continues with the next instruction cycle. If either flag is set to 1 while 1EN = 1, flip-flop R is set to 1. At the end of the execute phase, control checks the value of R, and if it is equal to 1, it goes to an interrupt cycle instead of an instruction cycle.



Interrupt cycle

The interrupt cycle is a hardware implementation of a branch and save return address operation. The return address available in PC is stored in a specific location. This location may be a processor register, a memory stack, or a specific memory location. An example that shows what happens during the interrupt cycle is shown in the following Figure.



(a) Before interrupt (b) After interrupt cycle

When an interrupt occurs and R is set to 1 while the control is executing the instruction at address 255. At this time, the returns address 256 is in PC. The programmer has previously placed an input—output service program in memory starting from address 1120 and a BUN 1120 instruction at address 1. This is shown in Figure(a). When control reaches timing signal T0and finds that R = 1, it proceeds with the interrupt cycle. The content of PC (256) is stored in memory location 0, PC is set to 1, and R is cleared to 0. The branch instruction at address 1 causes the program to transfer to the input—output service program at address 1120. This program checks the flags, determines which flag is set, and then transfers the required input or output information. Once this is done, the instruction ION is executed to set IEN to 1 (to enable further interrupts), and the program returns to the location where it was interrupted. This is shown in Figure(b).

UNIT-2

Data Representation, Basic Computer Organization

Data Types

The data types found in the registers of digital computers may be classified as being one of the following categories:

(1) Numbers used in arithmetic computations,

(2) Letters of the alphabet used in data processing. and

(3) Other discrete symbols used for specific purposes.

All types of data, except binary numbers, are represented in computer registers in binary coded form. This is because registers are made up of flip-flops and flip-flops are two-state devises that can store only l's and 0's. The binary number system is the most natural system to use in a digital computer. But sometimes it is convenient to employ different number systems, especially the decimal number system, since it is used by people to perform arithmetic computations.

Number Systems

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

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

that is, 7 hundreds, plus 2 tens, plus 4 units, plus 5 tenths. Every decimal number can be similarly interpreted to find the quantity it represents. The binary number system uses the radix 2. The two digit symbols used are 0 and 1. The string of digits 101101 is interpreted to represent the quantity

$$1 \text{ x}2^5+0\text{x}2^4+1\text{x}2^3+1\text{x}2^2+0\text{x}2^1+1\text{x}2^0=45$$

To distinguish between different radix numbers, the digits will be enclosed in parentheses and the radix of the number inserted as a subscript. For example, to show the equality between decimal and binary forty-five we will write $(101101)_2 = (45)_{10}$.

Besides the decimal and binary number systems, the octal (radix 8) and hexadecimal (radix 16) are important in digital computer work. The eight symbols of the octal system are 0, 1, 2, 3, 4, 5, 6, and 7. The 16 symbols of the hexadecimal system are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. The last six symbols are, unfortunately, identical to the letters of the alphabet and can cause confusion at times. However, this is the convention that has been adopted. When used to represent hexadecimal digits, the symbols A, B, C, D, E, F correspond to the decimal numbers 10, 11, 12, 13, 14, 15, respectively. A number in radix r can be converted to the familiar decimal system by forming the sum of the weighted digits. For example, octal 736.4 is converted to decimal as follows:

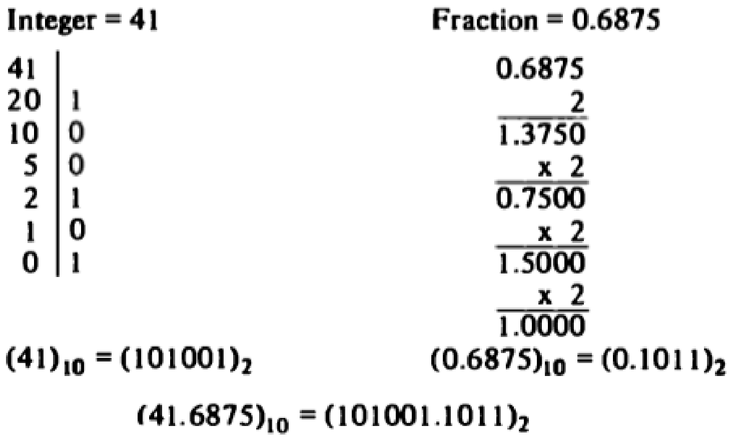
$$(736.4)_8 = 7 \text{ X } 8^2 + 3 \text{ X } 8^1 + 6 \text{ X } 8^0 + 4 \text{ X } 8^{-1} = 7 \text{ X } 64 + 3 \text{ X } 8 + 6 \text{ X } 1 + 4/8 = (478.5)_{10}$$

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

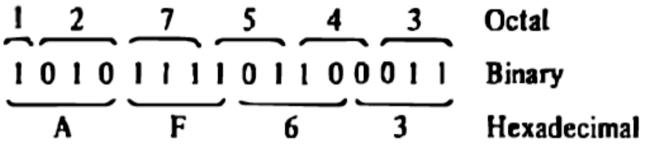
$$(F3)_{16} = F \text{ X } 16 + 3 = 15 \text{ X } 16 + 3 = (243)_{10}$$

Octal and Hexadecimal Numbers

The conversion from Decimal to Binary, Octal, and Hexadecimal representation plays an important part in digital computers. Since $2^3 = 8$ and $2^4 = 16$, each octal digit corresponds to three binary digits and each hexadecimal digit corresponds to four binary digits. The conversion from binary to octal is easily accomplished by partitioning the binary number into groups of three bits each. The corresponding octal digit is then assigned to each group of bits and the string of digits so obtained gives the octal equivalent of the binary number. Consider, for example, a 16-bit register.



Conversion of decimal 41.6875 into binary.



Binary, octal, and hexadecimal conversion

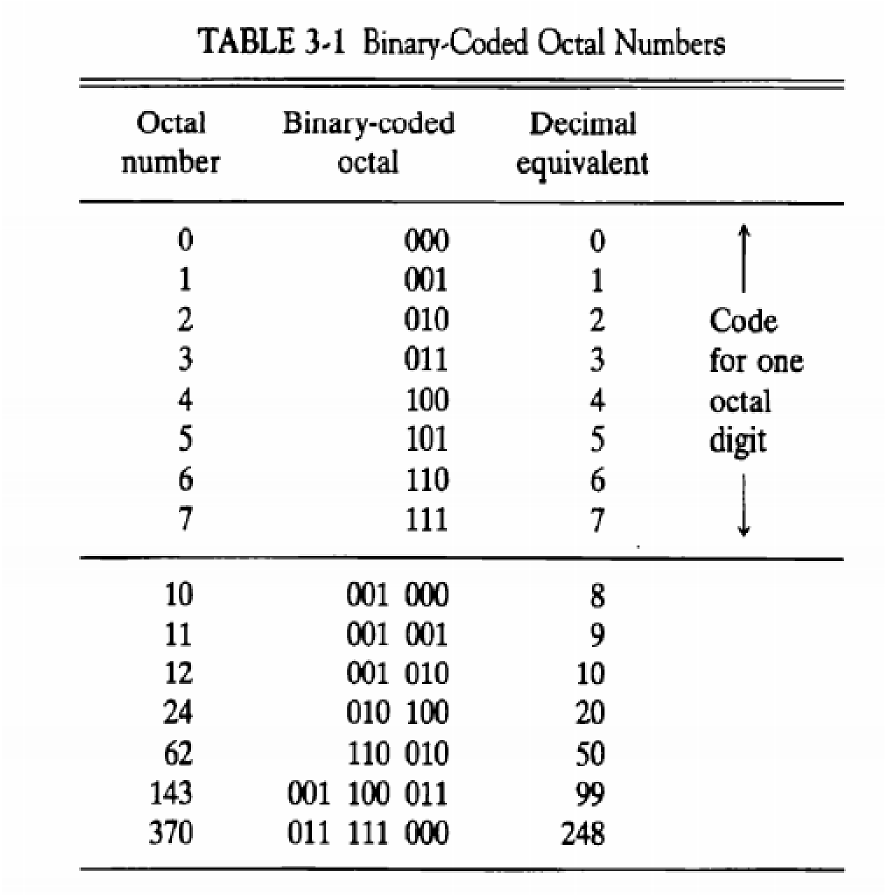
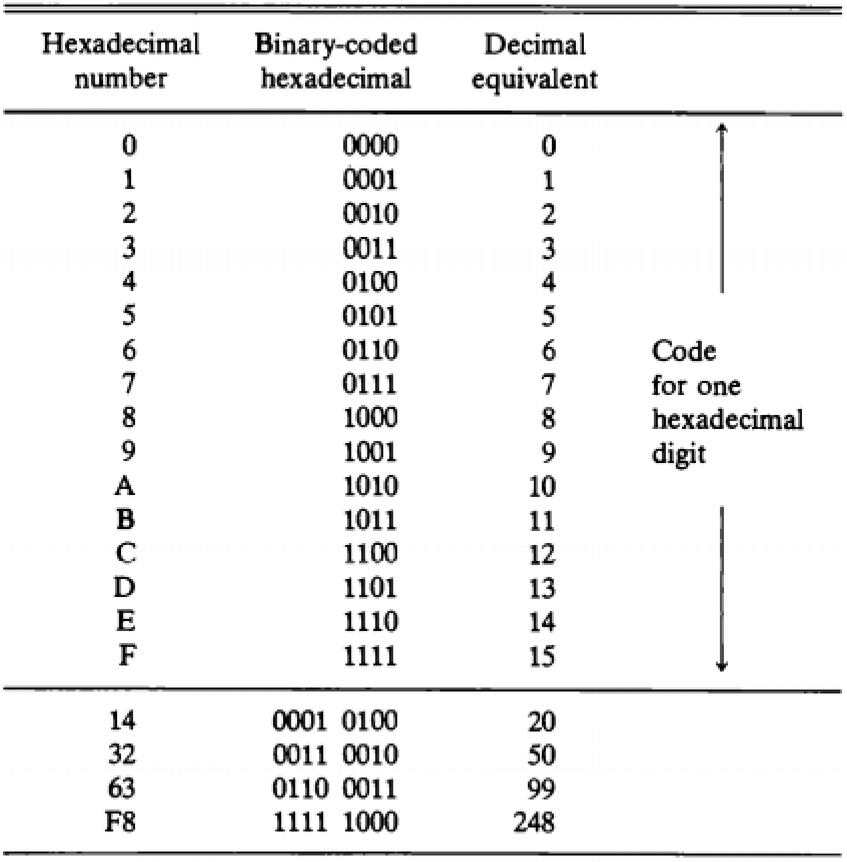


TABLE 3-1 Binary-Coded Octal Numbers

Octal number	Binary-coded octal	Decimal equivalent	
0	000	0	↑ Code for one octal digit ↓
1	001	1	
2	010	2	
3	011	3	
4	100	4	
5	101	5	
6	110	6	
7	111	7	
10	001 000	8	
11	001 001	9	
12	001 010	10	
24	010 100	20	
62	110 010	50	
143	001 100 011	99	
370	011 111 000	248	

TABLE 3-2 Binary-Coded Hexadecimal Numbers



Hexadecimal number	Binary-coded hexadecimal	Decimal equivalent	
0	0000	0	Code for one hexadecimal digit
1	0001	1	
2	0010	2	
3	0011	3	
4	0100	4	
5	0101	5	
6	0110	6	
7	0111	7	
8	1000	8	
9	1001	9	
A	1010	10	
B	1011	11	
C	1100	12	
D	1101	13	
E	1110	14	
F	1111	15	
14	0001 0100	20	
32	0011 0010	50	
63	0110 0011	99	
F8	1111 1000	248	

Decimal Representation

Binary Number System is the most natural system for a computer, but the people are accustomed to the decimal system. One way to solve this conflict is to convert all input decimal numbers into binary numbers, let the computer perform all the operations in binary and convert the result into decimal. It is also possible for the computer to perform all arithmetic operations directly with decimal numbers provided they are placed in the registers in a coded form.

It is important to understand the difference between the conversion of decimal numbers into binary and the binary coding of decimal numbers. For example when converted into binary number, the decimal number 99 is represented as 1100011. But when represented in BCD it is

1001 1001

Few decimal numbers and their representation in BCD are as follows

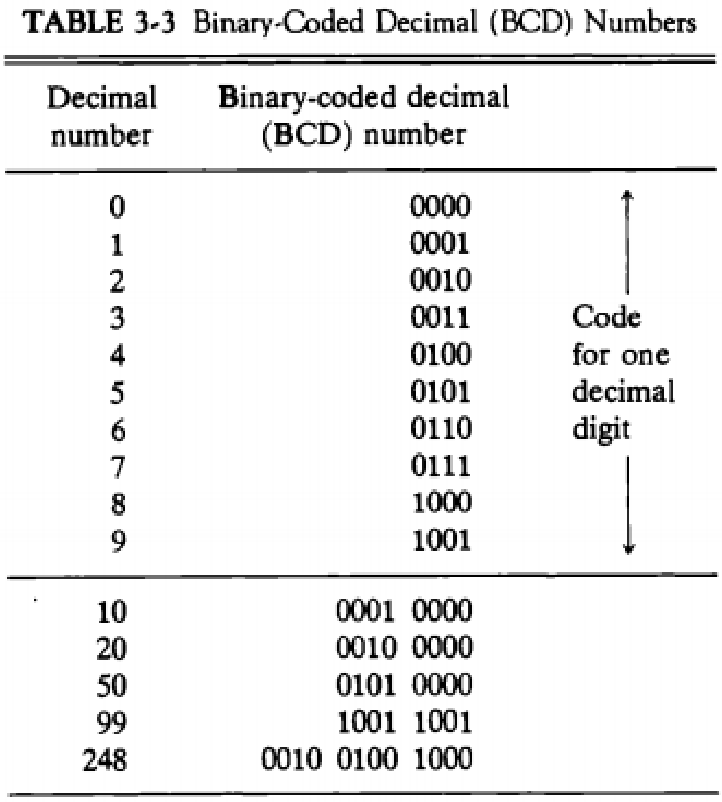


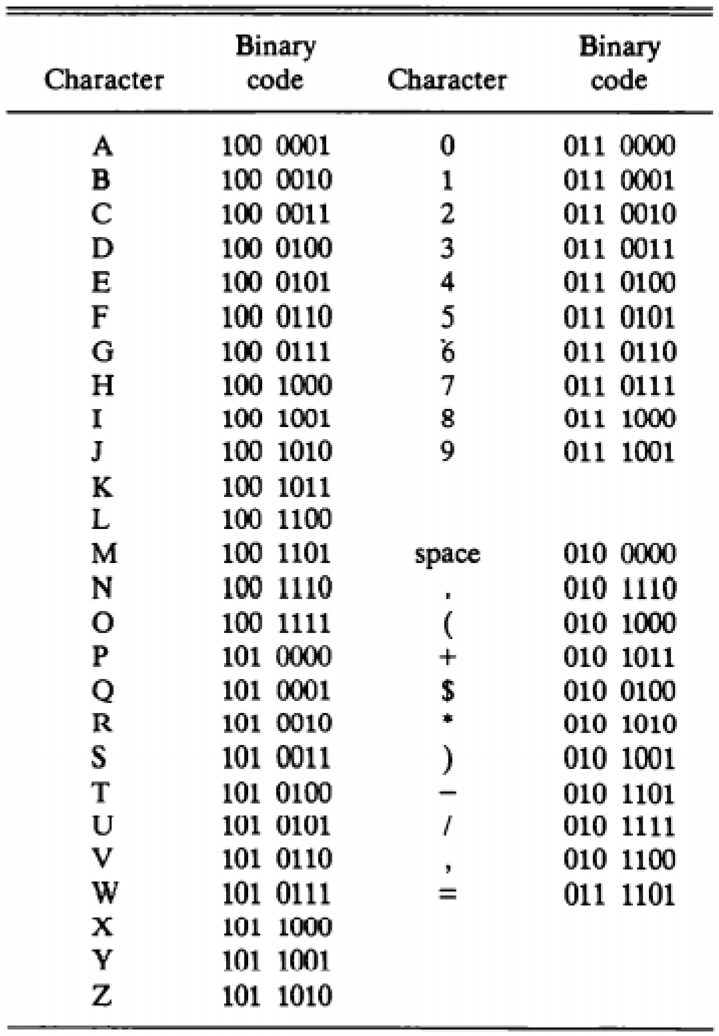
TABLE 3-3 Binary-Coded Decimal (BCD) Numbers

Decimal number	Binary-coded decimal (BCD) number	
0	0000	Code for one decimal digit
1	0001	
2	0010	
3	0011	
4	0100	
5	0101	
6	0110	
7	0111	
8	1000	
9	1001	
10	0001 0000	
20	0010 0000	
50	0101 0000	
99	1001 1001	
248	0010 0100 1000	

Alphanumeric Representation

Many applications of digital computers require handling of data that consists of not only of numbers but also letters and special characters. The standard alphanumeric binary code is ASCII (American Standard Code for Information Interchange) which uses 7 bits to code 128 characters.

TABLE 3-4 American Standard Code for Information Interchange (ASCII)



Character	Binary code	Character	Binary code
A	100 0001	0	011 0000
B	100 0010	1	011 0001
C	100 0011	2	011 0010
D	100 0100	3	011 0011
E	100 0101	4	011 0100
F	100 0110	5	011 0101
G	100 0111	6	011 0110
H	100 1000	7	011 0111
I	100 1001	8	011 1000
J	100 1010	9	011 1001
K	100 1011		
L	100 1100		
M	100 1101	space	010 0000
N	100 1110	.	010 1110
O	100 1111	(	010 1000
P	101 0000	+	010 1011
Q	101 0001	$	010 0100
R	101 0010	*	010 1010
S	101 0011	)	010 1001
T	101 0100	−	010 1101
U	101 0101	/	010 1111
V	101 0110	,	010 1100
W	101 0111	=	011 1101
X	101 1000		
Y	101 1001		
Z	101 1010		

Complements

Complements are used in digital computers for simplifying the subtraction operation and for logical manipulation. There are two types of complements for each base r system: the r's complement and the (r - l)'s complement.

9' s complement ,10's complement

l's complement ,2's complement

When the value of the base r is substituted in the name, the two types are referred to as the 2's and 1's complement for binary numbers and the 10's and 9's complement for decimal numbers.

(r - l)'s Complement

Given a number N in base r having n digits, the (r - 1)'s complement of N is defined as (r' - 1) - N.

For decimal numbers r = 10 and r - 1 = 9, so the 9's complement of N is

(10' - 1) - N.

Now, 10' represents a number that consists of a single 1 followed by n 0's. 10' - 1 is a number represented by n 9's.

For example, with n = 4 we have 10^4 = 10000 and 10^4 - 1 = 9999. It follows that the 9' s complement of a decimal number is obtained by subtracting each digit from 9.

For example, the 9's complement of 546700 is

999999 - 546700 = 453299

and the 9's complement of I2389 is 99999 - 12389 = 876I0.

For binary numbers, r = 2 and r - 1 = 1, so the 1's complement of N is

(2' - 1) - N.

Again, 2' is represented by a binary number that consists of a 1 followed by n 0's. 2' - 1 is a binary number represented by n 1's.

For example, with n = 4, we have 2^4 = (10000), and 2^4 - 1 = $(1111)_2$. Thus the 1's complement of a binary number is obtained by subtracting each digit from 1. However, the subtraction of a binary digit from 1 causes the bit to change from 0 to 1 or from 1 to 0. Therefore, the 1's complement of a binary number is formed by changing 1's into 0's and 0's into 1's. For example, the 1's complement of I011001 is 0100110 and the 1' s complement of 0001111 is 1110000. The (r - 1)'s complement of octal or hexadecimal numbers are obtained by subtracting each digit from 7 or F (decimal 15) respectively.

(r's) Complement

The r's complement of an n-digit number N in base r is defined as r' - N for N ≠ 0 and 0 for N = D. Comparing with the (r - 1)'s complement, we note that the r's complement is obtained by adding 1 to the (r - 1)'s complement since

r' - N = [(r' - 1) - N] + 1.

Thus the 10's complement of the decimal 2389 is 7610 + 1 = 7611 and is obtained by adding 1 to the 9' s complement value.

The 2's complement of binary 101100 is

010011 + 1 = 010100

and is obtained by adding 1 to the 1's complement value. Since 10' is a number represented by a 1 followed by n 0's, then 10' - N, which is the 10's complement of N, can be formed also be leaving all least significant 0's unchanged, subtracting the first nonzero least significant digit from 10, and then subtracting all higher significant digits from 9.

The 10's complement of 246700 is 753300 and is obtained by leaving the two zeros unchanged, subtracting 7 from 10, and subtracting the other three digits from 9. Similarly, the 2's complement can be formed by leaving all least significant 0's and the first 1 unchanged, and then replacing 1's by 0's and 0's by 1's in all other higher significant bits. The 2's complement of 1101100 is 0010100 and is obtained by leaving the two low-order 0's and the first 1 unchanged, and then replacing 1's by 0's and 0's by 1's in the other four most significant bits.

Subtraction of Unsigned Numbers

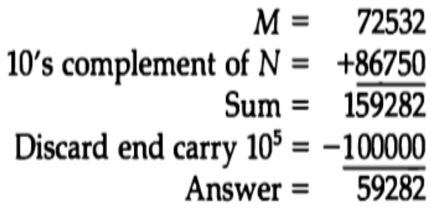
The direct method of subtraction taught in elementary schools uses the borrow concept. In this method we borrow a 1 from a higher significant position when the minuend digit is smaller than the corresponding subtrahend digit. This seems to be easiest when people perform subtraction with paper and pencil. When subtraction is implemented with digital hardware, this method is found to be less efficient than the method that uses complements. The subtraction of two n-digit unsigned numbers M - N (N ≠ 0) in base r can be done as follows:

2. Add the minuend M to the r's complement of the subtrahend N.
   This performs M + (r' - N) = M-N + r'.

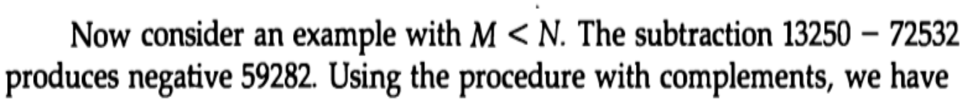
2. If M > N, the sum will produce an end carry r' which is discarded, and what is left is the result M - N.

3. If M < N, the sum does not produce an end carry and is equal to r' - (N - M), which is the r's complement of (N - M).

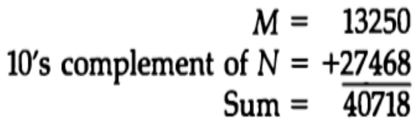
To obtain the answer in a familiar form, take the r' s complement of the sum and place a negative sign in front. Consider, for example, the subtraction 72532 - 13250 = 59282. The l0's complement of 13250 is 86750. Therefore:



$$\begin{array}{rr} M = & 72532 \\ \text{10's complement of } N = & +\underline{86750} \\ \text{Sum} = & 159282 \\ \text{Discard end carry } 10^5 = & -\underline{100000} \\ \text{Answer} = & 59282 \end{array}$$



Now consider an example with $M < N$. The subtraction 13250 − 72532 produces negative 59282. Using the procedure with complements, we have

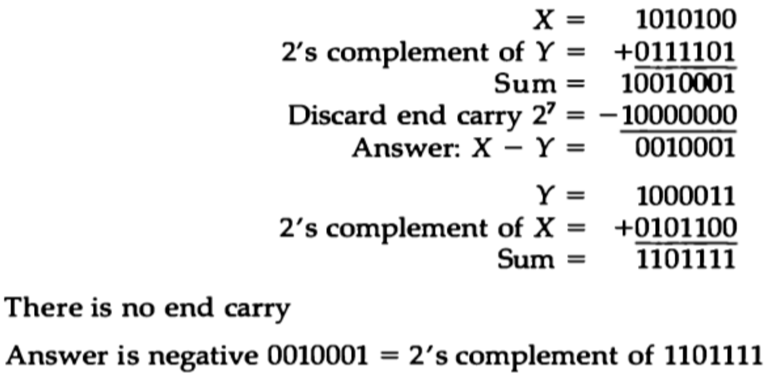


$$\begin{array}{rr} M = & 13250 \\ \text{10's complement of } N = & +\underline{27468} \\ \text{Sum} = & 40718 \end{array}$$

There is no end carry

Answer is negative 59282 = 10's complement of 40718

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



$$\begin{array}{rr}
X = & 1010100 \\
\text{2's complement of } Y = & +\underline{0111101} \\
\text{Sum} = & 10010001 \\
\text{Discard end carry } 2^7 = & -\underline{10000000} \\
\text{Answer: } X - Y = & 0010001 \\
Y = & 1000011 \\
\text{2's complement of } X = & +\underline{0101100} \\
\text{Sum} = & 1101111
\end{array}$$

There is no end carry

Answer is negative 0010001 = 2's complement of 1101111

Fixed-Point Representation

Positive integers, including zero, can be represented as unsigned numbers. However, to represent negative integers, we need a notation for negative values. In ordinary arithmetic, a negative number is indicated by a minus sign and a positive number by a plus sign. Because of hardware limitations, computers must represent everything with 1's and 0's, including the sign of a number. As a consequence, it is customary to represent the sign with a bit placed in the leftmost position of the number. The convention is to make the sign bit equal to 0 for positive and to 1 for negative. In addition to the sign, a number may have a binary (or decimal) point. The position of the binary point is needed to represent fractions, integers, or mixed integer-fraction numbers. The representation of the binary point in a register is complicated by the fact that it is characterized by a position in the register. There are two ways of specifying the position of the binary point in a register: by giving it a fixed position or by employing a floating-point representation.

The fixed-point method assumes that the binary point is always fixed in one position. The two positions most widely used are

(1) a binary point in the extreme left of the register to make the stored number a fraction, and

(2) a binary point in the extreme right of the register to make the stored number an integer.

In either case, the binary point is not actually present, but its presence is assumed from the fact that the number stored in the register is treated as a fraction or as an integer. The floating-point representation uses a second register to store a number that designates the position of the decimal point in the first register. Floating-point representation is discussed further in the next section.

Integer Representation

When an integer binary number is positive the sign is represented with 0, when the number is negative the sign is represented by 1 but the rest of the number is represented in one of three possible ways:

4. Signed Magnitude representation.
5. Signed 1's Complement representation.
6. Signed 2's Complement representation.

Signed Magnitude representation of negative number consists of the magnitude and negative number. In the other two representations, the negative number is represented in either 1's complement or 2's complement of its positive value.
For example,

+ 14 is represented in the 8 bit register as 00001110

There are three different ways to represent **-14**

In Signed Magnitude representation: 1 0001110
In Signed 1's Complement representation : 1 1110001
In Signed 2's Complement representation : 1 1110010

The Signed Magnitude representation of **-14 is** obtained from **+14** by complementing signed bit.

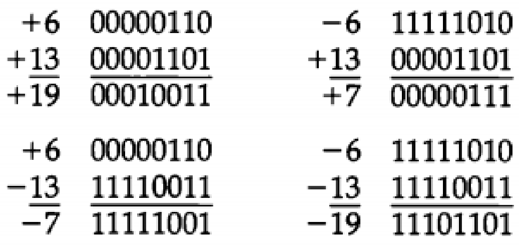
Signed magnitude method is employed in ordinary arithmetic but it is not suitable for computer arithmetic. Signed 1's complement method has some difficulties, it has two representation for 0 (+0 and -0). The signed binary arithmetic entirely deals with 2's complement.

Arithmetic Addition

Addition of two numbers in signed magnitude system follows the rules of ordinary arithmetic. For example

(+32) + (- 45) = - (45 -32) = - 13

This process requires comparison of signs and magnitudes and then performing addition or subtraction. In Signed 2's complement method, no comparison and no subtraction requires only addition and complementation.



```
 +6   00000110        −6   11111010
+13   00001101       +13   00001101
+19   00010011        +7   00000111

 +6   00000110        −6   11111010
−13   11110011       −13   11110011
 −7   11111001       −19   11101101
```

In each of the above four cases, the operation performed is only addition. Any carry out of the sign bit position is discarded and negative results are automatically in 2's complement form.

Arithmetic Subtraction

Subtraction of two signed binary numbers is as follows: Take the 2's complement of subtrahend and add it to the minuend, a carry out of the sign bit position is discarded. Therefore computers need only one common circuit to handle both types of operations.

Consider the subtraction (- 6) – (- 13) = +7

11111010 – 11110011 = 11111010 + 00001101 = 100000111

Discard carry, which is equivalent to +7

Overflow

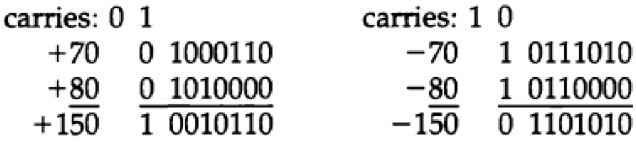
When two digits of length n digits, each are added and the sum occupies (n+1) digits, we say that an overflow occurred. An overflow is a problem in digital computers, because the length of the register is fixed. The result of (n+1) bits can't be accommodated in a register with a standard length of n bits.

Many computers detect the occurrence of the overflow and when it occurs, a corresponding Flip-flop is set, which can then be used for other processing.

The detection of overflow after the addition of two binary numbers depends on whether the numbers are considered as signed or unsigned.

- When the two numbers added are unsigned, an overflow is detected from the end carry out of the most significant digit.
- When the numbers are signed, the left most bit is represented for sign and negative numbers are in 2'complemented form. When they are added, the sign bit is considered as part of the number and the end carry does not indicate an overflow.

An overflow cannot occur if one number is positive and one number is negative. Since adding a positive number to negative number produces a result that is smaller than the larger of the two umbers. An overflow occur when two numbers both are positive or negative.



```
carries: 0  1              carries: 1  0
    +70   0 1000110            −70   1 0111010
    +80   0 1010000            −80   1 0110000
   +150   1 0010110           −150   0 1101010
```

If two carries are applied to EX-OR gate, an over flow can be detected, when it gives 1 as output.

Floating Point Representation

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

Mantissa	**Exponent**
+0.6132789	+04

The value of the exponent indicates that the actual position of the decimal point if four positions to the right of the indicated decimal point in the fraction. This representation is equivalent to the scientific notation**:** **$0.6132789 \times 10^{+4}$**

i.e in the format $m \times r^e$

Only the mantissa and exponent are physically represented in the register. The radix and radix point position of mantissa are always assumed. Floating point binary number is represented in a similar manner except that it uses base 2 for the exponent. For example a binary number +1001.11 is represented with 8 bit mantissa and 6 bit exponent is as follows:

Mantissa	**Exponent**
01001110	000100

Left most 0 represent positive number, this is equivalent to:

$$+(.1001110)_2 \text{ X } 2^{+4}$$

Normalization

A Floating point number is said to be normalized if the most significant digit of the mantissa is non zero. For example the number 350 is normalized but 00035 is not. Regardless of where the radix point is assumed to be in the mantissa, the number is normalized only when its left most digit is non zero.

For example the number 00010110 is not normalized, this number can be normalized by shifting the three position left and discarding the leading zeroes to obtain 10110000. The three shift multiply the number with 2^3. Normalized numbers provide the maximum possible precision for the floating point numbers.

Other Binary Codes

Some of the commonly used Binary Codes. The following is the list:

- 8421 Codes
- 2421 Codes
- 5211 Codes
- Excess-3 Codes
- Gray Codes

In the above list, the first three i.e. 8421, 2421 and 5211 are Weighted codes while the other two are non-weighted binary codes.

8421 Code or BCD Code

The decimal numbers 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 can be expressed in Binary numbers as shown below. All these binary numbers again expressed in the last column by expanding into 4 bits. As per the weighted binary digits, the 4 Bit binary numbers can be expressed according to their place value from left to right as 8421 (2^3 2^2 2^1 2^0 = 8421).

DECIMAL NUMBER	BINARY NUMBER	4 BIT EXPRESSION(8421)
0	0	0000
1	1	0001
2	10	0010
3	11	0011
4	100	0100
5	101	0101
6	110	0110
7	111	0111
8	1000	1000
9	1001	1001

As per the above expression all the decimal numbers written in the 4 Bit binary code in the form of 8421 and this is called as 8421 Code and also as Binary coded decimal BCD.

As this is a straight code, any Decimal number can be expressed easily because the weights of the positions are straight for easy conversion into this 8421 code.

There are other forms of codes which are not so popular but rather confusing. They are 2421 code, 5211 code, reflective code, sequential code, non- weighted coded, excess-3 code and Grey code. They are having their own importance for some of the exclusive applications and may be useful for some of the typical applications.

2421 Code

This code also a 4 bit application code where the binary weights carry 2, 4, 2, 1 from left to right.

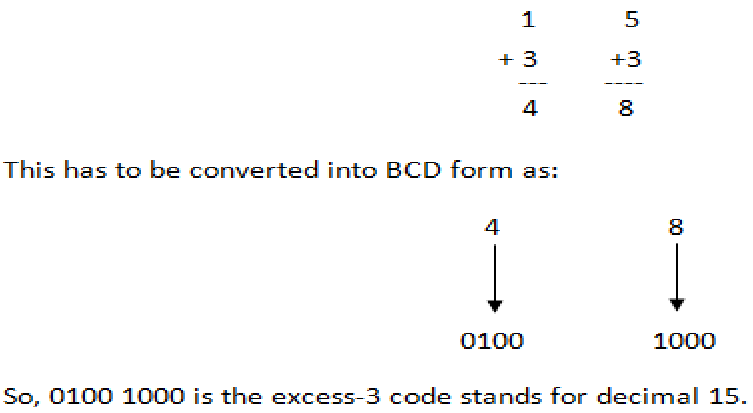
DECIMAL NUMBER	BINARY NUMBER	2421 CODE
0	0	0000

DECIMAL NUMBER	BINARY NUMBER	2421 CODE
1	1	0001
2	10	0010
3	11	0011
4	100	0100
5	101	1011
6	110	1100
7	111	1101
8	1000	1110
9	1001	1111

Excess-3 Code

As mentioned above, some of the codes will not follow the binary weights, Excee-3 code is an example of it and it is an important 4 bit code. The excess – 3 code of a decimal number is achieved by adding the number 3 to the 8421 code.

For example to convert 15 to an excess-3 code, first 3 to be added to each digit as shown below.



```
  1      5
+ 3     +3
 ---    ----
  4      8
```

This has to be converted into BCD form as:

```
   4          8
   |          |
   v          v
 0100       1000
```

So, 0100 1000 is the excess-3 code stands for decimal 15.

BACK TO TOP

Excess -3 Code Examples

3. Find the excess-3 code of $(237.75)_{10}$
4. Find the decimal number of excess-3 number 110010100011.01110101.

Sol:

1) The excess-3 code for $(237)_{10}$ is obtained by adding 3 to all the digits individually, that is 2, 3 an

d 7 will become 5, 6 and 10 respectively. These 5, 6 and 10 decimals have to be converted into binary form and the result is 010101101010.

The excess-3 code for $(.75)_{10}$ is obtained by replacing 7 and 5 with 10 and 8 respectively by adding 3 to each digit. That is, the excess-3 code for $(.75)_{10}$ is.10101000.

Combining the results of the integral and fractional parts, the excess-3 code for $(237.75)_{10}$ is 010101101010.10101000.

2) The excess-3 code is 110010100011.01110101

By separating 4 bits as group the equivalent excess-3 code is given as 1100 1010 0011.0111 0101.

Subtracting 0011 from each four-bit group, we obtain the new number as: 1001 0111 0000.0100 0010.

Therefore, the decimal equivalent is $(970.42)_{10}$.

Gray Code

The gray code is the code where one bit will be differed to the preceding number. For example, decimal numbers 13 and 14 are represented by gray code numbers 1011 and 1001, these numbers differ only in single position that is the second position from the right. In the same way first position on the left changes for 7 and 8 which are 0100 and 1100 and this is also called Unit-distance code. The gray code has very special place in digital electronics.

DECIMAL NUMBER	BINARY CODE	GRAY CODE
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110

DECIMAL NUMBER	BINARY CODE	GRAY CODE
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

EBCDIC Code

EBCDIC stands for Extended Binary Coded Decimal Interchange Code. IBM invented this code to extend the Binary Coded Decimal which existed at that time. All the IBM computers and peripherals use this code. It is an 8-bit code and therefore can accommodate 256 characters. Below are given some characters of **EBCDIC code** to get familiar with it.

Error Detection Codes

The binary information is transferred from one location to another location through some communication medium. The external noise can change bits from 1 to 0 or 0 to 1.This changes in values are called errors. For efficient data transfer, there should be an error detection and correction codes. An error detection code is a binary code that detects digital errors during transmission. A famous error detection code is a Parity Bit method.

Parity Bit Method :

A parity bit is an extra bit included in binary message to make total number of 1's either odd or even. Parity word denotes number of 1's in a binary string. There are two parity system- even and odd. In even parity system 1 is appended to binary string it there is an odd number of 1's in string otherwise 0 is appended to make total even number of 1's.

In odd parity system, 1 is appended to binary string if there is even a number of 1's to make an odd number of 1's. The receiver knows that whether sender is an odd parity generator or even parity generator. Suppose if sender is an odd parity generator then there must be an odd number of 1's in received binary string. If an error occurs to a single bit that is either bit is changed to 1 to 0 or O to 1, received binary bit will have an even number of 1's which will indicate an error.

The limitation of this method is that only error in a single bit would be identified.

Message (XYZ)	P(Odd)	P(Even)
000	1	0
001	0	1
010	0	1
011	1	0
100	0	1
101	1	0
110	1	0
111	0	1

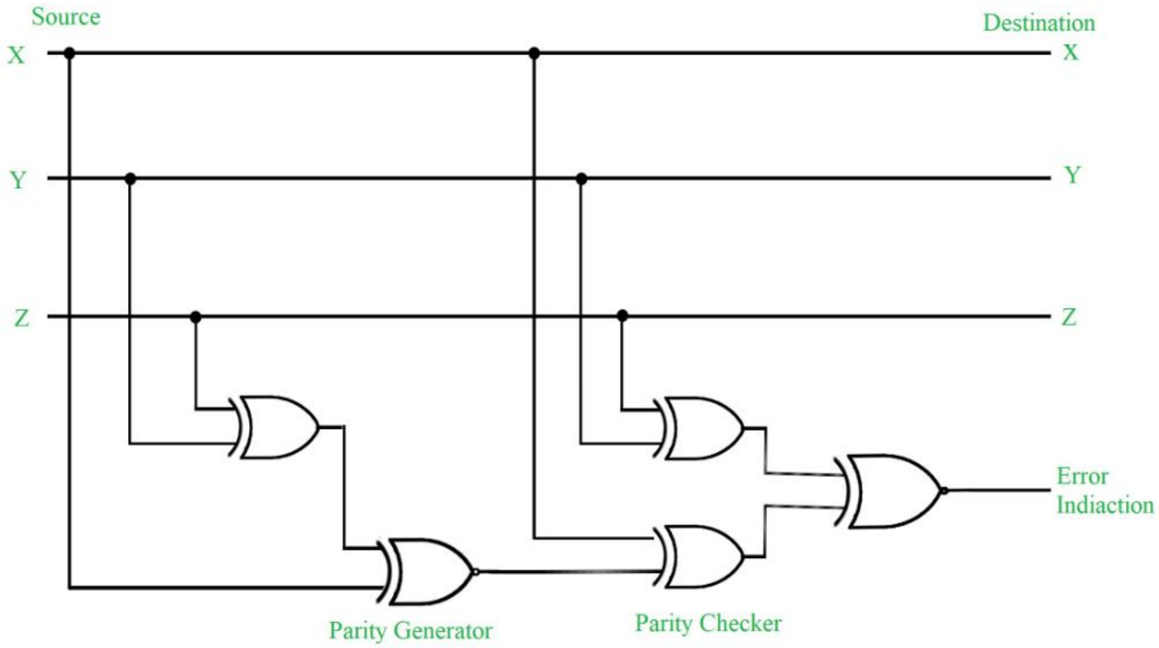


Fig: Parity Checker

Basic Computer Organization

Instruction code

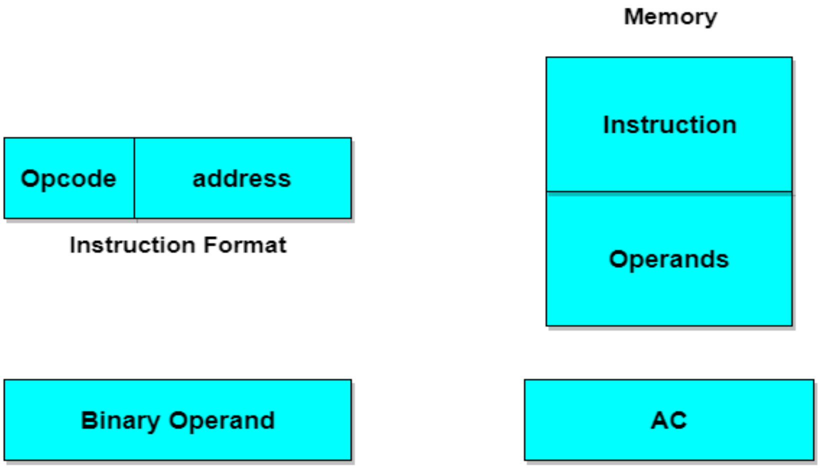
A **Program**, as we all know, is a set of instructions that specify the operations, operands, and the sequence by which processing has to occur. An **instruction code** is a group of bits that tells the computer to perform a specific operation part.

The operation code of an instruction is a group of bits that define operations such as add, subtract, multiply, shift and compliment. The number of bits required for the operation code depends upon the total number of operations available on the computer. The operation code must consist of at least **n bits** for a given **2^n** operations. The operation part of an instruction code specifies the operation to be performed.

The operation must be performed on the data stored in registers. An instruction code therefore specifies not only operations to be performed but also the registers where the operands(data) will be found as well as the registers where the result has to be stored.

Stored Program Organization

The simplest way to organize a computer is to have **Processor Register** and instruction code with two parts. The first part specifies the operation to be performed and second specifies an address. The memory address tells where the operand in memory will be found. Instructions are stored in one section of memory and data in another.



Computers with a single processor register is known as **Accumulator (AC)**. The operation is performed with the memory operand and the content of AC.

Addressing Modes

The operation field of an instruction specifies the operation to be performed. This operation will be executed on some data which is stored in computer registers or the main memory. The way any operand is selected during the program execution is dependent on the addressing mode of the instruction.

Types of Addressing Modes

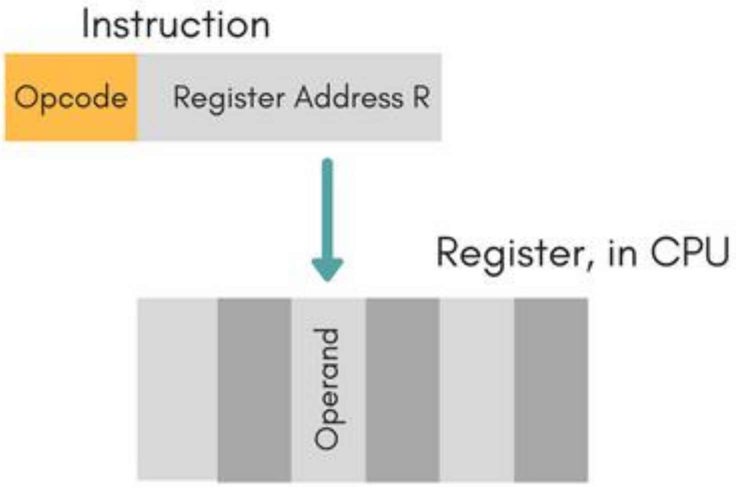
Below we have discussed different types of addressing modes one by one:

10.Immediate Mode

In this mode, the operand is specified in the instruction itself. An immediate mode instruction has an operand field rather than the address field.

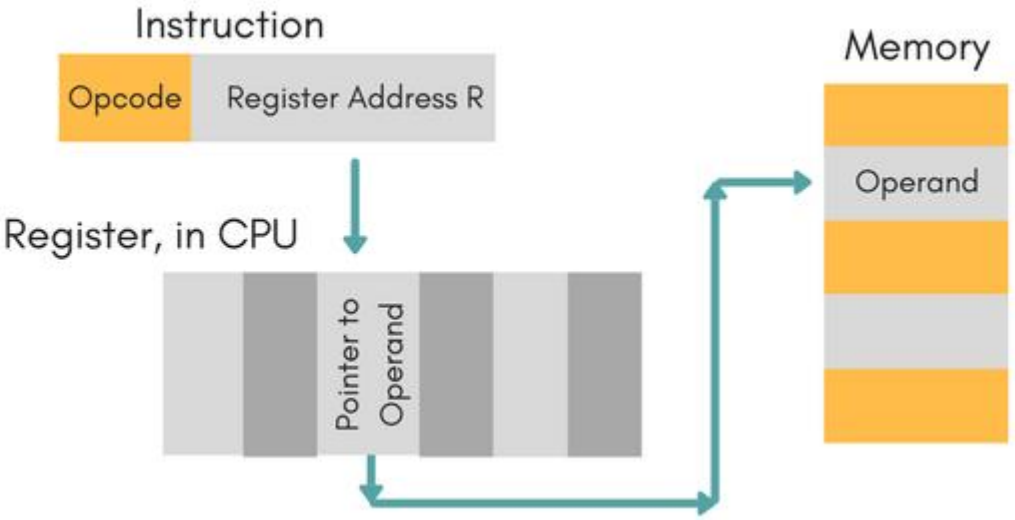
11.Register Mode

In this mode the operand is stored in the register and this register is present in CPU. The instruction has the address of the Register where the operand is stored.



12.Register Indirect Mode

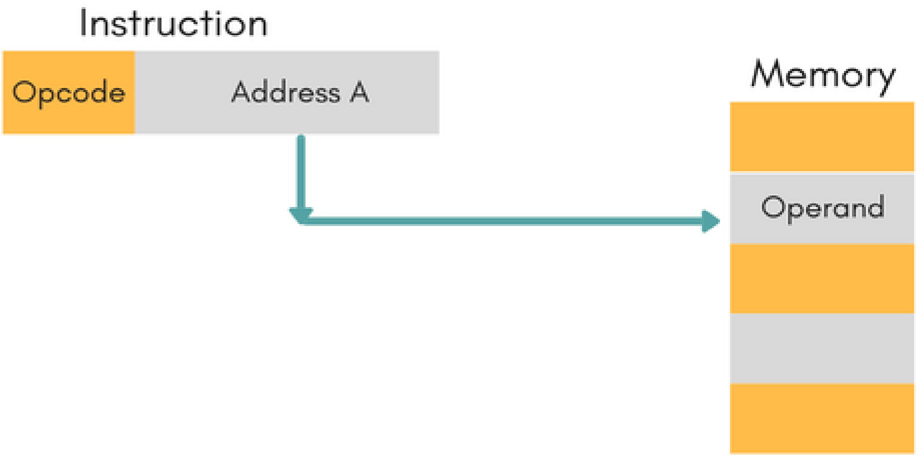
In this mode, the instruction specifies the register whose contents give us the address of operand which is in memory. Thus, the register contains the address of operand rather than the operand itself.



13.Direct Addressing Mode

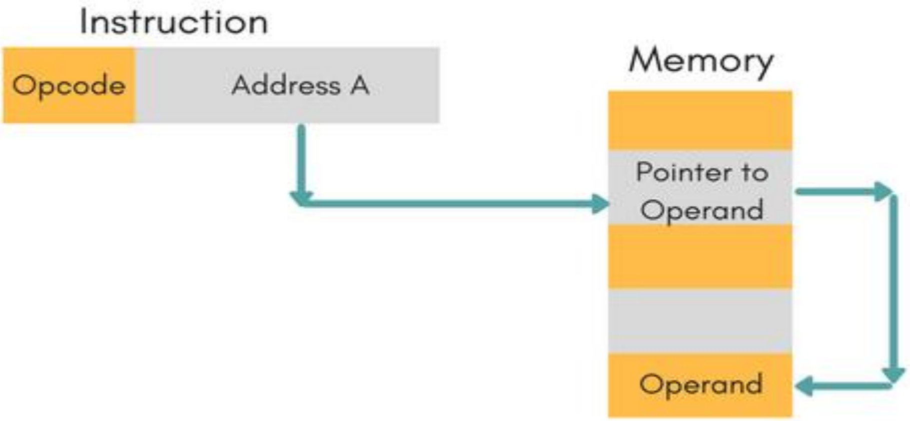
In this mode, effective address of operand is present in instruction itself.

- Single memory reference to access data.
- No additional calculations to find the effective address of the operand.



14.Indirect Addressing Mode

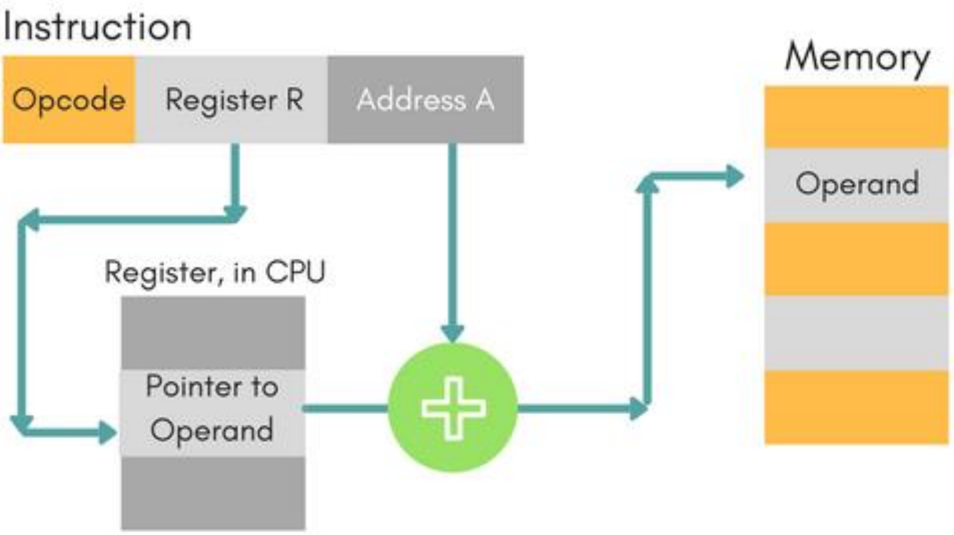
In this, the address field of instruction gives the address where the effective address is stored in memory. This slows down the execution, as this includes multiple memory lookups to find the operand.



15.Displacement Addressing Mode

In this the contents of the indexed register is added to the Address part of the instruction, to obtain the effective address of operand.

EA = A + (R), In this the address field holds two values, A(which is the base value) and R(that holds the displacement), or vice versa.



16.Relative Addressing Mode

It is a version of Displacement addressing mode.

In this the contents of PC(Program Counter) is added to address part of instruction to obtain the effective address.

EA = A + (PC), where EA is effective address and PC is program counter.

The operand is A cells away from the current cell(the one pointed to by PC)

17.Base Register Addressing Mode

It is again a version of Displacement addressing mode. This can be defined as EA = A + (R), where A is displacement and R holds pointer to base address.

18.Stack Addressing Mode

In this mode, operand is at the top of the stack. For example: ADD, this instruction will *POP* top two items from the stack, add them, and will then *PUSH* the result to the top of the stack.

Computer Registers

Registers are a type of computer memory used to quickly accept, store, and transfer data and instructions that are being used immediately by the CPU. The registers used by the CPU are often termed as Processor registers.

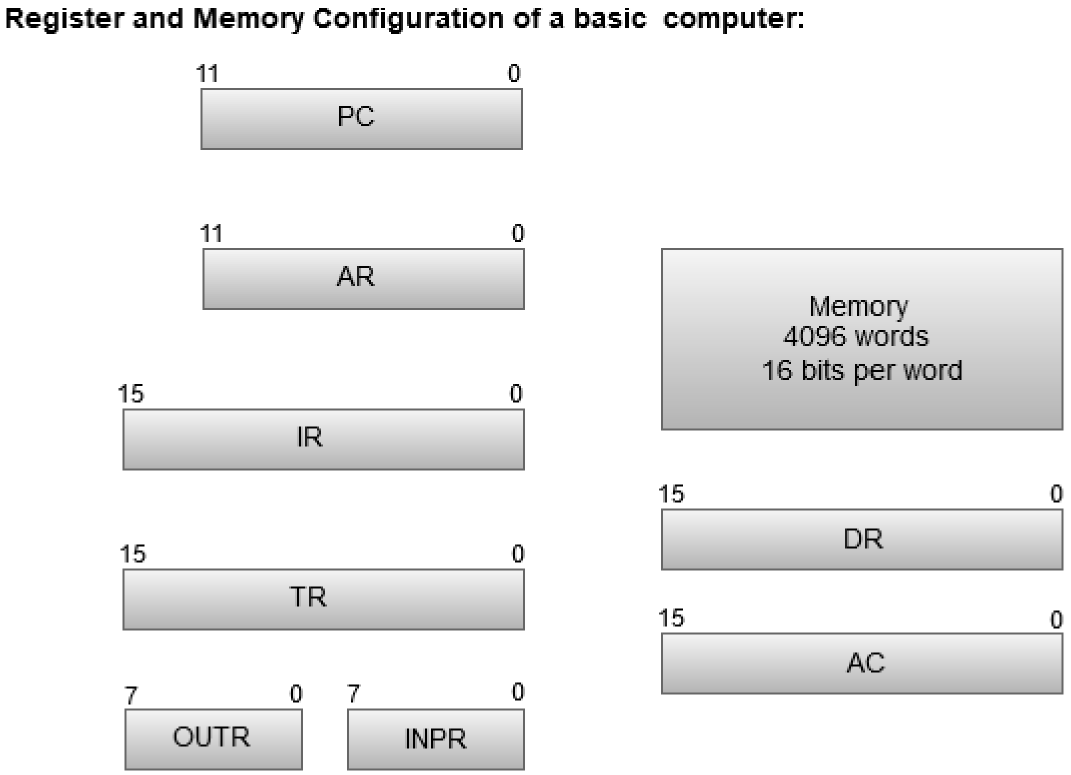
A processor register may hold an instruction, a storage address, or any data (such as bit sequence or individual characters).

The computer needs processor registers for manipulating data and a register for holding a memory address. The register holding the memory location is used to calculate the address of the next instruction after the execution of the current instruction is completed.

Following is the list of some of the most common registers used in a basic computer:

Register	Symbol	Number of bits	Function
Data register	DR	16	Holds memory operand
Address register	AR	12	Holds address for the memory
Accumulator	AC	16	Processor register
Instruction register	IR	16	Holds instruction code
Program counter	PC	12	Holds address of the instruction
Temporary register	TR	16	Holds temporary data
Input register	INPR	8	Carries input character
Output register	OUTR	8	Carries output character

The following image shows the register and memory configuration for a basic computer.



- The Memory unit has a capacity of 4096 words, and each word contains 16 bits.
- The Data Register (DR) contains 16 bits which hold the operand read from the memory location.
- The Memory Address Register (MAR) contains 12 bits which hold the address for the memory location.
- The Program Counter (PC) also contains 12 bits which hold the address of the next instruction to be read from memory after the current instruction is executed.
- The Accumulator (AC) register is a general purpose processing register.
- The instruction read from memory is placed in the Instruction register (IR).
- The Temporary Register (TR) is used for holding the temporary data during the processing.
- The Input Registers (IR) holds the input characters given by the user.
- The Output Registers (OR) holds the output after processing the input data.

Common Bus System

The basic computer has 8 registers, a memory unit and a control unit. Paths must be provided to transfer data from one register to another. An efficient method for transferring data in a system is to use a **Common Bus System**. The output of registers and memory are connected to the common bus.

Paths must be provided to transfer information from one register to another and between memory and registers. The number of wires will be excessive if connections are made between the outputs of each register and the inputs of the other registers. A more efficient scheme for transferring information in a system with many registers is to use a common bus. We can construct a bus system using multiplexers or three-state buffer gates. We just have to connect the registers and memory of the basic computer to a common bus system.

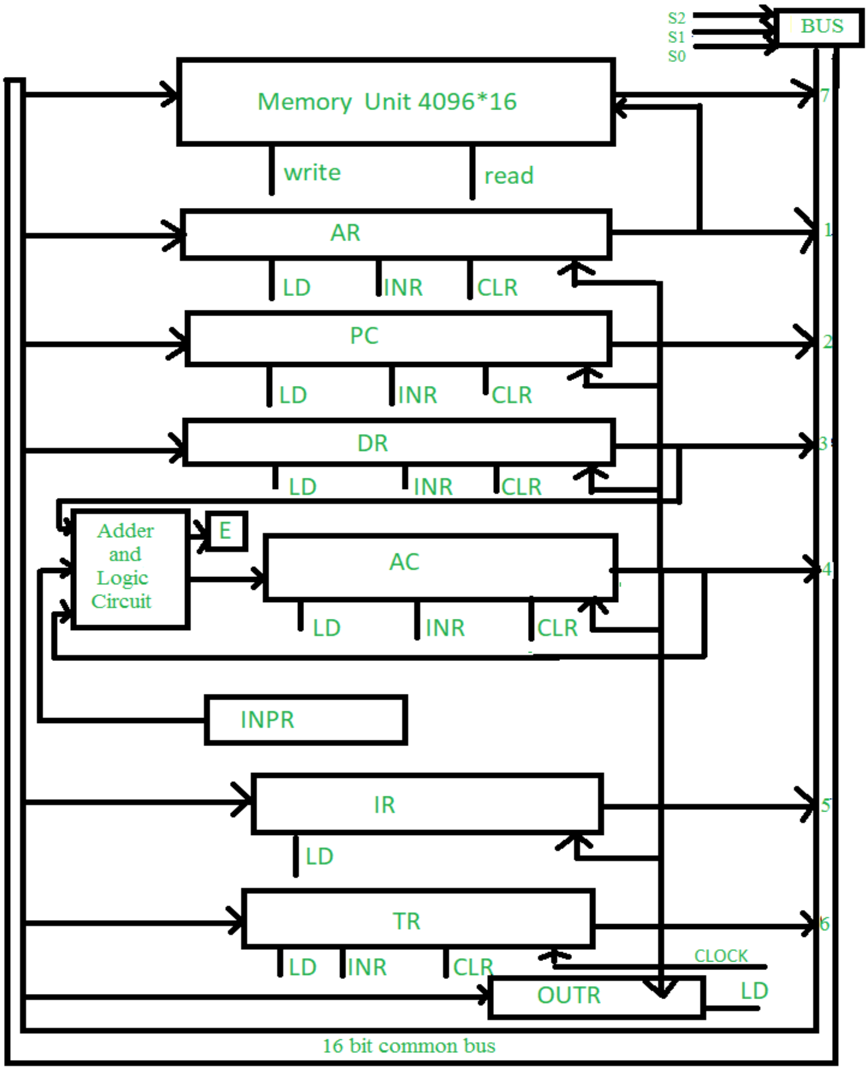
The outputs of seven registers and memory are connected to the common bus. The specific output that is selected for the bus lines at any given time is determined from the binary value of the selection variables S2, S1, and S0. The number along each output shows the decimal equivalent of the required binary selection. For example, the number along the output of DR is 3. The 16-bit outputs of DR are placed on the bus lines when S2S1S0 = 011 since this is the binary value of decimal 3.

The lines from the common bus are connected to the inputs of each register and the data inputs of the memory. The particular register whose LD (load) input is enabled receives the data from the bus during the next clock pulse transition. The memory receives the contents of the bus when its write input is activated. The memory places its 16-bit output onto the bus when the read input is activated and S2S1S0 = 111.

Four registers, DR, AC, IR, and TR, have 16 bits each. Two registers, AR and PC, have 12 bits each since they hold a memory address. When the contents of AR or PC are applied to the 16-bit common bus, the four most significant bits are set to O's. When AR or PC receive information from the bus, only the 12 least significant bits are transferred into the register.

The input register INPR and the output register **OUTR** have 8 bits each and communicate with the eight least significant bits in the bus. INPR is connected to provide information to the bus but OUTR can only receive information from the bus. This is because INPR receives a character from an input device which is then transferred to AC.

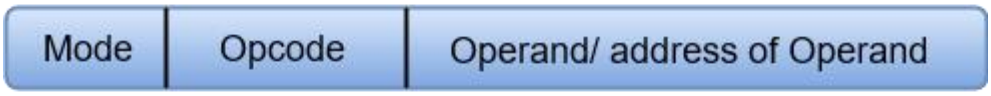
The 16 inputs of AC come from an adder and logic circuit. This circuit has three sets of inputs. One set of 16-bit inputs come from the outputs of AC . They are used to implement register rnlcrooperations such as complement AC and shift AC . Another set of 16-bit inputs come from the data register DR. The inputs from DR and AC are used for arithmetic and logic microoperations, such as add DR to AC or AND DR to AC . The result of an addition is transferred to AC and the end carry-out of the addition is transferred to flip-flop E (Extended AC bit). A third set of 8-bit inputs come from the input register INPR.



Computer Instructions

Computer instructions are a set of machine language instructions that a particular processor understands and executes. A computer performs tasks on the basis of the instruction provided. An instruction comprises of groups called fields. These fields include:

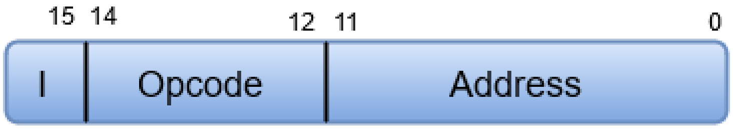
- The Operation code (Opcode) field which specifies the operation to be performed.
- The Address field which contains the location of the operand, i.e., register or memory location.
- The Mode field which specifies how the operand will be located.



A basic computer has three instruction code formats which are:

4. Memory - reference instruction
5. Register - reference instruction
6. Input-Output instruction

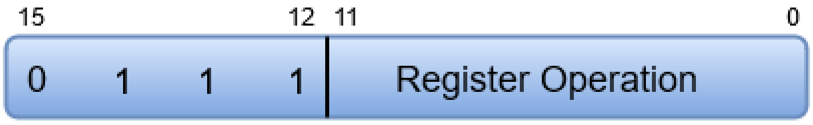
Memory - reference instruction



(Opcode = 000 through 110)

In Memory-reference instruction, 12 bits of memory is used to specify an address and one bit to specify the addressing mode 'I'.

Register - reference instruction

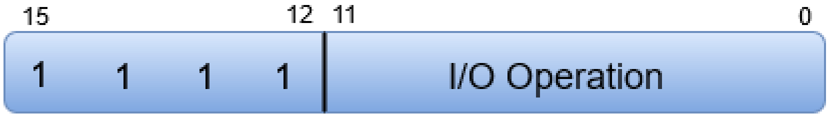


(Opcode = 111, I = 0)

The Register-reference instructions are represented by the Opcode 111 with a 0 in the leftmost bit (bit 15) of the instruction.

A Register-reference instruction specifies an operation on or a test of the AC (Accumulator) register.

Input-Output instruction



(Opcode = 111, I = 1)

Just like the Register-reference instruction, an Input-Output instruction does not need a reference to memory and is recognized by the operation code 111 with a 1 in the leftmost bit of the instruction. The remaining 12 bits are used to specify the type of the input-output operation or test performed.

Instruction Set Completeness

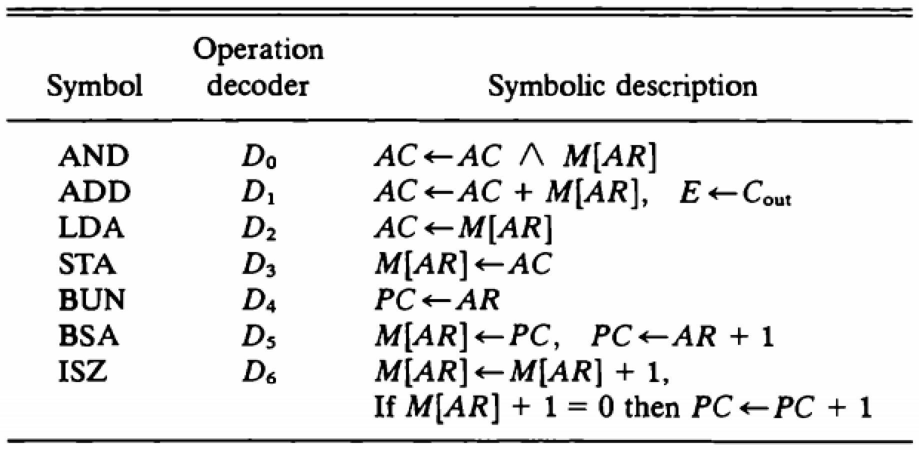
A set of instructions is said to be complete if the computer includes a sufficient number of instructions in each of the following categories:

- Arithmetic, logical and shift instructions
- A set of instructions for moving information to and from memory and processor registers.
- Instructions which controls the program together with instructions that check status conditions.
- Input and Output instructions

In order to specify the micro operations needed for the execution of each operation, it is necessary that the function intended to perform be defined precisely.

Memory- Reference Instructions

In order to specify the micro operations needed for the execution of each operation, it is necessary that the function intended to perform be defined precisely. The decoded output D_i for i=0,1,2,3,4,5,6 from the operation decoder that belongs to each instruction is included in the following table.



Symbol	Operation decoder	Symbolic description
AND	D_0	$AC \leftarrow AC \wedge M[AR]$
ADD	D_1	$AC \leftarrow AC + M[AR], \quad E \leftarrow C_{out}$
LDA	D_2	$AC \leftarrow M[AR]$
STA	D_3	$M[AR] \leftarrow AC$
BUN	D_4	$PC \leftarrow AR$
BSA	D_5	$M[AR] \leftarrow PC, \quad PC \leftarrow AR + 1$
ISZ	D_6	$M[AR] \leftarrow M[AR] + 1$, If $M[AR] + 1 = 0$ then $PC \leftarrow PC + 1$

AND to AC

This is an instaruction that perform the AND logic operation on pairs of bits in AC and the memory word specified by the effective address. The result of the operation is transferred to AC . The micro operations that execute this instruction are:

D0T4:DR<-M[AR]
D0T5:AC<-AC/\DR, SC<---0

The control function for this instruction uses the operation decoder D0 since this output of the decoder is active when the instruction has an AND operation whose binary code value 000. Two timing signals are needed to execute the instruction. The clock transition associated with timing signal T4 transfers the operand from memory into DR . The clock transition associated with the next timing signal T5 transfers to AC the result of the AND logic operation between the contents of DR and AC. The same clock transition clears SC to 0, transferring control to timing signal T0 to start a new instruction cycle.

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_{out} is transferred to the E (extended accumulator) flip-flop. The rnicrooperations needed to execute this instruction are

D_1T_4:DR←M[AR]
D_1T_5: AC← AC + DR, E← C_{out} , SC ← 0

Same Two timing signals, T, and T5, are used again but with operation decoder D1 instead of D0, which was used for the AND instruction. After the instruction is fetched from memory and decoded, only one output of the operation decoder will be active, and that output determines the sequence of microoperations that the control follows during the execution of a memory-reference instruction.

LDA: Load to AC

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

D_2T_4: DR← M[AR]
D_2T_5: AC← DR← 0

Looking back at the bus system shown in Figure 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.

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:

D3T4: M [AR] ← AC, SC ← 0

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:

D4T4: PC ← AR, SC ← 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 .

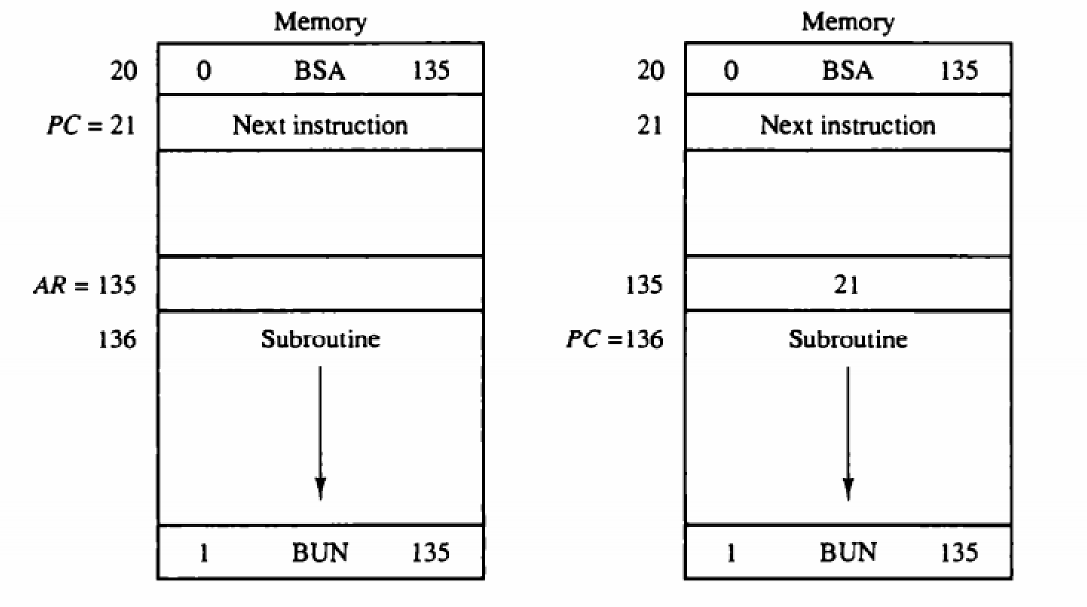
BSA: Branch and Save Return Address

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] <-- PC, PC <-- AR + I

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] <-- 21, PC <-- 135 + 1 = 136



(a) Memory, PC, and AR at time T_4 (b) Memory and PC after execution

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 instruction at the return address.

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:

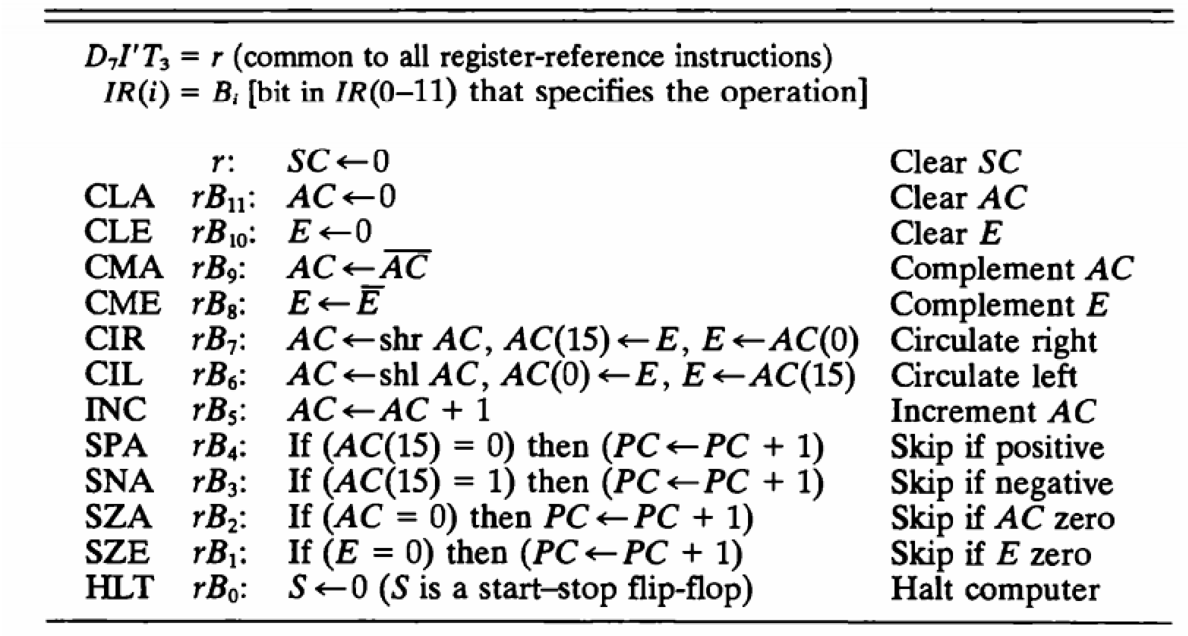
D6T4: DR<--M[AR]
D6T5: DR<--DR+1
D,T,: M[AR]<--DR, if (DR = 0) then (PC ← PC + 1), SC ← 0

Register – Reference Instructions

Register-reference instructions are recognized by the control when 07 = 1 and I = 0. These instructions use bits 0 through 11 of the instruction code to specify one of 12 instructions. These 12 bits are available in IR(0-11). They were also transferred to AR during time T2•

The control functions and microoperations for the register-reference instructions are. listed in the following Table. These instructions are executed with the clock transition associated with timing variable T3. Each control function needs the Boolean relation D7I'T3, which we designate for convenience by the symbol r. The control function is distinguished by one of the bits in IR(0-11). By assigning the symbol B, to bit i of IR, all control functions can be simply denoted by rB;.

For example, the instruction CLA has the hexadecimal code 7800 (see Table 5-2), which gives the binary equivalent 011I 1000 0000 0000. The first bit is a zero and is equivalent to I'. The next three bits constitute the operation code and are recognized from decoder output D7. Bit 11 in IR is I and is recognized from 811. The control function that initiates the rnicrooperation for this instruction is D7I'T3B11 = rB11. The execution of a register-reference instruction is completed at time T3. The sequence counter SC is cleared to 0 and the control goes back to fetch the next instruction with timing signal T0.



$D_7I'T_3 = r$ (common to all register-reference instructions)
$IR(i) = B_i$ [bit in $IR(0–11)$ that specifies the operation]

	r:	$SC \leftarrow 0$	Clear SC
CLA	rB_{11}:	$AC \leftarrow 0$	Clear AC
CLE	rB_{10}:	$E \leftarrow 0$	Clear E
CMA	rB_9:	$AC \leftarrow \overline{AC}$	Complement AC
CME	rB_8:	$E \leftarrow \overline{E}$	Complement E
CIR	rB_7:	$AC \leftarrow$ shr AC, $AC(15) \leftarrow E$, $E \leftarrow AC(0)$	Circulate right
CIL	rB_6:	$AC \leftarrow$ shl AC, $AC(0) \leftarrow E$, $E \leftarrow AC(15)$	Circulate left
INC	rB_5:	$AC \leftarrow AC + 1$	Increment AC
SPA	rB_4:	If $(AC(15) = 0)$ then $(PC \leftarrow PC + 1)$	Skip if positive
SNA	rB_3:	If $(AC(15) = 1)$ then $(PC \leftarrow PC + 1)$	Skip if negative
SZA	rB_2:	If $(AC = 0)$ then $PC \leftarrow PC + 1)$	Skip if AC zero
SZE	rB_1:	If $(E = 0)$ then $(PC \leftarrow PC + 1)$	Skip if E zero
HLT	rB_0:	$S \leftarrow 0$ (S is a start–stop flip-flop)	Halt computer

Timing and Control

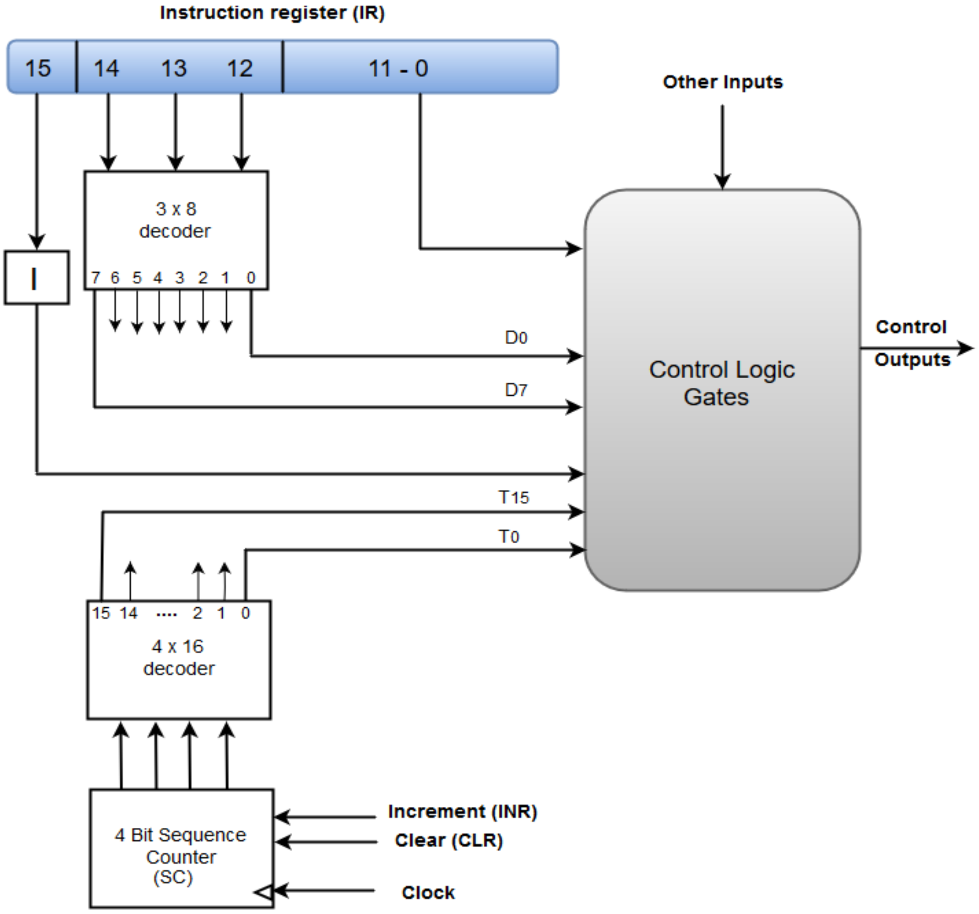
The timing for all registers in the basic computer is controlled by a master clock generator. The clock pulses are applied to all flip-flops and registers in the system, including the flip-flops and registers in the control unit. The clock pulses do not change the state of a register unless the register is enabled by a control signal. The control signals are generated in the control unit and provide control inputs for the multiplexers in the common bus, control inputs in processor registers, and microoperations for the accumulator.

There are two major types of control organization:

3. hardwired control and
4. microprogrammed control.

In the hardwired organization, the control logic is implemented with gates, flip-flops, decoders, and other digital circuits. It has the advantage that it can be optimized to produce a fast mode of operation. In the microprogrammed organization, the control information is stored in a control memory. The control memory is programmed to initiate the required sequence of microoperations. A hardwired control, as the name implies, requires changes in the wiring among the various components if the design has to be modified or changed.

Control Unit of a Basic Computer:



In the microprogrammed control, any required changes or modifications can be done by updating the microprogram in control memory.

Instruction Cycle

A program residing in the memory unit of a computer consists of a sequence of instructions. These instructions are executed by the processor by going through a cycle for each instruction.

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

5. Fetch instruction from memory.
6. Decode the instruction.
7. Read the effective address from memory.
8. 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 and Decode

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 T0. After each clock pulse, SC is incremented by one, so that the timing signals go through a sequence T0, T1, T2, and so on. The microoperations for the fetch and decode phases can be specified by the following register transfer statements.

T0: AR ← PC

T1: IR ← M[AR], PC ← PC + 1

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

Since only AR is connected to the address inputs of memory, it is necessary to transfer the address from PC to AR during the clock transition associated with timing signal T0. The instruction read from memory is then placed in the instruction register IR with the clock transition associated with timing signal T1.

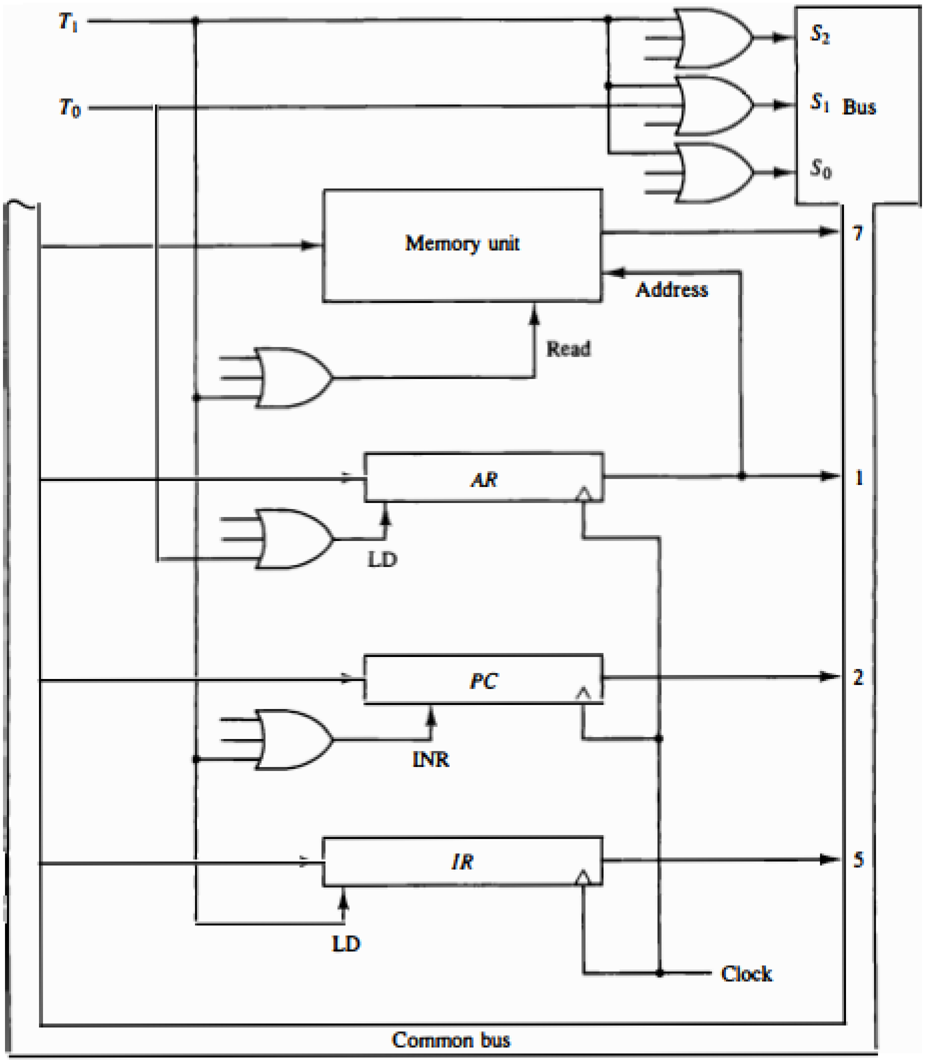


Figure Register transfers for the fetch phase.

At the same time, PC is incremented by one to prepare it for the address of the next instruction in the program. At time T2, the operation code in IR is decoded, the indirect bit is transferred to flip-flop I, and the address part of the instruction is transferred to AR. Note that SC is incremented after each clock pulse to produce the sequence T0, T1, and T2. Figure above shows how the first two register transfer statements are implemented in the bus system. To provide the data path for the transfer of PC to AR we must apply timing signal T0 to achieve the following connection:

1. Place the content of PC onto the bus by making the bus selection inputs

$S_2S_1S_0$ equal to 010.

2. Transfer the content of the bus to AR by enabling the LD input of AR .

The next clock transition initiates the transfer from PC to AR since T0 = 1. In order to implement the second statement : T1: IR ← M[AR], PC ← PC + 1 it is necessary to use timing signal T1 to provide the following connections in the bus system.

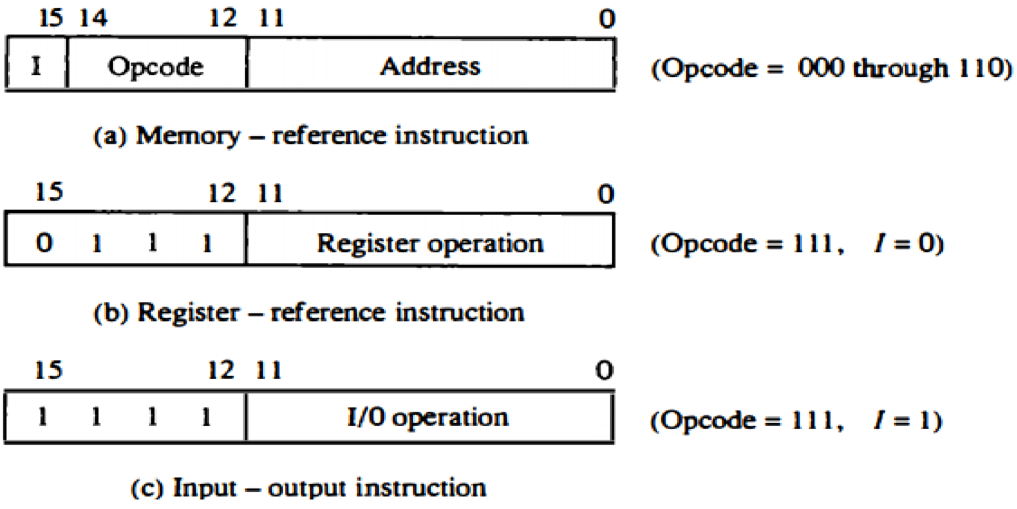
1. Enable the read input of memory.
2. Place the content of memory onto the bus by making $S_2S_1S_0 = 111$.
3. Transfer the content of the bus to IR by enabling the LD input of IR.
4. Increment PC by enabling the INR input of PC.

The next clock transition initiates the read and increment operations since T1 = 1. Figure above duplicates a portion of the bus system and shows how T0 and T1 are connected to the control inputs of the registers, the memory, and the bus selection inputs. Multiple input OR gates are included in the diagram because there are other control functions that will initiate similar operations.

Determine the Type of Instruction

The timing signal that is active after the decoding is T_3. During time T_3, the control unit determines the type of instruction that was just read from memory. The flowchart of Fig. below presents an initial configuration for the instruction cycle and shows how the control determines the instruction type after the decoding. The three possible instruction types available in the basic computer are specified in Fig. on basic computer formats.

Figure Basic computer instruction formats.

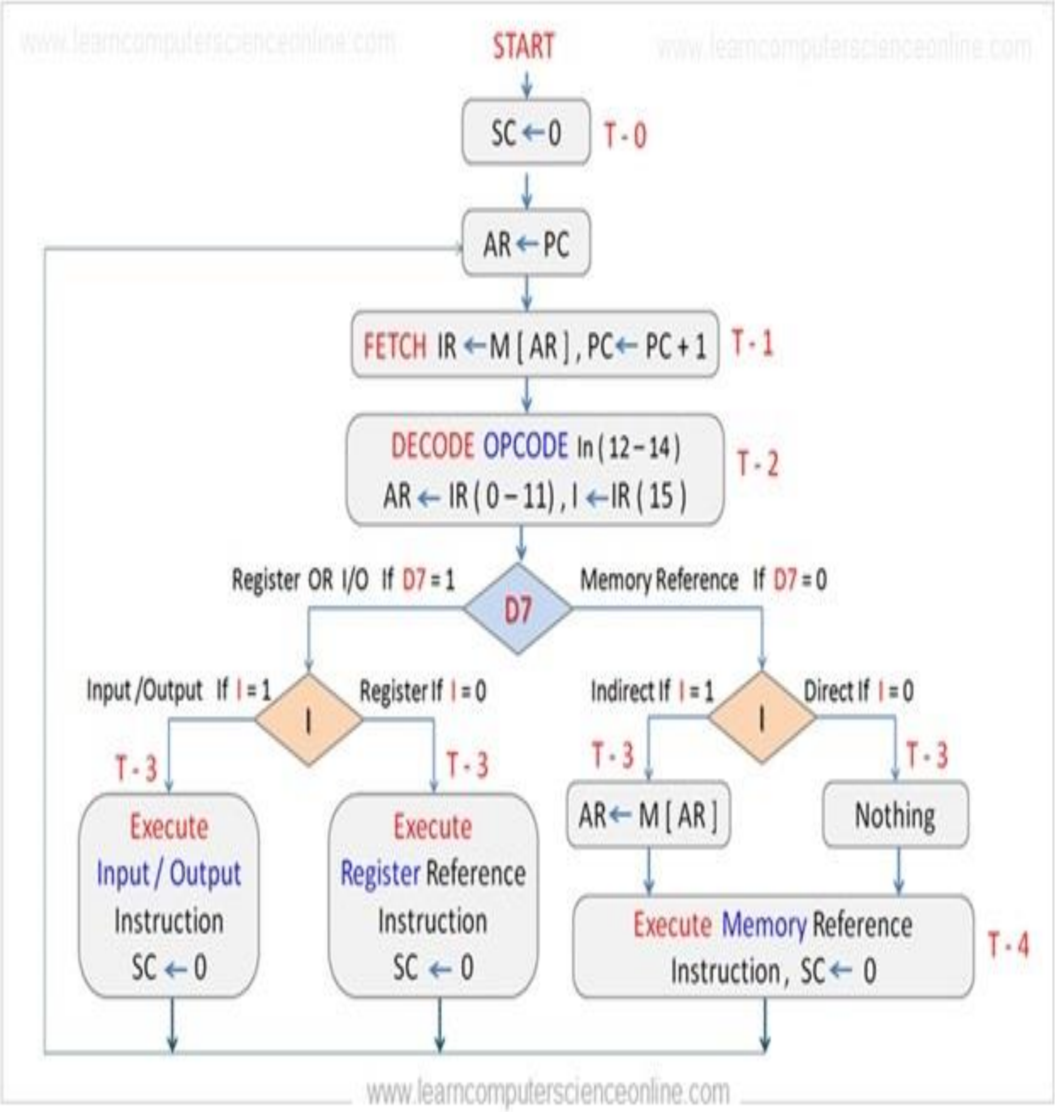


(a) Memory – reference instruction

(b) Register – reference instruction

(c) Input – output instruction

Decoder output D_7 is equal to 1 if the operation code is equal to binary 111. From Fig. on basic computer formats we determine that if $D_7 = 1$, the instruction must be a register-reference or input-output type.



If $D_7 = 0$, the operation code must be one of the other seven values 000 through 110, specifying a memory-reference instruction. Control then inspects the value of the first bit of the instruction, which is now available in flip-flop I. If D7 = 0 and I = 1, we have a memory reference instruction with an indirect address.

It is then necessary to read the effective address from memory. The microoperation for the indirect address condition can be symbolized by the register transfer statement : AR ← M[AR]. Initially, AR holds the address part of the instruction. This address is used during the memory read operation. The word at the address given by AR is read from memory and placed on the common bus. The LD input of AR is then enabled to receive the indirect address that resided in the 12 least significant bits of the memory word.

The three instruction types are subdivided into four separate paths. The selected operation is activated with the clock transition associated with timing signal T_3. This can be symbolized as follows:

$D'_7 IT_3$: AR ← M[AR]

$D'_7 I'T_3$: Nothing

$D_7 I'T_3$: Execute a register-reference instruction

D_7IT_3: Execute an input-output instruction

When a memory-reference instruction with I = 0 is encountered, it is not necessary to do anything since the effective address is already in AR. However, the sequence counter SC must be incremented when $D'_7T_3 = 1$, so that the execution of the memory-reference instruction can be continued with timing variable T_4.

A register-reference or input-output instruction can be executed with the clock associated with timing signal T_3. After the instruction is executed, SC is cleared to 0 and control returns to the fetch phase with $T_0 = 1$. Note that the sequence counter SC is either incremented or cleared to 0 with every positive clock transition. We will adopt the convention that if SC is incremented, we will not write the statement SC ← SC + 1, but it will be implied that the control goes to the next timing signal in sequence. When SC is to be cleared, we will include the statement SC ← 0. The register transfers needed for the execution of the register-reference instructions are presented in this section.

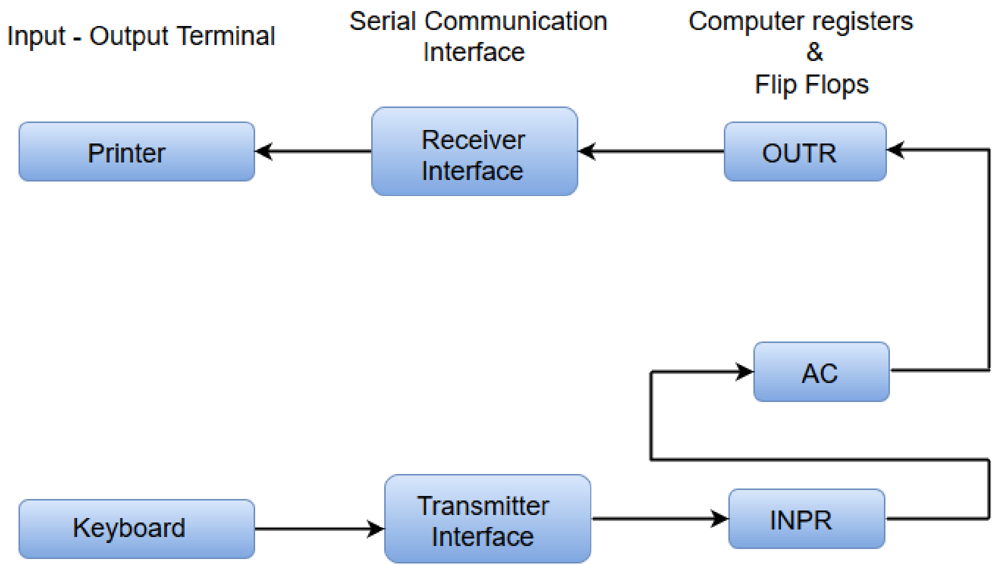
Input-Output and Interrupt

A computer can serve no useful purpose unless it communicates with the external environment. Instructions and data stored in memory must come from some input device. Computational results must be transmitted to the user through some output device. Commercial computers include many types of input and output devices. To demonstrate the most basic requirements for input and output communication, we will use as an illustration a terminal unit with a keyboard and printer.

Input-Output Configuration

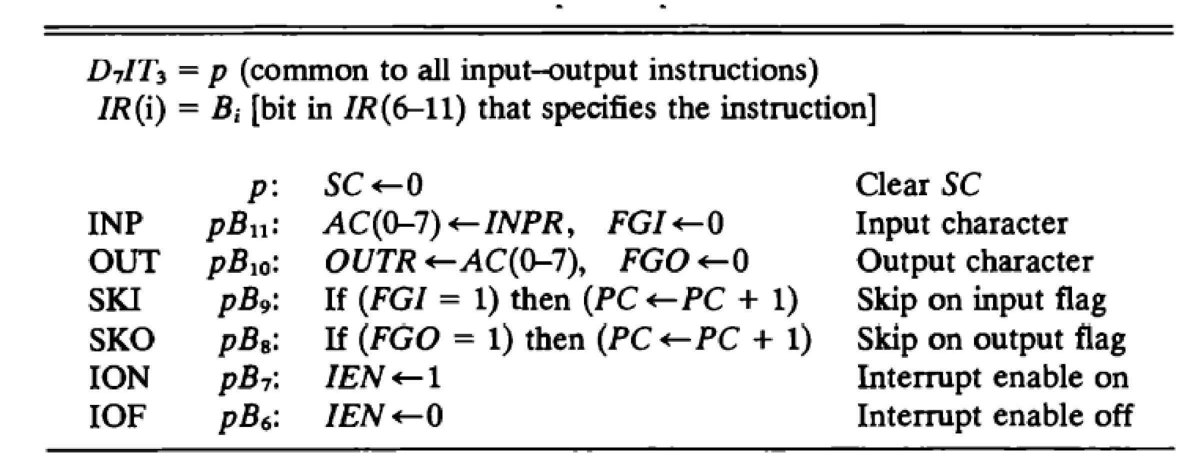
The terminal sends and receives serial information. Each quantity of information has eight bits of an alphanumeric code. The serial information from the keyboard is shifted into the input register INPR. The serial information for the printer is stored in the output register OUTR. These two registers communicate with a communication interface serially and with the AC in parallel. The input-output configuration is shown in Figure. The transmitter interface receives serial information from the keyboard and transmits it to INPR. The receiver interface receives information from OUTR and sends it to the printer serially.

Input - Output Configuration:



The input register INPR consists of eight bits and holds an alphanumeric input information. The 1-bit 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. The flag is needed to synchronize the timing rate difference between the input device and the computer. The process of information transfer is as follows. Initially, the input flag FGI is cleared to 0. When a key is struck in the keyboard, an 8-bit alphanumeric code is shifted into INPR and the input flag FGI is set to 1. As long as the flag is set, the information in INPR cannot be changed by striking another key. The computer checks the flag bit; if it is 1, the information from INPR is transferred in parallel into AC and FGI is cleared to 0. Once the flag is cleared, new information can be shifted into INPR by striking another key.

Input-Output Reference Instructions



$D_7IT_3 = p$ (common to all input–output instructions)
$IR(i) = B_i$ [bit in $IR(6–11)$ that specifies the instruction]

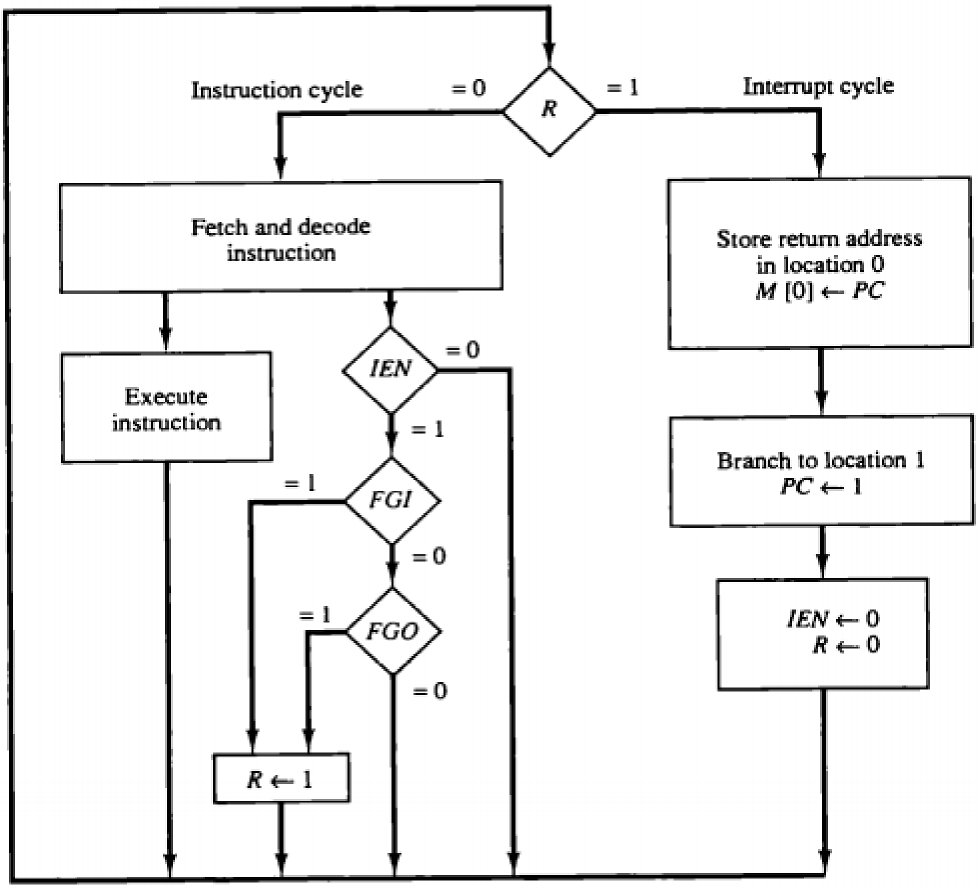
	p:	$SC \leftarrow 0$	Clear SC
INP	pB_{11}:	$AC(0–7) \leftarrow INPR, \quad FGI \leftarrow 0$	Input character
OUT	pB_{10}:	$OUTR \leftarrow AC(0–7), \quad FGO \leftarrow 0$	Output character
SKI	pB_9:	If $(FGI = 1)$ then $(PC \leftarrow PC + 1)$	Skip on input flag
SKO	pB_8:	If $(FGO = 1)$ then $(PC \leftarrow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

Program Interrupt

The computer keeps checking the flag bit, and when it finds it set, it initiates an information transfer. The difference of information flow rate between the computer and that of the input—output device makes this type of transfer inefficient. An alternative to the programmed controlled procedure is to let the external device inform the computer when it is ready for the transfer. In the meantime the computer can be busy with other tasks. This type of transfer uses the interrupt facility. While the computer is running a program, it does not check the flags. When a flag is set, the computer is momentarily interrupted from the current program. The computer deviates momentarily from what it is doing to perform of the input or output transfer. It then returns to the current program to continue what it was doing before the interrupt.

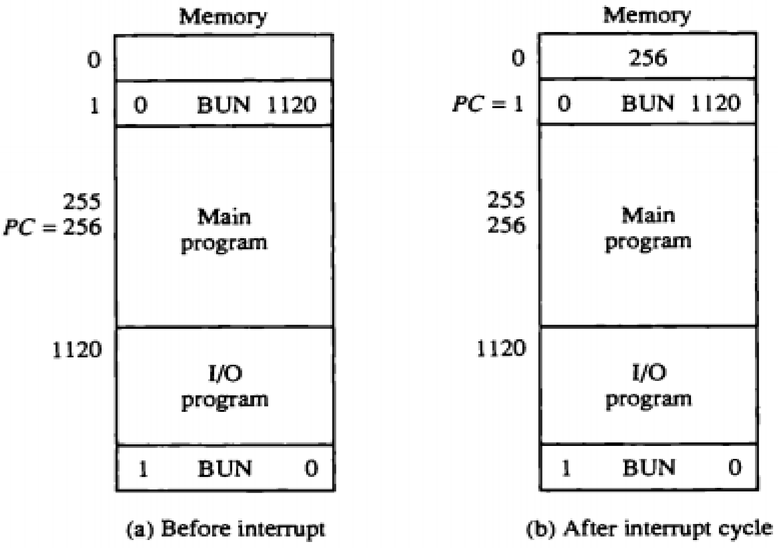
The interrupt enable flip-flop IEN can be set and cleared with two instructions. When IEN is cleared to 0 (with the IOF instruction), the flags cannot interrupt the computer. When IEN is set to (with the ION instruction), the computer can be interrupted. The way that the interrupt is handled by the computer can be explained by means of the flowchart.

An interrupt flip-flop R is included in the computer. When R = 0, the computer goes through an instruction cycle. During the execute phase of the instruction cycle IEN is checked by the control. If it is 0, it indicates that the programmer does not want to use the interrupt,so control continues with the next instruction cycle. If IEN is 1, control checks the flag bits. If both flags are 0, it indicates that neither the input nor the output registers are ready for transfer of information. In this case, control continues with the next instruction cycle. If either flag is set to 1 while 1EN = 1, flip-flop R is set to 1. At the end of the execute phase, control checks the value of R, and if it is equal to 1, it goes to an interrupt cycle instead of an instruction cycle.



Interrupt cycle

The interrupt cycle is a hardware implementation of a branch and save return address operation. The return address available in PC is stored in a specific location. This location may be a processor register, a memory stack, or a specific memory location. An example that shows what happens during the interrupt cycle is shown in the following Figure.



(a) Before interrupt (b) After interrupt cycle

When an interrupt occurs and R is set to 1 while the control is executing the instruction at address 255. At this time, the returns address 256 is in PC. The programmer has previously placed an input—output service program in memory starting from address 1120 and a BUN 1120 instruction at address 1. This is shown in Figure(a). When control reaches timing signal T0and finds that R = 1, it proceeds with the interrupt cycle. The content of PC (256) is stored in memory location 0, PC is set to 1, and R is cleared to 0. The branch instruction at address 1 causes the program to transfer to the input—output service program at address 1120. This program checks the flags, determines which flag is set, and then transfers the required input or output information. Once this is done, the instruction ION is executed to set IEN to 1 (to enable further interrupts), and the program returns to the location where it was interrupted. This is shown in Figure(b).

SREENIVASA INSTITUTE OF TECHNOLOGY AND MANAGEMENT STUDIES.
(AUTONOMOUS)
MCA DEPARTMENT

LECTURE NOTES

COMPUTER ORGANIZATION AND ARCHITECTURE

UNIT-4

Micro Programmed Control And Central Processing Unit

Introduction

- The function of the control unit in a digital computer is to initiate sequence of microoperations.
- Control unit can be implemented in two ways
  1. Hardwired control
  2. Microprogrammed control

Hardwired Control:

- When the control signals are generated by hardware using conventional logic design techniques, the control unit is said to be hardwired.
- The key characteristics are
  - High speed of operation
  - Expensive
  - Relatively complex
  - No flexibility of adding new instructions

Examples of CPU with hardwired control unit are

- Intel 8085,
- Motorola 6802,
- Zilog 80,
- and any RISC CPUs.

Microprogrammed Control:

- Control information is stored in control memory.
- Control memory is programmed to initiate the required sequence of micro-operations.
- The key characteristics are
  - Speed of operation is low when compared with hardwired
  - Less complex
  - Less expensive
  - Flexibility to add new instructions
- Examples of CPU with microprogrammed control unit are
  - Intel 8080,
  - Motorola 68000
  - and any CISC CPUs.

1. Control Memory:

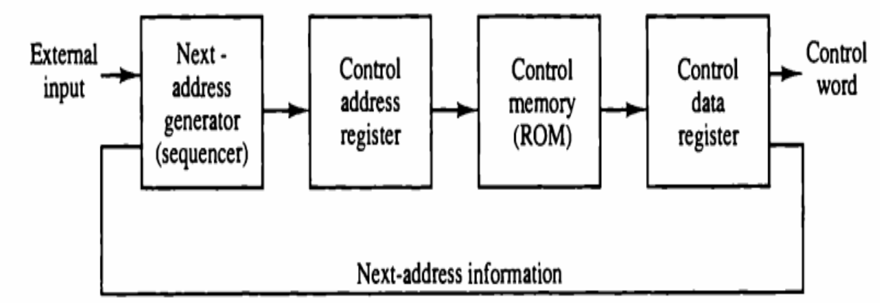
- A control memory is a part of the control unit. Any computer that involves microprogrammed control consists of two memories. They are the main memory and the control memory.

- Programs are usually stored in the main memory by the users. Whenever the programs change, the data is also modified in the main memory. They consist of machine instructions and data.
- The control memory consists of microprograms that are fixed and cannot be modified frequently. They contain microinstructions that specify the internal control signals required to execute register micro-operations.
- The machine instructions generate a chain of microinstructions in the control memory. Their function is to generate micro-operations that can fetch instructions from the main memory, compute the effective address, execute the operation, and return control to fetch phase and continue the cycle.
- The control function that specifies a microoperation is called as **control variable.**
- When control variable is in one binary state, the corresponding microoperation is executed. For the other binary state the state of registers does not change.
- The active state of a control variable may be either 1 state or the 0 state, depending on the application.

Control Word: The control variables at any given time can be represented by a string of 1's and 0's called a control word. All control words can be programmed to perform various operations on the components of the system.

Microprogram control unit: A control unit whose binary control variables are stored in memory is called a microprogram control unit.

- The control word in control memory contains a microinstruction.
- The microinstruction specifies one or more micro-operations for the system.
- A sequence of microinstructions constitutes a microprogram.
- The control unit consists of control memory used to store the microprogram.
- Control memory is a permanent i.e., read only memory (ROM).
- The general configuration of a micro-programmed control unit organization is shown as block diagram below.

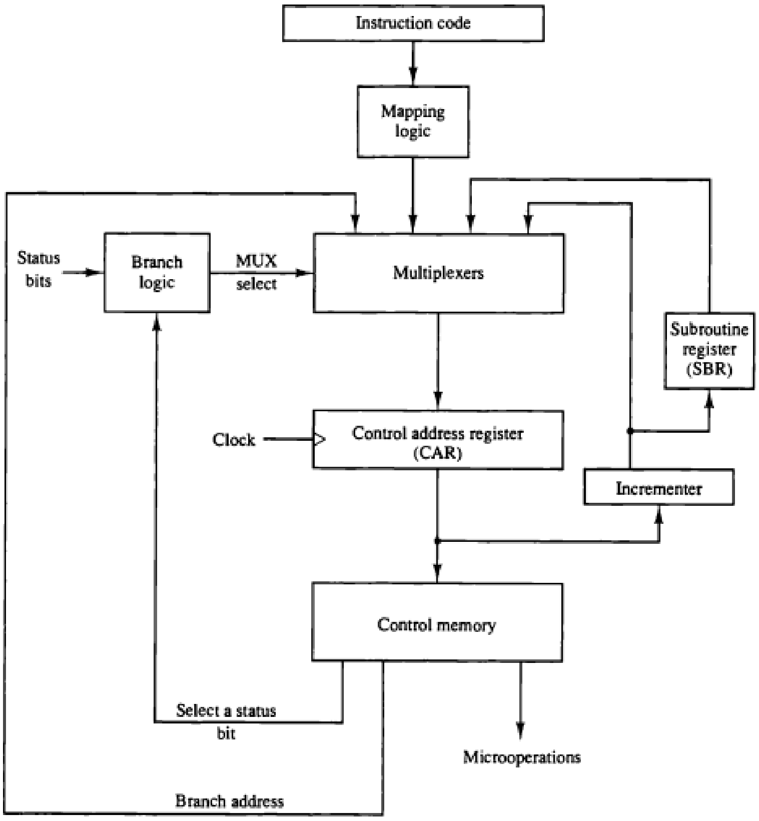


- ➢ The control memory is ROM so all control information is permanently stored.

- The control memory address register (CAR) specifies the address of the microinstruction and the control data register (CDR) holds the microinstruction read from memory.
- The next address generator is sometimes called a microprogram sequencer. It is used to generate the next micro instruction address.
- The location of the next microinstruction may be the one next in sequence or it may be located somewhere else in the control memory.
- So it is necessary to use some bits of the present microinstruction to control the generation of the address of the microinstruction.
- Sometimes the next address may also be a function of external input conditions.
- The control data register holds the present microinstruction while next address is computed and read from memory. The data register is times called a pipeline register.

2. Address Sequencing:

- Microinstructions are stored in control memory in groups, with each group specifying a routine.
- Each computer instruction has its own microprogram routine to generate the microoperations.
- The hardware that controls the address sequencing of the control memory must be capable of sequencing the microinstructions within a routine and be able to branch from one routine to another
- Control must undergo the following steps during the execution of a single computer instruction:
- Load an initial address into the CAR when power is turned on in the computer. This address is usually the address of the first microinstruction that activates the instruction fetch routine – IR holds instruction
- The control memory then goes through the routine to determine the effective address of the operand – AR holds operand address
- The next step is to generate the microoperations that execute the instruction by considering the opcode and applying a mapping process.
- The transformation of the instruction code bits to an address in control memory where the routine of instruction located is referred to as **mapping process.**
- After execution, control must return to the fetch routine by executing an unconditional branch.
- The below figure shows a block diagram of a control memory and the associated hardware needed for selecting the next microinstruction address.



- The microinstruction in control memory contains a set of bits to initiate microoperations in computer registers and other bits to specify the method by which the next address is obtained.
- In the figure four different paths form which the control address register (CAR) receives the address.
  - The incrementer increments the content of the control register address register by one, to select the next microinstruction in sequence.
  - Branching is achieved by specifying the branch address in one of the fields of the microinstruction.
  - Conditional branching is obtained by using part of the microinstruction to select a specific status bit in order to determine its condition.
  - An external address is transferred into control memory via a mapping logic circuit.
  - The return address for a subroutine is stored in a special register, that value is used when the micoprogram wishes to return from the subroutine.

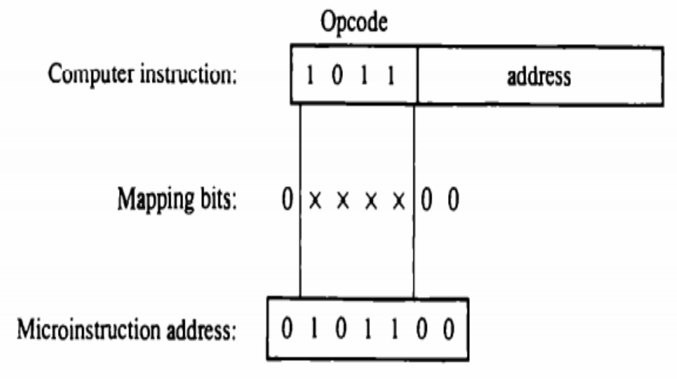
Conditional Branching:

- Conditional branching is obtained by using part of the microinstruction to select a specific status bit in order to determine its condition.

- The status conditions are special bits in the system that provide parameter information such as the carry-out of an adder, the sign bit of a number, the mode bits of an instruction, and i/o status conditions.
- The status bits, together with the field in the microinstruction that specifies a branch address, control the branch logic.
- The branch logic tests the condition, if met then branches, otherwise, increments the CAR.
- If there are 8 status bit conditions, then 3 bits in the microinstruction are used to specify the condition and provide the selection variables for the multiplexer.
- For unconditional branching, fix the value of one status bit to be one load the branch address from control memory into the CAR.

Mapping of Instruction:

- A special type of branch exists when a microinstruction specifies a branch to the first word in control memory where a microprogram routine is located.
- The status bits for this type of branch are the bits in the opcode.
- Assume an opcode of four bits and a control memory of 128 locations. The mapping process converts the 4-bit opcode to a 7-bit address for control memory shown in below figure



- Mapping consists of placing a 0 in the most significant bit of the address, transferring the four operation code bits, and clearing the two least significant bits of the control address register.
- This provides for each computer instruction a microprogram routine with a capacity of four microinstructions.

Subroutines:

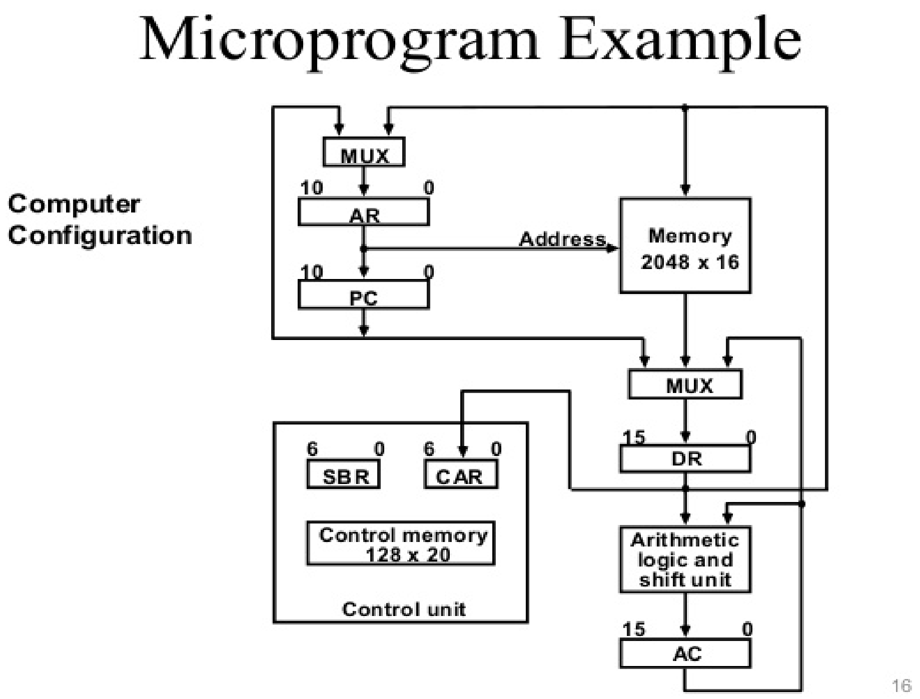
- Subroutines are programs that are used by other routines to accomplish a particular task and can be called from any point within the main body of the microprogram.
- Frequently many microprograms contain identical section of code.
- Microinstructions can be saved by employing subroutines that use common sections of microcode.
- Microprograms that use subroutines must have a provision for storing the return address during a subroutine call and restoring the address during a subroutine return.
- A subroutine register is used as the source and destination for the addresses.

3. Microprogram Example:

- The process of code generation for the control memory is called microprogramming.
- The block diagram of the computer configuration is shown in the figure.
- Two memory units:
    1. Main memory – stores instructions and data
    2. Control memory – stores microprogram
- Four processor registers
    1. Program counter – PC
    2. Address register – AR
    3. Data register – DR

Accumulator register – AC

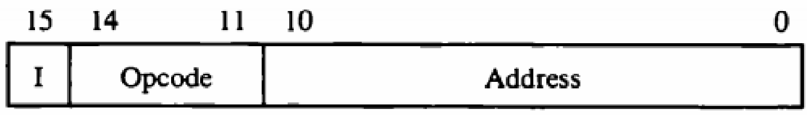
- Two control unit registers
    1. Control address register – CAR
    2. Subroutine register – SBR
- Transfer of information among registers in the processor is through MUXs rather than a bus.



Microprogram Example

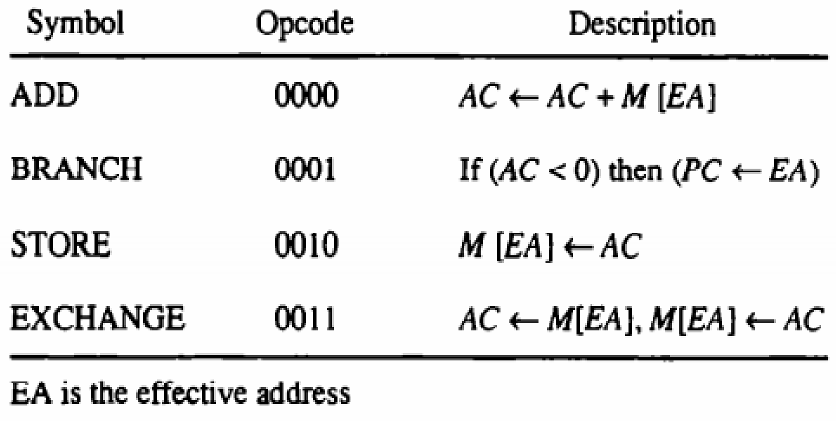
Computer Configuration

- The computer instruction format is shown in below figure.



(a) Instruction format

- Three fields for an instruction:
    1. 1-bit field for indirect addressing
    2. 4-bit opcode
    3. 11-bit address field
- The example will only consider the following 4 of the possible 16 memory instructions

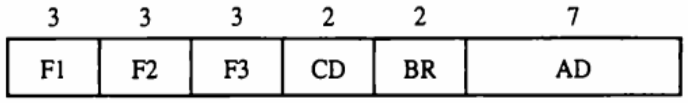


Symbol	Opcode	Description
ADD	0000	$AC \leftarrow AC + M[EA]$
BRANCH	0001	If $(AC < 0)$ then $(PC \leftarrow EA)$
STORE	0010	$M[EA] \leftarrow AC$
EXCHANGE	0011	$AC \leftarrow M[EA], M[EA] \leftarrow AC$

EA is the effective address

(b) Four computer instructions

The microinstruction format for the control memory is shown in below figure.



F1, F2, F3: Microoperation fields

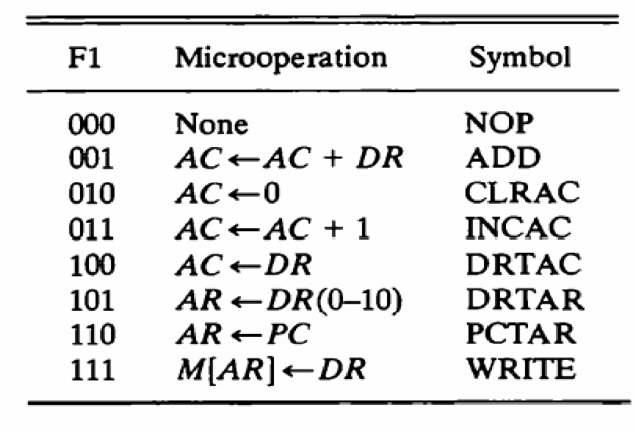
CD: Condition for branching

BR: Branch field

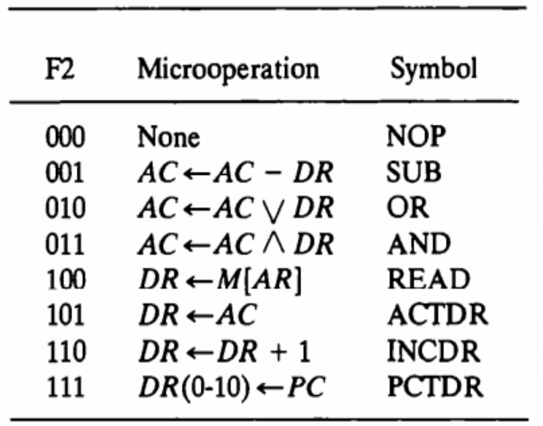
AD: Address field

- The microinstruction format is composed of 20 bits with four parts to it
- Three fields F1, F2, and F3 specify microoperations for the computer [3 bits each]
- The CD field selects status bit conditions [2 bits]
- The BR field specifies the type of branch to be used [2 bits]
- The AD field contains a branch address [7 bits]
- Each of the three microoperation fields can specify one of seven possibilities.
- No more than three microoperations can be chosen for a microinstruction.

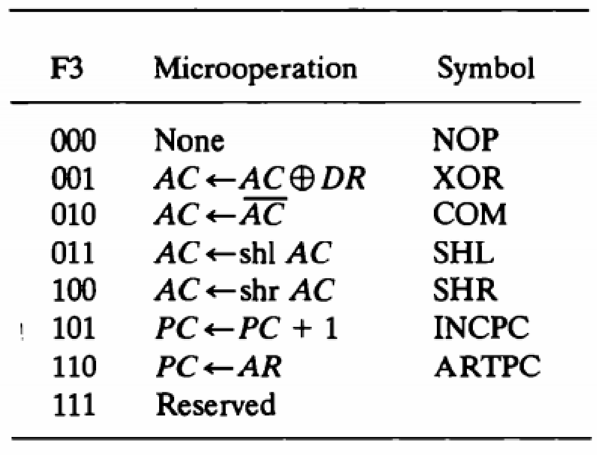
- If fewer than three are needed, the code 000 = NOP.
- The three bits in each field are encoded to specify seven distinct microoperations listed in below table



F1	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC + DR$	ADD
010	$AC \leftarrow 0$	CLRAC
011	$AC \leftarrow AC + 1$	INCAC
100	$AC \leftarrow DR$	DRTAC
101	$AR \leftarrow DR(0\text{–}10)$	DRTAR
110	$AR \leftarrow PC$	PCTAR
111	$M[AR] \leftarrow DR$	WRITE



F2	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC - DR$	SUB
010	$AC \leftarrow AC \vee DR$	OR
011	$AC \leftarrow AC \wedge DR$	AND
100	$DR \leftarrow M[AR]$	READ
101	$DR \leftarrow AC$	ACTDR
110	$DR \leftarrow DR + 1$	INCDR
111	$DR(0\text{-}10) \leftarrow PC$	PCTDR

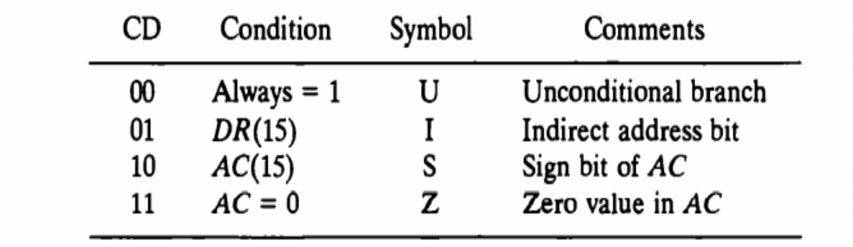


F3	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC \oplus DR$	XOR
010	$AC \leftarrow \overline{AC}$	COM
011	$AC \leftarrow \text{shl } AC$	SHL
100	$AC \leftarrow \text{shr } AC$	SHR
101	$PC \leftarrow PC + 1$	INCPC
110	$PC \leftarrow AR$	ARTPC
111	Reserved	

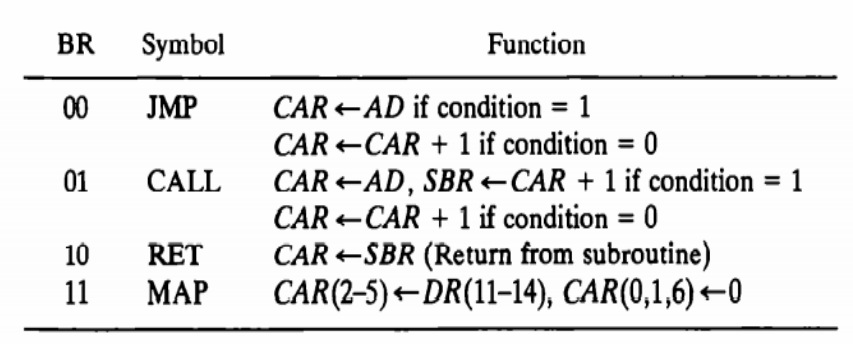
- Five letters to specify a transfer-type microoperation
- First two designate the source register
- Third is a ‘T’
- Last two designate the destination register AC ← DR

F1 = 100 = DRTAC

- The condition field (CD) is two bits to specify four status bit conditions shown below
- The branch field (BR) consists of two bits and is used with the address field to choose the address of the next microinstruction



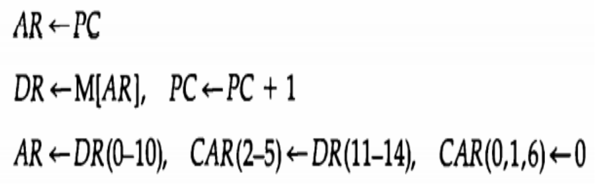
CD	Condition	Symbol	Comments
00	Always = 1	U	Unconditional branch
01	$DR(15)$	I	Indirect address bit
10	$AC(15)$	S	Sign bit of AC
11	$AC = 0$	Z	Zero value in AC



BR	Symbol	Function
00	JMP	$CAR \leftarrow AD$ if condition = 1 $CAR \leftarrow CAR + 1$ if condition = 0
01	CALL	$CAR \leftarrow AD$, $SBR \leftarrow CAR + 1$ if condition = 1 $CAR \leftarrow CAR + 1$ if condition = 0
10	RET	$CAR \leftarrow SBR$ (Return from subroutine)
11	MAP	$CAR(2\text{-}5) \leftarrow DR(11\text{-}14)$, $CAR(0,1,6) \leftarrow 0$

Fetch Routine:

- The control memory has 128 locations, each one is 20 bits.
- The first 64 locations are occupied by the routines for the 16 instructions, addresses 0-63.
- Can start the fetch routine at address 64.
- The fetch routine requires the following three microinstructions (locations 64-66).
- The microinstructions needed for fetch routine are:

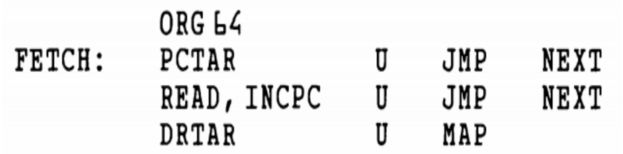


$AR \leftarrow PC$

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

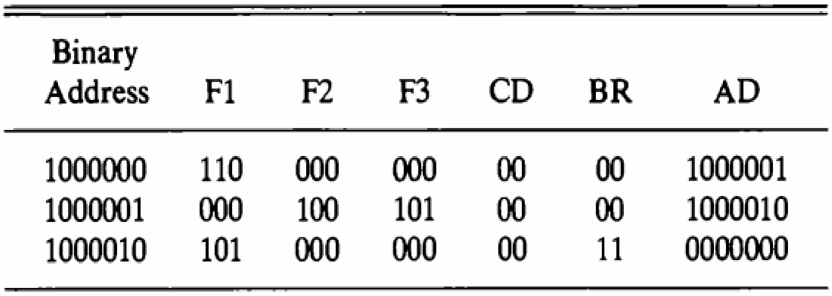
$AR \leftarrow DR(0\text{-}10),\quad CAR(2\text{-}5) \leftarrow DR(11\text{-}14),\quad CAR(0,1,6) \leftarrow 0$

- It's Symbolic microprogram:



```
          ORG 64
FETCH:    PCTAR        U    JMP    NEXT
          READ, INCPC  U    JMP    NEXT
          DRTAR        U    MAP
```

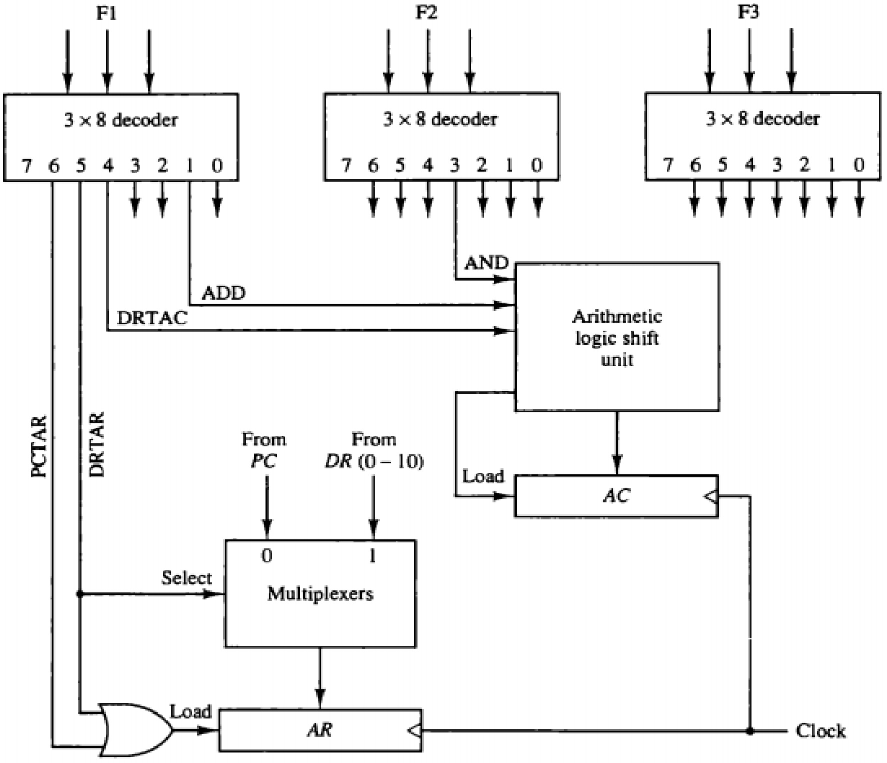
- It's Binary microprogram:



Binary Address	F1	F2	F3	CD	BR	AD
1000000	110	000	000	00	00	1000001
1000001	000	100	101	00	00	1000010
1000010	101	000	000	00	11	0000000

4. Design of control Unit:

- The control memory out of each subfield must be decoded to provide the distinct microoperations.
- The outputs of the decoders are connected to the appropriate inputs in the processor unit.
- The below figure shows the three decoders and some of the connections that must be made from their outputs.
- The three fields of the microinstruction in the output of control memory are decoded with a 3x8 decoder to provide eight outputs.
- Each of the output must be connected to proper circuit to initiate the corresponding microoperation as specified in previous topic.

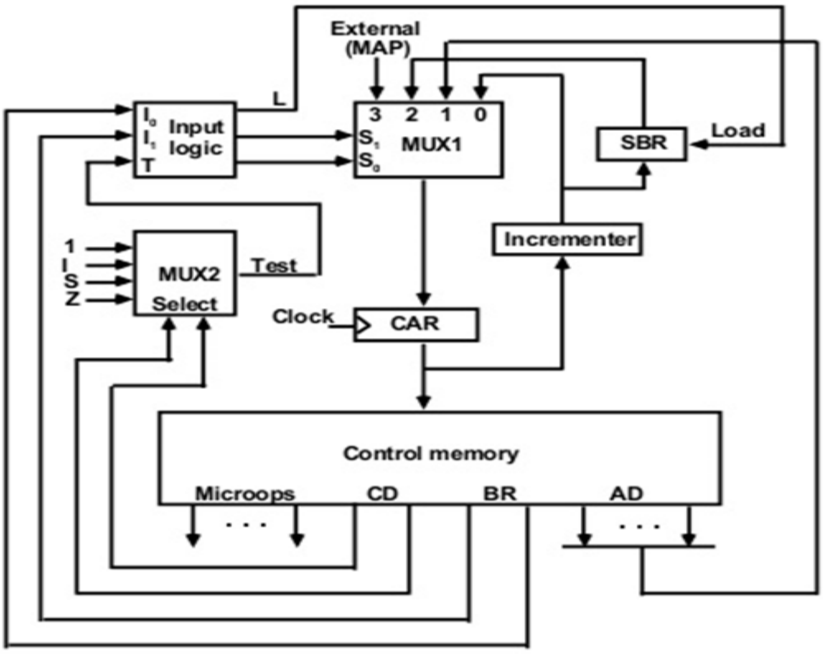


- ➢ When F1 = 101 (binary 5), the next pulse transition transfers the content of DR (0-10) to AR.
- ➢ Similarly, when F1= 110 (binary 6) there is a transfer from PC to AR (symbolized by PCTAR). As shown in Fig, outputs 5 and 6 of decoder F1 are connected to the load input of AR so that when either one of these outputs is active, information from the multiplexers is transferred to AR.
- ➢ The multiplexers select the information from DR when output 5 is active and from PC when output 5 is inactive.
- ➢ The transfer into AR occurs with a clock transition only when output 5 or output 6 of the decoder is active.

- For the arithmetic logic shift unit the control signals are instead of coming from the logical gates, now these inputs will now come from the outputs of AND, ADD and DRTAC respectively

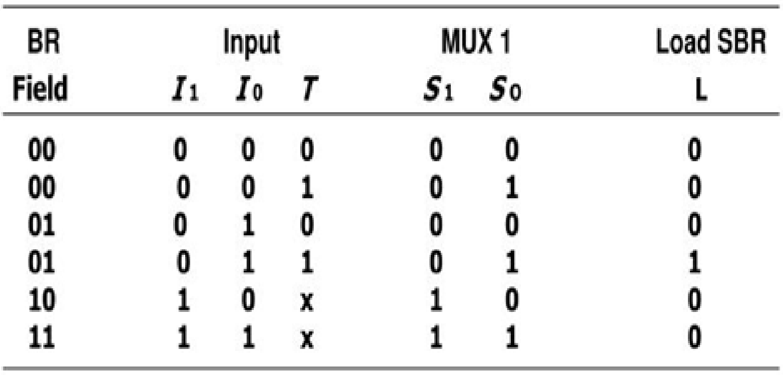
Microprogram Sequencer:

- The basic components of a microprogrammed control unit are the control memory and the circuits that select the next address.
- The address selection part is called a microprogram sequencer.
- The purpose of a microprogram sequencer is to present an address to the control memory so that a microinstruction may be read and executed.
- The next-address logic of the sequencer determines the specific address source to be loaded into the control address register.
- The block diagram of the microprogram sequencer is shown in below figure.



- The control memory is included in the diagram to show the interaction between the sequencer and the memory attached to it.
- There are two multiplexers in the circuit.
  - ❑ The first multiplexer selects an address from one of four sources and routes it into control address register CAR.
  - ❑ The second multiplexer tests the value of a selected status bit and the result of the test is applied to an input logic circuit.
- The output from CAR provides the address for the control memory.

- The content of CAR is incremented and applied to one of the multiplexer inputs and to the subroutine register SBR.
- The other three inputs to multiplexer come from
  - ❑ The address field of the present microinstruction
  - ❑ From the out of SBR
  - ❑ From an external source that maps the instruction
- The CD (condition) field of the microinstruction selects one of the status bits in the second multiplexer.
- If the bit selected is equal to 1, the T variable is equal to 1; otherwise, it is equal to 0.
- The T value together with two bits from the BR (branch) field goes to an input logic circuit.
- The input logic in a particular sequencer will determine the type of operations that are available in the unit.
- The input logic circuit in above figure has three inputs I0, I1, and T, and three outputs, S0, S1, and L.
- Variables S0 and S1 select one of the source addresses for CAR. Variable L enables the load input in SBR.
- The binary values of the selection variables determine the path in the multiplexer.
- For example, with S1,S0 = 10, multiplexer input number 2 is selected and establishes transfer path from SBR to CAR.
- The truth table for the input logic circuit is shown in Table below.



BR	Input			MUX 1		Load SBR
Field	I_1	I_0	T	S_1	S_0	L
00	0	0	0	0	0	0
00	0	0	1	0	1	0
01	0	1	0	0	0	0
01	0	1	1	0	1	1
10	1	0	x	1	0	0
11	1	1	x	1	1	0

- Inputs I1 and I0 are identical to the bit values in the BR field.
- The bit values for S1 and S0 are determined from the stated function and the path in the multiplexer that establishes the required transfer.
- The subroutine register is loaded with the incremented value of CAR during a call microinstruction (BR = 01) provided that the status bit condition is satisfied (T = 1).
- The truth table can be used to obtain the simplified Boolean functions for the input logic circuit:

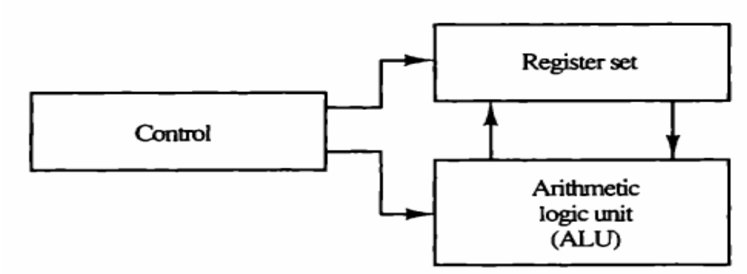
S1=I1
S0=I1I0+I’ 1T

L = I ' 1T I0

Central Processing Unit

Introduction:

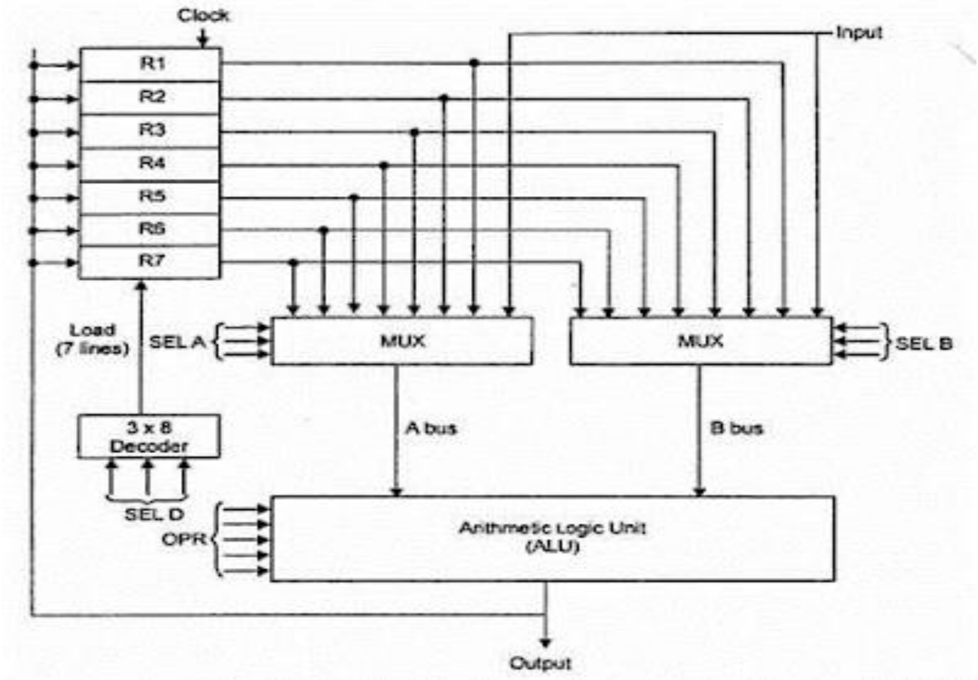
- The main part of the computer that performs the bulk of data-processing operations is called the central processing unit and is referred to as the CPU.
- The CPU is made up of three major parts, as shown in Figure



- 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

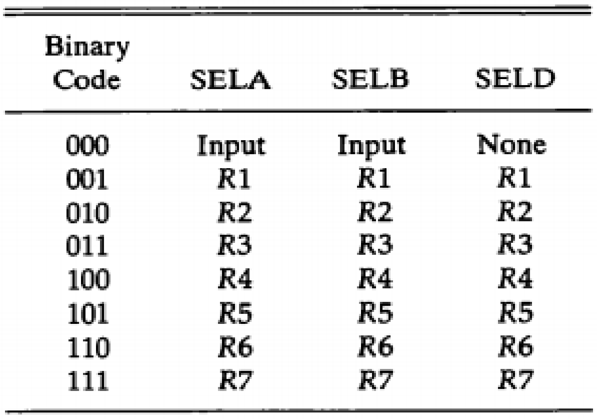
- Generally CPU has seven general registers.
- Register organization show how registers are selected and how data flow between register and ALU.
- A decoder is used to select a particular register.
- The output of each register is connected to two multiplexers to form the two buses A and B.
- The selection lines in each multiplexer select the input data for the particular bus.
- The A and B buses form the two inputs of an ALU.
- The operation select lines decide the micro operation to be performed by ALU.
- The result of the micro operation is available at the output bus.
- The output bus connected to the inputs of all registers, thus by selecting a destination register it is possible to store the result in it.



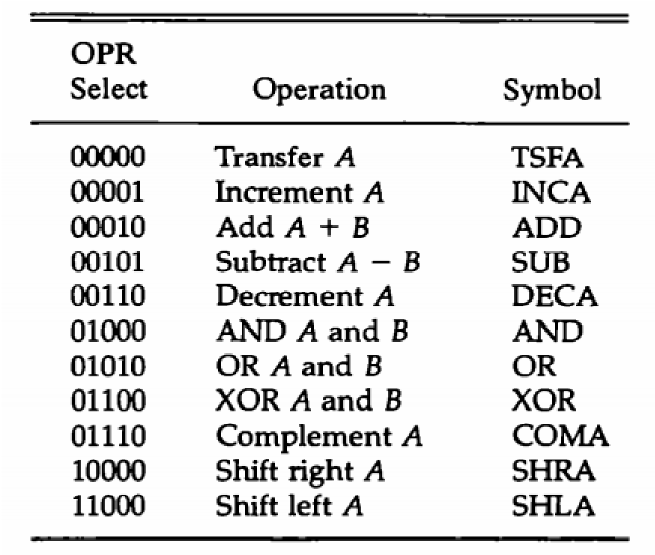
EXAMPLE:

To perform the operation **R3 = R1+R2** We have to provide following binary selection variable to the select inputs.

1. **SEL A** : **001** -To place the contents of R1 into bus A.
2. **SEL B** : **010** - to place the contents of R2 into bus B
3. **SEL OPR** : **10010** – to perform the arithmetic addition A+B
4. **SEL REG or SEL D : 011** – to place the result available on output bus in R3.



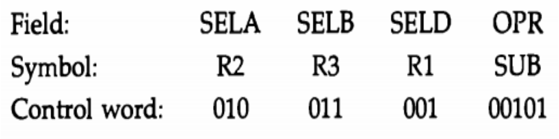
Binary Code	SELA	SELB	SELD
000	Input	Input	None
001	R1	R1	R1
010	R2	R2	R2
011	R3	R3	R3
100	R4	R4	R4
101	R5	R5	R5
110	R6	R6	R6
111	R7	R7	R7



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

CONTROL WORD?

- The combined value of a binary selection inputs specifies the control word.
- It consist of four fields SELA, SELB,and SELD or SELREG contains three bit each and SELOPR field contains four bits thus the total bits in the control word are 13-bits.
- Control Word For Operation R2 = R1+r3



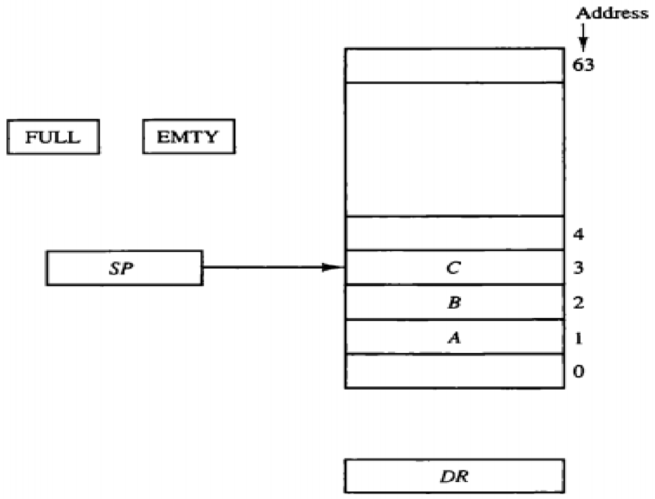
Field:	SELA	SELB	SELD	OPR
Symbol:	R2	R3	R1	SUB
Control word:	010	011	001	00101

1. Stack Organization:
    - **Stack**: A stack is a storage device that stores information in such a manner that the item stored last is the first item retrieved.
    - In the computer stack is a memory unit with an address register that can count the address only.
    - The register that holds the address for the stack is called a stack pointer (SP). It always points at the top item in the stack

- The two operations that are performed on stack are the insertion and deletion.
- The operation of insertion is called PUSH.
- The operation of deletion is called POP.
- These operations are simulated by incrementing and decrementing the stack pointer register (SP)

Register Stack:

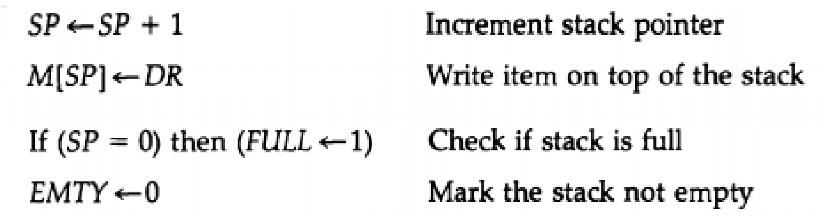
- A stack can be placed in a portion of a large memory or it can be organized as a collection of a finite number of memory words or registers.
- The below figure shows the organization of a 64-word register stack



- ➢ The stack pointer register SP contains a binary number whose value is equal to the address of the word is currently on top of the stack. Three items are placed in the stack: A, B, C, in that order.
- ➢ In above figure C is on top of the stack so that the content of SP is 3.
- ➢ For removing the top item, the stack is popped by reading the memory word at address 3 and decrementing the content of stack SP.
- ➢ Now the top of the stack is B, so that the content of SP is 2.
- ➢ Similarly for inserting the new item, the stack is pushed by incrementing SP and writing a word in the next higher location in the stack.
- ➢ In a 64-word stack, the stack pointer contains 6 bits because 26 = 64.
- ➢ Since SP has only six bits, it cannot exceed a number greater than 63 (111111 in binary).
- ➢ When 63 is incremented by 1, the result is 0 since 111111 + 1. = 1000000 in binary, but SP can accommodate only the six least significant bits.
- ➢ Then the one-bit register FULL is set to 1, when the stack is full.
- ➢ Similarly when 000000 is decremented by 1, the result is 111111, and then the one-bit register EMTY is set 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

PUSH:

- ➢ Initially, SP is cleared to 0, EMTY 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.
- ➢ The push operation is implemented with the following sequence of microoperations

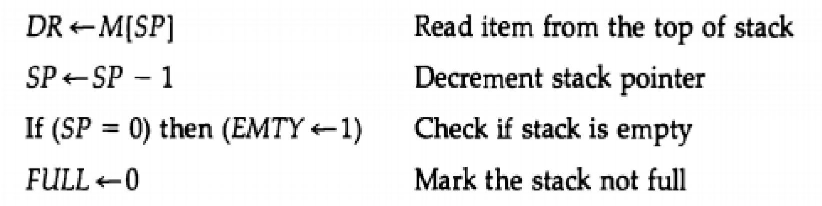


$SP \leftarrow SP + 1$	Increment stack pointer
$M[SP] \leftarrow DR$	Write item on top of the stack
If $(SP = 0)$ then $(FULL \leftarrow 1)$	Check if stack is full
$EMTY \leftarrow 0$	Mark the stack not empty

- ➢ The stack pointer is incremented so that it points to the address of next-higher word.
- ➢ A memory write operation inserts the word from DR the top of the stack.
- ➢ The first item stored in the stack is at address 1.
- ➢ The last item is stored at address 0.
- ➢ If SP reaches 0, the stack is full of items, so FULL is to 1.
- ➢ This condition is reached if the top item prior to the last push way 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, so the EMTY is cleared to 0.

POP:

- ➢ A new item is deleted from the stack if the stack is not empty (if EMTY = 0).
- ➢ The pop operation consists of the following sequence of min operations:

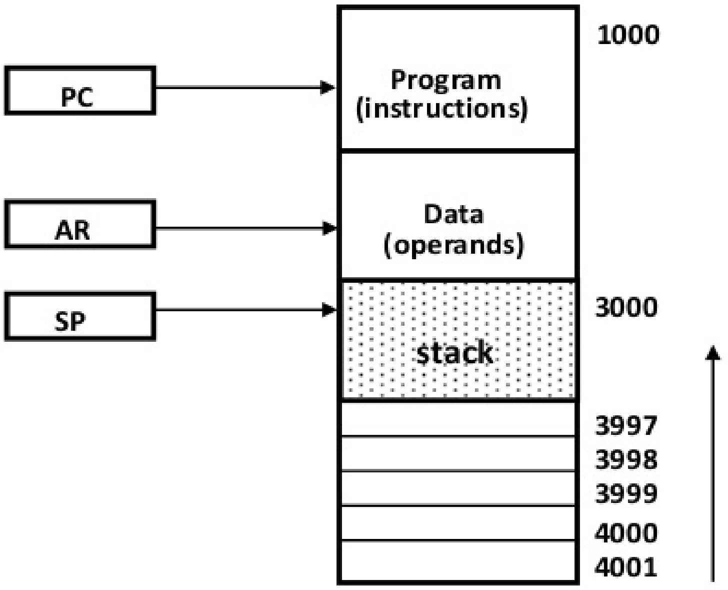


$DR \leftarrow M[SP]$	Read item from the top of stack
$SP \leftarrow SP - 1$	Decrement stack pointer
If $(SP = 0)$ then $(EMTY \leftarrow 1)$	Check if stack is empty
$FULL \leftarrow 0$	Mark the stack not full

- ➢ 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 EMTY is set 1.
- ➢ This condition is reached if the item read was in location 1. Once this it is read out, SP is decremented and reaches the value 0, which is the initial value of SP.
- ➢ If a pop operation reads the item from location 0 and then is decremented, SP changes to 111111, which is equivalent to decimal 63 in above configuration, the word in address 0 receives the last item in the stack.

Memory Stack:

- ➢ In the above discussion a stack can exist as a stand-alone unit.
- ➢ But in the CPU implementation of a stack is done by assigning a portion of memory to a stack operation and using a processor register as stack pointer.
- ➢ The below figure shows a portion computer memory partitioned into three segments: program, data, and stack.



- ➢ The program counter PC points at the address of the next instruction in program.
- ➢ The address register AR points at an array of data.
- ➢ 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.
  - ▪ PC is used during the fetch phase to read an instruction.
  - ▪ AR is used during the exec phase to read an operand.
  - ▪ SP is used to push or pop items into or from stack.
- ➢ As shown in Fig. 8-4, 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 limit checks.
- ➢ The items in the stack communicate with a data register DR. A new item is inserted with the push operation as follows:

SP
SP-1 M [SP]
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 stack. A new item is deleted with a pop operation as follows:

DR
M [SP] SP
SP+1

- The top item is read from the stack into DR. The stack pointer is then decremented 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 processor registers:
  - One to hold the upper limit (3000 in this case)
  - Other to hold the lower limit (4001 in this case).
- After a push operation, SP compared with the upper-limit register and after a pop operation, SP is a compared with the lower-limit register.
- The two microoperations needed for either the push or pop are
  - An access to memory through SP
  - Updating SP.
- The advantage of a memory stack is that the CPU can refer to it without having specify an address, since the address is always available and automatically updated in the stack pointer.

Reverse Polish Notation:

- A stack organization is very effective for evaluating arithmetic expressions.
- The common arithmetic expressions are written in infix notation, with each operator written between the operands.
- Consider the simple arithmetic expression. A*B+C*D
- For evaluating the above expression it is necessary to compute the product A*B, store this product result while computing C*D, and then sum the two products.
- For doing this type of infix notation, it is necessary to scan back and forth along the expression to determine the next operation to be performed.
- The Postfix notation, referred to as reverse Polish notation (RPN), places the operator after the operands.
- The following examples demonstrate the three representations

Eg: A+B ----->	Infix notation
+AB ------>	Prefix or Polish notation
AB+ ------->	Post or reverse Polish notation

- The reverse Polish notation is in a form suitable for stack manipulation.
- The expression A*B+C*D Is written in reverse polish notation as AB* CD* + And it is evaluated as follows
- Scan the expression from left to right. When operator is reached, perform the operation with the two operands found on the left side of the operator.
- Remove the two operands and the operator and replace them by the number obtained from the result of the operation.
- Continue to scan the expression and repeat the procedure for every operation encountered until there are no more operators.

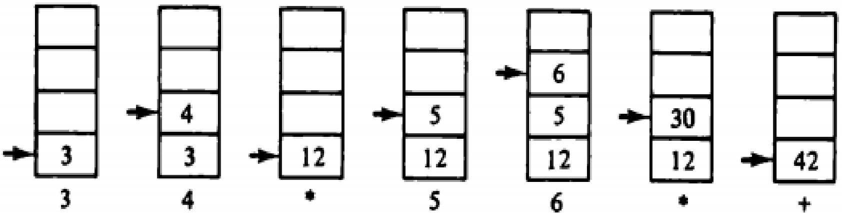
- For the expression above it find the operator * after A and B. So it perform the operation A*B and replace A, B and * with the result.
- The next operator is a * and it previous two operands are C and D, so it perform the operation C*D and places the result in places C, D and *.
- The next operator is + and the two operands to be added are the two products, so we add the two quantities to obtain the result.
- The conversion from infix notation to reverse Polish notation must take into consideration the operational hierarchy adopted for infix notation.
- This hierarchy dictates that we first perform all arithmetic inside inner parentheses, then inside outer parentheses, and do multiplication and division operations before addition and subtraction operations.

Evaluation of Arithmetic Expressions:

- Reverse Polish notation, combined with a stack arrangement of registers, is the most efficient way known for evaluating arithmetic expressions.
- This procedure is employed in some electronic calculators and also in some computer.
- 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 Figure.



- ➢ 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 top most items 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 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.

Computer Instruction Formats

- Computers may have instructions of several different lengths containing varying number of addresses.
- The number of address fields in the instruct 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.

Single Accumulator Organization:

- In an accumulator type organization all the operations are performed with an implied accumulator register.
- The instruction format in this type of computer uses one address field.
- For example, the instruction that specifies an arithmetic addition defined by an assembly language instruction as
- ADD X
- Where X is the address of the operand. The ADD instruction in this case results in the operation AC
- AC +M[X]. AC is the accumulator register and M[X] symbolizes the memory word located at address X.

General register organization:

- The instruction format in this type of computer needs three register address fields.
- Thus the instruction for an arithmetic addition may be written in an assembly language as ADD R1, R2, R3 to denote the operation R1
- R2 + R3. The number of address fields in the instruction can be reduced from three to two if the destination register is the same as one of the source registers.
- Thus the instruction ADD R1, R2 would denote the operation R1
- R1 + R2. Only register addresses for R1 and R2 need be specified in this instruction.
- General register-type computers employ two or three address fields in their instruction format.
- Each address field may specify a processor register or a memory word.
- An instruction symbolized by ADD R1, X would specify the operation R1
- R1 + M[X].
- It has two address fields, one for register R1 and the other for the memory address X.

Stack organization:

- The stack-organized CPU has PUSH and POP instructions which require an address field.
- Thus the instruction PUSH X will push the word at address X to the top of the stack.
- The stack pointer is updated automatically.

- Operation-type instructions do not need an address field in stack-organized computers.
- This is because the operation is performed on the two items that are on top of the stack.
- The instruction ADD in a stack computer consists of an operation code only with no address field.
- This operation has the effect of popping the two top numbers from the stack, adding the numbers, and pushing the sum into the stack.
- There is no need to specify operands with an address field since all operands are implied to be in the stack.
- Most computers fall into one of the three types of organizations.
- Some computers combine features from more than one organizational structure.
- The influence of the number of addresses on computer programs, we will evaluate the arithmetic statement X= (A+B) * (C+D)
- Using zero, one, two, or three address instructions and using the symbols
    - ADD, SUB, MUL and DIV for four arithmetic operations;
    - MOV for the transfer type operations; and
    - LOAD and STORE for transfer to and from memory and AC register.
- Assuming that the operands are in memory addresses A, B, C, and D and the result must be stored in memory ar address X and also the CPU has general purpose registers R1, R2, R3 and R4.

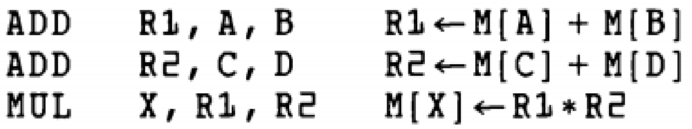
Instruction Formats:

Three Address Instructions:

- Three-address instruction formats can use each address field to specify either a processor register or a memory operand.
- The program assembly language that evaluates

X = (A+B) * (C+D)

- is shown below, together with comments that explain the register transfer operation of each instruction

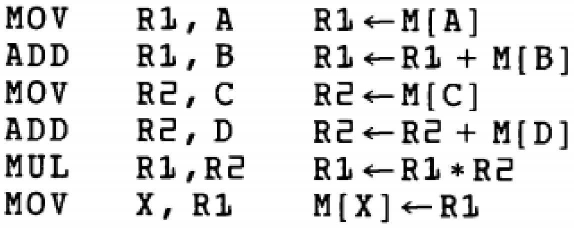


```
ADD    R1, A, B     R1←M[A] + M[B]
ADD    R2, C, D     R2←M[C] + M[D]
MUL    X, R1, R2    M[X]←R1*R2
```

- The symbol M [A] denotes the operand at memory address symbolized by A.
- The advantage of the three-address format is that it results in short programs when evaluating arithmetic expressions.
- The disadvantage is that the binary-coded instructions require too many bits to specify three addresses.

Two Address Instructions:

- Two-address instructions formats use each address field can specify either a processor register or memory word.
- The program to evaluate X = (A+B) * (C+D) is as follows

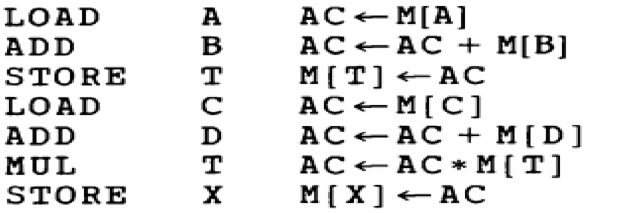


```
MOV    R1, A     R1←M[A]
ADD    R1, B     R1←R1 + M[B]
MOV    R2, C     R2←M[C]
ADD    R2, D     R2←R2 + M[D]
MUL    R1,R2     R1←R1*R2
MOV    X, R1     M[X]←R1
```

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

One Address Instructions:

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

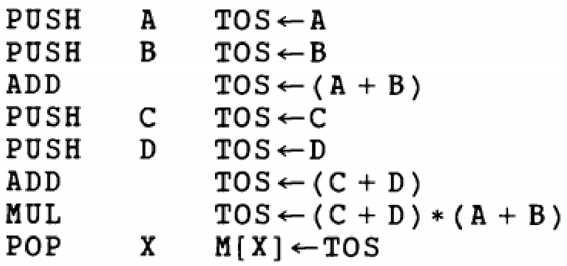


```
LOAD     A    AC←M[A]
ADD      B    AC←AC + M[B]
STORE    T    M[T]←AC
LOAD     C    AC←M[C]
ADD      D    AC←AC + M[D]
MUL      T    AC←AC*M[T]
STORE    X    M[X]←AC
```

- ➢ 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) * (C+D) will be written for a stack-organized computer. (TOS stands for top of stack).

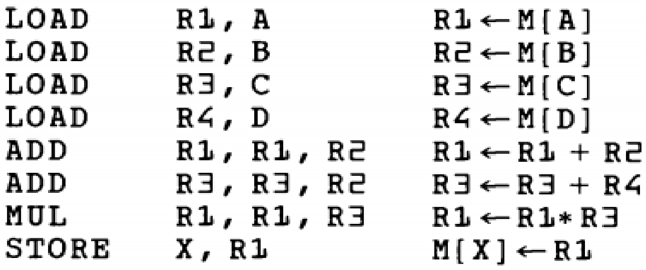


```
PUSH    A    TOS←A
PUSH    B    TOS←B
ADD          TOS←(A + B)
PUSH    C    TOS←C
PUSH    D    TOS←D
ADD          TOS←(C + D)
MUL          TOS←(C + D)*(A + B)
POP     X    M[X]←TOS
```

- 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.

RISC Instructions:

- The instruction set of a typical RISC processor is used to only load and store instructions for communicating between memory and CPU.
- All other instructions are executed within the registers of CPU without referring to memory.
- LOAD and STORE instructions that have one memory and one register address, and computational type instructions that have three addresses with all three specifying processor registers.
- The following is a program to evaluate X=(A+B)*(C+D)

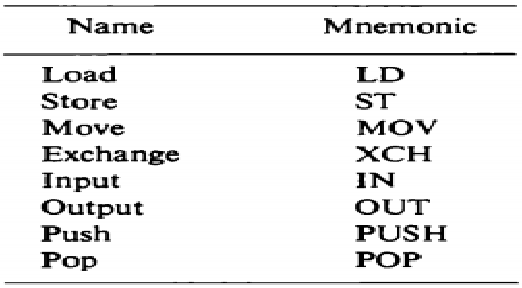


```
LOAD     R1, A          R1←M[A]
LOAD     R2, B          R2←M[B]
LOAD     R3, C          R3←M[C]
LOAD     R4, D          R4←M[D]
ADD      R1, R1, R2     R1←R1 + R2
ADD      R3, R3, R2     R3←R3 + R4
MUL      R1, R1, R3     R1←R1*R3
STORE    X, R1          M[X]←R1
```

Data Transfer and Manipulation

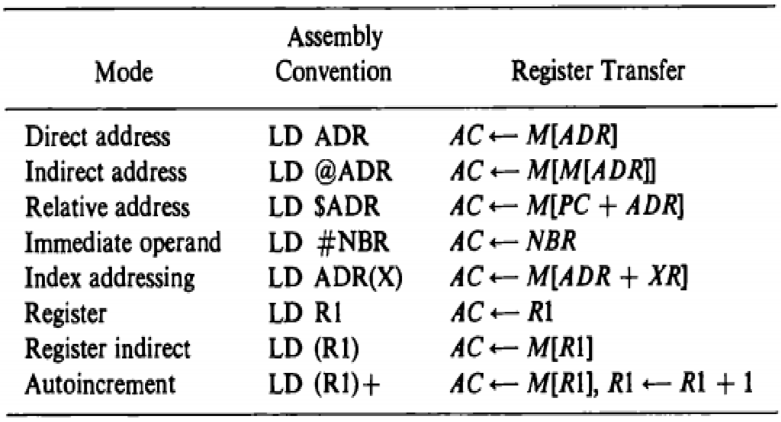
Data Transfer Instructions:

- Data transfer instructions move data from one place in the computer to another without changing the data content.
- The most common transfers are between memory and processor registers, between processor registers and input or output, and between the processor registers themselves.
- The following table gives a list of eight data transfer instructions used in many computers.



Name	Mnemonic
Load	LD
Store	ST
Move	MOV
Exchange	XCH
Input	IN
Output	OUT
Push	PUSH
Pop	POP

- The load instruction has been used mostly to designate a transfer from memory to a processor register, usually an accumulator.
- The store instruction designates a transfer from a processor register into memory.
- The move instruction has been used in computers with multiple CPU registers to designate a transfer from one register to another and also between CPU registers and memory or between two memory words.
- The exchange instruction swaps information between two registers or a register and a memory word.
- The input and output instructions transfer data among processor registers and input or output terminals.
- The push and pop instructions transfer data between processor registers and a memory stack.
- Different computers use different mnemonics symbols for differentiate the addressing modes.
- As an example, consider the load to accumulator instruction when used with eight different addressing modes.
- The following table shows the recommended assembly language convention and actual transfer accomplished in each case



Mode	Assembly Convention	Register Transfer
Direct address	LD ADR	$AC \leftarrow M[ADR]$
Indirect address	LD @ADR	$AC \leftarrow M[M[ADR]]$
Relative address	LD $ADR	$AC \leftarrow M[PC + ADR]$
Immediate operand	LD #NBR	$AC \leftarrow NBR$
Index addressing	LD ADR(X)	$AC \leftarrow M[ADR + XR]$
Register	LD R1	$AC \leftarrow R1$
Register indirect	LD (R1)	$AC \leftarrow M[R1]$
Autoincrement	LD (R1)+	$AC \leftarrow M[R1],\ R1 \leftarrow R1 + 1$

- ADR stands for an address.
- NBA a number or operand.
- X is an index register.
- R1 is a processor register.

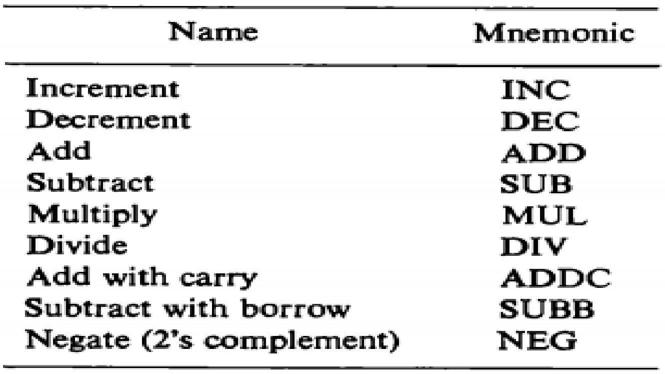
- AC is the accumulator register.
- The @ character symbolizes an indirect addressing.
- The $ character before an address makes the address relative to the program counter PC.
- The # character precedes the operand in an immediate-mode instruction.
- An indexed mode instruction is recognized by a register that placed in parentheses after the symbolic address.
- The register mode is symbolized by giving the name of a processor register.
- In the register indirect mode, the name of the register that holds the memory address is enclosed in parentheses.
- The auto-increment mode is distinguished from the register indirect mode by placing a plus after the parenthesized register.
- The auto-decrement mode would use a minus instead.

Data Manipulation Instructions:

- Data manipulation instructions perform operations on data and provide the computational capabilities for the computer.
- The data manipulation instructions in a typical computer are usually divided into three basic types:
    1. Arithmetic instructions
    2. Logical and bit manipulation instructions
    3. Shift instructions

1. **Arithmetic instructions**

- ➢ The four basic arithmetic operations are addition, subtraction, multiplication and division.
- ➢ Most computers provide instructions for all four operations.
- ➢ Some small computers have only addition and possibly subtraction instructions. The multiplication and division must then be generated by mean software subroutines.
- ➢ A list of typical arithmetic instructions is given in Table.



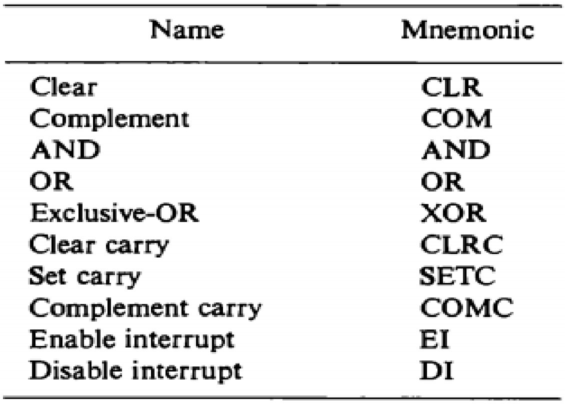
Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with carry	ADDC
Subtract with borrow	SUBB
Negate (2's complement)	NEG

- The increment instruction adds 1 to the value stored in a register or memory word.
- A number with all 1's, when incremented, produces a number with all 0's.
- The decrement instruction subtracts 1 from a value stored in a register or memory word.

- A number with all 0's, when decremented, produces number with all 1's.
- The add, subtract, multiply, and divide instructions may be use different types of data.
- The data type assumed to be in processor register during the execution of these arithmetic operations is defined by an operation code.
- An arithmetic instruction may specify fixed-point or floating-point data, binary or decimal data, single-precision or double-precision data.
- The mnemonics for three add instructions that specify different data types are shown below.
    - ADDI - Add two binary integer numbers
    - ADDF - Add two floating-point numbers
    - ADDD - Add two decimal numbers in BCD
- A special carry flip-flop is used to store the carry from an operation.
- The instruction "add carry" performs the addition on two operands plus the value of the carry the previous computation.
- Similarly, the "subtract with borrow" instruction subtracts two words and borrow which may have resulted from a previous subtract operation.
- The negate instruction forms the 2's complement number, effectively reversing the sign of an integer when represented it signed-2's complement form.

2. Logical and bit manipulation instructions

- Logical instructions perform binary operations on strings of bits store, registers.
- They are useful for manipulating individual bits or a group of that represent binary-coded information.
- The logical instructions consider each bit of the operand separately and treat it as a Boolean variable.
- By proper application of the logical instructions it is possible to change bit values, to clear a group of bits, or to insert new bit values into operands stored in register memory words.
- Some typical logical and bit manipulation instructions are listed in Table



Name	Mnemonic
Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Set carry	SETC
Complement carry	COMC
Enable interrupt	EI
Disable interrupt	DI

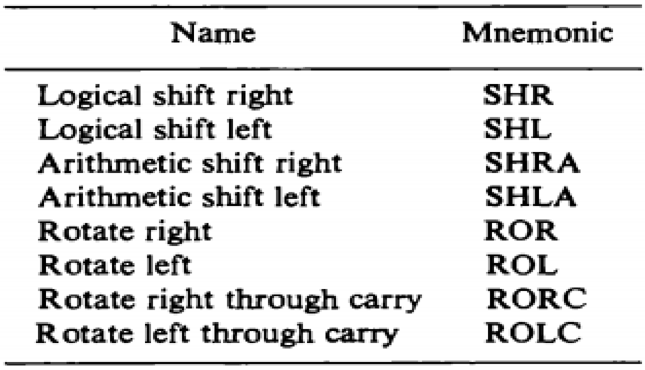
There are three bit manipulation operations possible:

1. a selected bit can cleared to 0,
2. or can be set to 1,

3. or can be complemented.

3. Shift Instructions:

- Shifts are operations in which the bits of a word are moved to the left or right.
- The bit shifted in at the end of the word determines the type of shift used.
- Shift instructions may specify logical shifts, arithmetic shifts, or rotate-type operations.
- In either case the shift may be to the right or to the left.
- The following table lists four types of shift instructions.



Name	Mnemonic
Logical shift right	SHR
Logical shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate right through carry	RORC
Rotate left through carry	ROLC

- The logical shift insert 0 to the end bit position.
- The end position is the leftmost bit position for shift rights the rightmost bit position for the shift left.
- Arithmetic shifts usually conform to the rules for signed-2's complement numbers.
- The arithmetic shift-right instruction must preserve the sign bit in the leftmost position.
- The sign bit is shifted to the right together with the rest of the number, but the sign bit itself remains unchanged.
- This is a shift-right operation with the end bit remaining the same.
- The arithmetic shift-left instruction inserts 0 to the end position and is identical to the logical shift-instruction.
- The rotate instructions produce a circular shift. Bits shifted out at one of the word are not lost as in a logical shift but are circulated back into the other end.
- The rotate through carry instruction treats a carry bit as an extension of the register whose word is being rotated.
- Thus a rotate-left through carry instruction transfers the carry bit into the rightmost bit position of the register, transfers the leftmost bit position into the carry, and at the same time, shift the entire register to the left.

6. Reduced Instruction Set Computer (RISC):

- A computer with large number instructions is classified as a complex instruction set computer, abbreviated as CISC.
- The computer which is having the fewer instructions is classified as a reduced instruction set computer, abbreviated as RISC.

RISC Characteristics:

- Relatively few instructions
- Relatively few addressing modes
- Memory access limited to load and store instructions
- All operations done within the registers of the CPU
- Fixed-length, easily decoded instruction format
- Single-cycle instruction execution
- Hardwired rather than microprogrammed control
- A relatively large number of registers in the processor unit
- Efficient instruction pipeline

CISC Characteristics:

- A large number of instructions--typically from 100 to 250 instructions.
- Some instructions that perform specialized tasks and are used infrequently.
- A large variety of addressing modes—typically from 5 to 20 differ modes.
- Variable-length instruction formats

Instructions that manipulate operands in memory

SREENIVASA INSTITUTE OF TECHNOLOGY AND MANAGEMENT STUDIES.
(AUTONOMOUS)
MCA DEPARTMENT

LECTURE NOTES

COMPUTER ORGANIZATION AND ARCHITECTURE

UNIT-4

Micro Programmed Control And Central Processing Unit

Introduction

- The function of the control unit in a digital computer is to initiate sequence of microoperations.
- Control unit can be implemented in two ways
    1. Hardwired control
    2. Microprogrammed control

Hardwired Control:

- When the control signals are generated by hardware using conventional logic design techniques, the control unit is said to be hardwired.
- The key characteristics are
    - High speed of operation
    - Expensive
    - Relatively complex
    - No flexibility of adding new instructions

Examples of CPU with hardwired control unit are

- Intel 8085,
- Motorola 6802,
- Zilog 80,
- and any RISC CPUs.

Microprogrammed Control:

- Control information is stored in control memory.
- Control memory is programmed to initiate the required sequence of micro-operations.
- The key characteristics are
    - Speed of operation is low when compared with hardwired
    - Less complex
    - Less expensive
    - Flexibility to add new instructions
- Examples of CPU with microprogrammed control unit are
    - Intel 8080,
    - Motorola 68000
    - and any CISC CPUs.

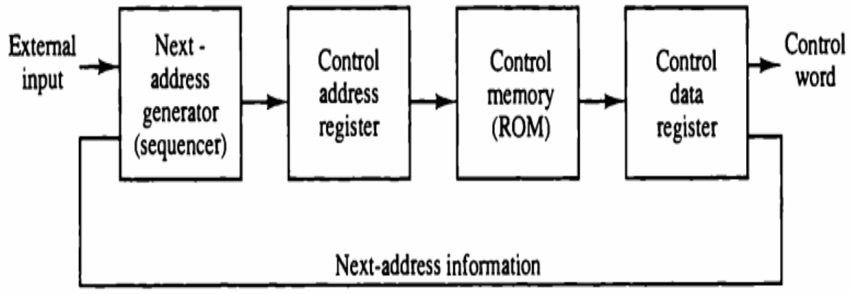
3. **Control Memory:**

- A control memory is a part of the control unit. Any computer that involves microprogrammed control consists of two memories. They are the main memory and the control memory.

- Programs are usually stored in the main memory by the users. Whenever the programs change, the data is also modified in the main memory. They consist of machine instructions and data.
- The control memory consists of microprograms that are fixed and cannot be modified frequently. They contain microinstructions that specify the internal control signals required to execute register micro-operations.
- The machine instructions generate a chain of microinstructions in the control memory. Their function is to generate micro-operations that can fetch instructions from the main memory, compute the effective address, execute the operation, and return control to fetch phase and continue the cycle.
- The control function that specifies a microoperation is called as **control variable.**
- When control variable is in one binary state, the corresponding microoperation is executed. For the other binary state the state of registers does not change.
- The active state of a control variable may be either 1 state or the 0 state, depending on the application.
- **Control Word:** The control variables at any given time can be represented by a string of 1's and 0's called a control word.
- All control words can be programmed to perform various operations on the components of the system.

Microprogram control unit: A control unit whose binary control variables are stored in memory is called a microprogram control unit.

- The control word in control memory contains a microinstruction.
- The microinstruction specifies one or more micro-operations for the system.
- A sequence of microinstructions constitutes a microprogram.
- The control unit consists of control memory used to store the microprogram.
- Control memory is a permanent i.e., read only memory (ROM).
- The general configuration of a micro-programmed control unit organization is shown as block diagram below.

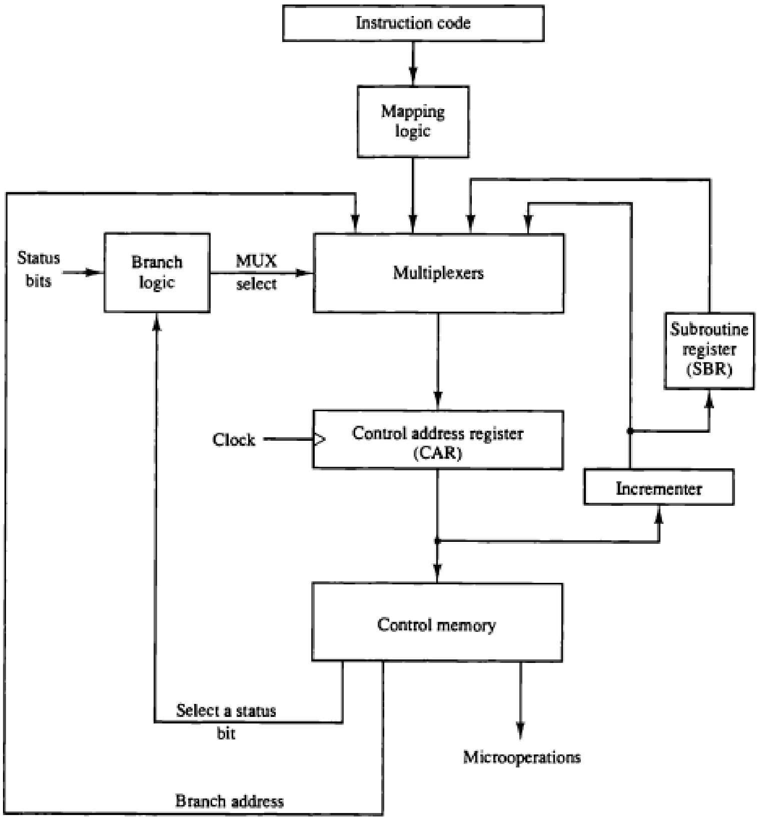


- ➢ The control memory is ROM so all control information is permanently stored.

- The control memory address register (CAR) specifies the address of the microinstruction and the control data register (CDR) holds the microinstruction read from memory.
- The next address generator is sometimes called a microprogram sequencer. It is used to generate the next micro instruction address.
- The location of the next microinstruction may be the one next in sequence or it may be located somewhere else in the control memory.
- So it is necessary to use some bits of the present microinstruction to control the generation of the address of the microinstruction.
- Sometimes the next address may also be a function of external input conditions.
- The control data register holds the present microinstruction while next address is computed and read from memory. The data register is times called a pipeline register.

4. Address Sequencing:

- Microinstructions are stored in control memory in groups, with each group specifying a routine.
- Each computer instruction has its own microprogram routine to generate the microoperations.
- The hardware that controls the address sequencing of the control memory must be capable of sequencing the microinstructions within a routine and be able to branch from one routine to another
- Control must undergo the following steps during the execution of a single computer instruction:
- Load an initial address into the CAR when power is turned on in the computer. This address is usually the address of the first microinstruction that activates the instruction fetch routine – IR holds instruction
- The control memory then goes through the routine to determine the effective address of the operand – AR holds operand address
- The next step is to generate the microoperations that execute the instruction by considering the opcode and applying a mapping process.
- The transformation of the instruction code bits to an address in control memory where the routine of instruction located is referred to as **mapping process.**
- After execution, control must return to the fetch routine by executing an unconditional branch.
- The below figure shows a block diagram of a control memory and the associated hardware needed for selecting the next microinstruction address.



- The microinstruction in control memory contains a set of bits to initiate microoperations in computer registers and other bits to specify the method by which the next address is obtained.
- In the figure four different paths form which the control address register (CAR) receives the address.
  - The incrementer increments the content of the control register address register by one, to select the next microinstruction in sequence.
  - Branching is achieved by specifying the branch address in one of the fields of the microinstruction.
  - Conditional branching is obtained by using part of the microinstruction to select a specific status bit in order to determine its condition.
  - An external address is transferred into control memory via a mapping logic circuit.
  - The return address for a subroutine is stored in a special register, that value is used when the micoprogram wishes to return from the subroutine.

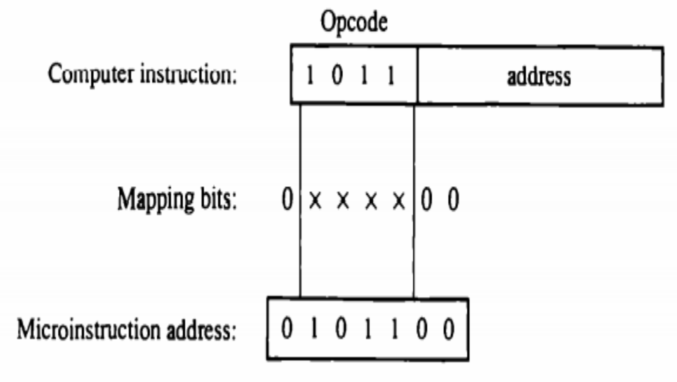
Conditional Branching:

- Conditional branching is obtained by using part of the microinstruction to select a specific status bit in order to determine its condition.

- The status conditions are special bits in the system that provide parameter information such as the carry-out of an adder, the sign bit of a number, the mode bits of an instruction, and i/o status conditions.
- The status bits, together with the field in the microinstruction that specifies a branch address, control the branch logic.
- The branch logic tests the condition, if met then branches, otherwise, increments the CAR.
- If there are 8 status bit conditions, then 3 bits in the microinstruction are used to specify the condition and provide the selection variables for the multiplexer.
- For unconditional branching, fix the value of one status bit to be one load the branch address from control memory into the CAR.

Mapping of Instruction:

- A special type of branch exists when a microinstruction specifies a branch to the first word in control memory where a microprogram routine is located.
- The status bits for this type of branch are the bits in the opcode.
- Assume an opcode of four bits and a control memory of 128 locations. The mapping process converts the 4-bit opcode to a 7-bit address for control memory shown in below figure



- Mapping consists of placing a 0 in the most significant bit of the address, transferring the four operation code bits, and clearing the two least significant bits of the control address register.
- This provides for each computer instruction a microprogram routine with a capacity of four microinstructions.

Subroutines:

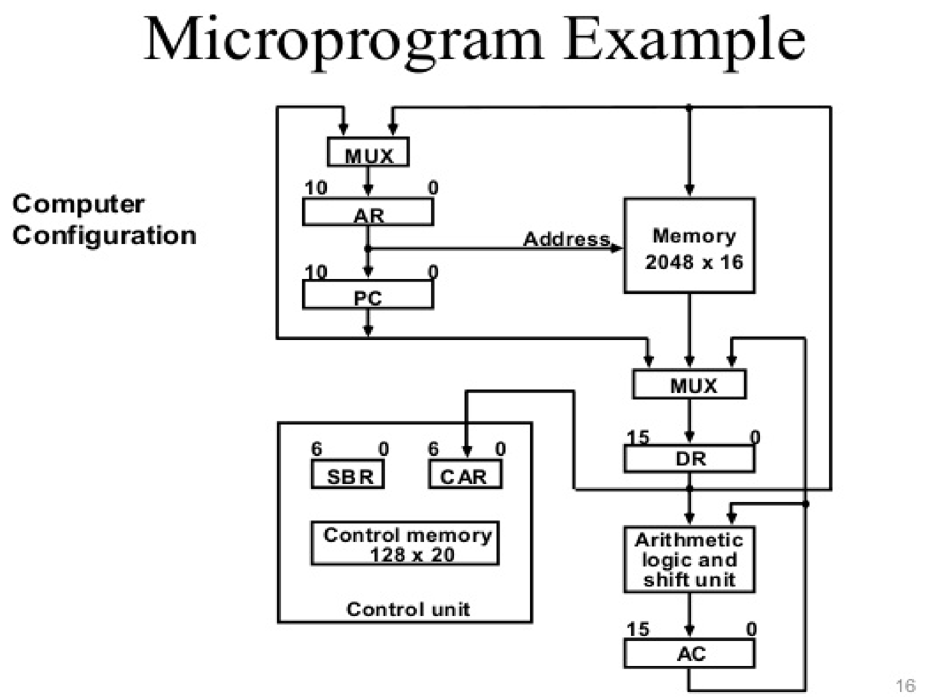
- Subroutines are programs that are used by other routines to accomplish a particular task and can be called from any point within the main body of the microprogram.
- Frequently many microprograms contain identical section of code.
- Microinstructions can be saved by employing subroutines that use common sections of microcode.
- Microprograms that use subroutines must have a provision for storing the return address during a subroutine call and restoring the address during a subroutine return.
- A subroutine register is used as the source and destination for the addresses.

3. Microprogram Example:

- The process of code generation for the control memory is called microprogramming.
- The block diagram of the computer configuration is shown in the figure.
- Two memory units:
  1. Main memory – stores instructions and data
  2. Control memory – stores microprogram
- Four processor registers
  1. Program counter – PC
  2. Address register – AR
  3. Data register – DR

Accumulator register – AC

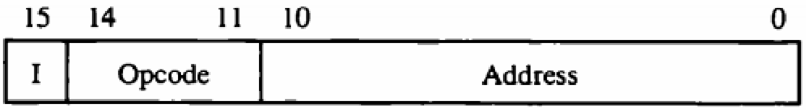
- Two control unit registers
  1. Control address register – CAR
  2. Subroutine register – SBR
- Transfer of information among registers in the processor is through MUXs rather than a bus.



Microprogram Example

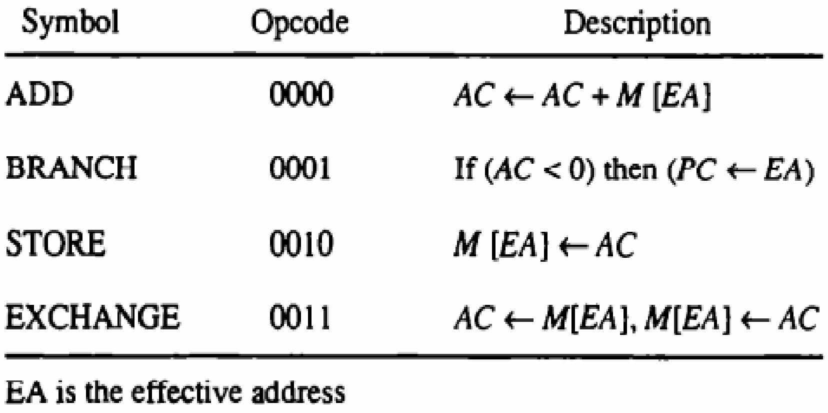
Computer Configuration

- The computer instruction format is shown in below figure.



(a) Instruction format

- Three fields for an instruction:
  1. 1-bit field for indirect addressing
  2. 4-bit opcode
  3. 11-bit address field
- The example will only consider the following 4 of the possible 16 memory instructions

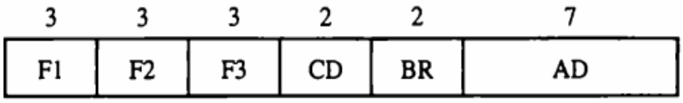


Symbol	Opcode	Description
ADD	0000	$AC \leftarrow AC + M[EA]$
BRANCH	0001	If $(AC < 0)$ then $(PC \leftarrow EA)$
STORE	0010	$M[EA] \leftarrow AC$
EXCHANGE	0011	$AC \leftarrow M[EA], M[EA] \leftarrow AC$

EA is the effective address

(b) Four computer instructions

The microinstruction format for the control memory is shown in below figure.



F1, F2, F3: Microoperation fields

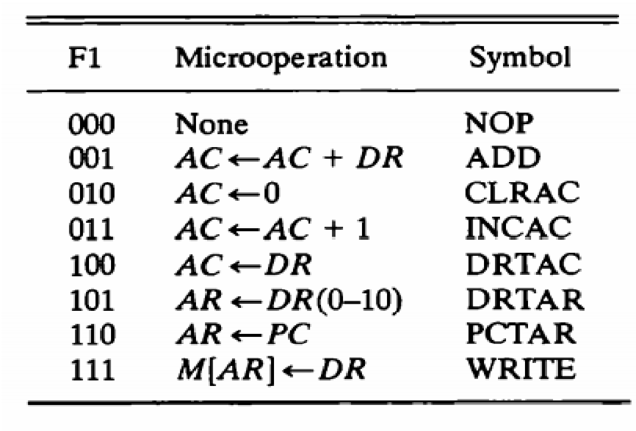
CD: Condition for branching

BR: Branch field

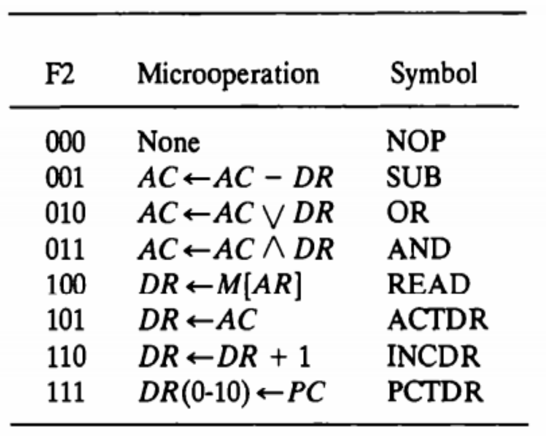
AD: Address field

- The microinstruction format is composed of 20 bits with four parts to it
- Three fields F1, F2, and F3 specify microoperations for the computer [3 bits each]
- The CD field selects status bit conditions [2 bits]
- The BR field specifies the type of branch to be used [2 bits]
- The AD field contains a branch address [7 bits]
- Each of the three microoperation fields can specify one of seven possibilities.
- No more than three microoperations can be chosen for a microinstruction.

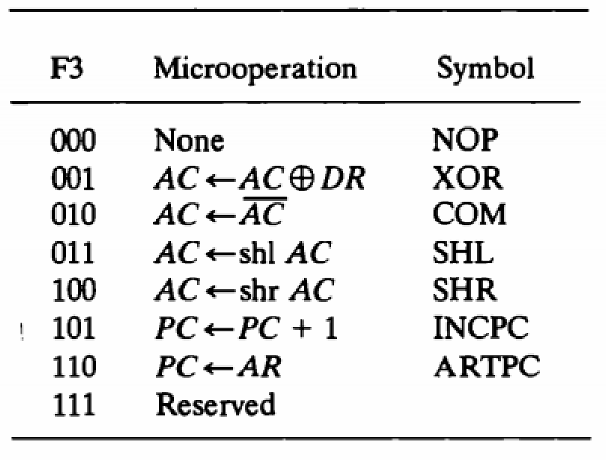
- If fewer than three are needed, the code 000 = NOP.
- The three bits in each field are encoded to specify seven distinct microoperations listed in below table



F1	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC + DR$	ADD
010	$AC \leftarrow 0$	CLRAC
011	$AC \leftarrow AC + 1$	INCAC
100	$AC \leftarrow DR$	DRTAC
101	$AR \leftarrow DR(0\text{–}10)$	DRTAR
110	$AR \leftarrow PC$	PCTAR
111	$M[AR] \leftarrow DR$	WRITE



F2	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC - DR$	SUB
010	$AC \leftarrow AC \vee DR$	OR
011	$AC \leftarrow AC \wedge DR$	AND
100	$DR \leftarrow M[AR]$	READ
101	$DR \leftarrow AC$	ACTDR
110	$DR \leftarrow DR + 1$	INCDR
111	$DR(0\text{-}10) \leftarrow PC$	PCTDR

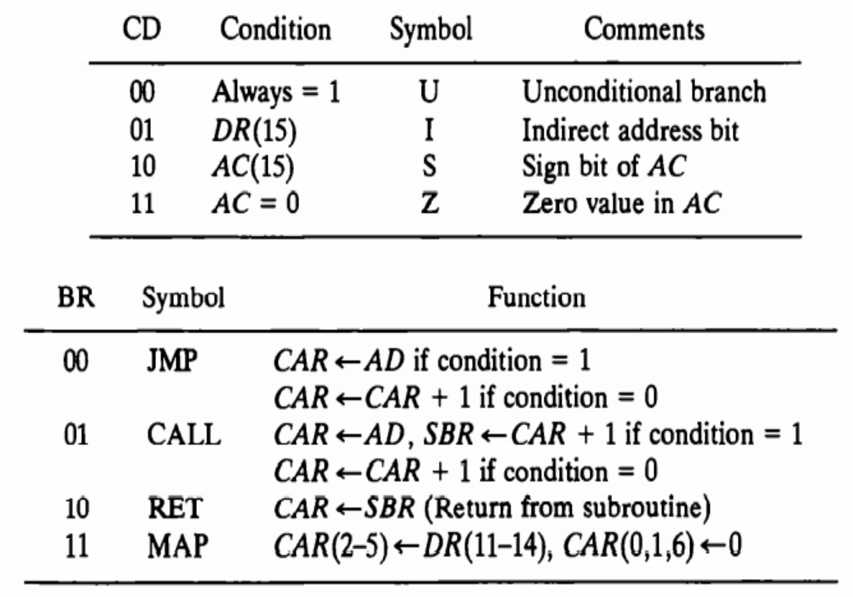


F3	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC \oplus DR$	XOR
010	$AC \leftarrow \overline{AC}$	COM
011	$AC \leftarrow \text{shl } AC$	SHL
100	$AC \leftarrow \text{shr } AC$	SHR
101	$PC \leftarrow PC + 1$	INCPC
110	$PC \leftarrow AR$	ARTPC
111	Reserved	

- Five letters to specify a transfer-type microoperation
- First two designate the source register
- Third is a ‘T’
- Last two designate the destination register AC ← DR

F1 = 100 = DRTAC

- The condition field (CD) is two bits to specify four status bit conditions shown below
- The branch field (BR) consists of two bits and is used with the address field to choose the address of the next microinstruction

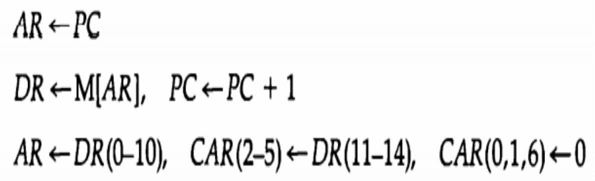


CD	Condition	Symbol	Comments
00	Always = 1	U	Unconditional branch
01	$DR(15)$	I	Indirect address bit
10	$AC(15)$	S	Sign bit of AC
11	$AC = 0$	Z	Zero value in AC

BR	Symbol	Function
00	JMP	$CAR \leftarrow AD$ if condition = 1 $CAR \leftarrow CAR + 1$ if condition = 0
01	CALL	$CAR \leftarrow AD$, $SBR \leftarrow CAR + 1$ if condition = 1 $CAR \leftarrow CAR + 1$ if condition = 0
10	RET	$CAR \leftarrow SBR$ (Return from subroutine)
11	MAP	$CAR(2\text{-}5) \leftarrow DR(11\text{-}14)$, $CAR(0,1,6) \leftarrow 0$

Fetch Routine:

- The control memory has 128 locations, each one is 20 bits.
- The first 64 locations are occupied by the routines for the 16 instructions, addresses 0-63.
- Can start the fetch routine at address 64.
- The fetch routine requires the following three microinstructions (locations 64-66).
- The microinstructions needed for fetch routine are:

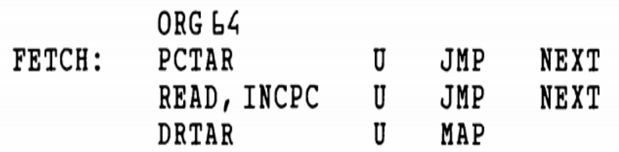


$AR \leftarrow PC$

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

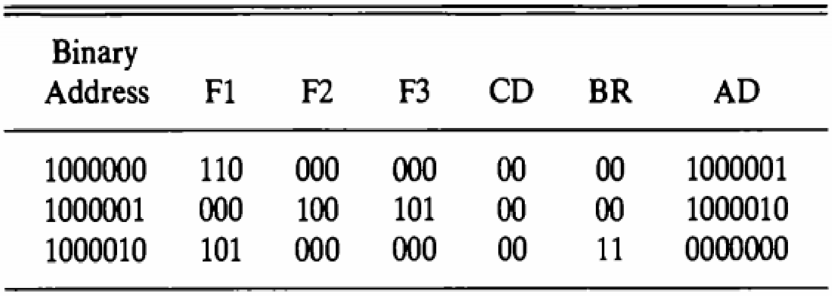
$AR \leftarrow DR(0\text{-}10), \quad CAR(2\text{-}5) \leftarrow DR(11\text{-}14), \quad CAR(0,1,6) \leftarrow 0$

- It's Symbolic microprogram:



```
         ORG 64
FETCH:   PCTAR         U   JMP   NEXT
         READ, INCPC   U   JMP   NEXT
         DRTAR         U   MAP
```

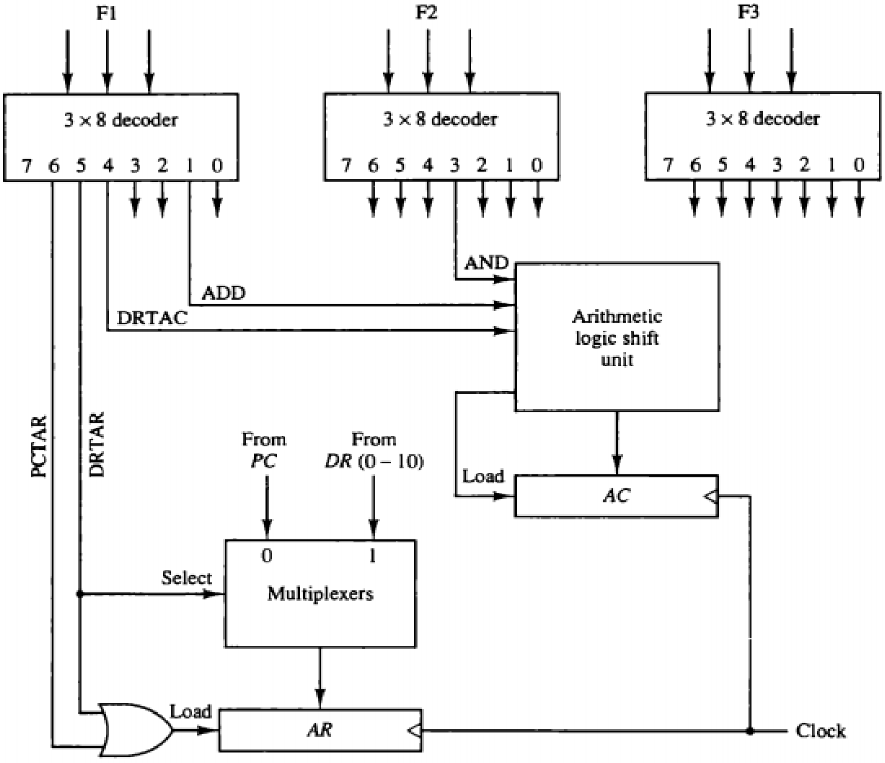
- It's Binary microprogram:



Binary Address	F1	F2	F3	CD	BR	AD
1000000	110	000	000	00	00	1000001
1000001	000	100	101	00	00	1000010
1000010	101	000	000	00	11	0000000

4. **Design of control Unit:**

- The control memory out of each subfield must be decoded to provide the distinct microoperations.
- The outputs of the decoders are connected to the appropriate inputs in the processor unit.
- The below figure shows the three decoders and some of the connections that must be made from their outputs.
- The three fields of the microinstruction in the output of control memory are decoded with a 3x8 decoder to provide eight outputs.
- Each of the output must be connected to proper circuit to initiate the corresponding microoperation as specified in previous topic.

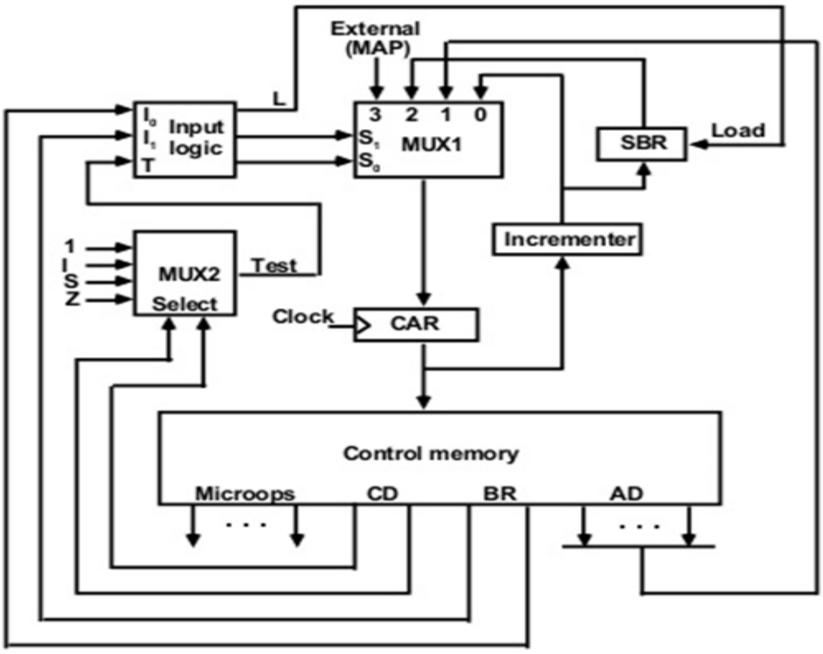


- ➢ When F1 = 101 (binary 5), the next pulse transition transfers the content of DR (0-10) to AR.
- ➢ Similarly, when F1= 110 (binary 6) there is a transfer from PC to AR (symbolized by PCTAR). As shown in Fig, outputs 5 and 6 of decoder F1 are connected to the load input of AR so that when either one of these outputs is active, information from the multiplexers is transferred to AR.
- ➢ The multiplexers select the information from DR when output 5 is active and from PC when output 5 is inactive.
- ➢ The transfer into AR occurs with a clock transition only when output 5 or output 6 of the decoder is active.

- For the arithmetic logic shift unit the control signals are instead of coming from the logical gates, now these inputs will now come from the outputs of AND, ADD and DRTAC respectively

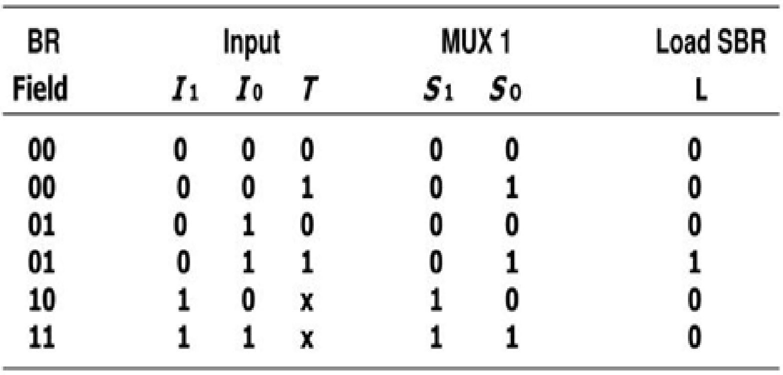
Microprogram Sequencer:

- The basic components of a microprogrammed control unit are the control memory and the circuits that select the next address.
- The address selection part is called a microprogram sequencer.
- The purpose of a microprogram sequencer is to present an address to the control memory so that a microinstruction may be read and executed.
- The next-address logic of the sequencer determines the specific address source to be loaded into the control address register.
- The block diagram of the microprogram sequencer is shown in below figure.



- The control memory is included in the diagram to show the interaction between the sequencer and the memory attached to it.
- There are two multiplexers in the circuit.
  - ❑ The first multiplexer selects an address from one of four sources and routes it into control address register CAR.
  - ❑ The second multiplexer tests the value of a selected status bit and the result of the test is applied to an input logic circuit.
- The output from CAR provides the address for the control memory.

- The content of CAR is incremented and applied to one of the multiplexer inputs and to the subroutine register SBR.
- The other three inputs to multiplexer come from
  - ❑ The address field of the present microinstruction
  - ❑ From the out of SBR
  - ❑ From an external source that maps the instruction
- The CD (condition) field of the microinstruction selects one of the status bits in the second multiplexer.
- If the bit selected is equal to 1, the T variable is equal to 1; otherwise, it is equal to 0.
- The T value together with two bits from the BR (branch) field goes to an input logic circuit.
- The input logic in a particular sequencer will determine the type of operations that are available in the unit.
- The input logic circuit in above figure has three inputs I0, I1, and T, and three outputs, S0, S1, and L.
- Variables S0 and S1 select one of the source addresses for CAR. Variable L enables the load input in SBR.
- The binary values of the selection variables determine the path in the multiplexer.
- For example, with S1,S0 = 10, multiplexer input number 2 is selected and establishes transfer path from SBR to CAR.
- The truth table for the input logic circuit is shown in Table below.



BR	Input			MUX 1		Load SBR
Field	I_1	I_0	T	S_1	S_0	L
00	0	0	0	0	0	0
00	0	0	1	0	1	0
01	0	1	0	0	0	0
01	0	1	1	0	1	1
10	1	0	x	1	0	0
11	1	1	x	1	1	0

- Inputs I1 and I0 are identical to the bit values in the BR field.
- The bit values for S1 and S0 are determined from the stated function and the path in the multiplexer that establishes the required transfer.
- The subroutine register is loaded with the incremented value of CAR during a call microinstruction (BR = 01) provided that the status bit condition is satisfied (T = 1).
- The truth table can be used to obtain the simplified Boolean functions for the input logic circuit:

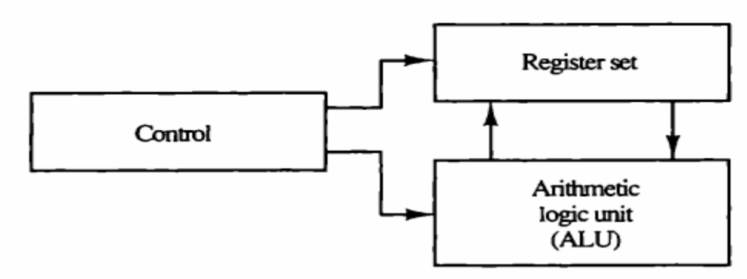
S1=I1
S0=I1I0+I’ 1T

L = I ' 1T I0

Central Processing Unit

Introduction:

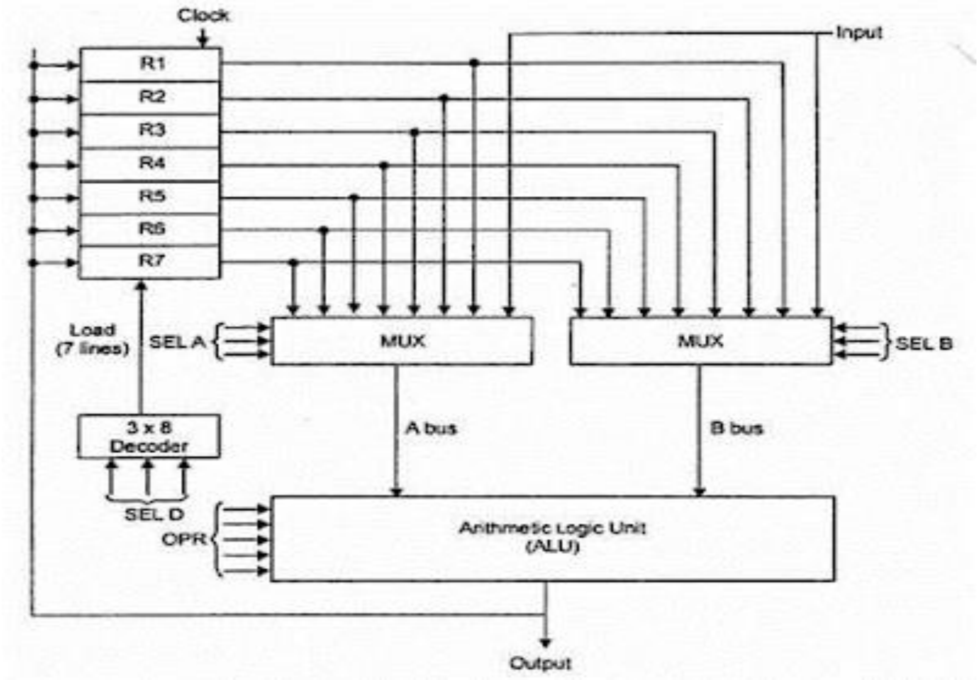
- The main part of the computer that performs the bulk of data-processing operations is called the central processing unit and is referred to as the CPU.
- The CPU is made up of three major parts, as shown in Figure



- 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

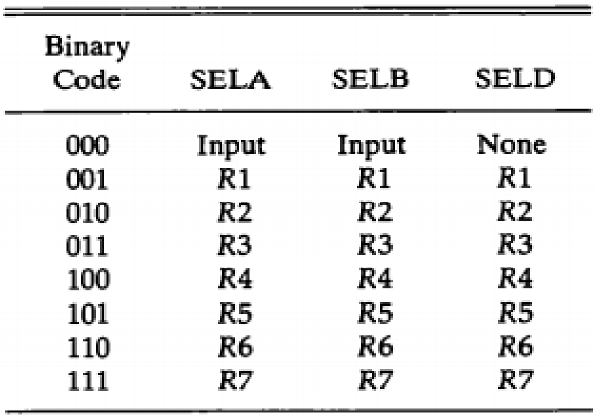
- Generally CPU has seven general registers.
- Register organization show how registers are selected and how data flow between register and ALU.
- A decoder is used to select a particular register.
- The output of each register is connected to two multiplexers to form the two buses A and B.
- The selection lines in each multiplexer select the input data for the particular bus.
- The A and B buses form the two inputs of an ALU.
- The operation select lines decide the micro operation to be performed by ALU.
- The result of the micro operation is available at the output bus.
- The output bus connected to the inputs of all registers, thus by selecting a destination register it is possible to store the result in it.



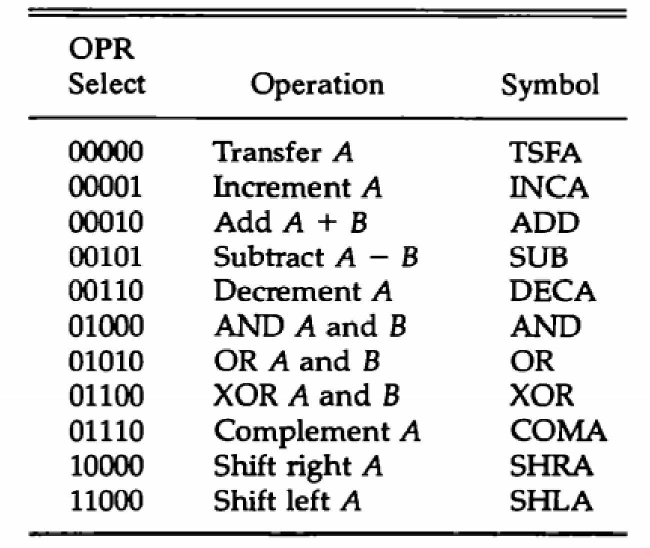
EXAMPLE:

To perform the operation **R3 = R1+R2** We have to provide following binary selection variable to the select inputs.

1. **SEL A** : **001** -To place the contents of R1 into bus A.
2. **SEL B** : **010** - to place the contents of R2 into bus B
3. **SEL OPR** : **10010** – to perform the arithmetic addition A+B
4. **SEL REG or SEL D : 011** – to place the result available on output bus in R3.



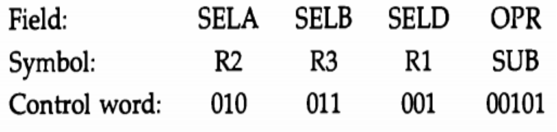
Binary Code	SELA	SELB	SELD
000	Input	Input	None
001	R1	R1	R1
010	R2	R2	R2
011	R3	R3	R3
100	R4	R4	R4
101	R5	R5	R5
110	R6	R6	R6
111	R7	R7	R7



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

CONTROL WORD?

- The combined value of a binary selection inputs specifies the control word.
- It consist of four fields SELA, SELB,and SELD or SELREG contains three bit each and SELOPR field contains four bits thus the total bits in the control word are 13-bits.
- Control Word For Operation R2 = R1+r3



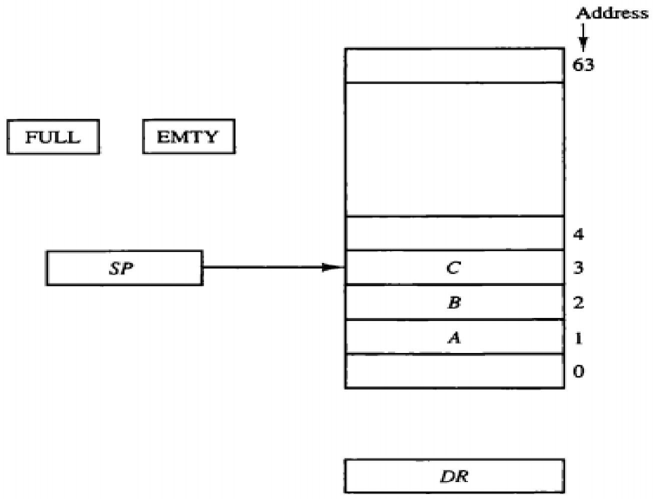
Field:	SELA	SELB	SELD	OPR
Symbol:	R2	R3	R1	SUB
Control word:	010	011	001	00101

2. Stack Organization:
    - **Stack**: A stack is a storage device that stores information in such a manner that the item stored last is the first item retrieved.
    - In the computer stack is a memory unit with an address register that can count the address only.
    - The register that holds the address for the stack is called a stack pointer (SP). It always points at the top item in the stack

- The two operations that are performed on stack are the insertion and deletion.
- The operation of insertion is called PUSH.
- The operation of deletion is called POP.
- These operations are simulated by incrementing and decrementing the stack pointer register (SP)

Register Stack:

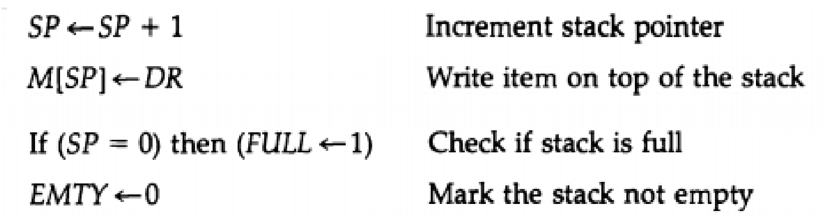
- A stack can be placed in a portion of a large memory or it can be organized as a collection of a finite number of memory words or registers.
- The below figure shows the organization of a 64-word register stack



- ➢ The stack pointer register SP contains a binary number whose value is equal to the address of the word is currently on top of the stack. Three items are placed in the stack: A, B, C, in that order.
- ➢ In above figure C is on top of the stack so that the content of SP is 3.
- ➢ For removing the top item, the stack is popped by reading the memory word at address 3 and decrementing the content of stack SP.
- ➢ Now the top of the stack is B, so that the content of SP is 2.
- ➢ Similarly for inserting the new item, the stack is pushed by incrementing SP and writing a word in the next higher location in the stack.
- ➢ In a 64-word stack, the stack pointer contains 6 bits because 26 = 64.
- ➢ Since SP has only six bits, it cannot exceed a number greater than 63 (111111 in binary).
- ➢ When 63 is incremented by 1, the result is 0 since 111111 + 1. = 1000000 in binary, but SP can accommodate only the six least significant bits.
- ➢ Then the one-bit register FULL is set to 1, when the stack is full.
- ➢ Similarly when 000000 is decremented by 1, the result is 111111, and then the one-bit register EMTY is set 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

PUSH:

- ➢ Initially, SP is cleared to 0, EMTY 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.
- ➢ The push operation is implemented with the following sequence of microoperations

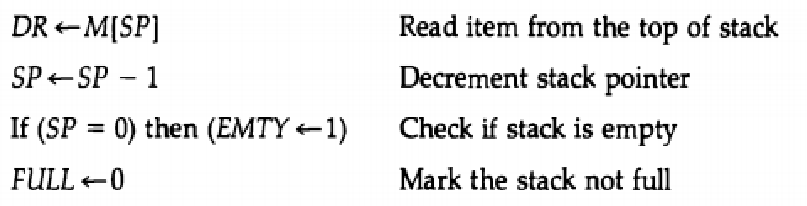


$SP \leftarrow SP + 1$	Increment stack pointer
$M[SP] \leftarrow DR$	Write item on top of the stack
If $(SP = 0)$ then $(FULL \leftarrow 1)$	Check if stack is full
$EMTY \leftarrow 0$	Mark the stack not empty

- The stack pointer is incremented so that it points to the address of next-higher word.
- A memory write operation inserts the word from DR the top of the stack.
- The first item stored in the stack is at address 1.
- The last item is stored at address 0.
- If SP reaches 0, the stack is full of items, so FULL is to 1.
- This condition is reached if the top item prior to the last push way 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, so the EMTY is cleared to 0.

POP:

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

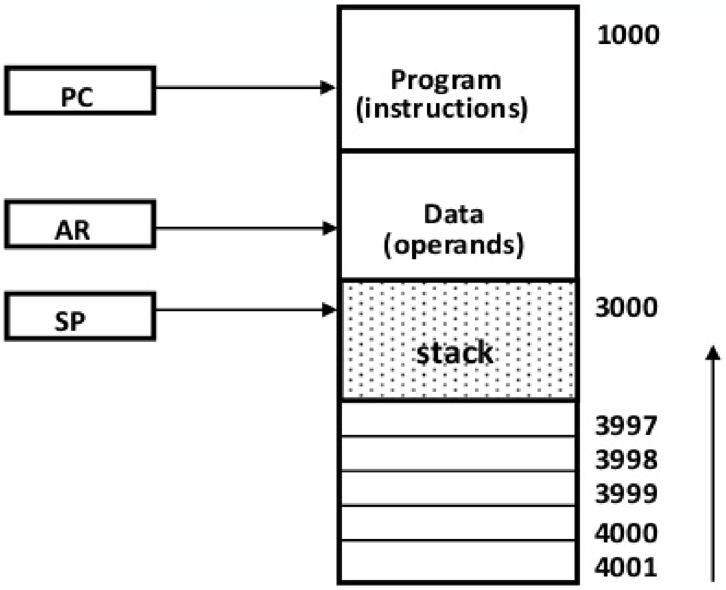


$DR \leftarrow M[SP]$	Read item from the top of stack
$SP \leftarrow SP - 1$	Decrement stack pointer
If $(SP = 0)$ then $(EMTY \leftarrow 1)$	Check if stack is empty
$FULL \leftarrow 0$	Mark the stack not full

- 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 EMTY is set 1.
- This condition is reached if the item read was in location 1. Once this it is read out, SP is decremented and reaches the value 0, which is the initial value of SP.
- If a pop operation reads the item from location 0 and then is decremented, SP changes to 111111, which is equivalent to decimal 63 in above configuration, the word in address 0 receives the last item in the stack.

Memory Stack:

- In the above discussion a stack can exist as a stand-alone unit.
- But in the CPU implementation of a stack is done by assigning a portion of memory to a stack operation and using a processor register as stack pointer.
- The below figure shows a portion computer memory partitioned into three segments: program, data, and stack.



- The program counter PC points at the address of the next instruction in program.
- The address register AR points at an array of data.
- 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.
  - PC is used during the fetch phase to read an instruction.
  - AR is used during the exec phase to read an operand.
  - SP is used to push or pop items into or from stack.
- As shown in Fig. 8-4, 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 limit checks.
- The items in the stack communicate with a data register DR. A new item is inserted with the push operation as follows:

SP
SP-1 M [SP]
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 stack. A new item is deleted with a pop operation as follows:

DR
M [SP] SP
SP+1

- The top item is read from the stack into DR. The stack pointer is then decremented 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 processor registers:
    - One to hold the upper limit (3000 in this case)
    - Other to hold the lower limit (4001 in this case).
- After a push operation, SP compared with the upper-limit register and after a pop operation, SP is a compared with the lower-limit register.
- The two microoperations needed for either the push or pop are
    - An access to memory through SP
    - Updating SP.
- The advantage of a memory stack is that the CPU can refer to it without having specify an address, since the address is always available and automatically updated in the stack pointer.

Reverse Polish Notation:

- A stack organization is very effective for evaluating arithmetic expressions.
- The common arithmetic expressions are written in infix notation, with each operator written between the operands.
- Consider the simple arithmetic expression. A*B+C*D
- For evaluating the above expression it is necessary to compute the product A*B, store this product result while computing C*D, and then sum the two products.
- For doing this type of infix notation, it is necessary to scan back and forth along the expression to determine the next operation to be performed.
- The Postfix notation, referred to as reverse Polish notation (RPN), places the operator after the operands.
- The following examples demonstrate the three representations

Eg: A+B ----->	Infix notation
+AB ------>	Prefix or Polish notation
AB+ ------->	Post or reverse Polish notation

- The reverse Polish notation is in a form suitable for stack manipulation.
- The expression A*B+C*D Is written in reverse polish notation as AB* CD* + And it is evaluated as follows
- Scan the expression from left to right. When operator is reached, perform the operation with the two operands found on the left side of the operator.
- Remove the two operands and the operator and replace them by the number obtained from the result of the operation.
- Continue to scan the expression and repeat the procedure for every operation encountered until there are no more operators.

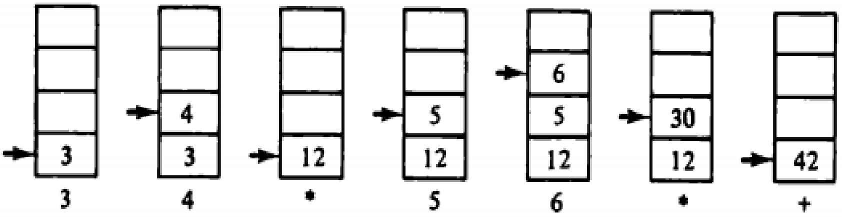
- For the expression above it find the operator * after A and B. So it perform the operation A*B and replace A, B and * with the result.
- The next operator is a * and it previous two operands are C and D, so it perform the operation C*D and places the result in places C, D and *.
- The next operator is + and the two operands to be added are the two products, so we add the two quantities to obtain the result.
- The conversion from infix notation to reverse Polish notation must take into consideration the operational hierarchy adopted for infix notation.
- This hierarchy dictates that we first perform all arithmetic inside inner parentheses, then inside outer parentheses, and do multiplication and division operations before addition and subtraction operations.

Evaluation of Arithmetic Expressions:

- Reverse Polish notation, combined with a stack arrangement of registers, is the most efficient way known for evaluating arithmetic expressions.
- This procedure is employed in some electronic calculators and also in some computer.
- 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 Figure.



- ➢ 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 top most items 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 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.

Computer Instruction Formats

- Computers may have instructions of several different lengths containing varying number of addresses.
- The number of address fields in the instruct 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.

Single Accumulator Organization:

- In an accumulator type organization all the operations are performed with an implied accumulator register.
- The instruction format in this type of computer uses one address field.
- For example, the instruction that specifies an arithmetic addition defined by an assembly language instruction as
- ADD X
- Where X is the address of the operand. The ADD instruction in this case results in the operation AC
- AC +M[X]. AC is the accumulator register and M[X] symbolizes the memory word located at address X.

General register organization:

- The instruction format in this type of computer needs three register address fields.
- Thus the instruction for an arithmetic addition may be written in an assembly language as ADD R1, R2, R3 to denote the operation R1
- R2 + R3. The number of address fields in the instruction can be reduced from three to two if the destination register is the same as one of the source registers.
- Thus the instruction ADD R1, R2 would denote the operation R1
- R1 + R2. Only register addresses for R1 and R2 need be specified in this instruction.
- General register-type computers employ two or three address fields in their instruction format.
- Each address field may specify a processor register or a memory word.
- An instruction symbolized by ADD R1, X would specify the operation R1
- R1 + M[X].
- It has two address fields, one for register R1 and the other for the memory address X.

Stack organization:

- The stack-organized CPU has PUSH and POP instructions which require an address field.
- Thus the instruction PUSH X will push the word at address X to the top of the stack.
- The stack pointer is updated automatically.

- Operation-type instructions do not need an address field in stack-organized computers.
- This is because the operation is performed on the two items that are on top of the stack.
- The instruction ADD in a stack computer consists of an operation code only with no address field.
- This operation has the effect of popping the two top numbers from the stack, adding the numbers, and pushing the sum into the stack.
- There is no need to specify operands with an address field since all operands are implied to be in the stack.
- Most computers fall into one of the three types of organizations.
- Some computers combine features from more than one organizational structure.
- The influence of the number of addresses on computer programs, we will evaluate the arithmetic statement X= (A+B) * (C+D)
- Using zero, one, two, or three address instructions and using the symbols
    - ADD, SUB, MUL and DIV for four arithmetic operations;
    - MOV for the transfer type operations; and
    - LOAD and STORE for transfer to and from memory and AC register.
- Assuming that the operands are in memory addresses A, B, C, and D and the result must be stored in memory ar address X and also the CPU has general purpose registers R1, R2, R3 and R4.

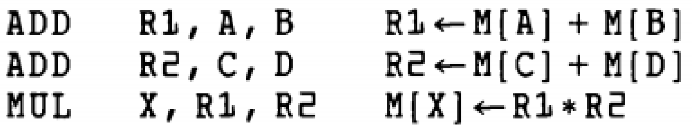
Instruction Formats:

Three Address Instructions:

- Three-address instruction formats can use each address field to specify either a processor register or a memory operand.
- The program assembly language that evaluates

X = (A+B) * (C+D)

- is shown below, together with comments that explain the register transfer operation of each instruction

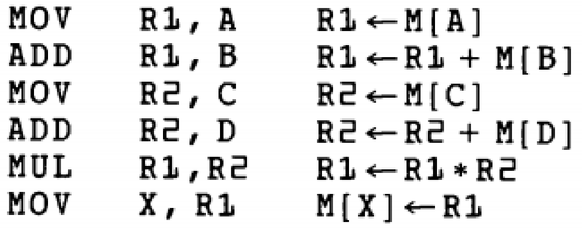


```
ADD    R1, A, B     R1←M[A] + M[B]
ADD    R2, C, D     R2←M[C] + M[D]
MUL    X, R1, R2    M[X]←R1*R2
```

- The symbol M [A] denotes the operand at memory address symbolized by A.
- The advantage of the three-address format is that it results in short programs when evaluating arithmetic expressions.
- The disadvantage is that the binary-coded instructions require too many bits to specify three addresses.

Two Address Instructions:

- Two-address instructions formats use each address field can specify either a processor register or memory word.
- The program to evaluate X = (A+B) * (C+D) is as follows

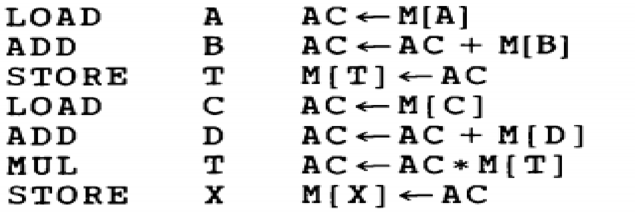


```
MOV    R1, A    R1←M[A]
ADD    R1, B    R1←R1 + M[B]
MOV    R2, C    R2←M[C]
ADD    R2, D    R2←R2 + M[D]
MUL    R1,R2    R1←R1*R2
MOV    X, R1    M[X]←R1
```

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

One Address Instructions:

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

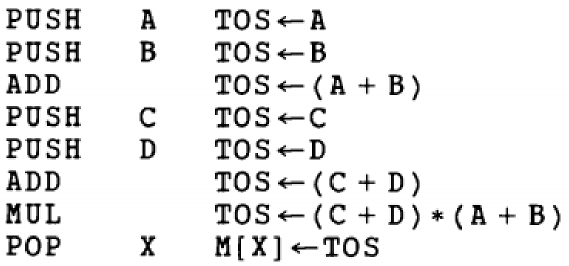


```
LOAD     A    AC←M[A]
ADD      B    AC←AC + M[B]
STORE    T    M[T]←AC
LOAD     C    AC←M[C]
ADD      D    AC←AC + M[D]
MUL      T    AC←AC*M[T]
STORE    X    M[X]←AC
```

- ➢ 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) * (C+D) will be written for a stack-organized computer. (TOS stands for top of stack).

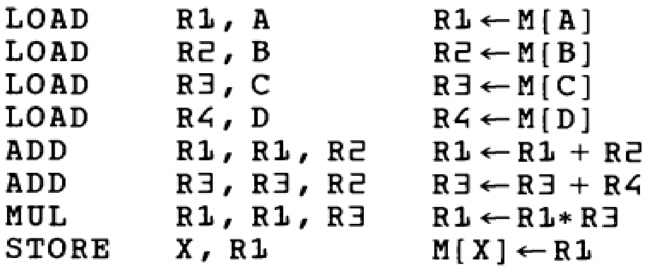


```
PUSH   A    TOS←A
PUSH   B    TOS←B
ADD         TOS←(A + B)
PUSH   C    TOS←C
PUSH   D    TOS←D
ADD         TOS←(C + D)
MUL         TOS←(C + D)*(A + B)
POP    X    M[X]←TOS
```

- 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.

RISC Instructions:

- The instruction set of a typical RISC processor is used to only load and store instructions for communicating between memory and CPU.
- All other instructions are executed within the registers of CPU without referring to memory.
- LOAD and STORE instructions that have one memory and one register address, and computational type instructions that have three addresses with all three specifying processor registers.
- The following is a program to evaluate X=(A+B)*(C+D)

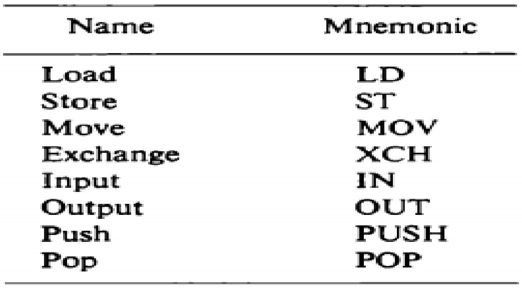


```
LOAD     R1, A          R1←M[A]
LOAD     R2, B          R2←M[B]
LOAD     R3, C          R3←M[C]
LOAD     R4, D          R4←M[D]
ADD      R1, R1, R2     R1←R1 + R2
ADD      R3, R3, R2     R3←R3 + R4
MUL      R1, R1, R3     R1←R1*R3
STORE    X, R1          M[X]←R1
```

Data Transfer and Manipulation

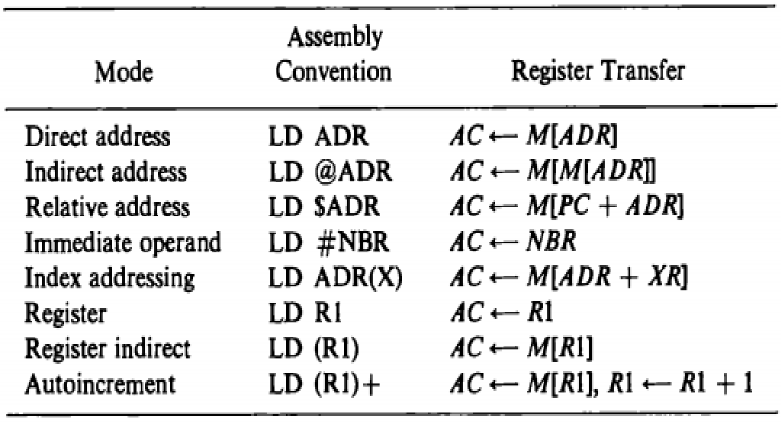
Data Transfer Instructions:

- Data transfer instructions move data from one place in the computer to another without changing the data content.
- The most common transfers are between memory and processor registers, between processor registers and input or output, and between the processor registers themselves.
- The following table gives a list of eight data transfer instructions used in many computers.



Name	Mnemonic
Load	LD
Store	ST
Move	MOV
Exchange	XCH
Input	IN
Output	OUT
Push	PUSH
Pop	POP

- ➢ The load instruction has been used mostly to designate a transfer from memory to a processor register, usually an accumulator.
- ➢ The store instruction designates a transfer from a processor register into memory.
- ➢ The move instruction has been used in computers with multiple CPU registers to designate a transfer from one register to another and also between CPU registers and memory or between two memory words.
- ➢ The exchange instruction swaps information between two registers or a register and a memory word.
- ➢ The input and output instructions transfer data among processor registers and input or output terminals.
- ➢ The push and pop instructions transfer data between processor registers and a memory stack.
- ➢ Different computers use different mnemonics symbols for differentiate the addressing modes.
- ➢ As an example, consider the load to accumulator instruction when used with eight different addressing modes.
- ➢ The following table shows the recommended assembly language convention and actual transfer accomplished in each case



Mode	Assembly Convention	Register Transfer
Direct address	LD ADR	$AC \leftarrow M[ADR]$
Indirect address	LD @ADR	$AC \leftarrow M[M[ADR]]$
Relative address	LD \$ADR	$AC \leftarrow M[PC + ADR]$
Immediate operand	LD #NBR	$AC \leftarrow NBR$
Index addressing	LD ADR(X)	$AC \leftarrow M[ADR + XR]$
Register	LD R1	$AC \leftarrow R1$
Register indirect	LD (R1)	$AC \leftarrow M[R1]$
Autoincrement	LD (R1)+	$AC \leftarrow M[R1],\ R1 \leftarrow R1 + 1$

- ADR stands for an address.
- NBA a number or operand.
- X is an index register.
- R1 is a processor register.

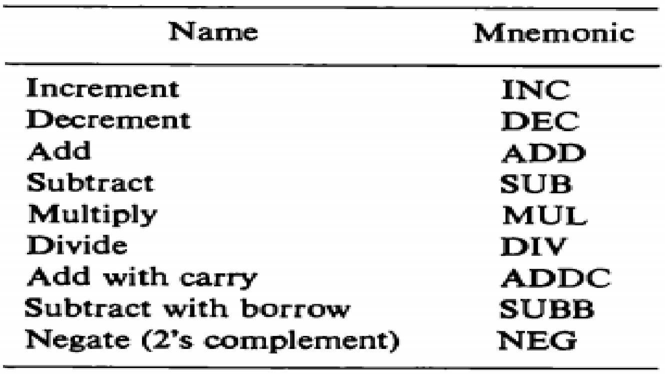
- AC is the accumulator register.
- The @ character symbolizes an indirect addressing.
- The $ character before an address makes the address relative to the program counter PC.
- The # character precedes the operand in an immediate-mode instruction.
- An indexed mode instruction is recognized by a register that placed in parentheses after the symbolic address.
- The register mode is symbolized by giving the name of a processor register.
- In the register indirect mode, the name of the register that holds the memory address is enclosed in parentheses.
- The auto-increment mode is distinguished from the register indirect mode by placing a plus after the parenthesized register.
- The auto-decrement mode would use a minus instead.

Data Manipulation Instructions:

- Data manipulation instructions perform operations on data and provide the computational capabilities for the computer.
- The data manipulation instructions in a typical computer are usually divided into three basic types:
    1. Arithmetic instructions
    2. Logical and bit manipulation instructions
    3. Shift instructions

2. **Arithmetic instructions**

- ➢ The four basic arithmetic operations are addition, subtraction, multiplication and division.
- ➢ Most computers provide instructions for all four operations.
- ➢ Some small computers have only addition and possibly subtraction instructions. The multiplication and division must then be generated by mean software subroutines.
- ➢ A list of typical arithmetic instructions is given in Table.



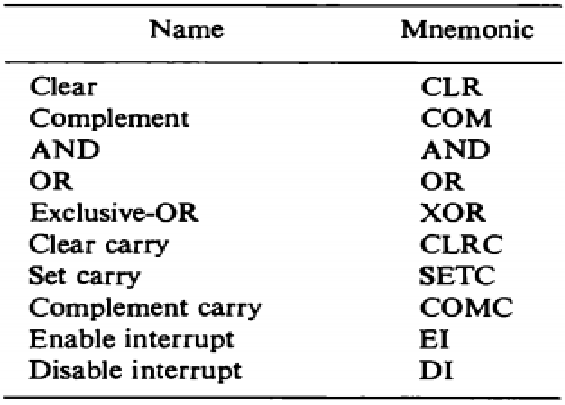
Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with carry	ADDC
Subtract with borrow	SUBB
Negate (2's complement)	NEG

- The increment instruction adds 1 to the value stored in a register or memory word.
- A number with all 1's, when incremented, produces a number with all 0's.
- The decrement instruction subtracts 1 from a value stored in a register or memory word.

- A number with all 0's, when decremented, produces number with all 1's.
- The add, subtract, multiply, and divide instructions may be use different types of data.
- The data type assumed to be in processor register during the execution of these arithmetic operations is defined by an operation code.
- An arithmetic instruction may specify fixed-point or floating-point data, binary or decimal data, single-precision or double-precision data.
- The mnemonics for three add instructions that specify different data types are shown below.
    - ADDI - Add two binary integer numbers
    - ADDF - Add two floating-point numbers
    - ADDD - Add two decimal numbers in BCD
- A special carry flip-flop is used to store the carry from an operation.
- The instruction "add carry" performs the addition on two operands plus the value of the carry the previous computation.
- Similarly, the "subtract with borrow" instruction subtracts two words and borrow which may have resulted from a previous subtract operation.
- The negate instruction forms the 2's complement number, effectively reversing the sign of an integer when represented it signed-2's complement form.

2. Logical and bit manipulation instructions

- Logical instructions perform binary operations on strings of bits store, registers.
- They are useful for manipulating individual bits or a group of that represent binary-coded information.
- The logical instructions consider each bit of the operand separately and treat it as a Boolean variable.
- By proper application of the logical instructions it is possible to change bit values, to clear a group of bits, or to insert new bit values into operands stored in register memory words.
- Some typical logical and bit manipulation instructions are listed in Table



Name	Mnemonic
Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Set carry	SETC
Complement carry	COMC
Enable interrupt	EI
Disable interrupt	DI

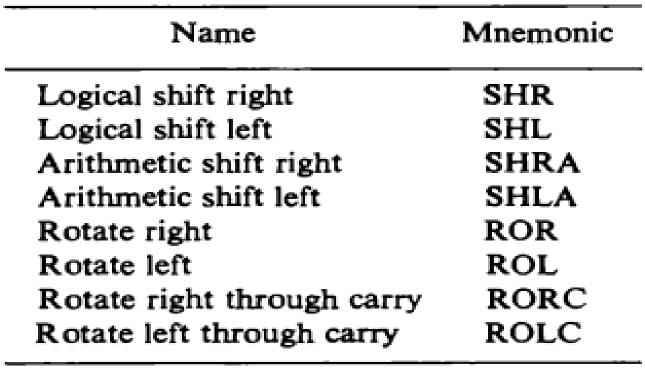
There are three bit manipulation operations possible:

4. a selected bit can cleared to 0,
5. or can be set to 1,

6. or can be complemented.

3. Shift Instructions:

- Shifts are operations in which the bits of a word are moved to the left or right.
- The bit shifted in at the end of the word determines the type of shift used.
- Shift instructions may specify logical shifts, arithmetic shifts, or rotate-type operations.
- In either case the shift may be to the right or to the left.
- The following table lists four types of shift instructions.



Name	Mnemonic
Logical shift right	SHR
Logical shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate right through carry	RORC
Rotate left through carry	ROLC

- The logical shift insert 0 to the end bit position.
- The end position is the leftmost bit position for shift rights the rightmost bit position for the shift left.
- Arithmetic shifts usually conform to the rules for signed-2's complement numbers.
- The arithmetic shift-right instruction must preserve the sign bit in the leftmost position.
- The sign bit is shifted to the right together with the rest of the number, but the sign bit itself remains unchanged.
- This is a shift-right operation with the end bit remaining the same.
- The arithmetic shift-left instruction inserts 0 to the end position and is identical to the logical shift-instruction.
- The rotate instructions produce a circular shift. Bits shifted out at one of the word are not lost as in a logical shift but are circulated back into the other end.
- The rotate through carry instruction treats a carry bit as an extension of the register whose word is being rotated.
- Thus a rotate-left through carry instruction transfers the carry bit into the rightmost bit position of the register, transfers the leftmost bit position into the carry, and at the same time, shift the entire register to the left.

6. Reduced Instruction Set Computer (RISC):

- A computer with large number instructions is classified as a complex instruction set computer, abbreviated as CISC.
- The computer which is having the fewer instructions is classified as a reduced instruction set computer, abbreviated as RISC.

RISC Characteristics:

- Relatively few instructions
- Relatively few addressing modes
- Memory access limited to load and store instructions
- All operations done within the registers of the CPU
- Fixed-length, easily decoded instruction format
- Single-cycle instruction execution
- Hardwired rather than microprogrammed control
- A relatively large number of registers in the processor unit
- Efficient instruction pipeline

CISC Characteristics:

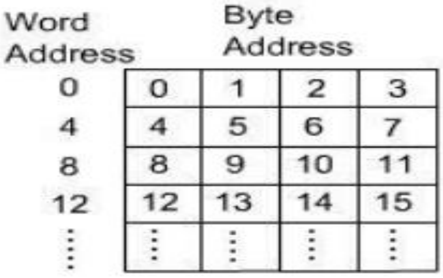
- A large number of instructions--typically from 100 to 250 instructions.
- Some instructions that perform specialized tasks and are used infrequently.
- A large variety of addressing modes—typically from 5 to 20 differ modes.
- Variable-length instruction formats

Instructions that manipulate operands in memory

UNIT-5

Memory Organization

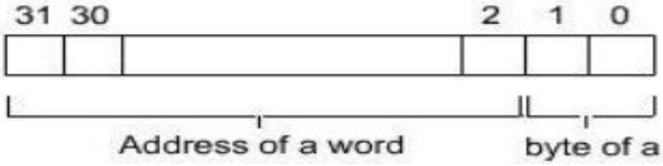
- A memory is just like a human brain.
- It is used to store data and instructions.
- Computer memory is the storage space in the computer, where data is to be processed and instructions required for processing are stored.
- The memory is divided into large number of small parts called cells.
- Each location or cell has a unique address, which varies from zero to memory size minus one.
- For example, if the computer has 64k words, then this memory unit has 64 * 1024 = 65536 memory locations. The address of these locations varies from 0 to 65535.
- Memory is primarily of three types −
  1. Cache Memory
  2. Primary Memory/Main Memory
  3. Secondary Memory
- If the smallest addressable unit of information is a memory word, the machine is called word-addressable.
- If individual memory bytes are assigned distinct addresses, the computer is called byte-addressable.
- Most of the commercial machines are byte addressable.
- For example in a byte-addressable 32-bit computer, each memory word contains 4 bytes.
- A possible word-address assignment would be: Word Address Byte Address



Word Address	Byte Address			
0	0	1	2	3
4	4	5	6	7
8	8	9	10	11
12	12	13	14	15
⋮	⋮	⋮	⋮	⋮

Organization of the main memory in a 32-bit byte addressable computer

32 bit address bus/word size is 32 bit

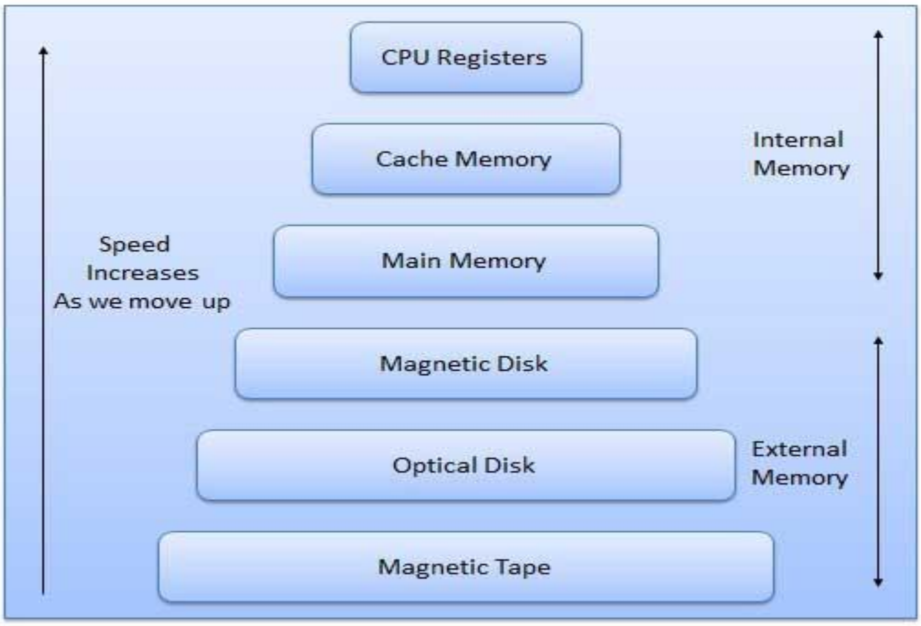


- With the above structure a READ or WRITE may involve an entire memory word or it may involve only a byte.

- In the case of byte read, other bytes can also be read but ignored by the CPU. However, during a write cycle, the control circuitry of the MM must ensure that only the specified byte is altered.
- In this case, the higher-order 30 bits can specify the word and the lower-order 2 bits can specify the byte within the word.

Memory Hierarchy

- The term memory hierarchy is used in computer architecture when discussing performance issues in computer architectural design, algorithm predictions, and the lower level programming constructs such as involving locality of reference.
- A "memory hierarchy" in computer storage distinguishes each level in the "hierarchy" by response time.
- Since response time, complexity, and capacity are related, the levels may also be distinguished by the controlling technology.



Register:

- This is a part of Central Processor Unit, so they reside inside the CPU.
- The information from main memory is brought to CPU and keep the information in register.
- Due to space and cost constraints, we have got a limited number of registers in a CPU.
- These are basically faster devices.

Cache Memory:

- Cache memory is a storage device placed in between CPU and main memory.
- These are semiconductor memories.
- These are basically fast memory device, faster than main memory.
- We can not have a big volume of cache memory due to its higher cost and some constraints of the CPU.
- Due to higher cost we can not replace the whole main memory by faster memory.
- Generally, the most recently used information is kept in the cache memory.

- It is brought from the main memory and placed in the cache memory. Now a days, we get CPU with internal cache.

Main Memory:

- Like cache memory, main memory is also semiconductor memory.
- But the main memory is relatively slower memory. We have to first bring the information (whether it is data or program), to main memory.
- CPU can work with the information available in main memory only.

Magnetic Disk:

- This is bulk storage device. We have to deal with huge amount of data in many application.
- But we don't have so much semiconductor memory to keep these information in our computer.
- On the other hand, semiconductor memories are volatile in nature. It loses its content once we switch off the computer.
- For permanent storage, we use magnetic disk.
- The storage capacity of magnetic disk is very high.

Removable media:

- For different application, we use different data.
- It may not be possible to keep all the information in magnetic disk.
- So, which ever data we are not using currently, can be kept in removable media.

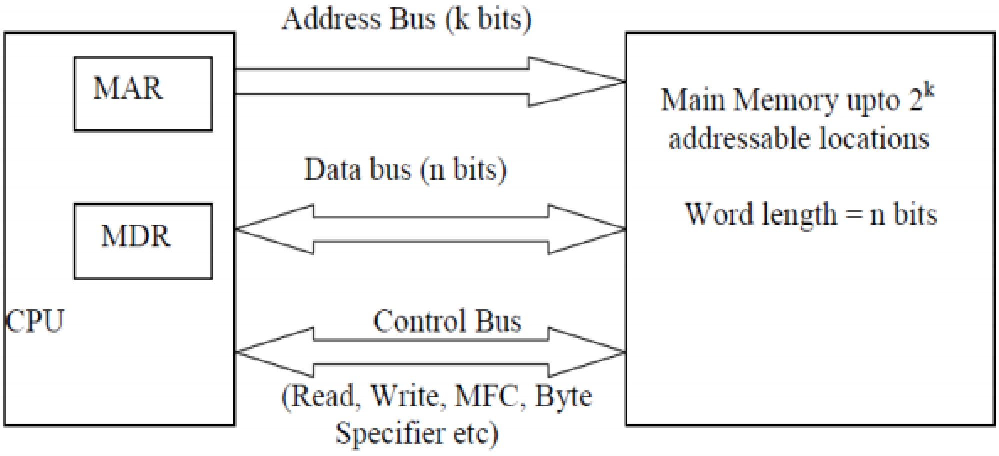
Magnetic tape is one kind of removable medium. CD is also a removable media, which is an optical.

CPU-Main Memory Connection – A block schematic:

- From the system standpoint, the Main Memory (MM) unit can be viewed as a “block box”.
- Data transfer between CPU and MM takes place through the use of two CPU registers, usually called MAR (Memory Address Register) and MDR (Memory Data Register).
- This transfer takes place over the processor bus, which has k address lines (address bus), n data lines (data bus) and control lines like Read, Write, Memory Function completed (MFC), Bytes specifiers etc (control bus).
- For a read operation, the CPU loads the address into MAR, set READ to 1 and sets other control signals if required.

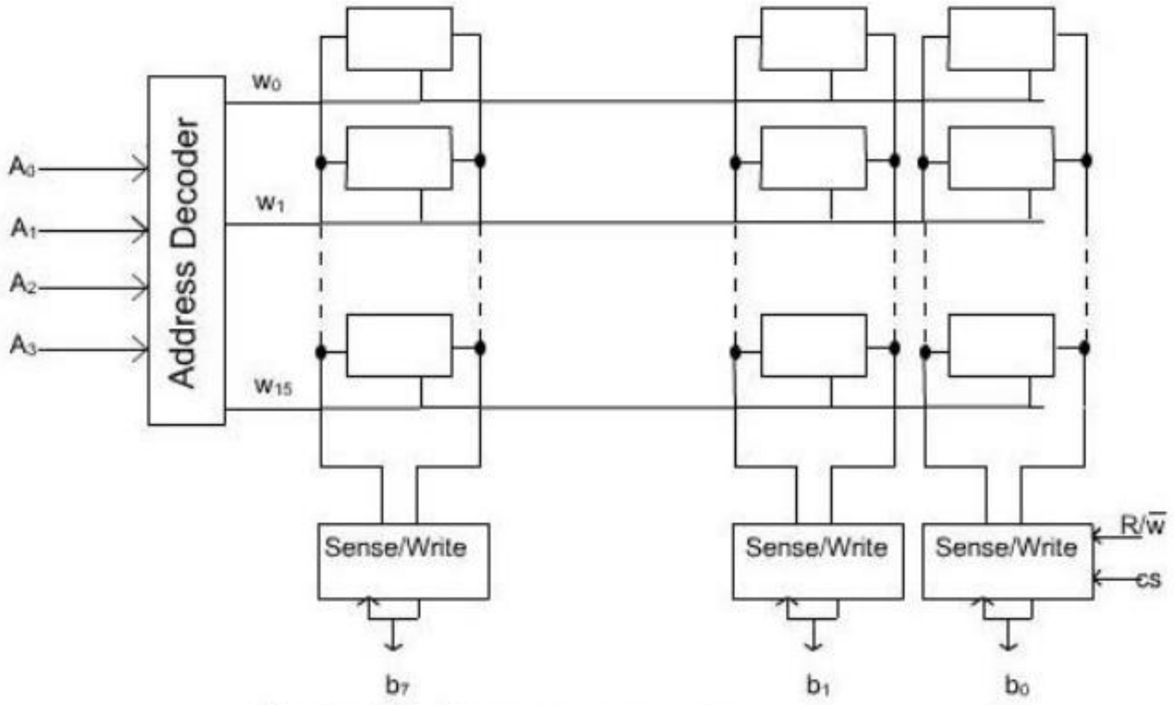
The data from the MM is loaded into MDR and MFC is set to

- For a write operation, MAR, MDR are suitably loaded by the CPU, write is set to 1 and other control signals are set suitably.
- The MM control circuitry loads the data into appropriate locations and sets MFC to 1.
- This organization is shown in the following block schematic.


Address Bus (k bits)
MAR
Main Memory upto 2^k addressable locations
Data bus (n bits)
MDR
Word length = n bits
CPU
Control Bus
(Read, Write, MFC, Byte Specifier etc)

Internal Organization of Semiconductor Memory Chips:

- Memory chips are usually organized in the form of an array of cells, in which each cell is capable of storing one bit of information.
- A row of cells constitutes a memory word, and the cells of a row are connected to a common line referred to as the word line, and this line is driven by the address decoder on the chip.
- The cells in each column are connected to a sense/write circuit by two lines known as bit lines.
- The sense/write circuits are connected to the data input/output lines of the chip.
- During a READ operation, the Sense/Write circuits sense, or read, the information stored in the cells selected by a word line and transmit this information to the output lines.
- During a write operation, they receive input information and store it in the cells of the selected word.
- A memory cell is capable of storing 1-bit of information. A number of memory cells are organized in the form of a matrix to form the memory chip. One such internal organization of memory chip is shown in the Figure.

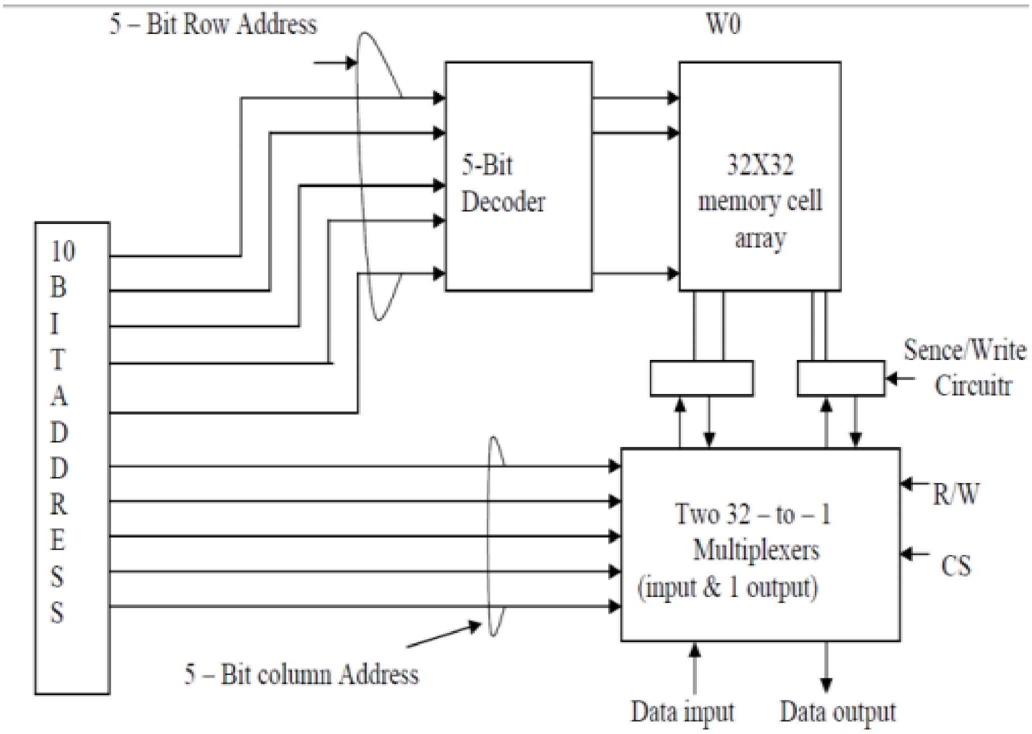


16 memory location w_0, w_1,..................w_{15}

8 bits in each location b_0, b_1, ...b_7

- Each row of cells constitutes a memory word, and all cell of a row are connected to a common line which is referred as word line.
- An address decoder is used to drive the word line.
- At a particular instant, one word line is enabled depending on the address present in the address bus.
- The cells in each column are connected by two lines. These are known as bit lines.
- These bit lines are connected to data input line and data output line through a Sense/ Write circuit.
- During a Read operation, the Sense/Write circuit sense, or read the information stored in the cells selected by a word line and transmit this information to the output data line.
- During a write operation, the sense/write circuit receive information and store it in the cells of the selected word
- The data input and data output line of each Sense/Write circuit are connected to a single bidirectional data line in order to reduce the pin required.
- For 16 words, we need an address bus of size 4. In addition to address and data lines, two control lines, R/W and CS, are provided.
- The line is to used to specify the required operation about read or write.
- The CS (Chip Select) line is required to select a given chip in a multi chip memory system.
- The following figure shows such an organization of a memory chip consisting of 16 words of 8 bits each, which is usually referred to as a 16 x 8 organization.
- The data input and the data output of each Sense/Write circuit are connected to a single bi-directional data line in order to reduce the number of pins required.

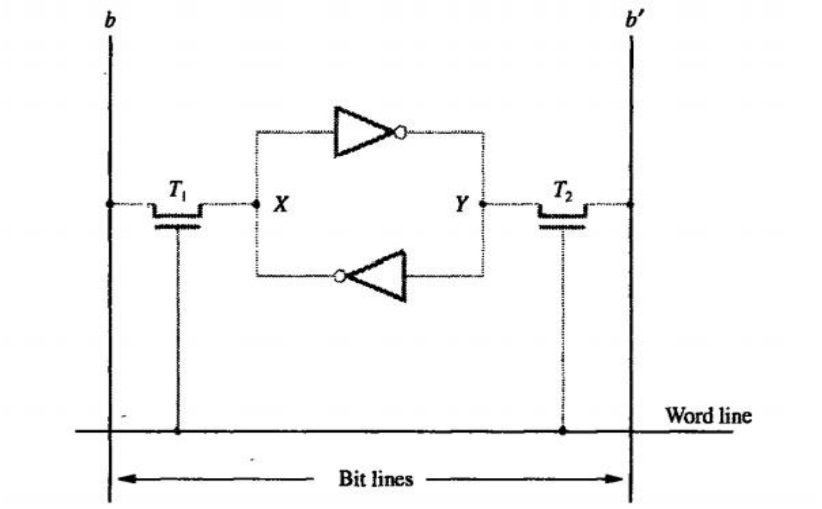
- One control line, the R/W (Read/Write) input is used a specify the required operation and another control line, the CS (Chip Select) input is used to select a given chip in a multichip memory system.
- This circuit requires 14 external connections, and allowing 2 pins for power supply and ground connections, can be manufactured in the form of a 16-pin chip.
- It can store 16 x 8 = 128 bits. Another type of organization for 1k x 1 format is shown below



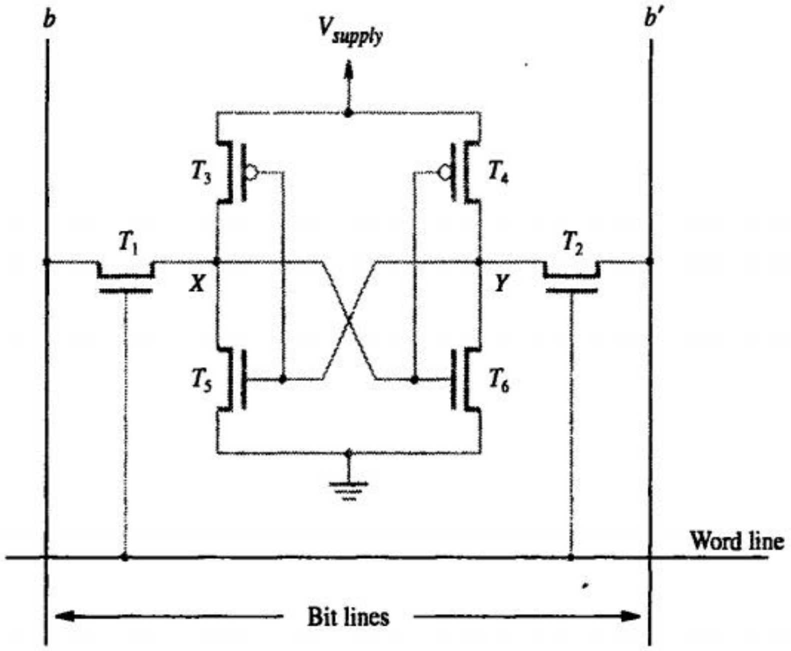
- Depending on the technology used to construct a RAM, there are two types of RAM –
  1. SRAM: Static Random Access Memory.
  2. DRAM: Dynamic Random Access Memory.

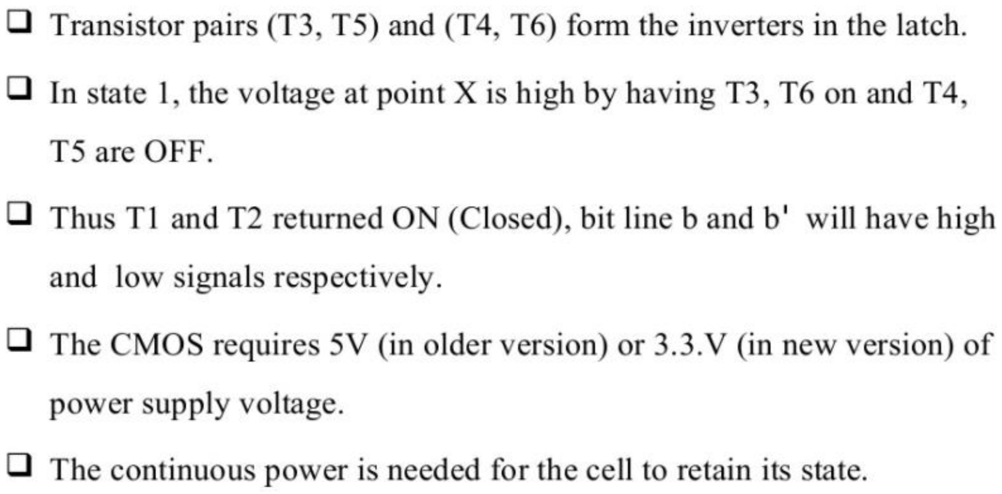
Static RAM (SRAM):

- In an SRAM, binary values are stored using traditional flip-flop constructed with the help of transistors.
- A static RAM will hold its data as long as power is supplied to it.
- A typical SRAM constructed with transistors is shown in the figure.



CMOS Memory Cell





- Transistor pairs (T3, T5) and (T4, T6) form the inverters in the latch.
- In state 1, the voltage at point X is high by having T3, T6 on and T4, T5 are OFF.
- Thus T1 and T2 returned ON (Closed), bit line b and b' will have high and low signals respectively.
- The CMOS requires 5V (in older version) or 3.3.V (in new version) of power supply voltage.
- The continuous power is needed for the cell to retain its state.

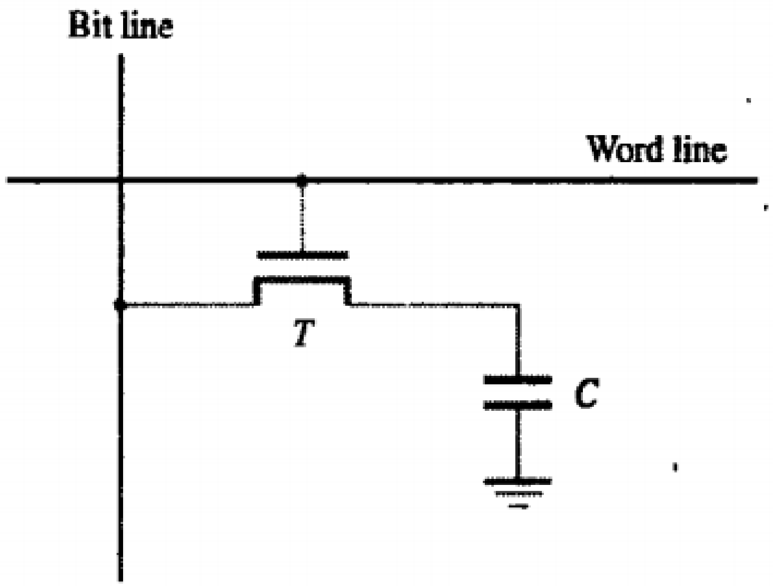
Dynamic Ram (DRAM):

There are two types of DRAM available they are

1. Asynchronous DRAM
2. Synchronous DRAM

Asynchronous DRAM

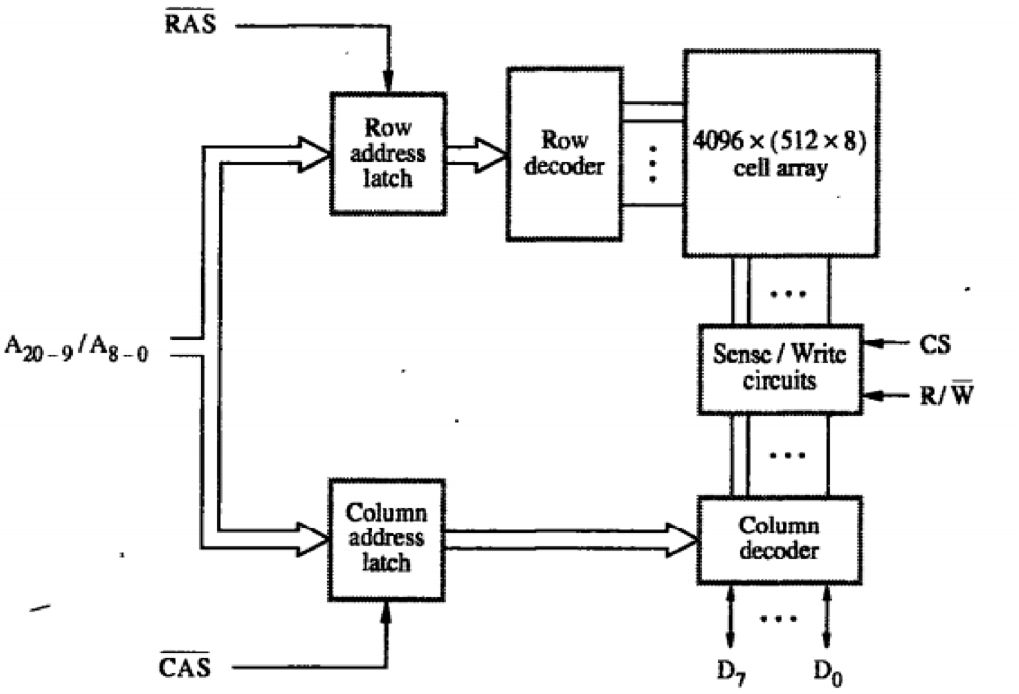
- A DRAM is made with cells that store data as charge on capacitors. The presence or absence of charge in a capacitor is interpreted as binary 1 or 0.
- Because capacitors have a natural tendency to discharge due to leakage current, dynamic RAM require periodic charge refreshing to maintain data storage.
- The term dynamic refers to this tendency of the stored charge to leak away, even with power continuously applied.
- A typical DRAM structure for an individual cell that stores one bit information is shown in the figure.



- For the write operation, a voltage signal is applied to the bit line, a high voltage represents 1 and a low voltage represents 0.
- A signal is then applied to the address line, which will turn on the transistor T, allowing a charge to be transferred to the capacitor.
- For the read operation, when a signal is applied to the address line, the transistor T turns on and the charge stored on the capacitor is fed out onto the bit line.

Typical Organization of a Dynamic Memory Chip

- The cells are organized in the form of a square array such that the high-and lower-order 8 bits of the 16-bit address constitute the row and column addresses of a cell, respectively.
- In order to reduce the number of pins needed for external connections, the row and column address are multiplexed on 8 pins.
- To access a cell, the row address is applied first.
- It is loaded into the row address latch in response to a single pulse on the Row Address Strobe (RAS) input.
- This selects a row of cells.
- Now, the column address is applied to the address pins and is loaded into the column address latch under the control of the Column Address Strobe (CAS) input and this address selects the appropriate sense/write circuit.
- If the R/W signal indicates a Read operation, the output of the selected circuit is transferred to the data output. Do.
- For a write operation, the data on the DI line is used to overwrite the cell selected.

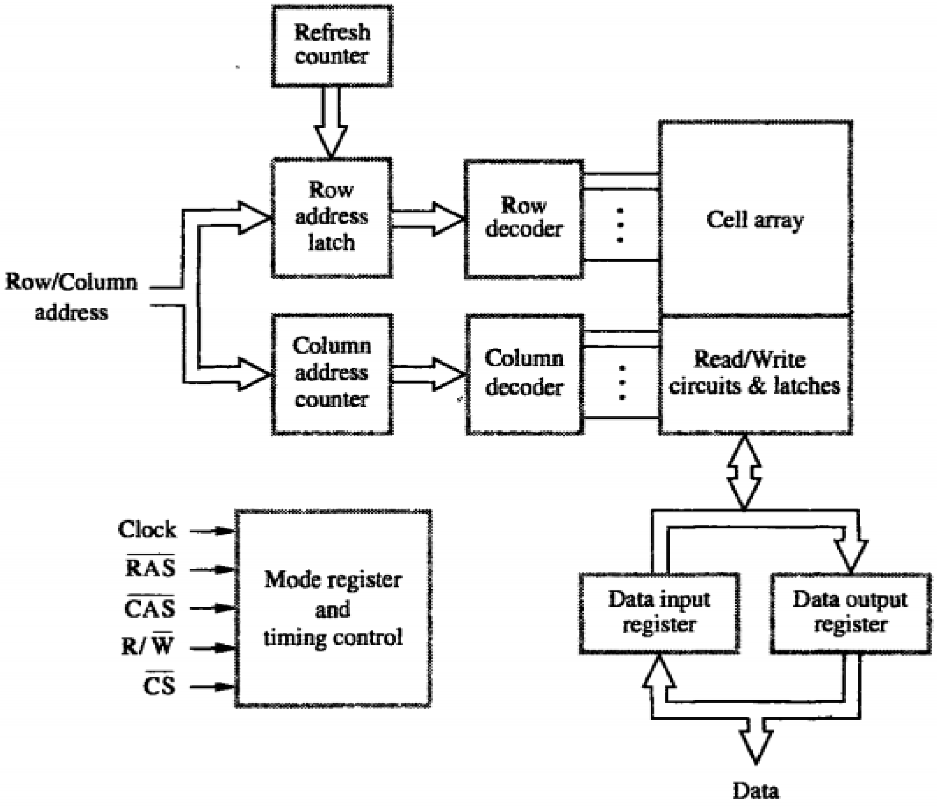


Synchronous DRAM

- Most recent developments in memory technology has resulted in DRAMs whose operation is directly synchronized with a clock signal. Such memories known as Synchronous DRAMs.

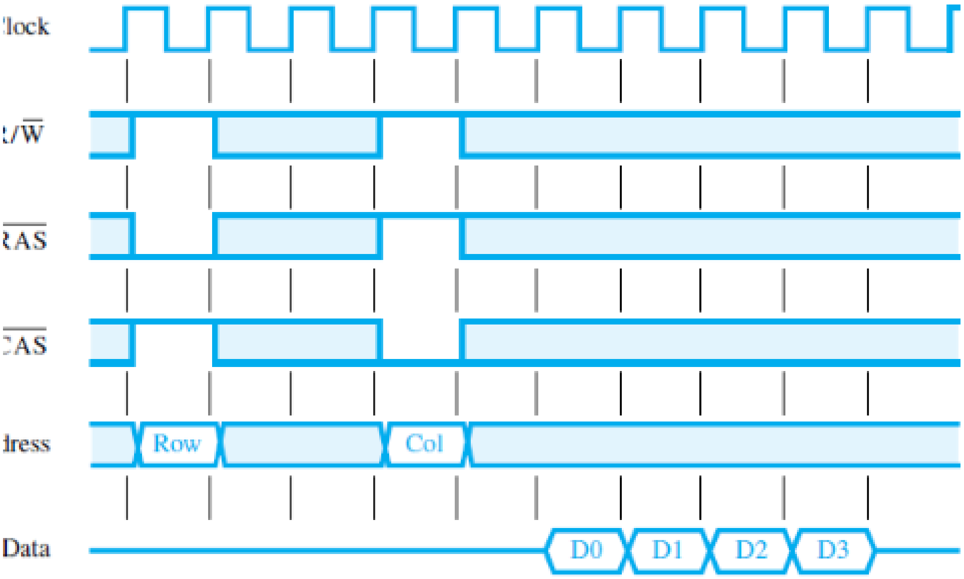
Cell array is same as Asynchronous DRAM.

- ❑ Here the operations are directly synchronized with clock signal.
- ❑ The address and data connections are buffered by means of registers.
- ❑ The output of each sense amplifier is connected to a latch.
- ❑ A Read operation causes the contents of all cells in the selected row to be loaded in these latches.
- ❑ The Figure shows the structure of SDRAM.
- ❑ Data held in the latches that correspond to the selected columns are transferred into the data output register, thus becoming available on the data output pins.



- ❑ The different modes of operation can be selected by writing control information into a mode register.
- ❑ SDRAM have a built in refresh circuitry.
- ❑ Apart of this circuitry is a refresh counter, which provides the address of the rows that are selected for refreshing.
- ❑ In a typical SDRAM each row must be refreshed at least every 64ms.

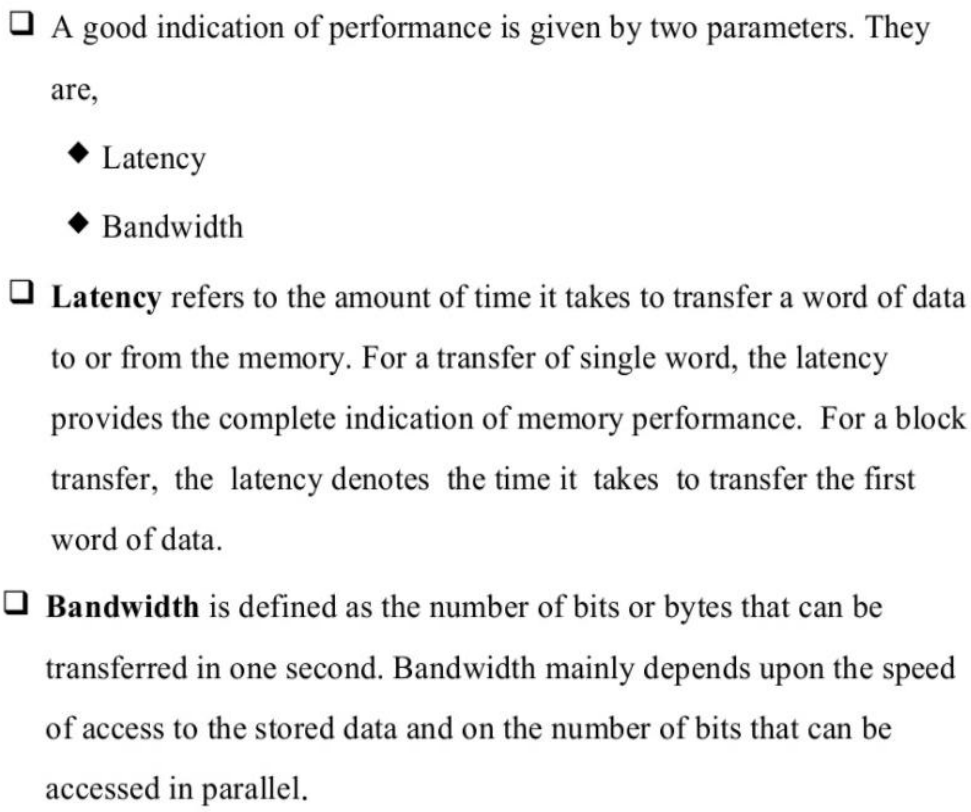
nce of fig. 8.9



ure 8.9 A burst read of length 4 in an SDRAM.

- ❑ First, the row address is latched under control of RAS signal.
- ❑ The memory typically takes 2 or 3 clock cycles to activate the selected row.
- ❑ Then the column address is latched under the control of CAS signal.
- ❑ After a delay of one clock cycle, the first set of data bits is placed on the data lines.
- ❑ The SDRAM automatically increments the column address to access the next 3 sets of bits in the selected row, which are placed on the data lines in the next 3 clock cycles.

Latency and Bandwidth



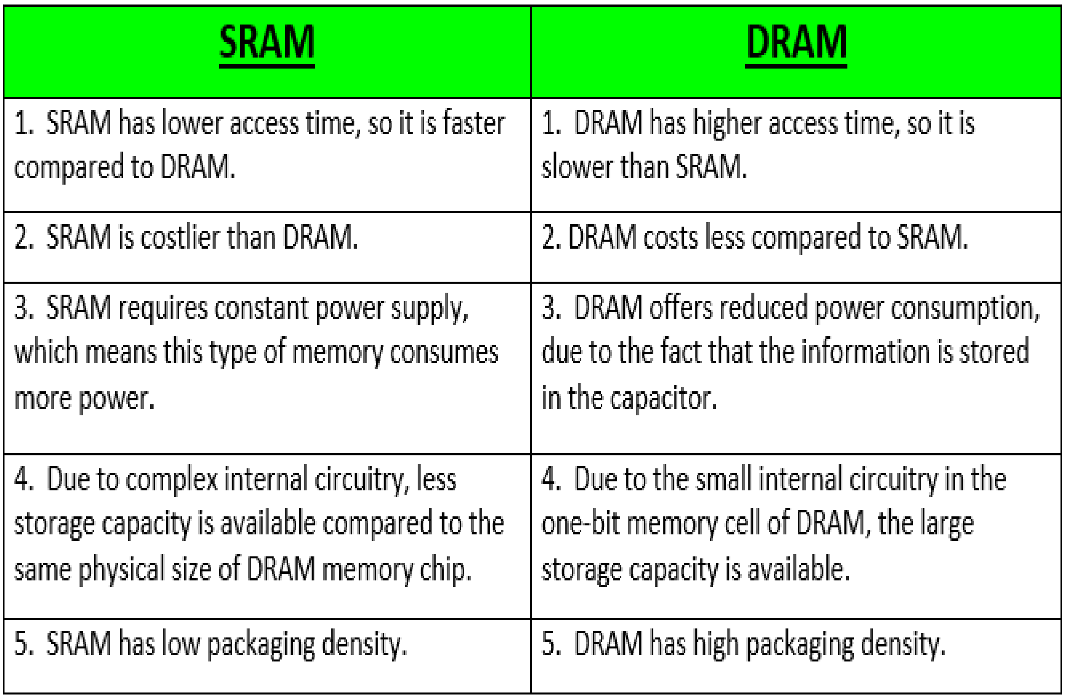
❑ A good indication of performance is given by two parameters. They are,

◆ Latency

◆ Bandwidth

❑ **Latency** refers to the amount of time it takes to transfer a word of data to or from the memory. For a transfer of single word, the latency provides the complete indication of memory performance. For a block transfer, the latency denotes the time it takes to transfer the first word of data.

❑ **Bandwidth** is defined as the number of bits or bytes that can be transferred in one second. Bandwidth mainly depends upon the speed of access to the stored data and on the number of bits that can be accessed in parallel.

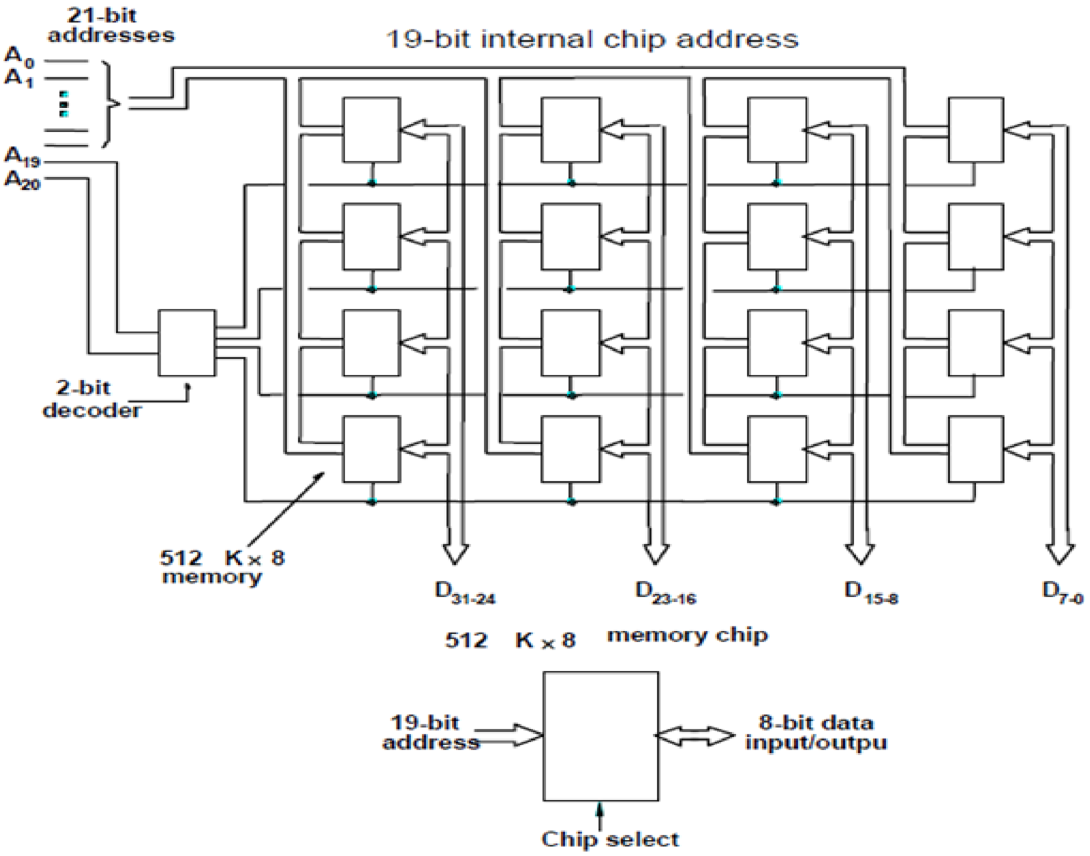


SRAM	**DRAM**
1. SRAM has lower access time, so it is faster compared to DRAM.	1. DRAM has higher access time, so it is slower than SRAM.
2. SRAM is costlier than DRAM.	2. DRAM costs less compared to SRAM.
3. SRAM requires constant power supply, which means this type of memory consumes more power.	3. DRAM offers reduced power consumption, due to the fact that the information is stored in the capacitor.
4. Due to complex internal circuitry, less storage capacity is available compared to the same physical size of DRAM memory chip.	4. Due to the small internal circuitry in the one-bit memory cell of DRAM, the large storage capacity is available.
5. SRAM has low packaging density.	5. DRAM has high packaging density.

Structure of Larger Memories

Static Memory System:

- Consider a memory consisting of 2M (2,097,152) words of 32 bits each.
- The following figure shows how we can implement this memory using 512K x 8 static memory chips.
- Each column in the figure consists of four chips, which implement one byte position.
- Four of these sets provide the required 2M x 32 memory.
- Each chip has a control input called Chip Select.
- When this input is set to 1, it enables the chip to accept data from or to place data on its data lines.
- The data output for each chip is of the three-state type.
- Only the selected chip places data on the data output line, while all other outputs are in the high-impedance state.
- Twenty one address bits are needed to select a 32-bit word in this memory.
- The high-order 2 bits of the address are decoded to determine which of the four Chip Select control signals should be activated and the remaining 19 address bits are used to access specific byte locations inside each chip of the selected row.
- The R/W inputs of all chips are tied together to provide a common Rea d/Write control (not shown in the figure).



Memory System Considerations

- ❑ To reduce the number of pins, the dynamic memory chips use multiplexed address inputs.
- ❑ The address is divided into two parts.
- ❑ They are,
  - ◆ **High Order Address Bit** (Select a row in cell array and it is provided first and latched into memory chips under the control of RAS signal).
  - ◆ **Low Order Address Bit** (Selects a column and they are provided on same address pins and latched using CAS signals).
- ❑ The Multiplexing of address bit is usually done by Memory Controller Circuit,

Memory Controller

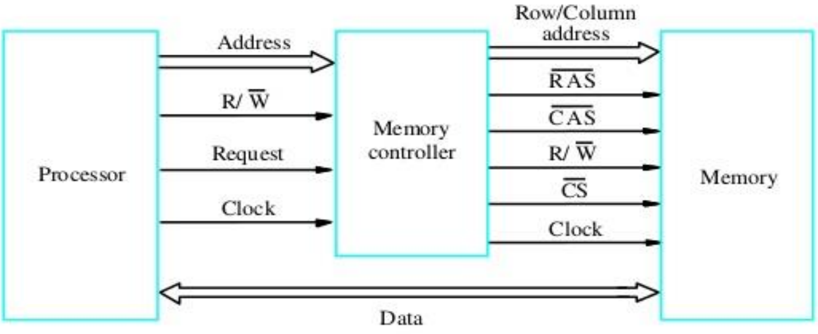
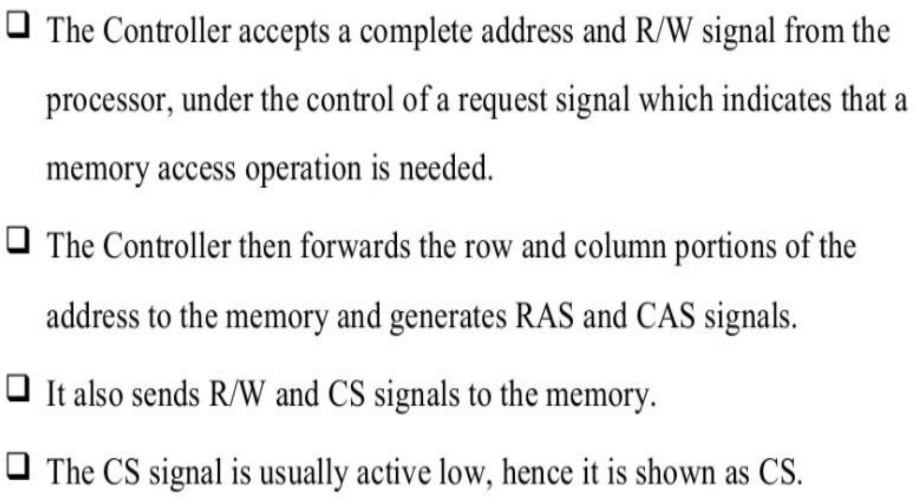
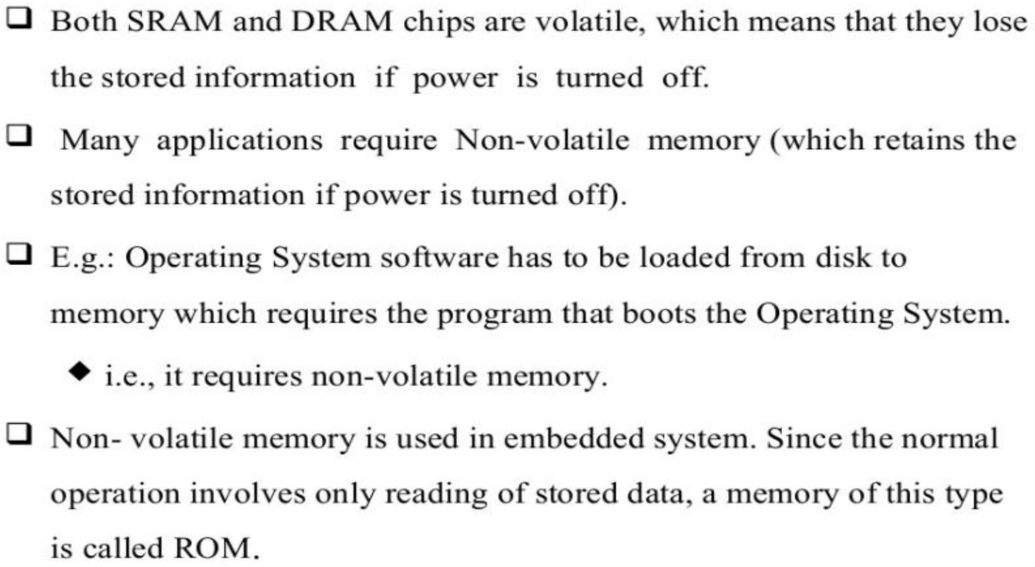


Figure 5.11. Use of a memory controller.



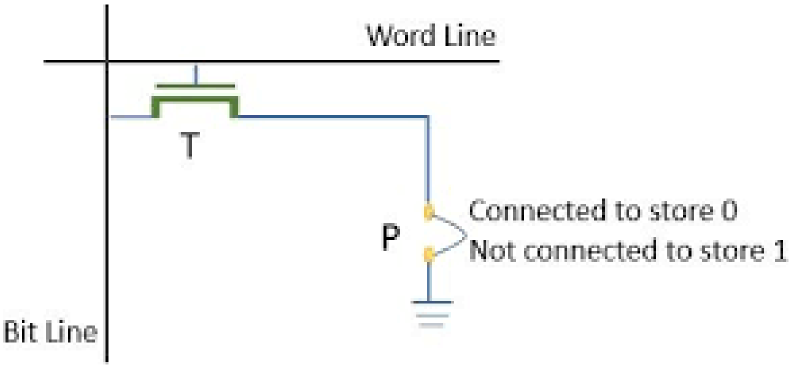
- ❑ The Controller accepts a complete address and R/W signal from the processor, under the control of a request signal which indicates that a memory access operation is needed.
- ❑ The Controller then forwards the row and column portions of the address to the memory and generates RAS and CAS signals.
- ❑ It also sends R/W and CS signals to the memory.
- ❑ The CS signal is usually active low, hence it is shown as CS.

Read Only Memory



- ❑ Both SRAM and DRAM chips are volatile, which means that they lose the stored information if power is turned off.
- ❑ Many applications require Non-volatile memory (which retains the stored information if power is turned off).
- ❑ E.g.: Operating System software has to be loaded from disk to memory which requires the program that boots the Operating System.
  - ◆ i.e., it requires non-volatile memory.
- ❑ Non- volatile memory is used in embedded system. Since the normal operation involves only reading of stored data, a memory of this type is called ROM.

ROM Cell



A ROM Memory Cell

- **At Logic value _0‘** → Transistor (T) is connected to the ground point (P).
- Transistor switch is closed and voltage on bit line nearly drops to zero.
- **At Logic value _1‘** → Transistor switch is open. The bit line remains at high voltage.
- To read the state of the cell, the word line is activated.
- A Sense circuit at the end of the bit line generates the proper output value.
- PROM allows the data to be loaded by the user.
- Programmability is achieved by inserting a fuse at point P in a ROM cell.
- Before it is programmed, the memory contains all 0‘s.
- The user can insert 1‘s at the required location by burning out the fuse at these locations using high current pulse.
- This process is irreversible.

- **Merit:**
  - It provides flexibility.
  - It is faster.
  - It is less expensive because they can be programmed directly by the user.

❑ EPROM allows the stored data to be erased and new data to be loaded.

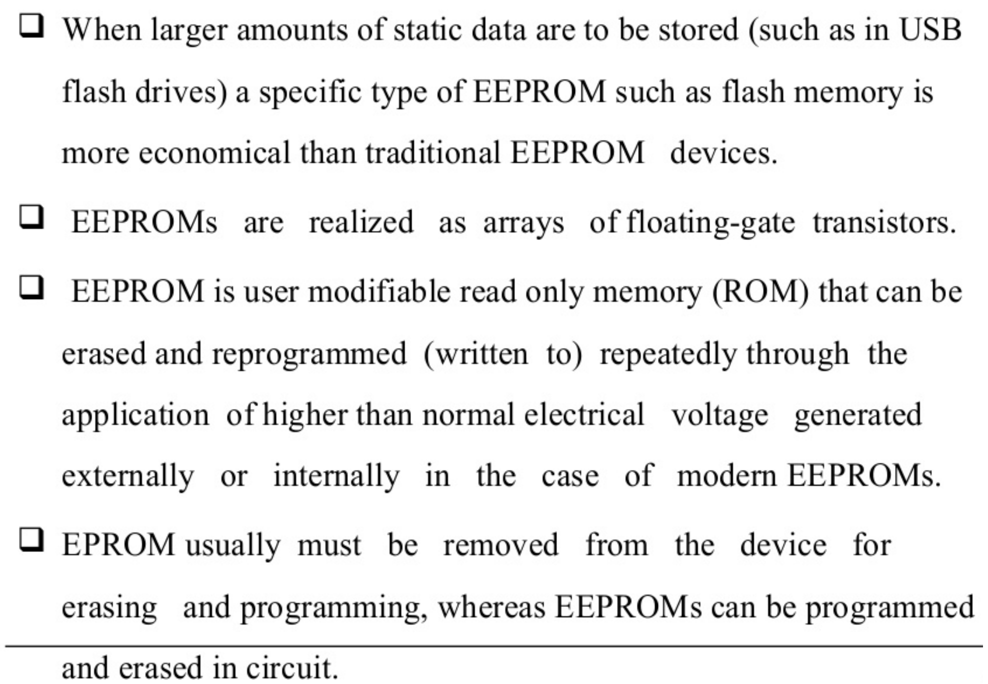
❑ In an EPROM cell, a connection to ground is always made at _P‘ and a special transistor is used, which has the ability to function either as a normal transistor or as a disabled transistor that is always turned off.

❑ This transistor can be programmed to behave as a permanently open switch, by injecting charge into it that becomes trapped inside.

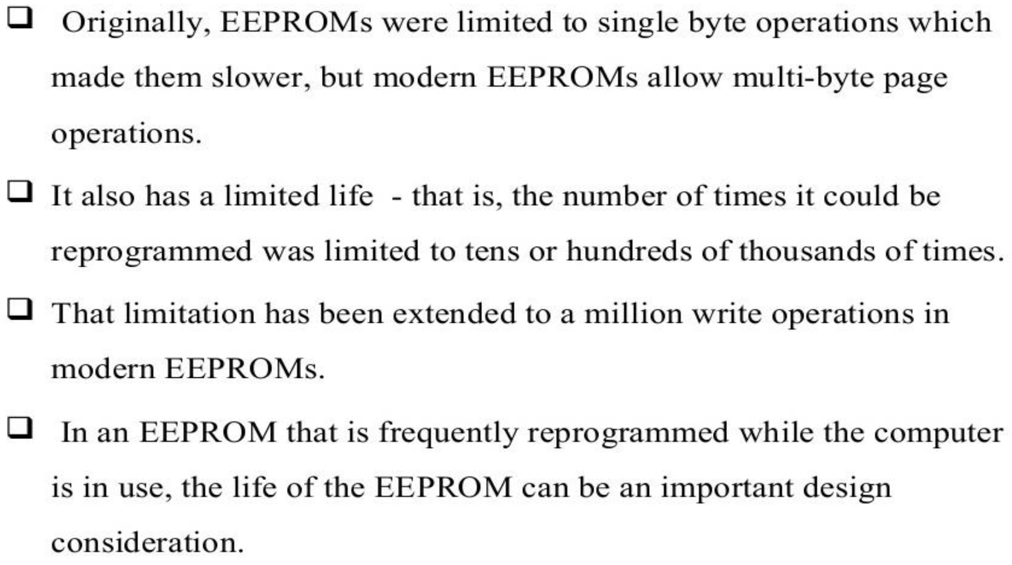
❑ Erasure requires dissipating the charges trapped in the transistor of memory cells.

❑ This can be done by exposing the chip to ultraviolet light, so that EPROM chips are mounted in packages that have transparent

❑ **EEPROM** (also written **E2PROM** and pronounced "e-e-prom," "double-e prom," "e-squared," or simply "e-prom") stands for **E**lectrically **E**rasable **P**rogrammable **R**ead- **O**nly **M**emory and is a type of non-volatile memory used in computers and other electronic devices to store small amounts of data that must be saved when power is removed, e.g., calibration tables or device configuration



- ❑ When larger amounts of static data are to be stored (such as in USB flash drives) a specific type of EEPROM such as flash memory is more economical than traditional EEPROM devices.
- ❑ EEPROMs are realized as arrays of floating-gate transistors.
- ❑ EEPROM is user modifiable read only memory (ROM) that can be erased and reprogrammed (written to) repeatedly through the application of higher than normal electrical voltage generated externally or internally in the case of modern EEPROMs.
- ❑ EPROM usually must be removed from the device for erasing and programming, whereas EEPROMs can be programmed and erased in circuit.

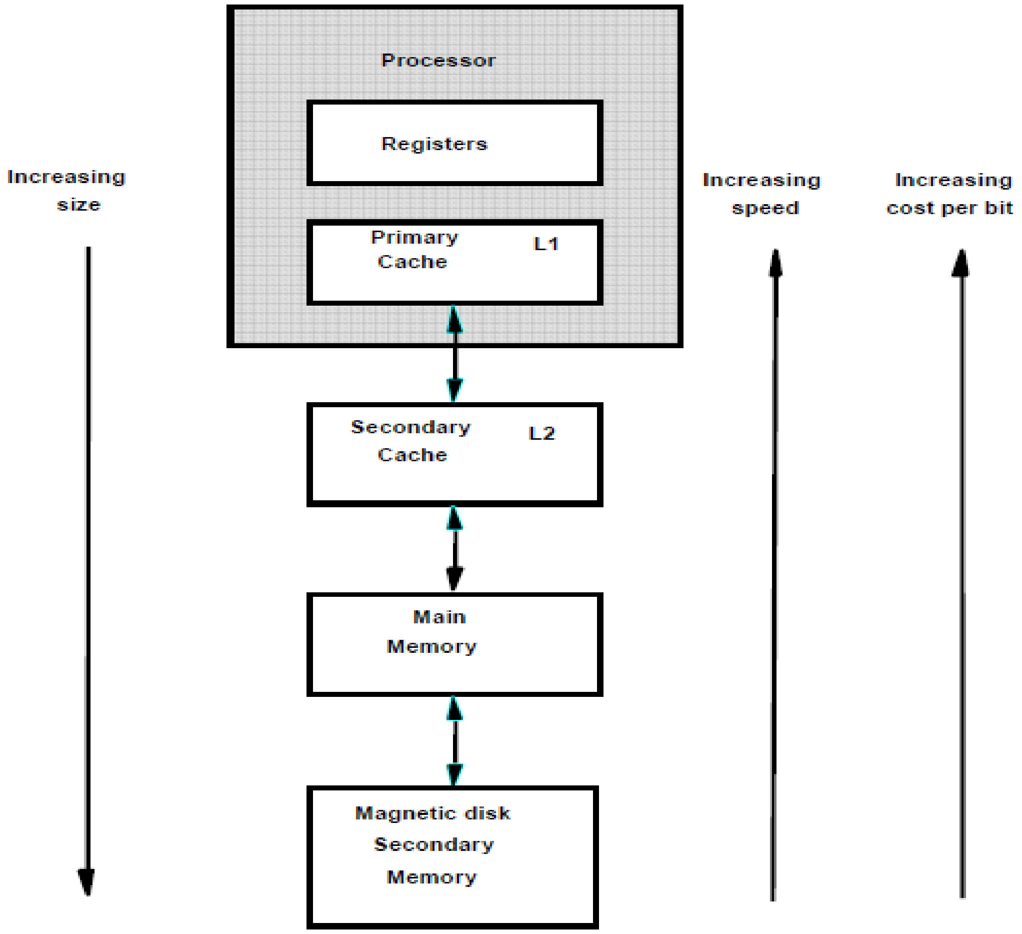


- ❑ Originally, EEPROMs were limited to single byte operations which made them slower, but modern EEPROMs allow multi-byte page operations.
- ❑ It also has a limited life - that is, the number of times it could be reprogrammed was limited to tens or hundreds of thousands of times.
- ❑ That limitation has been extended to a million write operations in modern EEPROMs.
- ❑ In an EEPROM that is frequently reprogrammed while the computer is in use, the life of the EEPROM can be an important design consideration.

Speed, Size and Cost of Memories

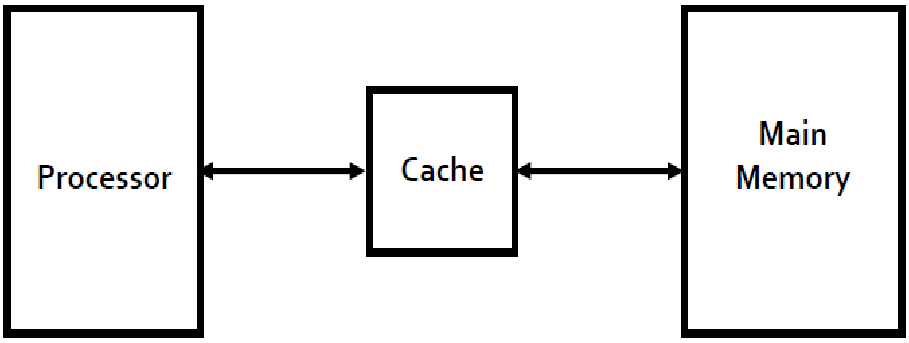
- An ideal memory would be fast, large, and inexpensive.
- A very fast memory can be implemented if SRAM chips are used.
- But these chips are expensive because their basic cells have six transistors, which preclude packing a very large number of cells onto a single chip.
- Thus, for cost reasons, it is impractical to build a large memory using SRAM chips.

- The alternative is to use Dynamic RAM chips, which have much simpler basic cells and thus are much less expensive.
- But such memories are significantly slower.
- Although dynamic memory units in the range of hundreds of megabytes can be implemented at a reasonable cost, the affordable size is still small compared to the demands of large programs with voluminous data.
- A solution is provided by using secondary storage, mainly magnetic disks, to implement large memory spaces.
- Very large disks are available at a reasonable price, and they are used extensively in computer systems.
- However, they are much slower than the semiconductor memory units.
- So A huge amount of cost-effective storage can be provided by magnetic disks.
- A large, yet affordable, main memory can be built with dynamic RAM technology.
- This leaves SRAMs to be used in smaller units where speed is of the essence, such as in cache memories.
- All of these different types of memory units are employed effectively in a computer.
- The entire computer memory can be viewed as the hierarchy depicted in the next slide.



Cache Memory

- Analysis of large number of programs has shown that a number of instructions are executed repeatedly.
- This may be in the form of a simple loops, nested loops, or a few procedures that repeatedly call each other.
- It is observed that many instructions in each of a few localized areas of the program are repeatedly executed, while the remainder of the program is accessed relatively less.
- This phenomenon is referred to as **locality of reference**.
- Now, if it can be arranged to have the active segments of a program in a fast memory, then the total execution time can be significantly reduced.
- It is the fact that CPU is a faster device and memory is a relatively slower device. Memory access is the main bottleneck for the performance efficiency.
- If a faster memory device can be inserted between main memory and CPU, the efficiency can be increased.
- The faster memory that is inserted between CPU and Main Memory is termed as Cache memory.



- The memory control circuitry is designed to take advantage of the property of locality of reference.
- Some assumptions are made while designing the memory control circuitry:
  1. The CPU does not need to know explicitly about the existence of the cache.
  2. The CPU simply makes Read and Write request. The nature of these two operations are same whether cache is present or not.
  3. The address generated by the CPU always refer to location of main memory.
  4. The memory access control circuitry determines whether or not the requested word currently exists in the cache.
- When a Read request is received from the CPU, the contents of a block of memory words containing the location specified are transferred into the cache.
- When any of the locations in this block is referenced by the program, its contents are read directly from the cache.
- The cache memory can store a number of such blocks at any given time.

- The correspondence between the Main Memory Blocks and those in the cache is specified by means of a mapping function.
- When the cache is full and a memory word is referenced that is not in the cache, a decision must be made as to which block should be removed from the cache to create space to bring the new block to the cache that contains the referenced word.
- **Replacement algorithms** are used to make the proper selection of block that must be replaced by the new one.
- When a write request is received from the CPU, there are two ways that the system can proceed. In the first case, the cache location and the main memory location are updated simultaneously.
- This is called the store through method or **write through protocol.**

Mapping Functions

- The mapping functions are used to map a particular block of main memory to a particular block of cache.
- This mapping function is used to transfer the block from main memory to cache memory.
- Three different mapping functions are available:
  1. Direct mapping:
  2. Associative mapping:
  3. Set-associative mapping:

Direct Mapping Technique:

- The simplest way of associating main memory blocks with cache block is the direct mapping technique.
- In this technique, block k of main memory maps into block k modulo m of the cache, where m is the total number of blocks in cache.
- In this example, the value of m is 128.
- In direct mapping technique, one particular block of main memory can be transferred to a particular block of cache which is derived by the modulo function.
- Thus, whenever one of the main memory blocks 0, 128, 256,… is loaded in the cache, it is stored in cache block 0. Blocks 1, 129, 257,… are stored in cache block 1, and so on.
- Since more than one main memory block is mapped onto a given cache block position, contention may arise for that position.
- This situation may occurs even when the cache is not full.
- Contention is resolved by allowing the new block to overwrite the currently resident block.

So the replacement algorithm is trivial. The detail operation of direct mapping technique is as follows:

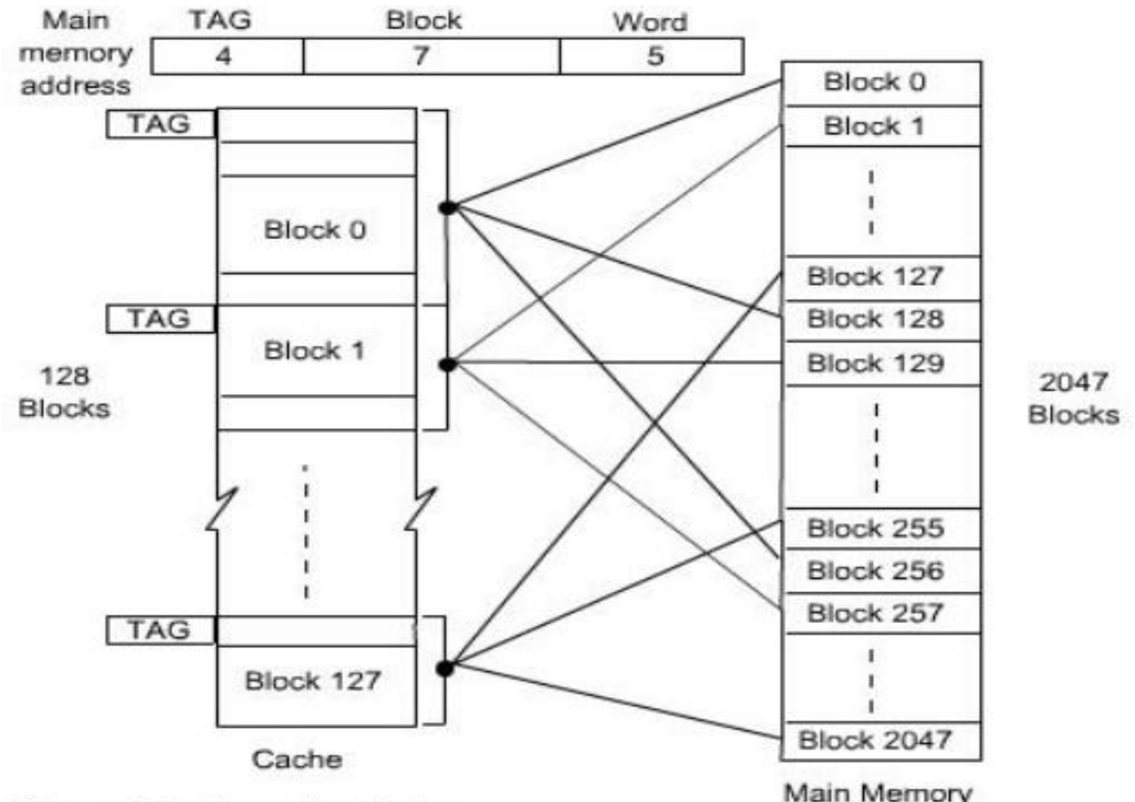


Figure : Direct-mapping Cache

- The main memory address is divided into three fields.
- The field size depends on the memory capacity and the block size of cache.
- In this example, the lower 5 bits of address is used to identify a word within a block. Next 7 bits are used to select a block out of 128 blocks (which is the capacity of the cache).
- The remaining 4 bits are used as a TAG to identify the proper block of main memory that is mapped to cache.
- When a new block is first brought into the cache, the high order 4 bits of the main memory address are stored in four TAG bits associated with its location in the cache.
- When the CPU generates a memory request, the 7-bit block address determines the corresponding cache block.
- The TAG field of that block is compared to the TAG field of the address.
- If they match, the desired word specified by the low-order 5 bits of the address is in that block of the cache.
- If there is no match, the required word must be accessed from the main memory, that is, the contents of that block of the cache is replaced by the new block that is specified by the new address generated by the CPU and correspondingly the TAG bit will also be changed by the high order 4 bits of the address.

Associated Mapping Technique:

- In the associative mapping technique, a main memory block can potentially reside in any cache block position.

- In this case, the main memory address is divided into two groups, low-order bits identifies the location of a word within a block and high-order bits identifies the block.
- In the example here, 11 bits are required to identify a main memory block when it is resident in the cache , high-order 11 bits are used as TAG bits and low-order 5 bits are used to identify a word within a block.
- The TAG bits of an address received from the CPU must be compared to the TAG bits of each block of the cache to see if the desired block is present.

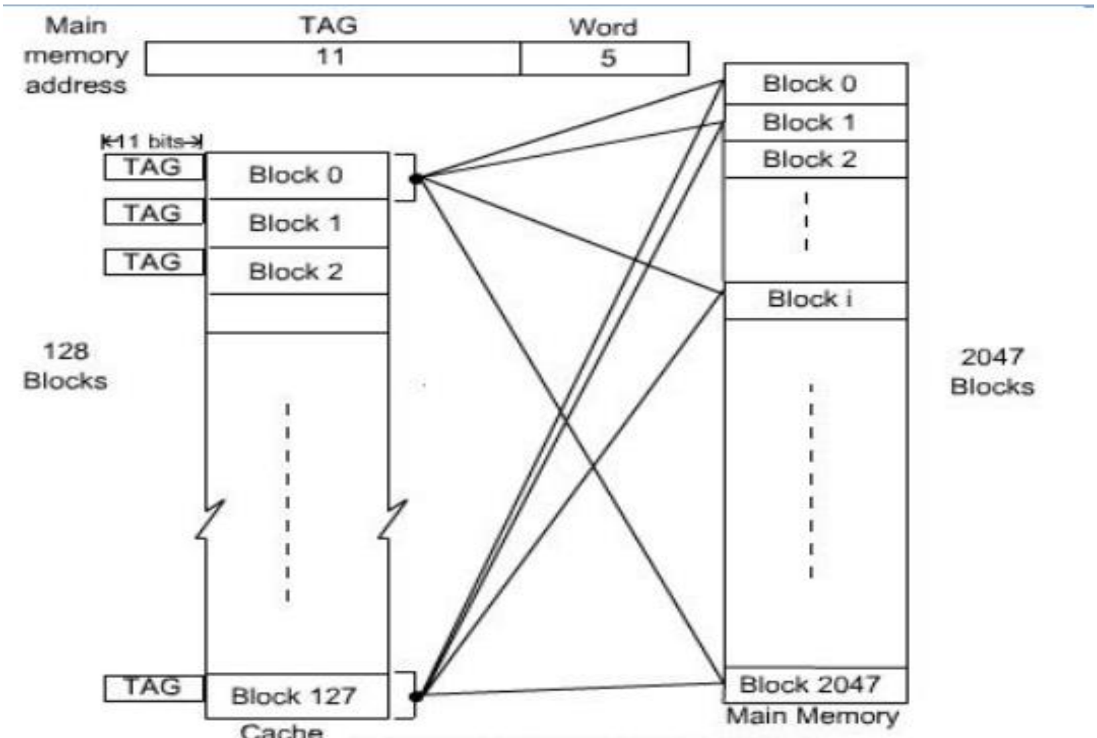


Figure : Associated-mapping Cache

- In the associative mapping, any block of main memory can go to any block of cache, so it has got the complete flexibility and we have to use proper replacement policy to replace a block from cache if the currently accessed block of main memory is not present in cache.
- It might not be practical to use this complete flexibility of associative mapping technique due to searching overhead, because the TAG field of main memory address has to be compared with the TAG field of all the cache block.
- In this example, there are 128 blocks in cache and the size of TAG is 11 bits.

Set-Associative Mapping Technique:

- A combination of the direct- and associative mapping techniques can be used.
- Blocks of the cache are grouped into sets, and the mapping allows a block of main memory to reside in any block of a specific set.
- Therefore, the flexibility of associative mapping is reduced from full freedom to a set of specific blocks.

- This also reduces the searching overhead, because the search is restricted to number of sets, instead of number of blocks.
- Also the contention problem of the direct mapping is eased by having a few choices for block replacement.
- Consider the same cache memory and main memory organization of the previous example.
- Organize the cache with 4 blocks in each set.
- The TAG field of associative mapping technique is divided into two groups, one is termed as SET bit and the second one is termed as TAG bit.

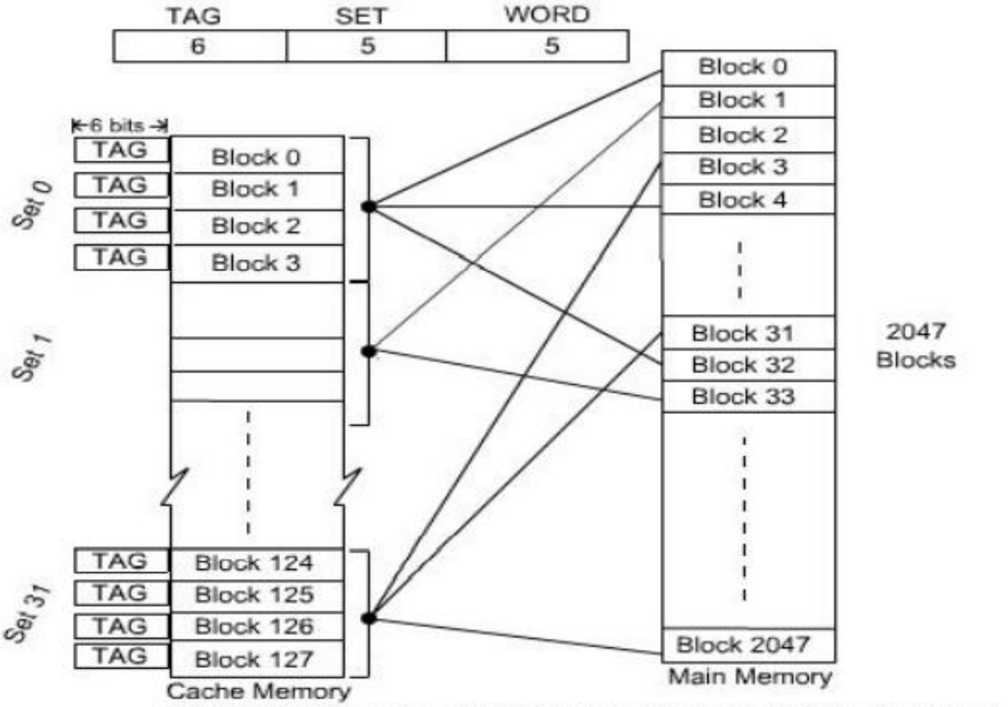


Figure : Block-set-Associated mapping Cache with 4 blocks per set

- Since each set contains 4 blocks, total number of set is 32.
- The main memory address is grouped into three parts: low-order 5 bits are used to identifies a word within a block.
- Since there are total 32 sets present, next 5 bits are used to identify the set. High-order 6 bits are used as TAG bits.
- The 5-bit set field of the address determines which set of the cache might contain the desired block.
- This is similar to direct mapping technique, in case of direct mapping, it looks for block, but in case of set-associative mapping, it looks for set.
- The TAG field of the address must then be compared with the TAGs of the four blocks of that set.
- If a match occurs, then the block is present in the cache; otherwise the block containing the addressed word must be brought to the cache.
- This block will potentially come to the corresponding set only.

- Since, there are four blocks in the set, we have to choose appropriately which block to be replaced if all the blocks are occupied.
- Since the search is restricted to four block only, so the searching complexity is reduced.
- It is clear that if we increase the number of blocks per set, then the number of bits in SET field is reduced.
- Due to the increase of blocks per set, complexity of search is also increased.
- The extreme condition of 128 blocks per set requires no set bits and corresponds to the fully associative mapping technique with 11 TAG bits.
- The other extreme of one block per set is the direct mapping method

Replacement Algorithms

- When a new block must be brought into the cache and all the positions that it may occupy are full, a decision must be made as to which of the old blocks is to be overwritten.
- In general, a policy is required to keep the block in cache when they are likely to be referenced in near future.
- However, it is not easy to determine directly which of the block in the cache are about to be referenced.
- The property of locality of reference gives some clue to design good replacement policy.

Least Recently Used (LRU) Replacement policy:

- When a block is to be overwritten, it is a good decision to overwrite the one that has gone for longest time without being referenced.
- This is defined as the least recently used (LRU) block.

For Example:

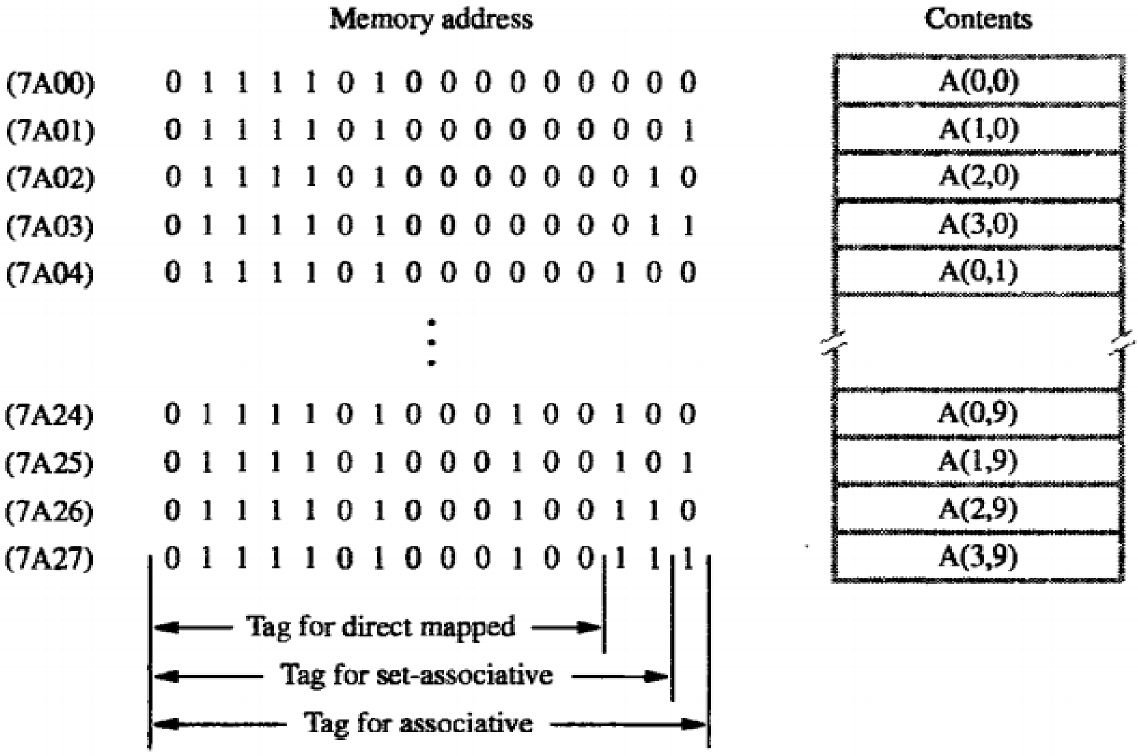
- Consider a specific cache with a four-block set.
- It is required to track the LRU block of this four-block set.
- A 2-bit counter may be used for each block.
- When a hit occurs, that is, when a read request is received for a word that is in the cache, the counter of the block that is referenced is set to 0.
- All counters which values originally lower than the referenced one are incremented by 1 and all other counters remain unchanged.
- When a miss occurs, that is, when a read request is received for a word and the word is not present in the cache, we have to bring the block to cache.
- There are two possibilities
    - In case of a miss: If the set is not full, the counter associated with the new block loaded from the main memory is set to 0, and the values of all other counters are incremented by 1.
    - If the set is full and a miss occurs, the block with the counter value 3 is removed , and the new block is put in its place.
- The counter value is set to zero. The other three block counters are incremented by 1.
- It is easy to verify that the counter values of occupied blocks are always distinct.
- Also it is trivial that highest counter value indicates least recently used block.

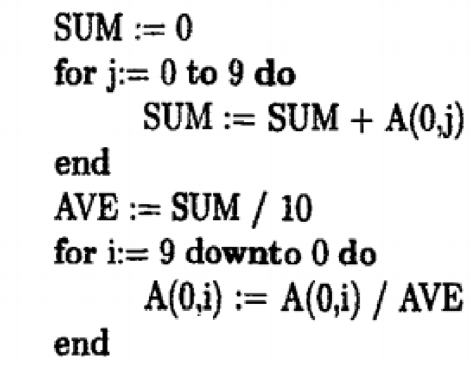
First In First Out (FIFO) replacement policy:

- A reasonable rule may be to remove the oldest from a full set when a new block must be brought in.
- While using this technique, no updation is required when a hit occurs.
- When a miss occurs and the set is not full, the new block is put into an empty block and the counter values of the occupied block will be increment by one.
- When a miss occurs and the set is full, the block with highest counter value is replaced by new block and counter is set to 0, counter value of all other blocks of that set is incremented by 1.
- The overhead of the policy is less, since no updation is required during hit.

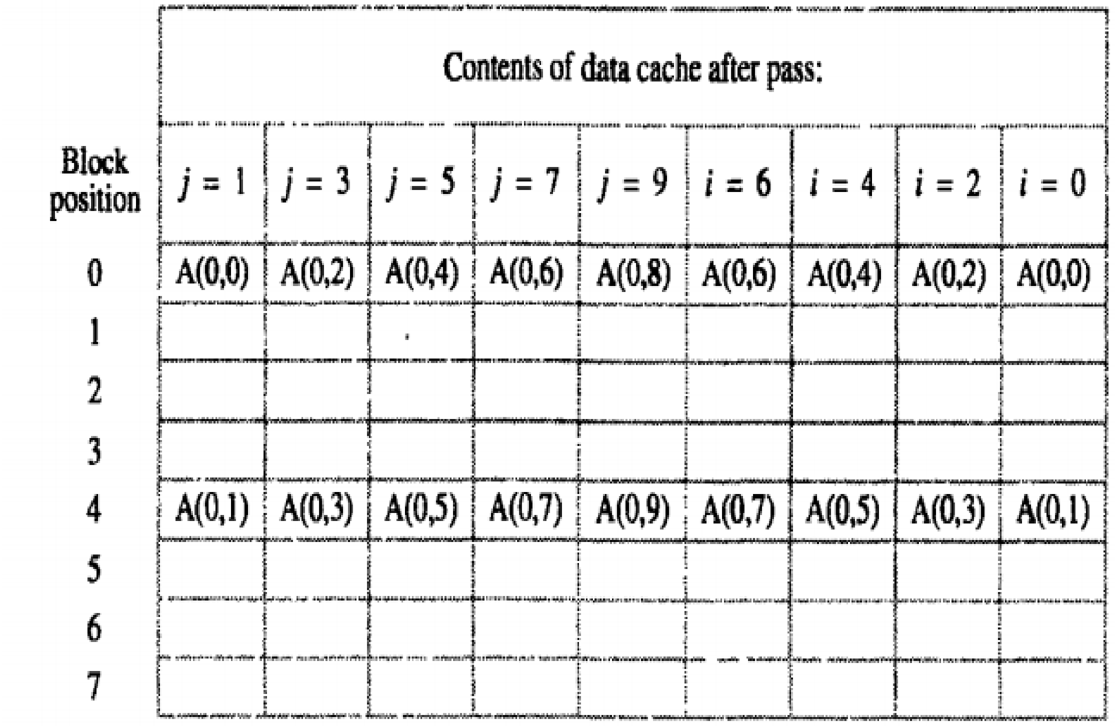
Random replacement policy:

- The simplest algorithm is to choose the block to be overwritten at random.
- Interestingly enough, this simple algorithm has been found to be very effective in practice.


Memory address
Contents
(7A00) 0 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0
(7A01) 0 1 1 1 1 0 1 0 0 0 0 0 0 0 0 1
(7A02) 0 1 1 1 1 0 1 0 0 0 0 0 0 0 1 0
(7A03) 0 1 1 1 1 0 1 0 0 0 0 0 0 0 1 1
(7A04) 0 1 1 1 1 0 1 0 0 0 0 0 0 1 0 0
(7A24) 0 1 1 1 1 0 1 0 0 0 1 0 0 1 0 0
(7A25) 0 1 1 1 1 0 1 0 0 0 1 0 0 1 0 1
(7A26) 0 1 1 1 1 0 1 0 0 0 1 0 0 1 1 0
(7A27) 0 1 1 1 1 0 1 0 0 0 1 0 0 1 1 1
A(0,0)
A(1,0)
A(2,0)
A(3,0)
A(0,1)
A(0,9)
A(1,9)
A(2,9)
A(3,9)
Tag for direct mapped
Tag for set-associative
Tag for associative

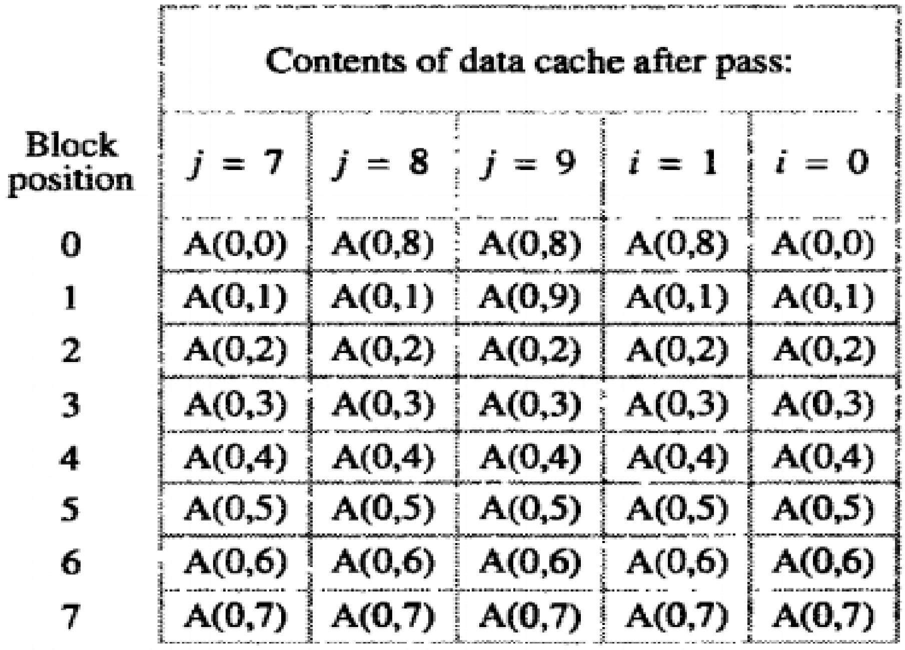


```
SUM := 0
for j:= 0 to 9 do
        SUM := SUM + A(0,j)
end
AVE := SUM / 10
for i:= 9 downto 0 do
        A(0,i) := A(0,i) / AVE
end
```



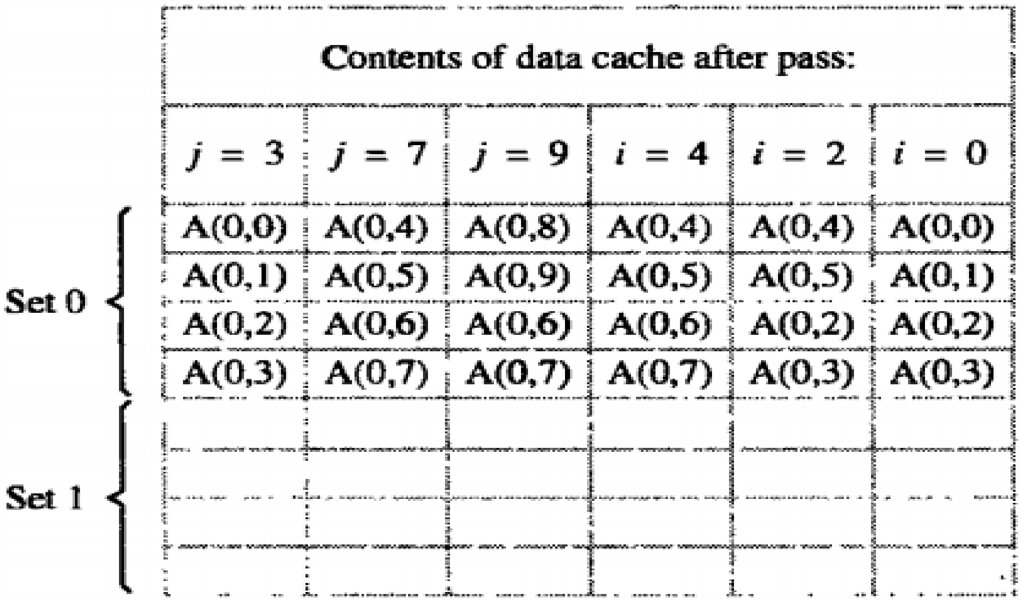
	Contents of data cache after pass:								
Block position	*j* = 1	*j* = 3	*j* = 5	*j* = 7	*j* = 9	*i* = 6	*i* = 4	*i* = 2	*i* = 0
0	A(0,0)	A(0,2)	A(0,4)	A(0,6)	A(0,8)	A(0,6)	A(0,4)	A(0,2)	A(0,0)
1									
2									
3									
4	A(0,1)	A(0,3)	A(0,5)	A(0,7)	A(0,9)	A(0,7)	A(0,5)	A(0,3)	A(0,1)
5									
6									
7									

Fig: **Contents of direct mapped data cache**



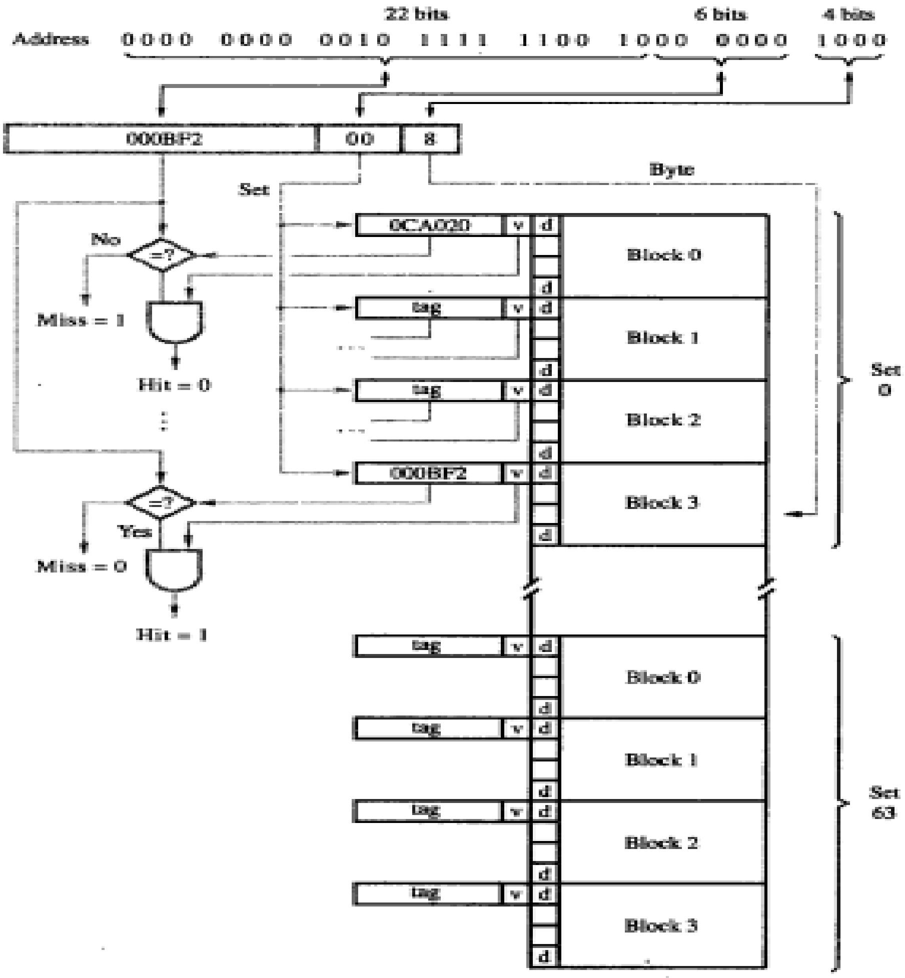
	Contents of data cache after pass:				
Block position	$j = 7$	$j = 8$	$j = 9$	$i = 1$	$i = 0$
0	A(0,0)	A(0,8)	A(0,8)	A(0,8)	A(0,0)
1	A(0,1)	A(0,1)	A(0,9)	A(0,1)	A(0,1)
2	A(0,2)	A(0,2)	A(0,2)	A(0,2)	A(0,2)
3	A(0,3)	A(0,3)	A(0,3)	A(0,3)	A(0,3)
4	A(0,4)	A(0,4)	A(0,4)	A(0,4)	A(0,4)
5	A(0,5)	A(0,5)	A(0,5)	A(0,5)	A(0,5)
6	A(0,6)	A(0,6)	A(0,6)	A(0,6)	A(0,6)
7	A(0,7)	A(0,7)	A(0,7)	A(0,7)	A(0,7)

Fig: **Contents of an associative mapped data cache**



	Contents of data cache after pass:					
	$j = 3$	$j = 7$	$j = 9$	$i = 4$	$i = 2$	$i = 0$
Set 0	A(0,0)	A(0,4)	A(0,8)	A(0,4)	A(0,4)	A(0,0)
	A(0,1)	A(0,5)	A(0,9)	A(0,5)	A(0,5)	A(0,1)
	A(0,2)	A(0,6)	A(0,6)	A(0,6)	A(0,2)	A(0,2)
	A(0,3)	A(0,7)	A(0,7)	A(0,7)	A(0,3)	A(0,3)
Set 1						

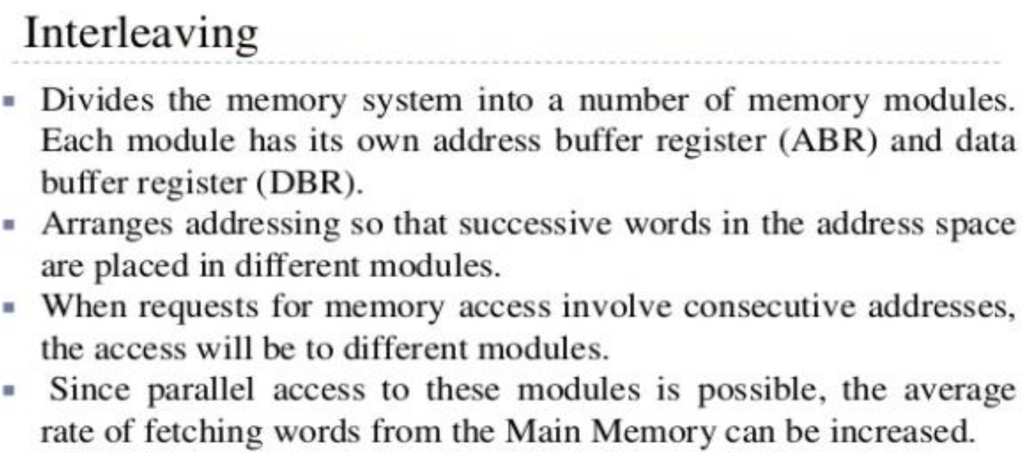
Fig: **Contents of an set-associative mapped data cache**



- Other than set bits, tag bits there is a control bits **Valid Bit** and **Dirty Bit** provided for each block**.**
- Valid bit indicates whether the block contains valid data or not.
- Dirty bit indicates whether the block has been modified during its cache residency.
- Valid bits all set to 0, when power is initially applied to the system or when main memory is loaded with new programs and data from the disk.
- Transfers from the disk to main memory is carried out by DMA mechanism.
- The valid bit of a particular cache block is set to 1,the first time this block is loaded from the main memory.

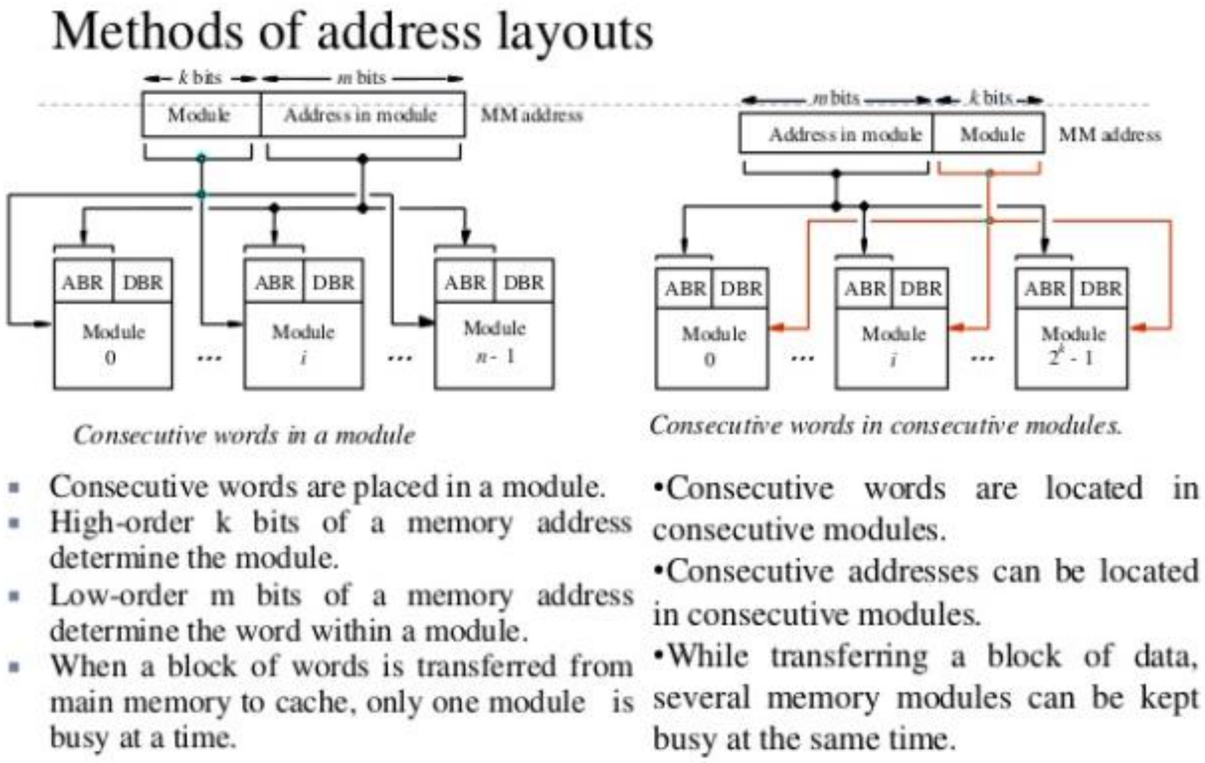
Performance Considerations

- Two key factors in the commercial success of a computer are performance and cost; the best possible performance at the lowest cost is the objective.
- The challenge in considering design alternatives is to improve the performance without increasing the cost.
- A common measure of success is the price/performance ratio



Interleaving

- Divides the memory system into a number of memory modules. Each module has its own address buffer register (ABR) and data buffer register (DBR).
- Arranges addressing so that successive words in the address space are placed in different modules.
- When requests for memory access involve consecutive addresses, the access will be to different modules.
- Since parallel access to these modules is possible, the average rate of fetching words from the Main Memory can be increased.



Methods of address layouts

Consecutive words in a module

Consecutive words in consecutive modules.

- Consecutive words are placed in a module.
- High-order k bits of a memory address determine the module.
- Low-order m bits of a memory address determine the word within a module.
- When a block of words is transferred from main memory to cache, only one module is busy at a time.

•Consecutive words are located in consecutive modules.

•Consecutive addresses can be located in consecutive modules.

•While transferring a block of data, several memory modules can be kept busy at the same time.

Access Time Reduction:

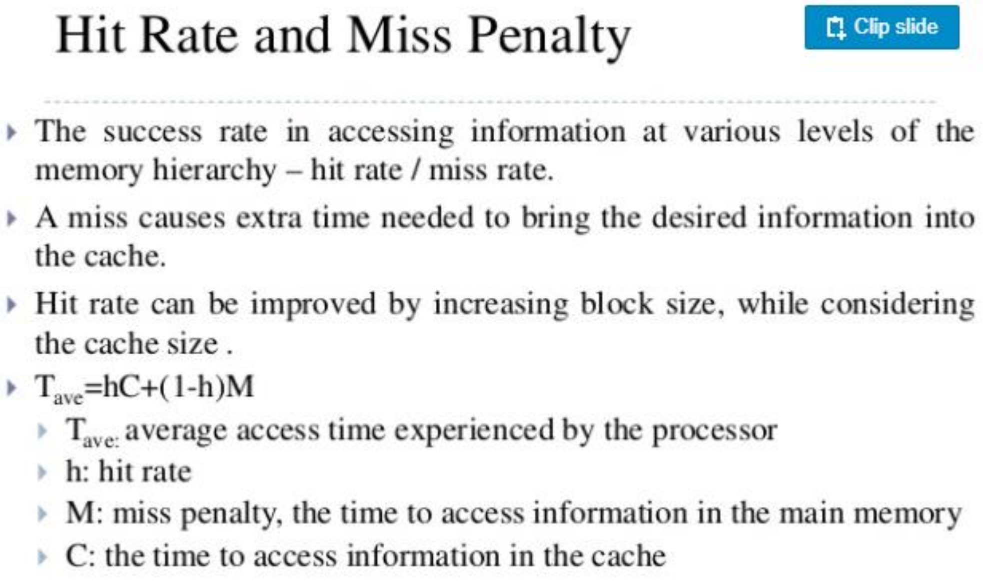
- Consider the time needed to transfer a block of data from the main memory to the cache when a read miss occurs. ‹
- Suppose that a cache with 8-word blocks is used.

- On a read miss, the block that contains the desired word must be copies from the memory into the cache .
- Assume that the hardware has the following properties.
- It takes one clock cycle to send an address to the main memory.
- The main memory allows the first word to be accessed in 8 cycles, but subsequent words of the block area accessed in 4 cycles per word ‹
- Also, one clock cycle is needed to send one block to the cache.
- If a single memory module is used, then the time needed to load the desired block into the cache is ‹ 1+8+(7x4)+1=38 cycles
- Suppose now that the memory is constructed as four interleaved modules, using the scheme shown above.
- When the starting address of the block arrives at the memory, all four modules begin accessing the required data, using the high-order bits of the address.
- After 8 cycles, each module has one word of data in its data buffer register (DBR).
- These words are transferred to the cache, one word at a time, during the next 4 cycles.
- During this time, the next word in each module is accessed.
- Then it takes another 4 cycles to transfer these words to the cache.

Therefore, the total time needed to load the block from the interleaved memory is ‹ 1+8+4+4=17 cycles

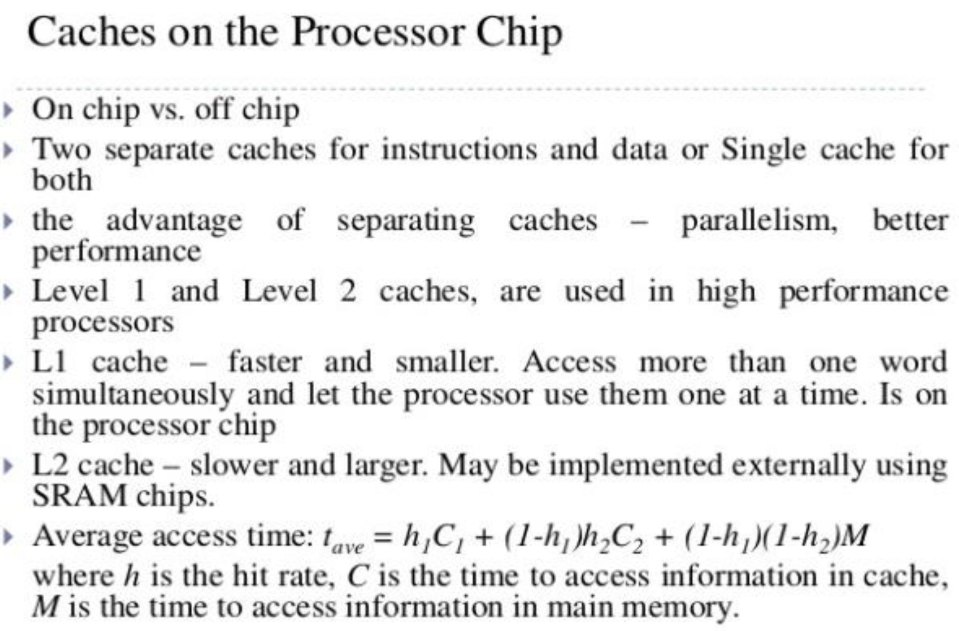
Hit Rate and Miss Penalty

- A successful access to data in a cache is called a hit.
- The number of hits stated as a fraction of all attempted accesses is called the hit rate, and the miss rate is the number of misses stated as a fraction of attempted accesses.
- High hit rates, well over 0.9, are essential for high-performance computers.
- Performance is adversely affected by the actions that must be taken after a miss.
- The extra time needed to bring the desired information into the cache is called the miss penalty



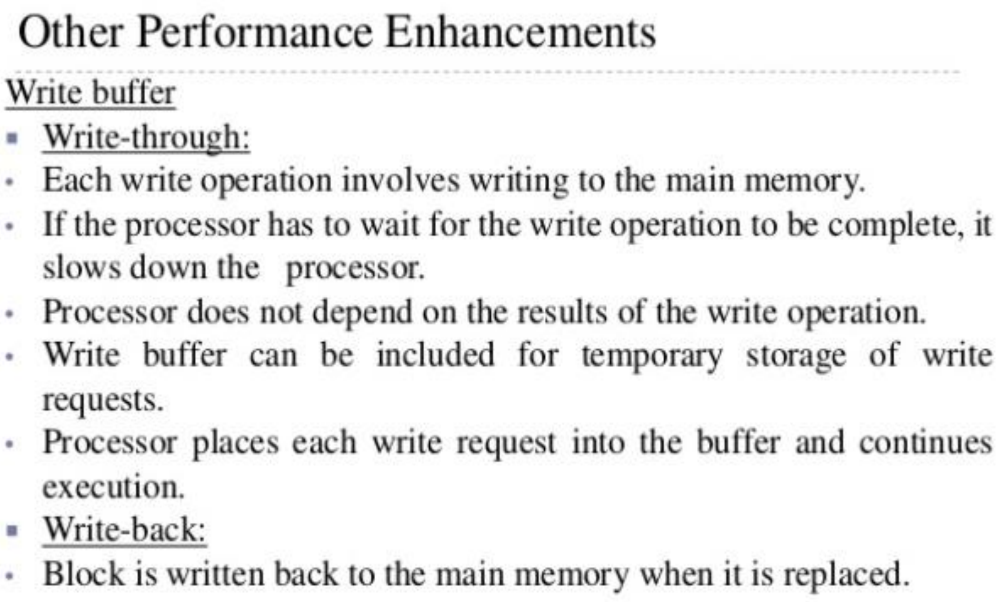
Hit Rate and Miss Penalty

- The success rate in accessing information at various levels of the memory hierarchy – hit rate / miss rate.
- A miss causes extra time needed to bring the desired information into the cache.
- Hit rate can be improved by increasing block size, while considering the cache size .
- T_{ave}=hC+(1-h)M
  - $T_{ave:}$ average access time experienced by the processor
  - h: hit rate
  - M: miss penalty, the time to access information in the main memory
  - C: the time to access information in the cache



Caches on the Processor Chip

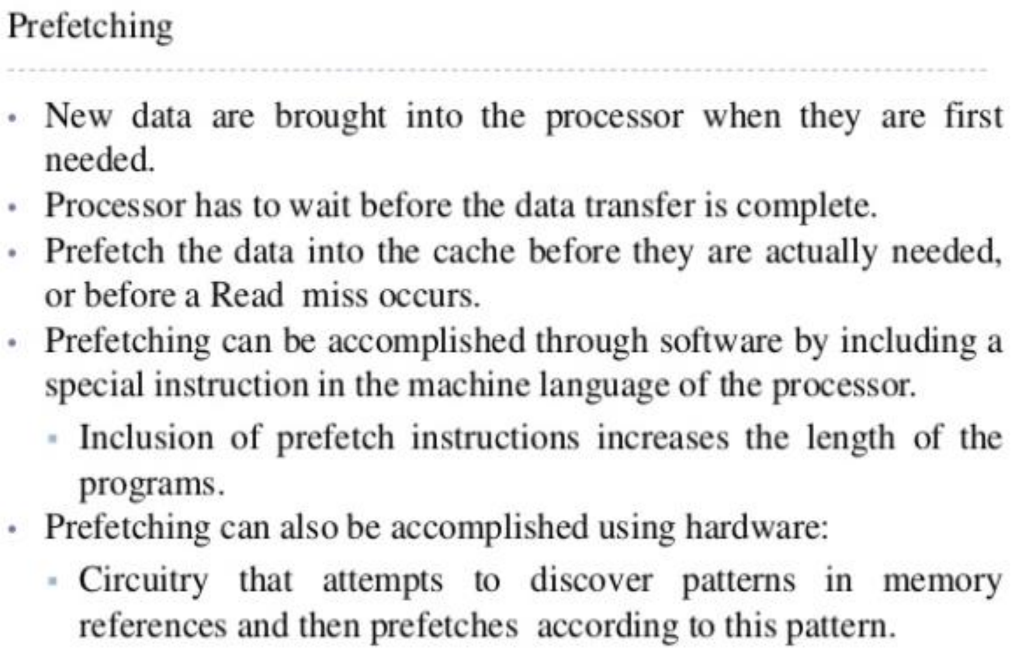
- On chip vs. off chip
- Two separate caches for instructions and data or Single cache for both
- the advantage of separating caches – parallelism, better performance
- Level 1 and Level 2 caches, are used in high performance processors
- L1 cache – faster and smaller. Access more than one word simultaneously and let the processor use them one at a time. Is on the processor chip
- L2 cache – slower and larger. May be implemented externally using SRAM chips.
- Average access time: $t_{ave} = h_1C_1 + (1\text{-}h_1)h_2C_2 + (1\text{-}h_1)(1\text{-}h_2)M$ where h is the hit rate, C is the time to access information in cache, M is the time to access information in main memory.



Other Performance Enhancements

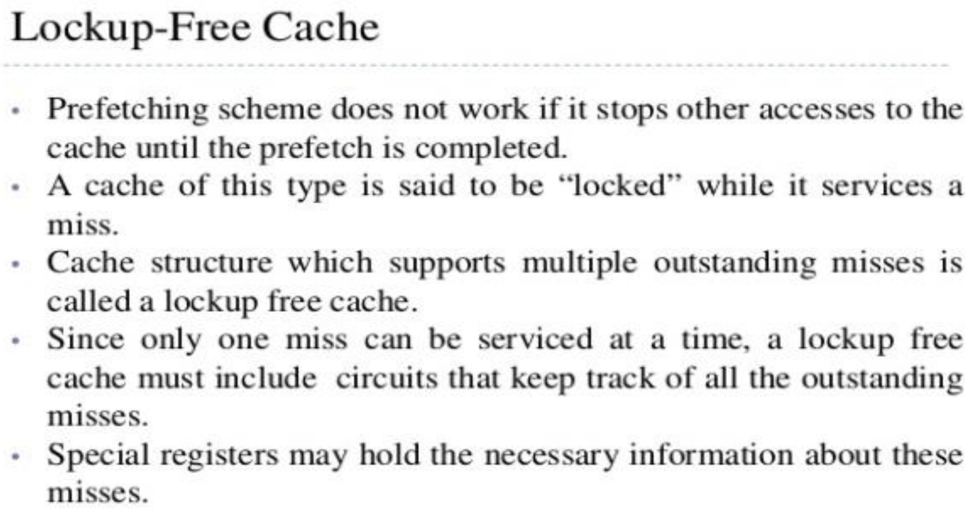
Write buffer

- Write-through:
- Each write operation involves writing to the main memory.
- If the processor has to wait for the write operation to be complete, it slows down the processor.
- Processor does not depend on the results of the write operation.
- Write buffer can be included for temporary storage of write requests.
- Processor places each write request into the buffer and continues execution.
- Write-back:
- Block is written back to the main memory when it is replaced.



Prefetching

- New data are brought into the processor when they are first needed.
- Processor has to wait before the data transfer is complete.
- Prefetch the data into the cache before they are actually needed, or before a Read miss occurs.
- Prefetching can be accomplished through software by including a special instruction in the machine language of the processor.
  - Inclusion of prefetch instructions increases the length of the programs.
- Prefetching can also be accomplished using hardware:
  - Circuitry that attempts to discover patterns in memory references and then prefetches according to this pattern.



Lockup-Free Cache

- Prefetching scheme does not work if it stops other accesses to the cache until the prefetch is completed.
- A cache of this type is said to be “locked” while it services a miss.
- Cache structure which supports multiple outstanding misses is called a lockup free cache.
- Since only one miss can be serviced at a time, a lockup free cache must include circuits that keep track of all the outstanding misses.
- Special registers may hold the necessary information about these misses.

Virtual Memory

Paging is a method of writing and reading data from a secondary storage(Drive) for use in primary storage(RAM). When a computer runs out of RAM, the operating system (OS) will move pages of memory over to the computer’s hard disk to free up RAM for other processes. This ensures that the operating system will never run out of memory and crash. Too much reliance on memory paging can impair performance, however, because RAM operates much faster than disk memory. This means the operating system has to wait for the disk to catch up every time a page is swapped; The concept of paging helps us to develop truly effective multiprogramming systems. Since a process need not be loaded into contiguous memory locations, it helps us to put a page of a process in any free page frame. On the other hand, it is not required to load the whole process to the main memory, because the execution may be confined to a small section of the program. (eg. a subroutine).It would clearly be wasteful to load in many pages for a process when only a few pages will be used before the program is suspended. Instead of loading all the pages of a process, each page of process is brought in only when it is needed, i.e on demand. This scheme is known as **demand paging.**

Demand paging also allows us to accommodate more process in the main memory, since we are not going to load the whole process in the main memory. Pages will be brought into the main memory as and when it is required. This concept leads us to an important consequence – It is possible for a process to be larger than the size of main memory. So, while developing a new process, it is not required to look for the main memory available in the machine. Because, the process will be divided into pages and pages will be brought to memory on demand. Because a process executes only in main memory, so the main memory is referred to as real memory or physical memory. A programmer or user perceives a much larger memory that is allocated on the disk. This memory is referred to as **virtual memory**.

The program enjoys a huge virtual memory space to develop his or her program or software. The execution of a program is the job of operating system and the underlying hardware. To improve the performance some special hardware is added to the system. This hardware unit is known as **Memory Management Unit (MMU).** In paging system, we make a page table for the process. Page table helps us to find the physical address from virtual address. The virtual address space is used to develop a process. The special hardware unit , called Memory Management Unit (MMU) translates virtual address to physical address. When the desired data is in the main memory, the CPU can work with these data. If the data are not in the main memory, the MMU causes the operating system to bring into the memory from the disk.

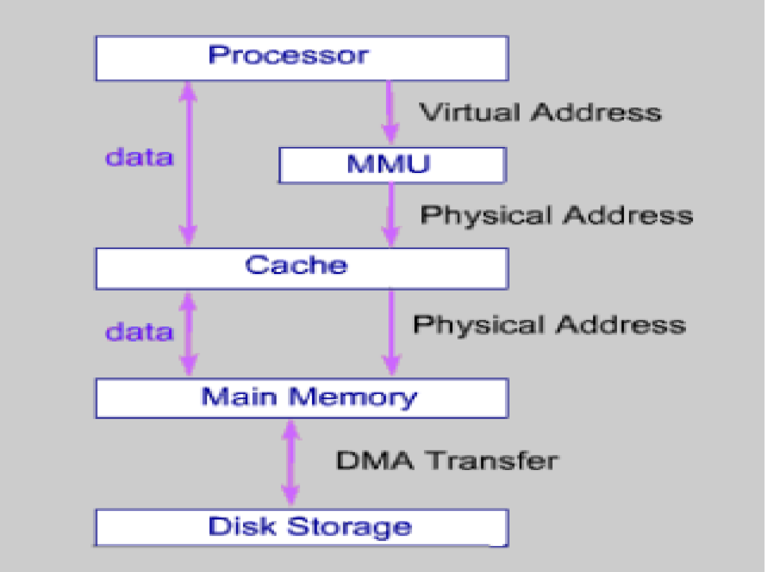


Figure : Virtual Memory Organization

Address Translation

The basic mechanism for reading a word from memory involves the translation of a virtual or logical address, consisting of page number and offset, into a physical address, consisting of frame number and offset, using a page table. There is one page table for each process. But each process can occupy huge amount of virtual memory. But the virtual memory of a process cannot go beyond a certain limit which is restricted by the underlying hardware of the MMU. One of such component may be the size of the virtual address register. The sizes of pages are relatively small and so the size of page table increases as the size of process increases. Therefore, size of page table could be unacceptably high. To overcome this problem, most virtual memory scheme store page table in virtual memory rather than in real memory. When a process is running, at least a part of its page table must be in main memory, including the page table entry of the currently executing page.

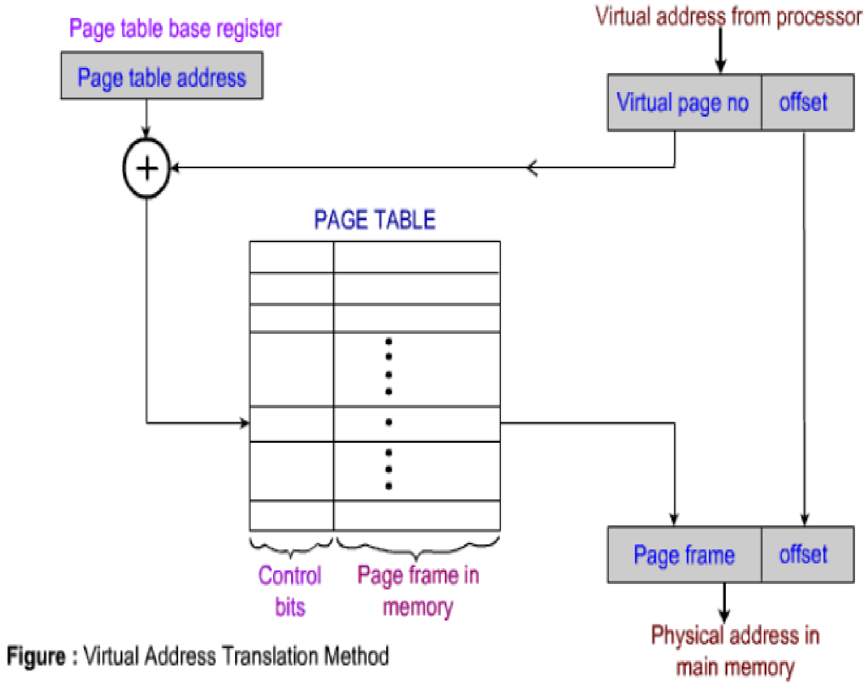


Figure : Virtual Address Translation Method

Each virtual address generated by the processor is interpreted as virtual page number (high order list) followed by an offset (lower order bits) that specifies the location of a particular word within a page. Information about the main memory location of each page kept in a page table.

Translation Look aside Buffer (TLB)

Every virtual memory reference can cause two physical memory accesses. One to fetch the appropriate page table entry One to fetch the desired data. Thus a straight forward virtual memory scheme would have the effect of doubling the memory access time. To overcome this problem, most virtual memory schemes make use of a special cache for page table entries, usually called Translation Lookaside Buffer (TLB). This cache functions in the same way as a memory cache and contains those page table entries that have been most recently used. In addition to the information that constitutes a page table entry, the TLB must also include the virtual address of the entry

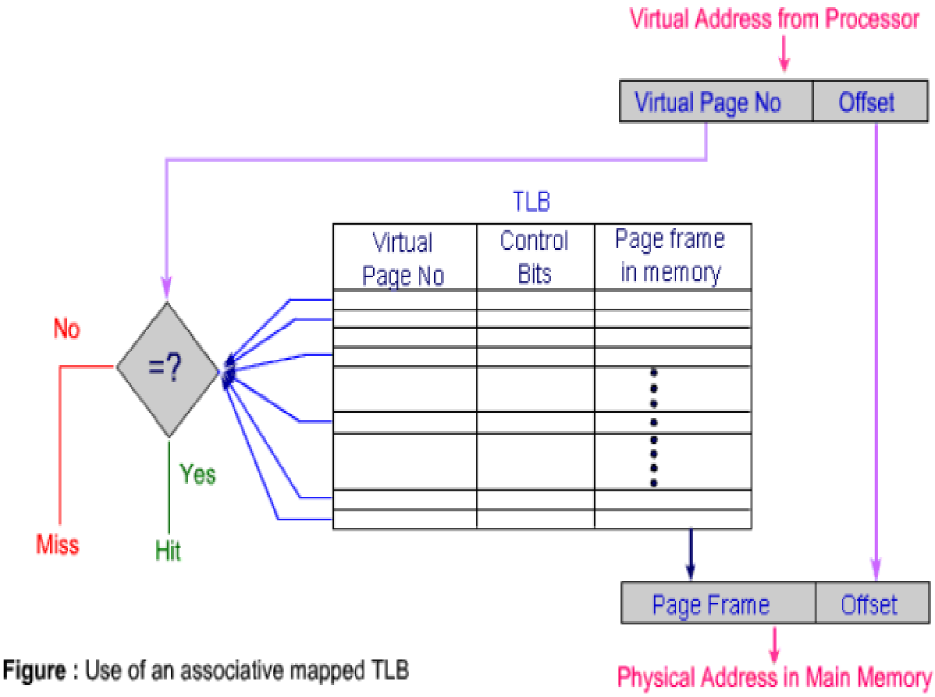


Figure : Use of an associative mapped TLB

Set-associative mapped TLBs are also found in commercial products. An essential requirement is that the contents of the TLB be coherent with the contents of the page table in the main memory. When the operating system changes the contents of the page table it must simultaneously invalidate the corresponding entries in the TLB. One of the control bits in the TLB is provided for this purpose.

Address Translation proceeds as follows:

Given a virtual address, the MMU looks in the TLB for the reference page. If the page table entry for this page is found in the TLB, the physical address is obtained immediately. If there is a miss in the TLB, then the required entry is obtained from the page table in the main memory and the TLB is updated. When a program generates an access request to a page that is not in the main memory, a page fault is said to have occurred. The whole page must be brought from the disk into the memory before access can proceed. When it detects a page fault, the MMU asks the operating system to intervene by raising an exception. (interrupt). Processing of active task is interrupted, and control is transferred to the operating system. The operating system then copies the requested page from the disk into the main memory and returns control to the interrupted task. Because a long delay occurs due to a page transfer takes place, the operating system may suspend execution of the task that caused the page fault and begin execution of another task whose page are in the main memory.