MASTER OF COMPUTER APPLICATION

MCA-24

COMPUTER SYSTEM ARCHITECTURE

Directorate of Distance Education
Guru Jambheshwar University of Science & Technology
Hisar - 125001

CONTENTS

SUBJECT: COMPUTER SYSTEM ARCHITECTURE	
COURSE CODE: MCA-24	**AUTHOR: DR. MANOJ DUHAN**
LESSON NO. 1	

DIGITAL LOGIC GATES

REVISED / UPDATED SLM BY NEERAJ VERMA

STRUCTURE

1.0 LEARNING OBJECTIVES

After reading this lesson, you should be able to describe the operations and construct the truth tables for the AND, OR, NOT (INVERTER), NAND, NOR and the Exclusive-OR circuits.

1.1 INTRODUCTION

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.

Binary logic deals with binary variables and with operations that assume a logical meaning. It is used to describe, in algebraic or tabular form, the manipulation and processing of binary information. 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.

1.2 ANALOG AND DIGITAL SIGNALS

A signal is an electromagnetic or electrical current that carries data from one system or network to another. In electronics, a signal is often a time-varying voltage that is also an electromagnetic wave carrying information, though it can take on other forms, such as

current. There are two main types of signals used in electronics: analog and digital signals.

1.2.1 ANALOG SIGNALS

We are very familiar with analog signals. The reading of a moving coil or moving iron voltmeter and ammeter, dynamometer wattmeter etc., are all analog quantities. The trace on a CRO screen is also analog. Analog methods for communication system have long been in use. Frequency division multiplexing is the means of analog communication. An electronic amplifier is an analog circuit. The low level analog signal (audio, video, etc.) is amplified to provide strength to the signal. Analog circuit systems (position control, process control) have been in use for the past many decades. Analog Computers use voltages, resistances and potentiometric rotations to represent the numbers and perform arithmetic operations. Analog differentiation, integration, etc., is also done. Operational amplifier is a very versatile analog electronic circuit used to perform a variety of operations (addition, subtraction, multiplication, division, exponentiation, differentiation, integration etc.). Analog integrated circuits are widely used in electronic industry.

1.2.2 DIGITAL SIGNALS

The term digital is derived from digits. Any device or system which works on digits is a digital device or system. A digital voltmeter indicates the value of voltage in the form of digits, e.g., 230.25. Reading an analog instrument introduces human error and also requires more time. A digital reading is more accurate, eliminates human error and can be read quickly.

Communication systems have also gone digital. The initial signal waveform is always analog. To use digital transmission, the signal waveform is sampled and the digital representation transmitted. The process of converting analog signal to digital form is also known as digitizing. For multiple channels of transmission, Time Division Multiplexing is used.

Digital control systems are fast replacing analog control systems. In digital control systems the error is in the form of digital pulses. Digital computers have revolutionalized the concept of computers. Their capability ranges from simple calculations to complex calculations using numerical techniques. Many computing tasks which required hours and days take only a few minutes on digital computers.

Digital signal processing is concerned with the representation of continuous time (analog) signals in digital form. It is based on Claude Shannon's sampling theorem which states that "A band limited continuous time signal can be reconstructed in its entirety from a sequence of samples taken at intervals of less than $\frac{1}{2f_N}$ where f_N is the highest frequency present in the signal." It is essential that the analog signal is band limited which limits how much it can change between samples. The sampling rate has to high to be ensure accuracy.

Since the initial signal is always analog and the final required signal is also mostly analog, a digital system requires three essential aspects (1) conversion of analog signal to digital form (2) transmission of digital signal (3) reconstruction of analog signal from the received digital signal as shown in Fig. 1.1

A continuous time function x(t) is converted into a digital signal x(n) by an analog to digital (A/D) converter. The output of discrete time system is y(n) and is converted to continuous time function by digital to analog (D/A) converter. The discrete time system, in digital communications, is a digital communication channel. To achieve high fidelity, the sampling rate may have to be very high say 50000 samples per second. Each sample may be encoded by (say) 18 bits. The frequency f_s (in Fig. 1.1) must be more than twice f_N the highest frequency in the analog signal. Very large scale integration (VLSI) digital circuits have capability to sample at very fast rate so that high fidelity is achieved.

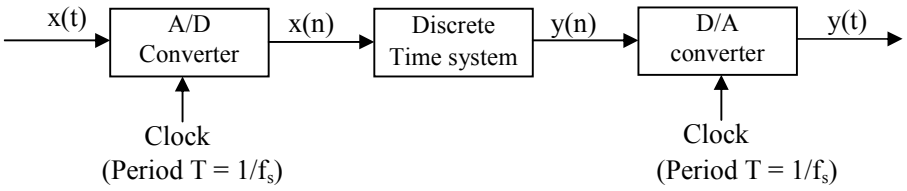


Fig. 1.1 Digital system

A DSP (digital signal processing) chip is the core of digital system used in cellular phones, modems, disk drives, digital automotive systems etc. It was invented only about 15 years ago but its applications have grown tremendously.

Digital methods have the following advantages over analog methods:

1. Digital devices work only in two states (say on and off). Thus their operation is very simple and reliable.

2. Digital display is very accurate and can be read at a fast speed. Human error is eliminated.
3. Electronic components exhibit change in behaviour due to ageing, change of ambient temperature etc. Therefore, the behaviour of analog circuits tends to be somewhat unpredictable. However, digital circuits are free from these defects.
4. Digital ICs are very cheap and compact in size.
5. Variety of digital ICs are available.
6. Power requirement of digital circuits is very low.
7. Digital systems have the characteristic advantage of memory. Thus information can be stored over a period of time. The space required for this stage is very small. One compact disc can store information contained in many books.
8. Digital systems have high fidelity and provide noise free operations.
9. By integrating system peripheral functions on a DSP chip, the reliability can be enhanced and cost reduced.
10. When volumes are high, they can be manufactured at low cost.
11. The same digital system can be used with a variety of software for a number of tasks.
12. Standardisation & Repeatability.

1.3 BASIC DIGITAL CIRCUITS

In a digital system there are only a few basic operations performed, irrespective of the complexities of the system. These operations may be required to be performed a number of times in a large digital system like digital computer or a digital control system, etc. The basic operations are AND, OR, NOT, and FLIP-FLOP. The AND, OR, and NOT operations are discussed here and the FLIP-FLOP, which is a basic memory element used to store binary information (one bit is stored in one FLIP-FLOP).

1.3.1 AND OPERATION

A circuit which performs an AND operation is shown in Fig. 1.2. It has N inputs ($N \geq 2$) and one output. Digital signals are applied at the input terminals marked A, B, …, N, the other terminal being ground, which is not shown in the diagram. The output is obtained at the output terminal marked Y (the other terminal being ground) and it is also a digital

signal. The AND operation is defined as: the output is 1 if and only if all the inputs are 1. Mathematically, it is written as

$$\begin{aligned} Y &= A \text{ AND } B \text{ AND } C \ \ldots \text{ AND } N \\ &= A \cdot B \cdot C \cdot \ldots \cdot N \\ &= ABC \ \ldots N \end{aligned} \qquad \ldots(1.1)$$

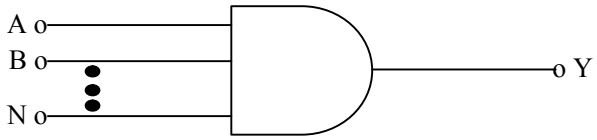


Fig. 1.2 The standard symbol for an AND gate

where A, B, C, … N are the input variables and Y is the output variable. The variables are binary, i.e. each variable can assume only one of the two possible values, 0 or 1. The *binary variables* are also referred to as *logical variables*.

Equation (1.1) is known as the *Boolean equation* or the *logical equation* of the AND *gate*. The term gate is used because of the similarity between the operation of a digital circuit and a gate. For example, for an AND operation the gate opens (Y = 1) only when all the inputs are present, i.e. at logic 1 level.

Truth Table: Since a logical variable can assume only two possible values (0 and 1), therefore, any logical operation can also be defined in the form of a table containing all possible input combinations (2^N combinations for N inputs) and their corresponding outputs. This is known as a *truth table* and it contains one row for each one of the input combinations.

For an AND gate with two inputs A, B and the output Y, the truth table is given in Table 1.1. Its logical equation is Y = AB and is read as "Y equals A AND B".

Since, there are only two inputs, A and B, therefore, the possible number of input combinations is four. It may be observed from the truth table that the input–output relationship for a digital circuit is completely specified by this table in contrast to the input–output relationship for an analog circuit. The pattern in which the inputs are entered in the truth table may also be observed carefully, which is in the ascending order of binary numbers formed by the input variables.

Table 1.1 Truth table of a 2-input AND gate

Inputs		Output
A	**B**	**Y**
0	0	0
0	1	0
1	0	0
1	1	1

1.3.2 OR OPERATION

Figure 1.3 shows an OR gate with N inputs (N ≥ 2) and one output. The OR operation is defined as: the output of an OR gate is 1 if and only if one or more inputs are 1. Its logical equation is given by

$$Y = A \text{ OR } B \text{ OR } C \ldots \text{ OR } N$$

$$= A + B + C + \ldots + N \qquad \ldots(1.2)$$

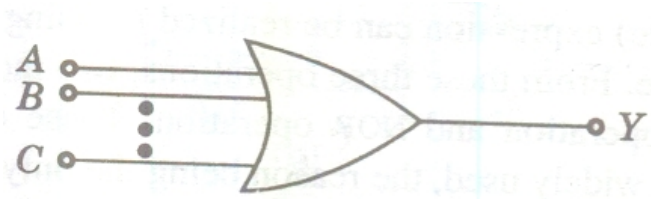


Fig. 1.3 The standard symbol for an OR gate

The truth table of a 2-input OR gate is given in Table 1.2. Its logic equation is Y = A + B and is read as "Y equals A or B".

Table 1.2 Truth table of a 2-input OR gate

Inputs		Output
A	**B**	**Y**
0	0	0
0	1	0

1	0	0
1	1	1

1.3.3 NOT OPERATION

Figure 1.4 shows a NOT gate, which is also known as an *inverter*. It has one input (A) and one output (Y). Its logic equation is written as and is read as "Y equals NOT A" or "Y equals complement of A". The truth table of a NOT gate is given in Table 1.3.

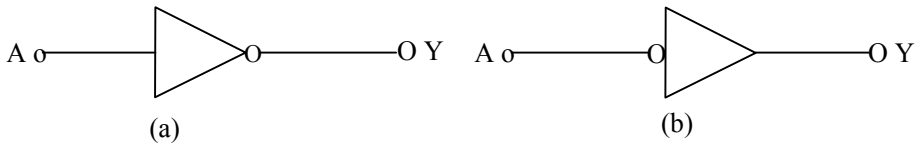


Fig. 1.4 The standard symbols for a NOT gate

$$Y = \text{NOT } A = \overline{A} \quad \ldots(1.3)$$

Table 1.3 Truth table of a NOT gate

Input **A**	**Output** **Y**
0	1
1	0

The NOT operation is also referred to as an inversion or complementation. The presence of a small circle, known as the *bubble*, always denotes inversion in digital circuits.

1.4 NAND AND NOR OPERATIONS

Any Boolean (or logic) expression can be realized by using the AND, OR and NOT gates discussed above. From these three operations, two more operations have been derived: the NAND operation and NOR operation. These operations have become very popular and are widely used, the reason being the only one type of gates, either NAND or NOR

are sufficient for the realization of any logical expression. Because of this reason, NAND and NOR gates are known as *universal gates*.

1.4.1 NAND OPERATION

The NOT-AND operation is known as the NAND operation. Figure 1.5a shows and N input ($N \geq 2$) AND gate followed by a NOT gate. The operation of this circuit can be described in the following way:

The output of the AND gate (Y′) can be written using Eq. (1.)

$$Y' = AB \ldots N \qquad \ldots(1.4)$$

Now, the output of the NOT gate (Y) can be written using Eq. (1.3)

$$Y = \overline{Y}' = (\overline{AB...N}) \qquad \ldots(1.5)$$

The logical operation represented by Eq. (1.5) is known as the NAND operation. The standard symbol of the NAND gate is shown in Fig. 1.5b. Here, a bubble on the output side of the NAND gate represents NOT operation, inversion or complementation.

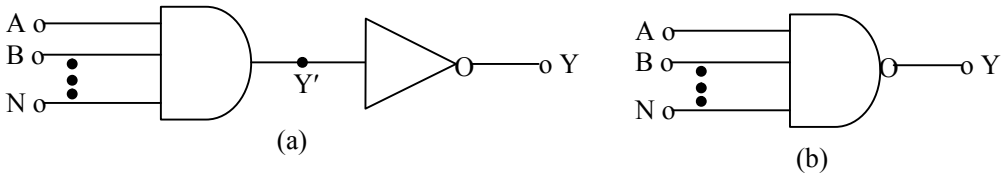


Fig. 1.5 (a) NAND operation as NOT-AND operation, (b) Standard symbol for the NAND gate.

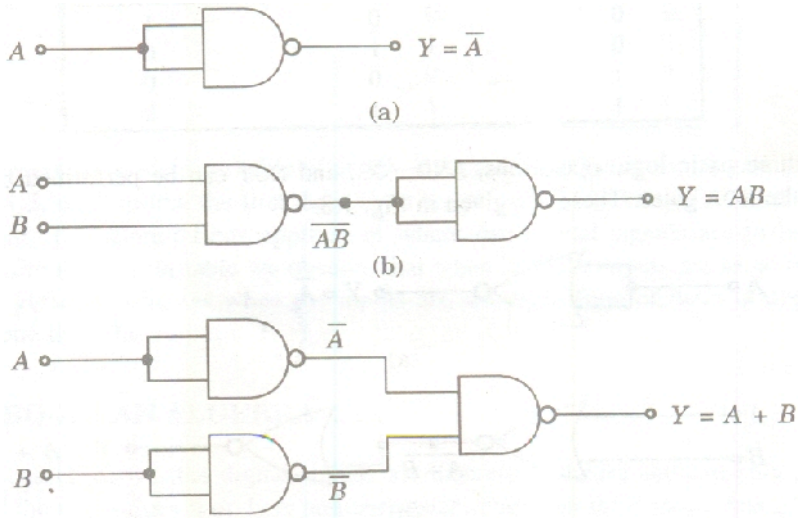
The truth table of a 2-input NAND gate is given in Table 1.4. Its logic equation is $Y = \overline{A \cdot B}$ and, is read as "Y equals NOT (A AND B)".

Table 1.4 Truth table of a 2-input NAND gate

Inputs		**Output**
A	**B**	**Y**
0	0	0
0	1	0
1	0	0

1	1	1

The three basic logic operations, AND, OR and NOT can be performed by using only NAND gates. These are given in Fig. 1.6.



(c)

Fig. 1.6 Realization of basic logic operations using NAND gates (a) NOT (b) AND (c) OR.

1.4.2 NOR OPERATION

The NOT-OR operation is known as the NOR operation. Figure 1.7a shows an N input (N ≥ 2) OR gate followed by a NOT gate. The operation of this circuit can be described in the following way:

The output of the OR gate Y′ can be written using Eq. (1.2) as

$$Y'. = A + B + \ldots + N \qquad \ldots(1.6)$$

and the output of the NOT gate (Y) can be written using Eq. (1.3)

$$Y = \overline{Y'} = \overline{A + B + \ldots + N} \qquad \ldots(1.7)$$

The logic operation represented by Equ. (1.7) is known as the NOR operation.

The standard symbol of the NOR gate is shown in Fig. 1.7b. Similar to the NAND gate, a bubble on the output side of the NOR gate represents the NOT operation.

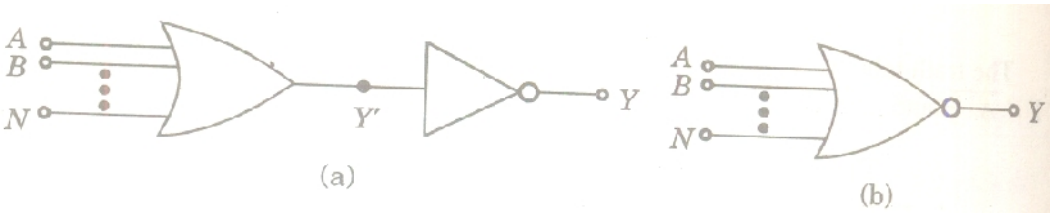


Fig. 1.7 (a) NOR operation as NOT-OR operation, (b) Standard symbol for the NOR gate

Table 1.5 gives the truth table of a 2-input NOR gate. Its logic equation is $Y = \overline{A + B}$ and is read as "Y equals NOT (A OR B)"

Table 1.5 Truth table of a 2-input NOR gate

INPUTS		**OUTPUT**
A	**B**	**Y**
0	0	0
0	1	0
1	0	0
1	1	1

The three basic logic operations, AND, OR, and NOT can be performed by using only the NOR gates. These are given in Fig. 1.8.

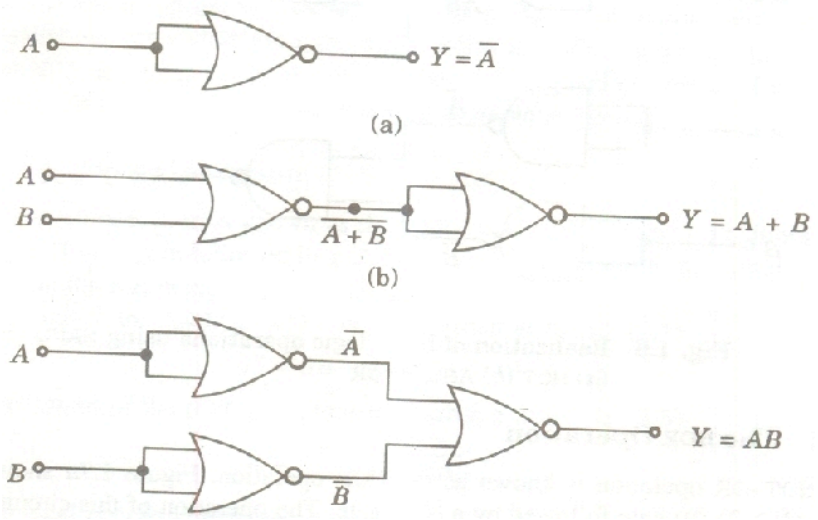


Fig. 1.8 Realization of basic logic operations using NOR gates (a) NOT (b) OR (c) AND

1.5 EXCLUSIVE-OR OPERATIONS

The EXCLUSIVE–OR (EX–OR) operation is widely used in digital circuits. It is not a basic operation and can be performed using the basic gates–AND, OR and NOT or universal gates NAND or NOR. Because of its importance, the standard symbol shown in Fig. 1.9 is used for this operation.

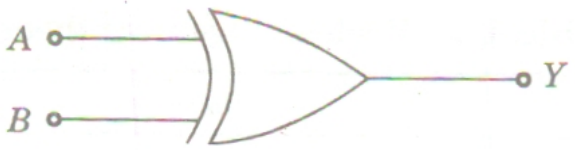


Fig. 1.9 Standard symbol for EX-OR gate.

The truth table of an EX–OR gate is given in Table 1.6 and its logic equation is written as

$$Y = A \text{ EX} - \text{OR } B = A \oplus B \qquad \ldots(1.8)$$

Table 1.6 Truth table of a 2-input EX–OR gate

Inputs		**Output**
A	**B**	**Y**
0	0	0
0	1	1
1	0	1
1	1	0

If we compare the truth table of an EX–OR gate with that of an OR gate given in Table 1.2, we find that the first three rows are same in both. Only the fourth row is different. This circuit finds application where two digital signals are to be compared. From the truth table we observe that when both the inputs are same (0 or 1) the output is 0, whereas when the inputs are not same (one of them is 0 and the other one is 1) the output is 1.

1.6 CHECK YOUR PROGRESS

1. How many AND gates are required to realize Y = CD + EF + G?
2. The output of an XOR gate is HIGH only when _____.
3. The NOR gate output will be high if the two inputs are _____ .
4. When does the output of a NAND gate = 1?
5. _____ are known as universal gates.

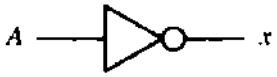
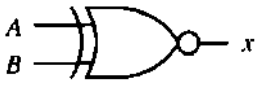
1.7 SUMMARY

Let's take a quick review of all that we have learned about digital logic gates.

1. Analog signals depict continuous variation of the magnitude over a certain time whereas digital signals depict discrete values at various instants.
2. Analog instruments indicate the magnitude through the position of pointer on the scale. A digital instrument displays the actual magnitude in the form of digits.
3. Digital communication system, Digital control systems and Digital computers are widely used.
4. The representation of analog signal in digital form is by the use of digital signal processor (DSP).
5. We have to feed a program and data to a digital computer so that the computer may process the data and produce the desired output.

Figure 1-10 Digital logic gates.

Name	*Graphic Symbol*	*Algebraic Function*	*Truth Table*
AND		x = A•B or x = AB	A B | x 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1

OR		x = A + B	A B | x 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1
Inverter		x = A′	A | x 0 | 1 1 | 0
Buffer		x = A	A | x 0 | 0 1 | 1
NAND		x = (AB)′	A B | x 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0
NOR		x = (A + B)′	A B | X 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0
Exclusive-OR (XOR)		x = A ⊕ B or x = A′B + AB′	A B | X 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 0
Exclusive-NOR Or equivalence		x = (A ⊕ B)′ or x = A′B′ + AB	A B | x 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 1

6. Digital computers work on binary numbers, i.e., 1 and 0.
7. Digital signals can be represented in positive or negative logic. In positive logic, the more positive level is level 1 and the other is level 0. In negative logic the more negative level is level 1 and the other is level 0. Positive logic is used more commonly.
8. An ideal pulse changes from low to high and high to low levels in zero time. An actual pulse has finite rise and full times.
9. To understand the digital logic gates (AND, OR, NOT, NAND, NOR, XOR) operations, symbols and interconnection.

1.8 KEYWORDS

1. **Logic gate** is comprised of resistors and transistors, or diodes. They can perform simple or highly complex operations by joining a variety of logic gates.
2. **Boolean logic** is a form of algebra in which all values are reduced to either TRUE or FALSE.
3. **Boolean algebra** is used to analyze and simplify the digital (logic) circuits. It uses only the binary numbers i.e. 0 and 1. It is also called as Binary Algebra or logical Algebra.
4. **Karnaugh map (**K-Map) is a pictorial method used to minimize Boolean expressions without having to use Boolean algebra theorems and equation manipulations.
5. M**interm** is a Boolean expression resulting in 1 for the output of a single cell, and 0s for all other cells in a Karnaugh map, or truth table.
6. **SOP** (Sum of Product) and **POS** (Product of Sum) are the methods for deducing a particular logic function.
7. **Don't-Care** condition allows us to replace the empty cell of a K-Map to form a grouping of the variables. While forming groups of cells, we can consider a "Don't Care" cell as either 1 or 0 or we can simply ignore that cell.

1.9 SELF ASSESSMENT TEST

1. Which of the following systems are analog and which are digital? Why?

a) Pressure gauge

b) An electronic counter used to count persons entering an exhibition

c) Clinical thermometer

d) Electronic calculator

e) Transistor radio receiver

f) Ordinary electric switch.

2. In the circuits of Figure 1.11, the switches may be ON (1) or OFF (0) and will cause the bulb to be ON (1) or OFF (0).

   a) Determine all possible conditions of the switches for the bulb to be ON (1)/ OFF (0) in each of the circuits.

   b) Represent the information obtained in part (a) in the form of truth table.

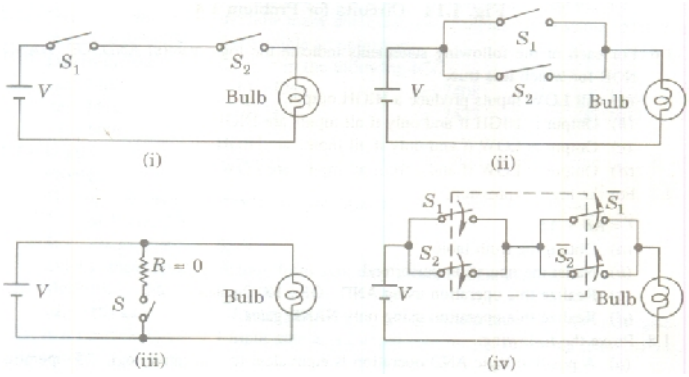


Fig. 1.11 Circuits for Problem 2

3. The voltage waveforms shown in Fig. 1.12 are applied at the inputs of 2-input AND, OR, NAND, NOR, and EX-OR gates. Determine the output waveform in each case.

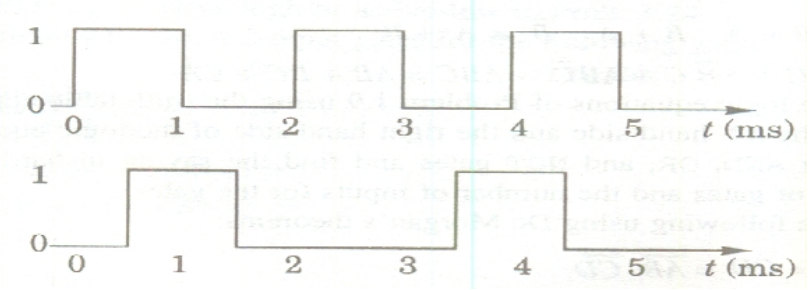


Figure 1.12 Waveforms for Problem 1.3

4. Find the relationship between the inputs and output for each of the gates shown in Fig. 1.13. Name the operation performed in each case.

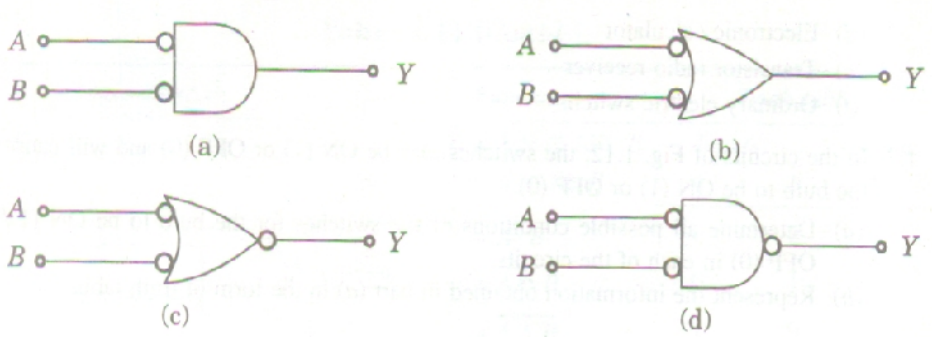


Figure 1.13 Circuits for Problem 4.

5. For each of the following statements indicate the logic gate(s), AND, OR, NAND, NOR for which it is true.
   a) All Low inputs produce a HIGH output.
   b) Output is HIGH if and only if all inputs are HIGH.
   c) Output is LOW if and only if all inputs are HIGH.
   d) Output is LOW if and only if all inputs are LOW.
6. For the logic expression,
   $Y = A\overline{B} + \overline{A}B$
   a) Obtain the truth table.
   b) Name the operation performed.
   c) Realize this operation using AND, OR, NOT gates.
   d) Realize this operation using only NAND gates.
7. Prove the following:
   a) A positive logic AND operation is equivalent to a negative logic OR operation and vice-versa.
   b) A positive logic NAND operation is equivalent to a negative logic NOR operation and vice-versa.

1.10 ANSWER TO CHECK YOUR PROGRESS

1. Two
2. the two inputs are unequal
3. 00
4. Whenever a 0 is present at an input

5. NAND and NOR

1.11 REFERENCES / SUGGESTED READINGS

1. Computer Organization and Architecture, Rajaram & Radhakrishan, PHI.
2. Computer Organization & Architecture: Designing for Performance, Stalling, PHI.
3. Computer Organization and Design, Pal Choudhary, PHI.
4. Computer Systems Organization & Architecture, Carpenelli, Pearson Education.
5. Computer Organization and Architecture, Stalling, Pearson Education.
6. Computer System Architecture, Morris Mano, PHI.
7. Computer Architecture and Organization, McGraw Hill Company, New Delhi. J.P. Hayes.

SUBJECT: COMPUTER SYSTEM ARCHITECTURE	
COURSE CODE: MCA-24	**AUTHOR: NEERAJ VERMA**
LESSON NO. 2	

BOOLEAN ALGEBRA

STRUCTURE

2.0 LEARNING OBJECTIVES

After reading this lesson, you should be able to:

1. Write the Boolean expression for the logic gates and combinations of logic gates.
2. Appreciate the potential of Boolean algebra to simplify complex logic expressions.
3. Use DeMorgan's theorems to simplify logic expressions.
4. Use either of the universal gates (NAND or NOR) to implement a circuit represented by a Boolean expression.

2.1 INTRODUCTION

Logic operations and Boolean algebra have already been discussed in Chapter 1. Boolean algebraic theorems are used for the manipulations of logical expressions. It has also been demonstrated that a logical expression can be realized using the logic gates. The number of gates and the number of input terminals for the gates required for the realization of a logical expression, in general, get reduced considerably if the expression can be simplified. Therefore, the simplification of logical expression is very important as it saves the hardware required to design a specific system. A large number of functions are available in IC form and therefore, we should be able to make optimum use of these ICs in the design of digital systems. That is, our aim should be to minimize the number of IC packages. Basically, digital circuits are divided into two broad categories:

1. Combinational circuits, and
2. Sequential circuits,

In *combinational circuits,* the outputs at any instant of time depend upon the inputs present at that instant of time. This means there is no memory in these circuits. There are other types of circuits in which the outputs at any instant of time depend upon the present inputs as well as past inputs/outputs. This means that there are elements used to store past information. These elements are known as *memory.* Such circuits are known as *sequential circuits.* A sequential logic system may have combinational logic sub-systems. The design of combinational circuits will be discussed here. Sequential circuit design will be discussed later.

The design requirements of combinational circuits may be specified in one of the following ways:

1. A set of statements
2. Boolean expression, and
3. Truth table.

The aim is to design a circuit using the gates already discussed or some other circuits which are in fact derived from the basic gates. As is usual in any engineering design, the number of components used should be minimum to ensure low cost, saving in space, power requirements, etc. There can be two different approaches to the design of combinational circuits. One of these is the traditional method, wherein the given Boolean expression or the truth table is simplified by using standard methods and the simplified expression is realized using the gates. The other method normally does not require any simplification of the logical expression or truth table, instead the complex logic functions available in medium scale integrated circuits (MSI) or large scale integrated circuits (LSI) can be directly used. Combinational circuit design using the traditional design methods has been discussed below.

The following methods can be used to simplify the Boolean function:

1. Algebraic method,
2. Karnaugh map technique,
3. Variable entered aping (VEM) technique, and
4. Quino-McCluskey method.

2.3 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, and y. 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 1; 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^n combinations of the n binary variables.

A Boolean function can be transformed from an algebraic expression into a logic diagram composed of AND, OR, and inverter gates. 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.

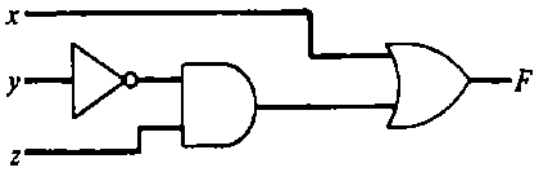
The purpose of Boolean algebra is to facilitate the analysis and design of digital circuits. It provides a convenient tool to:

1. Express in algebraic form a truth table relationship between binary variables.

Figure 2-1 Truth table and logic diagram for F = x + y′

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

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.

Table 2-1 Basic Identities of Boolean Algebra

(1) $x + 0 = x$	(2) $x \cdot 0 = 0$
(3) $x + 1 = 1$	(4) $x \cdot 1 = x$
(5) $x + x = x$	(6) $x \cdot x = x$
(7) $x + x' = 1$	(8) $x \cdot x' = 0$
(9) $x + y = y + x$	(10) $xy = yx$
(11) $x + (y + z) = (x + y) + z$	(12) $x(yz) = (xy)z$
(13) $x(y + z) = xy + xz$	(14) $x + yz = (x + y)(x + z)$
(15) $(x + y)' = x'y'$	(16) $(xy)' = x' + y'$
(17) $(x')' = x$	

Table 2-1 lists the most basic identities of Boolean algebra. All the identities in the table can be proven by means of truth tables. The first eight identities show the basic relationship between a single variable and itself, or in conjunction with the binary constants 1 and 0. The next five identities (9 through 13) are similar to ordinary algebra. Identity 14 does not apply in ordinary algebra but is very useful in manipulating Boolean expressions. Identities 15 and 16 are called DeMorgan's theorems and are discussed below. The last identity states that if a variable is complemented twice, one obtains the original value of the variable.

The identities listed in the table apply to single variables or to Boolean functions expressed in terms of binary variables. For example, consider the following Boolean algebra expression:

$$AB' + C'D + AB' + C'D$$

By letting $x = AB' + C'D$ the expression can be written as $x + x$. From identity 5 in Table 2-1 we find the $x + x = x$. Thus the expression can be reduced to only two terms:

$$AB' + C'D + A'B + C'D = AB' + C'D$$

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 Figure 2-2 and 2-3. 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 Figure 2-3.

To see how Boolean algebra manipulation is used to simplify digital circuits, consider the logic diagram of Figure 2-4(a). The output of the circuit can be expressed algebraically as follows:

$$F = ABC + ABC' + A'C$$

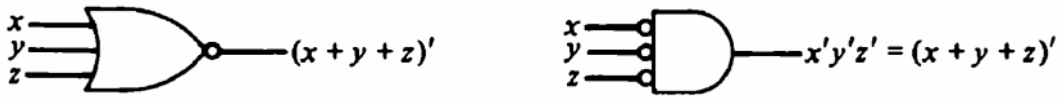
Each term corresponds to one AND gate, and the OR gate forms the logical sum of the three terms. Two inverters are needed to complement A′ and C′. The expression can be simplified using Boolean algebra.

$$\begin{aligned} F = ABC + ABC' + A'C &= AB(C + C') + A'C \\ &= AB + A'C \end{aligned}$$

Note that $(C + C)' = 1$ by identity 7 and $AB \cdot 1 = AB$ by identity 4 in Table 2-1.

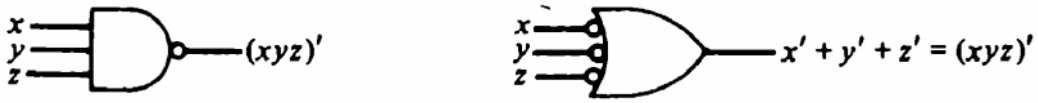
The logic diagram of the simplified expression is drawn in Figure 2-4(b). It requires only four gates rather than the six gates used in the circuit of Figure 2-4(a). The two circuits are equivalent and produce the same truth table relationship between inputs A, B, C and output F.

Figure 2-2 Two graphic symbols for NOR gate

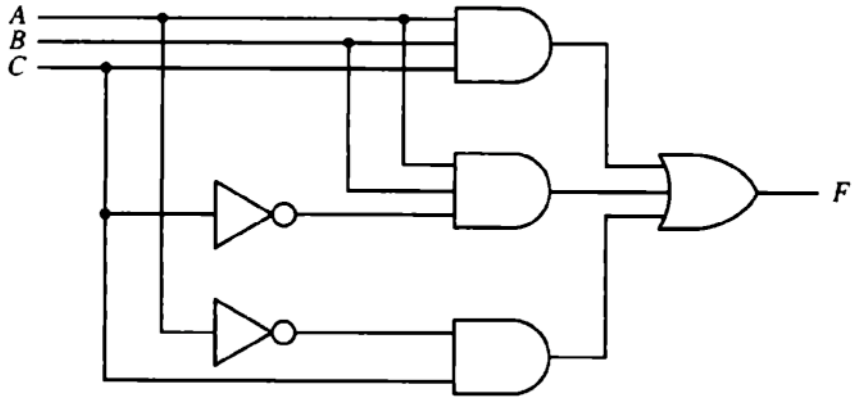


(a) OR-invert **(b) invert-AND**

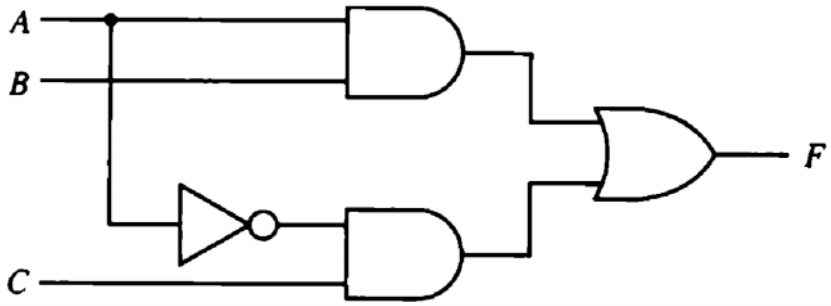
Table 2-3 Two graphic symbols for NAND gate



(a) AND-invert **(b) invert-OR**



(a) F = ABC + ABC′ + A′C



(b) F = AB + A′C

Figure 2-4 Two Logic diagrams for the same Boolean function

2.2.1 COMPLEMENT OF A FUNCTION

The complement of a function F when expressed in a truth table is obtained by interchanging 1's and 0'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:

$$(x_1 + x_2 + x_3 + \cdots + x_n)' = x_1' x_2' x_3' \cdots x_n'$$

$$(x_1 x_2 x_3 \cdots x_n)' = x_1' + x_2' + x_3' + \cdots + x_n'$$

From the general DeMorgan's theorem we can derive a simple procedure for obtaining the complement of an algebraic expression. This is done by changing all OR operations to AND operations and all AND operations to OR operations and then complementing each

individual letter variable. As an example, consider the following expression and its complement:

$$F = AB + C'D' + B'D$$

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

The complement expression is obtained by interchanging AND and OR operations and complementing each individual variable. Note that the complement of C′ is C.

2.3 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.

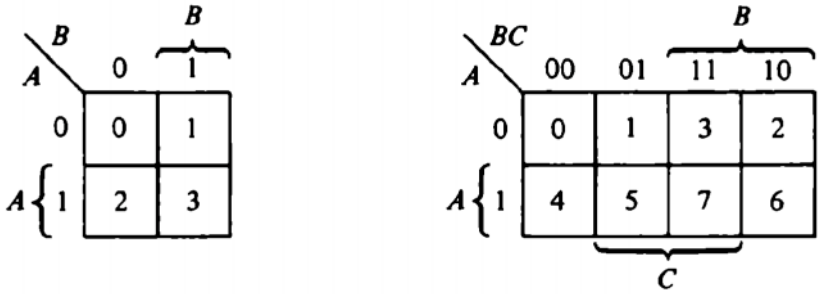
Each combination of the variables in a truth table is called a minterm. For example, the truth table of Figure 2-1 contains eight minterms. When expressed in a truth table a function of n variables will have 2^n minterms, equivalent to the 2^n binary numbers obtained from n bits. A Boolean function is 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 truth table of Figure 2-1 can be expressed as follows:

$$F(x, y, z) = \Sigma(1,4,5,6,7)$$

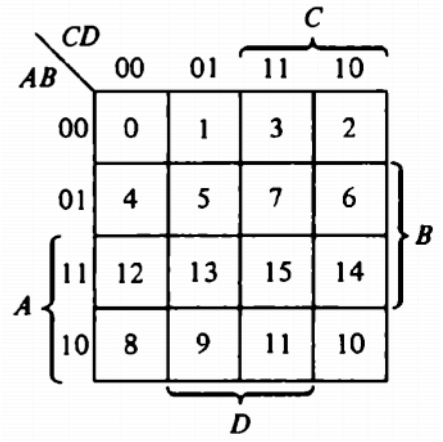
The letters in parentheses list the binary variables in the order that they appear in the truth table. The symbol Σ stands for the sum of the minterms that follow in parentheses. The minterms that produce 1 for the function are listed in their decimal equivalent. The minterms missing from the list are the ones that produce 0 for the function.

The map is a diagram made up of squares, with each square representing one minterm. The squares corresponding to minterms that produce 1 for the function are marked by a 1

and the others are marked by a 0 or are left empty. By recognizing various patterns and combining squares marked by 1's in the map, it is possible to derive alternative algebraic expressions for the function, from which the most convenient may be selected.



(a) Two-variable map **(b) Three-variable map**



(c) Four-variable map

Figure 2-5 Maps for Two-, three-, and four-variable functions

The map for functions of two, three, and four variables are shown in Figure 2-6. The number of squares in a map of n variables is 2^n. The 2^n 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 0's and 1's marked along each row and each column designates the value of the variables. Each variable under brackets contains half of the squares in the map where that variable appears unprimed. The variable appears with a prime (complemented) in the remaining half of the squares.

The minterm represented by a square is determined from the binary assignments of the variables along the left and top edges in the map. For example, minterm 5 in the three-variable map is 101 in binary, which may be obtained from the 1 in the second row

concatenated with the 01 of the second column. This minterm represents a value for the binary variables A, B, and C, with A and C being unprimed and B being primed (i.e., AB′C). On the other hand, minterm 5 in the four-variable map represents a minterm for four variables. The binary number contains the four bits 0101, and the corresponding term it represents is A′BC′D.

Minterms of adjacent squares in the map are identical except for one variable, which appears complemented in one square and uncomplemented in the adjacent square. According to this definition of adjacency, the squares at the extreme ends of the same horizontal row are also to be considered adjacent. The same applies to the top and bottom squares of a column. As a result, the four corner squares of a map must also be considered to be adjacent.

A Boolean function represented by a truth table is plotted into the map by inserting 1's in those squares where the function is 1. 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

$$F(A, B, C) = \Sigma (3, 4, 6, 7)$$

The three-variable map for this function is shown in Figure 2-6. There are four squares marked with 1's, one for each minterm that produces 1 for the function. These squares belong to minterms 3, 4, 6, and 7 and are recognized from Figure 2-5(b). Two adjacent squares are combined the third column. This column belongs to both B and C and produces the term BC. The remaining two squares with 1's in the two corners of the second row are adjacent and belong to row A and the two columns of C′, so they produce the term AC′. The simplified algebraic expression for the function is the OR of the two terms:

$$F = BC + AC'$$

The second example simplifies the following Boolean function:

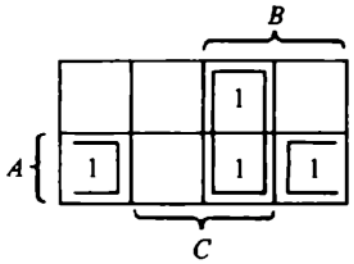
$$F(A, B, C) = \Sigma (0, 2, 4, 5, 6)$$

The five minterms are marked with 1's in the corresponding squares of the three-variable map shown in Figure 2-7. The four squares in the first and fourth columns are adjacent

and represent the term C′. The remaining square marked with a 1 belongs to minterm 5 and can be combined with the square of minterm 4 to produce the term AB′. The simplified function is

$$F = C' + AB'$$

Figure 2-6 Map for F(A, B, C) = Σ (3, 4, 6, 7)



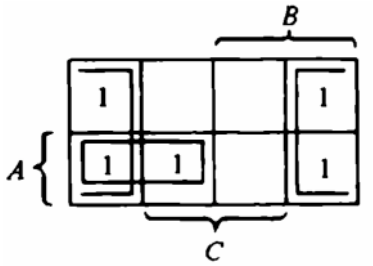


Figure 2-7 Map for F(A, B, C) = Σ (0, 2, 4, 5, 6)

The third example needs a four-variable map.

$$F(A, B, C, D) = \Sigma (0, 1, 2, 6, 8, 9, 10)$$

The area in the map covered by this four-variable function consists of the squares marked with 1's in Figure 2-8. The function contains 1's in the four corners that, when taken as a group, give the term B′D′. This is possible because these four squares are adjacent when the map is considered with top and bottom or left and right edges touching. The two 1's on the left of the top row are combined with the two 1's on the left of the bottom row to give the term B′C′. The remaining 1 in the square of minterm 6 is combined with minterm 2 to give the term A′CD′. The simplified function is

$$F = B'D' + B'C' + A'CD'$$

2.3.1 PRODUCT-OF-SUMS SIMPLIFICATION

The Boolean expressions derived from the maps in the preceding examples were expressed in sum-of-products form. The product terms are AND terms and the sum

denotes the ORing of these terms. It is sometimes convenient to obtain the algebraic expression for the function in a product-of-sums form. The sums are OR terms and the product denotes the ANDing of these terms. With a minor modification, a product-of-sums form can be obtained from a map.

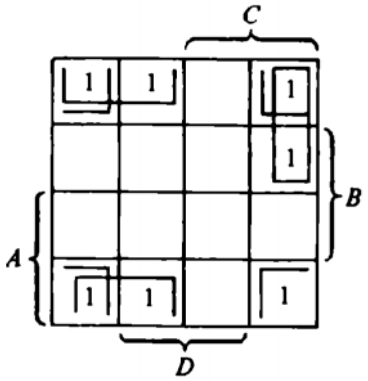


Figure 2-8 Map for F(A, B, C, D) = Σ (0, 1, 2, 6, 8, 9, 10)

The procedure for obtaining a product-of-sums expression follows from the basic properties of Boolean algebra. The 1's in the map represent the minterms that produce 1 for the function. The squares not marked by 1 represent the minterms that produce 0 for the function. The squares not marked by 1 represent the minterms that produce 0 for the function. If we mark the empty squares with 0's and combine them into groups of adjacent squares, we obtain the complement of the function, F′. Taking the complement of F′ produces an expression for F in product-of-sums form. The best way to show this is by example.

We wish to simplify the following Boolean function in both sum-of-products form and product-of-sums form:

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

The 1's marked in the map of Figure 2-10 represent the minterms that produce a 1 for the function. The squares marked with 0's represent the minterms not included in F and therefore denote the complement of F. Combining the squares with 1's gives the simplified function in sum-of-products form:

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

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

$$F' = AB + CD + BD'$$

Taking the complement of F′, we obtain the simplified function in product-of-sums form:

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

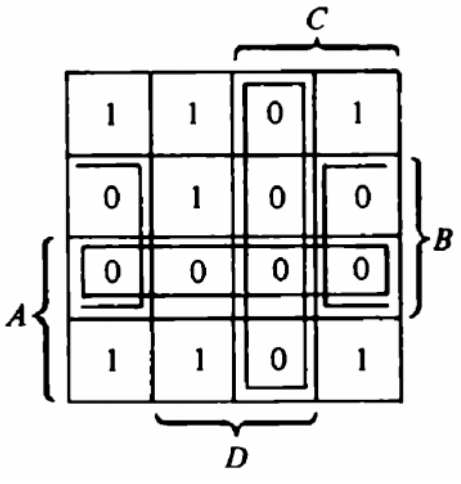


Figure 2-9 Map for F(A, B, C, D) = Σ (0, 1, 2, 5, 8, 9, 10)

The logic diagrams of the two simplified expressions are shown in Figure 2-10. The sum-of-products expression is implemented in Figure 2-10(a) with a group of AND gates, one for each AND term. The outputs of the AND gates are connected to the inputs of a single OR gate. The same function is implemented in Figure 2-10(b) in product-of-sums form with a group of OR gates, one for each OR term. The outputs of the OR gates are connected to the inputs of a single AND gate. In each case it is assumed that the input variables are directly available in their complement, so inverters are not included. The pattern established in Figure 2-10 is the general form by which any Boolean function is implemented when expressed in one of the standard forms. AND gates are connected to a single OR gate when in sum-of-products form. OR gates are connected to a single AND gate when in product-of-sums form.

A sum-of-products expression can be implemented with NAND gates as shown in Figure 2-11(a). Note that the second NAND gate is drawn with the graphic symbol of Figure 2-3(b). There are three lines in the diagram with small circles at both ends. Two circles in the same line designate double complementation, and since $(x')' = x$, the two circles can be removed and the resulting diagram is equivalent to the one shown in Figure 2-10(a). Similarly, a product-of-sums expression can be implemented with NOR gates as shown in Figure 2-11(b). The second NOR gate is drawn with the graphic symbol of Figure 2-

2(b). Again the two circles on both sides of each line may be removed, and the diagram so obtained is equivalent to the one shown in Figure 2-10(b).

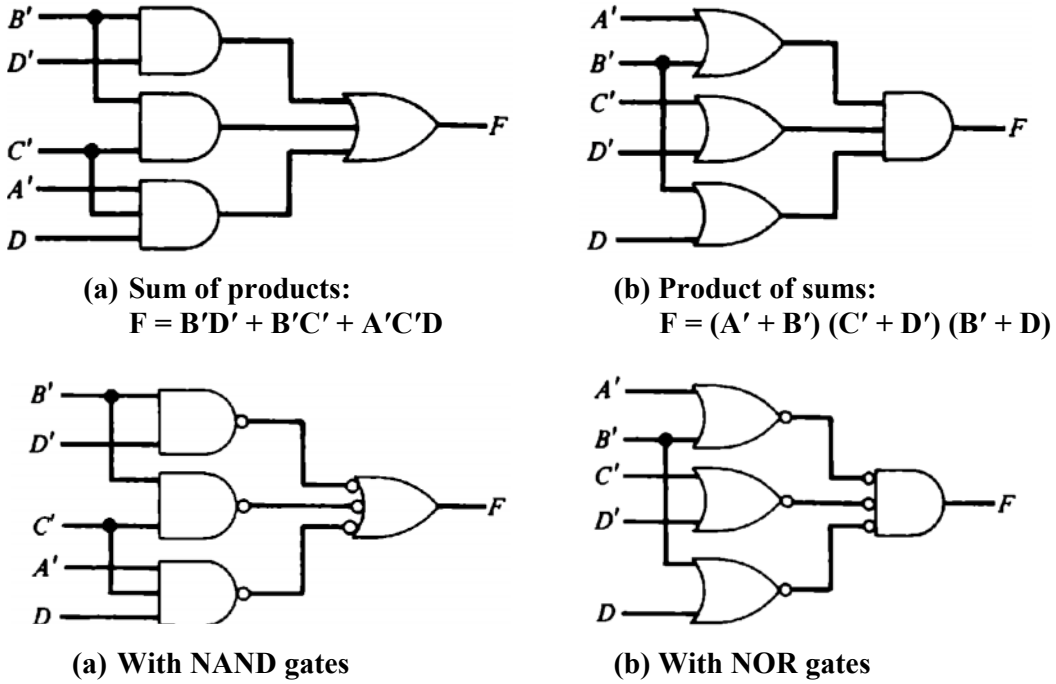
2.3.3 DON'T-CARE CONDITIONS

The 1's and 0's in the map represent the minterms 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 ea 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 minterm. Minterms 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.

When choosing adjacent squares for the function in the map, the x's may be assumed to be either 0 or 1, whichever gives the simplest expression. In addition, an x need not be used at all if it does not contribute to the simplification of the function. In each case, the choice depends only on the simplification that can be achieved. As an example, consider the following Boolean function together with the don't-care minterms:

$$F(A, B, C) = \Sigma (0, 2, 6)$$
$$d(A, B, C) = \Sigma (1, 3, 5)$$

Figure 2-10 Logic diagrams with AND and OR gates



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

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

(a) With NAND gates

(b) With NOR gates

Figure 2-11 Logic diagram with NAND or NOR gates

The minterms listed with F produce a 1 for the function. The don't-care minterms listed with d may produce either a 0 or a 1 for the function. The remaining minterms, 4 and 7, produce a 0 for the function. The map is shown in Figure 2-12. The minterms of F are marked with 1's, those of d are marked with x's, and the remaining squares are marked with 0's. The 1's and x's are combined in any convenient manner so as to enclose the maximum number of adjacent squares. It is not necessary to include all or any of the x's, but all the 1's must be included. By including the don't-care minterms 1 and 3 with the 1's in the first row we obtain the term A′. The remaining 1 for minterm 6 is combined with minterm 2 to obtain the term BC′. The simplified expression is

$$F = A' + BC'$$

Note that don't-care minterm 5 was not included because it does not contribute to the simplification of the expression. Note also that if don't-care minterms 1 and 3 were not included with the 1's, the simplified expression for F would have been

$$F = A'C' + BC'$$

This would require two AND gates and an OR gate, as compared to the expression obtained previously, which requires only one AND and one OR gate.

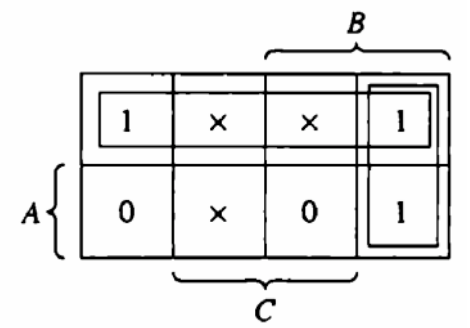


Figure 2-12 Example of map with don't-care conditions

The function is determined completely once the x's are assigned to the 1's and 0's in the map. Thus the expression

$$F = A' + BC'$$

Represents the Boolean function

$$F(A, B, C) = \Sigma (0, 1, 2, 3, 6)$$

It consists of the original minterms 0, 2, and 6 and the don't-care minterms 1 and 3. Minterm 5 is not included in the function. Since minterms 1, 3, and 5 were specified as being don't-care conditions, we have chosen minterms 1 and 3 to produce a 1 and

minterm 5 to produce a 0. This was chosen because this assignment produces the simplest Boolean expression.

2.4 CHECK YOUR PROGRESS

1. The application of Boolean algebra to the solution of digital logic circuits was first explored by _____ of _____.
2. One reason for using the sum-of-products form is that it can be implemented using all _____ gates without much alteration.
3. The involution of A is equal to _____ .
4. DeMorgan's theorem states that _____ .
5. There are _____ Minterms for 3 variables (a, b, c).

2.5 SUMMARY

Let's take a quick review of all that we have learned about binary logic.

1. Boolean algebra is used to simplify the complex logic expressions of a digital circuit. Thereby allowing us to reduce complex circuits into simpler ones.
2. Reading specification (truth) tables and DeMorgan's theorem.
3. A Boolean function can be simplified either by algebraic method or by Karnaugh map. K-maps provide a precise of steps to follow to find the minimal representation of a function, and thus the minimal circuit that function represents.
4. Understand the concept of sum-of-products (SOP), product-of-sums (POS) and the use of "don't-care" condition.

2.6 KEYWORDS

1. **Boolean logic** is a form of algebra in which all values are reduced to either TRUE or FALSE.
2. **Boolean algebra** is used to analyze and simplify the digital (logic) circuits. It uses only the binary numbers i.e. 0 and 1. It is also called as Binary Algebra or logical Algebra.

3. **Karnaugh map (**K-Map) is a pictorial method used to minimize Boolean expressions without having to use Boolean algebra theorems and equation manipulations.
4. M**interm** is a Boolean expression resulting in 1 for the output of a single cell, and 0s for all other cells in a Karnaugh map, or truth table.
5. **SOP** (Sum of Product) and **POS** (Product of Sum) are the methods for deducing a particular logic function.
6. **Don't-Care** condition allows us to replace the empty cell of a K-Map to form a grouping of the variables. While forming groups of cells, we can consider a "Don't Care" cell as either 1 or 0 or we can simply ignore that cell.

2.7 SELF ASSESSMENT TEST

1. What are logic gates? Give a brief idea of Boolean algebra.
2. Using DeMorgan's theorem, show that:
   a. $(A + B)' (A' + B')' = 0$
   b. $(BC' + A'D) (AB' + CD')$
4. Simplify the following expressions using Boolean algebra.
   a. $A + AB$
   b. $AB + AB'$
   c. $A'BC + AC$
   d. $A'B + ABC' + ABC$
5. Simplify the following Boolean functions using three-variable maps.
   a. $F(A, B, C) = \Sigma (0, 1, 5, 7)$
   b. $F(A, B, C) = \Sigma (1, 2, 3, 6, 7)$
   c. $F(A, B, C) = \Sigma (3, 5, 6, 7)$
   d. $F(A, B, C) = \Sigma (0, 2, 3, 4, 6)$
6. Simplify the following expressions in (1) sum-of-products form and (2) product-of-sums form.
   a. $x'z' + y'z' + yz' + xy$
   b. $AC' + B'D + A'CD + ABCD$

2.8 ANSWER TO CHECK YOUR PROGRESS

1. Claude Shannon, MIT
2. NAND
3. A
4. (AB)' = A' + B' & (A + B)' = A'B'
5. 8

2.9 REFERENCES / SUGGESTED READINGS

1. Computer Organization and Architecture, Rajaram & Radhakrishan, PHI.
2. Computer Organization & Architecture: Designing for Performance, Stalling, PHI.
3. Computer Organization and Design, Pal Choudhary, PHI.
4. Computer Systems Organization & Architecture, Carpenelli, Pearson Education.
5. Computer Organization and Architecture, Stalling, Pearson Education.
6. Computer System Architecture, Morris Mano, PHI.
7. Computer Architecture and Organization, McGraw Hill Company, New Delhi. J.P. Hayes.

SUBJECT: COMPUTER SYSTEM ARCHITECTURE	
COURSE CODE: MCA-24	**AUTHOR: NEERAJ VERMA**
LESSON NO. 3	

COMBINATIONAL CIRCUITS

STRUCTURE

3.0 LEARNING OBJECTIVES

After reading this lesson, you should be able to:

1. Understand combinational circuit with diagram.
2. Describe the functions of encoders, decoders, multiplexers, adders, subtractors, and comparators.
3. Identify the schematic symbols for encoders, decoders, multiplexers, adders, subtractors, and comparators.
4. Identify applications for combinational logic circuits.
5. Develop truth tables for the different combinational logic circuits.

3.1 INTRODUCTION

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 Figure 3-1. 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 transforms binary information from the given input data to the required output data. Combinational circuit are employed in digital computers for generating binary control decisions and for providing digital components required for data processing.

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^n 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.

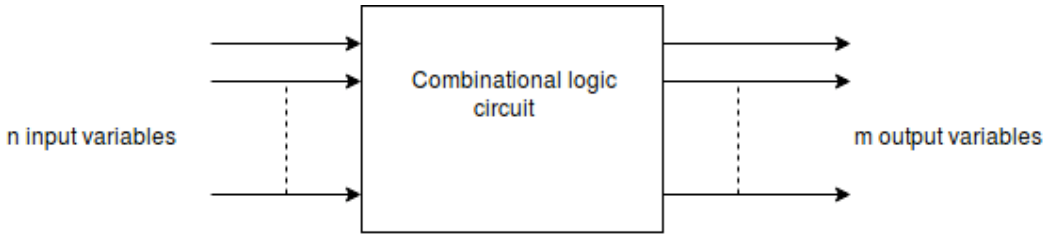


Figure 3-1 Block diagram of a combinational circuit

The analysis of a combinational circuit starts with a given logic circuit diagram and culminates with a set of Boolean functions or a truth table. If the digital circuit is a accompanied by a verbal explanation of its function, the Boolean functions or the truth table is sufficient for verification. If the function of the circuit is under investigation, it is necessary to interpret the operation of the circuit from the derived Boolean functions or the truth table. The success of such investigation is enhanced if one has experience and familiarity with digital circuits. The ability to correlate a truth table or a set of Boolean functions with an information processing task is an art that one acquires with experience.

3.2 DESIGN PROCEDURE

The design procedure of a 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.

To demonstrate the design of combinational circuits, we present two examples of simple arithmetic circuits. These circuits serve as basic building blocks for the construction of more complicated arithmetic circuits.

3.3 ADDER

An adder is a digital logic circuit in electronics that implements addition of numbers. In many computers and other types of processors, adders are used to calculate addresses, similar operations and table indices in the ALU and also in other parts of the processors. These can be built for many numerical representations like excess-3 or binary coded decimal.

Adders are classified into two types:

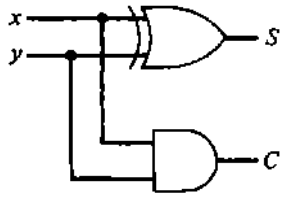
- Half Adder
- Full Adder

3.3.1 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.

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

Figure 3-2 Half-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 figure 3-2(a). The C output is 0 unless both inputs are 1. 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:

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

The logic diagram is shown in Figure 3-2(b). It consists of an exclusive-OR gate and an AND gate.

3.3.2 FULL ADDER

The 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, 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 Table 3-1. The eight rows under the input variables designate all possible combinations that the binary variables may have. The value of the output variables is 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 1. The C output has a carry of 1 if two or three inputs are equal to 1.

Table 3-1 Truth Table for Full-Adder

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 maps of Figure 3-3 are used to find algebraic expressions for the two output variables. The 1's in the squares for the maps of S and C are determined directly from the minterms in the truth table. The squares with 1's for the S output do not combine in groups of adjacent squares. But since the output is 1 when an odd number of inputs are 1, S is an odd function and represents the exclusive-OR relation of the variables. The squares with 1's for the C output may be combined in a variety of ways. One possible expression for C is

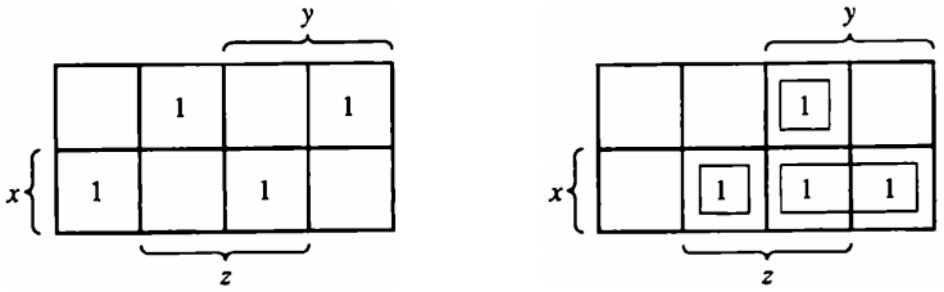
$$C = xy + (x'y + xy')z$$

Realizing that $x'y + xy' = x \oplus y$ and including the expression for output S, We obtain the two Boolean expressions for the full-adder:

$$S = x \oplus y \oplus z$$

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

The logic diagram of the full-adder is drawn in Figure 3-4. Note that the full-adder circuit consists of two half-adders and an OR gate. When used in subsequent chapters, the full-adder(FA) will be designated by a block diagram as shown in Figure 3.4(b).

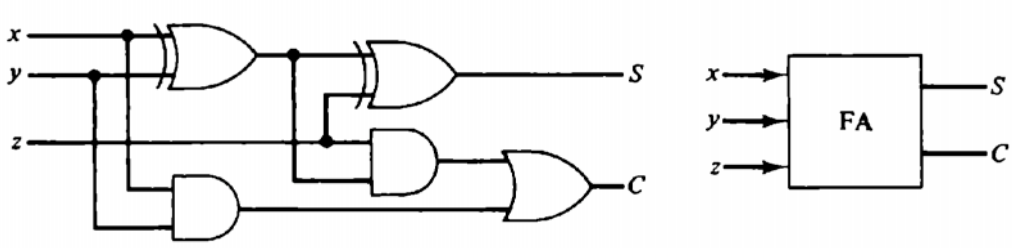


S = x'y'z + x'yz' + xy'z' + xyz
= x ⊕ y ⊕ z

C = xy + xz + yz
= xy + (x'y + xy')z

Figure 3-3 Maps of Full-Adder

The logic diagram for a full-adder circuit can be represented as:



(a) Logic Diagram **(b) Block diagram**

Figure 3-4 Full-adder circuit

3.4 SUBTRACTOR

Subtractor circuits take two binary numbers as input and subtract one binary number input from the other binary number input. Similar to adders, it gives out two outputs, difference and borrow (carry-in the case of Adder).

There are two types of subtractors.

- Half Subtractor
- Full Subtractor

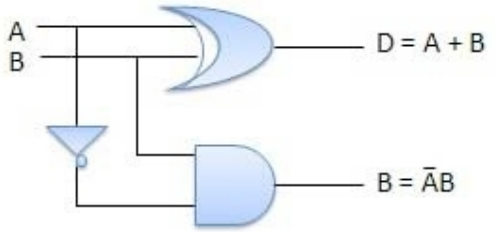
3.4.1 HALF SUBTRACTOR

Half subtractor is a combination circuit with two inputs and two outputs (difference and borrow). It produces the difference between the two binary bits at the input and also produces an output (Borrow) to indicate if a 1 has been borrowed. In the subtraction (A-B), A is called as Minuend bit and B is called as Subtrahend bit.

Table 3-2 Truth Table

Inputs		Output	
A	B	(A – B)	Borrow
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Figure 3-5 Circuit Diagram



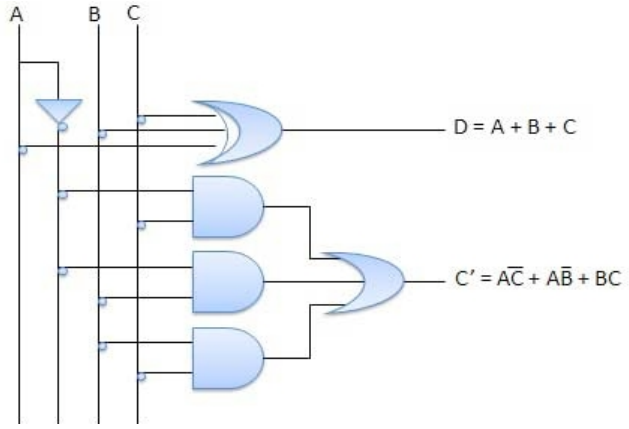
3.4.2 FULL SUBTRACTOR

The disadvantage of a half subtractor is overcome by full subtractor. The full subtractor is a combinational circuit with three inputs A,B,C and two output D and C'. A is the 'minuend', B is 'subtrahend', C is the 'borrow' produced by the previous stage, D is the difference output and C' is the borrow output.

Table 3-3 Truth Table

Inputs			Output	
A	B	C	(A-B-C)	C'
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Figure 3-6 Circuit Diagram



The subtraction can be carried out by taking the 1's or 2's complement of the number to be subtracted. For example we can perform the subtraction (A-B) by adding either 1's or 2's complement of B to A. That means we can use a binary adder to perform the binary subtraction.

3.5 DECODER

Discrete quantities of information are represented in digital computers with binary codes. A binary code of n bits is capable of representing up to 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 $m \leq 2^n$. 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 3-7. The three data inputs, A_0, A_1, and A_2 are decoded into eight outputs, each output representing one of the combinations of the three binary input variables. The three inverters provide the complement of the inputs. 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.

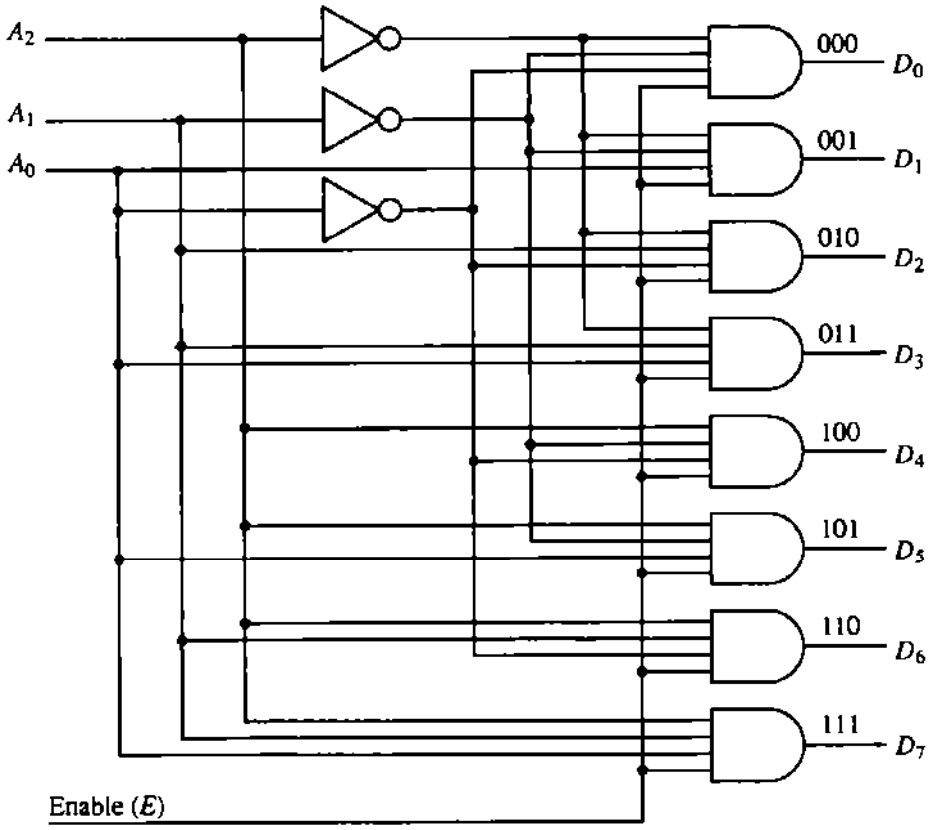


Figure 3-7 3-to-8-line decoder

Commercial decoders include one or more enable inputs to control the operation of the circuit. The decoder of Figure 3-7 has one enable input, E. The decoder is enabled when E is equal to 1 and disabled when E is equal to 0.

The operation of the decoder can be clarified using the truth table listed in Table 3-4. 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.

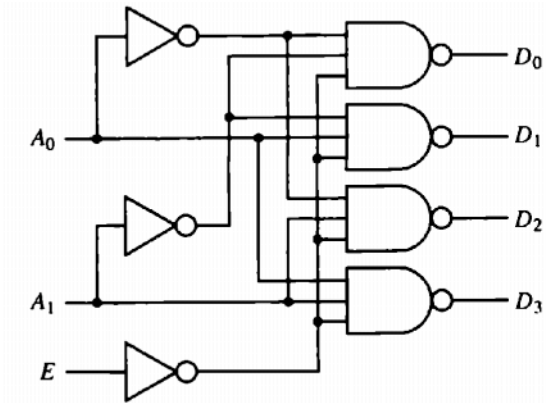
Table 3-4 Truth Table for 3-to-8-Line Decoder

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

3.5.1 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 3-8. 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 A_1 and A_0. The circuit is disabled when E is equal to 1, regardless of the values of the other two inputs.

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 Some decoders have two or more enable inputs that must satisfy a given logic condition in order to enable the circuit.



(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	x	X	1	1	1	1

(b) Truth table

Figure 3-8 2-to-4-line decoder with NAND gates

3.5.2 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.

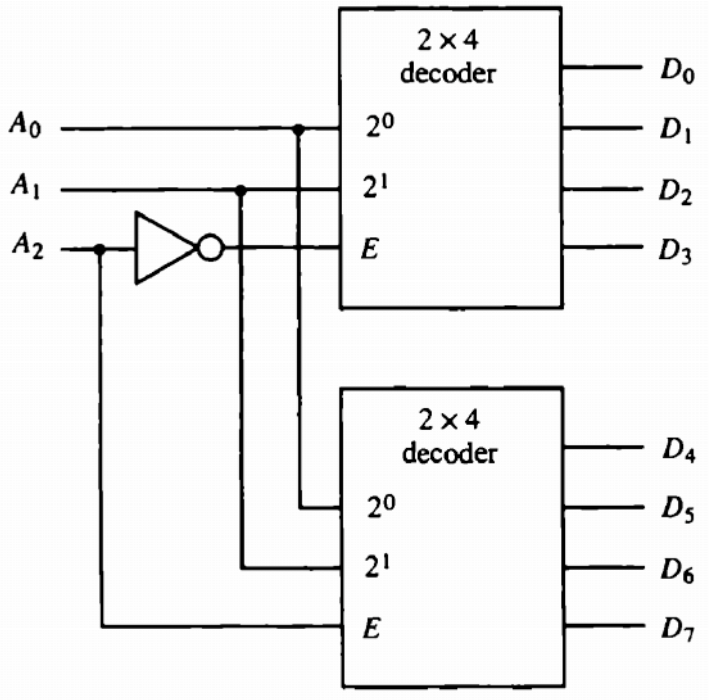


Figure 3-9 A 3 x 8 decoder constructed with two 2 x 4 decoders.

Figure 3-9 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 enabled and the lower is disabled. The lower decoder outputs become inactive with all outputs at 0. The outputs of the upper decoder generate outputs D_0 through D_3, depending on the values of A_1 and A_0 (while $A_2 = 0$). When $A_2 = 1$, the lower decoder is enabled and the upper is disabled. The lower decoder output generates the binary equivalent D_4 through D_7 since these binary numbers have a 1 in the A_2 position.

3.5.3 Encoders

An encoder is a digital circuit that performs the inverse operation of the decoder. An encoder has 2^n (or less) input lines and n output lines. The output lines generate the binary code corresponding to the input value. An example of an encoder is the octal-to-binary encoder, whose truth table is given in Table 3-5. 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.

Table 3-5 Truth Table for Octal-to-Binary Encoder

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

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.

3.6 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.

A 4-to-1-line multiplexer is shown in Figure 3-10. Each of the four data inputs I_0 through I_3 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_2 has two of its inputs equal to 1. The third 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.

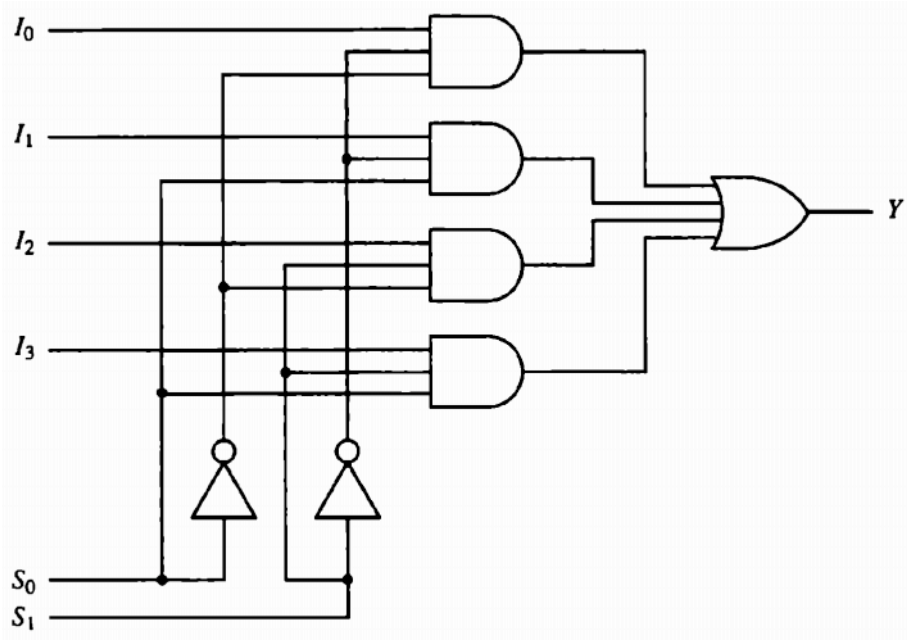


Figure 3-10 4-to-1 line multiplexer

The 4-to-1 line multiplexer of Figure 3-10 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. This is an excessively long table and will not be shown here. A more convenient way to describe the operation of multiplexers is by means of a function table. The function table for the multiplexer is shown in Table 3-6. 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, input I_1 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.

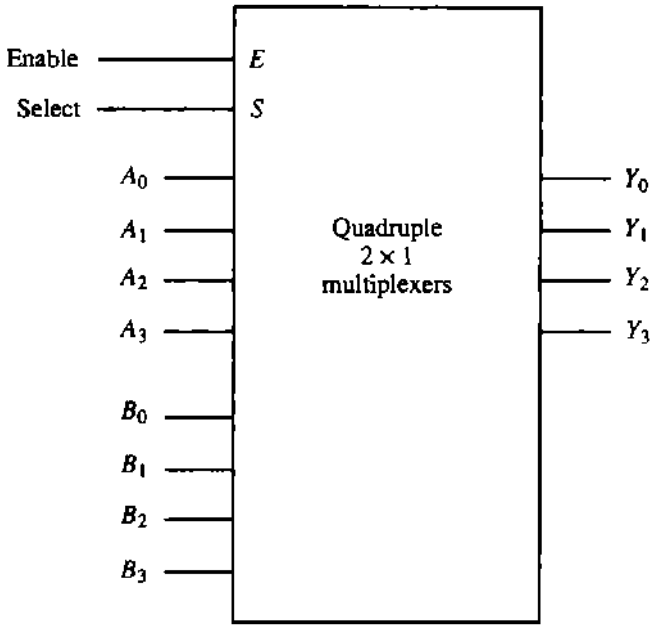
Table 3-6 Function Table for 4-to-1 Line Multiplexer

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^n-to-1-line multiplexer is constructed from an n-to-2^n decoder by adding to it 2^n input lines, one from each data input. The size of the multiplexer is specified by the number 2^n 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 as disabled, and when it is in the active state, the circuit functions as a normal multiplexer. The enable input is useful for the 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. As an illustration, the block diagram of a quadruple 2-to-1-line multiplexer is shown in Figure 3-11. The Circuit has four multiplexers, each capable of selecting one of two input lines. Output Y_0 can be selected to come from either input A_0 or B_0. Similarly, output Y_1 may have the value of A_1 or B_1, and so on. One input selection line S selects one of the lines in each of the four multiplexers. The enable input E must be active for normal operation. Although the circuit contains four multiplexers, we can also think of it as a circuit that selects one of two 4-bit data lines. As shown in the function table, the unit is enabled when E = 1. Then, if S = 0, the four A inputs have a path to the four outputs. On the other hand, if S = 1, the four B inputs are applied to the outputs. The outputs have all 0's when E = 0, regardless of the values of S.



(a) Block Diagram

E	S	Y
0	x	All 0's
1	0	A
1	1	B

(b) Function Table

Figure 3-11 Quadruple 2-to-1 line multiplexers

3.7 DE-MULTIPLEXER

De-Multiplexer is a combinational circuit that performs the reverse operation of Multiplexer. It has single input, 'n' selection lines and maximum of 2^n outputs. The input will be connected to one of these outputs based on the values of selection lines. Since

there are 'n' selection lines, there will be 2^n possible combinations of zeros and ones. So, each combination can select only one output. De-Multiplexer is also called as De-Mux. 1x4 De-Multiplexer of Figure 3-12 has one input I, two selection lines, s_1 & s_0 and four outputs Y_3, Y_2, Y_1 &Y_0.

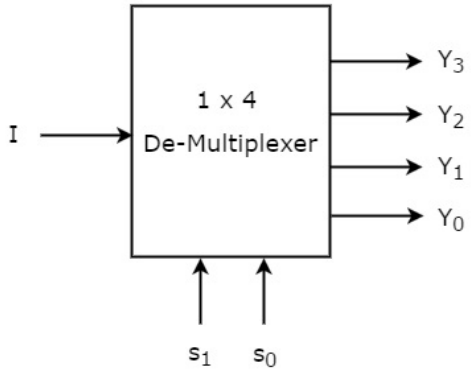


Figure 3-12 Block diagram of 1x4 De-Multiplexer

The single input 'I' will be connected to one of the four outputs, Y_3 to Y_0 based on the values of selection lines s_1 & s0.

Table 3-7 Truth table of 1x4 De-Multiplexer

Selection Inputs		Outputs			
S_1	S_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	I
0	1	0	0	I	0
1	0	0	I	0	0
1	1	I	0	0	0

From the Truth table 3-7, we can directly write the Boolean functions for each output as

$$Y_3 = s_1 s_0 I$$

$$Y_2 = s_1 s_0' I$$

$$Y_1 = s_1' s_0 I$$

$$Y_0 = s_1' s_0' I$$

We can implement these Boolean functions using Inverters & 3-input AND gates. The circuit diagram of 1x4 De-Multiplexer is shown in the Figure 3-13.

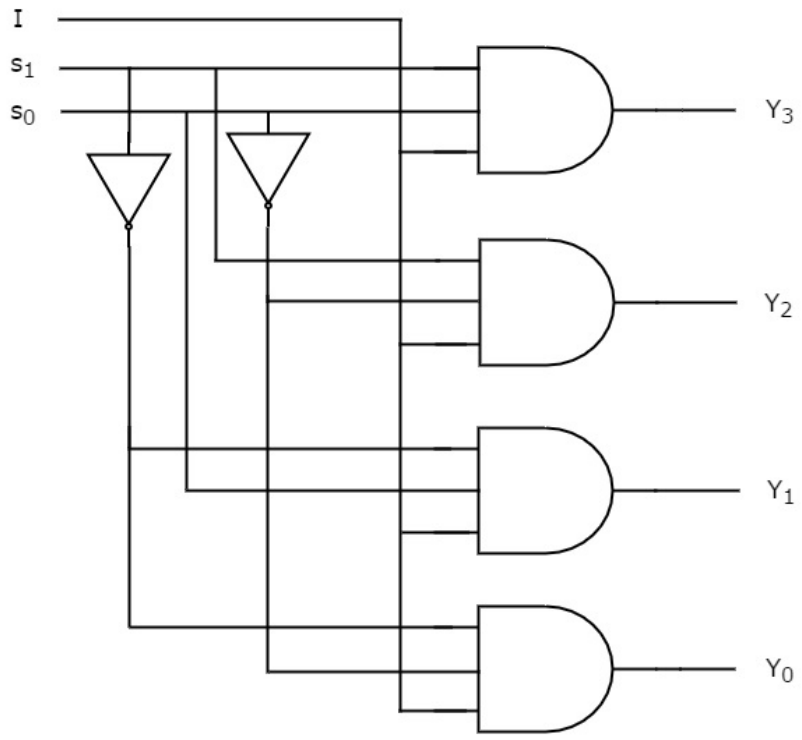


Figure 3-13 Circuit diagram of 1x4 De-Multiplexer

We can easily understand the operation of the above circuit. Similarly, you can implement 1x8 De-Multiplexer and 1x16 De-Multiplexer by following the same procedure.

3.8 CHECK YOUR PROGRESS

1. If A and B are the inputs of a half adder, the sum is given by _____
2. Which are the fundamental inputs assigned or configured in the full adder circuit ?
3. Half subtractor is used to perform subtraction of _____ 2 bits
4. For subtracting 1 from 0, we use to take a _____ from neighbouring bits.
5. Let the input of a subtractor is A and B then what the output will be if A = B?
6. A decoder converts n inputs to _____ outputs.
7. Decoder is constructed from _____
8. In a multiplexer, the enable input is also known as _____

9. If the number of n selected input lines in multiplexer is equal to 2^m then it requires _____ select lines.
10. How many select lines would be required for an 8-line-to-1-line multiplexer?

3.9 SUMMARY

Let's take a quick review of all that we have learned about combinational logic.

1. We began by introducing the combinational circuit and discussing its design procedure. We learned that the Combinational circuit is employed in digital computers for generating binary control decisions and for providing digital components required for data processing.
2. Two types of adders, one that performs the arithmetic addition of two bits is called a half-adder. And second one, performs the addition of three bits (two significant bits and a previous carry) is called a full-adder.
3. Subtractors take two binary numbers as input and subtract one binary number input from the other binary number input. Similar to adders, it gives out two outputs, difference and borrow (carry-in the case of Adder).
4. A Decoder converts binary information from the n coded inputs to a maximum of 2^n unique outputs. And an encoder is a digital circuit that performs the inverse operation of the decoder. An encoder has 2^n (or less) input lines and n output lines.
5. A multiplexer that receives binary information from one of 2^n input data lines and directs it to a single output line. And a De-Multiplexer that performs the reverse operation of Multiplexer. It has single input, 'n' selection lines and maximum of 2^n outputs.

3.10 KEYWORDS

1. **Adder** is a digital circuit that performs addition of numbers.
2. **Subtractor** is a circuit which can subtract a binary digit from another one.
3. **Decoder** is used to change the code into a set of signals.

4. **Encoder** is a combinational circuit that performs the reverse operation of Decoder.
5. **multiplexer** (or mux; spelled sometimes as multiplexor), also known as a data selector, is a device that selects between several analog or digital input signals and forwards it to a single output line.
6. **De-multiplexer** (or de-mux or data distributor) takes a single input line and routes it to one of several digital output lines.

3.11 SELF ASSESSMENT TEST

1. Define adder. Differentiate between half adder and full adder.
2. What is subtractor? Explain its types.
3. Implement the 3 to 8 decoder using 2 to 4 decoders.
4. What is the working principle of encoder?
5. What are multiplexer and de-multiplexer? Implement 8x1 multiplexer using 4x1 multiplexers and 2x1 multiplexer.

3.12 ANSWER TO CHECK YOUR PROGRESS

1. A XOR B
2. Addend, Augend & Input Carry
3. 2 bits
4. Borrow
5. 0
6. 2^n
7. Inverters and AND gates
8. Strobe
9. m
10. 3

3.13 REFERENCES / SUGGESTED READINGS

1. Computer Organization and Architecture, Rajaram & Radhakrishan, PHI.
2. Computer Organization & Architecture: Designing for Performance, Stalling, PHI.
3. Computer Organization and Design, Pal Choudhary, PHI.
4. Computer Systems Organization & Architecture, Carpenelli, Pearson Education.
5. Computer Organization and Architecture, Stalling, Pearson Education.
6. Computer System Architecture, Morris Mano, PHI.
7. Computer Architecture and Organization, McGraw Hill Company, New Delhi. J.P. Hayes.

SUBJECT: COMPUTER SYSTEM ARCHITECTURE	
COURSE CODE: MCA-24	**AUTHOR: NEERAJ VERMA**
LESSON NO. 4	

SEQUENTIAL LOGIC

STRUCTURE

4.0 LEARNING OBJECTIVES

After reading this unit, you should be able to:

1. Define sequential circuit with its behavior and graphically representation.
2. Understand clock and its classification
3. Understand the design and working principle of various Flip-Flops (S-R FF, J-K FF, D FF, T FF and Edge-Triggered FF).
4. Define Shift Register and understand bidirectional Shift Register with Parallel Load
5. Know the Binary Counter with 4-bit synchronous binary counter graphically representation.

4.1 INTRODUCTION

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 Figure 4-1. 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.

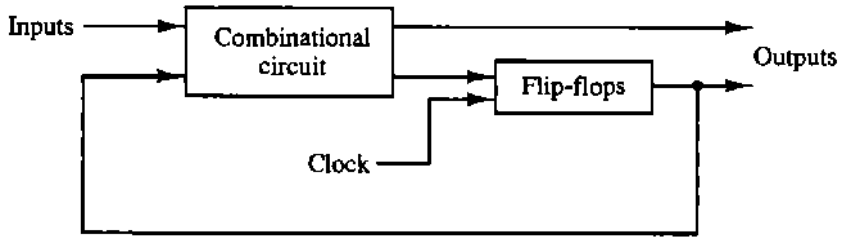


Figure 4-1 Block diagram of a clocked synchronous sequential circuit

4.2 FLIP-FLOPS

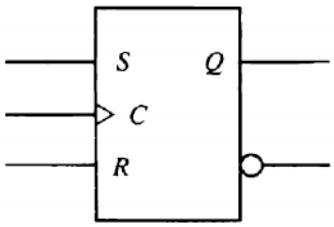
The digital circuits considered thus far have been combinational, where the outputs at any given time are entirely dependent on the inputs that are present at that time. Although every digital system is likely to have a combinational circuit, most systems encountered in practice also include storage elements, which require that the system be described in terms of sequential circuits. The most common type of sequential circuit is the synchronous type. Synchronous sequential circuits employ signals that affect the storage elements only at discrete instants of time. Synchronization is achieved by a timing device called a clock pulse generator that produces a periodic train of clock pulses. The clock pulses are distributed throughout the system in such a way that storage elements are affected only with the arrival of the synchronization pulse. Clocked synchronous sequential circuits are the type most frequently encountered in practice. They seldom manifest instability problems and their timing is easily broken down into independent discrete steps, each of which may be considered separately.

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 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.

4.2.1 SR FLIP-FLOP

The graphic symbol of the SR flip-flop is shown in Figure 4-2(a). It has three inputs, labeled 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.

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 cleared to 0. If both S and R are 0 during the clock transition, the output does not change. When both S and R are equal to 1, the output is unpredictable and may go to either 0 or 1, depending on internal timing delays that occur within the circuit.



(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

Figure 4-2 SR Flip-Flop

The characteristic table shown in Figure 4-2(b) summarizes the operation of the SR flip-flop in tabular form. The 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 the clock transition (referred to as next state). If S = R = 0, a clock transition produces no change of state [i.e., Q(t+1) = Q(t)]. If S = 0 and R = 1, the flip-flop goes to the 0 (clear) state. If S = 1 and R = 0, the flip-flop goes to the 1 (set) state. The SR flip-flop should not be pulsed when S = R = 1 since it produces an indeterminate next state. This indeterminate condition makes the SR flip-flop difficult to manage and therefore it is seldom used in practice.

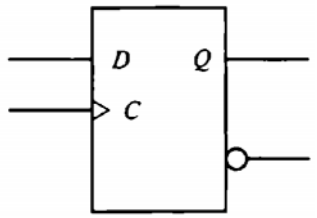
4.2.2 D FLIP-FLOP

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 Figure 4-3. 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:

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

This means that the Q output of the flip-flop receives it s value from the D input every time that the clock signal goes through a transition from 0 to 1.



(a) Graphic symbol

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

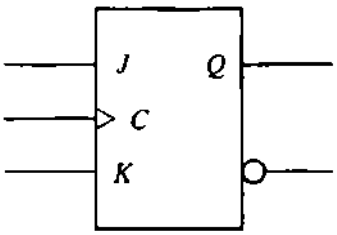
(b) Characteristic table

Figure 4-3 D Flip-Flop

Note that no input condition exists that will leave the state of the D flip-flop unchanged. 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.

4.2.3 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.



(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

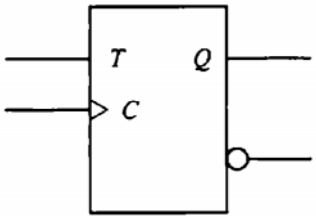
(b) Characteristic table

Figure 4-4 JK Flip-Flop

The graphic symbol and characteristic table of the JK flip-flop are shown in Figure 4-4. 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.

4.2.4 T FLIP-FLOP

Another type of flip-flop found in textbooks is the T (toggle) flip-flop. This flip-flop, shown in Figure 4-5, is obtained from a JK type when inputs J and K are connected to provide a single input designated by T. The T flip-flop therefore has only two conditions.



(a) Graphic symbol

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

(b) Characteristic table

Figure 4-5 T Flip-Flop

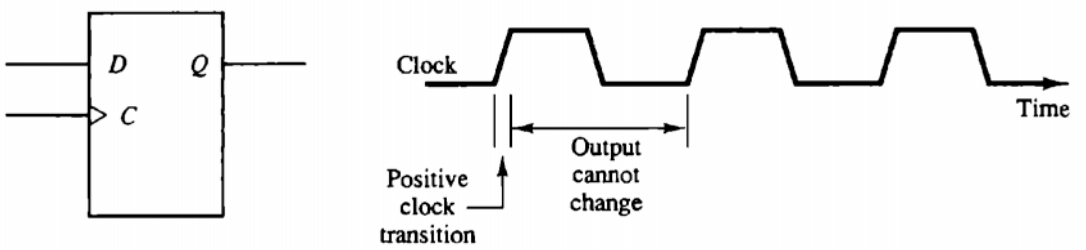
When T = 0 (J = K = 0) a clock transition does not change the state of the flip-flop. When T = 1 (J = K = 1) a clock transition complements the state of the flip-flop. These conditions can be expressed by a characteristic equation:

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

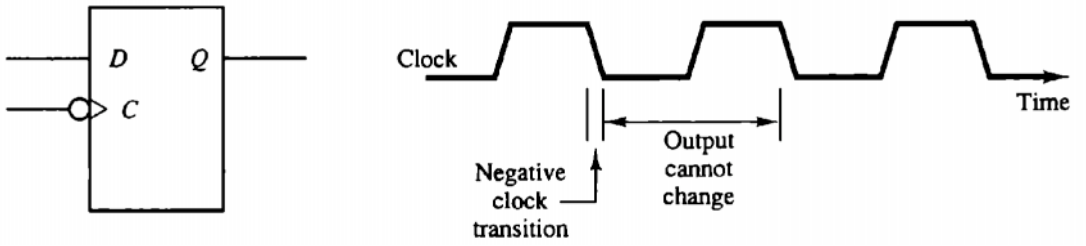
4.2.5 EDGE-TRIGGERED FLIP-FLOPS

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 edge (positive-edge transition), and others cause a transition on the falling edge (negative-edge transition).

Figure 4-6(a) shows the clock pulse signal in a positive-edge-triggered D flip-flop. The value in the D input is transferred to the Q output when the clock makes a positive transition. The output cannot change when the clock is in the 1 level, in the 0 level, or in a transition from the 1 level to the 0 level.



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



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

Figure 4-6 Edge-triggered flip-flop

The effective positive clock transition includes a minimum time called the setup time in which the D input must remain at a constant value before the transition, and a definite time called the hold time in which the D input must not change after the positive transition. The effective positive transition is usually a very small fraction of the total period of the clock pulse.

Figure 4-6(b) shows the corresponding graphic symbol and timing diagram for a negative-edge-triggered D flip-flop. The graphic symbol includes a negation small circle in front of the dynamic indicator at the C input. This denotes a negative-edge-triggered behavior. In this case the flip-flop responds to a transition from the 1 level to the 0 level of the clock signal.

Another type of flip-flop used in some systems is the master-slave flip-flop. This type of circuit consists of two flip-flops. The first is the master, which responds to the positive level of the clock, and the second is the slave, which responds to the negative level of the clock. The result is that the output changes during the 1-to-0 transition of the clock signal. The trend is away from the use of master-slave flip-flops and toward edge-triggered flip-flops.

Flip-flops available in integrated circuit packages will sometimes provide special input terminals for setting or clearing the flip-flop asynchronously. These inputs are usually called “preset” and “clear”. They affect the flip-flop on a negative level of the input signal

without the need of a clock pulse. These inputs are useful for bringing the flip-flops to an initial state prior to its clocked operation.

4.2.6 EXCITATION TABLE

The characteristic tables of flip-flops specify the next state when the inputs and the present state are known. During the design of sequential circuits we usually know the required transition from present state to next state and wish to find the flip-flop input conditions that will cause the required transition. For this reason we need a table that lists the required input combinations for a given change of state. Such a table is called a flip-flop excitation table.

Table 4-1 lists the excitation tables for the four types of flip-flops. Each table consists of two columns, Q(t) and Q(t + 1), and a column for each input to show how the required transition is achieved. There are four possible transitions from present state Q(t) to next state Q(t + 1). The required input conditions from present state Q(t) to next state Q(t + 1).

Table 4-1 Excitation Table for Four Flip-Flops

SR flip-flop

Q(t)	Q(t + 1)	S	R
0	0	0	x
0	1	1	0
1	0	0	1
1	1	x	0

D flip-flop

Q(t)	Q(t + 1)	D
0	0	0
0	1	1
1	0	0
1	1	1

JK flip-flop

Q(t)	Q(t + 1)	J	K
0	0	0	x
0	1	1	x
1	0	x	1
1	1	x	0

T flip-flop

Q(t)	Q(t + 1)	T
0	0	0
0	1	1
1	0	1
1	1	0

The required input conditions for each of these transitions are derived from the information available in the characteristic tables. The symbol x in the tables represents a don't-care condition; that is, it does not matter whether the input to the flip-flop is 0 or 1.

The reason for the don't-care conditions in the excitation tables is that there are two ways of achieving the required transition. For example, in a JK flip-flop, a transition from present state of 0 to a next state of 0 can be achieved by having inputs J and K equal to 0 (to obtain no change) or by letting J = 0 and K = 1 to clear the flip-flop (although it is already cleared). In both cases J must be 0, but K is 0 in the first case and 1 in the second. Since the required transition will occur in either case, we mark the K input with a don't-care and let the designer choose either 0 or 1 for the K input, whichever is more convenient.

4.3 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 4-8. 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. When the shift register of Figure 4-7 is used, the shift can be controlled by connecting the clock to the input of an AND gate, and a second input of the AND gate can then control the shift by inhibiting the clock. However, it is also possible to provide extra circuits to control the shift operation through the D inputs of the flip-flops rather than the clock input.

4.3.1 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. Some shift registers provide, the necessary input and output terminals for parallel transfer. The

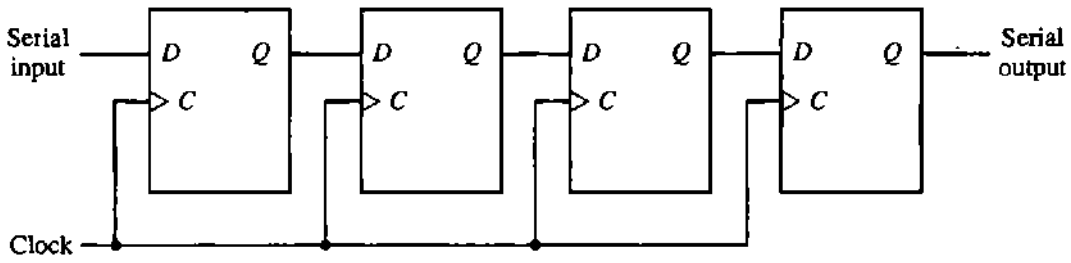


Figure 4-7 4-bit shift register

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 shift-right.
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 Figure 4-8. Each stage consists of a D flip-flop and a 4x1 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 Table 4-2. 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 $S_1S_0 = 11$, the binary information from each input I0 through I3 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.

Table 4-2 Function Table for Register of Figure 4-9

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

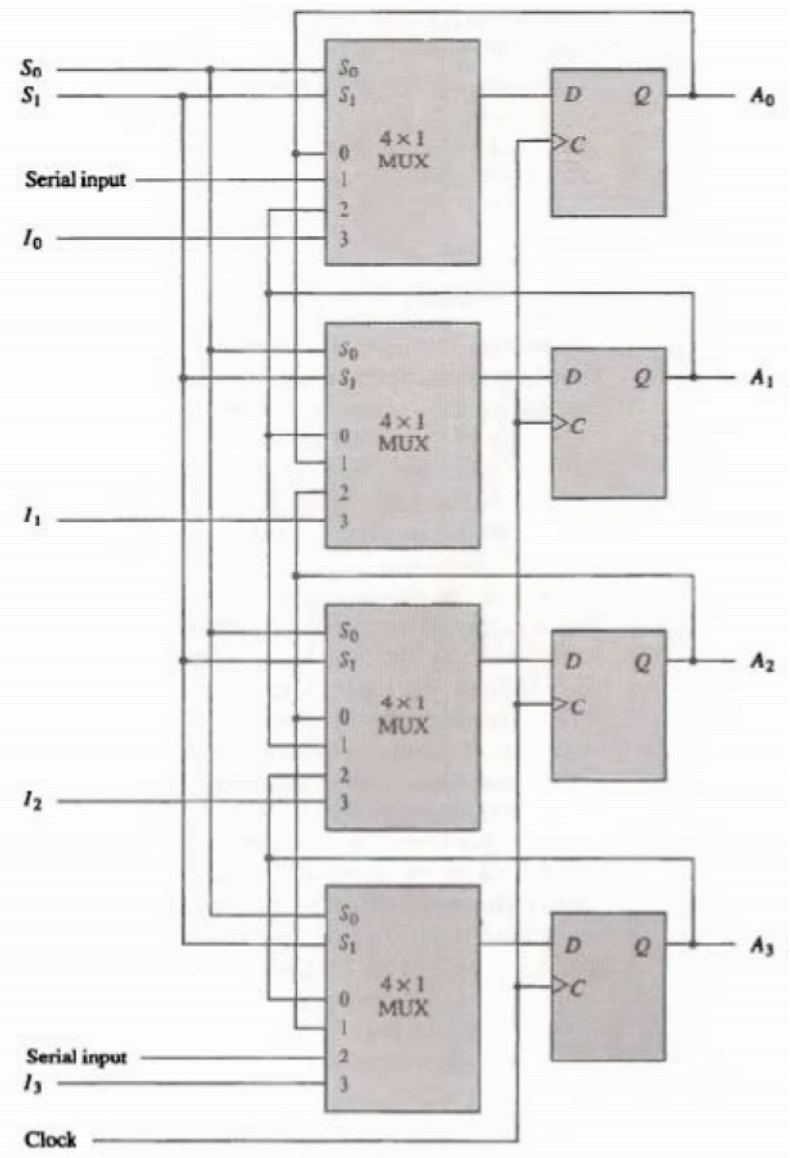


Figure 4-8 Bidirectional shift register with parallel load.

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

4.4 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 form 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 $2^n - 1$. A simpler alternative design procedure may be carried out from a direct inspection of the sequence of states that the register must undergo to achieve a straight binary count.

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 all 1's.

A counter circuit will usually employ flip-flops with complementing capabilities. Both T and JK flip-flops have this property. Remember that a JK flip-flop is complemented if both its J and K inputs are 1 and the clock goes through a positive transition. The output of the flip-flop does not change if J = K = 0. In addition, the counter may be controlled with an enable input that turns the counter on or off without removing the clock signal from the flip-flops.

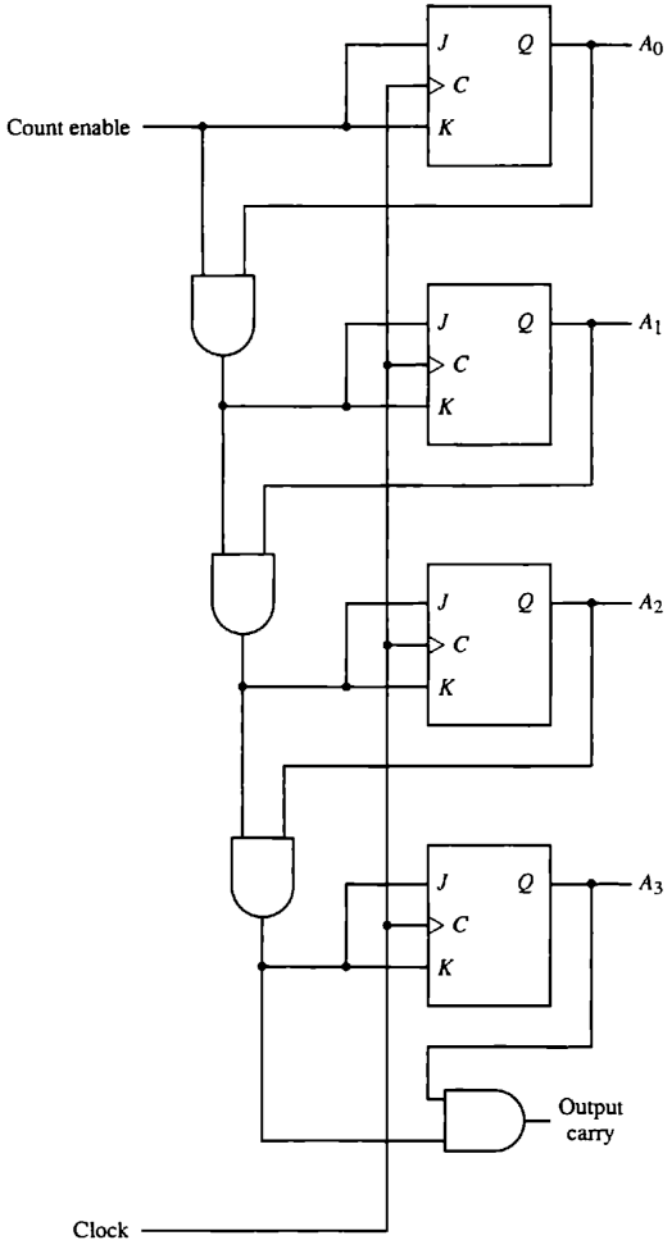


Figure 4-9 4-bit synchronous binary counter.

Synchronous binary counters have a regular pattern, as can be seen from the 4-bit binary counter shown in Figure 4-9. The C inputs of all flip-flops receive the common clock. If the count enable is 0, all J and K inputs are maintained at 0 and the output of the counter does not change. The first stage A0 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. The output carry can be used to extend the counter to more stages, with each stage having an additional flip-flop and an AND gate.

4.4.1 Binary Counter with Parallel Load

Counters employed in digital systems quite often require a parallel load capability for transferring an initial binary number prior to the count operation. Figure 4-10 shows the logic diagram of a binary counter that has a parallel load capability and can also be cleared to 0 synchronous with the clock. When equal to 1, the clear input sets all the K inputs to 1, thus clearing all flip-flops with the next clock transition. The input load control when equal to 1, disables the count operation and causes a transfer of data from the four parallel inputs into the four flip-flops (provided that the clear input is 0). If the clear and load inputs are both 0 and the increment input is 1, the circuit operates as a binary counter.

Table 4-3 Function Table for the register of Figure 4-11

Clock	Clear	Load	Increment	Operation
↑	0	0	0	No change
↑	0	0	1	Increment count by 1
↑	0	1	x	Load inputs I_0 through I_3
↑	1	x	x	Clear outputs to 0

The operation of the circuit is summarized in Table 4-3. With the clear, load, and increment inputs all at 0, the outputs do not change even when pulses are applied to the C terminals. If the clear and load inputs are maintained at logic 0, the increment input controls the operation of the counter and the outputs change to the next binary count for each positive transition of the clock. The input data are loaded into the flip-flops when the load control input to equal to 1 provided that the clear is disabled, but the increment input

can be 0 or 1. The register is cleared to 0 with the clear control regardless of the values in the load and increment inputs.

Counters with parallel load are very useful in the design of digital computers. In subsequent chapters we refer to them as registers with load and increment operations. The increment operation adds one to the content of a register. By enabling the count input during one clock period, the content of the register can be incremented by one.

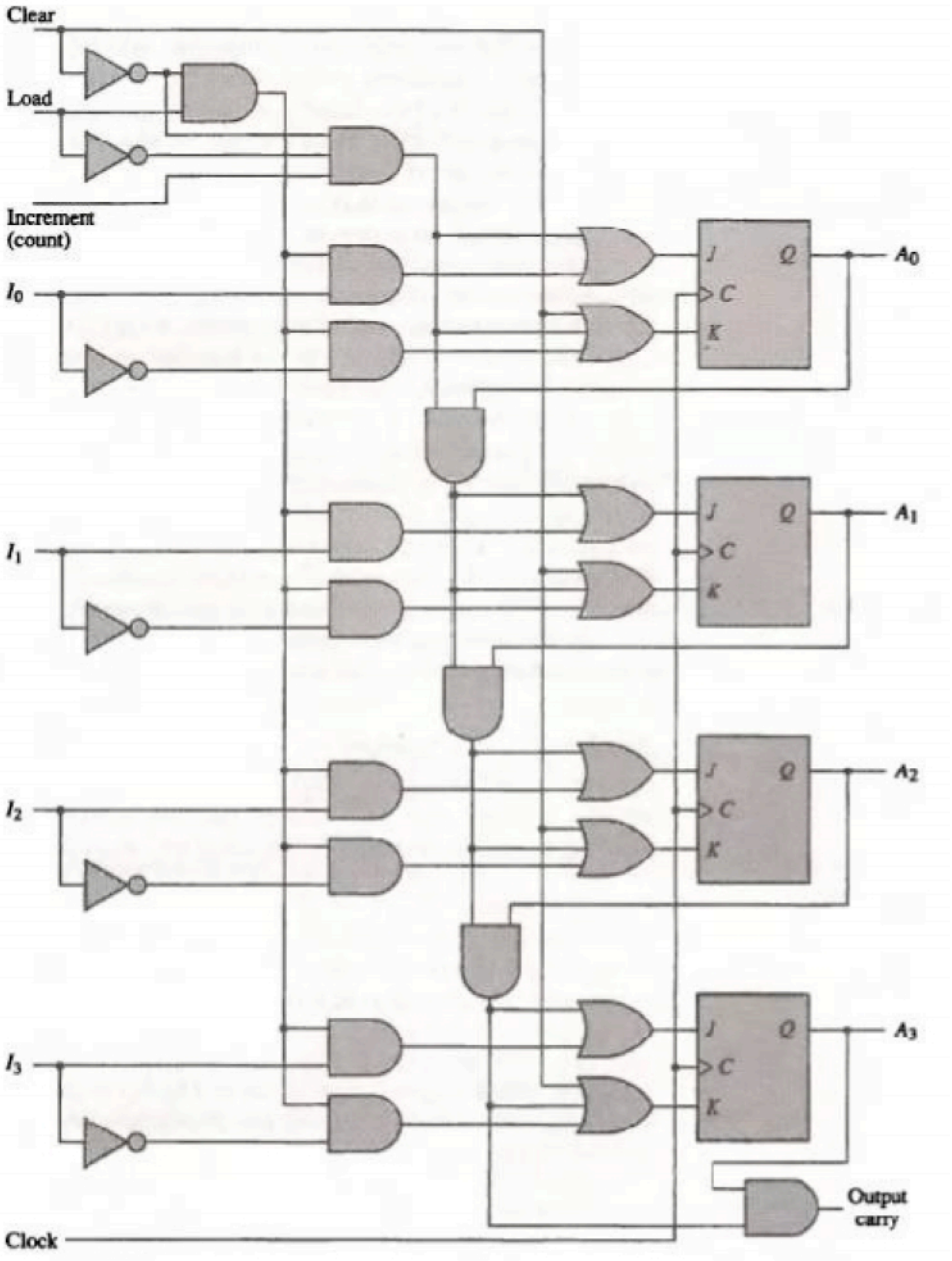


Figure 4-10 4-bit binary counter with parallel load and synchronous clear

4.5 CHECK YOUR PROGRESS

1. How many flip-flops are required to produce a divide-by-128 device?
2. When is a flip-flop said to be transparent?
3. A J-K flip-flop is in a "no change" condition when _____.
4. How is a *J-K* flip-flop made to toggle?
5. On a positive edge-triggered S-R flip-flop, the outputs reflect the input condition when _____.
6. What is the hold condition of a flip-flop?
7. Edge-triggered flip-flops must have _____.
8. A serial in/parallel out, 4-bit shift register initially contains all 1s. The data nibble 0111 is waiting to enter. After four clock pulses, the register contains _____.
9. What is/are the directional mode/s of shifting the binary information in a shift register?
10. A counter circuit is usually constructed of _____.

4.6 SUMMARY

This lesson covered several kinds of flip-flops. You have become familiar with the circuit diagrams for and the operation of five types of common flip-flops. We consider using flip-flops in the presence of a clock signal. The clock signal causes the flip-flop to sample the value of the input towards the end of a clock cycle and output the sampled value during the next clock cycle. Flip-flops play a crucial role in bounding the segments of time during which signals may be instable.

In a sense, flip-flops and combinational circuits have opposite roles. Combinational circuits compute interesting Boolean functions but increase uncertainty (namely, lengthen segments of time during which signals may be instable). Flip-flops, one the other hand, output the same value that is fed as input but they limit uncertainty.

We have learned about shift registers. A register capable of shifting its binary information in one or both directions is called a shift register. We studied 4-bit bidirectional shift register with parallel load with diagram.

We learned about counters and shift registers. A register that goes through a predetermined sequence of states upon the application of input pulses is called a counter. We studied 4-bit synchronous binary counters and binary counter with parallel load with diagram.

4.7 KEYWORDS

1. **Flip-Flop** is a special type of gated latch circuit.
2. **SR Flip-Flop** operates with only positive clock transitions or negative clock transitions. Whereas, SR latch operates with enable signal.
3. **JK Flip-Flop** is the modified version of SR flip-flop. It operates with only positive clock transitions or negative clock transitions.
4. **D Flip-Flop** is also called as DATA or delay flip flop & it is stores a bit of data. It operates with only positive clock transitions or negative clock transitions.
5. **T Flip-Flop** is the simplified version of JK flip-flop. It is obtained by connecting the same input 'T' to both inputs of JK flip-flop. It operates with only positive clock transitions or negative clock transitions.
6. **Edge-Triggered Flip-Flop** changes states either at the positive edge (rising edge) or at the negative edge (falling edge) of the clock pulse on the control input.
7. **Register** is a group of flip-flops, which are used to hold/store the binary data.
8. **Shift Register** is capable of shifting bits either towards right hand side or towards left hand side.
9. **Counter** is a device which stores (and sometimes displays) the number of times a particular event or process has occurred.

4.8 SELF ASSESSMENT TEST

1. What is flip-flop?
2. Why we use SR flip flop?

3. How does JK flip-flop work?
4. Why JK flip flop is called universal flip-flop?
5. What is toggle condition?
6. What is the difference between D flip-flop and T flip-flop?
7. Why do we use counter?
8. How does a 4 bit shift register work?

4.9 ANSWER TO CHECK YOUR PROGRESS

1. 7
2. when the *Q* output follows the input
3. J = 0, K = 0
4. J = 1, K = 1
5. the clock pulse transitions from LOW to HIGH
6. no active *S* or *R* input
7. positive edge-detection circuits
8. 0111
9. Left - Right
10. A number of flip-flops connected in cascade

4.10 REFERENCES / SUGGESTED READINGS

1. Computer Organization and Architecture, Rajaram & Radhakrishan, PHI.
2. Computer Organization & Architecture: Designing for Performance, Stalling, PHI.
3. Computer Organization and Design, Pal Choudhary, PHI.
4. Computer Systems Organization & Architecture, Carpenelli, Pearson Education.
5. Computer Organization and Architecture, Stalling, Pearson Education.
6. Computer System Architecture, Morris Mano, PHI.
7. Computer Architecture and Organization, McGraw Hill Company, New Delhi. J.P. Hayes.

SUBJECT: COMPUTER SYSTEM ARCHITECTURE	
COURSE CODE: MCA-24	**AUTHOR: DR. MANOJ DUHAN**
LESSON NO. 5	

BASIC COMPUTER ORGANIZATION AND DESIGN

REVISED / UPDATED SLM BY NEERAJ VERMA

STRUCTURE

5.0 LEARNING OBJECTIVES

After reading this lesson, you should be able to:

1. Know the Instruction Representation
2. Describe the various components of computer stored program organization
3. Recognize different types of computer registers
4. Understand the concept of Timing and Control
5. Define functionality of Instruction Cycles
6. Elaborate complete computer description

5.1 INTRODUCTION

In this lesson, we introduce a basic computer and show how its operation can be specified with register transfer statements. The organization of the computer is defined by its internal registers, the timing and control structure, and the set of instructions that it uses. The design of the computer is then carried out in detail. Although the basic computer presented in this unit is very small compared to commercial computers, it has the advantage of being simple enough so we can demonstrate the design process without too many complications.

The internal organization of a digital system is defined by the sequence of microoperations it performs on data stored in its registers. The general purpose digital computer is capable of executing various microoperations and, in addition, can be instructed as to what specific sequence of operations it must perform. The user of a computer can control the process by means of a program. A program is a set of instructions that specify the operations, operands, and the sequence by which processing has to occur. The data processing task may be altered by specifying a new program with different instructions or specifying the same instructions with different data.

A computer instruction is a binary code that specifies a sequence of micro operations for the computer. Instruction codes together with data are stored in memory. The computer reads each instruction from memory and places it in a control register. The control then interprets the binary code of the instructions and proceeds to execute it by issuing a sequence of micro operations. Every computer has its own unique instruction set. The

ability to store and execute instructions, the stored program concept, is the most important property of a general-purpose computer.

An instruction code is a group of bits that instruct the computer to perform a specific operation. It is usually divided into parts, each having its own particular interpretation. The most basic part of an instruction code is its operation part. The operation code of an instruction is a group of bits that define such operations as add, subtract, multiply, shift, and complement. The number of bits required for the operation code of an instruction depends on the total number of operations available in the computer. The operation code must consists of at least n bits for a given 2^n (or less) distinct operations. As an illustration, consider a computer with 64 distinct operations, one of them being an ADD operation. The operation code consists of six bits, with a bit configuration 110010 assigned to the ADD operation. When this operation code is decoded in the control unit, the computer issues control signals to read an operand from memory and add the operand to a processor register.

At this point we must recognize the relationship between a computer operation and a micro operation. An operation is part of an instruction stored in computer memory. It is a binary code tells the computer to perform a specific operation. The control unit receives the instruction from memory and interprets the operation code bits. It then issues a sequence of control signals to initiate microoperations in internal computer registers. For every operation come, the control issues a sequence of microoperations needed for the hardware implementation of the specified operation. For this reason, an operation code is sometimes called a microoperation because it specifies a set of microoperations.

The operation part of an instruction code specifies the operation to be performed. This operation must be performed on some data stored in processor registers or in memory. An instruction code must therefore specify not only the operation but also the registers or the memory words where the operands are to be found, as well as the register or memory word where the result is to be stored. Memory words can be specified in instruction codes by their address. Processor registers can be specified by assigning to the instruction another binary code of k bits that specifies one of 2^k registers. There are many variations for arranging the binary code of instructions, and each computer has its own particular instruction code format. Instruction code formats are conceived computer designers who specify the architecture of the computer. In this lesson we choose a particular instruction code to explain the basic organization and design of digital computers.

5.1.1 STORED PROGRAM ORGANIZATION

The simplest way to organize a computer is to have one processor register and instruction code format with two parts. The first part specifies the operation to be performed and the second specifies an address. The memory address tells the control where to find an operand in memory. This operand is read from memory and used as the data to be operated on together with the data stored in the processor register.

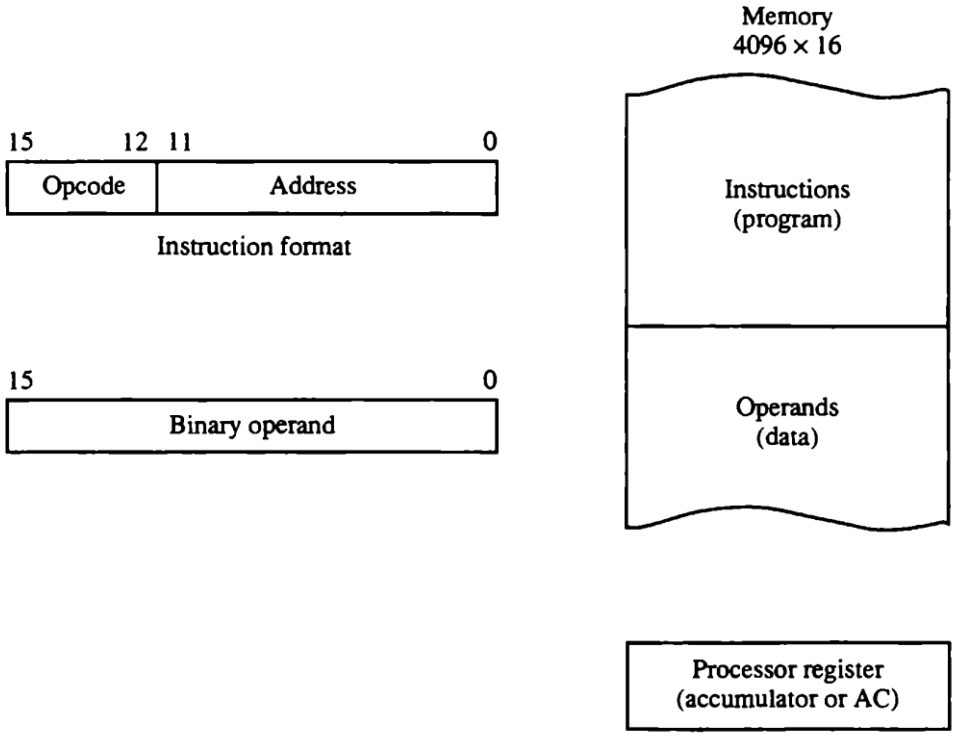


Figure 5-1 Stored program organization

Figure 5-1 depicts this type of organization. Instructions are stored in one section of memory and data in another. For a memory unit with 4096 words we need 12 bits to specify an address since $2^{12} = 4096$. If we store each instruction code in one 16-bit memory word, we have available four bits for the operation code (abbreviated op code) to specify one out of 16 possible operations, and 12 bits to specify the address of an operand. The control reads a 16-bit instruction from the program portion of memory. It uses the 12-bit address part of the instruction to read a 16-bit operand from the data portion of memory. It then executes the operation specified by the operation code. Computers that have a single- processor register usually assign to it the name accumulator and label it AC. The operation is performed with the memory operand and the content of AC.

If an operation in an instruction code does not need an operand from memory, the rest of the bits in the instruction can be used for other purposes. For example, operations such as clear AC, complement AC, and increment AC operate on data stored in the AC register. They do not need an operand from memory. For these types of operations, the second part of the instruction code (bits 0 through 11) is not needed for specifying a memory address and can be used to specify other operations for the computer.

5.1.2 INDIRECT ADDRESS

It is sometimes convenient to use the address bits of an instruction code not as an address but as the actual operand. When the second part of an instruction code specifies an operand, the instruction is said to have an immediate operand. When the second part specifies the address of an operand, the instruction is said to have a direct address. This is in contrast to a third possibility called indirect address, where the bits in the second part of the instruction designate an address of a memory word in which the address of the operand is found. One bit of the instruction code can be used to distinguish between a direct and an indirect address.

As an illustration of this configuration, consider the instruction code format shown in Figure 5-2(a). It consists of a 3-bit operation code, a 12-bit address, and an indirect address mode bit designated by I. The mode bit is 0 for a direct address and 1 for an indirect address. A direct address instruction is shown in Figure 5-2(b). It is placed in address 22 in memory. The I bit is 0, so the instruction is recognized as a direct address instruction. The opcode specifies an ADD instruction, and the address part is the binary equivalent of 457. The control finds the operand in memory at address 457 and adds it to the content of AC. The instruction in address 35 shown in Figure 5-2(c) has a mode bit I = 1. Therefore, it is recognized as an indirect address instruction. The address part is the binary equivalent of 300. The control goes to address 300 to find the address of the operand. The address of the operand in this case is 1350. The operand found in address 1350 is then added to the content of AC. The indirect address instruction needs two references to memory to fetch an operand. The first reference is needed to read the address of the operand; the second is for the operand itself. We define the effective address to be the address of the operand in a computation-type instruction or the target address in a branch-type instruction. Thus the effective address in the instruction of Figure 5-2(b) is 457 and in the instruction of Figure 5-2(c) is 1350.

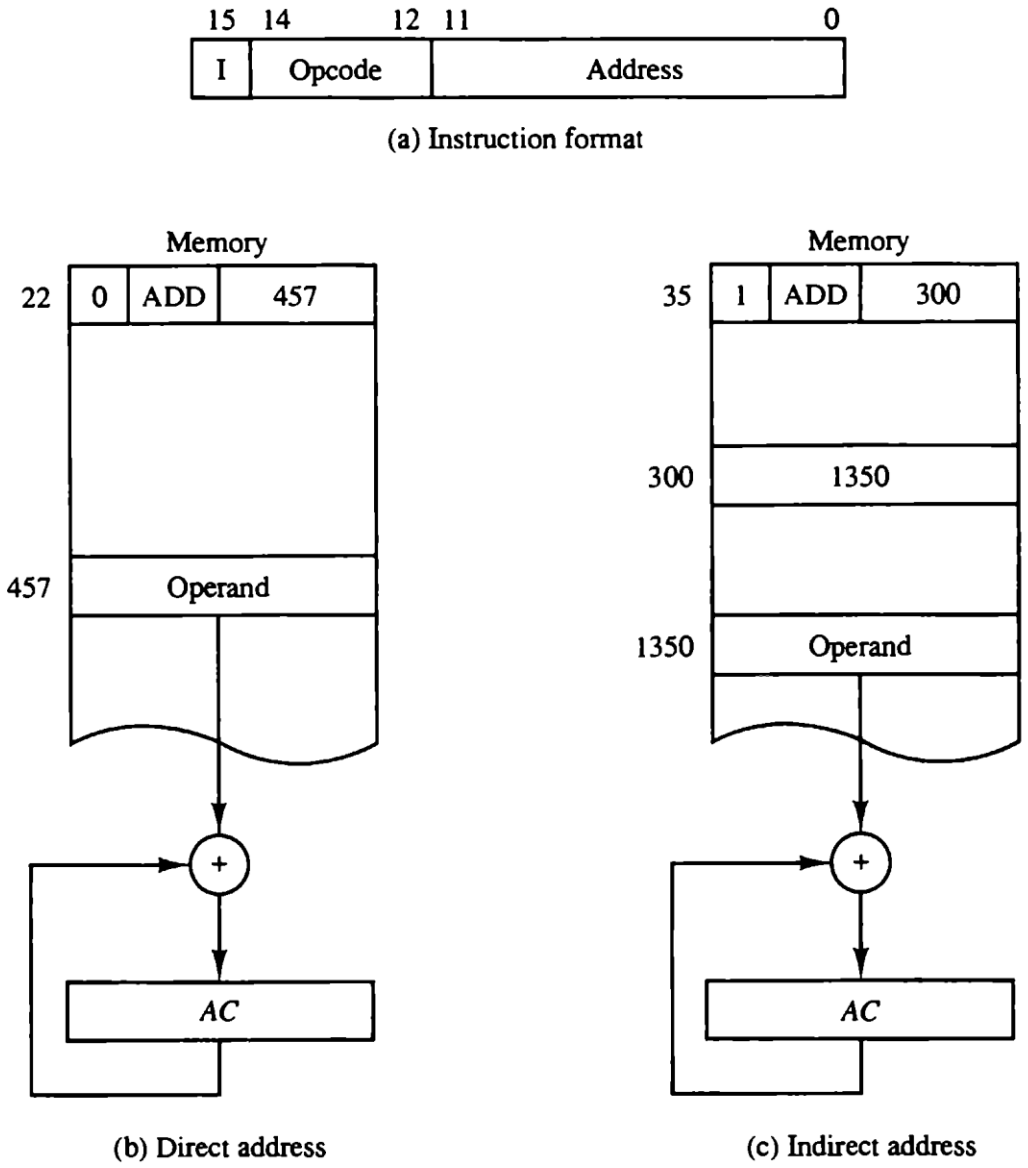


Figure 5-2 Demonstration of direct and indirect address

5.2 COMPUTER REGISTERS

Computer instructions are normally stored in consecutive memory locations and are executed sequentially one at a time. The control reads an instruction from a specific address in memory and executes it. It then continues by reading the next instruction in sequence and executes it, and so on. This type of instruction sequencing needs a counter to calculate the address of the next instruction after execution of the current instruction is completed. It is also necessary to provide a register in the control unit for storing the instruction code after it is read from memory. The computer needs processor registers for

manipulating data and a register for holding a memory address. These requirements dictate the register configuration shown in Figure 5-3. The registers are also listed in Table 5-1 together with a brief description of their function and the number of bits that they contain.

Table 5-1 List of Registers for the Basic Computer

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

The memory unit has a capacity of 4096 words and each word contains 16 bits. Twelve bits of an instruction word are needed to specify the address of an operand. This leaves three bits for the operation part of the instruction and a bit to specify a direct or indirect address. The data register (DR) holds the operand read from memory. 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 temporary data during the processing.

The memory address register (AR) has 12 bits since this is the width of a memory address. The program Counter (PC) also has 12 bits and it holds the address of the next instruction to be read from memory after the current instruction is executed. The PC goes through a counting sequence and causes the computer to read sequential instructions previously stored in memory. Instruction words are read and executed in sequence unless branch instruction is encountered. A branch instruction calls for a transfer to a nonconsecutive instruction in the program. The address part of a branch instruction is transferred to PC to become the address of the next instruction. To read an instruction, the content of PC is taken as the address for memory and a memory read cycle is initiated. PC is then incremented by one, so it holds the address of the next instruction in sequence.

Two registers are used for input and output. The input register (INPR) receives an 8 bit character from an input device. The output register (OUTR) holds an 8 bit character for an output device.

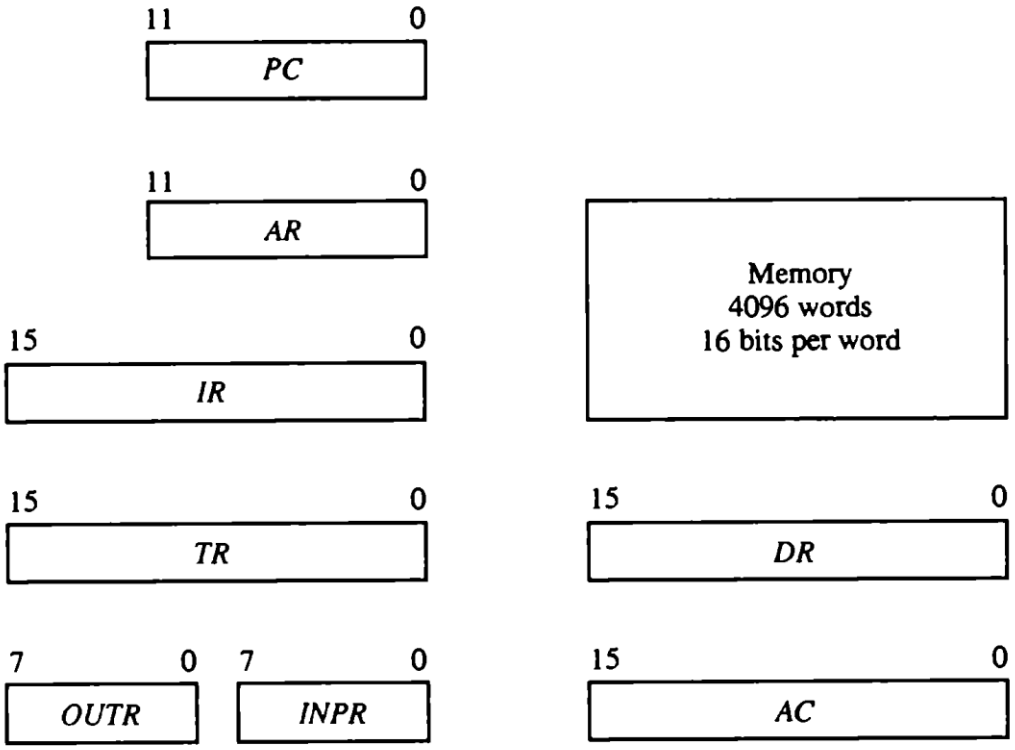


Figure 5-3 Basic computer register and memory

5.2.1 COMMON BUS SYSTEM

The basic computer has eight registers, a memory unit, and a control unit. 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. It is known that how to construct a bus system using multiplexers or three-state buffer gates. The connection of the registers and memory of the basic computer to a common bus system is shown in Figure 5-4.

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 S_2, S1 and S_0. 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

$S_2S_1S_0$ = 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 input 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 $S_2S_1S_0$ = 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 0's. When AR or PC receives 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 (LSB) 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. OUTR receives a character from AC and delivers it to an output device. There is no transfer from OUTR to any of the other registers.

The 16 lines of the common bus receive information from sex registers and the memory unit. The bus lines are connected to the inputs of six registers and the memory. Five registers have three control inputs: LD (load), INR (increment) and CLR (clear). This type of register is equivalent to a binary counter with parallel load and synchronous clear. The increment operation is achieved by enabling the count input of the counter. Two registers have only a LD input.

The input data and output data of the memory are connected to the common bus, but the memory address is connected to AR. Therefore, AR must always be used to specify a memory address. By using a single register for the address, we eliminate the need for an address bus that would have been needed otherwise. The content of any register can be specified for the memory data input during a write operation. Similarly, any register can receive the data from memory after a read operation except 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 micro-operations 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 micro-operations, 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.

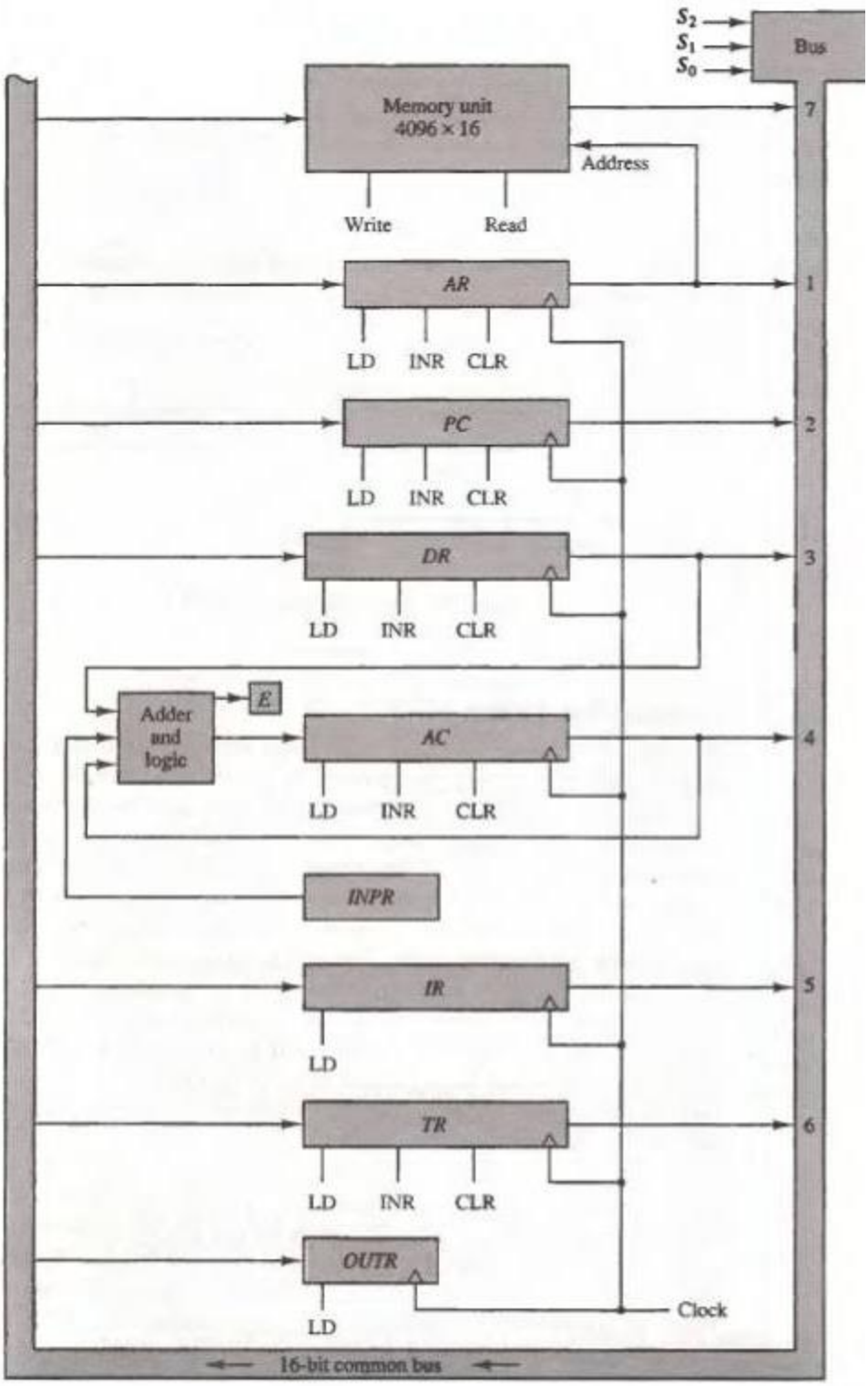


Figure 5-4 Basic computer registers connected to a common bus

Note that the content of any register can be applied onto the bus and an operation can be performed in the adder and logic circuit during the same clock cycle. The clock transition at the end of the cycle transfers the content of the bus into the designated destination register and the output of the adder and logic circuit into AC. For example, the two micro-operations

$$DR \leftarrow AC \text{ and } AC \leftarrow DR$$

can be executed at the same time. This can be done by placing the content of AC on the bus (with $S_2S_1S_0 = 100$), enabling the LD(load) input of DR, transferring the content of DR through the adder and logic circuit into AC, and enabling the LD (load) input of AC, all during the same clock cycle. The two transfers occur upon the arrival of the clock pulse transition at the end of the clock cycle.

5.3 COMPUTER INSTRUCTIONS

The basic computer has three instruction code formats, as shown in Figure 5-5. Each format has 16 bits. The operation code part of the instruction contains three bits and the meaning of the remaining 13 bits depends on the operation code encountered. A memory reference instruction uses 12 bits to specify an address and one bit to specify the addressing mode I. I is equal to 0 for direct address and to 1 for indirect address. The register-reference instructions are recognized by the operation code 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 register. An operand from memory is not needed; therefore, the other 12 bits are used to specify the operation or test to be executed. Similarly, 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 input-output operation or test performed.
The type of instruction is recognized by the computer control from the four bits in positions 12 through 15 of the instruction. If the three op code bits in positions 12 through 14 are not equal to 111, the instruction is a memory-reference type and the bit in position 15 is taken as the addressing mode I. If the 3-bit opcode is equal to 111, control then inspects the bit in position 15. If this bit is 0, the instruction is a register-reference type. If the bit is 1, the instruction is an input-output type. Note that the bit in position 15 of the

instruction code is designated by the symbol I but is not used as a mode bit when the operation code is equal to 111.

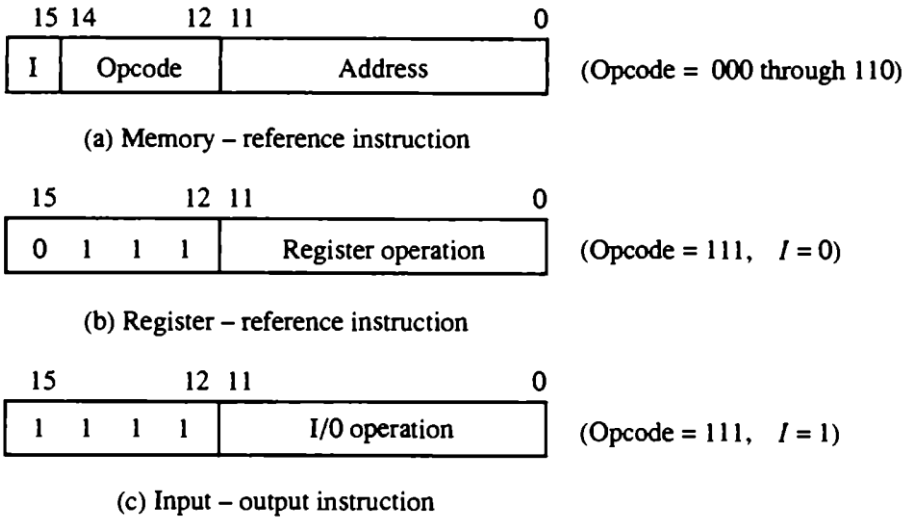


Figure 5-5 Basic computer instruction formats

Only three bits of the instruction are used for the operation code. It may seem that the computer is restricted to a maximum of eight distinct operations. However, since register reference and input-output instructions use the remaining 12 bits as part of the operation code, the total number of instruction chosen for the basic computer is equal to 25.

The instructions for the computer are listed in Table 5-2. The symbol designation is a three letter word and represents an abbreviation intended for programmers and users. The hexadecimal code is equal to the equivalent hexadecimal number of the binary code used for the instruction. By using the hexadecimal equivalent we reduced the 16 bits of an instruction code to four digits with each hexadecimal digit being equivalent to four bits. A memory-reference instruction has an address part of 12 bits. The address part is denoted by three x's and stand for the three hexadecimal digits corresponding to the 12-bit address. The last bit of the instruction is designated by the symbol I. When I = 0, the last four bits of an instruction have a hexadecimal digit equivalent from 0 to 6 since the last bit is 0. When I = 1, the hexadecimal digit equivalent of the last four bits of the instruction ranges from 8 to E since the last bit is 1.

Register-reference instructions use 16 bits to specify an operation. The leftmost four bits are always 0111, which is equivalent to hexadecimal 7. The other three hexadecimal digits give the binary equivalent of the remaining 12 bits. The input-output instructions

also use all 16 bits to specify an operation. The last four bits are always 1111, equivalent to hexadecimal F.

Table 5-2 Basic Computer Instructions

Symbol	Hexadecimal code $I = 0$	Hexadecimal code $I = 1$	Description
AND	0xxx	8xxx	AND memory word to AC
ADD	1xxx	9xxx	Add memory word to AC
LDA	2xxx	Axxx	Load memory word to AC
STA	3xxx	Bxxx	Store content of AC in memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and save return address
ISZ	6xxx	Exxx	Increment and skip if zero
CLA	7800		Clear AC
CLE	7400		Clear E
CMA	7200		Complement AC
CME	7100		Complement E
CIR	7080		Circulate right AC and E
CIL	7040		Circulate left AC and E
INC	7020		Increment AC
SPA	7010		Skip next instruction if AC positive
SNA	7008		Skip next instruction if AC negative
SZA	7004		Skip next instruction if AC zero
SZE	7002		Skip next instruction if E is 0
HLT	7001		Halt computer
INP	F800		Input character to AC
OUT	F400		Output character from AC
SKI	F200		Skip on input flag
SKO	F100		Skip on output flag
ION	F080		Interrupt on
IOF	F040		Interrupt off

5.3.1 INSTRUCTION SET COMPLETENESS

Before investigating the operations performed by the instructions, let us discuss the type of instructions that must be included in a computer. A computer should have a set of instructions so that the user can construct machine language programs to evaluate any function that is known to be computable. The set of instructions are said to be complete if the computer includes a sufficient number of instructions in each of the following categories:

1. Arithmetic, logical, and shift instructions
2. Instructions for moving information to and from memory and processor registers
3. Program control instructions together with instructions that check statues conditions
4. Input and output instructions

Arithmetic, logical, and shift instructions provide computational capabilities for processing the type of data that the user may wish to employ. The bulk of the binary information in a digital computer is stored in memory, but all computations are done in processor registers. Therefore, the user must have the capability of moving information between these two units. Decision-making capabilities are an important aspect of digital computers. For example, two numbers can be compared, and if the first is greater than the second, it may be necessary to proceed differently than if the second is greater than the first. Program control instructions such as branch instructions are used to change the sequence in which the program is executed. Input and output instructions are needed for communication between the computer are the user. Programs and data must be transferred into memory and results of computations must be transferred back to the user.

The instructions listed in Table 5-2 constitute a minimum set that provides all the capabilities mentioned above. There is one arithmetic instruction, ADD, and two related instructions, complement AC (CMA) and increment AC (INC). With these three instructions we can add and subtract binary numbers when negative numbers are in signed-2's complement representation. The circulate instructions, CIR ad CIL, can be used for arithmetic shifts as well as any other type of shifts desired. Multiplication and division can be performed using addition, subtraction, and shifting. There are three logic operations: AND, complement AC (CMA), and clear AC (CLA). The AND and complement provide a NAND operation. It can be shown that with the NAND operation it is possible to implement all the other logic operations with two variables. Moving information from memory to AC is accomplished with the load AC (LDA) instruction. Storing information from AC into memory is done with the store AC (STA) instruction. The branch instructions BUN, BSA, and ISZ, together with the four skip instructions, provide capabilities for program control and checking of status conditions. The input (INP) and output (OUT) instructions cause information to be transferred between the computer and external devices.

Although the set of instructions for the basic computer is complete, it is not efficient because frequently used operations are not performed rapidly. An efficient set of

instructions will include such instructions as subtract, multiply, OR, and exclusive-OR. These operations must be programmed in the basic computer.

5.4 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: hardwired control and 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. In the microprogrammed control, any required changes or modifications can be done by updating the microprogram in control memory. A hardwired control for the basic computer is presented in this section.

The block diagram of the control unit is shown in Figure 5-6. It consists of two decoders, a sequence counter, and a number of control logic gates. An instruction read from memory is placed in the instruction register (IR). The position of this register in the common bus system is indicated in Figure 5-4. The instruction register is shown again in Figure 5-6, where it is divided into three parts: the I bit, the operation code, and bits 0 through 11. The operation code in bits 12 through 14 are decoded with a 3 x 8 decoder. The eight outputs of the decoder are designated by the symbols D_0 through D_7. The subscripted decimal number is equivalent to the binary value of the corresponding operation code. Bit 15 of the instruction is transferred to a flip-flop designated by the symbol I. Bits 0 through 11 are applied to the control logic gates. The 4-bit sequence counter can count in binary from 0 through 15. The outputs of the counter are decoded

into 16 timing signals T_0 through T_{15}. The internal logic of the control gates will be derived later when we consider the design of the computer in detail.

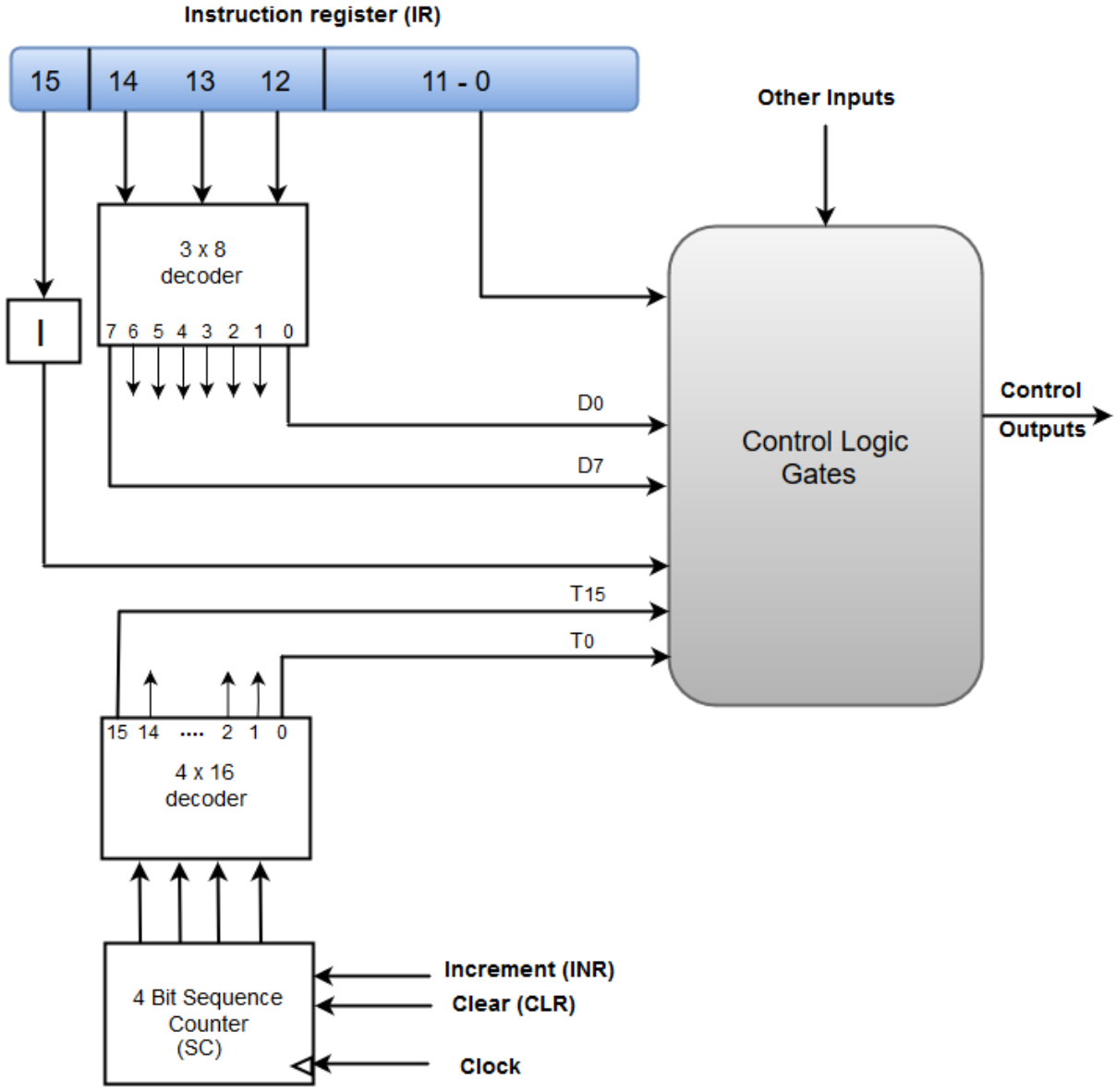


Figure 5-6 Control unit of basic computer

The sequence counter SC can be incremented or cleared synchronously. Most of the time, the counter is incremented to provide the sequence of timing signals out of the 4 x 16 decoder. Once in a while, the counter is cleared to 0, causing the next active timing signal to be T_0.

As an example, consider the case where SC is incremented to provide timing signals T_0, T_1, T_2, T_3, and T_4 in sequence. At time T_4, SC is cleared to 0 if decoder output D_3 is active. This is expressed symbolically by the statement

$$D_3T_4: \quad SC \leftarrow 0$$

The timing diagram of Figure 5-7 shows the time relationship of the control signals. The sequence counter SC responds to the positive transition of the clock. Initially, the CLR input of SC is active. The first positive transition of the clock clears SC to 0, which in turn activates the timing signal T_0 out of the decoder. T_0 is active during one clock cycle. The positive clock transition labeled T_0 in the diagram will trigger only those registers whose control inputs are transition, to timing signal T_0. SC is incremented with every positive clock transition, unless its CLR input is active. This produces the sequence of timing signals T_0, T_1, T_2, T_3, T_4 and so on, as shown in the diagram. (Note the relationship between the timing signal and its corresponding positive clock transition). If SC is not cleared, the timing signals will continue with T_5, T_6, up to T_{15} and back to T_0.
The last three waveforms in Figure 5-7 show how SC is cleared when $D_3T_4 = 1$. Output D_3 from the operation decoder becomes active at the end of timing signal T_2. When timing signal T_4 becomes active, the output of the AND gate that implements the control function D_3T_4 becomes active. This signal is applied to the CLR input of SC. On the next positive clock transition (the one marked T_4 in the diagram) the counter is cleared to 0. This causes the timing signal T_0 to become active instead of T_5 that would have been active if SC were incremented instead of cleared.

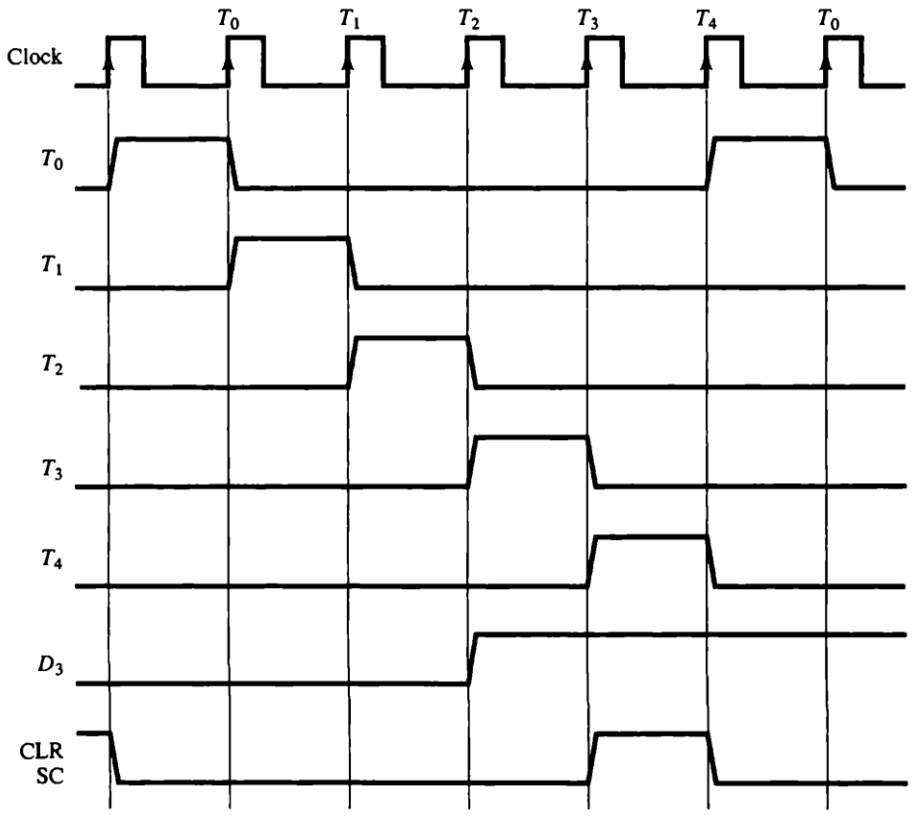


Figure 5-7 Example of control timing signals

5.5 INSTRUCTION CYCLE

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

1. Fetch instruction from memory
2. Decode the instruction
3. Read the effective address from memory if the instruction has an indirect address.
4. Execute the Instruction

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

5.5.1 FETCH AND DECODE

Initially the program counter (PC) is loaded with the address of first instruction in the program. The sequence counter SC is cleared to 0, providing a decoded timing signal T_0. After each clock pulse SC is incremented by one, so that the timing signals go through a sequence T_0, T_1, T_2 and so on.

Following register transfer statement specifies the micro operation for the fetch and decode phase.

T_0: AR←PC

T_1: IR←MW [AR], PC←PC+1

T_2: D_0,…., D_7 ← 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 address from PC to AR during clock transition associated with timing signal T_0. The instruction read from memory is then placed in the instruction register IR with clock transition associated with timing signal T_1. At the same time PC is incremented by one to prepare it for the address of the next instruction in the

program. At the time T_2, the operation code in IR is decoded, the direct bit is transferred to the flip-flop, and the address part of the instruction is transferred to AR.

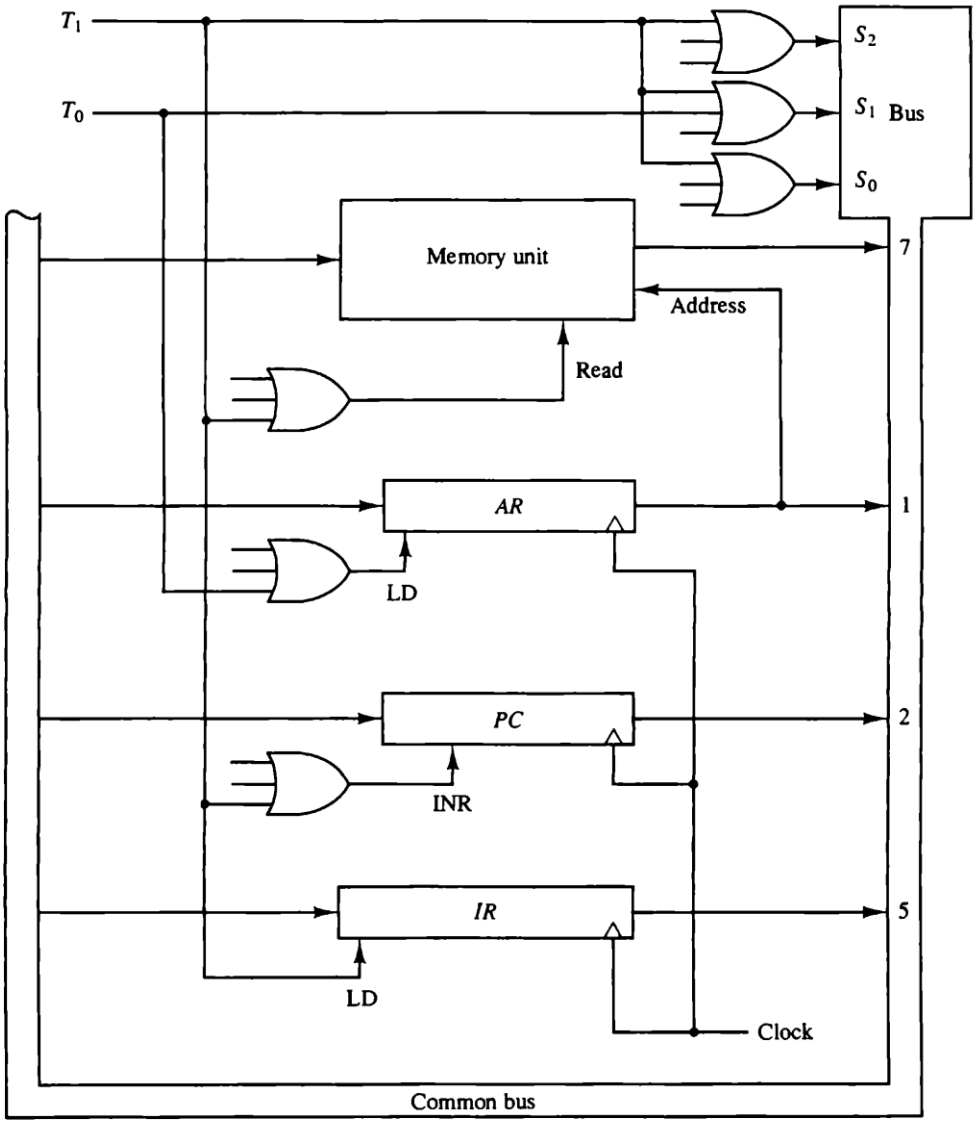


Figure 5-8 Register transfers for the fetch phase

Figure 5.8 shows how the first two register transfer statements are implemented in Bus system. To provide the data path for the transfer of PC to AR we must supply timing signal T_0 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 bus to AR by enabling the LD input of AR

The next clock transition initiates the transfer from PC to AR since T_0=1. In order to implement the second statement

$$T_1: \quad IR \leftarrow M[AR], PC \leftarrow PC+1$$

It is necessary to use timing signal T_1 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 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 T_1=1.

Figure 5.8 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.

5.5.2 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. Following flowchart in figure 5.8 presents the initial configuration for the instruction cycle and shows how control determines the instruction type after the decoding.

Decoder output D_7 is equal to 1 if the operation code is equal to binary 111. 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 instruction which is now available in flip flop I. if $D_7 = 0$ 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 micro-operation for the indirect address condition can be symbolized by the register transfer statement.

$$AR \leftarrow MW\ [AR]$$

Initially, AR holds the address part of the instruction. Thus 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.

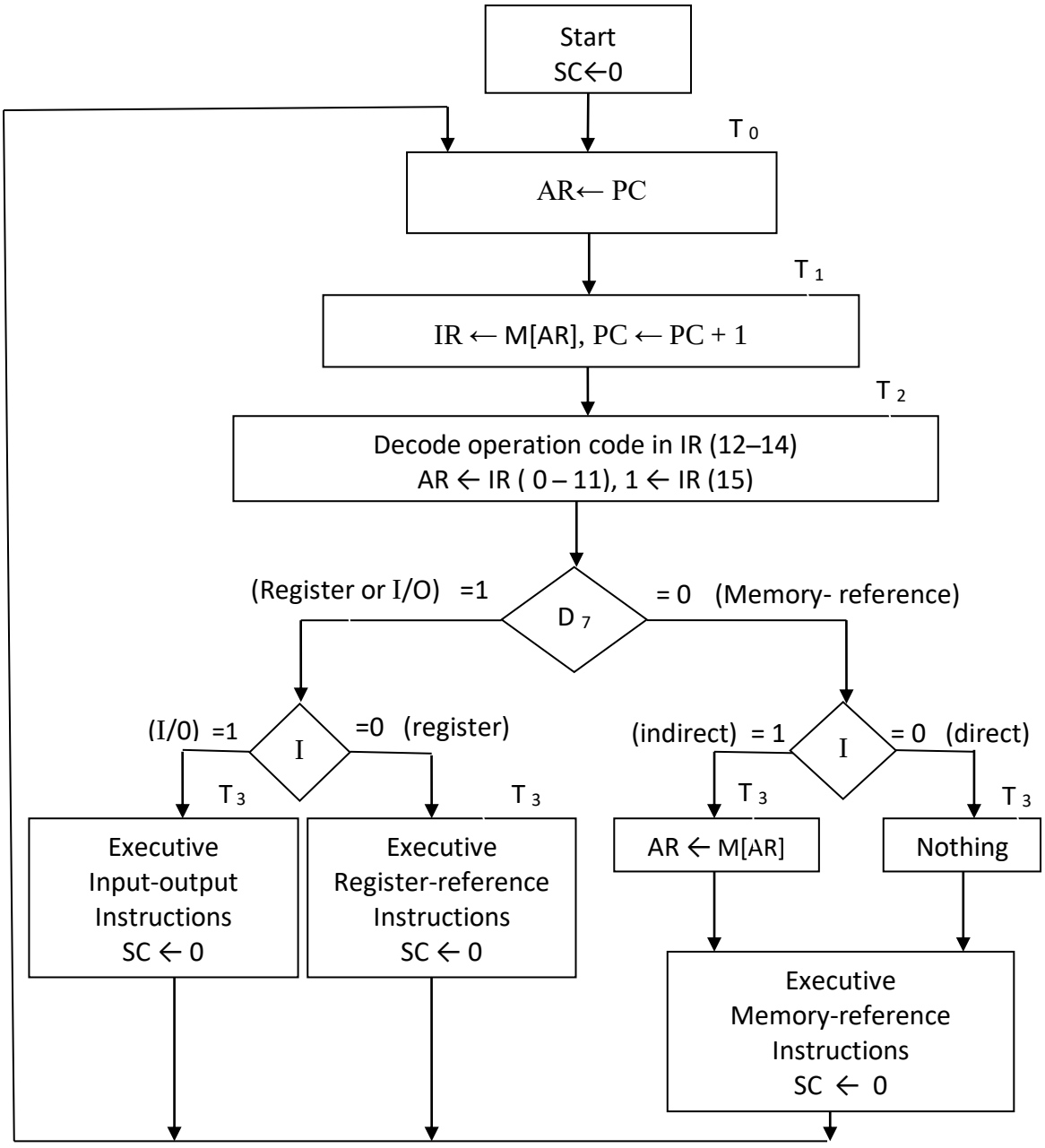


Figure 5-9 Flowchart for instruction cycle (initial configuration)

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

$D'_7 I T_3$	:	AR ← MW [AR]
$D'_7 I' T'_3$	:	Nothing
$D_7 I' T'_3$	:	Execute a register-reference instruction
$D_7 I T'_3$	:	Execute an input-output instruction

When a memory reference register 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 return to fetch phase with timing $T_0 = 1$.

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.

5.6 MEMORY REFERENCE INSTRUCTIONS

In order to specify the microoperations needed for the execution of each instruction, it is necessary that the function that they are intended to perform be defined precisely. Looking back to Table 5.4, where the instructions are listed, we find that some instructions have an ambiguous description. This is because the explanation of an instruction in words is usually lengthy, and not enough space is available in the table for such a lengthy explanation. We will now show that the function of the memory-reference instructions can be defined precisely by means of register transfer notation.

Table 5.3 lists the seven memory-reference instructions. The decoded output D_i for i = 0, 1, 2, 3, 4, 5, and 6 from the operation decoder that belongs to each instruction is included in the table. The effective address of the instruction is in the address register AR and was placed there during timing signal T_2 when I = 0, or during timing signal T_3 when I = 1. The execution of the memory-reference instructions starts with timing signal T4 the symbolic description of each instruction is specified in the table in terms of register transfer notation. The actual execution of the instruction in the bus system will require a sequence of microoperations. This is because data stored in memory cannot be processed directly. The data must be read from memory to a register where they can be operated on with logic circuits. We now explain the operation of each instruction and list the control

functions and microoperations needed for their execution. A flowchart that summarizes all the microoperations is presented at the end of this section.

AND to AC

This is an instruction that performs 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 microoperations that execute this instruction are :

D_0T_4 : DR ← M[AR]

D_0T_5: 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 is 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.

Table 5-5 Memory Reference Instructions

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 $PC \leftarrow PC + 1$

ADD to AC

The 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 Cout is transferred to the E (extended accumulator) flip-flop. The microoperations needed to execute this instruction are

D_1T_4 : DR ← M[AR]

D_1T_5 : AC ← AC + DR, E ← Cout, SC ← 0

The same two timing signals, T4 and T5, are used again but with operation decoder D_1 instead of D_0, 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, SC ← 0

From the bus diagram 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:

D_3T_4 : 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:

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

The effective address from AR is transferred through the common bus to PC. Resetting SC to 0 transfers control to T_0. 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 subordinate. This operation was specified in Table 5-5 with the following register transfer:

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

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

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

The result of this operation is shown in part (b) of the figure. The return address 21 is stored in memory location 135 and control continues with the subordinate 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.

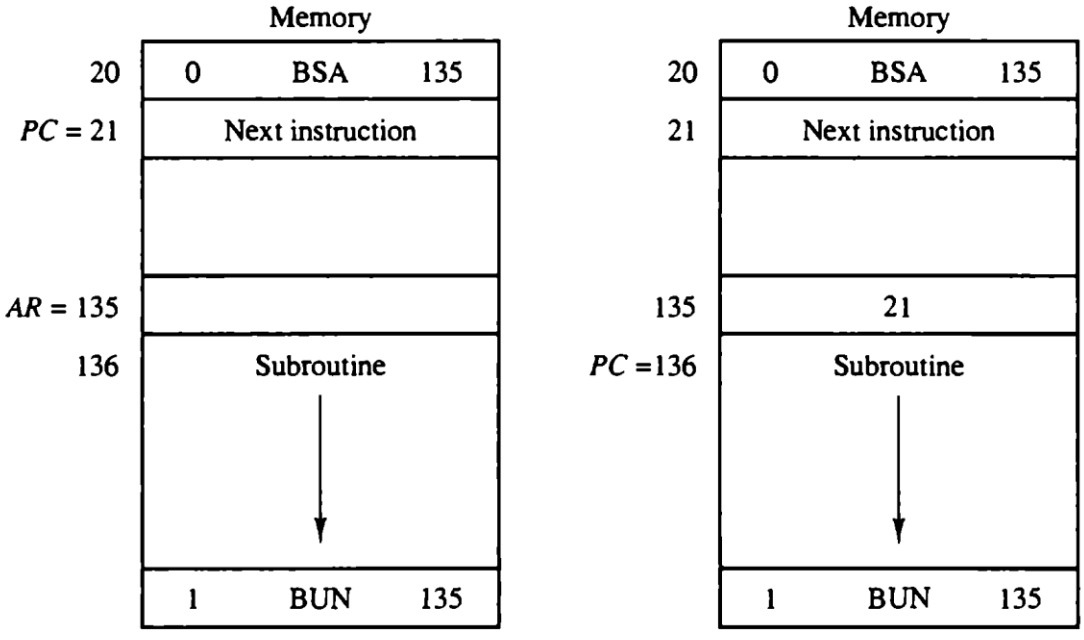
When the BUN instruction is executed, the effective address 21 is transferred to PC. The next instruction cycle finds PC with the value 21, so control continues to execute the instruction at the return address.

The BSA instruction performs the function usually referred to as a subroutine call. The indirect BUN instruction at the end of the subroutine performs the function referred to as a subroutine return. In most commercial computers, the return address associated with a subroutine is stored in either a processor register or in a portion of memory called a stack. It is not possible to perform the operation of the BSA instruction in one clock cycle when we use the bus system of the basic computer. To use the memory and the bus properly, the BSA instruction must be executed with a sequence of two microoperations:

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

Timing signal T_4 initiates a memory write operation, places the content of PC onto the bus, and enables the INR input of AR. The memory write operation is completed and AR is incremented by the time the next clock transition occurs. The bus is used at T_5 to transfer the content of AR to PC.

Figure 5-10 Example of BSA instruction execution.



(a) Memory PC, and AR at time T4 **(b) Memory and PC after execution**

It is not possible to perform the operation of the BSA instruction in one clock cycle when we use the bus system of the basic computer. To use the memory and the bus properly, the BSA instruction must be executed with a sequence of two microoperations:

D_5T_4: M[AR] ← PC, AR ← AR + 1
D_5T_5: PC ← AR, SC ← 0

Timing signal T4 initiates a memory write operation, places the content of PC onto the bus, and enables the INR input of AR. The memory write operation is completed and AR is incremented by the time the next clock transition occurs. The bus is used at T_5 to transfer the content of AR to PC.

ISZ : Increment and Skip if Zero

This instruction increments the word specified by the effective address, and if the incremented value is equal to 0, PC is incremented by 1. The programmer usually stores a negative number (in 2's complement) in the memory word. As this negative number is repeatedly incremented by one, it eventually reaches the value of zero. At that time PC is incremented by one in order to skip the next instruction in the program.

Since it is not possible to increment a word inside the memory, it is necessary to read the word into DR, increment DR, and store the word back into memory. This is done with the following sequence of microoperations:

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

5.6.1 CONTROL FLOWCHART

A flowchart showing all microoperations for the execution of the seven memory-reference instructions is shown in Fig. 5.11. The control functions are indicated on top of each box. The microoperations that are performed during time T_4, T_5, or T_6 depend on the operation code value.

This is indicated in the flowchart by six different paths, one of which the control takes after the instruction is decoded. The sequence counter SC is cleared to 0 with the last timing signal in each case. This causes a transfer of control to timing signal T0 to start the next instruction cycle. Note that we need only seven timing signals to execute the longest

instruction (ISZ). The computer can be designed with a 3-bit sequence counter. The reason for using a 4-bit counter for SC is to provide additional timing signals for other instructions that are presented in the problems section.

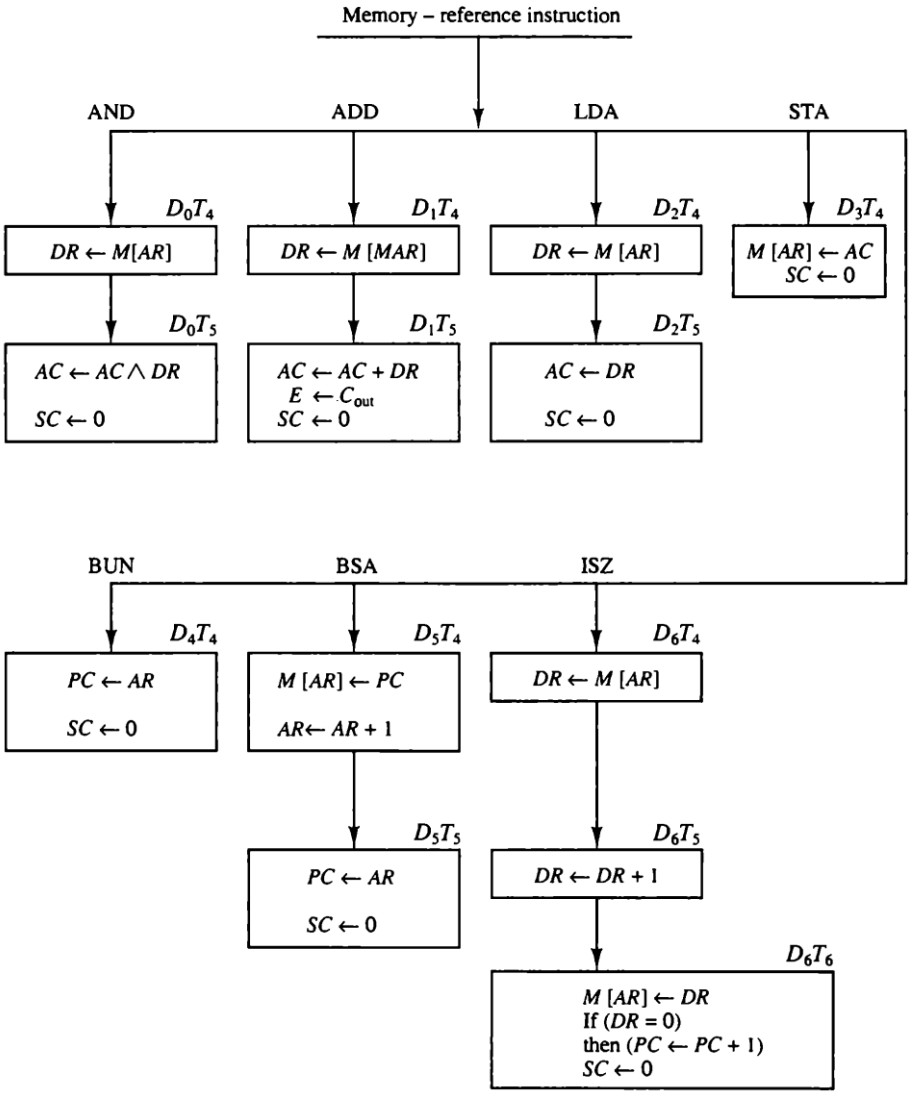


Figure 5-11 Flowchart for memory-reference instructions

5.7 REGISTER REFERENCE INSTRUCTIONS

Register-reference instructions are recognized by the control when $D_7 = 1$ 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 T_2.

The control functions and microoperations for the register-reference instructions are listed in Table 5-6. These instructions are executed with the clock transition associated with

timing variable T_3. Each control function needs the Boolean relation D_7 I' T_3, 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 Bi to bit i of IR, all control functions can be simply denoted by rB_i. For example, the instruction CLA has the hexadecimal code 7800, which gives the binary equivalent 0111 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 D_7. Bit 11 in IR is 1 and is recognized from B_{11}. The control function that initiates the microoperation for this instruction is D_7 I' T_3 B_{11} = rB_{11}. The execution of a register-reference instruction is completed at time T_3. The sequence counter SC is cleared to 0 and the control goes back to fetch the next instruction with timing signal T_0.

The first seven register-reference instructions perform clear, complement, circular shift, and increment microoperations on the AC or E registers. The next four instructions cause a skip of the next instruction in sequence when a stated condition is satisfied. The skipping of the instruction is achieved by incrementing PC once again (in addition, it is being incremented during the fetch phase at time T_1). The condition control statements must be recognized as part o the control conditions. The AC is positive when the sign bit in AC(15) = 0; it is negative when AC(15) = 1. The content of AC is zero (AC = 0) if all the flip-flops of the register are zero. The HLT instruction clears a start-stop flip-flops S and stops the sequence counter from counting. To restore the operation of the computer, the start-stop flip-flop must be set manually.

Table 5-6 Execution of Register Reference Instruction

$D_7I'T_3 = r$ (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

5.8 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.

5.8.1 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 5.12. 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.

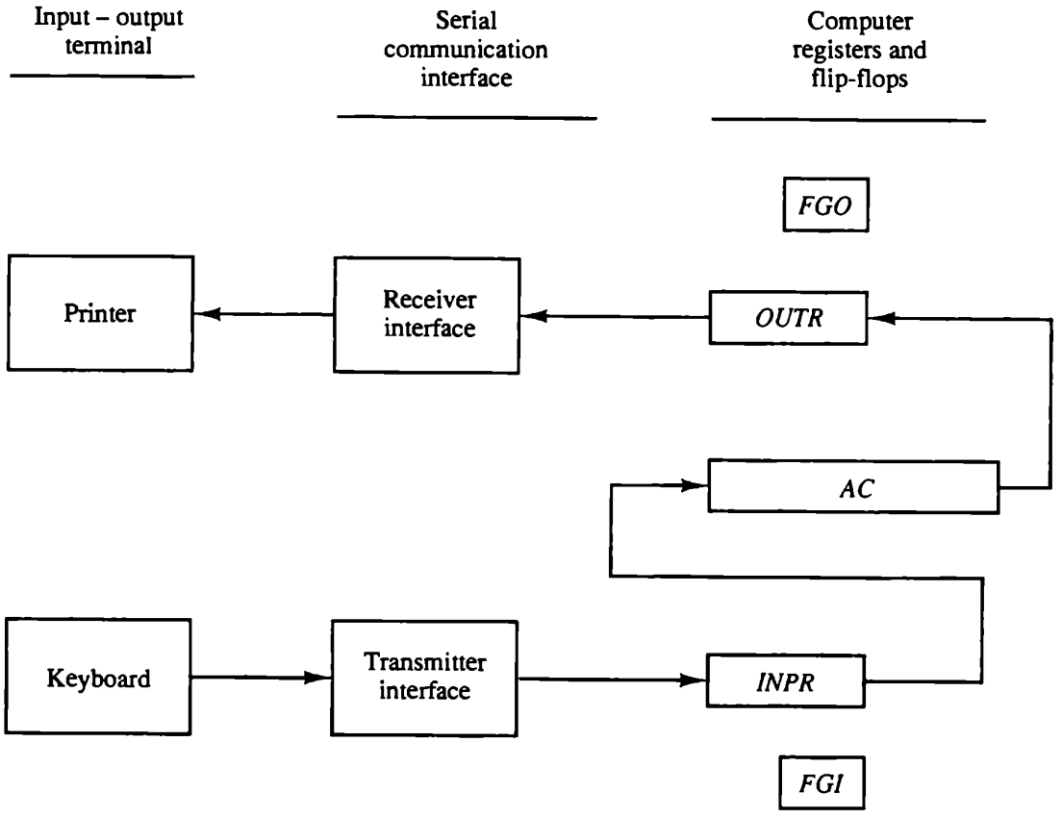


Figure 5-12 Input Output Configuration

The input register INPR consists of eight bits and holds 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.

The output register OUTR works similarly but the direction of information flow is reversed. Initially, the output flag FGO is set to 1. The computer checks the flag bit, if it is 1, the information from AC is transferred in parallel to OUTR and FGO is cleared to 0. The output device accepts the coded information, prints the corresponding character, and when the operation is completed, it sets FGO to the computer does not load a new character into OUTR when FGO is 0 because this condition indicates that the output device is in the process of printing the character.

5.8.2 INPUT OUTPUT INSTRUCTIONS

Input and output instructions are needed for transferring information to and from AC register, for checking the flag bits, and for controlling the interrupt facility. Input-output instructions have an operation code 1111 and are recognized by the control when $D_7 = 1$ and I = 1. The remaining bits of the instruction specify the particular operation. The control functions and microoperations for the input-output instructions are listed in Table 5.7. These instructions are executed with the clock transition associated with timing signal T_3. Each control function needs a Boolean relation $D\text{-}IT_3$, which we designate for convenience by the symbol p. The control function is distinguished by one of the bits in IR. By assigning the symbol Bi to bit i of IR, all control functions can be denoted by pB_i for i = 6 though 11. The sequence counter SC is cleared to 0 when $p = D_7IT_3 = 1$.

The INP instruction transfers the input information from INPR into the eight low-order bits of AC and also clears the input flag to 0. The OUT instruction transfers the eight least significant bits of AC into the output register OUTR and clears the output flag to 0. The next two instructions in Table 5.7 check the status of the flags and cause a skip of the next

instruction if the flag is 1. The instruction that is skipped will normally be a branch instruction to return and check the flag again. The branch instruction is not skipped if the flag is 1. If the flag is 1, the branch instruction is skipped and an input or output instruction is executed. The last two instructions set and clear an interrupt enable flip-flop IEN. The purpose of IEN is explained in conjunction with the interrupt operation.

Table 5-7 Input Output 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}:	$AC(0–7) \leftarrow INPR$, $FGI \leftarrow 0$	Input character
OUT	pB_{10}:	$OUTR \leftarrow AC(0–7)$, $FGO \leftarrow 0$	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

5.8.3 PROGRAM INTERRUPT

The process of communication just described is referred to as programmed control transfer. 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. To see why this is inefficient, consider a computer that can go through an instruction cycle in 1 μs. Assume that the input-output device can transfer information at a maximum rate of 10 characters per second. This is equivalent to one character every 100,000 μs. Two instructions are executed when the computer checks the flag bit and decides not to transfer the information. This means that at the maximum rate, the computer will check the flag 50,000 times between each transfer. The computer is wasting time while checking the flag instead of doing some other useful processing task. 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. However, when a flag is set, the computer is momentarily interrupted from proceeding with the current program and is informed of the fact that a flag has been

set. The computer is wasting time while checking the flag instead of doing some other useful processing task.

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. However, when a flag is set, the computer is momentarily interrupted from proceeding with the current program and is informed of the fact that a flag has been set. The computer deviates momentarily from what it is doing to take care 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 1 (with the ION instruction), the computer can be interrupted. These two instructions provide the programmer with the capability of making a decision as to whether or not to use the interrupt facility.

The way that the interrupt is handled by the computer can be explained by means of the flowchart of Figure 5-13. 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 IEN = 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.

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 where it can be found later when the program returns to the instruction at which it was interrupted. This location may be a processor register, a memory stack, or a specific memory location. Here we choose the memory location at address 0 as the place for storing the return address. Control then inserts address 1 into PC and clears IEN and R so that no more interruptions can occur until the interrupt request from the flag has been serviced.

An example that shows what happens during the interrupt cycle is shown in Figure 5-14. Suppose that an interrupt occurs and R is set to 1 while the control is executing the

instruction at address 255. At this time, the return 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 5-14 (a).

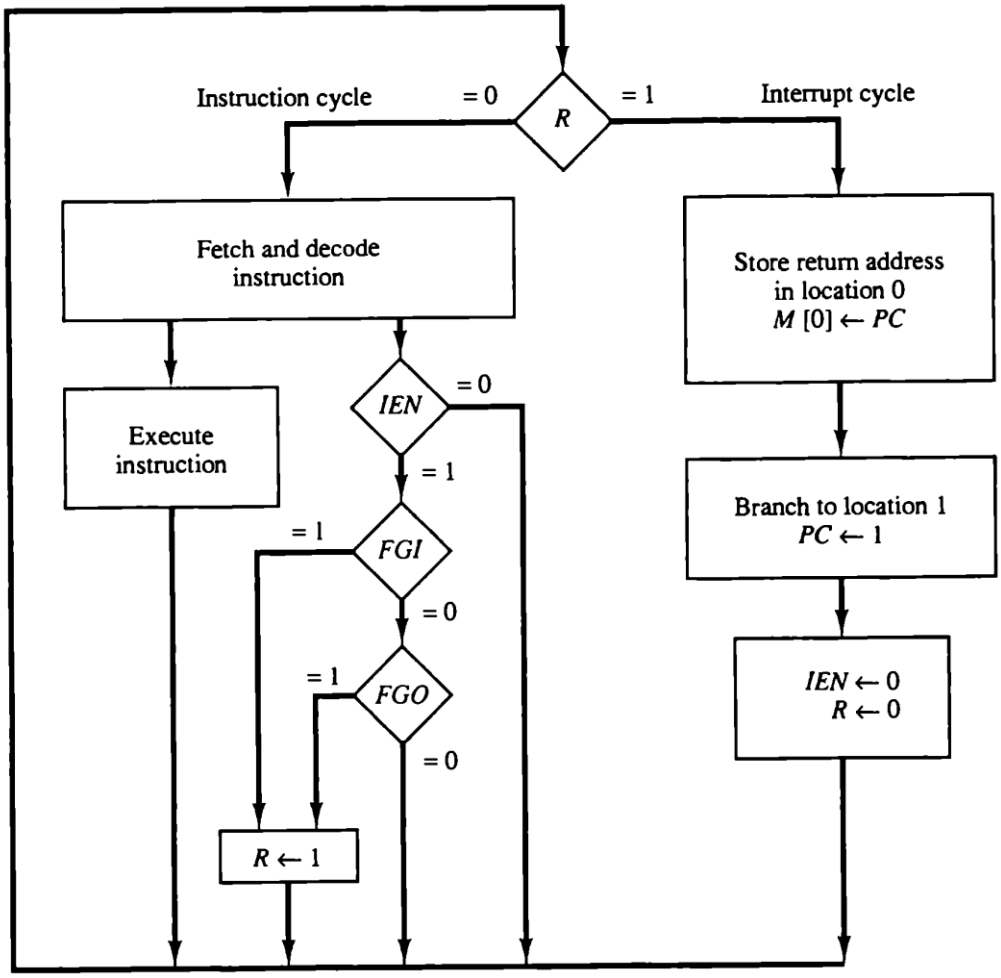


Figure 5-13 Flowchart of Interrupt Cycle

When control reaches timing signal T0 and 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. At the beginning of the next instruction cycle, the instruction that is read from memory is in address 1 since this is the content of PC. 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. One 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 5-14 (b).

The instruction that returns the computer to the original place in the main program is a branch indirect instruction with an address part of 0. This instruction is placed at the end of the I/O service program. After this instruction is read from memory during the fetch phase, control goes to the indirect phase (because I = 1) to read the effective address. The effective address is in location 0 and is the return address that was stored there during the previous interrupt cycle. The execution of the indirect BUN instruction results in placing into PC the return address from location 0.

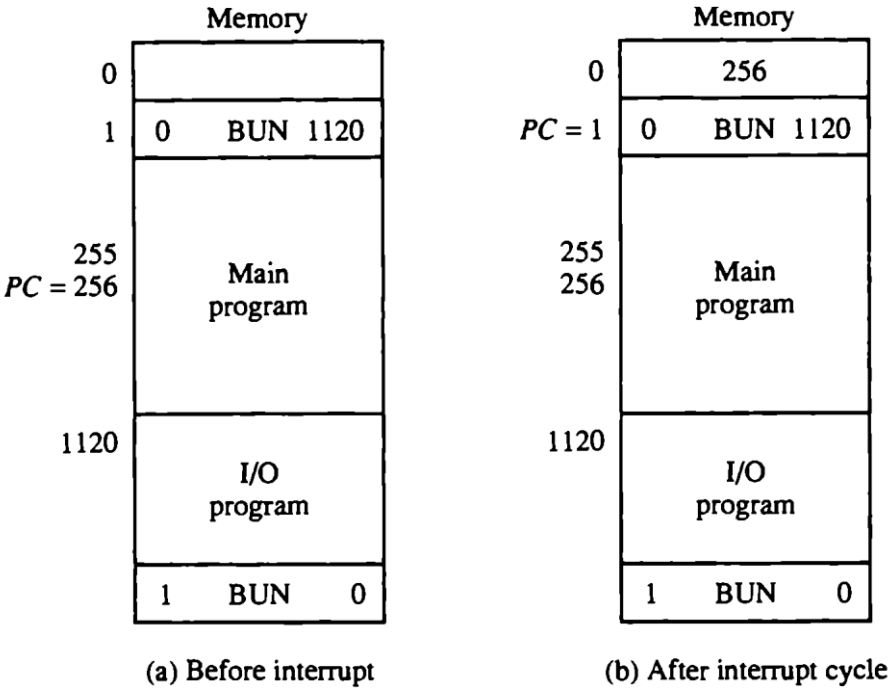


Figure 5-14 Demonstration of the interrupt cycle

5.8.4 INTERRUPT CYCLE

We are now ready to list the register transfer statements for the interrupt cycle. The interrupt cycle is initiated after the last execute phase if the interrupt flip-flop R is equal to 1. This flip-flop is set to 1 if IEN = 1 and either FGI or FGO are equal to 1. This can happen with any clock transition except when timing signals T_0, T_1 or T_2 are active. The condition for setting flip-flop R to 1 can be expressed with the following register transfer statement :

$$T'_0T'_1T'_2\ (IEN)\ (FGI + FGO) :\ R \leftarrow 1$$

The symbol + between FGI and FGO in the control function designates a logic OR operation. This is ANDed with IEN and $T'_0T'_1T'_2$.

We now modify the fetch and decode phases of the instruction cycle. Instead of using only timing signals T_0, T_1, and T_2 (as shown in Figure 5.9) we will AND the three timing signals with R′ so that the fetch and decode phases will be recognized from the three control functions $R'T_0$, $R'T_1$, and $R'T_2$. The reason for this is that after the instruction is executed and SC is cleared to 0, the control will go through a fetch phase only if R = 0. Otherwise, if R = I, the control will go through an interrupt cycle. The interrupt cycle stores the return address (available in PC) into memory location 0, branches to memory location 1, and clears IEN, R, and SC to 0. This can be done with the following sequence of micro operations.

RT_0: AR ← 0, TR ← PC

RT_1: M[AR] ← TR, PC ← 0

RT_2: PC ← PC + 1, IEN ← 0, R ← 0, SC ← 0

During the first timing signal AR is cleared to 0, and the content of PC is transferred to the temporary register TR. With the second timing signal, the return address is stored in memory at location 0 and PC is cleared to 0. The third timing signal increments PC to 1, clears IEN and R, and control goes back to T_0 by clearing SC to 0. The beginning of the next instruction cycle has the condition $R'T_0$ and the content of PC is equal to 1. The control then goes through an instruction cycle that fetches and executes the BUN instruction in location 1.

5.9 CHECK YOUR PROGRESS

1. Register which is used to store values of arithmetic and logical operations is termed as _____ .
2. Counter that holds addresses of next fetched instruction is called _____ .
3. The register that includes the address of the memory unit is termed as the _____.
4. A group of bits that tell the computer to perform a specific operation is known as _____ .
5. N bits in operation code imply that there are _____ possible distinct operators.
6. MRI indicates _____.

7. _____ is represented by $D_7I'T$ in the instruction cycle.
8. A k-bit field can specify any one of _____ .
9. Interrupts which are initiated by an instructions are _____.
10. The BSA instruction is _____ .

5.10 SUMMARY

1. The organization of the computer is defined by its internal registers, the timing and control structure, and the set of instructions that it uses.
2. The user of a computer can control the process by means of a program. A program is a set of instructions that specify the operations, operations operands, and the sequence by which processing has to occur.
3. The general-purpose digital computer is capable of executing various micro operations and, in addition, can be instructed as to what specific sequence of operations it must perform.
4. An operation is part of an instruction stored in computer memory. It is a binary code tells the computer to perform a specific operation.
5. The operation part of an instruction code specifies the operation to be performed.
6. Instruction code formats are conceived computer designers who specify the architecture of the computer.
7. The simplest way to organize a computer is to have one processor register and instruction code format with two parts. The first part specifies the operation to be performed and the second specifies an address.
8. Computer instructions are normally stored in consecutive memory locations and are executed sequentially one at a time.
9. The direct and indirect addressing modes are used in the computer.
10. The memory word that holds the address of the operand in an indirect address instruction is used as a pointer to an array of data. The pointer could be placed in processor register instead of memory as done in commercial computers.
11. The basic computer has eight registers, a memory unit, and a control unit. Paths should be provided to transfer information from one register to another and between memory and registers.
12. The output of seven registers and memory are connected to the common bus.

13. The lines from the common bus are connected to the inputs of each register and the data input of each register and the data inputs of the memory.
14. The 16 lines of the common bus receive information from sex registers and the memory unit. The bus lines are connected to the inputs of six registers and the memory.
15. The content of any register can be applied onto the bus and an operation can be performed in the adder and logic circuit during the same clock cycle.
16. The input data and output data of the memory are connected to the common bus, but the memory address is connected to AR.
17. Content of any register can be applied onto the bus and an operation can be performed in the adder and logic circuit during the same clock cycle.
18. The clock transition at the end of the cycle transfers the content of the bus into the designated destination register and the output of the adder and logic circuit into AC.

5.11 KEYWORDS

1. **Instruction Code** is a group of bits that tells the computer to perform a specific operation part.
2. **Opcode** field specifies the operation to be performed.
3. **Operand** is a term used to describe any object that is capable of being manipulated.
4. **Memory Reference Instructions** refer to memory address as an operand. The other operand is always accumulator.
5. **Register Reference Instructions** perform operations on registers rather than memory addresses.
6. **Input/Output Instructions** are for communication between computer and outside environment.

5.12 SELF ASSESSMENT TEST

1. What is a computer instruction and what are the ways for specifying them?
2. What is addressing? What is direct and indirect addressing?

3. Draw and explain the common bus system.
4. What are computer registers? Draw and explain the computer registers organization.
5. A computer uses a memory unit with 256K words of 32 bits each. A binary instruction code is stored in one word of memory. The instruction has four parts: an indirect bit, an operation code, a register code part to specify one of 64 registers, and an address part.
   a. How many bits are there in the operation code, the register code part, and the address part?
   b. Draw the instruction word format and indicate the number of bits in each part.
   c. How many bits are there in the data and address inputs of the memory?
6. What is the difference between a direct and an indirect address instruction? How many references to memory are needed for each type of instruction to bring an operand into a processor register?
7. Explain why each of the following microoperations cannot be executed during a single clock pulse in the system shown in Figure 5-8. Specify a sequence of microoperations that will perform the operation.
   a. IR ← M[PC]
   b. AC ← AC + TR
   c. DR ← DR + AC (AC does not change)
8. What are the two instructions needed in the basic computer in order to set the E flip-flop to 1?
9. Draw a timing diagram similar to Figure 5-7 assuming that SC is cleared to 0 at time T_3 if control signal C_7 is active.

   $$C_7T_3: SC \leftarrow 0$$

   C_7 is activated with the positive clock transition associated with T_1.
10. The content of AC in the basic computer is hexadecimal A937 and the initial value of E is 1. Determine the contents of AC, E, PC, AR, and IR in hexadecimal after the execution of the CLA instruction. Repeat 11 more times, starting from each one of the register-reference instructions. The initial value of PC is hexadecimal 021.

5.13 ANSWER TO CHECK YOUR PROGRESS

1. Accumulator
2. Program Counter
3. MAR
4. Instruction code
5. N / 2
6. Memory Reference Instruction
7. Register Reference Instruction
8. 2^k registers.
9. external
10. Branch and save return address

5.14 REFERENCES / SUGGESTED READINGS

1. Computer Organization and Architecture, Rajaram & Radhakrishan, PHI.
2. Computer Organization & Architecture: Designing for Performance, Stalling, PHI.
3. Computer Organization and Design, Pal Choudhary, PHI.
4. Computer Systems Organization & Architecture, Carpenelli, Pearson Education.
5. Computer Organization and Architecture, Stalling, Pearson Education.
6. Computer System Architecture, Morris Mano, PHI.
7. Computer Architecture and Organization, McGraw Hill Company, New Delhi. J.P. Hayes.

SUBJECT: COMPUTER SYSTEM ARCHITECTURE	
COURSE CODE: MCA-24	**AUTHOR: DR. MANOJ DUHAN**
LESSON NO. 6	

CENTRAL PROCESSING UNIT

REVISED / UPDATED SLM BY NEERAJ VERMA

STRUCTURE

6.0 LEARNING OBJECTIVES

After reading this lesson, you should be able to:

1. Understand the internal architecture of CPU and its various functions.
2. Elaborate the General Register Organization that can perform all the arithmetic, logic, and shift microoperations in the processor.
3. Learn about Stack Organization including register stack, memory stack and reverse polish notation.
4. Learn about various addressing modes with suitable example.

6.1 INTRODUCTION

The part of the computer that performs the bulk of data-processing operations is called the central processing unit and is referred to as the CPU. The CPU is made up of three major parts, as shown in Figure 6-1. 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.

The CPU performs a variety of functions dictated by the type of instructions that are incorporated in the computer. Computer architecture is sometimes defined as the computer structure and behavior as seen by the programmer that uses machine language instructions. This includes the instruction formats, addressing modes, the instruction set, and the general organization of the CPU registers.

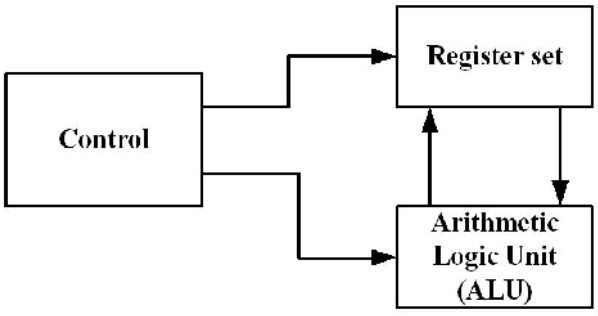


Figure 6-1 Major component of CPU

One boundary where the computer designer and the computer programmer see the same machine is the part of the CPU associated with the instruction set. From the designer's point of view, the computer instruction set provides the specifications for the design of the CPU. The design of a CPU is a task that in large part involves choosing the hardware for implementing the machine instructions. The user who programs the computer in machine or assembly language must be aware of the register set, the memory structure, instruction performs.

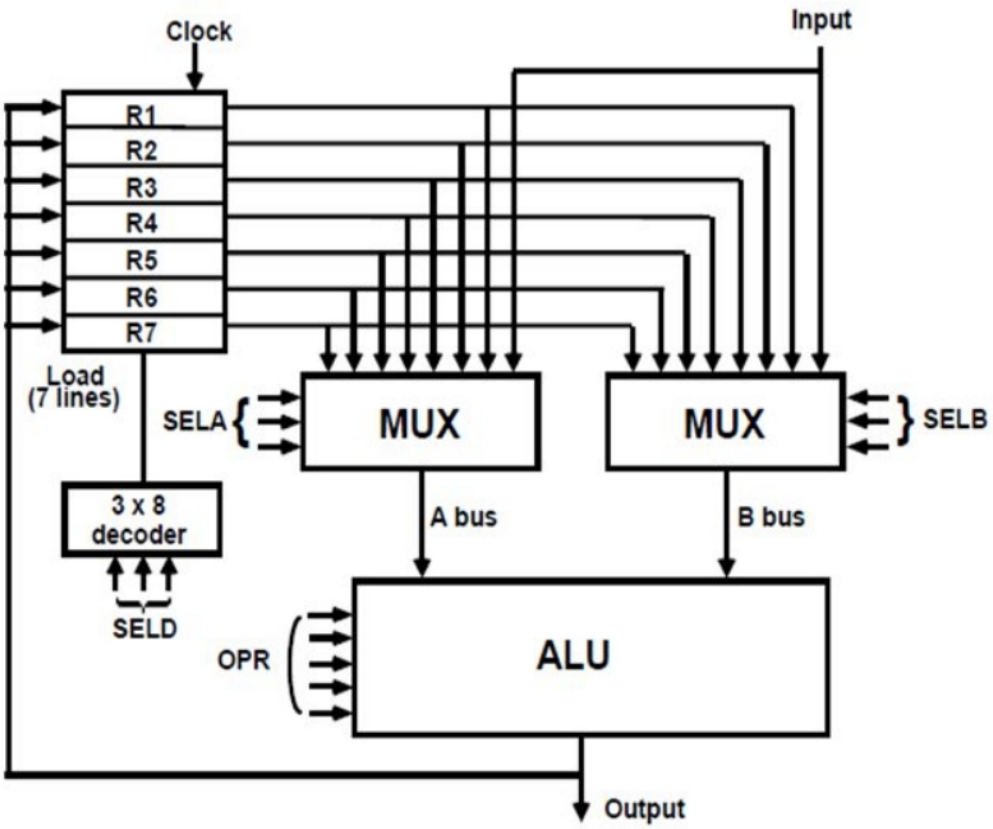
This chapter describes the organization and architecture of the CPU with an emphasis on the user's view of the computer. We briefly describe how the registers communicate with the ALU through buses and explain the operation of the memory stack. We then present the type of instruction formats available, the addressing modes used to retrieve data from memory, and typical instructions commonly incorporated in computers. The last section presents the concept of reduced instruction set computer (RISC).

6.2 GENERAL REGISTER ORGANIZATION

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

A bus organization for seven CPU registers is shown in Figure 6-2. The output of each register is connected to two multiplexers (MUX) to form the two buses A and B. The selection lines in each multiplexer select one register or the input data for the particular bus. The A and B buses form the inputs to a common arithmetic logic unit (ALU). The operation selected in the ALU determines the arithmetic or logic microoperation that is to be performed. The result of the microoperation is available for output data and also goes into the inputs of all the registers. The register that receives the information from the

output bus is selected by a decoder. The decoder activates one of the register load inputs, thus providing a transfer path between the data in the output bus and the inputs of the selected destination register.



(a) Block diagram

Number of bits	3	3	3	5
Field	SELA	SELB	SELD	OPR

(b) Control Word

Figure 6-2 Register set with common ALU.

The control unit that operates the CPU bus system directs the information flow through the registers and ALU by selecting the various components in the system. For example, to perform the operation

$$R1 \leftarrow R2 + R3$$

the control must provide binary selection variables to the following selector inputs:

1. MUX A selector (SELA): to place the content of R2 into bus A.
2. MUX B selector (SELB): to place the content of R3 into bus B.
3. ALU operation selector (OPR): to provide the arithmetic addition A + B.
4. Decoder destination selector (SELD): to transfer the content of the output bus into R1.

The four control selection variables are generated in the control unit and must be available at the beginning of a clock cycle. The data from the two source registers propagate through the gates in the multiplexers and the ALU, to the output bus, and into the inputs of the destination register, all during the clock cycle interval. Then, when the next clock transition occurs, the binary information from the output bus is transferred into R1. To achieve a fast response time, the ALU is constructed with high-speed circuits. The buses are implemented with multiplexers or three-state gates.

6.2.1 CONTROL WORD

There are 14 binary selection inputs in the unit, and their combined value specifies a control word. The 14-bit control word is defined in Figure 6-2. It consists of four fields. Three fields contain three bits each, and one field has five bits. The three bits of SELA select a source register for the A input of the ALU. The three bits of SELB select a register for the B input of the ALU. The three bits of SELD select a destination register using the decoder and its seven load output. The five bits of OPR select one of the operations in the ALU. The 14-bit control word when applied to the selection inputs specify a particular microoperation.

Table 6-1 Encoding of Register Selection Fields

Binary Code	SELA	SELB	SELD
000	Input	Input	Input
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

The encoding of the register selections is specified in Table 6-1. The 3-bit binary code listed in the first column of the table specifies the binary code for each of the three fields. The register selected by fields SELA, SELB and SELD is the one whose decimal number is equivalent to the binary number in the code. When SELA or SELB is 000, the corresponding multiplexer selects the external input data. When SELD = 000, no destination register is selected but the contents of the output bus are available in the external output.

The ALU provides arithmetic and logic operations. In addition, the CPU must provide shift operations. The shifter may be placed in the input of the ALU to provide a preshift capability, or at the output of the ALU to provide post shifting capability. In some cases, the shift operations are included with the ALU. The encoding of the ALU operations for the CPU is specified in Table 6-2. The OPR field has five bits and each operation is designated with a symbolic name.

Table 6-2 Encoding of ALU Operations

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

6.2.2 EXAMPLE OF MICRO OPERATIONS

A control word of 14 bits is needed to specify a microoperation in the CPU. The control word for a given microoperation can be derived from the selection variables. For example, the subtract microoperation given by the statement.

$$R1 \leftarrow R2 \leftarrow R3$$

specifies R2 for the A input of the ALU, R3 for the B input of the ALU, R1 for the destination register, and an ALU operation to subtract A − B. Thus the control word is specified by the four fields and the corresponding binary value for each field is obtained from the encoding listed in Tables 6-1 and 6-2. The binary control word for the subtract microoperation is 010 011 001 00101 and is obtained as follows :

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

The control word for this micro operation and a few others are listed in Table 6-3. The increment and transfer microoperations do not use the B input of the ALU. For these cases, the B field is marked with a dash. We assign 000 to any unused field when formulating the binary control word, although any other binary number may be used. To place the content of a register into the output terminals we place the content of the register into the A input of the ALU, but none of the registers are selected to accept the data. The ALU operation TSFA places the data from the register, through the ALU, into the output terminals. The direct transfer from input to output is accomplished with a control word of all 0's (making the B field 000). A register can be cleared to 0 with an exclusive-OR operation. This is because $x \oplus x = 0$.

It is apparent from these examples that from these ex apples that many other microoperations can be generated in the CPU. The most efficient way to generate control words with a large number of bits is to store them in a memory unit. A memory unit that stores control words is referred to as a control memory. By reading consecutive control words from memory, it is possible to initiate the desired sequence of microoperation for the CPU. This type of control is referred to as microprogrammed control. The binary control word for the CPU will come from the outputs of the control memory marked "micro-ops."

Table 6-3 Examples of Microoperations for the CPU

Microoperation	Symbolic Designation				Control Word
	SELA	SELB	SELD	OPR	
$R1 \leftarrow R2 - R3$	R2	R3	R1	SUB	010 011 001 00101
$R4 \leftarrow R4 \vee R5$	R4	R5	R4	OR	100 101 100 01010
$R6 \leftarrow R6 + 1$	R6	—	R6	INCA	110 000 110 00001
$R7 \leftarrow R1$	R1	—	R7	TSFA	001 000 111 00000
Output $\leftarrow R2$	R2	—	None	TSFA	010 000 000 00000
Output $\leftarrow$ Input	Input	—	None	TSFA	000 000 000 00000
$R4 \leftarrow$ sh1 $R4$	R4	—	R4	SHLA	100 000 100 11000
$R5 \leftarrow 0$	R5	R5	R5	XOR	101 101 101 01100

6.3 STACK ORGANIZATION

A useful feature that is included in the CPU of most computers is a stack or last-in, first-out (LIFO) list. A stack is a storage device that stores information in such a manner that the item stored last is the first item retrieved. The operation of a stack can be compared to a stack of trays. The last tray placed on top of the stack is the first to be taken off.

The stack in digital computers is essentially a memory unit with an address register that can count only (after an initial value is loaded into it). The register that holds the address for the stack is called a stack pointer (SP) because its value always points at the top item in the stack. Contrary to a stack of trays where the tray itself may be taken out or inserted, the physical registers of a stack are always available for reading or writing. It is the content of the word that is inserted or deleted.

The two operations of a stack are the insertion and deletion of items. The operation of insertion is called push (or push-down) because it can be through of as the result of pushing a new item on top. The operation of deletion is called pop (or pop-up) because it can be thought of as the result of removing one item so that the stack pops up. However, nothing is pushed or popped in a computer stack. These operations are simulated by incrementing or decrementing the stack pointer register.

6.3.1 REGISTER STACK

A stack can be placed in portion of a large memory or it can be organized as a collection of a finite number of memory words or registers. Figure 6-3 shows the organization of a

64-word register stack. The stack pointer register SP contains a binary number whose value is equal to the address of the word that is currently on top of the stack. Three items are placed in the stack: A, B, and C, in that order. Item C is on top of the stack so that the content of SP is now 3. To remove the top item, the stack is popped by reading the memory word at address 3 and decrementing the content of SP. Item B is now on top the stack since SP holds address 2. To insert a new item, the stack is pushed by incrementing SP and writing a word in the next-higher location in the stack. Note that item C has been read out but not physically removed. This does not matter because when the stack is pushed, a new item is written in its place.

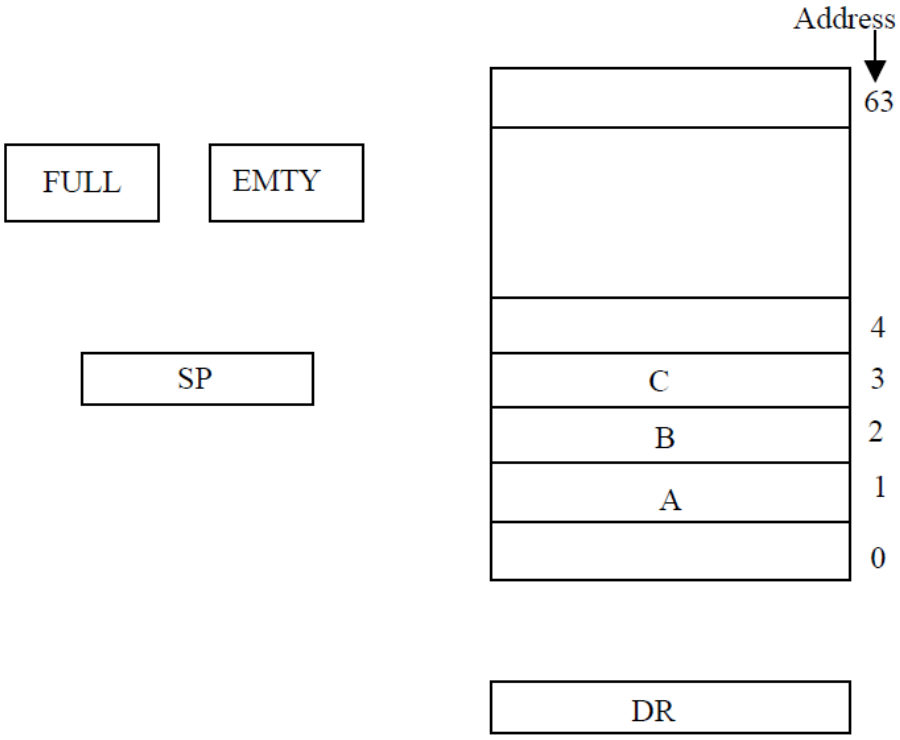


Figure 6-3 Block diagram of a 64-word stack

In a 64-word stack, the stack pointer contains 6 bits because $2^6 = 64$. 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. Similarly, when 000000 is decremented by 1, the result is 111111. The one-bit register FULL is set to 1 when the stack is full, and the one-bit register EMTY is set to 1 when the stack is empty of items. DR is the data register that holds the binary data to be written into or read out of the stack.

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

The stack pointer is incremented so that it points to the address of the next-higher word. A memory write operation inserts the word form DR into the top of the stack. Note that SP holds the address of the top of the stack and that M[SP] denotes the memory word specified by the address presently available in SP. The first item stored in the stack is at address 1. The last item is stored at address 0. If SP reaches 0, the stack is full of items so FULL is set to 1. This condition is reached if the top item prior to the last push was in location 63 and, after incrementing SP, the last item is stored in location 0.Once an item is stored in location 0, there are no more empty registers in the stack. If an item is written in the stack, obviously the stack cannot be empty, so EMTY is cleared to 0.

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 microoperations:

DR ← M [SP]	Read item from the top of stack
SP ← SP – 1	Decrement stack pointer
If (SP = 0) then (EMTY ← 1)	Check if stack is empty
FULL ← 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 to 1. This condition is reached if the item read was in location 1. Once this item is read out, SP is decremented and reaches the value 0, which is the initial value of SP. Note that if a pop operation reads the item from location 0 and then SP is decremented, SP changes to 111111, which is equivalent to decimal 63. In this configuration, the word in address 0 receives the last item in the stack. Note also that an erroneous operation will result if the stack is pushed when FULL = 1 or popped when EMTY = 1.

6.3.2 MEMORY STACK

A stack can exist as a stand-alone unit as in Fig. 6-3 or can be implemented in a random-access memory attached to a CPU. The implementation of a stack in the CPU is done by assigning a portion of memory to a stack operation and using a processor register as a stack pointer. Figure 6-4 shows a portion of computer memory partitioned into three segments: program, data, and stack. The program counter PC points at the address of the next instruction in the program. The address register AR points at an array of data. 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 execute phase to read an operand. SP is used to push or pop items into or from the stack

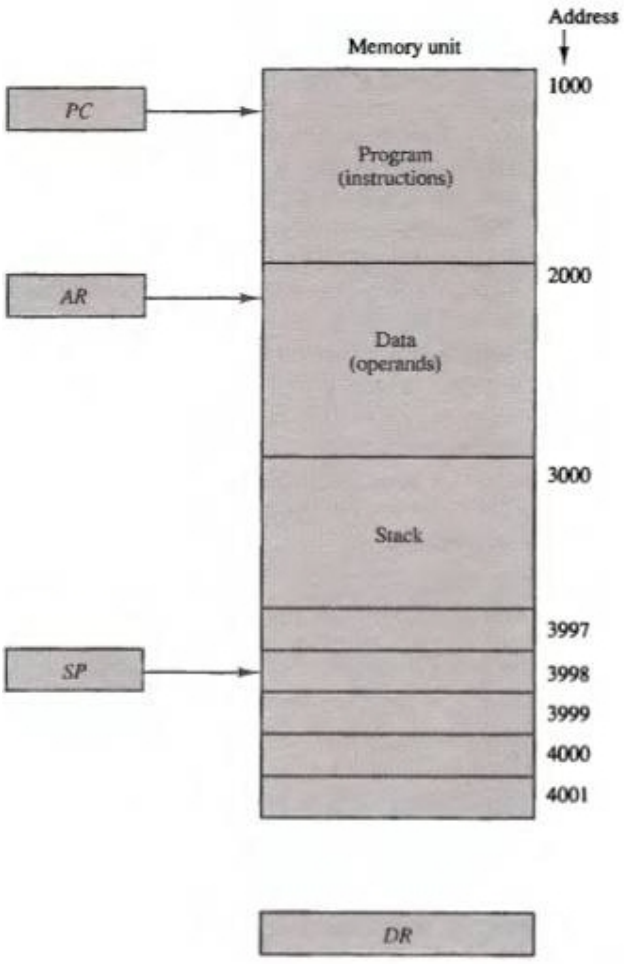


Figure 6-4 Computer memory with program, data, and stack segments.

As shown in Fig. 6-4, 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.

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

$$SP \leftarrow SP - 1$$

$$M[SP] \leftarrow DR$$

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

$$DR \leftarrow M[SP]$$

$$SP \leftarrow SP + 1$$

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

Most computers do not provide hardware to check for stack overflow (full stack) or underflow (empty stack). The stack limits can be checked by using two processor registers: one to hold the upper limit (3000 in this case), and the other to hold the lower limit (4001 in this case). After a push operation, SP is compared with the upper-limit register and after a pop operation, SP is compared with the lower-limit register.

The two microoperations needed for either the push or pop are (1) an access to memory through SP, and (2) updating SP. Which of the two microoperations is dine first and whether SP is updated by incrementing or decrementing depends on the organization of the stack. In Figure 6-4 the stack grows by decreasing the memory address. The stack may be constructed to grow by increasing the memory address as in Figure 6-3. In such a case, SP is incremented for the push operation and decremented for the pop operation. A stack may be constructed so that SP points at the next empty location above the top of the stack. In this case the sequence of microoperations must be interchanged.

A stack pointer is loaded with an initial value. This initial value must be the bottom address of an assigned stack in memory. Henceforth, SP is automatically decremented or incremented with every push or pop operation. The advantage of a memory stack is that the CPU can refer to it without having to specify an address, since the address is always available and automatically updated in the stack pointer.

6.3.3 REVERSE POLISH NOTATION

A stack organization is very effective for evaluating arithmetic expressions. The common mathematical method of writing arithmetic expression imposes difficulties when evaluated by a computer. The common arithmetic expressions are written in infix notation, with each operation written between the operands. Consider the simple arithmetic expression

$$A * B + C * D$$

The star (denoting multiplication) is placed between two operands A and B or C and D. The plus is between the two products. To evaluate this arithmetic expression it is necessary to compute the product A * B, store this product while computing C * D, and then sum the two products. From this example we see that to evaluate arithmetic expressions in infix notation it is necessary to scan back and forth along the expression to determine the next operation to be performed.

The polish mathematician Lukasiewicz showed that arithmetic expressions can be represented in prefix notation. This representation often referred to as Polish notation, places the operator before the operands. The postifix notation, referred to as reverse polish notation (RPN), places the operator after the operands. The following examples demonstrate the three representations:

A + B	Infix notation
+ AB	Prefix or Polish notation
AB +	Postfix or reverse Polish notation

The reverse Polish notation is in a form suitable for stack manipulation. The expression

$$A * B + C * D$$

is written is reverse Polish notation as

$$AB * CD * +$$

and is evaluated as follows: scan the expression from left to right. When an 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 form the result of the operation. Continue to scan the expression and repeat the procedure for every operator encountered until there are no more operators.

For the expression above we find the operator * after A and B. We perform the operation A * B and replace A, B, and * by the product to obtain

$$(A * B)\ CD * +$$

where (A * B) is a single quantity obtained from the product. The next operator is a * and its previous two operands are C and D, so we perform C * D and obtain an expression with two operands and one operator:

$$(A * B)\ (C * D) +$$

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. Consider the expression

$$(A + B) * [C * (D + E) + F]$$

to evaluate the expression we must first perform the arithmetic inside the parentheses (A + B) and (D + E). Next we must calculate the expression inside the square brackets. The multiplication of C * (D + E) must be done prior to the addition of F since multiplication has precedence over addition. The last operation is the multiplication of the two terms between the parentheses and brackets. The expression can be converted to reverse Polish notation, without the use of parentheses, by taking into consideration the operation hierarchy. The converted expression is

$$AB + DE + C * F + *$$

Proceeding from left to right, we first add A and B, then add D and E. At this point we are left with

$$(A + B)\ (D + E)\ C * E + *$$

where (A + B) and (D + E) are each a single number obtained from the sum. The two operands for the next * are C and (D + E). These two numbers are multiplied and the product added to F. The final * causes the multiplication of the two terms.

6.3.4 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 computers. The stack is particularly useful

for handling long, complex problems involving chain calculations. It is based on the fact that any arithmetic expression can be expressed in parentheses-free Polish notation.

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

The following numerical example may clarify this procedure. Consider the arithmetic expression

$$(3 * 4) + (5 * 6)$$

In reverse Polish notation, it is expressed as

$$34 * 56 * +$$

Now consider the stack operating shown in Figure 6-5. Each box represents one represents one stack operation and the arrow always points to the top of the stack. Scanning the expression from left to right, we encounter two operands. First the number 3 is pushed into the stack, then the number 4. The next symbol is the multiplication operator $*$. This causes a multiplication of the two topmost items in the stack. The stack is then popped and the product is placed on top of the stack, replacing the two original operands. Next we encounter the two operands 5 and 6, so they are pushed into the stack. The stack operation that results from the next $*$ replaces these two numbers by their product. The last operation causes an arithmetic addition of the two topmost numbers in the stack to produce the final result of 42.

Scientific calculators that employ an internal stack require that the user convert the arithmetic expressions into reverse Polish notation. Computers that use a stack-organized CPU provide a system program to perform the conversion for the user. Most compilers, irrespective of their CPU organization, convert all arithmetic expressions into Polish notation anyway because this is the most efficient method for translating arithmetic

expressions into machine language instructions. So in essence, a stack-organized CPU may be more efficient in some applications than a CPU without a stack.

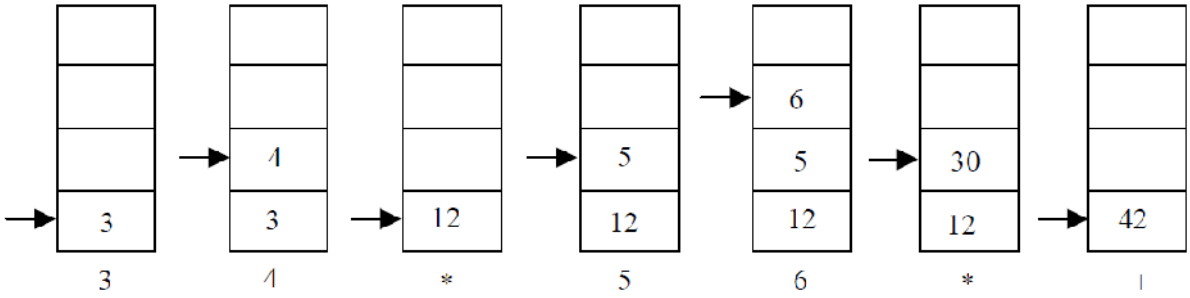


Figure 6.5 Stack operations to evaluate $(3 * 4 + 5 * 6)$

6.4 ADDRESSING MODES

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

1. To give programming versatility to the user by providing such facilities as pointers to memory, counters for loop control, indexing of data, and program relocation.
2. To reduce the number of bits in the addressing field of the instruction.
3. The availability of the addressing modes gives the experienced assembly language programmer flexibility for writing programs that are more efficient with respect to the number of instructions and execution time.

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

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

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

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

An example of an instruction format with a distinct addressing mode field is shown in Figure 6-6. The operation code specified the operation to be performed. The mode field is sued to locate the operands needed for the operation. There may or may not be an address field in the instruction. If there is an address field, it may designate a memory address or a processor register. Moreover, as discussed in the preceding section, the instruction may have more than one address field, and each address field may be associated with its own particular addressing mode.

Figure 6.6 Instruction format with mode field

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

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

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

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

Figure 6-7 Immediate addressing mode

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

Example:

1. ADD 10 will increment the value stored in the accumulator by 10.
2. MOV R #20 initializes register R to a constant value 20.

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

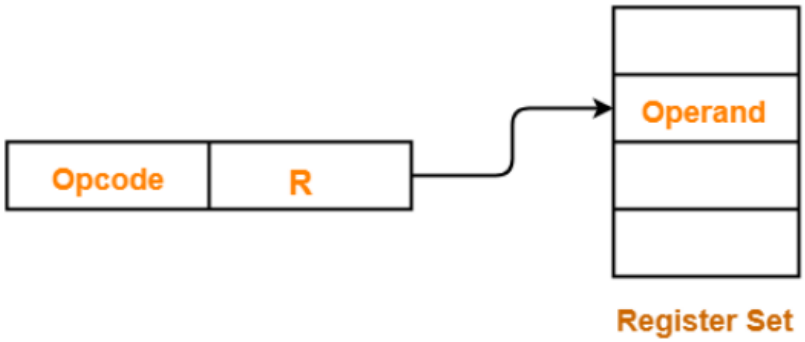


Figure 6-8 Register Addressing Mode

Example:

1. ADD R will increment the value stored in the accumulator by the content of register R.

$$AC \leftarrow AC + [R]$$

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

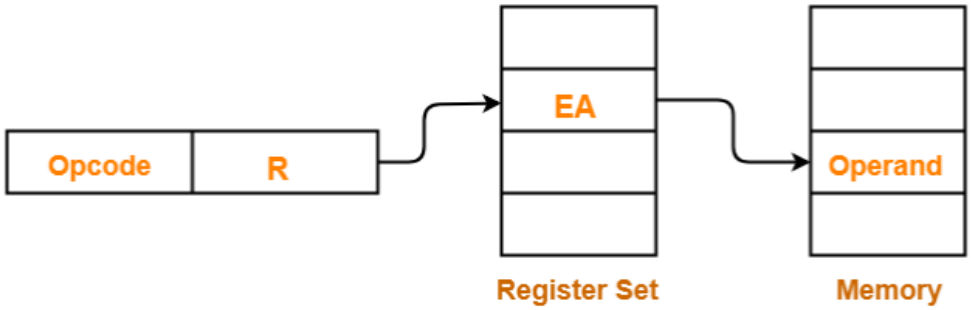


Figure 6-9 Register Indirect Addressing Mode

Example:

1. ADD R will increment the value stored in the accumulator by the content of memory location specified in register R.

$$AC \leftarrow AC + [[R]]$$

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

The address field of an instruction is used by the control unit in the CPU to obtain the operand from memory. Sometimes the value given in the address field is the address of the operand, but sometimes it is just an address from which the address of the operand is calculated.

Effective Address of the Operand

= Content of Register

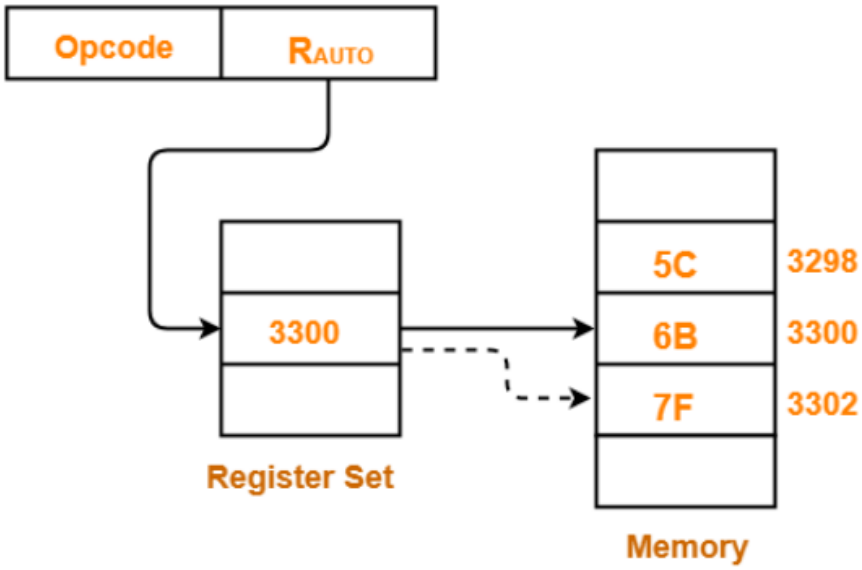


Figure 6-10 Autoincrement Addressing Mode

Assume operand size = 2 bytes.

Here,

- After fetching the operand 6B, the instruction register R_{AUTO} will be automatically incremented by 2.
- Then, updated value of R_{AUTO} will be 3300 + 2 = 3302.
- At memory address 3302, the next operand will be found.

To differentiate among the various addressing modes it is necessary to distinguish between the address part of the instruction and the effective address used by the control when executing the instruction. The effective address is defined to be the memory address obtained from the computation dictated by the given addressing mode. The effective address is the address of the operand in a computational-type instruction. It is the address where control branches in response to a branch-type instruction.

Effective Address of the Operand

= Content of Register – Step Size

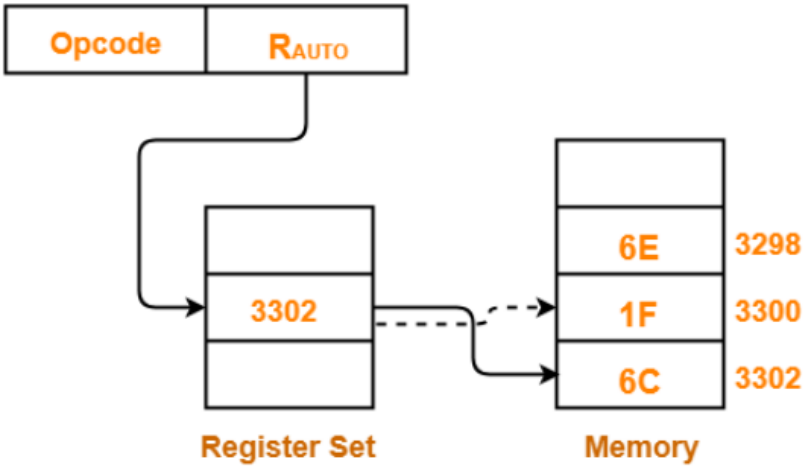


Figure 6-11 Autodecrement Addressing Mode

Assume operand size = 2 bytes.

Here,

- First, the instruction register R_{AUTO} will be decremented by 2.
- Then, updated value of R_{AUTO} will be 3302 – 2 = 3300.
- At memory address 3300, the operand will be found.

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

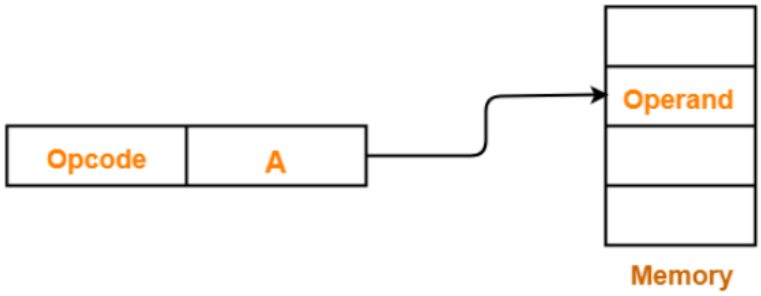


Figure 6-12 Direct Address Mode

Example:

1. ADD X will increment the value stored in the accumulator by the value stored at memory location X.

$$AC \leftarrow AC + [X]$$

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

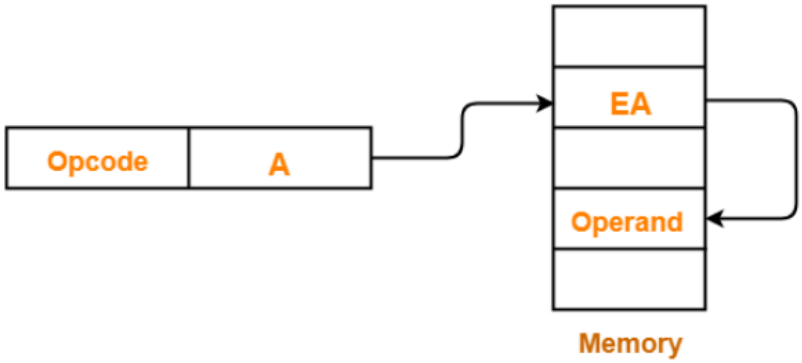


Figure 6-13 Indirect Address Mode

Example:

1. ADD X will increment the value stored in the accumulator by the value stored at memory location specified by X.

$$AC \leftarrow AC + [[X]]$$

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

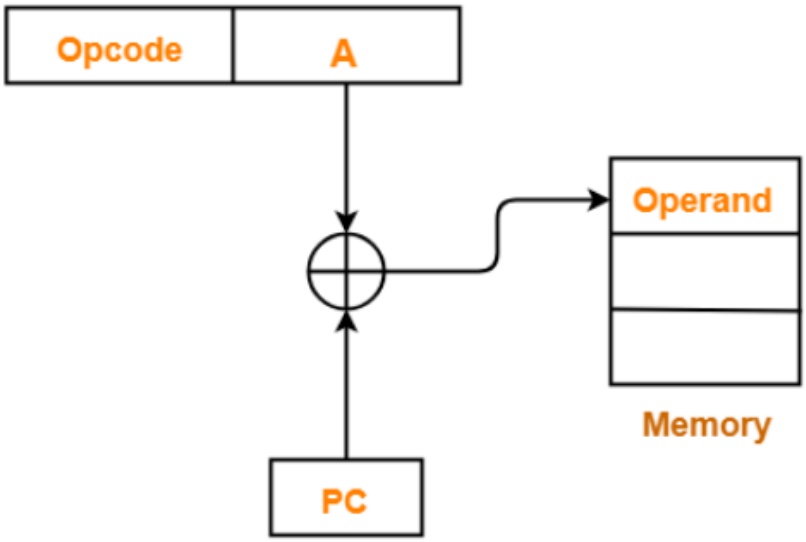


Figure 6-14 Relative Addressing Mode

Indexed Addressing Mode: In this mode the content of an index register is added to the address part of the instruction to obtain the effective address. The index register is a special CPU register that contains an index value. The address field of the instruction defines the beginning address of a data array in memory. Each operand in the array is stored in memory relative to the beginning address. The distance between the beginning address and the address of the operand is the index value stores in the index register. Any operand in the array can be accessed with the same instruction provided that the index register contains the correct index value. The index register can be incremented to facilitate access to consecutive operands. Note that if an index type instruction does not include an address field in its format, the instruction converts to the register indirect mode of operation.

Some computers dedicate one CPU register to function solely as an index register. This register is involved implicitly when the index-mode instruction is used. In computers with many processor registers, any one of the CPU registers can contain the index number. In such a case the register must be specified explicitly in a register field within the instruction format.

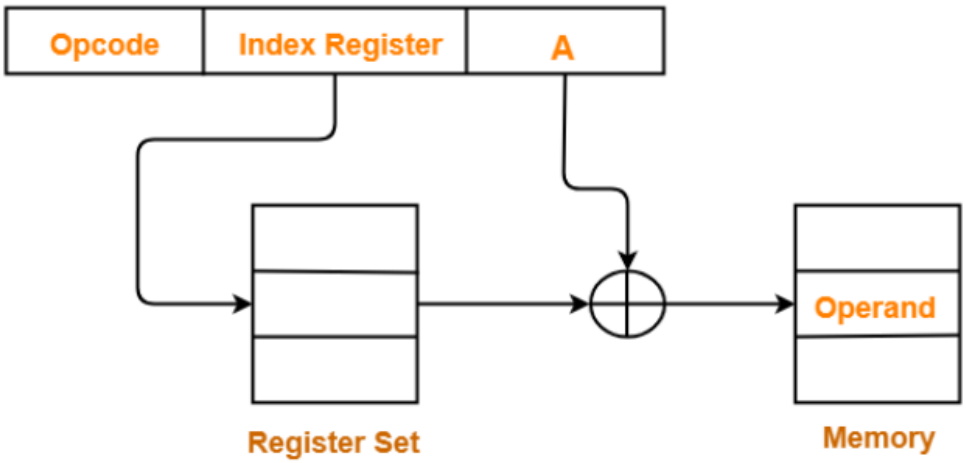


Figure 6-15 Indexed Addressing Mode

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

Effective Address

= Content of Base Register + Address part of the instruction

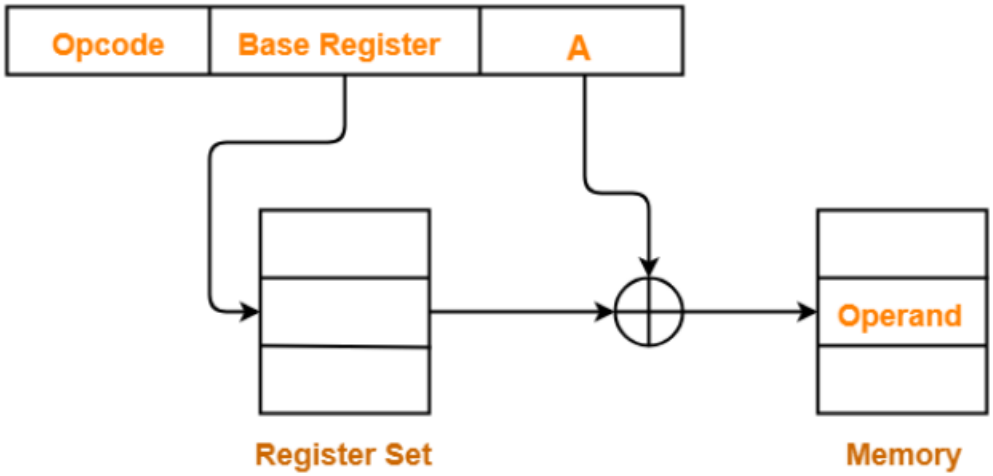


Figure 6-16 Base Register Addressing Mode

6.4.1 NUMERICAL EXAMPLE

To show the differences between the various modes, we will show the effect of the addressing modes on the instruction defined in Figure 6-17 The two-word instruction at address 200 and 201 is a "load to AC" instruction with an address field equal to 500. The first word of the instruction specifies the operation code and mode, and the second word specifies the address part. PC has the value and the content of an index register XR is 100. AC receives the operand after the instruction is executed. The figure lists a few pertinent addresses and shows the memory content at each of these addresses.

The mode field of the instruction can specify any one of a number of modes. For each possible mode we calculate the effective address and the operand that must be loaded into AC. In the direct address mode the effective address is the address part of the instruction 500 and the operand to be loaded into AC is 800. In the immediate mode the second word of the instruction is taken as the operand rather than an address, so 500 is loaded into AC. (The effective address in this case is 201.) In the direct mode the effective address is stored in memory at address 500. Therefore, the effective address is 800 and the operand is 300. In the relative mode the effective address is 500 + 202 = 702 and the operand is 325. (Note that the value in PC after the fetch phase and during the execute phase is 202.) In the index mode the effective address is XR + 500 = 100 + 500 = 600 and operand is 900. In the register mode the operand is in R1 and 400 is loaded into AC. (There is no effective address in this case.) In the register indirect mode the effective address is 400, equal to the content of R1 and the operand loaded into AC is 700.

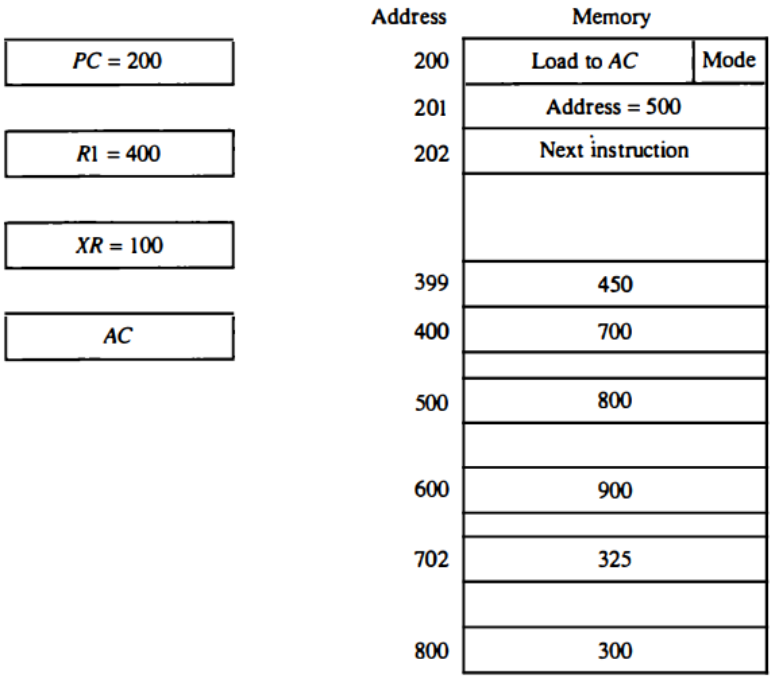


Figure 6-17 Numerical example for addressing modes

The autoincrement mode is the same as the register indirect mode except that R1 is incremented to 401 after the execution of the instruction. The autodecrement mode decrements R1 to 399 prior to the execution of the instruction. The operand loaded into AC is now 450. Table 6-4 lists the value of the effective address and the operand loaded into AC for the nine addressing modes.

Table 6-4 Tabular List of Numerical Example

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

6.5 FLYNN'S CLASSICAL TAXONOMY

Among mentioned above the one widely used since 1966, is Flynn's Taxonomy. This taxonomy distinguishes multi-processor computer architectures according two independent dimensions of Instruction stream and Data stream. An instruction stream is sequence of instructions executed by machine. And a data stream is a sequence of data including input, partial or temporary results used by instruction stream. Each of these dimensions can have only one of two possible states: Single or Multiple. Flynn's classification depends on the distinction between the performance of control unit and the data processing unit rather than its operational and structural interconnections. Following are the four category of Flynn classification and characteristic feature of each of them.

6.5.1 SINGLE INSTRUCTION STREAM, SINGLE DATA STREAM (SISD)

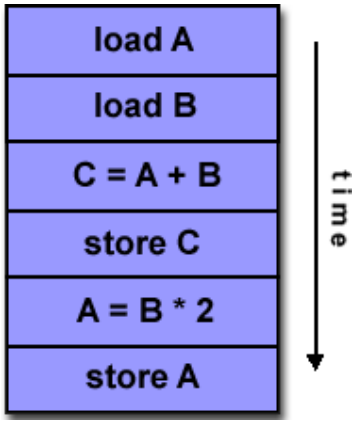


Figure 6.18 Execution of instruction in SISD processors

The figure 6.18 is represents a organization of simple SISD computer having one control unit, one processor unit and single memory unit.

Figure 6.19 SISD Processor Organization

- They are also called scalar processor i.e., one instruction at a time and each instruction have only one set of operands.
- Single instruction: only one instruction stream is being acted on by the CPU during any one clock cycle
- Single data: only one data stream is being used as input during any one clock cycle
- Deterministic execution
- Instructions are executed sequentially.
- This is the oldest and until recently, the most prevalent form of computer
- Examples: most PCs, single CPU workstations and mainframes

6.5.2 SINGLE INSTRUCTION STREAM, MULTIPLE DATA STREAM (SIMD) PROCESSORS

- A type of parallel computer
- Single instruction: All processing units execute the same instruction issued by the control unit at any given clock cycle where there are multiple processor executing instruction given by one control unit.
- Multiple data: Each processing unit can operate on a different data element as shown if figure below the processor are connected to shared memory or interconnection network providing multiple data to processing unit

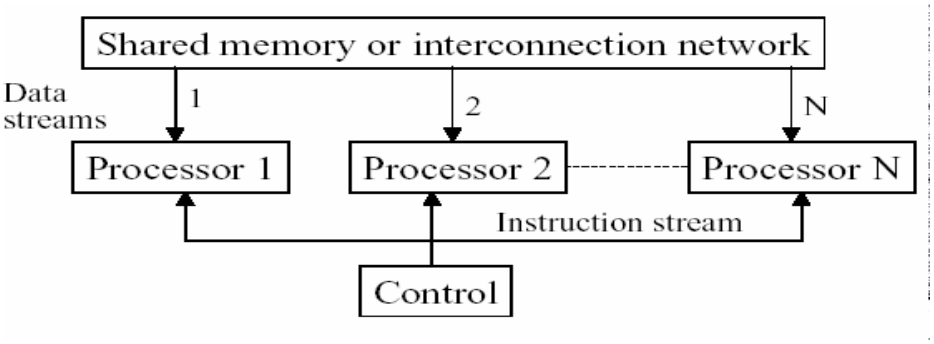


Figure 6.20 SIMD Processor Organization

- This type of machine typically has an instruction dispatcher, a very high bandwidth internal network, and a very large array of very small-capacity instruction units.

- Thus single instruction is executed by different processing unit on different set of data as shown in figure 6.20.
- Best suited for specialized problems characterized by a high degree of regularity, such as image processing and vector computation.
- Synchronous (lockstep) and deterministic execution.
- Two varieties: Processor Arrays e.g., Connection Machine CM-2, Maspar MP-1, MP-2 and Vector Pipelines processor e.g., IBM 9000, Cray C90, Fujitsu VP, NEC SX-2, Hitachi S820.

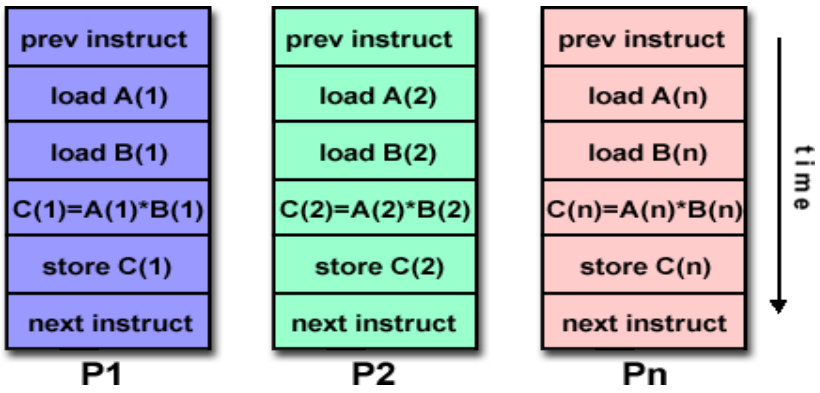


Figure 6.21 Execution of instructions in SIMD processors

6.5.3 MULTIPLE INSTRUCTION STREAM, SINGLE DATA STREAM (MISD)

- A single data stream is fed into multiple processing units.
- Each processing unit operates on the data independently via independent instruction streams as shown in figure 6.22 a single data stream is forwarded to different processing unit which are connected to different control unit and execute instruction given to it by control unit to which it is attached.

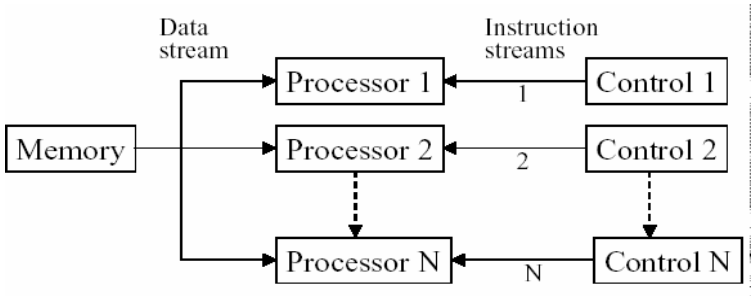


Figure 6.22 MISD processor organization

- Thus in these computers same data flow through a linear array of processors executing different instruction streams as shown in figure 6.23.
- This architecture is also known as systolic arrays for pipelined execution of specific instructions.
- Few actual examples of this class of parallel computer have ever existed. One is the experimental Carnegie-Mellon C.mmp computer (1971).
- Some conceivable uses might be:
  1. multiple frequency filters operating on a single signal stream
  2. multiple cryptography algorithms attempting to crack a single coded message.

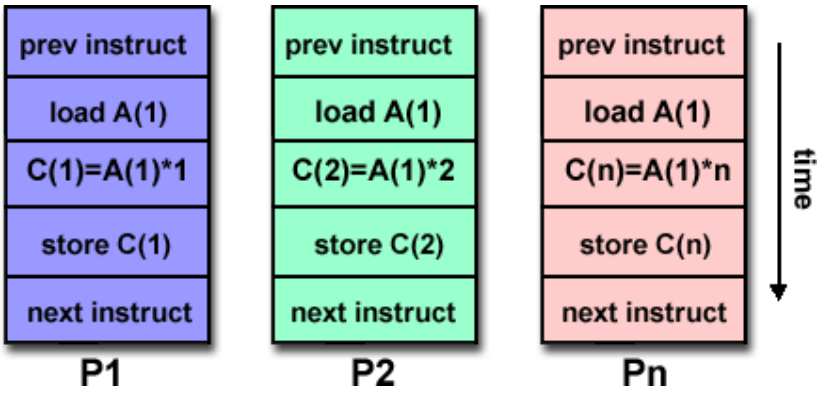


Figure 6.23 Execution of instructions in MISD processors

6.5.4 MULTIPLE INSTRUCTION STREAM, MULTIPLE DATA STREAM (MIMD)

- Multiple Instruction: every processor may be executing a different instruction stream
- Multiple Data: every processor may be working with a different data stream as shown in figure 6.24 multiple data stream is provided by shared memory.
- Can be categorized as loosely coupled or tightly coupled depending on sharing of data and control
- Execution can be synchronous or asynchronous, deterministic or nondeterministic

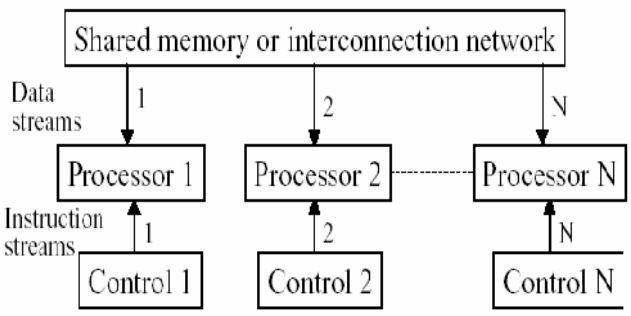


Figure 6.24 MIMD processor organizations

- As shown in figure 6.25 there are different processor each processing different task.
- Examples: most current supercomputers, networked parallel computer "grids" and multi-processor SMP computers - including some types of PCs.

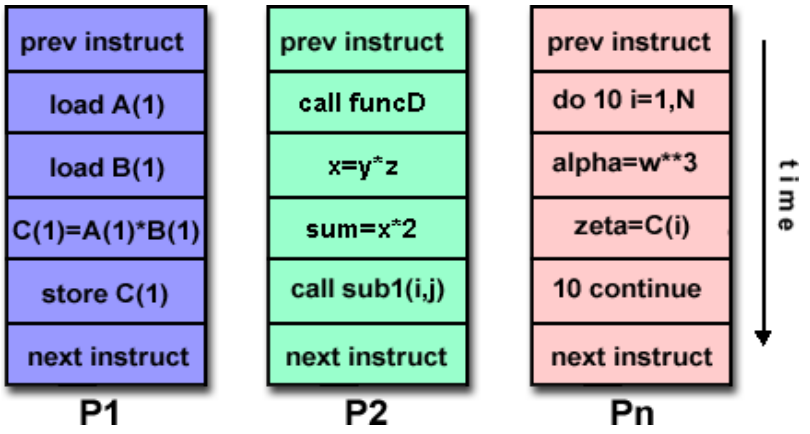


Figure 6.25 Execution of instructions MIMD processors

Here the some popular computer architecture and there types:

1) SISD IBM 701, IBM 1620, IBM 7090, PDP VAX11/ 780
2) SISD (With multiple functional units) IBM360/91 (3); IBM 370/168 UP
3) SIMD (Word Slice Processing) Iliac – IV ; PEPE
4) SIMD (Bit Slice processing) STARAN; MPP; DAP
5) MIMD (Loosely Coupled) IBM 370/168 MP; Univac 1100/80
6) MIMD (Tightly Coupled) Burroughs- D – 825

6.6 PERFORMANCE ATTRIBUTES

Performance of a system depends on

- hardware technology
- architectural features
- efficient resource management
- algorithm design
- data structures
- language efficiency
- programmer skill
- compiler technology

When we talk about performance of computer system we would describe how quickly a given system can execute a program or programs. Thus we are interested in knowing the turnaround time. Turnaround time depends on:

- disk and memory accesses
- input and output
- compilation time
- operating system overhead
- CPU time

An ideal performance of a computer system means a perfect match between the machine capability and program behavior. The machine capability can be improved by using better hardware technology and efficient resource management. But as far as program behavior is concerned it depends on code used, compiler used and other run time conditions. Also a machine performance may vary from program to program. Because there are too many programs and it is impractical to test a CPU's speed on all of them, benchmarks were developed. Computer architects have come up with a variety of metrics to describe the computer performance.

6.6.1 CLOCK RATE AND CPI / IPC

Since I/O and system overhead frequently overlaps processing by other programs, it is fair to consider only the CPU time used by a program, and the user CPU time is the most important factor. CPU is driven by a clock with a (usually measured in nanoseconds,

which controls the rate of internal constant cycle time operations in the CPU. The clock mostly has the constant cycle time (t in nanoseconds). The inverse of the cycle time is the clock rate ($f = 1/\tau$, measured in megahertz). A shorter clock cycle time, or equivalently a larger number of cycles per second, implies more operations can be performed per unit time. The size of the program is determined by the instruction count (Ic). The size of a program is determined by its instruction count, Ic, the number of machine instructions to be executed by the program. Different machine instructions require different numbers of clock cycles to execute. CPI (cycles per instruction) is thus an important parameter.

6.6.2 AVERAGE CPI

It is easy to determine the average number of cycles per instruction for a particular processor if we know the frequency of occurrence of each instruction type.

Of course, any estimate is valid only for a specific set of programs (which defines the instruction mix), and then only if there are sufficiently large number of instructions.

In general, the term CPI is used with respect to a particular instruction set and a given program mix. The time required to execute a program containing Ic instructions is just $T = Ic * CPI * \tau$.

Each instruction must be fetched from memory, decoded, then operands fetched from memory, the instruction executed, and the results stored.

The time required to access memory is called the memory cycle time, which is usually k times the processor cycle time τ. The value of k depends on the memory technology and the processor-memory interconnection scheme. The processor cycles required for each instruction (CPI) can be attributed to cycles needed for instruction decode and execution (p), and cycles needed for memory references (m* k).

The total time needed to execute a program can then be rewritten as

$T = Ic* (p + m*k)*\tau$.

6.6.3 MILLIONS OF INSTRUCTIONS PER SECOND (MIPS)

The millions of instructions per second, this is calculated by dividing the number of instructions executed in a running program by time required to run the program. The MIPS rate is directly proportional to the clock rate and inversely proportion to the CPI. All four systems attributes (instruction set, compiler, processor, and memory technologies) affect the MIPS rate, which varies also from program to program. MIPS

does not proved to be effective as it does not account for the fact that different systems often require different number of instruction to implement the program. It does not inform about how many instructions are required to perform a given task. With the variation in instruction styles, internal organization, and number of processors per system it is almost meaningless for comparing two systems.

MFLOPS (pronounced ``megaflops'') stands for ``millions of floating point operations per second.'' This is often used as a ``bottom-line'' figure. If one know ahead of time how many operations a program needs to perform, one can divide the number of operations by the execution time to come up with a MFLOPS rating. For example, the standard algorithm for multiplying n*n matrices requires 2n3 – n operations (n2 inner products, with n multiplications and n-1additions in each product). Suppose you compute the product of two 100 *100 matrices in 0.35 seconds. Then the computer achieves (2(100)3 – 100)/0.35 = 5,714,000 ops/sec = 5.714 MFLOPS

The term ``theoretical peak MFLOPS'' refers to how many operations per second would be possible if the machine did nothing but numerical operations. It is obtained by calculating the time it takes to perform one operation and then computing how many of them could be done in one second. For example, if it takes 8 cycles to do one floating point multiplication, the cycle time on the machine is 20 nanoseconds, and arithmetic operations are not overlapped with one another, it takes 160ns for one multiplication, and (1,000,000,000 nanosecond/1sec)*(1 multiplication / 160 nanosecond) = 6.25*106 multiplication /sec so the theoretical peak performance is 6.25 MFLOPS. Of course, programs are not just long sequences of multiply and add instructions, so a machine rarely comes close to this level of performance on any real program. Most machines will achieve less than 10% of their peak rating, but vector processors or other machines with internal pipelines that have an effective CPI near 1.0 can often achieve 70% or more of their theoretical peak on small programs.

6.6.4 THROUGHPUT RATE

Another important factor on which system’s performance is measured is throughput of the system which is basically how many programs a system can execute per unit time Ws. In multiprogramming the system throughput is often lower than the CPU throughput Wp which is defined as

Wp = f/(Ic * CPI)

Unit of Wp is programs/second.

Ws <Wp as in multiprogramming environment there is always additional overheads like timesharing operating system etc. An Ideal behavior is not achieved in parallel computers because while executing a parallel algorithm, the processing elements cannot devote 100% of their time to the computations of the algorithm. Efficiency is a measure of the fraction of time for which a PE is usefully employed. In an ideal parallel system efficiency is equal to one. In practice, efficiency is between zero and one

s of overhead associated with parallel execution

Speed or Throughput (W/Tn) - the execution rate on an n processor system, measured in FLOPs/unit-time or instructions/unit-time.

Speedup (Sn = T1/Tn) - how much faster in an actual machine, n processors compared to is called the asymptotic speedup. 1 will perform the workload. The ratio T1/T∞

Efficiency (En = Sn/n) - fraction of the theoretical maximum speedup achieved by n processors

Degree of Parallelism (DOP) - for a given piece of the workload, the number of processors that can be kept busy sharing that piece of computation equally. Neglecting overhead, we assume that if k processors work together on any workload, the workload gets done k times as fast as a sequential execution.

Scalability - The attributes of a computer system which allow it to be gracefully and linearly scaled up or down in size, to handle smaller or larger workloads, or to obtain proportional decreases or increase in speed on a given application. The applications run on a scalable machine may not scale well. Good scalability requires the algorithm and the machine to have the right properties

Thus in general there are five performance factors (Ic, p, m, k, t) which are influenced by four system attributes:

- instruction-set architecture (affects Ic and p) 19
- compiler technology (affects Ic and p and m)) cache and memory hierarchy
- CPU implementation and control (affects p *t) (affects memory access latency, k ´t
- Total CPU time can be used as a basis in estimating the execution rate of a processor.

6.7 CHECK YOUR PROGRESS

1. Component of CPU which is responsible for comparing the contents of two pieces of data is _____ .

2. The primary functions of a processor are _____, _____ and _____ .
3. The device which is used to connect a peripheral to bus is called _____.
4. _____ register is used in the control unit of the CPU to indicate the next instruction which is to be executed.
5. The addressing mode which makes use of in-direction pointers is _____ .
6. The addressing mode, where you directly specify the operand value is _____ .
7. _____ addressing mode execute its instructions within CPU without the necessity of reference memory for operands.
8. The effective address of base-register calculated by addition of _____ .
9. In Reverse Polish notation, expression A*B+C*D is written as _____ .
10. The addressing mode used in an instruction of the form ADD X Y, is _____.

6.8 SUMMARY

1. The part of the computer that performs the bulk of data-processing operations is called the central processing unit and is referred to as the CPU.
2. The CPU performs a variety of functions dictated by the type of instructions that are incorporated in the computer.
3. The registers communicate with each other not only for direct data transfers, but also while performing various micro operations. Hence it is necessary to provide a common unit that can perform all the arithmetic, logic, and shift micro operations in the processor.
4. The control unit that operates the CPU bus system directs the information flow through the registers and ALU by selecting the various components in the system.
5. A stack can be placed in portion of a large memory or it can be organized as a collection of a finite number of memory words or registers.
6. Reverse Polish notation, combined with a stack arrangement of registers, is the most efficient way known for evaluating arithmetic expressions.

7. A stack organization is very effective for evaluating arithmetic expressions. The common mathematical method of writing arithmetic expression imposes difficulties when evaluated by a computer.
8. The physical and logical structure of computers is normally described in reference manuals provided with the system.
9. The operation code field of an instruction is a group of bits that define various processor operations, such as add, subtract, complement, and shift.
10. The instruction set of a typical of a typical RISC processor is restricted to the use of load and store instructions when communicating between memory and CPU.
11. The operation field of an instruction specifies the operation to be performed. This operation must be executed on some data stored in computer registers or memory words. The way the operands are chosen during program execution in dependent on the addressing mode of the instruction.

6.9 KEYWORDS

1. **General purpose register** can store a data or a memory location address.
2. **Stack** is a storage structure that stores information in such a way that the last item stored is the first item retrieved.
3. **Addressing mode** specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually executed.
4. **Effective address** is any operand to an instruction which references memory.
5. **Indirect addressing** is a scheme in which the address specifies which memory word or register contains not the operand but the address of the operand.
6. **Immediate addressing** is an addressing form in which the byte value to be used or retrieved in the instruction, is located immediately after the opcode for the instruction itself.
7. **Implied Addressing** also known as "Implicit" or "Inherent" addressing mode is the addressing mode in which, no operand(register or memory location or data) is specified in the instruction. As in this mode the operand are specified implicit in the definition of instruction.

6.10 SELF ASSESSMENT TEST

1. What is the difference between a microprocessor and a microprogram? Is it possible to design a microprocessor without a microprogram? Are all micro programmed computers also microprocessors?
2. Explain the difference between hardwired control and micro programmed control. Is it possible to have a hardwired control associated with a control memory?
3. Define the following: (a) microoperation; (b) microinstruction; (c) microprogram; (d) microcode.
4. What is addressing mode? How many addressing modes are there? Use particular cases to explain the concept of the addressing modes.
5. The micro programmed control organization showing in Figure 6-1 has the following propagation delay times. 40 ns to generate the next address, 10 ns to transfer the address into the control address register, 40 ns to access the control memory ROM, 10 ns to transfer the microinstruction into the control data register, and 40 ns to perform the required micro operations specified by the control word. What is the maximum clock frequency that the control can use? What would the clock frequency be if the control data register is not used?
6. The system shown in Figure 6-2 uses a control memory of 1024 words of 32 bits each. The microinstruction has three fields as shown in the diagram. The micro operations field has 16 bits.
   a. How many bits are there in the branch address field and the select field?
   b. If there are 16 status bits in the system, how many bits of the branch logic are used to select a status bit?
   c. How many bits are left to select an input for the multiplexers?
7. A bus-organized CPU similar to Figure 6-2 has 16 registers with 32 bits in each, ALU, and a destination decoder.
   a. How many multiplexers are there in the A bus, and what is the size of each multiplexer?
   b. How many selection inputs are needed for MUX A and MUX B?
   c. How many inputs and outputs are there in the decoder?
   d. How many inputs and outputs are there in the ALU for data, including input and output carries?

e. Formulate a control word for the system assuming that the ALU has 35 operations.

8. The bus system of Figure 6-2 has the following propagation delay times: 30 ns for the signals to propagate through the multiplexers, 80 ns to perform the ADD operation in the ALU, 20 ns delay in the destination decoder, and 10 ns to clock the data into the destination register. What is the minimum cycle time that can be used for the clock?
9. Specify the control word that must be applied to the processor of Figure 6-2 to implement the following micro operations.
   a. R1 ← R2 + R3
   b. R4 ← R4
   c. R5 ← R5 − 1
   d. R6 ← sh1 R1
   e. R7 ← input
10. Determine the microoperations that will be executed in the processor of Figure 6-2 when the following 14-bit control words are applied.
   1. 00101001100101
   2. 00000000000000
   3. 01001001001100
   4. 11110001110000

6.11 ANSWER TO CHECK YOUR PROGRESS

1. ALU
2. fetch, decode and execute
3. Interface
4. Program Counter
5. Indirect addressing mode
6. Immediate
7. Register Mode
8. index register contents to the partial address in instruction
9. AB*CD*+
10. index

6.12 REFERENCES / SUGGESTED READINGS

1. Computer Organization and Architecture, Rajaram & Radhakrishan, PHI.
2. Computer Organization & Architecture: Designing for Performance, Stalling, PHI.
3. Computer Organization and Design, Pal Choudhary, PHI.
4. Computer Systems Organization & Architecture, Carpenelli, Pearson Education.
5. Computer Organization and Architecture, Stalling, Pearson Education.
6. Computer System Architecture, Morris Mano, PHI.
7. Computer Architecture and Organization, McGraw Hill Company, New Delhi. J.P. Hayes.

SUBJECT: COMPUTER SYSTEM ARCHITECTURE	
COURSE CODE: MCA-24	**AUTHOR: DR. MANOJ DUHAN**
LESSON NO. 7	

MEMORY ORGANIZATION

STRUCTURE

7-0 LEARNING OBJECTIVES

After reading this lesson, you should be able to describe the Main Memory, Auxiliary Memory, Associative Memory, Cache Memory and Virtual Memory.

7-1 INTRODUCTION

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

The total memory capacity of a computer can be visualized as being a hierarchy of components. The memory hierarchy system consists of tall storage devices employed in a computer system from the slow but high-capacity auxiliary memory to a relatively faster main memory, to an even smaller and faster cache memory accessible to the high-speed processing logic. Figure 7-1 illustrates the components in a typical memory hierarchy. At the bottom of the hierarchy are the relatively slow magnetic tapes used to store removable files. Next are the magnetic disks used as backup storage. The main memory occupies a central position by being able to communicate directly with the CPU and with auxiliary memory devices through an I/O processor. When programs not residing in main memory are needed by the CPU, they are brought in from auxiliary memory. Programs not currently needed in main memory are transferred into auxiliary memory to provide space for currently used programs and data.

A special very-high speed memory called a cache is sometimes used to increase the speed of processing by making current programs and data available to the CPU at a rapid rate. The cache memory is employed in computer systems to compensate for the speed differential between main memory access time and processor logic. CPU logic is usually faster than main memory access time, with the result that processing speed is limited primarily by the speed of main memory.

A technique used to compensate for the mismatch in operating speeds is to employ in extremely fast, small cache between the CPU and main memory whose access time is close to processor logic clock cycle time. The cache is used for storing segments of programs currently being executed in the CPU and temporary data frequently needed in the present calculations by

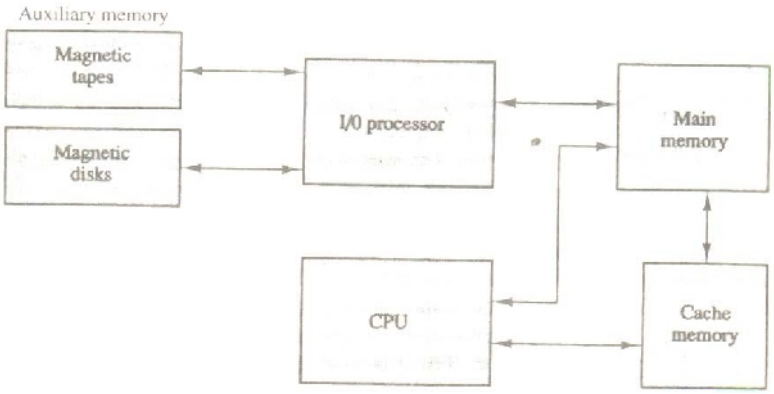


Figure 7-1 Memory hierarchy in a computer system.

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

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

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

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

With multiprogramming the need arises for running partial programs, for varying the amount of main memory in use by a given program, and for moving programs around the memory hierarchy. Computer programs are sometimes too long to be accommodated in the total space available in main memory. Moreover, a computer system uses many programs and all the programs cannot reside in main memory at all times. A program with its data normally resides in auxiliary memory. When the program or a segment of the program is to be executed, it is transferred to main memory to be executed by the CPU. Thus one may think of auxiliary memory as containing the totality of information stored in a computer system. It is the task of the operating system to maintain in main memory a portion of this information that is currently active. The part of the computer system that supervises the flow of information between auxiliary memory and main memory is called the memory management system.

7.2 MAIN MEMORY

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

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

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

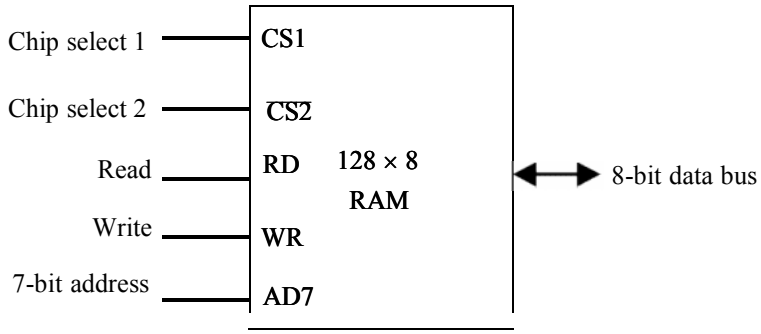
RAM and ROM chips are available in a variety of sizes. If the memory needed for the computer is larger than the capacity of one chip, it is necessary to combine a number of chips to form the required memory size. To demonstrate the chip interconnection, we will show an example of a 1024×8 memory constructed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

7.2.1 RAM AND ROM CHIPS

A RAM chip is better suited for communication with the CPU if it has one or more control inputs that select the chip only when needed. Another common feature is a bidirectional data bus that allows the transfer of data either from memory to CPU during a read operation or from CPU to memory during a write operation. A bidirectional bus can be constructed with three-state buffers. A three-state buffer output can be placed in one of three possible states: a signal equivalent to logic 1, a signal equivalent to logic 0, or a high-impedance state. The logic 1 and 0 are normal digital signals. The high-impedance state behaves like an open circuit, which means that the output does not carry a signal and has no logic significance.

The block diagram of a RAM chip is shown in Fig. 7-2. The capacity of the memory is 128 words of eight bits (one byte) per word. This requires a 7-bit

Figure 7.2 Typical RAM chip.



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

Function table

address and an 8-bit bidirectional data bus. The read and write inputs specify the memory operation and the two chips select (CS) control inputs are for enabling the chip only when it is selected by the microprocessor. The availability of more than one control input to select the chip facilitates the decoding of the address lines when multiple chips are used in the microcomputer. The read and write inputs are sometimes combined into one line labeled R/W. When the chip is selected, the two binary states in this line specify the two operations or read or write.

The function table listed in Fig. 7-2(b) specifies the operation of the RAM chip. The unit is in operation only when CSI = 1 and $\overline{CS2}$ = 0. The bar on top of the second select variable indicates that this input in enabled when it is equal to 0. If the chip select inputs are not enabled,

or if they are enabled but the read but the read or write inputs are not enabled, the memory is inhibited and its data bus is in a high-impedance state. When SC1 = 1 and $\overline{CS2}$ = 0, the memory can be placed in a write or read mode. When the WR input is enabled, the memory stores a byte from the data bus into a location specified by the address input lines. When the RD input is enabled, the content of the selected byte is placed into the data bus. The RD and WR signals control the memory operation as well as the bus buffers associated with the bidirectional data bus.

A ROM chip is organized externally in a similar manner. However, since a ROM can only read, the data bus can only be in an output mode. The block diagram of a ROM chip is shown in Fig. 7-3. For the same-size chip, it is possible to have more bits of ROM occupy less space than in RAM. For this reason, the diagram specifies a 512-byte ROM, while the RAM has only 128 bytes.

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

7.2.2 MEMORY ADDRESS MAP

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

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

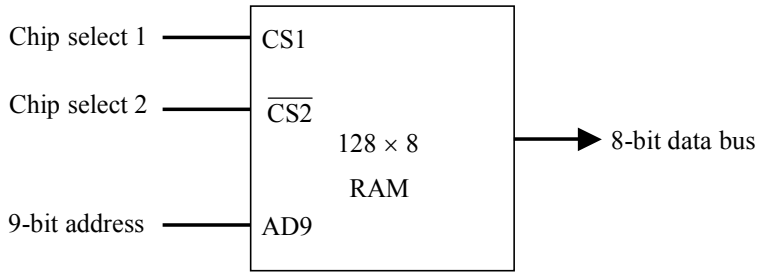


Figure 7-3 Typical ROM chip.

to be used are specified in Figs. 7-2 and 7-3. The memory address map for this configuration is shown in Table 7-1. The component column specifies whether a RAM or a ROM chip is used.

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

ROM. It is now necessary to distinguish between four RAM chips by assigning to each a different address. For this particular example we choose bus lines 8 and 9 to represent four distinct binary combinations. Note that any other pair of unused bus lines can be chosen for this purpose. The table clearly shows that the nine low-order bus lines constitute a memory space fro RAM equal to $2^9 = 512$ bytes. The distinction between a RAM and ROM address is done with another bus line. Here we choose line 10 for this purpose. When line 10 is 0, the CPU selects a RAM, and when this line is equal to 1, it selects the ROM.

The equivalent hexadecimal address for each chip is obtained form the information under the address bus assignment. The address bus lines are subdivided into groups of four bits each so

TABLE 7-1 Memory Address Map for Microprocomputer

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

that each group can be represented with a hexadecimal digit. The first hexadecimal digit represents lines 13 to 16 and is always 0. The next hexadecimal digit represents lines 9 to 12, but lines 11 and 12 are always 0. The range of hexadecimal addresses for each component is determined from the x's associated with it. These x's represent a binary number that can range from an all-0's to an all-1's value.

7.2.3 MEMORY CONNECTION TO CPU

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

The selection between RAM and ROM is achieved through bus line 10. The RAMs are selected when the bit in this line is 0, and the ROM when the bit is 1. The other chip select input in the ROM is connected to the RD control line for the ROM chip to be enabled only during a read operation. Address bus lines 1 to 9 are applied to the input address of ROM without going through the decoder. This assigns addresses 0 to 511 to RAM and 512 to 1023 to ROM. The data bus of the ROM has only an output capability, whereas the data bus connected to the RAMs can transfer information in both directions.

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

7.3 AUXILIARY MEMORY

The most common auxiliary memory devices used in computer systems are magnetic disks and tapes. Other components used, but not as frequently, are magnetic drums, magnetic bubble memory, and optical disks. To understand fully the physical mechanism of auxiliary memory devices one must have a knowledge of magnetics, electronics, and electromechanical systems. Although the physical properties of these storage devices can be quite complex, their logical properties can be characterized and compared by a few parameters. The important characteristics of any device are its access mode, access time, transfer rate, capacity, and cost.

The average time required to reach a storage location in memory and obtain its contents is called the access time. In electromechanical devices with moving parts such as disks and tapes, the access time consists of a seek time required to position the read-write head to a location and a transfer time required to transfer data to or from the device. Because the seek time is usually much longer than the transfer time, auxiliary storage is organized in records or blocks. A record is a specified number of characters or words. Reading or writing is always done on entire records. The transfer rate is the number of characters or words that the device can transfer per second, after it has been positioned at the beginning of the record.

Magnetic drums and disks are quite similar in operation. Both consist of high-speed rotating surfaces coated with a magnetic recording medium. The rotating surface of the drum is a cylinder and that of the disk, a round flat plate. The recording surface rotates at uniform speed and is not stared or stopped during access operations. Bits are recorded as magnetic spots on the surface as it passes a stationary mechanism called a write head. Stored bits are detected by a change in magnetic field produced by a recorded spot on the surface as it passes through a read head. The amount of surface available for recording in a disk is greater than in a drum of equal physical size. Therefore, more information can be stored on a disk than on a drum of comparable size. For this reason, disks have replaced drums in more recent computers

7.3.1 MAGNETIC DISKS

A magnetic disk is a circular plate constructed of metal or plastic coated with magnetized material. Often both sides of the disk are used and several disks may be stacked on one spindle with read/write heads available on each surface. All disks rotate together at high speed and are not stopped or started from access purposes. Bits are stored in the magnetized surface in spots along concentric circles called tracks. The tracks are commonly divided into sections called sectors. In most systems, the minimum quantity of information which can be transferred is a sector. The sub division of tone disk surface into tracks and sectors.

Some units use a single read/write head from each disk surface. In this type of unit, the track address bits are used by a mechanical assembly to move the head into the specified track position before reading or writing. In other disk systems, separate read/write heads are provided for each track in each surface. The address can then select a particular track electronically through a decoder circuit. This type of unit is more expensive and is found only in very large computer systems.

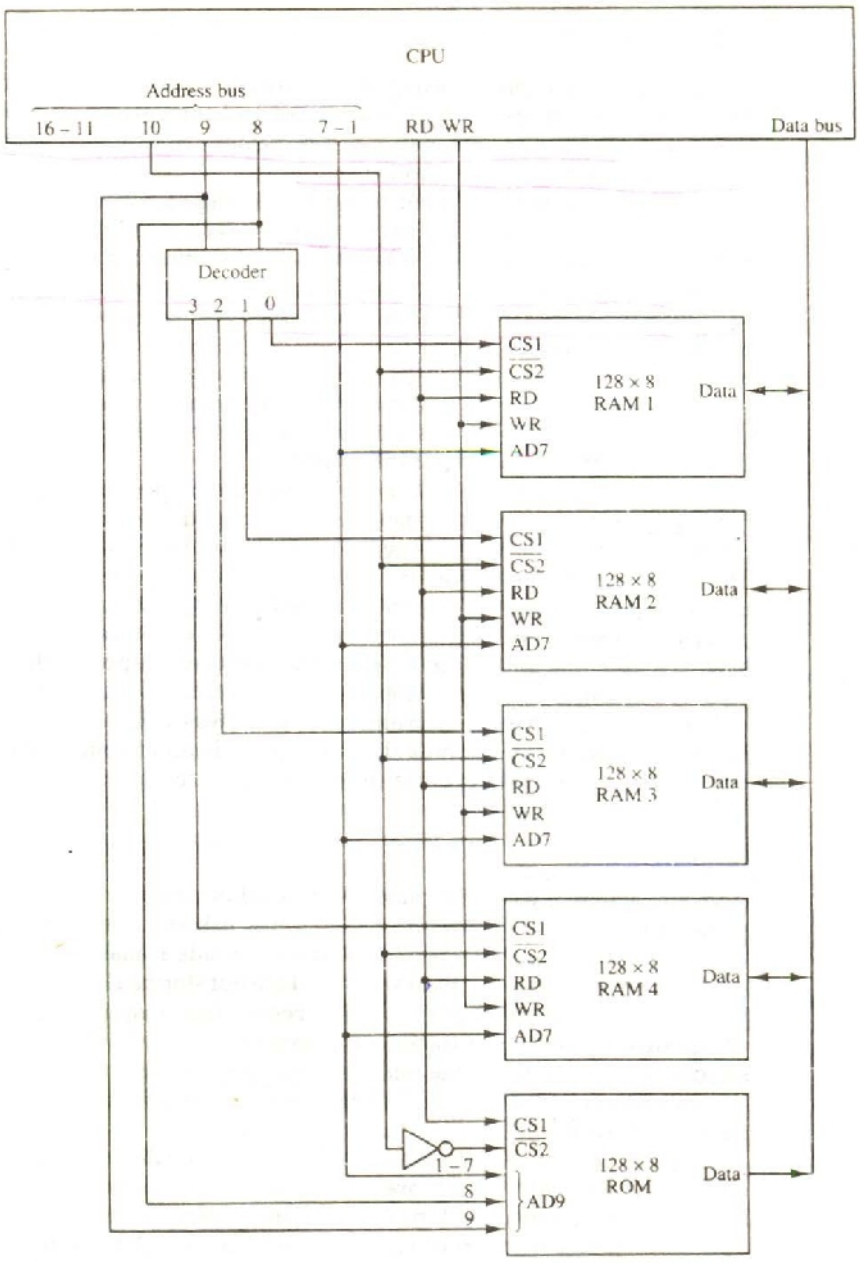


Figure 7-4 Memory connection to the CPU.

Permanent timing tracks are used in disks to synchronize the bits and recognize the sectors. A disk system is addressed by address bits that specify the disk number, the disk surface, the sector number and the track within the sector. After the read/write heads are positioned in the specified track, the system has to wait until the rotating disk reaches the specified sector under the read/write head. Information transfer is very fast once the beginning of a sector has been reached.

Disks may have multiple heads and simultaneous transfer of bits from several tracks at the same time.

A track in a given sector near the circumference is longer than a track near the center of the disk. If bits are recorded with equal density, some tracks will contain more recorded bits than others. To make all the records in a sector of equal length, some disks use a variable recording density with higher density on tracks near the center than on tracks near the circumference. This equalizes the number of bits on all tracks of a given sector.

Disks that are permanently attached to the unit assembly and cannot be removed by the occasional user are called hard disks. A disk drive with removable disks is called a floppy disk. The disks used with a floppy disk drive are small removable disks made of plastic coated with magnetic recording material. There are two sizes commonly used, with diameters of 5.25 and 3.5 inches. The 3.5-inch disks are smaller and can store more data than can the 5.25-inch disks. Floppy disks are extensively used in personal computers as a medium for distributing software to computer users.

7.3.2 MAGNETIC TAPE

A magnetic tape transport consists of the electrical, mechanical, and electronic components to provide the parts and control mechanism for a magnetic-tape unit. The tape itself is a strip of plastic coated with a magnetic recording medium. Bits are recorded as magnetic spots on the tape along several tracks. Usually, seven or nine bits are recorded simultaneously to form a character together with a parity bit. Read/write heads are mounted one in each track so that data can be recorded and read as a sequence of characters.

Magnetic tape units can be stopped, started to move forward or in reverse, or can be rewound. However, they cannot be started or stopped fast enough between individual characters. For this reason, information is recorded in blocks referred to as records. Gaps of unrecorded tape are inserted between records where the tape can be stopped. The tape starts moving while in a gap and attains its constant speed by the time it reaches the next record. Each record on tape has an identification bit pattern at the beginning and end. By reading the bit pattern at the beginning, the tape control identifies the record number. By reading the bit pattern at the end of the record, the control recognizes the beginning of a gap. A tape unit is addressed by specifying the record number of characters in the record. Records may be of fixed or variable length.

7.4 ASSOCIATIVE MEMORY

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

The time required to find an item stored in memory can be reduced considerably if stored data can be identified for access by the content of the data itself rather than by an address. A memory unit accessed by content is called an associative memory or content addressable memory (CAM). This type of memory is accessed simultaneously and in parallel on the basis of data content rather than by specific address or location. When a word is written in an associative

memory, no address is given,. The memory is capable of finding an empty unused location to store the word. When a word is to be read from an associative memory, the content of the word, or part of the word, is specified. The memory locaters all words which match the specified content and marks them for reading.

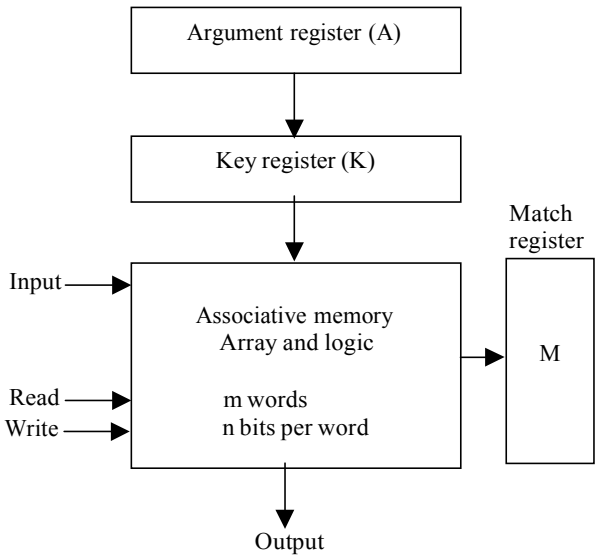
Because of its organization, the associative memory is uniquely suited to do parallel searches by data association. Moreover, searches can be done on an entire word or on a specific field within a word. An associative memory is more expensive then a random access memory because each cell must have storage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

7.4.1 HARDWARE ORGANIZATION

The block diagram of an associative memory is shown in Fig. 7-6. It consists of a memory array and logic from words with n bits per word. The argument register A and key register K each have n bits, one for each bit of a word. The match register M has m bits, one for each memory word. Each word in memory is compared in parallel with the content of the argument register. The words that match the bits of the argument register set a corresponding bit in the match register. After the matching process, those bits in the match register that have been set indicate the fact that their corresponding words have been matched. Reading is accomplished by a sequential access to memory for those words whose corresponding bits in the match register have been set.

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

Figure 7-6 Block diagram of associative memory



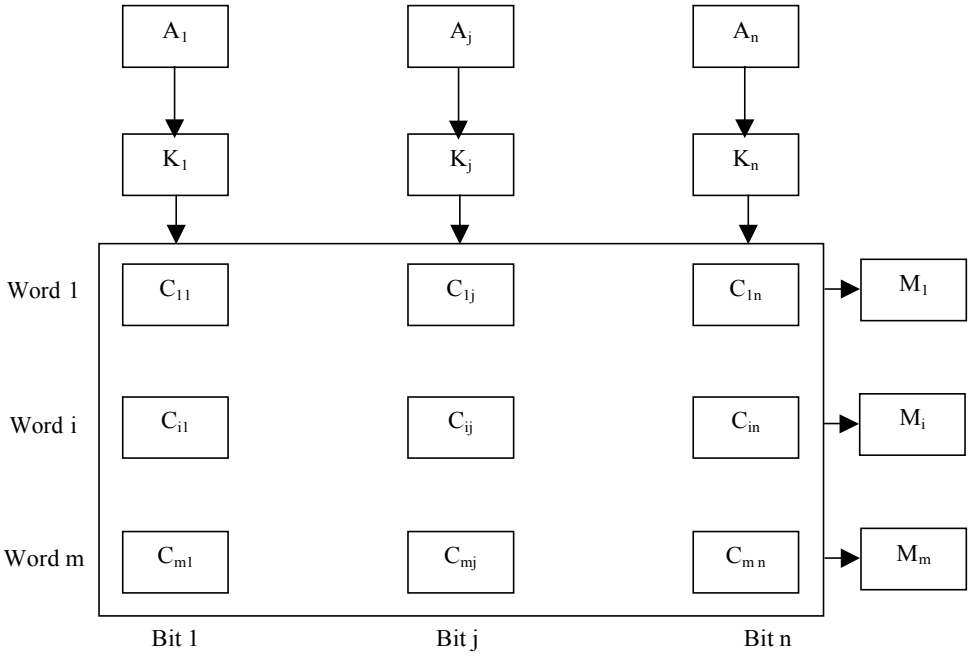
To illustrate with a numerical example, suppose that the argument register A and the key register K have the bit configuration shown below. Only the three leftmost bits of A are compared with memory words because K has 1's in these positions.

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

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

The relation between the memory array and external registers in an associative memory is shown in Fig. 7-7. The cells in the array are marked by the letter C with two subscripts. The first subscript gives the word number and the second specifies the bit position in the word. Thus cell C_{ij} is the cell for bit j in word i. A bit A_j in the argument register is compared with all the bits in column j of the array provided that $K_j = 1$. This is done for all columns j = 1, 2,…,n. If a match occurs between all the unmasked bits of the argument and the bits in word i, the corresponding bit M_i in the match register is set to 1. If one or more unmasked bits of the argument and the word do not match, M_i is cleared to 0.

Figure 7-7 Associative memory of m word, n cells per word



The internal organization of a typical cell C_{ij} is shown in Fig. 7-8. It consists of a flip-flop storage element F_{ij} and the circuits for reading, writing, and matching the cell. The input bit is transferred into the storage cell during a write operation. The bit stored is read out during a read operation. The match logic compares the content of the storage cell with the corresponding unmasked bit of the argument and provides an output for the decision logic that sets the bit in M_i.

7.4.2 MATCH LOGIC

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

$$x_j = A_j F_{ij} + A_j' F_{ij}'$$

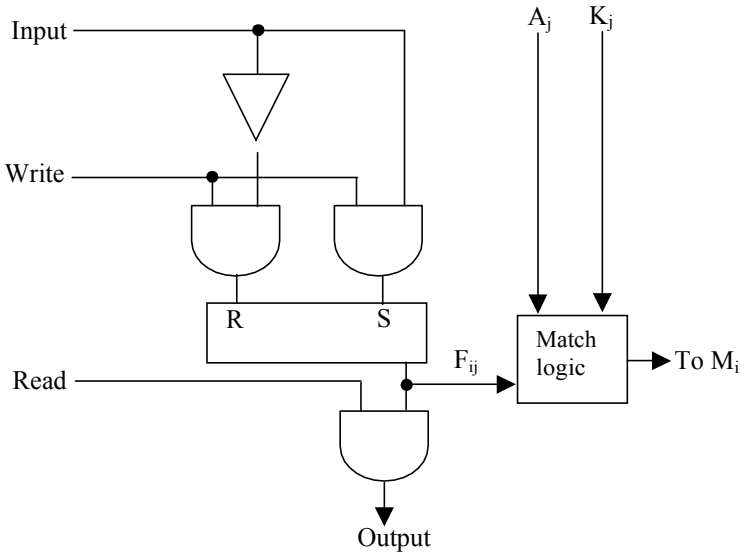
where $x_j = 1$ if the pair of bits in position j are equal; otherwise, $x_j = 0$.

For a word i to be equal to the argument in A we must have all x_j variables equal to 1. This is the condition for setting the corresponding match bit M_i to 1. The Boolean function for this condition is

$$M_i = x_1 x_2 x_3 \ldots x_n$$

and constitutes the AND operation of all pairs of matched bits in a word.

Figure 7-8 One cell of associative memory.



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

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

When $K_j = 1$, we have $K_j' = 0$ and $x_j + 0 = x_j$. When $K_j = 0$, then $K_j' = 1$ $x_j + 1 = 1$. A term ($x_j + K_j'$) will be in the 1 state if its pair of bitsis not compared. This is necessary because each term is ANDed with all other terms so that an output of 1 will have no effect. The comparison of the bits has an effect only when $K_j = 1$.

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

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

Each term in the expression will be equal to 1 if its corresponding $K'_j = 0$. if $K_j = 1$, the term will be either 0 or 1 depending on the value of x_j. A match will occur and M_i will be equal to 1 if all terms are equal to 1.

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

$$M_i = \prod_{j=1}^{n} (A_j F_{ij} + A'_j F'_j + K'_j)$$

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

The circuit for catching one word is shown in Fig. 7-9. Each cell requires two AND gates and one OR gate. The inverters for A_j and K_j are needed once for each column and are used for all bits in the column. The output of all OR gates in the cells of the same word go to the input of a common AND gate to generate the match signal for M_i. M_i will be logic 1 if a catch occurs and 0 if no match occurs. Note that if the key register contains all 0's, output M_i will be a 1 irrespective of the value of A or the word. This occurrence must be avoided during normal operation.

7.4.3 READ OPERATION

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

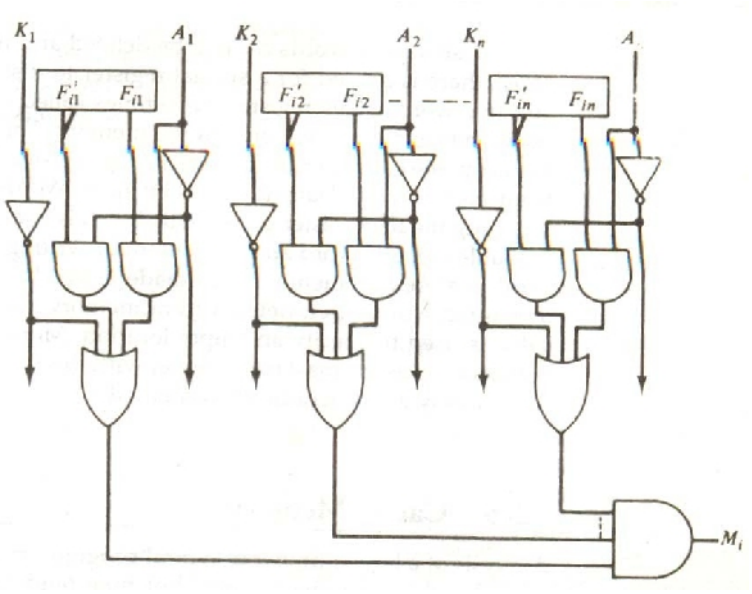


Figure 7-9 Match logic for one word of associative memory

In most applications, the associative memory stores a table with no two identical items under a given key. In this case, only one word may match the unmasked argument field. By connecting output M_i directly to the read line in the same word position (instead of the M register), the content of the matched word will be presented automatically at the output lines and no special read command signal is needed. Furthermore, if we exclude words having a zero content, an all-zero output will indicate that no match occurred and that the searched item is not available in memory.

7.4.4 WRITE OPERATION

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

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

7-5 CACHE MEMORY

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

If the active portions of the program and data are placed in a fast small memory, the average memory access time can be reduced, thus reducing the total execution time of the program. Such a fast small memory is referred to as a cache memory. It is placed between the

CPU and main memory as illustrated in Fig. 7-10. The cache memory access time is less than the access time of main memory by a factor of 5 to 10. The cache is the fastest component in the memory hierarchy and approaches the speed of CPU components.

The fundamental idea of cache organization is that by keeping the most frequently accessed instructions and data in the fast cache memory, the average memory access time will approach the access time of the cache. Although the cache is only a small fraction of the size of main memory, a large fraction of memory requests will be found in the fast cache memory because of the locality of reference property of programs.

The basic operation of the cache is as follows. When the CPU needs to access memory, the cache is examined. If the word is found in the cache, it is read from the fast memory. If the word addressed by the CPU is not found in the cache, the main memory is accessed to read the word. A block of words containing the one just accessed is then transferred from main memory to cache memory. The block size may vary from one word (the one just accessed) to about 16 words adjacent to the one just accessed. In this manner, some data are transferred to cache so that future references to memory find the required words in the fast cache memory.

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

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

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

1. Associative mapping
2. Direct mapping
3. Set-associative mapping

To helping the discussion of these three mapping procedures we will use a specific example of a memory organization as shown in Fig. 7-10. The main memory can store 32K words of 12 bits each. The cache is capable of storing 512 of these words at any given time. For every word stored in cache, there is a duplicate copy in main memory.

The CPU communicates with both memories. It first sends a 15-bit address to cache. If there is a hit, the CPU accepts the 12-bit data from cache. If there is a miss, the CPU reads the word from main memory and the word is then transferred to cache.

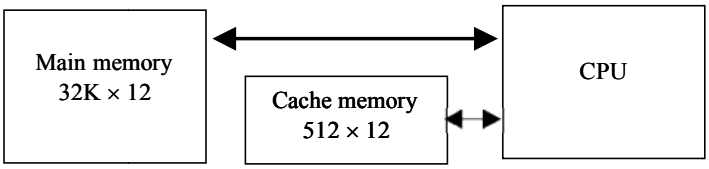
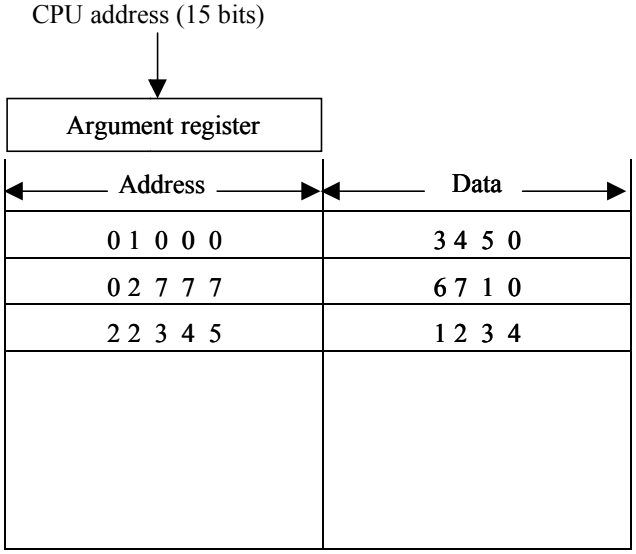


Figure 7-10 Example of cache memory

7.5.1 ASSOCIATIVE MAPPING

The fasters and most flexible cache organization uses an associative memory. This organization is illustrated in Fig. 7-11. The associative memory stores both the address and content (data) of the memory word. This permits any location in cache to store any word from main memory. The diagram shows three words presently stored in the cache. The address value of 15 bits is shown as a five-digit octal number and its corresponding 12-bit word is shown as a four-digit octal number. A CPU address of 15 bits is placed in the argument register and the associative memory is searched for a matching address. If the address is found, the corresponding 12-bit data is read

Figure 7-11 Associative mapping cache (all numbers in octal)



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

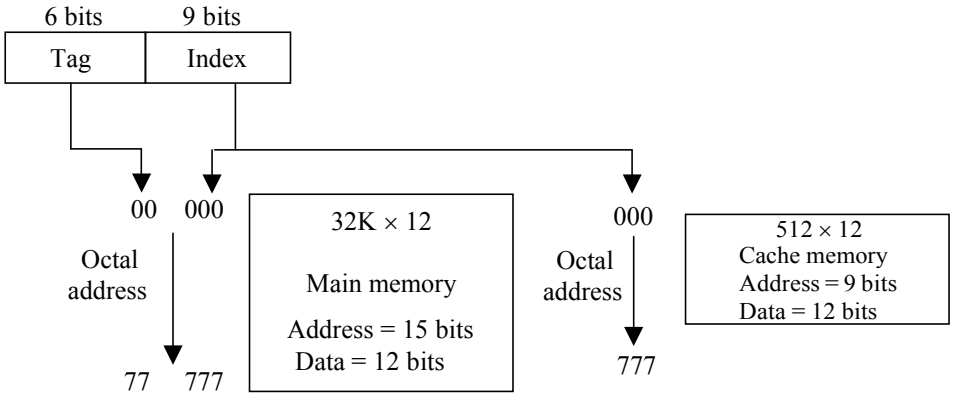
7.5.2 DIRECT MAPPING

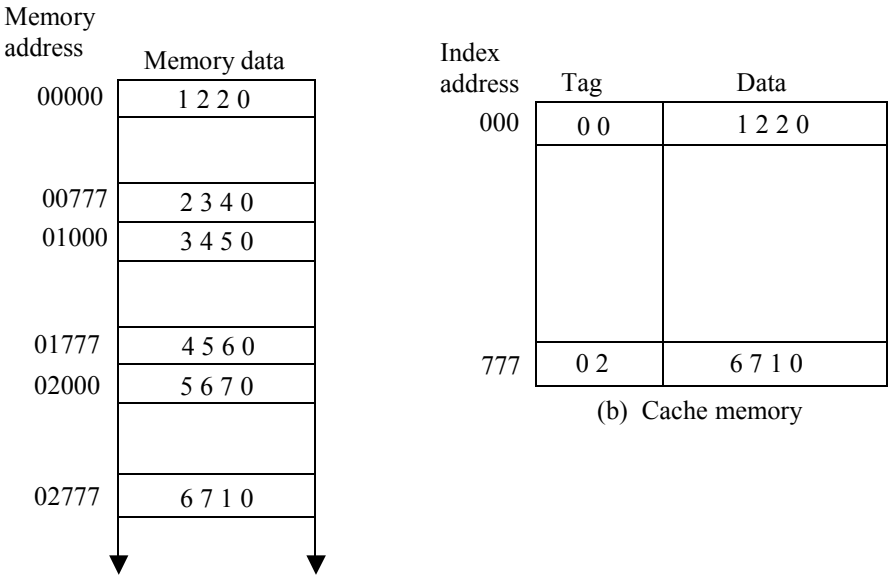
Associative memories are expensive compared to random-access memories because of the added logic associated with each cell. The possibility of using a random-access memory for the cache is investigated in Fig. 7-12. The CPU address of 15 bits is divided into two fields. The

nine least significant bits constitute the index field and the remaining six bits form the tag and the index bits. The number of bits in the index field is equal to the number of address bits required to access the cache memory.

In the general case, there are 2^k words in cache memory and 2^n words in main memory. The n-bit memory address is divided into two fields: k bits for the index field and n – k bits for the tag field. The direct mapping cache organization uses the n-bit address to access the main memory and the k-bit index to access the cache. The internal organization of the words in the cache memory is as shown in Fig. 7–13(b). Each word in cache consists of the data word and its associated tag. When a new word is first brought into the cache, the tag bits are stored alongside the data bits. When the CPU generates a memory request, the index field is used for the address to access the cache.

Figure 7-12 Addressing relationships between main and cache memories.





(a) Main memory

(b) Cache memory

Figure 7-13 Direct mapping cache organization

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

To see how the direct-mapping organization operates, consider the numerical example shown in Fig. 7-13. The word at address zero is presently stored in the cache (index = 000, tag = 00, data = 1220). Suppose that the CPU now wants to access the word at address 02000. The index address is 000, so it is sued to access the cache. The two tags are then compared. The cache tag is 00 but the address tag is 02, which does not produce a match. Therefore, the main memory is accessed and the data word 5670 is transferred to the CPU. The cache word at index address 000 is then replaced with a tag of 02 and data of 5670.

The direct-mapping example just described uses a block size of one word. The same organization but using a block size of 8 words is shown in Fig. 7-14. The index

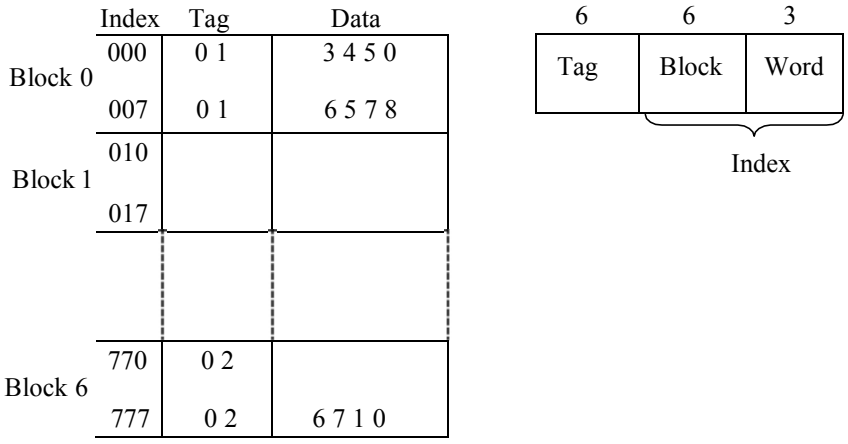


Fig 7.14

field is now divided into two parts: the block field and the word field. In a 512-word cache there are 64 block of 8 words each, since $64 \times 8 = 512$. The block number is specified with a 6-bit field and the word within the block is specified with a 3-bit field. The tag field stored within the cache is common to all eight words of the same block. Every time a miss occurs, an entire block of eight words must be transferred form main memory to cache memory. Although this takes extra time, the hit ratio will most likely improve with a larger block size because of the sequential nature of computer programs.

7.5.3 SET-ASSOCIATIVE MAPPING

It was mentioned previously that the disadvantage of direct mapping is that two words with the same index in their address but with different tag values cannot reside in cache memory at the same time. A third type of cache organization, called set-associative mapping, is an improvement over the direct-mapping organization in that each word of cache can store two or more words of

memory under the same index address. Each data word is stored together with its tag and the number of tag-data items in one word of cache is said to form a set. An example of a set-associative cache organization for a set size of two is shown in Fig. 7-15. Each index address refers to two data words and their associated tags. Each tag requires six bits and each data word has 12 bits, so the word length is 2(6 + 12) = 36 bits. An index address of nine bits can accommodate 512 words. Thus the size of cache memory is 512 × 36. It can accommodate 1024 words of main memory since each word of cache contains two data words. In general, a set-associative cache of set size k will accommodate k words of main memory in each word of cache.

Index	Tag	Data		Tag	Data
000	0 1	3 4 5 0		0 2	5 6 7 0
777		6 7 1 0		0 0	2 3 4 0

Figure 7-15 Two-way set-associative mapping cache.

The octal numbers listed in Fig. 7-15 are with reference to the main memory content illustrated in Fig. 7-13(a). The words stored at addresses 01000 and 02000 of main memory are stored in cache memory at index address 000. Similarly, the words at addresses 02777 and 00777 are stored in cache at index address 777. When the CPU generates a memory request, the index value of the address is used to access the cache. The tag field of the CPU address is then compared with both tags in the cache to determine if a catch occurs. The comparison logic is done by an associative search of the tags in the set similar to an associative memory search: thus the name "set-associative". The hit ratio will improve as the set size increases because more words with the same index but different tage can reside in cache. However, an increase in the set size increases the number of bit s in words of cache and requires more complex comparison logic.

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

7.5.4 WRITING INTO CACHE

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

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

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

7.5.5 CACHE INITIALIZATION

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

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

7.6 VIRTUAL MEMORY

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

7.6.1 ADRESS SPACE AND MEMORY SPACE

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

As an illustration, consider a computer with a main-memory capacity of 32K words (K = 1024). Fifteen bits are needed to specify a physical address in memory since $32K = 2^{15}$. Suppose that the computer has available auxiliary memory for storing $2^{20} = 1024K$ words. Thus auxiliary memory has a capacity for storing information equivalent to the capacity of 32 main memories. Denoting the address space by N and the memory space by M, we then have for this example N = 1024K and M = 32K.

In a multiprogram computer system, programs and data are transferred to and from auxiliary memory and main memory based on demands imposed by the CPU. Suppose that program 1 is currently being executed in the CPU. Program 1 and a portion of its associated data re moved from auxiliary memory into main memory as shown in Fig. 7-16. Portions of programs and data need not be in contiguous locations in memory since information is being moved in and out, and empty spaces may be available in scattered locations in memory.

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

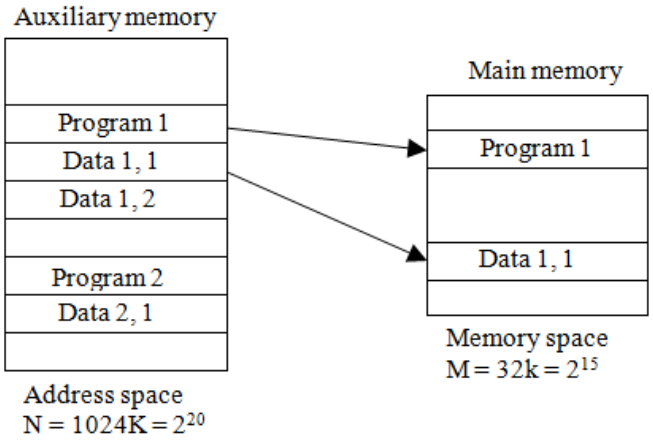
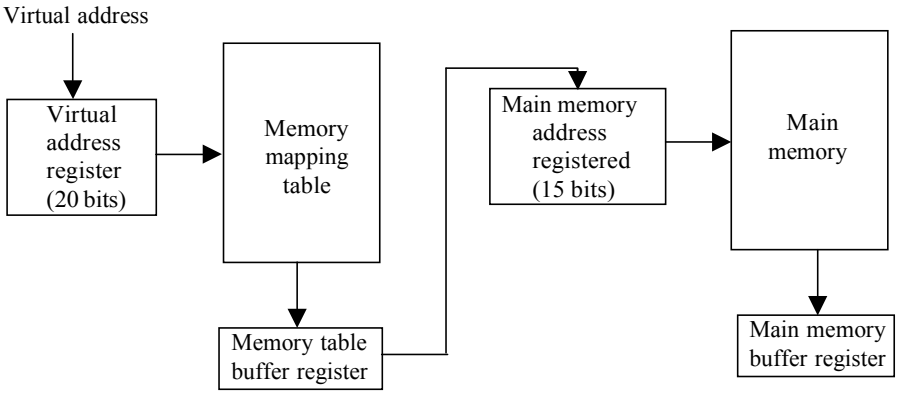


Figure 7-16 Relation between address and memory space in a virtualmemory system.

that for efficient transfers, auxiliary storage moves an entire record to the main memory). A table is then needed, as shown in Fig. 7-17, to map a virtual address of 20 bits to a physical address of 15 bits. The mapping is a dynamic operation, which means that every address is translated immediately as a word is referenced by CPU.

The mapping table may be stored in a separate memory as shown in Fig. 7-17 or in main memory. In the first case, an additional memory unit is required as well as one extra memory access time. In the second case, the table

Figure 7-17 Memory table for mapping a virtual address.



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

7.6.2 ADDRESS MAPPING USING PAGES

The table implementation of the address mapping is simplified if the information in the address space and the memory space are each divided into groups of fixed size. The physical memory is broken down into groups of equal size called blocks, which may range from 64 to 4096 words each. The term page refers to groups of address space of the same size.

For example,if a page or block consists of 1K words, then, using the previous example, address space is divided into 1024 pages and main memory is divided into 32 blocks. Although both a page and a block are split into groups of 1K words, a page refers to the organization of address space, while a block refers to the organization of memory space. The programs are also considered to be split into pages. Portions of programs are moved from auxiliary memory to main memory in records equal to the size of a page. The term "page frame" is sometimes used to denote a block.

Consider a computer with an address space of 8K and a memory space of 4K. If we split each into groups of 1K words we obtain eight pages and four blocks as shown in Fig. 7-18. At any given time, up to four pages of address space may reside in main memory in any one of the four blocks.

The mapping from address space to memory space is facilitated if each virtual address is considered to be represented by two numbers: a page number address and a line within the page. In a computer with 2^p words per page, p bits are used to specify a line address and the remaining high-order bits of the virtual address specify the page number. In the example of Fig. 7-18, a virtual address has 13 bits. Since each page consists of 2^{10} = 1024 words, the high-order three bits of a virtual address will specify one of the eight pages and the low-order 10 bits give the line address within the page. Note that the line address in address space and memory space is the same; the only mapping required is from a page number to a block number.

The organization of the memory mapping table in a paged system is shown in Fig. 7-19. The memory-page table consists of eight words, one for each page. The address in the page table denotes the page number and the content of the word gives the block number where that page is stored in main memory. The table shows that pages 1, 2, 5 and 6 are now available in main memory in blocks 3, 0, 1, and 2, respectively. A presence bit in each location indicates whether the page has been transferred from auxiliary memory into main memory. A 0 in the presence bit indicates that this page is not available in main memory. The CPU references a word in memory with a virtual address of 13 bits. The three high-order bits of the virtual address specify a page number and also an address for the memory-page table. The content of the word in the memory

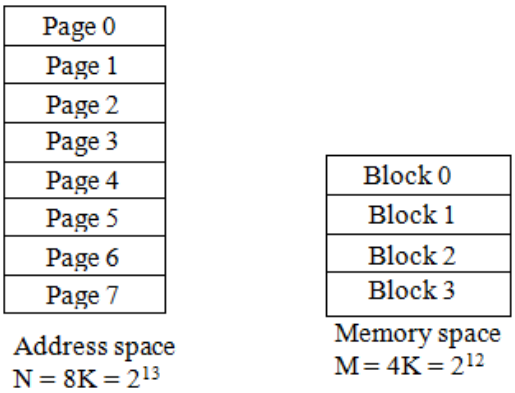


Figure 7-18 Address space and memory space split into groups of 1K words.

page table at the page number address is read out into the memory table buffer register. If the presence bit is a 1, the block number thus read is transferred to the two high-order bits of the main memory address register. The line number from the virtual address is transferred into the 10 low-order bits of the memory address register. A read signal to main memory transfers the content of

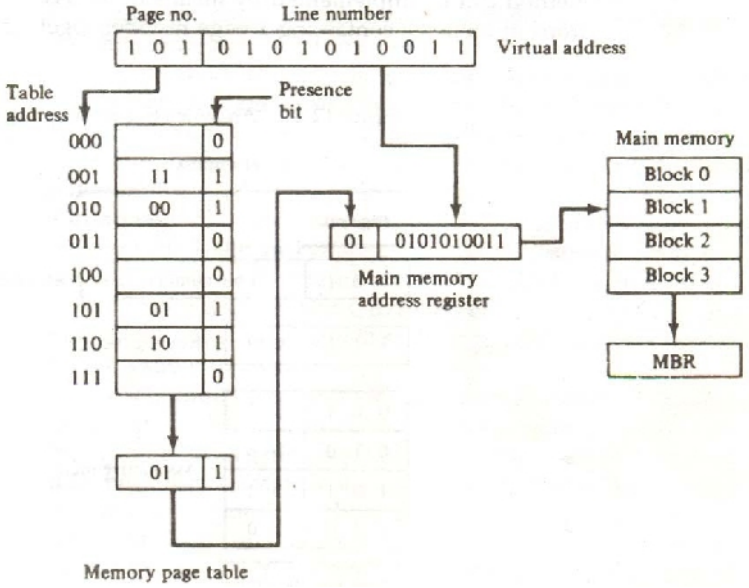


Figure 7-19 Memory table in a paged system.

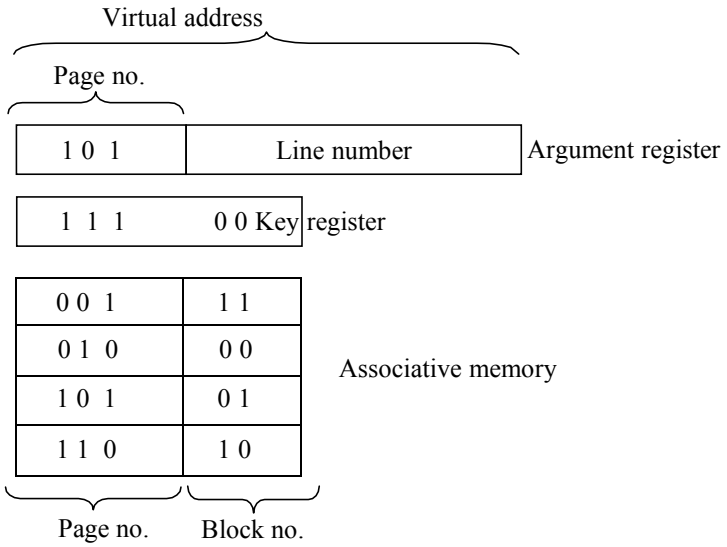
the word to the main memory buffer register ready to be used by the CPU. If the presence bit in the word read from the page table is 0, it signifies that the content of the word referenced by the virtual address does not reside in main memory. A call to the operating system is then generated to fetch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

7.6.3 ASSOCIATIVE MEMORY PAGE TABLE

A random-access memory page table is inefficient with respect to storage utilization. In the example of Fig. 7-19 we observe that eight words of memory are needed, one for each page, but at least four words will always be marked empty because main memory cannot accommodate more than four blocks. In general, system with n pages and m blocks would require a memory- page table of n locations of which up to m blocks will be marked with block numbers and all others will be empty. As a second numerical example, consider an address space of 1024K words and memory space of 32K words. If each page or block contains 1K words, the number of pagesis 1024 and the number of blocks 32. The capacity of the memory-page table must be 1024 words and only 32 locations may have a presence bit equal to 1. At any given time, at least 992 locations will be empty and not in use.

A more efficient way to organize the page table would be to construct it with a number of words equal to the number of blocks in main memory. In this way the size of the memory is reduced and each location is fully utilized. This method can be implemented by means of an associative memory with each word in memory containing a page number together with its corresponding

Figure 7-20 An associative memory page table.



block number. The page field in each word is compared with the page number in the virtual address. If a match occurs, the word is read from memory and its corresponding block number is extracted.

Consider again the case of eight pages and four blocks as in the example of Fig. 7-19. We replace the random access memory-page table with an associative memory of four words as shown in Fig. 7-20. Each entry in the associative memory array consists of two fields. The first three bits specify a field for storing the page number. The last two bits constitute a field for storing the block number. The virtual address is placed in the argument register. The page number bits in the argument are compared with all page numbers in the page field of the associative memory. If the page number is found, the 5-bit word is read out from memory. The corresponding block number, being in the same word, is transferred to the main memory address register. If no match occurs, a call to the operating system is generated to bring the required page from auxiliary memory.

7.6.4 PAGE REPLACEMENT

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

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

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

immediate future.

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

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

7.7 CHECK YOUR PROGRESS

1. The chip by which both the operation of read and write is performed _____ .
2. If a RAM chip has n address input lines then it can access memory locations upto _____ .
3. _____ control signals are selected for read and write operations in a RAM.
4. _____ is the permanent memory built into your computer called.
5. Magnetic tape is not practical for applications where data must be quickly recalled because tape is _____.

7.8 SUMMARY

1. The memory unit is an essential component in any digital computer since it is needed for storing programs and data.
2. The total memory capacity of a computer can be visualized as being a hierarchy ofcomponents.
3. A special very-high speed memory called a cache is sometimes used to increase the speed of processing by making current programs and data available to the CPU at a rapid rate.
4. Auxiliary and cache memories are used for different purposes. The cache holds those parts of the program and data that are most heavily used, while the auxiliary memory holds those parts that are not presently used by the CPU.
5. The main memory is the central storage unit in a computer system. It is a relatively large and fast memory used to store programs and data during the computer operation.
6. A RAM chip is better suited for communication with the CPU if it has one or morecontrol inputs that select the chip only when needed.
7. Most of the main memory in a general-purpose computer is made up of RAM integratedcircuit chips, but a portion of the memory may be constructed with ROM chips.
8. RAM and ROM chips are connected to a CPU through the data and address buses. The low-order lines in the address bus select the byte within the chips and other lines in the address bus select a particular chip through its chip select inputs.
9. The designer of a computer system must calculate the amount of memory required for the particular application and assign it to either RAM or ROM.

10. The most common auxiliary memory devices used in computer systems are magnetic disks and tapes.
11. A magnetic disk is a circular plate constructed of metal or plastic coated with magnetized material.
12. Many data-processing applications require the search of items in a table stored in memory. An assembler program searches the symbol address table in order to extract the symbol's binary equivalent.
13. The match logic for each word can be derived from the comparison algorithm for two binary numbers.
14. If more than one word in memory matches the unmasked argument field, all the matched words will have 1's in the corresponding bit position of the catch register.
15. An associative memory must have a write capability for storing the information to be searched.
16. Analysis of a large number of typical programs has shown that the references, to memoryat any given interval of time tend to be confined within a few localized areas in memory.
17. The fasters and most flexible cache organization uses an associative memory.

18. It was mentioned previously that the disadvantage of direct mapping is that two words with the same index in their address but with different tag values cannot reside in cache memory at the same time.
19. A virtual memory system is a combination of hardware and software techniques. The memory management software system handles all the software operations for the efficient utilization of memory space.
20. Two of the most common replacement algorithms used are the first-in first-out (FIFO) and the least recently used (LRU).

7.9 KEYWORDS

1. **Magnetic Disk** is a storage device that uses a magnetization process to write, rewrite and access data. It is covered with a magnetic coating and stores data in the form of tracks, spots and sectors.
2. **Magnetic Tape** is a medium for magnetic recording, made of a thin, magnetizable coating on a long, narrow strip of plastic film
3. **Optical Memory** technology, a laser beam encodes digital data onto an optical, or laser, disk in the form of tiny pits arranged in concentric tracks on the disk's surface.
4. **Access Time** is how long it takes for a character in RAM to be transferred to or from the CPU.

7.10 SELF-ASSESSMENT TEST

1. Explain the memory hierarchy in the computer systems.
2. Explain different types of memory available with us.
3. What is memory address mapping? Explain the concept with the help of ROM chips.
4. How we can connect a memory with any CPU? Explain.
5. What is the concept of Associative memory? Explain.
6. Explain the significance on Match Logic.
7. What is cache memory? How it is fast as compared to conventional memory?
8. Explain the concept of virtual memory.
9. What is the concept of page replacement?

7.11 ANSWERS TO CHECK YOUR PROGRESS

1. RAM
2. 2^n
3. Read and write
4. ROM
5. A sequential-access medium

7.12 REFERENCES / SUGGESTED READINGS

1. Computer Organization and Architecture, Rajaram & Radhakrishan, PHI.
2. Computer Organization & Architecture: Designing for Performance, Stalling, PHI.
3. Computer Organization and Design, Pal Choudhary, PHI.
4. Computer Systems Organization & Architecture, Carpenelli, Pearson Education.
5. Computer Organization and Architecture, Stalling, Pearson Education.
6. Computer System Architecture, Morris Mano, PHI.
7. Computer Architecture and Organization, McGraw Hill Company, New Delhi. J.P. Hayes.

SUBJECT: COMPUTER ORGANIZATION	
COURSE CODE: CSL 612	**AUTHOR: DR. MANOJ DUHAN**
LESSON NO. 8	

INPUT-OUTPUT ORGANIZATION

REVISED / UPDATED SLM BY NEERAJ VERMA

STRUCTURE

8.0 LEARNING OBJECTIVES

After reading this unit, you should be able to:

1. Understand how a processor accesses and addresses the I/O devices.
2. Identify some most common I/O devices
3. Understand the different techniques implemented for I/O communications.
4. Understand interrupts, and be able to compare the interrupts and polling with the programmed I/O
5. Be familiar with interrupt hardware, interrupt handling and control and processing of the multiple device requests
6. Be familiar with bus arbitration hardware, daisy chaining, and polling methods of a bus controller
7. Understand the synchronous and asynchronous data transfer on the buses, bus control, and processing of the multiple devices requests to get access to a bus

8.1 INTRODUCTION

The input-output subsystem of a computer, referred to as I/O, provides an efficient mode of communication between the central system and the outside environment. Programs and data must be entered into computer memory for processing and results obtained from computations must be recorded or displayed for the user. A computer serves no useful purpose without the ability to receive information from an outside source and to transmit results in a meaningful form.

The most familiar means of entering information into a computer is through a typewriter like keyboard that allows a person to enter alphanumeric information directly. Every time a key is depressed, the terminal sends a binary coded character to the computer. The fastest possible speed for entering information this way depends on the person's typing speed. On the other hand, the central processing unit is an extremely fast device capable of performing operation this way depends on the person's typing speed. On the other hand, the central processing unit is an extremely fast device capable of performing operations at very high speed. When input information is transferred to the processor via a slow keyboard, the processor will be idle most of the time while waiting for the

information to arrive. To use a computer efficiently, a large amount of programs and data must be prepared in advance and transmitted into a storage medium such as magnetic tapes or disks. The information in the disk is then transferred into computer memory at a rapid rate. Results of programs are also transferred into a high-speed storage, such as disks, from which they can be transferred later into a printer to provide a printed output of results.

Devices that are under the direct control of the computer are said to be connected on-line. These devices are designed to read information into or out of the memory unit upon command from the CPU and are considered to be part of the total computer system. Input or output devices attached to the computer are also called peripherals. Among the most common peripherals are keyboards, display units, and printers. Peripherals that provide auxiliary storage for the system are magnetic disks and tapes. Peripherals are electro-mechanical and electromagnetic devices of some complexity.

Video monitors are the most commonly used peripherals. They consist of a keyboard as the input device and a display unit as the output device. There are different types of video monitors, but the most popular use a cathode ray tube (CRT). The CRT contains an electronic gun that sends an electronic beam to a phosphorescent screen in front of the tube. The beam can be deflected horizontally and vertically. To produce a pattern on the screen, a grid inside the CRT receives a variable voltage that causes the beam to hit the screen and make it glow at selected spots. Horizontal and vertical signals deflect the beam and make it sweep across the tube, causing the visual pattern to appear on the screen. A characteristic feature of display devices is a cursor that marks the position in the screen where the next character will be inserted. The cursor can be moved to any position in the screen, to a single character, the beginning of a word, or to any line. Edit dyes add or delete information based on the cursor position. The display terminal can operate in a single-character mode where all character entered on the screen through the keyboard are transmitted to the computer simultaneously. In the block mode, the edited text is first stored in a logical memory inside the terminal. The text is transferred to the computer as a block of data.

Printers provide a permanent record on paper of computer output data or text. There are three basic types of character printers: daisywheel, dot matrix, and laser printers. The daisywheel printer contains a wheel with the characters placed along the circumference. To print a character, the wheel rotates to the proper position and an energized magnet then presses the letter against the ribbon. The dot matrix printer contains a set of dots

along the printing mechanism. For example, a 5 × 7 dot matrix printer that prints 80 characters per line has seven horizontal lines, each consisting of 5 × 7 dot matrix printer that prints 80 characters per line has seven horizontal lines, each consisting of 5 × 80 = 400 dots. Each dot can be printed or not, depending on the specific characters that are printed on the line. The laser printer uses a rotating photographic drum that is used to imprint the character images. The pattern is then transferred onto paper in the same manner as a copying machine.

Magnetic tapes are used mostly for storing files of data: for example, a company's payroll record. Access is sequential and consists of records that can be accessed one after other as the tape moves along a stationary read-write mechanism. It is one of the cheapest and slowest methods for storage and has the advantage that tapes can be recovered when not in use. Magnetic disks have high-speed rotational surfaces coated with magnetic material. Access is achieved by moving a read-write mechanism to a track in the magnetized surface. Disks are used mostly for bulk storage of programs and data.

Other input and output devices encountered in computer systems are digital incremental plotters, optical and magnetic character readers, analog-to-digital converters, and various data acquisition equipment. Not all input comes from people, and not all output is intended for people.

Computers are used to control various processes in real time, such as machine tooling, assembly line procedures, and chemical and industrial processes. For such applications, a method must be provided for sensing status condition in the process and sending control signals to the process being controlled.

The input-output organization of a computer is a function of the size of the computer and the devices connected to it. The difference between a small and a large system is mostly dependent on the amount of hardware the computer has available for communicating with peripheral units and the number of peripherals connected to the system. Since each peripheral behaves differently from any other, it would be prohibitive to dwell on the detailed interconnections needed between the computer and each peripheral. Certain techniques common to most peripherals are presented in this chapter.

8.1.1 ASCII ALPHANUMERIC CHARACTERS

Input and output devices that communicate with people and the computer are usually involved in the transfer of alphanumeric information to and from the device and the computer is ASCII (American Standard Code for Information Interchange). It uses seven

bits to code 128 characters as shown in Table 8-1. The seven bits of the code are designated by b_1 through b_7 being the most significant bit. The letter A, for example, is represented in ASCII as 1000001 (column 100, row 0001). The ASCII code contains 94 characters that can be printed and 34 nonprinting characters used for various control functions. The printing characters consist of the 26 uppercase letters A through Z, the 26 lowercase letters, the 10 numerals 0 through 9, and 32 special printable characters such as %, *, and $.

Table 8-1 American Standard Code for Information Interchange (ASCII)

	$b_7 b_6 b_5$							
$b_4 b_3 b_2 b_1$	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	‘	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	’	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	¦
1101	CR	GS	−	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	—	o	DEL

Control characters

NUL	Null	DLE	Data link escape
SOH	Start of heading	DC1	Device control 1
STX	Start of text	DC2	Device control 2
ETX	End of text	DC3	Device control 3
EOT	End of transmission	DC4	Device control 4
ENQ	Enquiry	NAK	Negative acknowledge
ACK	Acknowledge	SYN	Synchronous idle
BEL	Bell	ETB	End of transmission block
BS	Backspace	CAN	Cancel
HT	Horizontal tab	EM	End of medium
LF	Line feed	SUB	Substitute
VT	Vertical tab	ESC	Escape
FF	Form feed	FS	File separator
CR	Carriage return	GS	Group separator
SO	Shift out	RS	Record separator
SI	Shift in	US	Unit separator
SP	Space	DEL	Delete

The 34 control characters are designated in the ASCII table with abbreviated names. They are listed again below the table with their functional names. The control characters are used for routing data and arranging the printed text into a prescribed format. There are three types of control characters: format effectors, information separators, and communication control characters. Format effectors are characters that control the layout of printing. They include the familiar typewriter controls, such as backspace (BS), horizontal tabulation (HT), and carriage return (CR). Information separators are used to separate the data into divisions like paragraphs and pages. They include characters such as record separator (RS) and file separator (FS). The communication control characters are useful during the transmission of text between remote terminals. Examples of communication control characters are STX (start of text) and ETX (end of text), which are used to frame a text message when transmitted through a communication medium.

ASCII is a 7 bit code, but most computer manipulate an 8-bit quantity as a single unit called a byte. Therefore, ASCII characters most often are stored one per byte. Therefore, ASCII characters most often are stored one per byte. The extra bit is sometimes used for other purposes, depending on the application. For example, some printers recognize 8-bit ASCII characters with the most significant bit set to 0. Additional 128 8-bit characters with the most significant bit set to 1 are used for other symbols, such as the Greek alphabet or italic type font. When used in data communication, the eighth bit may be employed to indicate the parity of the binary-coded character.

8.2 INPUT-OUTPUT INTERFACE

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

1. Peripherals are electromechanical and electromagnetic devices and their manner of operation is different from the operation of the CPU and memory, which are electronic devices. Therefore, a conversion of signal values may be required.
2. The data transfer rate of peripherals is usually slower than the transfer rate of the CPU, and consequently, a synchronization mechanism may be need.

3. Data codes and formats in peripherals differ from the word format in the CPU and memory.
4. The operating modes of peripherals are different from each other and each must be controlled so as not to disturb the operation of other peripherals connected to the CPU.

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

8.2.1 I/O BUS AND INTERFACE MODULES

A typical communication link between the processor and several peripherals is shown in Figure 8-1. The I/O bus consists of data lines, address lines, and control lines. The magnetic disk, printer, and terminal are employed in practically any general-purpose computer. The magnetic tape is used in some computers for backup storage. Each peripheral device has associated with it an interface unit. Each interface decodes the address and control received from the I/O bus, interprets them for the peripheral, and provides signals for the peripheral controller. It also synchronizes the data flow and supervises the transfer between peripheral and processor. Each peripheral has its own controller that operates the particular electromechanical device. For example, the printer controller controls the paper motion, the print timing, and the selection of printing characters. A controller may be housed separately or may be physically integrated with the peripheral.

The I/O bus from the processor is attached to all peripheral interfaces. To communicate with a particular device, the processor places a device address on the address lines. Each interface attached to the I/O bus contains an address decoder that monitors the address lines. When the interface detects its own address, it activates the path between the bus lines and the device that it controls. All peripherals whose address does not correspond to the address in the bus are disabled their interface.

At the same time that the address is made available in the address lines, the processor provides a function code in the control lines. The interface selected responds to the function code and proceeds to execute it. The function code is referred to as an I/O command and is in essence an instruction that is executed in the interface and its attached

peripheral unit. The interpretation of the command depends on the peripheral that the processor is addressing. There are four types of commands that an interface may receive. They are classified as control, status, status, data output, and data input.

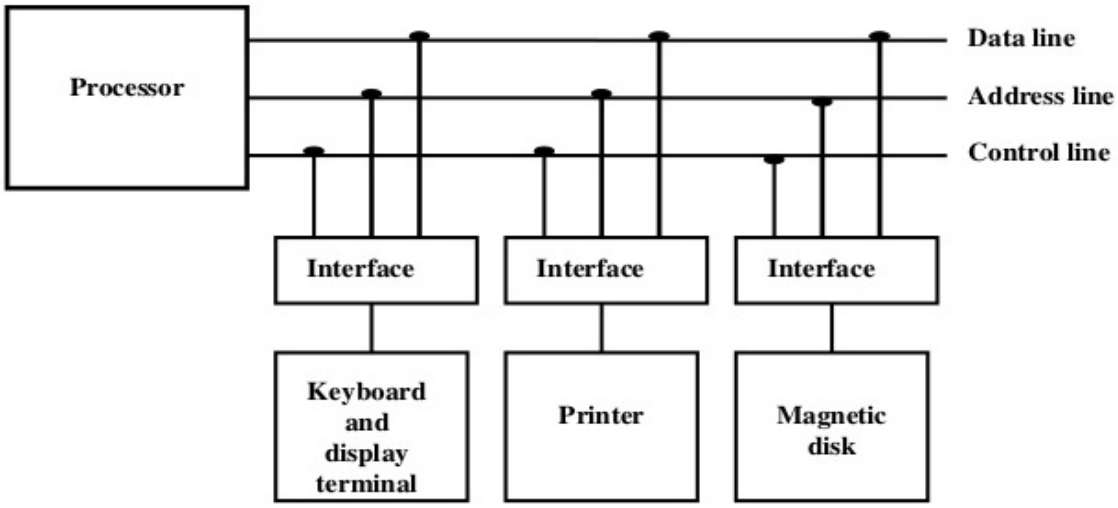


Figure 8.1 Connection of I/O bus to input devices.

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

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

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

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

8.2.2 I/O VERSUS MEMORY BUS

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

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

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

8.2.3 ISOLATED VERSUS MEMORY-MAPPED I/O

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

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

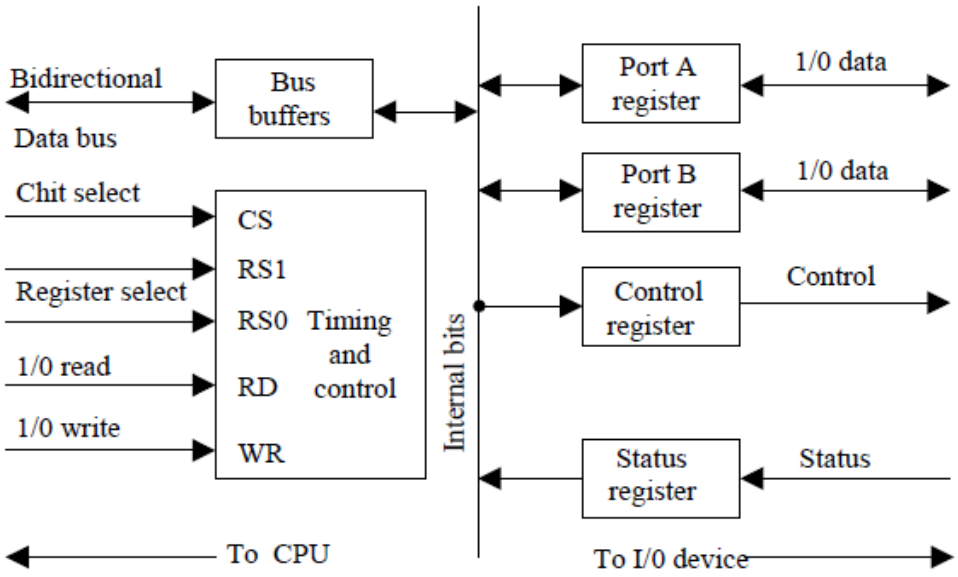
The isolated I/O method isolates memory word and not for an I/O addresses so that memory address values are not affected by interface address assignment since each has its own address space. The other alternative is to use the same address space for both memory and I/O. This is the case in computers that employ only one set of read and write signals and do not distinguish between memory and I/O addresses. This configuration is referred to as memory mapped I/O. The computer treats an interface register as being part of the memory system. The assigned addresses for interface registers cannot be used for memory words, which reduce the memory address range available.

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

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

8.2.4 EXAMPLE OF I/O INTERFACE

An example of an I/O interface unit is shown in block diagram form in Figure 8-2. It consists of two data registers called ports, a control register, a status register, bus buffers, and timing and control circuits. The interface communicates with the CPU through the data bus. The chip select and register select inputs determine the address assigned to the interface. The I/O read and write are two control lines that specify an input or output, respectively. The four registers communicate directly with the I/O device attached to the interface.



CS	RS1	RS0	Register selected
0	×	×	None: data bus in high-impedance
1	0	0	Port A register
1	0	1	Port B register
1	1	0	Control register
1	1	1	Status register

Figure 8.2 Example of I/O interface unit.

The I/O data to and from the device can be transferred into either port A or Port B. The interface may operate with an output device or with an input device, or with a device that requires both input and output. If the interface is connected to a printer, it will only output data, and if it services a character reader, it will only input data. A magnetic disk unit transfers data in both directions but not at the same time, so the interface can use

bidirectional lines. A command is passed to the I/O device by sending a word to the appropriate interface register. In a system like this, the function code in the I/O bus is not needed because control is sent to the control register, status information is received from the status register, and data are transferred to and from ports A and B registers. Thus the transfer of data, control, and status information is always via the common data bus. The distinction between data, control, or status information is determined from the particular register with which the CPU communicates.

The control register receives control information from the CPU. By loading appropriate bits into the control register, the interface and the I/O device attached to it can be placed in a variety of operating modes. For example, port A may be defined as an input port and port B as an output port. A magnetic tape unit may be instructed to rewind the tape or to start the tape moving in the forward direction. The bits in the status register are used for status conditions and for recording errors that may occur during the data transfer. For example, a status bit may indicate that port A has received a new data item from the I/O device. Another bit in the status register may indicate that a parity error has occurred during the transfer.

The interface registers communicate with the CPU through the bidirectional data bus. The address bus selects the interface unit through the chip select and the two register select inputs. A circuit must be provided externally (usually, a decoder) to detect the address assigned to the interface registers. This circuit enables the chip select (CS) input when the interface is selected by the address bus. The two register select inputs RS1 and RS0 are usually connected to the two least significant lines of the lines address bus. These two inputs select one of the four registers in the interface as specified in the table accompanying the diagram. The content of the selected register is transfer into the CPU via the data bus when the I/O read signal is enabled. The CPU transfers binary information into the selected register via the data bus when the I/O write input is enabled.

8.3 ASYNCHRONOUS DATA TRANSFER

The internal operations in a digital system are synchronized by means of clock pulses supplied by a common pulse generator. Clock pulses are applied to all registers within a unit and all data transfers among internal registers occur simultaneously during the occurrence of a clock pulse. Two units, such as a CPU and an I/O interface, are designed

independently of each other. If the registers in the interface share a common clock with the CPU registers, the transfer between the two units is said to be synchronous. In most cases, the internal timing in each unit is independent from the other in that each uses its own private clock for internal registers. In that case, the two units are said to be asynchronous to each other. This approach is widely used in most computer systems.

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

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

8.3.1 STROBE CONTROL

The strobe control method of asynchronous data transfer employs a single control line to time each transfer. The strobe may be activated by either the source or the destination unit. Figure 8-3(a) shows a source-initiated transfer. The data bus carries the binary information from source unit to the destination unit. Typically, the bus has multiple lines to transfer an entire byte or word. The strobe is a single line that informs the destination unit when a valid data word is available in the bus.

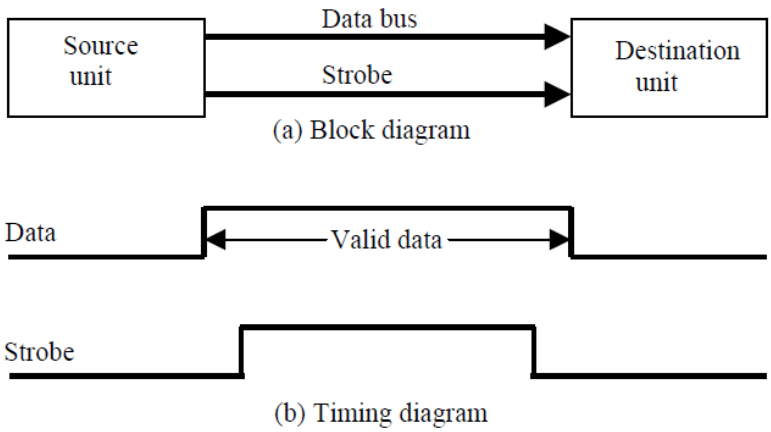


Figure 8-3 Source-initiated strobe for data transfer.

As shown in the timing diagram of Figure 8-3(b), the source unit first places the data on the data bus. After a brief delay to ensure that the data settle to a steady value, the source activates the strobe pulse. The information on the data bus and the strobe signal remain in the active state for a sufficient time period to allow the destination unit to receive the data. Often, the destination unit uses the falling edge of the strobe pulse to transfer the contents of the data bus into one of its internal registers. The source removes the data from the bus a brief period after it disables its strobe pulse. Actually, the source does not have to change the information in the data bus. The fact that the strobe signal is disabled indicates that the data bus does not contain valued data. New valid data will be available only after the strobe is enabled again.

Figure 8-4 shows a data transfer initiated by the destination unit. In this case the destination unit activates the strobe pulse, informing the source to provide the data. The source unit responds by placing the requested binary information on the data bus. The data must be valid and remain in the bus long enough for the destination unit to accept it. The falling edge of the strobe pulse can be used again to trigger a destination register. The destination unit then disables the strobe. The source removes the data from the bus after a predetermined time interval.

In many computers the strobe pulse is actually controlled by the clock pulses in the CPU. The CPU is always in control of the buses and informs the external units how to transfer data. For example, the strobe of Figure 8-3 could be a memory-write control signal from the CPU to a memory unit. The source, being the CPU, places a word on the data bus and informs the memory units which is the destination, that this is a write operation. Similarly, the strobe of figure 8-4 could be a memory-read control signal from the CPU to

a memory unit. The destination, the CPU, initiates the read operation to inform the memory, which is the source, to place a selected word into the data bus.

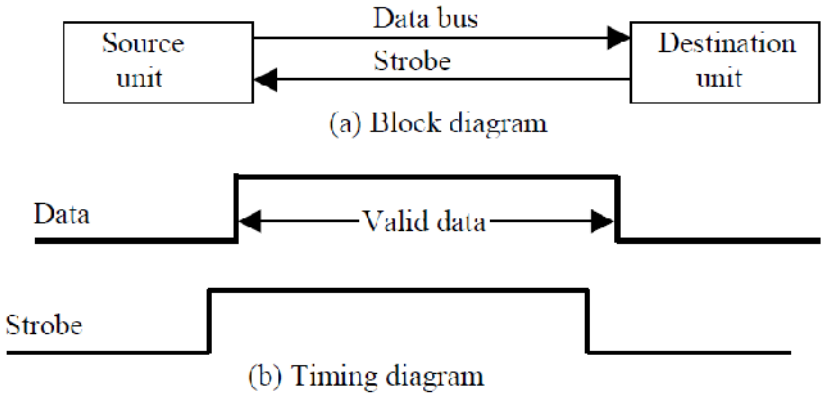


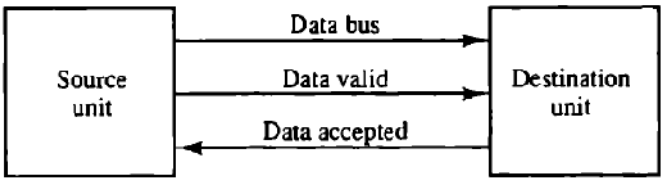
Figure 8-4 Destination-initiated strobe for data transfer.

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

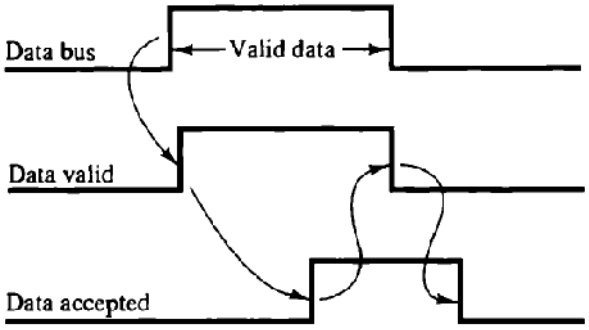
8.3.2 HANDSHAKING

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

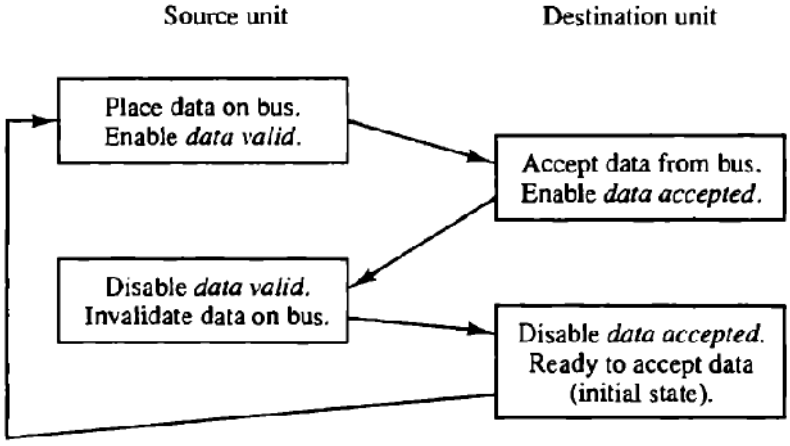
Figure 8-5 shows the data transfer procedure when initiated by the source. The two handshaking lines are data valid, which is generated by the source unit, and data accepted, generated by the destination unit.



(a) Block diagram



(b) Timing diagram



(c) Sequence of events

Figure 8-5 Source-initiated transfer using handshaking.

The timing diagram shows the exchange of signals between the two units. The sequence of events listed in part (c) shows the four possible states that the system can be at any given time. The source unit initiates the transfer by placing the data on the bus and enabling its data valid signal. The data accepted signal is activated by the destination unit after it accepts the data from the bus. The source unit then disables its data valid signal, which invalidates the data on the bus. The destination unit then disables its data accepted signal and the system goes into its initial state. The source dies not send the next data item until after the destination unit shows its readiness to accept new data by disabling its data

accepted signal. This scheme allows arbitrary delays from one state to the next and permits each unit to respond at its own data transfer rate. The rate of transfer is determined by the slowest unit.

The destination-initiated transfer using handshaking lines is shown in Figure 8-6. Note that the name of the signal generated by the destination unit has been changed to ready for data to reflect its new meaning. The source unit in this case does not place data on the bus until after it receives the ready for data signal from the destination unit. From there on, the handshaking procedure follows the same pattern as in the source-initiated case.

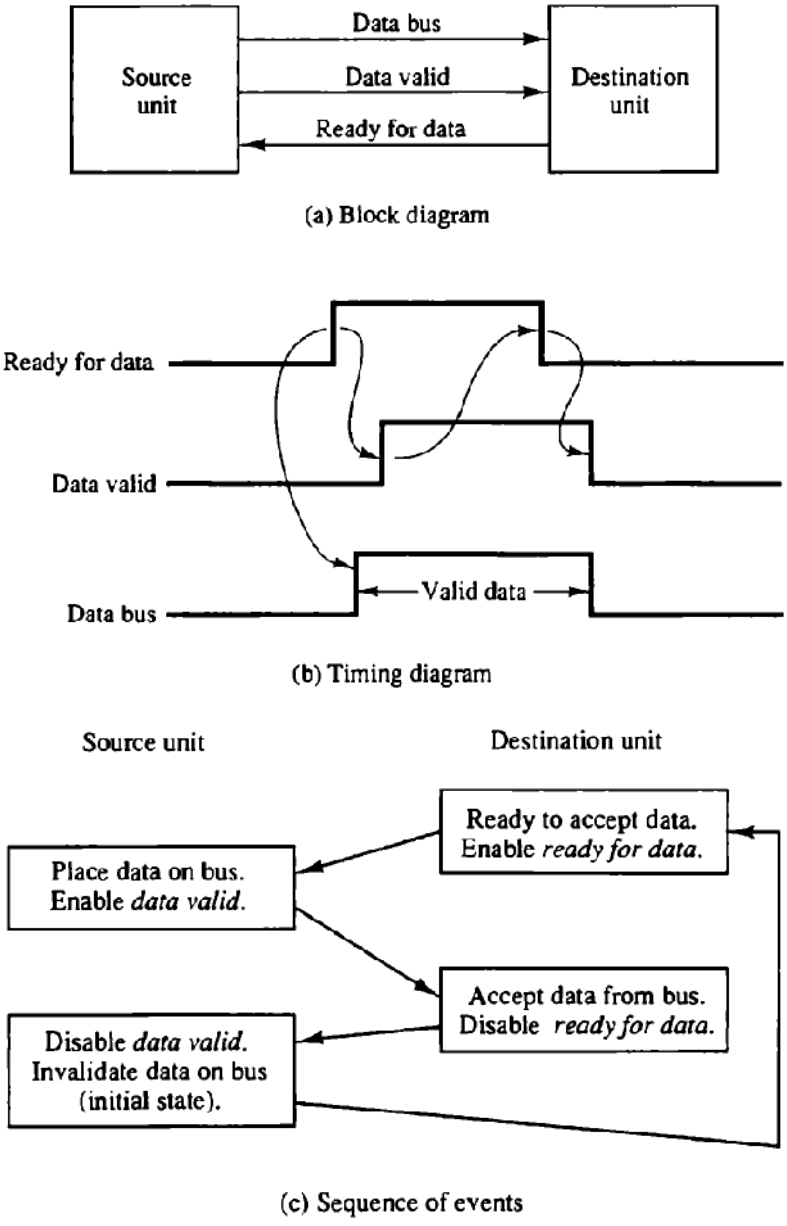


Figure 8-6 Destination-initiated transfer using handshaking.

Note that the sequence of events in both cases would be identical if we consider the ready for data signal as the complement of data accepted. In fact, the only difference between the source-initiated and the destination initiated transfer is in their choice of initial state.

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

8.3.3 ASYNCHRONOUS SERIAL TRANSFER

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

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

Serial asynchronous data transmission technique used in many interactive terminals employs special bits that are inserted at both ends of the character code. With this

technique, each character consists of three parts: a start bit, the character bits, and stop bits. The convention is that the transmitter rests at the 1-state when no characters are transmitted. The first bit, called the start bit, is always a 0 and is used to indicate the beginning of a character. The last bit called the stop bit is always a 1. An example of this format is shown in Figure 8-7.

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

1. When a character is not being sent, the line is kept in the 1-state.
2. The initiation of a character transmission is detected from the start bit, which is always 0.
3. The character bits always follow the start bit.
4. After the last bit of the character is transmitted, a stop bit is detected when the line returns to the 1-state for at least one bit time.

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

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

As illustration, consider the serial transmission of a terminal whose transfer rate is 10 characters per second. Each transmitted character consists of a start bit, eight information bits, of a start bit, eight information bits, and two stop bits, for a total of 11 bits. Ten characters per second means that each character takes 0.1’s for transfer. Since there are 11 bits to be transmitted, it follows that the bit time is 9.09 ms. The baud rate is defined as the rate at which serial information is transmitted and is equivalent to the data transfer in bits per second. Ten characters per second with an 11-bit format has a transfer rate of 110 baud.

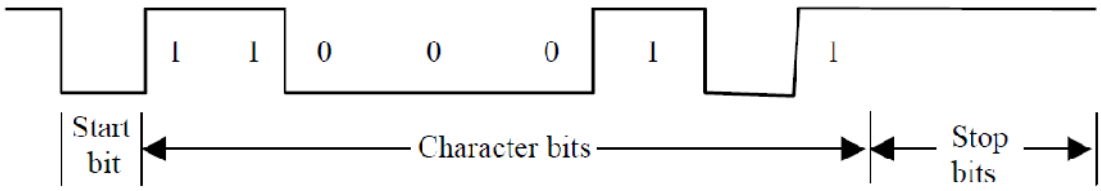


Figure 8-7 Asynchronous serial transmission

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

8.4 MODES OF TRANSFER

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

1. Programmed I/O
2. Interrupt-initiated I/O
3. Direct memory access (DMA)

Programmed I/O operations are the result of I/O instructions written in the computer program. Each data item transfer is initiated by an instruction in the program. Usually, the transfer is to and from a CPU register and peripheral. Other instructions are needed to

transfer the data to and from CPU and memory. Transferring data under program control requires constant monitoring of the peripheral by the CPU. Once a data transfer is initiated, the CPU is required to monitor the interface to see when a transfer can again be made. It is up to the programmed instructions executed in the CPU to keep close tabs on everything that is taking place in the interface unit and the I/O device.

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

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

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

8.4.1 EXAMPLE OF PROGRAMMED I/O

In the programmed I/O method, the I/O device dies not have direct access to memory. A transfer from an I/O device to memory requires the execution of several instructions by the CPU, including an input instruction to transfer the data from the device to the CPU,

and a store instruction to transfer the data from the CPU to memory. Other instructions may be needed to verify that the data are available from the device and to count the numbers of words transferred.

An example of data transfer from an I/O device through an interface into the CPU is shown in Figure 8-8. The device transfers bytes of data one at a time as they are available. When a byte of data is available, the device places it in the I/O bus and enables its data valid line. The interface accepts the byte into its data register and enables the data accepted line. The interface sets a it in the status register that we will refer to as an F or "flag" bit. The device can now disable the data valid line, but it will not transfer another byte until the data accepted line is disabled by the interface. This is according to the handshaking procedure established in Figure 8-5.

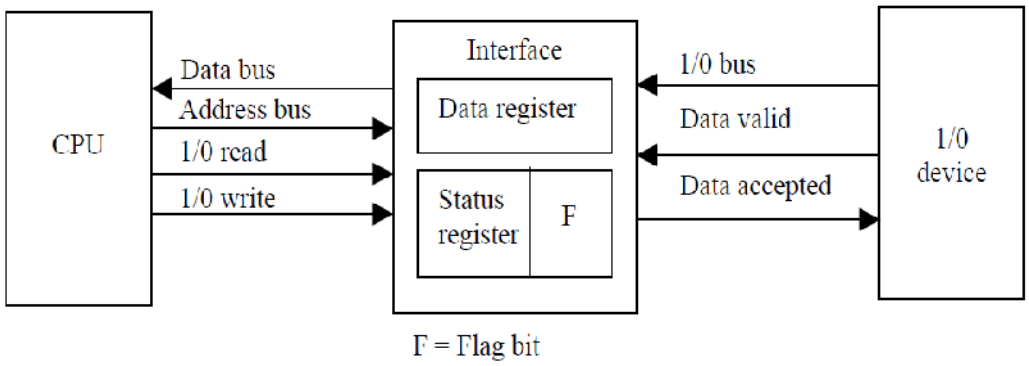


Figure 8-8 Data transfer form I/O device to CPU

A program is written for the computer to check the flag in the status register to determine if a byte has been placed in the data register by the I/O device. This is done by reading the status register into a CPU register and checking the value of the flag bit. If the flag is equal to 1, the CPU reads the data from the data register. The flag bit is then cleared to 0 by either the CPU or the interface, depending on how the interface circuits are designed. Once the flag is cleared, the interface disables the data accepted line and the device can then transfer the next data byte.

A flowchart of the program that must be written for the CPU is shown in Figure 8-9. It is assumed that the device is sending a sequence of bytes that must be stored in memory. The transfer of each byte requires three instructions:

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

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

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

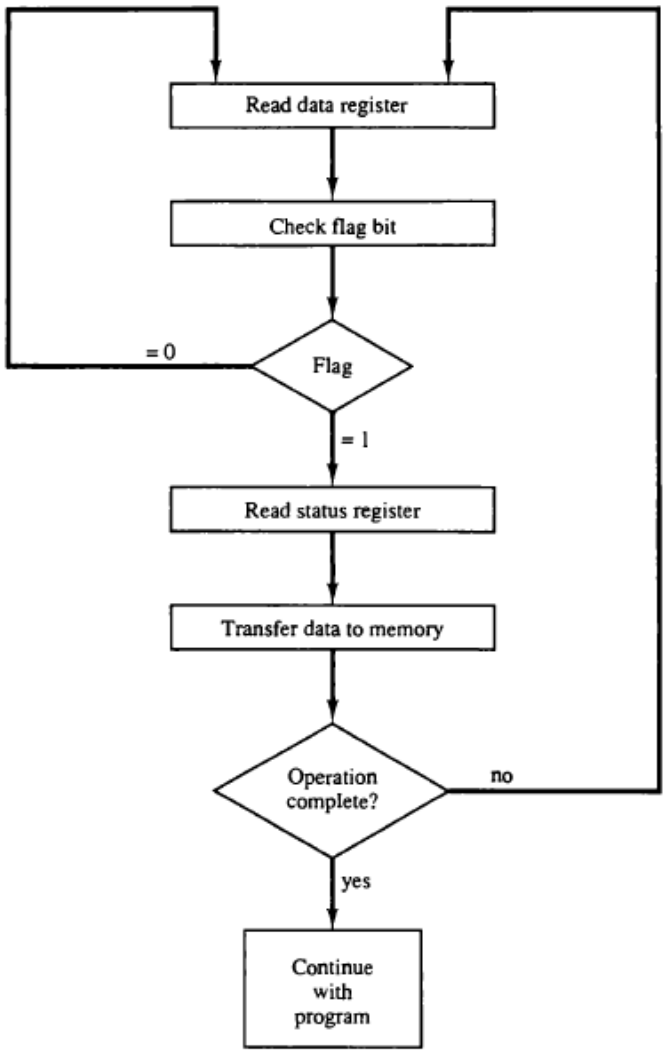


Figure 8-9 Flowchart for CPU program to input data.

8.4.2 INTERRUPT-INITIATED I/O

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

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

8.4.3 SOFTWARE CONSIDERATIONS

The previous discussion was concerned with the basic hardware needed to interface I/O devices to a computer system. A computer must also have software routines for controlling peripherals and for transfer of data between the processor and peripherals. I/O routines must issue control commands to activate the peripheral and to check the device status to determine when it is ready for data transfer. Once ready, information is transferred item by item until all the data are transferred. In some cases, a control command is then given to execute a device function such as stop tape or print characters. Error checking and other useful steps often accompany the transfers. In interrupt-controlled transfers, the I/O software must issue commands to the peripheral to interrupt when ready and to service the interrupt when it occurs. In DMA transfer, the I/O software must initiate the DMA channel to start its operation.

Software control of input-output equipment is a complex undertaking. For this reason I/O routines for standard peripherals are provided by the manufacturer as part of the computer system. They are usually included within the operating system. Most operating systems are supplied with a variety of I/O programs to support the particular line of peripherals offered for the computer. I/O routines are usually available as operating system procedures and the user refers to the established routines to specify the type of transfer required without going into detailed machine language programs.

8.5 PRIORITY INTERRUPT

Data transfer between the CPU and an I/O device is initiated by the CPU. However, the CPU cannot start the transfer unless the device is ready to communicate with the CPU. The readiness of the device can be determined from an interrupt signal. The CPU responds to the interrupt request by storing the return address from PC into a memory stack and then the program branches to a service routine that processes the required transfer. Some processors also push the current PSW for the service routine. We neglect the PSW here in order not to complicate the discussion of I/O interrupts.

In a typical application a number of I/O devices are attached to the computer, with each device being able to originate an interrupt request. The first task of the interrupt system is to identify the source of the interrupt. There is also the possibility that several sources will request service simultaneously. In this case the system must also decide which device to service first.

A priority interrupts is a system that establishes a priority over the various sources to determine which condition is to be serviced first when two or more request arrive simultaneously. The system may also determine which conditions are permitted to interrupt the computer while another interrupt is being serviced. Higher-priority interrupt levels are assigned to request which, if delayed of interrupted, could have serious consequences. Devices with high-speed transfers such as keyboards receive low priority. When two devices interrupt the computer at the same time, the computer services the devices interrupt the computer at the same time, the computer services the device, with the higher priority first.

Establishing the priority of simultaneous interrupts can be done by software or hardware. A polling procedure is used to identify the highest-priority source by software means. In

this method there is one common branch address for all interrupts. The program that takes care of interrupts begins at the branch address and polls the interrupt sources in sequence. The order in which they are tested determines the priority of each interrupt. The highest-priority source is tested first, and if its interrupt signal is on, control branches to a service routine for this source.

Otherwise, the next-lower-priority source is tested, and so on. Thus the initial service routine for all interrupt consists of a program that tests the interrupt sources in sequence and branches to one of many possible service routines. The particular service routine reached belongs to the highest priority device among all devices that interrupted the computer. The disadvantage of the soft ware method is that if there are many interrupts, the time required to poll them can exceed the time available to service the I/O device. In this situation a hardware priority-interrupt unit can be used to speed up the operation.

A hardware priority-interrupt unit functions as an overall manager in an interrupt system environment. It accepts interrupt requests from many sources, determines which of the incoming requests has the highest priority, and issues an interrupt request to the computer based on this determination. To speed up the operation, each interrupt source has its own interrupt vector to access its own service routine directly. Thus no polling is required because all the decisions are established by the hardware priority-interrupt unit. The hardware priority function can be established by either a serial or a parallel connection of interrupt lines. The serial connection is also known as the daisy chaining method.

8.5.1 DAISY-CHAINING PRIORITY

The daisy-chaining method of establishing priority consists of a serial connection of all devices that request an interrupt. The device with the highest priority is placed in the first position, followed by lower-priority devices up to the device with the lowest priority, which is placed last in the chain. This method of connection between three devices and the CPU is shown in Figure 8-10. The interrupt request line is common to all devices and forms a wired logic connection. If any device has its interrupt signal in the low-level state, the interrupt line goes to the low-level state and enables the interrupt input in the CPU. When no interrupts are pending, the interrupt line stays in the high-level state and no interrupts are recognized by the CPU. This is equivalent to a negative logic OR operation. The CPU responds to an interrupt request by enabling the interrupt acknowledge line. This signal is received by device 1 at its PI (priority in) input. The acknowledge signal passes on to the next device through the PO (priority out) output only

if device 1 is not requesting an interrupt. If device 1 has a pending interrupt, it blocks the acknowledge signal from the next device by placing a 0 in the PO output. It then proceeds to insert its own interrupt vector address (VAD) into the data bus for the CPU to use during the interrupt cycle.

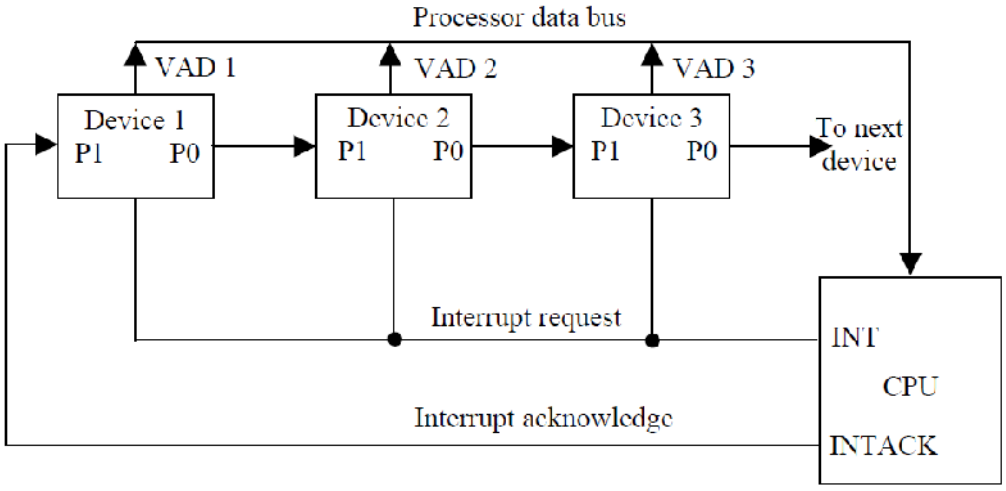


Figure 8-10 Daisy-chain priority interrupt

A device with a 0 in its PI input generates a 0 in its PO output to inform the next-lower priority device that the acknowledge signal has been blocked. A device that is requesting an interrupt and has a 1 in its PI input will intercept the acknowledge signal by placing a 0 in its PO output. If the device does not have pending interrupts, it transmits the acknowledge signal to the next device by placing a 1 in its PO output. Thus the device with PI = 1 and PO = 0 is the one with the highest priority that is requesting an interrupt, and this device places its VAD on the data bus. The daisy chain arrangement gives the highest priority to the device that receives the interrupt acknowledge signal from the CPU. The farther the device is from the first position, the lower is its priority.

Figure 8.11 shows the internal logic that must be included with in each device when connected in the daisy-chaining scheme. The device sets its RF flip-flop when it wants to interrupt the CPU. The output of the RF flip-flop goes through an open-collector inverter, a circuit that provides the wired logic for the common interrupt line. If PI = 0, both PO and the enable line to VAD are equal to 0, irrespective of the value of RF. If PI = 1 and RF = 0, then PO = 1 and the vector address is disabled. This condition passes the acknowledge signal to the next device through PO. The device is active when PI = 1 and RF = 1. This condition places a 0 in PO and enables the vector address for the data bus. It

is assumed that each device has its own distinct vector address. The RF flip-flop is reset after a sufficient delay to ensure that the CPU has received the vector address.

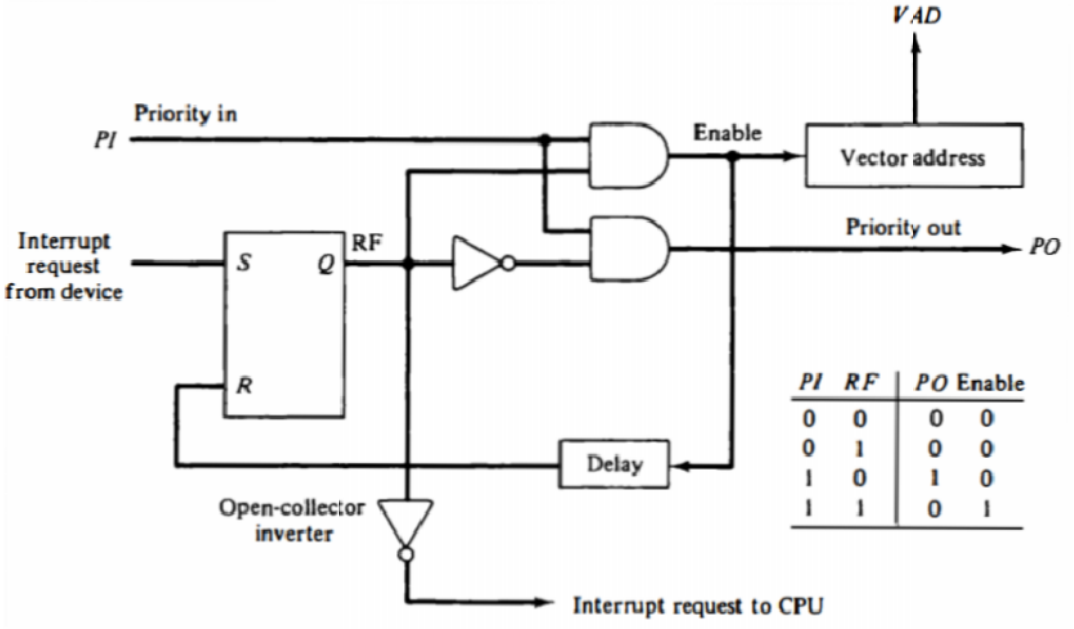


Figure 8-11 One state of the daisy-chain priority arrangement.

8.5.2 PARALLEL PRIORITY INTERRUPT

The parallel priority interrupt method uses a register whose bits are set separately by the interrupt signal from each device. Priority is established according to the position of the bits in the register. In addition to the interrupt register the circuit may include a mask register whose purpose is to control the status of each interrupt request. The mask register can be programmed to disable lower-priority interrupts while a higher-priority device is being serviced. It can also provide a facility that allows a high-priority device to interrupt the CPU while a lower-priority device is being serviced.

The priority logic for a system of four interrupt sources is shown in Figure 8.12. It consists of an interrupt register whose individual bits are set by external conditions and cleared by program instructions. The magnetic disk, being a high-speed device, is given the highest priority. The printer has the next priority, followed by a character reader and a keyboard. The mask register has the same number of bits as the interrupt register. By means of program instructions, it is possible to set or reset any bit in the mask register. Each interrupt bit and its corresponding mask bit are applied to an AND gate to produce the four inputs to a priority encoder. In this way an interrupt is recognized only if its

corresponding mask bit is set to 1 by the program. The priority encoder generates two bits of the vector address, which is transferred to the CPU.

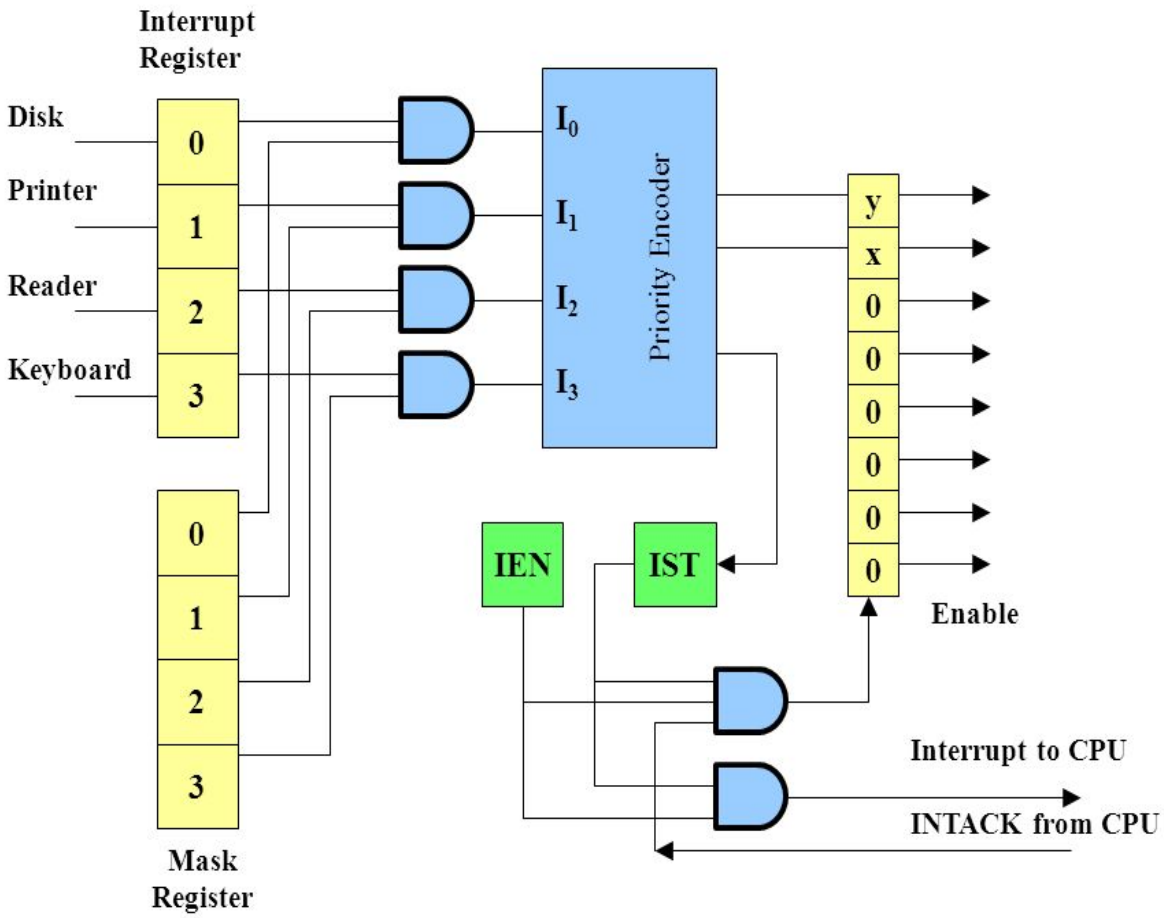


Figure 8-12 Priority interrupt hardware.

Another output from the encoder sets an interrupt status flip-flop IST when an interrupt that is not masked occurs. The interrupt enable flip-flop IEN can be set or cleared by the program to provide an overall control over the interrupt system. The outputs of IST ANDed with IEN provide a common interrupt signal for the CPU. The interrupt acknowledge INTACK signal from the CPU enables the bus buffers in the output register and a vector address VAD is placed into the data bus. We will now explain the priority encoder circuit and then discuss the interaction between the priority interrupt controller and the CPU.

8.6 DIRECT MEMORY ACCESS (DMA)

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

The CPU may be placed in an idle state in a variety of ways. One common method extensively used in microprocessors is to disable the buses through special control signals. Figure 8.13 shows two control signals in the CPU that facilitate the DMA transfer. The bus request (BR) input is used by the DMA controller to request the CPU to relinquish control of the buses. When this input is active, the CPU terminates the execution of the current instruction and places the address bus, the data bus, and the read and write lines into a high-impedance state behaves like an open circuit, which means that the output is disconnected and does not have a logic significance. The CPU activates the Bus grant (BG) output to inform the external DMA that the buses are in the high-impedance state. The DMA that originated the bus request can now take control of the buses to conduct memory transfers without processor intervention. When the DMA terminates the transfer, it disables the bus request line. The CPU disables the bus grant, takes control of the buses, and returns to its normal operation.

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

8.6.1 DMA CONTROLLER

The DMA controller needs the usual circuits of an interface to communicate with the CPU and I/O device. In addition, it needs an address register, a word count register, and a set of address lines. The address register and address lines are used for direct communication with the memory. The word count register specifies the number of words that must be transferred. The data transfer may be done directly between the device and memory under control of the DMA.

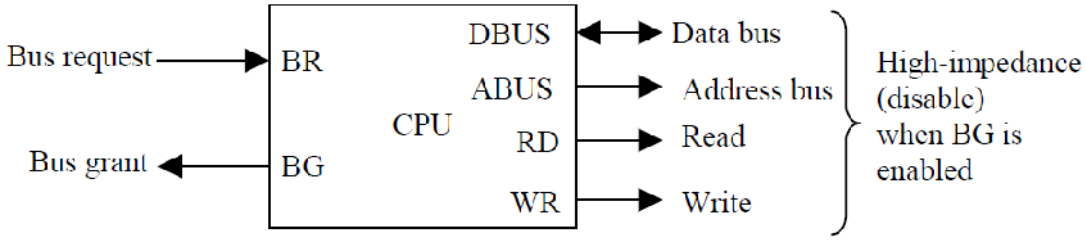


Figure 8.13 CPU bus signals for DMA transfer.

Figure 6.14 shows the block diagram of a typical DMA controller. The unit communicates with the CPU via the data bus and control lines. The registers in the DMA are selected by the CPU through the address bus by enabling the DS (DMA select) and RS (register select) inputs. The RD (read) and WR (write) inputs are bidirectional. When the BG (bus grant) input is 0, the CPU can communicate with the DMA registers through the data bus to read from or write to the DMA registers. When BG = 1, the CPU has relinquished the buses and the DMA can communicate directly with the memory by specifying an address in the address bus and activating the RD or WR control. The DMA communicates with the external peripheral through the request and acknowledge lines by using a prescribed handshaking procedure.

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

Thus the CPU can read from or write into the DMA registers under program control via the data bus.

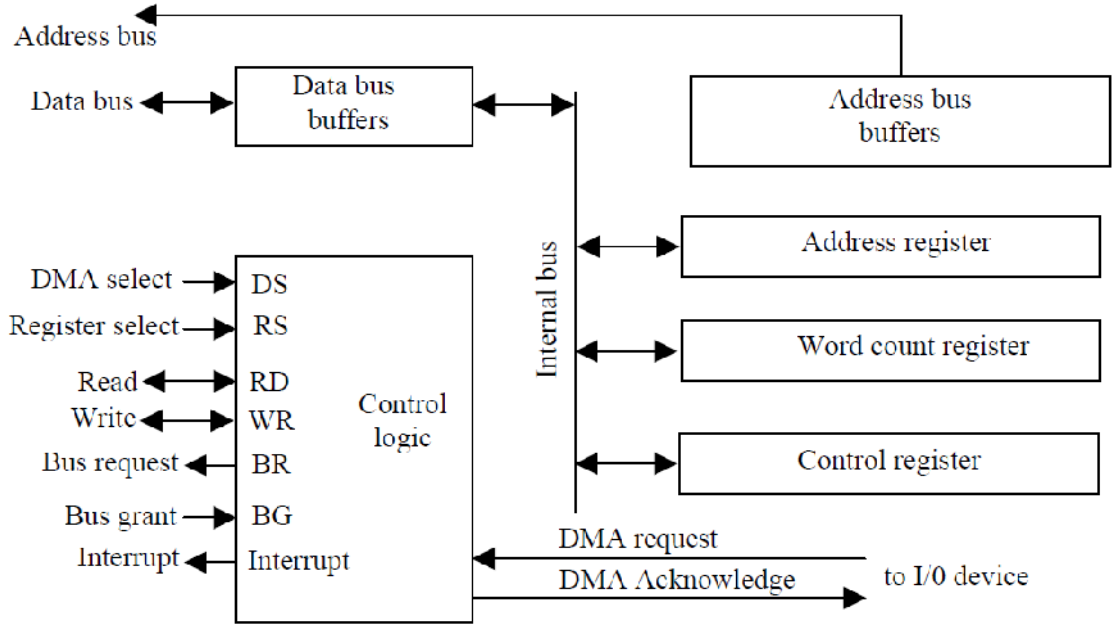


Figure 8.14 Block diagram of DMA controller.

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

1. The starting address of the memory block where data are available (for read) or where data are to be stored (for write)
2. The word count, which is the number of words in the memory block
3. Control to specify the mode of transfer such as read or write
4. A control to start the DMA transfer

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

8.6.2 DMA TRANSFER

The position of the DMA controller among the other components in a computer system is illustrated in Figure 8-15. The CPU communicates with the DMA through the address and data buses as with any interface unit. The DMA has its own address, which activates the DS and RS lines. The CPU initializes the DMA through the data bus. Once the DMA receives the start control command, it can start the transfer between the peripheral device and the memory.

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

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

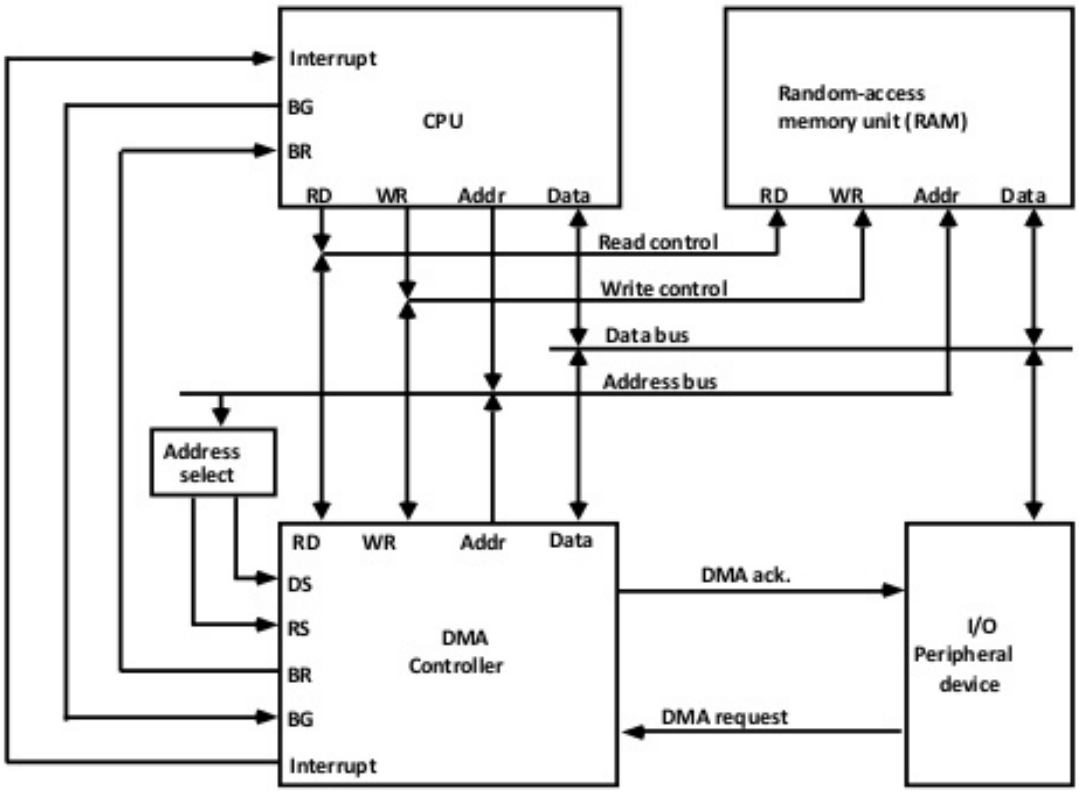


Figure 8-15 DMA transfer in a computer system

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

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

A DMA controller may have more than on channel. In this case, each channel has a request and acknowledges pair of control signals which are connected to separate peripheral devices. Each channel also has its own address register and word count register

within the DMA controller. A priority among the channels may be established so that channels with high priority are serviced before channels with lower priority.

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

8.7 CHECK YOUR PROGRESS

1. The CRT contains an _____ that sends an electronic beam to a phosphorescent screen in front of the tube.
2. The _____ uses a rotating photographic drum that is used to imprint the character images.
3. The ASCII code contains _____ characters that can be printed and 34 nonprinting characters used for various control functions.
4. Input-output interface provides a method for transferring information between _____ and external I/O devices.
5. The _____ method of asynchronous data transfer employs a single control line to time each transfer.
6. When two devices interrupt the computer at the same time, the computer services the devices interrupt the computer at the same time, the computer services the device, with the _____ first.
7. The _____ interrupt method uses a register whose bits are set separately by the interrupt signal from each device.
8. During _____ transfer, the CPU is idle and has no control of the memory buses.
9. A high-impedance state behaves like an _____ .
10. The DMA controller has three registers: an address register, a word count register, and a _____ .

8.8 SUMMARY

1. The input-output subsystem of a computer, referred to as I/O, provides an efficient mode of communication between the central system and the outside environment.
2. Devices that are under the direct control of the computer are said to be connected on-line.
3. Input and output devices that communicate with people and the computer are usually involved in the transfer of alphanumeric information to and from the device and the computer is ASCII
4. Input-output interface provides a method for transferring information between internal storage and external I/O devices. Peripherals connected to a computer need special communication links for interfacing them with the central processing unit.
5. The internal operations in a digital system are synchronized by means of clock pulses supplied by a common pulse generator.
6. The transfer of data between two units may be done in parallel or serial. In parallel data transmission, each bit of the message has its own path and the total message is transmitted at the same time.
7. Binary information received from an external device is usually stored in memory for later processing. Information transferred from the central computer into an external device originates in the memory unit.
8. An alternative to the CPU constantly monitoring the flag is to let the interface inform the computer when it is ready to transfer data.
9. Data transfer between the CPU and an I/O device is initiated by the CPU. However, the CPU cannot start the transfer unless the device is ready to communicate with the CPU. The readiness of the device can be determined from an interrupt signal.
10. The DMA controller needs the usual circuits of an interface to communicate with the CPU and I/O device. In addition, it needs an address register, a word count register, and a set of address lines.
11. The transfer of data between a fast storage device such as magnetic disk and memory is often limited by the speed of the CPU. Removing the CPU from the

path and letting the peripheral device manage the memory buses directly would improve the speed of transfer.

12. A DMA controller may have more than on channel. In this case, each channel has a request and acknowledges pair of control signals which are connected to separate peripheral devices.

8.9 KEYWORDS

1. **Input peripherals** allow user input, from the outside world to the computer. Example: Keyboard, Mouse etc.
2. **Output peripherals** allow information output, from the computer to the outside world. Example: Printer, Monitor etc
3. **Input-Output peripherals** allow both input (from outside world to computer) as well as, output (from computer to the outside world). Example: Touch screen etc.
4. **Interface** is the path for communication between two components.
5. **Programmed input–output** (also programmed input/output, programmed I/**O**, PIO) is a method of transferring data between the CPU and a peripheral.
6. **Interrupt** I/**O** is a way of controlling input/output activity whereby a peripheral or terminal that needs to make or receive a data transfer sends a signal. This will cause a program interrupt to be set.
7. **Direct memory access (DMA)** is a method that allows an input/output (I/O) device to send or receive data directly to or from the main memory, bypassing the CPU to speed up memory operations.
8. **Bus Request** is used by the DMA controller to request the CPU to relinquish the control of the buses.
9. **Bus Grant** is activated by the CPU to inform the external DMA controller that the buses are in high impedance state and the requesting DMA can take control of the buses.
10. **Cyclic Stealing** is an alternative method in which DMA controller transfers one word at a time after which it must return the control of the buses to the CPU.

8.10 SELF ASSESSMENT TEST

1. Enlist and explain that how many types of I/O devices can be connected to the Computer?
2. What is I/O bus interface module? Explain any one in detail?
3. Compare and contrast I/O and memory bus.
4. What is asynchronous data transfer? What are the modes with the help of which we can transfer the data to the said destination?
5. What are the different modes of transfer?
6. What is a priority interrupt? What are the different types of the priorities in DMA?
7. What is a DMA? Draw its block and IC diagram and also explain its working.
8. How DMA transfer takes place? Comment.

8.11 ANSWER TO CHECK YOUR PROGRESS

1. electronic gun
2. laser printer
3. 94
4. internal storage
5. strobe control
6. higher priority
7. parallel priority
8. DMA
9. open circuit
10. control register

8.12 REFERENCES / SUGGESTED READINGS

1. Computer Organization and Architecture, Rajaram & Radhakrishan, PHI.
2. Computer Organization & Architecture: Designing for Performance, Stalling, PHI.
3. Computer Organization and Design, Pal Choudhary, PHI.
4. Computer Systems Organization & Architecture, Carpenelli, Pearson Education.

5. Computer Organization and Architecture, Stalling, Pearson Education.
6. Computer System Architecture, Morris Mano, PHI.
7. Computer Architecture and Organization, McGraw Hill Company, New Delhi. J.P. Hayes.

SUBJECT: COMPUTER SYSTEM ARCHITECTURE	
COURSE CODE: MCA-24	**AUTHOR: DR. MANOJ DUHAN**
LESSON NO. 9	
MICRO PROGRAMMED CONTROL	
REVISED / UPDATED SLM BY NEERAJ VERMA	

STRUCTURE

9.0 LEARNING OBJECTIVES

After reading this Unit, the reader should be able to understand the concept of microprogramming and define control memory, microinstruction, micro program, control address register, address sequencer etc.

The design of micro programmed control unit, by designing hardware for the microprogram sequencer.

9.1 INTRODUCTION

The function of the control unit in a digital computer is to initiate sequences of microoperations. The number of different types of microoperations that are available in a given system is finite. The complexity of the digital system is derived from the number of sequences of microoperations that are performed. When the control signals are generated by hardware using conventional logic design techniques, the control unit is said to be hardwired. Microprogramming is a second alternative for designing the control unit of a digital computer. The principle of microprogramming is an elegant and systematic method for controlling the micro operation sequence in a digital computer.

The control function that specifies a microoperation is a binary variable. When it is in one binary state, the corresponding microoperation is executed. A control variable in the opposite binary state does not change the state of the registers in the system. The active state of a control variable may be either the 1 state or the 0 state, depending on the application. In a bus-organized system, the control signals that specify micro operation are groups of bits that select the paths in multiplexers, decoders, and arithmetic logic units.

The control unit initiates a series of sequential steps of microoperations. During any given time, certain microoperations are to be initiated, while others remain idle. ;the control variables at any given time can be represented by a string of 1's and 0' a called a control word. As such, control words can be programmed to perform various operations on the components are stored in memory is called a micro programmed control unit. Each word in control memory contains within it a microinstruction. The microinstruction specifies

one or more micro operations for the system. A sequence of microinstructions constitutes a microprogram. Since alterations of the microprogram are not need once the control unit is in operation, the control memory can be a read-only memory (ROM). The content of the words in ROM are fixed and cannot be altered by simple programming since no writing capability is available in the ROM. ROM words are made permanent during he hardware production of the unit. The use of a microprogram involves placing all control variables in words of ROM for use by the control unit through successive read operations. The content of the word in ROM at a given address specifies a microinstruction

A more advanced development known as dynamic microprogramming permits a microprogram to be loaded initially from an auxiliary memory such as a magnetic disk. Control units that use dynamic microprogramming employ a writable control memory. This type of memory can be used for writing (to change the microprogram) but is used mostly for reading. A memory that is part of a control unit is referred to as a control memory.

A computer that employs a micro programmed control unit will have two separate memories: a main memory and a control memory. The main memory is available to the user for storing the program. The contents of main memory may alter when the data are manipulated and every time that the program is changed. The user's program in main memory consists of machine instructions and data. In contrast, the control memory holds a fixed microprogram that cannot be altered by the occasional user. The microprogram consists of microinstructions that specify various internal control signals for execution of register microoperations. Each machine instruction initials a series of microinstructions in control memory. These microinstructions generate the rations to fetch the instruction from main memory; to evaluate the effective address, to execute the operation specified by the instruction, and to return control to the fetch phase in order to repeat the cycle for the next instruction.

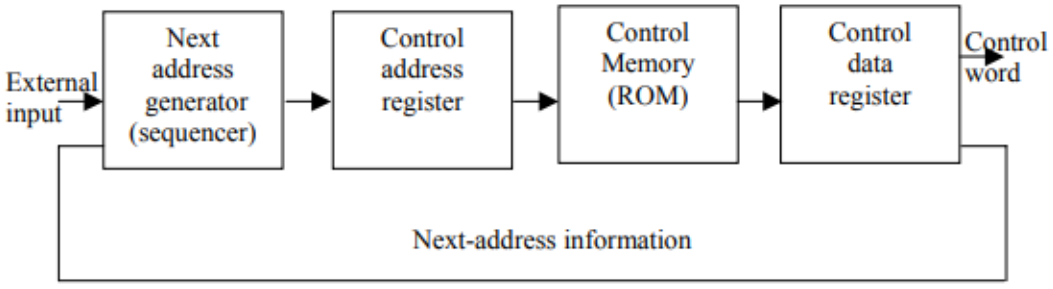


Figure 9-1 Microprogrammed Control Organization

The general configuration of a microprogrammed control unit is demonstrated in the block diagram of Figure 9-1. The control memory is assumed to be a ROM, within which all control information is permanently stored. The control memory address register specifies the address of the microinstruction, and the control data register holds the microinstruction read from memory. The microinstruction contains a control word that specifies one or more microoperations for the data processor. Once these operations are executed, the control must determine the nest 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. For this reason it is necessary to use some bits of the present microinstruction to control the generation of the address of the next microinstruction. The next address may also be a function of external input conditions. While the microoperations are being executed, the next address is computer in the next address generator circuit and then transferred into the control address register to read the next microinstruction. Thus a microinstruction contains bits for initiating microoperations in the data processor part and bits that determine the address sequence for the control memory.

The next address generator is sometimes called a microprogram sequencer, as it determines the address sequence that is read from control memory. ;the address of the next microinstruction can be specified in several ways, depending on the sequencer inputs. Typical functions of a microprogram sequencer are incrementing the control address register by one, loading into the control address register an address from control memory, transferring an external address, or loading an initial address to start the control operations.

The control data register holds the present microinstruction while the next address is computed and read from memory; the data register is sometimes called a pipeline register. It allows the execution of the microoperations specified by the control word simultaneously with the generation of the next microinstruction. This configuration requires a two-phase clock., with one clock applied to the address register and the other to the data register.

The system can operate without the control data register by applying a single-phase clock to the address register. The control word and next-address information are taken directly

from the control memory. ;it must be realized that a ROM operates as a combinational circuit, with the address value as the input and the corresponding word as the output. The content of the specified word in ROM remains in the output wires as long as its address value remains in the address register. No read signal is needed as in a random-access memory. Each clock pulse will execute the microoperations specified by the control word and also transfer a new address to the control address register. In the example that follows we assume a single-phase clock and therefore we do not use a control data register. In this way the address register is the only component in the control system that receives clock pulses. The other two components: the sequencer and the control memory are combinational circuits and do not need a clock.

The main advantage of the micro programmed control is the fact that once the hardware configuration is established; there should be no need for further hardware or wiring changes. If we want to establish a different control sequence for the system, all we need to do is specify a different set of microinstruction for control memory. The hardware configuration should not be changed for different operations; the only thing that must be changed is the microprogram residing in control memory.

It should be mentioned that must computers based on the reduced instruction set computer (RISC) architecture concept, use hardwired control rather than a control memory with a microprogram.

9.2 ADDRESS SEQUENCING

Microinstructions are stored in control memory in groups, with each group specifying routine. Each computer instruction has its own microprogram routine in control memory to generate the microoperations that execute the instruction. 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. To appreciate the address sequencing in a microprogram control unit, let us enumerate the steps that the control must undergo during the execution of a single computer instruction.

An initial address is loaded into the control address register when power is turned on in the computer. This address is usually the address of the first microinstruction that activates the instruction fetch routine. The fetch routine may be sequenced by

incrementing the control address register through the rest of its microinstructions. At the end of the fetch routine, the instruction is in the instruction register of the computer.

The control memory next must go through the routine that determines the effective address of the operand. A machine instruction may have bits that specify various addressing modes, such as indirect address and index registers. The effective address computation routine in control memory can be reached through a branch microinstruction, which is conditioned on the status of the mode bits of the instruction. When the effective address computation routine is completed, the address of the operand is available in the memory address register.

The next step is to generate the microoperations that execute the instruction fetched from memory. The microoperation steps to be generated in processor register depend on the operation code part of the instruction. Each instruction has its own microprogram routine stored in a given location of control memory. The transformation from the instruction code bits to an address in control memory where the routine is located is referred to as a mapping process. A mapping procedure is a rule that transforms the instruction code into a control memory address. Once the required routine is reached, the microinstructions that execute the instruction may be sequenced by incrementing the control address register, but sometimes the sequence of microoperations will depend on values of certain status bits in processor registers. Micro programs that employ subroutines will require an external register for storing the return address. Return addresses cannot be stored in ROM because the unit has no writing capability.

When the execution of the instruction is completed, control must return to the fetch routine. This is accomplished by executing an unconditional branch microinstruction to the first address of the fetch routine. In summary, the address sequencing capabilities required in control memory are:

1. Incrementing of the control address register.
2. Unconditional branch or conditional branch, depending on statues bit conditions.
3. A mapping process from the bits of the instruction to an address for control memory.
4. A facility for subroutine call and return.

Figure 9-2 shows a block diagram of a control memory and the associated hardware needed for selecting the next microinstruction. The microinstruction in control memory contains a set of bits to initiate micro oprerations in computer registers and other bits to

specify the method by which the next address is obtained. The diagram shows four different paths from which the control address register (CAR) receive the address. The incremented increments the content of the control 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 statues 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 whose value is then used when the microprogram wishes to return from the subroutine.

9.2.1 CONDITIONAL BRANCHING

The branch logic provides decision-making capabilities in the control unit. 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 input or output status conditions. Information in these bits can be tested and actions initiated based on their condition: whether their value is 1 or 0. The status bits, together with the field in the microinstruction that specifies a branch address, control the conditional branch decisions generated in the branch logic.

The branch logic hardware may be implemented in a variety of ways. The simplest way is to test the specified condition and branch to the indicated address if the condition is met; otherwise, the address register is incremented.

This can be implemented with a multiplexer. Suppose that there are eight status bit conditions in the system. Three bits in the microinstruction are used to specify any one of eight status bit conditions. These three bits provide the selection variables for the multiplexer. If the selected status bits in the 1 state, the output of the multiplexer is 1; otherwise, it is 0. A 1 output in the multiplexer generates a control signal to transfer the branch address from the microinstruction into the control address register. A 0 output in the multiplexer causes the address register to be incremented. In this configuration, the microprogram follows one of two possible paths, depending on the value of the selected status bit.

An unconditional branch microinstruction can be implemented by loading the branch address from control memory into the control address register. This can be accomplished by fixing the value of one status bit at the input of the multiplexer, so it is always equal to

1. A reference to this bit by the status bit select lines from control memory causes the branch address to loaded into the control address register unconditionally.

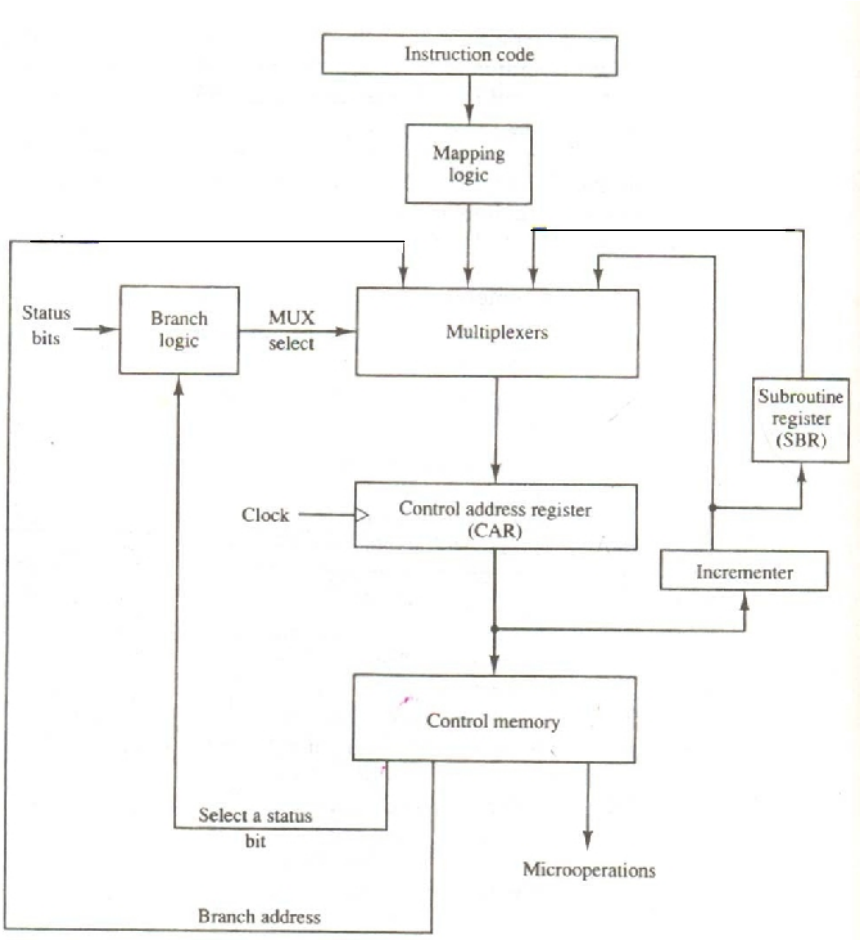


Figure 9.2 Selection of Address for control memory.

9.2.2 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 for an instruction is located. The status bits for this type of branch are the bits in the operation code part of the instruction. For example, a computer with a simple instruction format as shown in Fig. 9-3 has an operation code of four bits which can specify up to 16 distinct instructions. Assume further that the control memory has 128 words, requiring an address of seven bits. For each operation code there exists a micro program routine in control memory that executes the instruction. One simple mapping process that converts the 4-bit operation code to a 7-bit address for control memory is shown in fig. 9-3. This mapping consists of placing a 0

in the most significant bit of the address, transferring the four operation code bits, and clearing the two lest significant bits of the control address register. This provides for each computer instruction a microprogram routine with a capacity of four microinstructions. If the routine needs more than four microinstructions, it can use addresses 1000000 through 1111111. If it uses fewer than four microinstructions, the unused memory cautions would be available for other routines.

One can extend this concept to a more general mapping rule by using a ROM to specify the mapping function. In this configuration, the bits of the instruction specify the address of a mapping ROM. The contents of the mapping ROM give the bits for the control address register. In this way the microprogram routine that executes the instruction can be placed in any desired location in control memory. The mapping concept provides flexibility for adding instruction for control memory as the need arises.

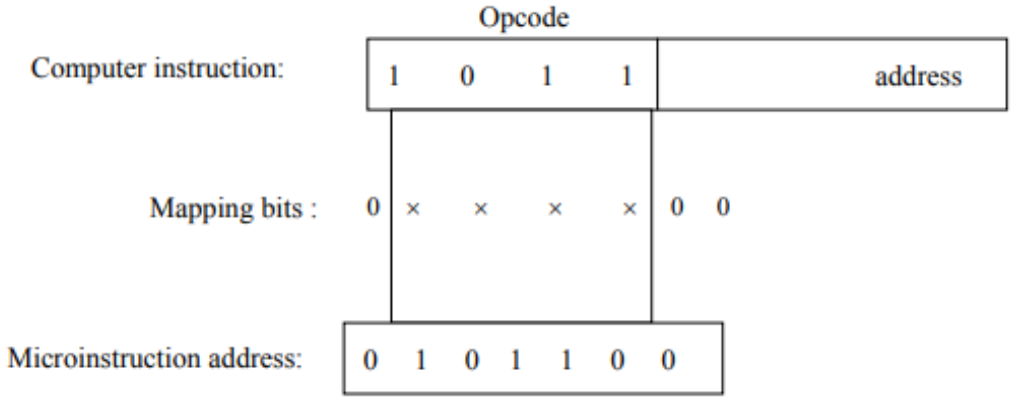


Figure 9-3 Mapping from instruction code to microinstruction address.

The mapping function is sometimes implemented by means of an integrated circuit called programmable logic device or PLD. A PLD is similar to ROM in concept except that it uses AND and OR gates with internal electronic fuses. The interconnection between inputs, AND gates, OR gates, and outputs can be programmed as in ROM. A mapping function that can be expressed in terms of Boolean expressions can be implemented conveniently.

9.2.3 SUBROUTINES

Subroutines are programs that are used by other routines to accomplish a particular task. A subroutine can be called from any point within the main body of the microprogram. Frequently, many micro programs contain identical sections of code. Micro instructions

can be saved by employing subroutines that use common sections of microcode. For example, the sequence of microoperation needed to generate the effective address of the operand for an instruction is common to all memory reference instructions. This sequence could be a subroutine that is called from within many other routines to execute the effective address computation.

Micro programs that use subroutines must have a provision for storing the return address during a subroutine call and restoring the address during a subroutine return. This may be accomplished by placing the incremented output form the control address register into a subroutine register and branching to the beginning of the subroutine. The subroutine register can then become the source for transferring the address for the return to the main routine. The best way to structure a register file that stores addresses for subroutines is to organize the registers in a last-in, first-out (LIFO) stack.

9.3 MICROPROGRAM EXAMPLE

Once the configuration of a computer and its micro programmed control unit is established, the designer's task is to generate the microcode for the control memory. This code generation is called microprogramming and is a process similar to conventional machine language programming. To appreciate this process, we present here a simple digital computer and show how it is micro programmed. The computer used here is similar but not identical to the basic computer.

9.3.1 COMPUTER CONFIGURATION

The block diagram of the computer is shown in Fig 9-4. It consists of two memory units: a main memory for storing instructions and data, and a control memory for storing he microprogram. Four register are associated with the processor unit and two with the control unit. The processor registers are program counter PC, address register AR, data register DR, and accumulator register DR, and accumulator register AC. The function of these registers is similar to the basic computer. The control unit has a control address register CAR and a subroutine register SBR. The control memory and its registers are organized as a micro programmed control unit, as shown in Fig. 9-2.

The transfer of information among the register in the processor is done through multiplexers rather than a common bus. DR can receive information from AC, PC, or memory. AR can receive information from PC or DR, PC can receive information only from AR. The arithmetic, logic, and shift unit performs micro opreations with data from AC and DR and places the result in AC. Note that memory receive its address from AR. Input data written to memory come from DR, and data read from memory can go only to DR.

The computer instruction format is depicted in Fig. 9-5(a). It consists of three fields: a 1-bit field for indirect addressing symbolized by I, a 4-bit operation code (opcode), and an 11-bit address field. Figure 9-5(b) lists four of the 16 possible memory-reference instructions. The ADD instruction adds the content of the operand found in the effective address to the content of AC. The BRANCH instruction causes a branch to the effective address if the operand in AC is negative. The program proceeds with the next consecutive instruction if AC is not negative. The AC is negative if its sign bit (the bit in the leftmost position of the register) is a 1. The STORE instruction transfers the content of AC into the memory word specified by the effective address. The EXCHANGE instruction swaps the data between AC and the memory word specified by the effective address.

It will be shown subsequently that each computer instruction must be micro programmed. In order not to complicate the microprogramming example, only four instructions are considered here. It should be realized that 12 other instructions can be included and each instruction must be micro programmed by the procedure outlined below.

9.3.2 MICROINSTRUCTION FORMAT

The microinstruction format for the control memory is shown in Fig. 9-6. The 20 bits of the microinstruction are divided into four functional parts. The three fields F1, F2, and F3 specify microoperations for the computer. The CD field selects status bit conditions. The BR field specifies the type of branch to be used. The AD field contains a branch address. The address field is seven bits wide, since the control memory has 128 = 27 words.

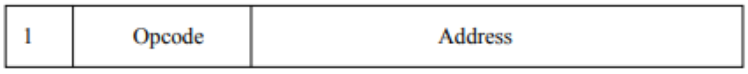
The microoperations are subdivided into three fields of three bits each. The three bits in each field are encoded to specify seven distinct microoperations as listed in Table 9-1. This gives a total of 21 microoperations. No more than three microoperations can be chosen for a microinstruction, one from each field. If fewer than three microoperations are used, one or more of the fields will use the binary code 000 for no operation. As an

illustration, a microinstruction can specify two simultaneous micro operation from F2 and F3 and none from F1.

$$DR \leftarrow M[AR] \text{ with } F2 = 100$$
$$\text{and } PC \leftarrow PC + 1 \text{ with } F3 = 101$$

The nine bits of the microoperation fields will than be 000 100 101. It is important to realize that two or more conflicting microoperations cannot be specified simultaneously. For example, a microoperation field 010 001 000 has no meaning because it specifies the operations to clear AC to 0 and subtract DR from AC at the same time.

Each microoperation in Table 9-1 is defined with a register transfer statement and is assigned a symbol for use in a symbolic microprogram. All transfer-type microoperations symbols use five letters. The first two letters designate the source register, the third letters is always a T, and the last two letters designate the destination register. For example, the microoperation that specifies the transfer AC ← DR (F1 = 100) has the symbol DRTAC, which stands for a transfer from DR to AC.



(a) Instruction format

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

EA is the effective address

(b) Four computer instructions

Figure 9-5 Computer Instructions

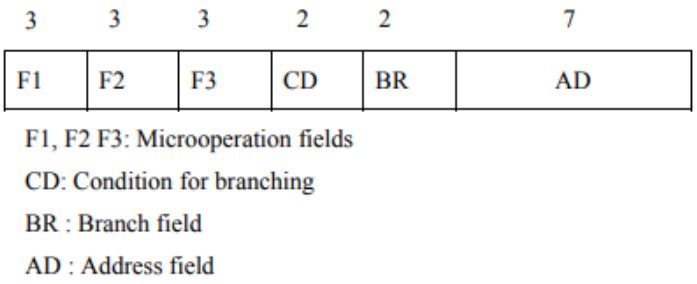


Figure 9-6 Microinstruction code format (20 bits)

The CD (condition) field consists of two bits which are encoded to specify four status bit conditions as listed in Table 7-1. The first condition is always a 1, so that a reference to CD = 00 (or the symbol U) will always find the condition to be true. When this condition is used in conjunction with the BR (branch) field, it provides an unconditional branch operation. The indirect bit I is available from bit 15 of DR after an instruction is read from memory. The sign bit of AC provides the next status bit. The zero value, symbolized by Z, is a binary variable whose value is 67 equal to 1 if all the bits in AC are equal to zero. We will use the symbols U, I, S, and Z for the four status bits when we write micro programs in symbolic form.

Table 9-1 Symbols and Binary Code for Microinstruction Fields

F1	Microoperation	Symbol
000	None	NOP
001	AC ← AC + DR	ADD
010	AC ← 0	CLRAC
011	AC ← AC + 1	INCAC
100	AC ← DR	DRTAC
101	AR ← DR(0-10)	DRTAR
110	AR ← PC	PCTAR
111	M [AR] ← DR	WRITE

F2	Microoperation	Symbol
000	None	NOP

001	AC ← AC – DR	SUB
010	AC ← AC ∨ DR	OR
011	AC ← AC ∧ DR	AND
100	DR ← M [AR]	READ
101	DR ← AC	ACTDR
110	DR ← DR + 1	INCDR
111	DR (0-10) ← PC	PCTDR

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

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 ← AD if condition = 1 CAR ← CAR + 1 if condition = 0
01	CALL	CAR ← AD, SBR ←CAR + 1 if condition = 1 CAR ← CAR + 1 if condition = 0
10	RET	CAR ← SBR (Return from subroutine)
11	MAP	CAR (2-5) ← DR (11-14), CAR (0,1,6) ← 0

The BR (branch) field consists of two bits. It is used, in conjunction with address field AD, to choose the address of the next microinstruction. As shown in Table 9-1, when BR = 00, the control performs a jump (JMP) operation (which is similar to a branch), and when BR = 01, it performs a call to subroutine (CALL) operation. The two operations are identical except that a call microinstruction stores the return address in the subroutine

register SBR. The jump and call operations depend on the value of the CD field. If the status bit condition specified in the CD field is equal to 1, the next address in the AD field is transferred to the control address register CAR. Otherwise, CAR is incremented by 1.

The return from subroutine is accomplished with a BR field equal to 10. This causes the transfer of the return address from SBR to CAR. The mapping from the operation code bits of the instruction to an address for CAR is accomplished when the BR field is equal to 11. This mapping is as depicted in Figure 9-3 the bits of the operation code are in DR after an instruction is read from memory. Note that the last two conditions in the BR field are independent of the values in the CD and AD fields.

9.3.3 SYMBOLIC MICROINSTRUCTIONS

The symbols defined in Table 9-1 can be used to specify microinstructions in symbolic form. A symbolic micro program can be translated into its binary equivalent by means of an assembler. A microprogram assembler is similar in concept to a conventional computer assembler. The simplest and most straightforward way to formulate an assembly language for a microprogram is to define symbols for each field of the microinstruction and to give users the capability for defining their own symbolic addresses.

Each line of the simply language micro program defines a symbolic microinstruction. Each symbolic microinstruction is divided into five fields: label, microoperation, CD, BR, and AD. The fields specify the following information.

1. The label field may be empty or it may specify a symbolic address. A label is terminated with a colon (:)
2. The microoperations field consists of one, two, or three symbols, separated by commas, from those defined in Table 9-1. There may be no more than one symbol from each F field. The NOP symbol is used when the microinstruction has no microoperations. This will be translated by the assembler to nine zeros.
3. The CD field has one of the letters U, I,S, or Z.
4. The BR field contains one of the four symbols defined in Table 9-1.
5. The AD field specifies a value for the address field of the microinstruction in one of three possible ways:
   a. With a symbolic address, which must also appear as a label.
   b. With the symbol NEXT to designate the next address in sequence.

c. When the BR field contains a RET or MAP symbol, the AD field is lift empty and is converted to seven zeros by the assembler.

We will use also the pseudo instruction ORG to define the origin, or first address, of a microprogram routine. Thus the symbol ORG 64 informs the assembler to place the next microinstruction in control memory at decimal address 64, which is equivalent to the binary address 1000000.

9.3.4 THE FETCH ROUTINE

The control memory has 128 words, and each word contains 20 bits. To microprogram the control memory, it is necessary to determine the bit values of each of the 128 words. The first 64 words (addresses 0 to 63) are to be occupied by the routines for the 16 instructions. The last 64 words may be used for any other purpose. A convenient starting location for the fetch routine is address 64. the microinstructions needed for the fetch routine are

AR ← PC DR ← M [AR],
PC ← PC + 1 AR ← DR (0-10),
CAR (2-5) ← DR (11-14), CAR 90,1,6)← 0

The address of the instruction is transferred from PC to AR and the instruction is then read from memory into DR. Since no instruction register is available, the instruction code remains in DR. The address part is transferred to AR and then control is transferred to one of 16 routines by mapping the operation code part of the instruction from DR into CAR.

The fetch routine needs three microinstructions, which are placed in control memory at addresses 64, 65, and 66. Using the assembly language conventions defined previously, we can write the symbolic microprogram for the fetch routine as follows:

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

The translation of the symbolic microprogram to biary produces the following binary microprogram. The bit values are obtained from Table 9-1.

The three microinstructions that constitute the fetch routine have been listed in three different representations. The register transfer representation shows the internal register transfer operations that each microinstruction implements. The symbolic representation is useful for writing microprograms in an assembly language format. The binary representation is the actual internal content that must be stored in control memory. It is customary to write microprograms in symbolic form and then use an assembler program to obtain a translation to binary.

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

9.3.5 SYMBOLIC MICROPROGRAM

The execution of the third (MAP) microinstruction in the fetch routine results in a branch to address 0××××0, where ×××× are the four bits of the operation code. For example, if the instruction is an ADD instruction whose operation code is 0000, the MAP microinstruction will 69 transfer to CAR the address 0000000, which is the start address for the ADD routine in control memory. The first address for the BRANCH and STORE routines are 0 0001 00 (decimal 4) and 0 0010 00 (decimal 8), respectively. The first address for the other 13 routines are at address values 12, 16, 20,…., 60. This gives four words in control memory for each routine.

In each routine we must provide microinstructions for evaluating the effective address and for executing the instruction. The indirect address mode is associated with all memory-reference instructions. A saving in the number of control memory words may be achieved if the microinstructions for the indirect address are stored as a subroutine. This subroutine, symbolized by INDRCT, is located right after the fetch routine, as shown in Table 9-2. the table also shows the symbolic microprogram for the fetch routine and the microinstruction in the ADD routine calls subroutine INDRCT, conditioned on status bit I. If I = 1 , a branch to INDRCT occurs and the return address (address 1 in this case) is stored in the subroutine register SBR. The INDRCT subroutine has two microinstructions:

INDRCT:	READ	U	JMP	NEXT
	KRTAR	U	RET	

Table 9.2 Symbolic Micro program

Label	Micro operations	CD	BR	AD
	ORG 0			
ADD	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ADD	U	JMP	FETCH
	ORG 4			
BRANCH:	NOP	S	JMP	OVER
	NOP	U	JMP	FETC
OVER:	NOP	I	CALL	INDRCT
	ARTPC	U	JMP	FETCH
	ORG 8			
STORE:	NOP	I	CALL	INDRCT
	ACTDR	U	JMP	NEXT
	WRITE	U	JMP	FETCH
	ORG 12			
EXCHANGE:	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ACTDR, DRTAC	U	JMP	NEXT
	WRITE	U	JMP	FETCH
	ORG 64			
FETCH:	PCTAR	U	JMP	NEXT
	READ, INCPC	U	JMP	NEXT

	DRTAR	U	MAP	
INDRCT:	READ	U	JMP	NEXT
	DRTAR	U	RET	

Remember that an indirect address considers the address part of the instruction as the address where the effective address is stored rather than the address of the operand. Therefore, the memory has to be accessed to get the effective address, which is then transferred to AR. The return from subroutine (RET) transfers the address from SBR to CAR, thus returning to the second microinstruction of the ADD routine.

The execution of the ADD instruction is carried out by the microinstruction at addresses 1 and 2. The first microinstruction reads the operand from memory into DR. the second microinstruction performs an add microoperation with the content of DR and AC and then jumps back to the beginning of the fetch routine.

The BRANCH instruction should cause a branch to the effective address if AC < 0. The AC will be less than zero if its sign is negative, which is detected from status bit S being a 1. The BRANCH routine in Table 9-2 starts by checking the value of S. If S is equal to 0, no branch occurs and the next microinstruction causes a jump back to the fetch routine without altering the content of PC. If S is equal to 1, the first JMP microinstruction transfers control to location OVER. The microinstruction at this location calls the INDRCT subroutine if I = 1. The effective address is then transferred from AR to PC and the microprogram jumps back to the fetch routine.

The STORE routine again uses the INDRCT subroutine if I = 1. The content of AC is transferred into DR. A memory write operation is initiated to store the content of DR in a location specified by the effective address in AR.

The EXCHANGE routine reads the operand from the effective address and places it in DR and AC are interchanged in the third microinstruction. This interchange is possible when the registers are of the edge-triggered type. The original content of AC that is now in now in DR is stored back in memory.

Note that Table 9-2 contains a partial list of the microprogram. Only four out of 16 possible computer instructions have been micro programmed. Also control memory words at locations 69 to 127 have not been used. Instructions such as multiply, divide, and others that require a long sequence of microoperations will need more than four

microinstructions for their execution. Control memory words 69 to 127 can be used for this purpose.

9.3.6 BINARY MICROPROGRAM

The symbolic microprogram is a convenient form for writing micro programs in a way that people can read and understand. But this is not the way that the microprogram is stored in memory. The symbolic microprogram must be translated to binary either by means of an assembler program or by the user if the microprogram is simple enough as in this example.

The equivalent binary form of the microprogram is listed in Table 9-3. The addresses for control memory are given in both decimal and binary. The binary content of each microinstruction is derived from the symbols and their equivalent binary values as defined in Table 9-1.

Note that address 3 has no equivalent in the symbolic microprogram since the ADD routine has only three microinstructions at addresses 0,1, and 2. The next routine starts at address 4. Even though address 3 is not used. Some binary value must be specified for each word in control memory. We could have specified all 0's in the word since this location will never be used. However, if some unforeseen error occurs, or if a noise signal sets CAR to the value of 3, it will be wise to jump to jump to address 64, which is the beginning of the fetch routine.

The binary micro program listed in Table 9-3 specifies the word content of the control memory. When a ROM is used for the control memory, the microprogram binary list provides the truth table for fabricating the unit. This fabrication is a hardware process and consists of creating a mask for the ROM so as to produce the 1's and 0's for each word. The bits of ROM are fixed once the internal links are fused during the hardware production. The ROM is made of IC packages that can be removed if necessary and replaced by other packages. To modify the instruction set of the computer, it is necessary to generate a new microprogram and mask a new ROM. The old one can be removed and the new one inserted in its place.

If a writable control memory is employed, the ROM is replaced by a RAM. The advantage of employing RAM for the control memory is that the microprogram can be altered simply by writing a new pattern of 1's and 0's without resorting to hardware procedures. A writable control memory possesses the flexibility of choosing the

instruction set a computer dynamically by changing the microprogram under processor control. However, most microprogrammed systems use a ROM for the control memory because it is cheaper and faster than a RAM and also to prevent the occasional user from changing the architecture of the system.

Micro Routine	Address		Binary Microinstruction					
	Decimal	Binary	F1	F2	F3	CD	BR	AD
ADD	0	0000000	000	000	000	01	01	1000011
	1	0000001	000	100	000	00	00	0000010
	2	0000010	001	000	000	00	00	1000000
	3	0000011	000	000	000	00	00	1000000
BRANCH	4	0000100	000	000	000	10	00	0000110
	5	0000101	000	000	000	00	00	1000000
	6	0000110	000	000	000	01	01	1000011
	7	0000111	000	000	110	00	00	1000000
STORE	8	0001000	000	000	000	01	01	1000011
	9	0001001	000	101	000	00	00	0001010
	10	0001010	111	000	000	00	00	1000000
	11	0001011	000	000	000	00	00	1000000
EXCHANGE	12	0001100	000	000	000	01	01	1000011
	13	0001101	001	000	000	00	00	0001110
	14	0001110	100	101	000	00	00	0001111
	15	0001111	111000	000	000	00	00	1000000
FETCH	64	1000000	110	000	000	00	00	1000001
	65	1000001	000	100	101	00	00	1000010
	66	1000010	101	000	000	00	11	0000000
INDRCT	67	1000011	000	100	000	00	00	1000100
	68	1000100	101	000	000	00	10	0000000

9.4 DESIGN OF CONTROL UNIT

The bits of the microinstruction are usually divided into fields, with each field defining a distinct, separate function. The various fields encountered in instruction formats provide

control bits to initiate microoperations in the system, special bits to specify the way that the next address is to be evaluated, and an address field for branching. The number of control bits that initiate micro operation can be reduced by grouping mutually exclusive variables into fields and encoding the k bits in each field to provide 2 microoperations. Each field requires a decoder to produce the corresponding control signals. This method reduces the size of the microinstruction bits but requires additional hardware external to the control memory. It also increases the delay time of the control signals because they must propagate through the decoding circuits.

The encoding of control bits was demonstrated in the programming example of the preceding section. The nine bits of the microoperation field are divided into three subfields of three bits each. The control memory output 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.

Figure 9-7 shows the three decoders and some of the connections that must be made from their outputs. Each of the three fields of the microinstruction presently available in the output of control memory are decoded with a 3 × 8 decoder to provide eight outputs. Each of these outputs must be connected to the proper circuit to initiate the corresponding microoperation as specified in Table 7-1. For example, when FI = 101 (binary 5), the next clock pulse transition transfers the content of DR (0- 10) to AR (symbolized by DRTAR in Table 7-1). Similarly, when F1 = 101 (binary 6) there is a transfer from PC to AR (symbolized by PCTAR). As shown in Fig. 9-7, 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 in inactive. The transfer into AR occurs with a clock pulse transition only when output 5 or output 6 of the decoder are active. The other outputs of the decoders that initiate transfers between registers must be connected in a similar fashion.

The arithmetic logic shift unit can be designed, instead of using gates to generate the control signals marked by the symbols AND, ADD, and DR in Fig 9.7, these inputs will now come from the outputs of the decoders associated with the symbols AND, ADD, and DRTAC, respectively as shown in Fig. 9-7. The other output of the decoders that are associated with an AC operation must also be connected to the arithmetic logic shift unit in a similar fashion.

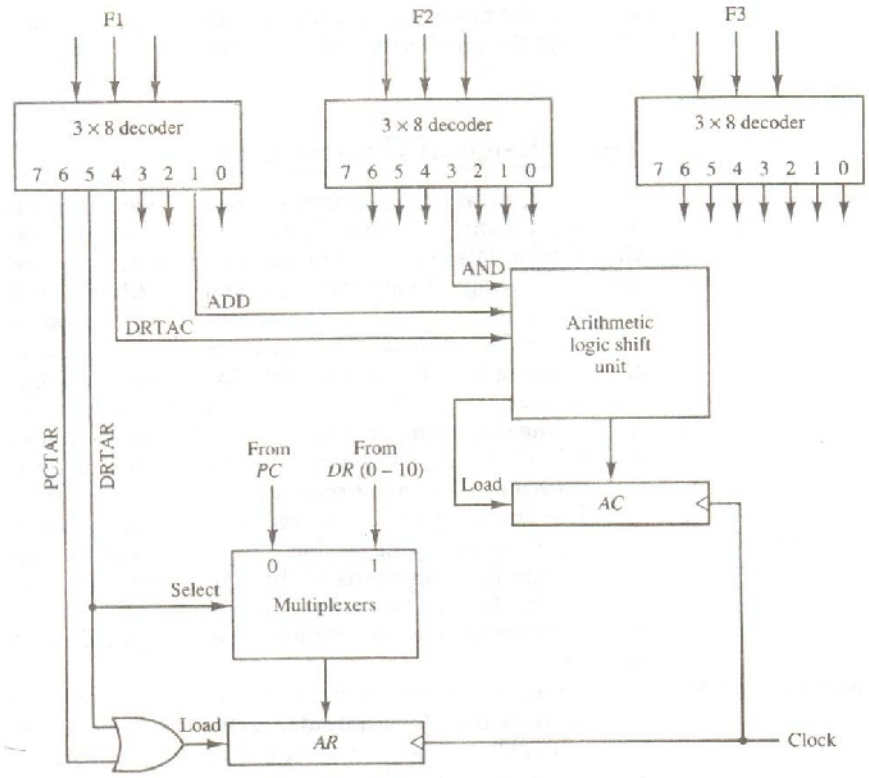


Figure 9-7 Decoding of microoperation fields

9.4.1 MICROPROGRAM SEQUENCER

The basic components of a micro programmed control unit are the control unit are the control memory and the circuits that select the next address. The address selection part is called a microprogram sequencer. A microprogram sequencer can be constructed with digital functions to suit a particular application. However, just as there are large ROM units available in integrated circuit packages, so are general purpose sequencers suited for the construction of microprogram control units. To guarantee a wide range of acceptability, an integrated circuit sequencer must provide an internal organization that can be adapted to a wide range of applications.

The purpose of a microprogram sequencer is to present 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 choice of the address source is guided by the next-address information bits that the sequencer receives from the present microinstruction. Commercial sequencers include within the unit an internal register stack used for

temporary storage of addresses during microprogram looping and subroutine calls; some sequencers provide an output register which can function as the address register for the control memory.

To illustrate the internal structure of a typical microprogram sequencer we will show a particular unit that is suitable for use in the microprogram computer example developed in the preceding section. The block diagram of the microprogram sequencer is shown in Fig. 9-8. 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 a 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 is incremented and applied to one of the multiplexer inputs and to the subroutine register SBR. The other three inputs to multiplexer number 1 come from the address field of the present microinstruction, from the output of SBR, and from an external source that maps the instruction. Although the diagram shows a single subroutine register, a typical sequencer will have a register stack about four to eight levels deep. In this way, a number of subroutines can be active at the same time. A push and pop operation, in conjunction with a stack pointer, stores and retrieves the return address during the call and return microinstructions.

The CD (condition) field of the microinstruction selecting one of the status bits in the second multiplexer. If the bit selected is equal to 1, the T (test) variable is equal to 1; otherwise, it is equal to 0. The T value together with the two bits from the BR (branch) field go to an input logic circuit.

The input logic in a particular sequencer will determine the type of operations that are available in the unit. Typical sequencer operations are: increment, branch or jump, call and return from subroutine, load an external address, push or pop the stack, and other address sequencing operations. With three inputs, the sequencer can provide up to eight address sequencing operations. Some commercial sequencers have three or four inputs in addition to the T input and thus provide a wider range of operations. The input logic circuit in Fig. 9-8 has three inputs, I_0, I_1, and T, and three outputs, S_0, S_1 and L. variables S_0 and S_1 select one of the source addresses for CAR. Variable L enables the load input in SBR. The binary values of the two selection variables determine the path in the multiplexer. For example, with S_1 S_0 = 10, multiplexer input number 2 is selected and

establishes a transfer path from SBR to CAR. Note that each of the four inputs as well as the output of MUX 1 contains a 7-bit address.

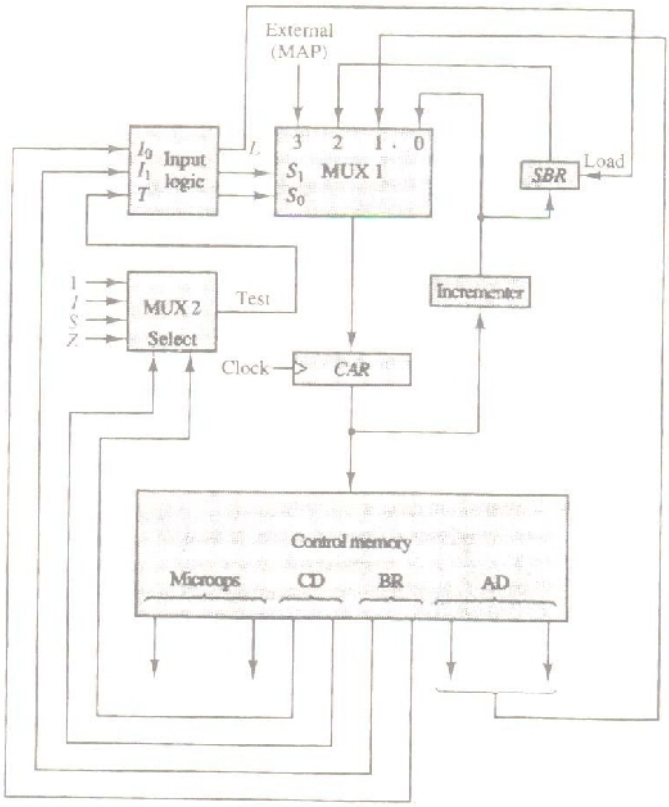


Figure 9-8 Microprogram sequencer for a control memory.

The truth table for the input logic circuit is shown in Table 9-4. inputs I_1 and I_0 are identical to the bit values in the BR field. The function listed in each entry was defined in Table 9-1. The bit values for S_1 and S_0 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:

$$S_1 = I_1$$

$$S_0 = I_1 I_0 + I'_1 T$$

$$L = I'_1 I_0 T$$

Table 9-4 Input Logic Truth Table for Microprogram Sequencer

BR Field		Input I_1	Input I_0	Input T	MUX 1 S_1	MUX 1 S_0	Load SBR L
0	0	0	0	0	0	0	0
0	0	0	0	1	0	1	0
0	1	0	1	0	0	0	0
0	1	0	1	1	0	1	1
1	0	1	0	×	1	0	0
1	1	1	1	×	1	1	0

The circuit cab be constructed with three AND gates, an OR gate, and an inverter. Note that the incremented circuit in the sequencer of Fig. 9-8 is not a counter constructed with flip-flops but rather a combinational circuit constructed with gates. A combinational circuit incrementer can be designed by cascading a series of half-adder circuits. The output carry from one state must be applied to the input of the next stage. One input in the first least significant stage must be equal to 1 to provide the increment-by-one operation.

9.5 CHECK YOUR PROGRESS

1. A word in control memory location is called microinstruction. (True / False)
2. A group of microinstructions constitute micro program. (True / False)
3. In dynamic microprogramming, the micro program can be initially loaded from disk. (True / False)
4. Each machine instruction initiate a microinstruction in control memory. (True / False)
5. The microinstruction specifies various internal control signals for execution of register micro operations. (True / False)
6. Each machine instruction has associated micro program in control memory. (True / False)
7. While the micro operations are being executed, the next address is computed in the next address generator circuit and then transferred into the control data register. (True / False)

9.6 SUMMARY

1. The function of the control unit in a digital computer is to initiate sequences of micro operations. The number of different types of micro operations that are available in a given system is finite.
2. The control function that specifies a micro operation is a binary variable. When it is in one binary state, the corresponding micro operation is executed.
3. A computer that employs a micro programmed control unit will have two separate memories: a main memory and a control memory.
4. Microinstructions are stored in control memory in groups, with each group specifying routine. Each computer instruction has its own micro program routine in control memory to generate the micro operations that execute the instruction.
5. The branch logic provides decision-making capabilities in the control unit.
6. A special type of branch exists when a microinstruction specifies a branch to the first word in control memory where a micro program routine for an instruction is located.
7. Subroutines are programs that are used by other routines to accomplish a particular task.
8. Once the configuration of a computer and its micro programmed control unit is established, the designer's task is to generate the microcode for the control memory.
9. Each line of the simply language micro program defines a symbolic microinstruction. Each symbolic microinstruction is divided into five fields: label, micro operation, CD, BR, and AD. The fields specify the following information.
10. The purpose of a micro program sequencer is to present is to present an address to the control memory so that a microinstruction may be read and executed.

9.7 KEYWORDS

1. A control unit whose binary control variables are stored in memory is called a **micro programmed control unit**.

2. A **control word** (CW) is a word whose individual bits represent the various control signals.
3. A **sequencer or microsequencer** generates the addresses used to step through the microprogram of a control store. It is used as a part of the control unit of a CPU or as a stand-alone generator for address ranges.
4. **Hardwire** is a function or device that is physically built into the computer instead of programmed into the software.

9.8 SELF ASSESSMENT TEST

1. What is the difference between a microprocessor and a microprogram? Is it possible to design a microprocessor without a microprogram? Are all micro programmed computers also micro processors?
2. Explain the difference between hardwired control and micro programmed control. Is it possible to have a hardwired control associated with a control memory?
3. Define the following:
   a) micro operation
   b) micro instruction
   c) micro program
   d) microcode
4. The micro programmed control organization showing in Fig. 9.1 has the following propagation delay times. 20 ns to generate the next address, 10 ns to transfer the address into the control address register, 20 ns to access the control memory ROM, 5 ns to transfer the microinstruction into the control data register, and 20 ns to perform the required microoperations specified by the control word. What is the maximum clock frequency that the control can use? What would the clock frequency be if the control data register is not used?
5. The system shown in Fig. 9.2 uses a control memory of 1024 words of 32 bits each. The microinstruction has three fields as shown in the diagram. The micro operations field has 16 bits.
   a) How many bits are there in the branch address field and the select field?
   b) If there are 16 status bits in the system, how many bits of the branch logic

areused to select a status bit?

c) How many bits are left to select an input for the multiplexers?

6. The control memory in Fig. 9.2 has 4096 words of 24 bits each.
   a) How many bits are there in the control address register?
   b) How many bits are there in each of the four inputs shown going into the multiplexers?
   c) What are the number of inputs in each multiplier and how many multiplexers are needed?
7. Using the mapping procedure described in Fig. 9.3, give the first microinstruction address for the following operation code:
   (a) 0010
   (b) 1011
   (c) 1111
8. Formulate a mapping procedure that provides eight consecutive microinstructions for each routine. The operation code has six bits and the control memory has 2048 words.
9. Explain how the mapping from an instruction code to a microinstruction address can be done by means of a read-only memory. What is the advantage of this method compared to the one in Fig. 9-3?
10. Why do we need the two multiplexers in the computer hardware configuration showing in Fig. 9.4? Is there another way that information from multiple sources can be transferred to a common destination?

9.9 ANSWER TO CHECK YOUR PROGESS

1. True
2. False because a sequence of microinstructions constitute a microprogram.
3. True
4. False because it initiates a series of microinstructions.
5. True
6. True
7. False

9.10 REFERENCES / SUGGESTED READINGS

1. Computer Organization and Architecture, Rajaram & Radhakrishan, PHI.
2. Computer Organization & Architecture: Designing for Performance, Stalling, PHI.
3. Computer Organization and Design, Pal Choudhary, PHI.
4. Computer Systems Organization & Architecture, Carpenelli, Pearson Education.
5. Computer Organization and Architecture, Stalling, Pearson Education.
6. Computer System Architecture, Morris Mano, PHI.
7. Computer Architecture and Organization, McGraw Hill Company, New Delhi. J.P. Hayes.

SUBJECT: COMPUTER SYSTEM ARCHITECTURE	
COURSE CODE: MCA-24	**AUTHOR: MR. NEERAJ VERMA**
LESSON NO. 10	

INTRODUCTION TO PARALLELISM

STRUCTURE

10.0 LEARNING OBJECTIVES

The main objective of this lesson is to introduce students about Instruction level parallelism and Processor level parallelism. Students will be able to learn about basic features of pipelining, super scaling in Instruction level parallelism and multiprocessor systems in Processor level parallelism. Cycle time, latency and throughput of pipelined processors are also discussed in this lesson.

10.1 INTRODUCTION

The speed of execution of programs is influenced by many factors. One way to improve performance is to use faster circuit technology to build the processor and the main memory. Another possibility is to arrange the hardware so that more than one operation can be performed at the same time. In this way, the number of operations performed per second is increased even though the elapsed time needed to perform any one operation is not changed.

In order to speed up the operation of a computer system beyond what is possible with sequential execution, methods must be found to perform more than one task at a time. One method for gaining significant speedup with modest hardware cost is the technique of pipelining. In this technique, a task is broken down into multiple steps, and independent processing units are assigned to each step. Once a task has completed its initial step, another task may enter that step while the original task moves on to the following step. Any operation that can be decomposed into a sequence of sub operations of about the same complexity can be implemented by a pipeline processor. The technique in efficient for those applications that need to repeat the same task many times with different sets of data. The process is much like an assembly line, with a different task in progress at each stage. In theory, a pipeline which breaks a process into N steps could achieve an N-fold increase in processing speed. Due to various practical problems, the actual gain may be significantly less.

Pipelining is most suited for tasks in which essentially the same sequence of steps must be repeated many times for different data. This is true, for example, in many numerical problems which systematically process data from arrays. Arithmetic pipelining is used in some specialized computers discussed elsewhere. One action

common to all computers, however, is the systematic fetch and execute of instructions. A superscalar processor is one in which multiple independent instruction pipelines are used. Each pipeline consists of multiple stages, so that each pipeline can handle multiple instructions at a time. Multiple pipelines introduce a new level of parallelism, enabling multiple streams of instructions to be processed at a time. A superscalar processor exploits what is known as instruction-level parallelism, which refers to the degree to which the instructions of a program can be executed in parallel.

10.2 CPU PIPELINING

Pipelining is an effective way of managing parallel activity in a computer system. The basic idea is very simple. It is normally encounter in manufacturing plants, where pipelining is commonly known as an assembly-line operation. Students are certainly familiar with the assembly line used in car manufacturing. The first station in an assembly line may prepare the chassis of a car, the next station adds the body, the next one installs the engine, and so on. While one group of workers is installing the engine on one car, another group is fitting a car body on the chassis of another car, and yet another group is preparing a new chassis for a third car. It may take days to complete work on a given car, but it is possible to have a new car rolling off the end of the assembly line every few minutes.

Pipelining is a technique of decomposing a sequential process into sub operations, with each sub process being executed in a special dedicated segment that operates concurrently with all other segments. A pipeline can be imagined as a collection of processing segments through which binary information flows. Each segment performs partial processing dictated by the way the task is partitioned. The result obtained from the computation in each segment is transferred to the next segment in the pipeline. The final result is obtained after the data have passed through all segments.

Pipelining is an implementation technique where multiple instructions are overlapped in execution. This is solved without additional hardware but only by leasing different parts of the hardware work for different instructions at the same time. The pipeline organization of a CPU is similar to an assembly line: the work to be done in an instruction is broken into smaller steps (pieces), each of which takes a fraction of the time needed to complete the entire instruction. Each of these steps is a *pipe stage* (or a *pipe segment*). Pipe stages are connected to form a pipe as shown in figure 10.1:

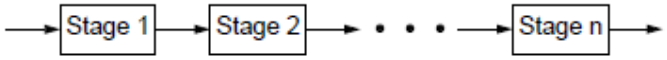


Figure 10.1 Pipe Stages

Instruction execution is extremely complex and involves several operations which are executed successively as shown in figure 10.2. This involves a large amount of hardware, but only one part of this hardware works at a given moment.

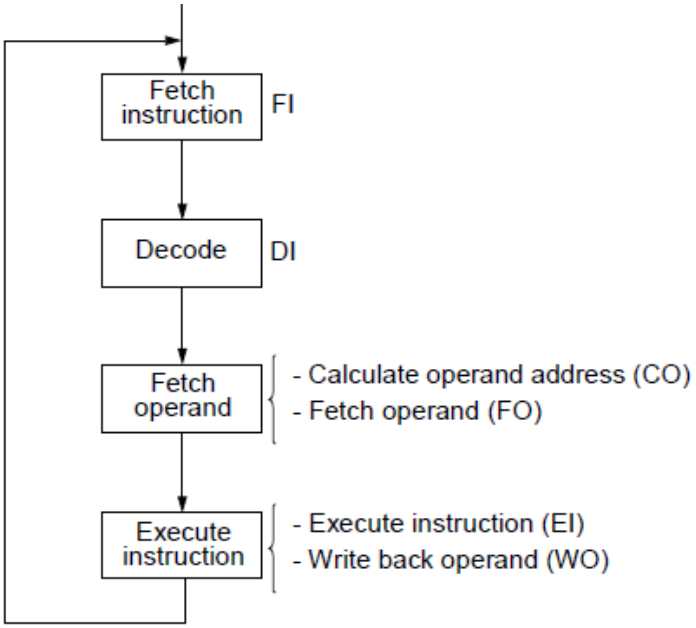


Figure 10.2 Instruction Cycle

The processor executes a program by fetching and executing instructions, one after the other. Let F*i* and E*i* refer to the fetch and execute steps for instruction I*i* . Executions of a program consist of a sequence of fetch and execute steps, as shown in Figure 10.3

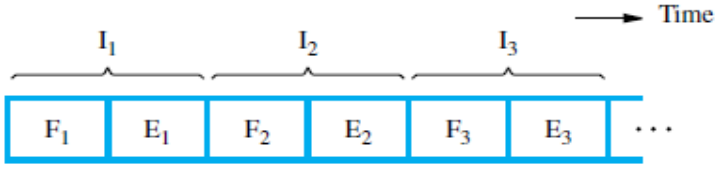


Figure 10.3 Sequential Execution

Now consider a computer that has two separate hardware units, one for fetching instructions and another for executing them, as shown in Figure 10.4. The instruction fetched by the fetch unit is deposited in an intermediate storage buffer, B1. This

buffer is needed to enable the execution unit to execute the instruction while the fetch unit is fetching the next instruction. The results of execution are deposited in the destination location specified by the instruction. For the purposes of this discussion, we assume that both the source and the destination of the data operated on by the instructions are inside the block labeled "Execution unit."

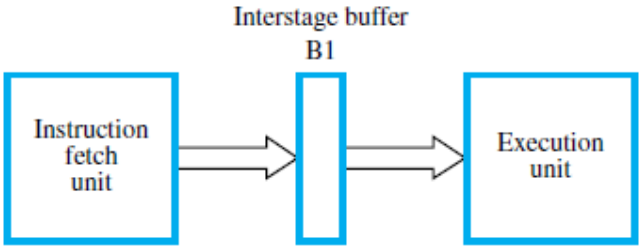


Figure 10.4 Hardware Organization

The computer is controlled by a clock whose period is such that the fetch and execute steps of any instruction can each be completed in one clock cycle. Operation of the computer proceeds as in Figure 10.5.

In the first clock cycle, the fetch unit fetches an instruction I1 (step F1) and stores it in buffer B1 at the end of the clock cycle. In the second clock cycle, the instruction fetch unit proceeds with the fetch operation for instruction I2 (step F2). Meanwhile, the execution unit performs the operation specified by instruction I1, which is available to it in buffer B1 (step E1). By the end of the second clock cycle, the execution of instruction I1 is completed and instruction I2 is available.

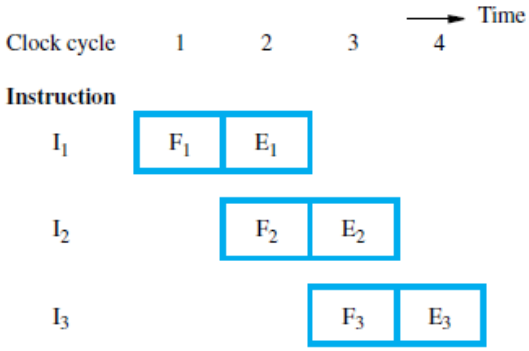


Figure 10.5 Pipelined Executions

Instruction I2 is stored in B1, replacing I1, which is no longer needed. Step E2 is performed by the execution unit during the third clock cycle, while instruction I3 is

being fetched by the fetch unit. In this manner, both the fetch and execute units are kept busy all the time. If the pattern in Figure 10.5 can be sustained for a long time, the completion rate of instruction execution will be twice that achievable by the sequential operation depicted in Figure 10.3. In summary, the fetch and execute units in Figure 10.4 constitute a two-stage pipeline in which each stage performs one step in processing an instruction. An interstate storage buffer, B1, is needed to hold the information being passed from one stage to the next. New information is loaded into this buffer at the end of each clock cycle. The processing of an instruction need not be divided into only two steps. For example, a pipelined processor may process each instruction in four steps, as follows:

F Fetch: read the instruction from the memory.

D Decode: decode the instruction and fetch the source operand(s).

E Execute: perform the operation specified by the instruction.

W Write: store the result in the destination location.

The sequence of events for this case is shown in Figure 10.6.

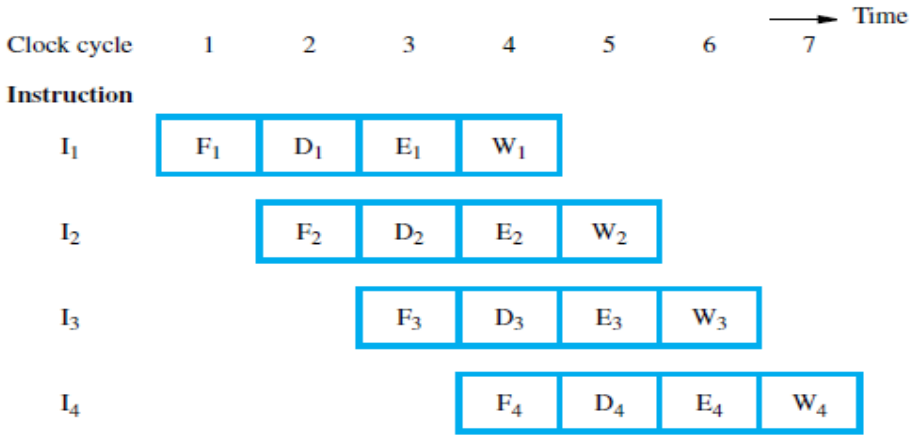


Figure 10.6 Instruction Execution Divided into Four Steps

Four instructions are in progress at any given time. This means that four distinct hardware units are needed, as shown in Figure 10.7.

These units must be capable of performing their tasks simultaneously and without interfering with one another. Information is passed from one unit to the next through a storage buffer. As an instruction progresses through the pipeline, all the information needed by the stages downstream must be passed along. For example, during clock cycle 4, the information in the buffers is as follows:

- Buffer B1 holds instruction I3, which was fetched in cycle 3 and is being decoded by the instruction decoding unit.
- Buffer B2 holds both the source operands for instruction I2 and the specification of the operation to be performed. This is the information produced by the decoding hardware in cycle 3. The buffer also holds the information needed for the write step of instruction I2 (stepW2). Even though it is not needed by stage E, this information must be passed on to stage W in the following clock cycle to enable that stage to perform the required Write operation.
- Buffer B3 holds the results produced by the execution unit and the destination information for instruction I1.

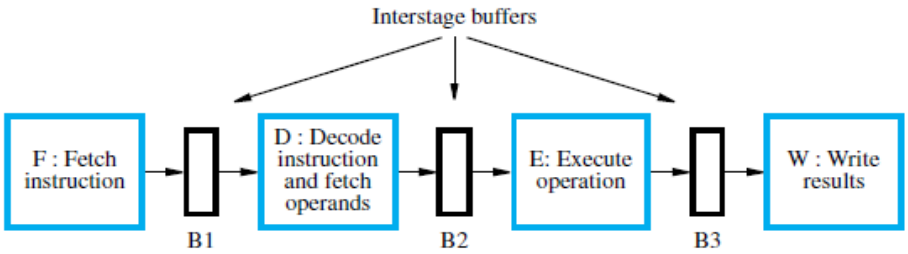


Figure 10.7 Hardware Organization

Let us consider an example of pipeline where each segment consists of an input register followed by an combinational circuit. The register holds the data and combinational circuit performs the sub-operation in a particular segment. The output of combinational circuit in a given segment is applied to the input register to the next segment. A clock is applied to all registers after enough time has elapsed to perform all segment activity. The information flows through the pipeline one step at a time. The pipeline organization will be demonstrated by means of a simple example. Suppose that we want to perform the combined multiply and add operation with stream of numbers.

$$A_i * B_i + C_i \quad \text{for } i = 1, 2, 3, \ldots, 7$$

Each sub operation is to be implemented in a segment within a pipeline. Each segment has one or two registers and a combinational circuit as shown in figure 10.3.8. R1 through R5 are registers that receive new data with every clock pulse. The multiplier

and adder are combinational circuits. The sub operations performed in each segment of the pipeline are as follows:

$R1 \leftarrow A_i$, $R2 \leftarrow B_i$	Input A_i and B_i
$R3 \leftarrow R1 * R2$, $R4 \leftarrow C_i$	Multiply and input C_i
$R5 \leftarrow R3 + R4$	Add C_i to product

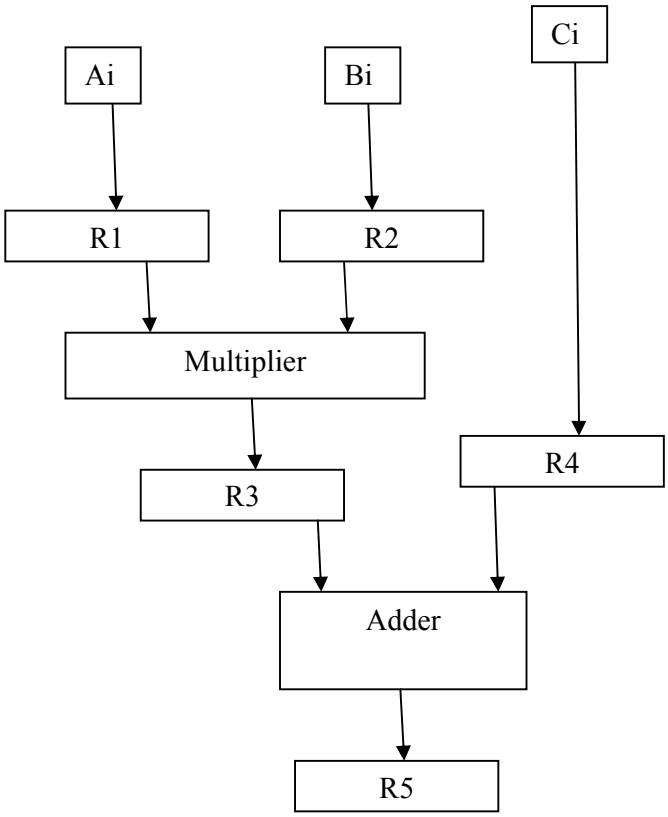


Figure 10.8 Example of Pipeline Processing

The five registers are loaded with new data every clock pulse. The effect of each clock is shown in Table 10.1.

Table 10.1 Content of registers in pipeline example.

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	$R1$	$R2$	$R3$	$R4$	$R5$
1	A_1	B_1	—	—	—
2	A_2	B_2	$A_1 * B_1$	C_1	—
3	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
5	A_5	B_5	$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$
6	A_6	B_6	$A_5 * B_5$	C_5	$A_4 * B_4 + C_4$
7	A_7	B_7	$A_6 * B_6$	C_6	$A_5 * B_5 + C_5$
8	—	—	$A_7 * B_7$	C_7	$A_6 * B_6 + C_6$
9	—	—	—	—	$A_7 * B_7 + C_7$

Four segment pipeline

The general structure of a four-segment pipeline is illustrated in Figure 10.9. The operands pass through all four segments in a fixed sequence. Each segment consists of a combinational circuit S_i that performs a sub operation over the data stream flowing through the pipe.

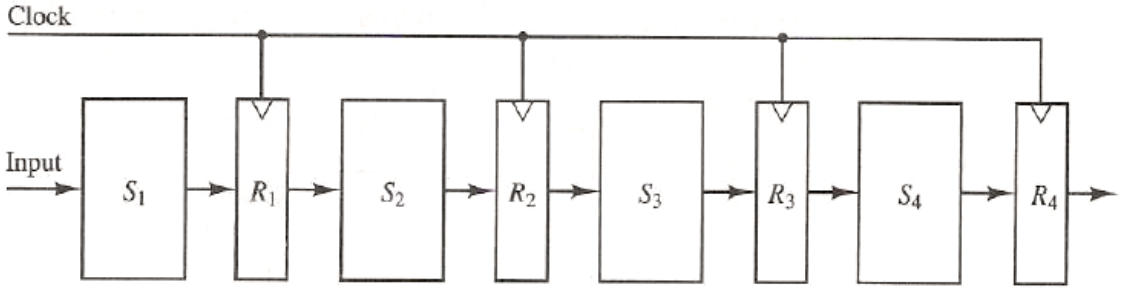


Figure 10.9 Four-segment pipeline

The segments are separated by registers R_i that hold the intermediate results between the stages. Information flows between adjacent stages under the control of a common clock applied to all the registers simultaneously. We define a task as the total operation performed going through all the segments in the pipeline.

The behavior of a pipeline can be illustrated with a space-time diagram. This is a diagram that shows the segment utilization as a function of time. The space-time diagram of a four-segment pipeline is demonstrated in Figure 10.10.

The horizontal axis displays the time in clock cycles and the vertical axis gives the segment number. The diagram shows six tasks T_1 through T_6 executed in four segments. Initially, task T is handled by segment 1. After the first clock, segment 2 is

busy with T, while segment 1 is busy with task T. Continuing in this manner, the first task T is completed after the fourth clock cycle. From then on, the pipe completes a task every clock cycle. No matter how many segments there are in the system. Once the pipeline is full, it takes only one clock period to obtain an output.

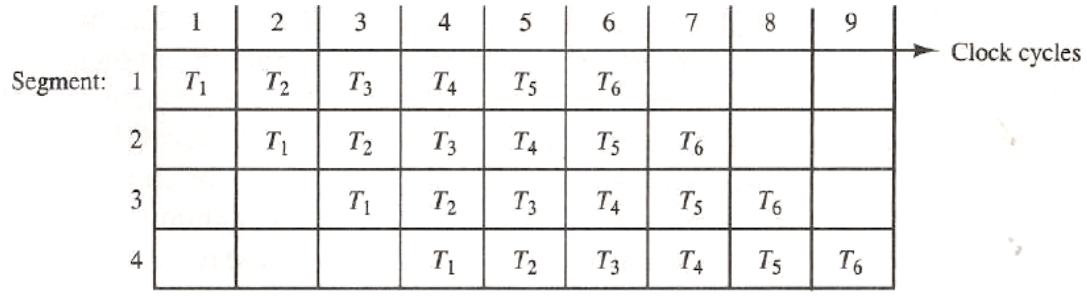


Figure 10.10 Space-time diagram for pipeline

Now consider the case where a k-segment pipeline with a clock cycle time t_p is used to execute n tasks. The first task T_1 requires a time equal to kt_p to complete its operation since there are k segments in the pipe. The remaining n-1 tasks emerge from the pipe at the rate of one task per clock cycle and they will be completed after a time equal to $(n-1)t_p$. Therefore, to complete n tasks using a k-segment pipeline requires k + (n-1) clock cycles. For example, the diagram of Figure 10.3.10 shows four segments and six tasks. The time required to complete all the operation is 4 + (6-1) = 9 clock cycles, as indicated in the diagram.

Next consider a nonpipeline unit that platform the same operation and takes a time equal to t_n to complete each task. The total time required for n tasks is nt_n. The speedup of a pipeline processing over an equivalent non-pipeline processing is defined by the ratio:

$$S = \frac{nt_n}{(k + n - 1)t_p}$$

As the number of tasks increases, n becomes much larger than k -1, and k + n -1 approaches the value of n. Under this condition, the speedup becomes

$$S = \frac{t_n}{t_p}$$

If we assume that the time it takes to process a task is the same in the pipeline and non-pipeline circuits, we will have $t_n = kt_p$. Including this assumption, the speedup reduces to

$$S = \frac{kt_p}{t_p} = k$$

This shows that the theoretical maximum speed up that a pipeline can provide is k, where k is the number of segments in the pipeline.

To clarify the meaning of the speedup ratio, consider the following numerical example. Let the time it takes to process a sub operation in each segment be equal to $t_p = 20$ ns. Assume that the pipeline has k = 4 segments and executes n = 100 tasks in sequence. The pipeline system will take $(k + n - 1)t_p = (4 + 99) \times 20 = 2060$ ns to complete. Assuming that $t_n = kt_p = 4 \times 20 = 80$ ns, a non-pipeline system requires $nkt_p = 100 \times 80 = 8000$ ns to complete the 100 tasks. The speedup ratio is equal to 8000/2060 = 3.88. as the number of tasks increases, the speedup will approach 4, which is equal to the number of segments in the pipeline. If we assume that $t_n = 60$ ns, the speedup becomes 60/20 = 3.

To duplicate the theoretical speed advantage of a pipeline process by means of multiple functional units, it is necessary to construct k identical units that will be operating in parallel. The implication is that a k-segment pipeline processor can be expected to equal the performance of k copies of an equivalent non-pipeline circuit under equal operating conditions. This is illustrated in Figure 10.11, where four identical circuits are connected in parallel.

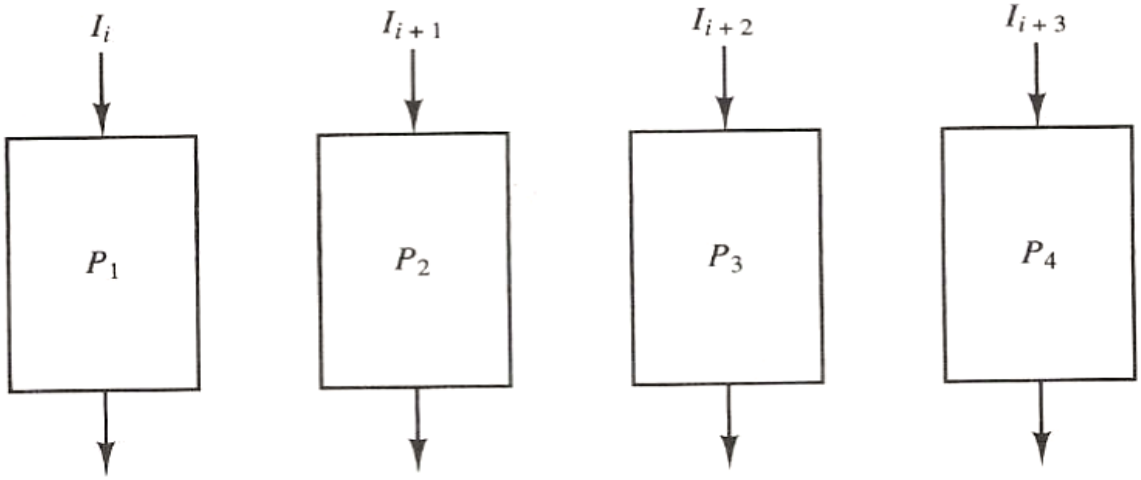


Figure 10.11 Multiple functional units in parallel

Each P circuit performs the same task of an equivalent pipeline circuit. Instead of operating with the input data in sequence as in pipeline, the parallel circuits accept four input data items simultaneously and perform four tasks at the same time. As far as the speed of operation is concerned, this is equivalent to a four segment pipeline. Note that the four-unit circuit of Figure 10.11 constitutes a single-instruction multiple-data (SIMD) organization since the same instruction is used to operate on multiple data in parallel.

There are various reasons why the pipeline cannot operate at its maximum theoretical rate. Different segments may take different times to complete their sub operation. The clock cycle must be chosen to equal the time delay of the segment with the maximum propagation time. This causes all other segments to waste time while waiting for the next clock. Moreover, it is not always correct to assume that a non pipeline circuit has the same time delay as that of an equivalent pipeline circuit. Many of the intermediate registers will not be needed in a single-unit circuit, which can usually be constructed entirely as a combinational circuit. Nevertheless, the pipeline technique provides a faster operation over purely serial sequences even through the maximum theoretical speed is never fully achieved.

There are two areas of computer design where the pipeline organization is applicable. An arithmetic pipeline divides an arithmetic operation into sub operations for execution in the pipeline segments. An instructions pipeline operates on a stream of instructions by overlapping the fetch, decode, and execute phases of the instruction cycle. The two types of pipelines are explained in the following sections.

Arithmetic Pipeline

Pipeline arithmetic units are usually found in very high speed computers. They are used to implement floating-point operations, multiplication of fixed-point numbers, and similar computations encountered in scientific problems. We will now show an example of a pipeline unit for floating-point addition and subtraction.

The inputs to the floating-point adder pipeline are two normalized floating-point binary numbers.

$$X = A \times 2^a$$
$$Y = B \times 2^b$$

A and B are two fractions that represent the mantissas and a and b are the exponents.

The floating-point addition and subtraction can be performed in four segments.

The registers labeled R are placed between the segments to store intermediate results.

The sub operations that are performed in the four segments are:

1. Compare the exponents.
2. Align the mantissas.
3. Add or subtract the mantissas.
4. Normalize the result.

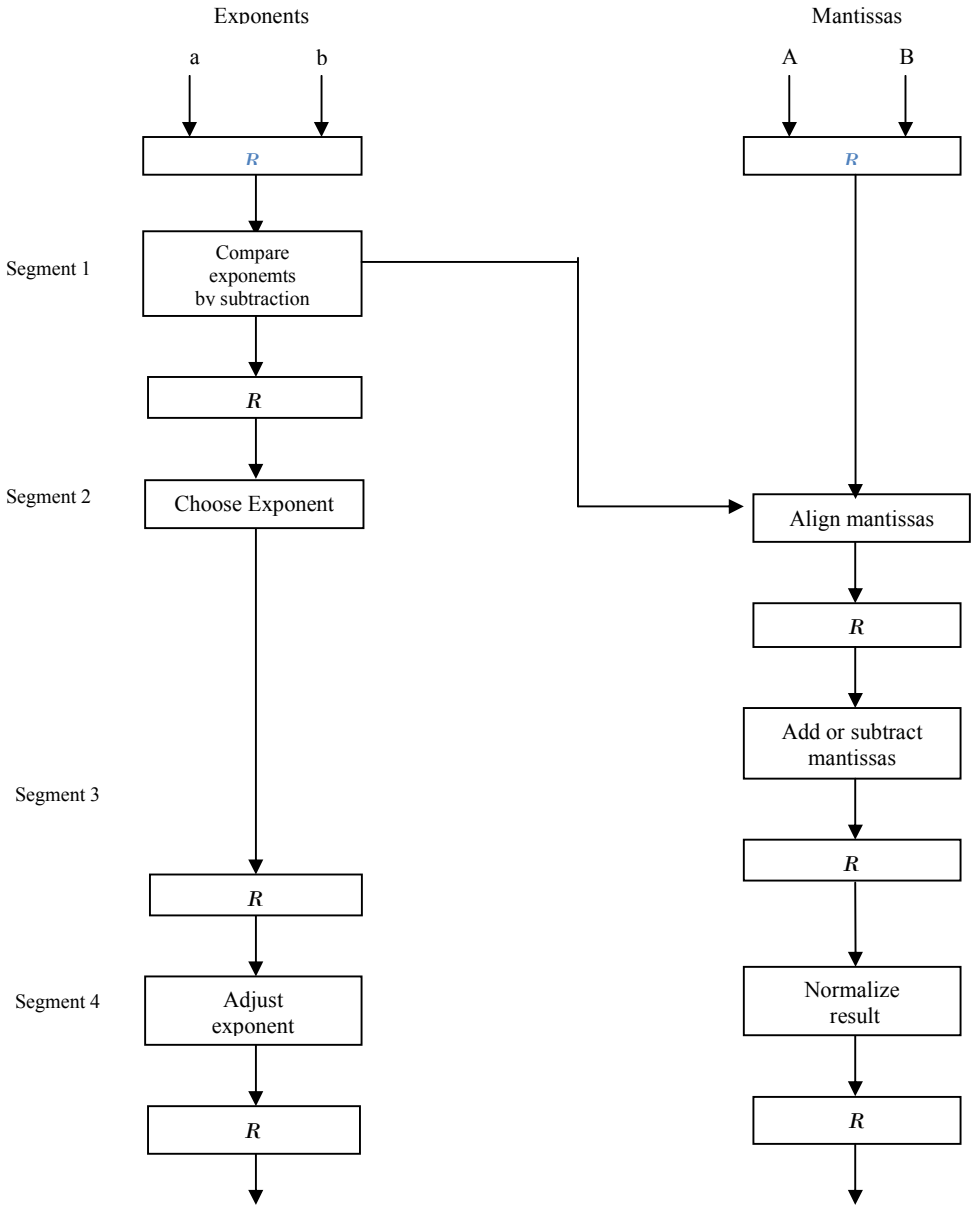


Figure 10.12 Pipeline for floating-point addition and subtraction.

The exponents are compared by subtracting them to determine their difference. The larger exponent is chosen as the exponent of the result. The exponent difference determine how many times the mantissa associated with the smaller exponent must be shifted to the right. This produces an alignment of the two mantissas. It should be noted that the shift must be designed as a combinational circuit to reduce the shift time. The two mantissa are added or subtracted in segment 3. The result is normalized in segment 4. When an overflow occurs, the mantissa of the sum or difference is shifted right and the exponent incremented by one. If an underflow occurs, the number of leading zeros in the mantissa determines the number of left shifts in the mantissa and the number that must be subtracted from the exponent.

The following numerical example may clarify the sub operations performed in each segment. For simplicity, we use decimal numbers, although Figure 10.12 refers to binary numbers. Consider the two normalized floating-point numbers:

$$X = 0.9504 \times 10^3$$

$$Y = 0.8200 \times 10^2$$

The two exponents are subtracted in the first segment to obtain 3 – 2 = 1. The larger exponent 3 is chosen as the exponent of the result. The next segment shifts the mantissa of Y to the right to obtain.

$$X = 0.9504 \times 10^3$$

$$Y = 0.0820 \times 10^3$$

This aligns two mantissa under the same exponent. The addition of two mantissa in segment 3 produces the sum

$$Z = 1.0324 \times 10^3$$

The sum is adjusted by normalizing the result so that it has a fraction with a nonzero first digit. This is done by shifting the mantissa once to the right and incrementing the exponent by one to obtain the normalized sum.

$$Z = 0.10324 \times 10^4$$

The comparator, shifter, adder-subtractor, incrementer, and decrementer in the floating-point pipeline are implemented with combinational circuits. Suppose that the time delays of the four segments are $t_1 = 60ns$, $t_2 = 70ns$, $t_3 = 100ns$, $t_4 = 80ns$, and the interface registers have a delay of $t_r = 10ns$. The clock cycle is chosen to be $t_p = t_3 + t_r = 110ns$. An equivalent nonpipeline floating-point adder-subtractor will have a delay

time $t_n = t_1 + t_2 + t_3 + t_4 + t_r = 320$ ns. In this case the pipelined adder has a speedup of 320/110 = 209 over the nonpipelined adder.

Instruction Pipeline

Pipeline processing can occur not only in the data stream but in the instruction stream as well. An instruction pipeline reads consecutive instructions from memory while previous instructions are being executed in other segments. This causes the instructions fetch and execute phases to overlap and perform simultaneous operations. One possible digression associated with such a scheme is that an instruction may cause a branch out of sequence. In that case the pipeline must be emptied and all the instructions that have been read from memory after the branch instruction must be discarded.

Consider a computer with an instruction fetch unit and an instruction execution unit designed to provide a two segment pipeline. The instruction fetch segment can be implemented by means of a first-in, first out (FIFO) the execution unit is not using memory, the control increments the program counter and uses its address value to read consecutive instructions from memory. The instructions are inserted into the FIFO buffer so that they can be executed on a first-in, first-out basis. Thus an instruction stream can be placed in a queue, waiting for decoding and processing by the execution segment. The instruction stream queuing mechanism provides an efficient way for reducing the average access time to memory for reading instructions. Whenever there is space in the FIFO buffer, the control unit initiates the next instruction fetch phase. The buffer acts as a queue from which control then extracts the instructions for the execution unit.

Computers with complex instructions require other phases in addition to the fetch and execute to process an instruction completely. In the most general case, the computer needs to process each instruction with the following sequences of steps.

1. Fetch the instruction from memory.
2. Decode the instruction.
3. Calculate the effective address.
4. Fetch the operands from memory.
5. Execute the instruction.
6. Store the result in the proper place.

There are certain difficulties that will prevent the instruction pipeline from operating at its maximum rate. Different segments may take different times to operate on the incoming information. Some segments are skipped for certain operations. For example, a register mode instruction does not need an effective address calculation. Two or more segments may require memory access at the same time causing one segment to wait until another is finished with the memory. Memory access conflicts are sometimes resolved by using two memory buses for accessing instructions and data in separate modules. In this way, an instruction word and a data word can be read simultaneously from two different modules.

The design of an instruction pipeline will be most efficient if the instruction cycle is divided into segments of equal duration. The time that each step takes to fulfill its function depends on the instruction and the way it is executed.

Example: Four-Segment Instruction Pipeline

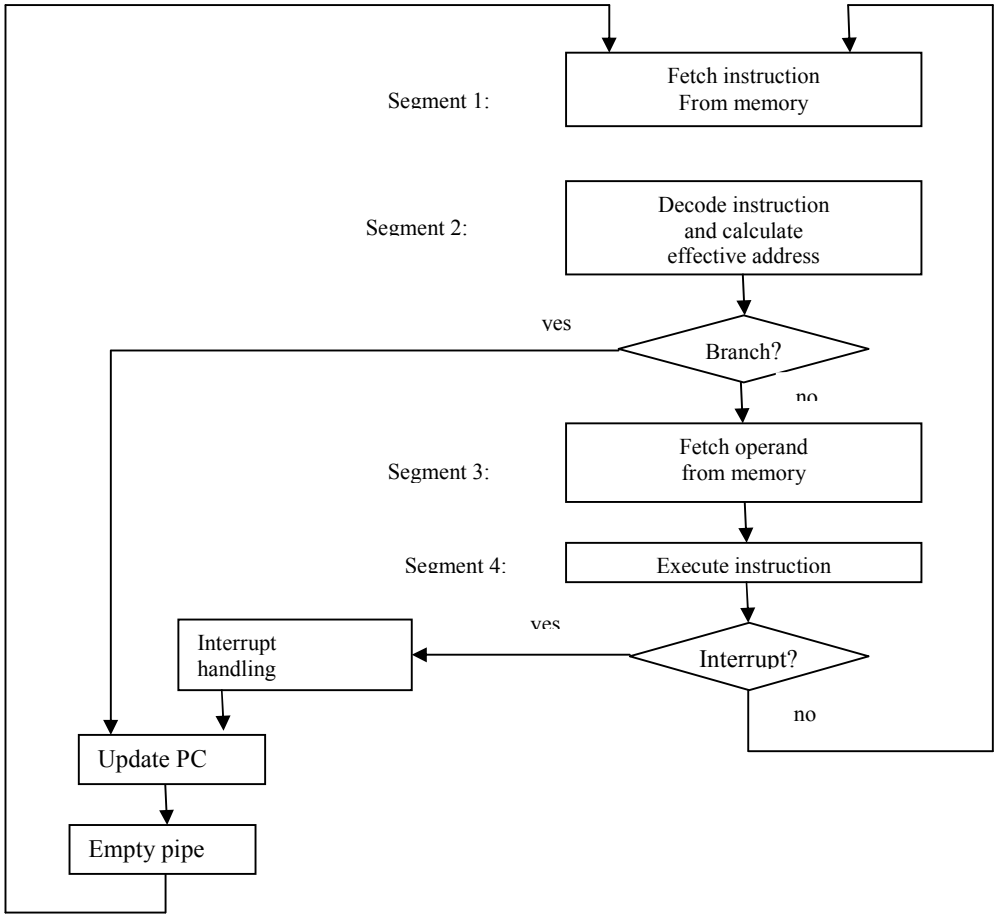


Figure 10.13 Four-segment CPU pipeline.

Assume that the decoding of the instruction can be combined with the calculation of the effective address into one segment. Assume further that most of the instructions place the result into a processor register so that the instruction execution and storing of the result can be combined into one segment. This reduces the instruction pipeline into four segments.

Figure 10.13 shows how the instruction cycle in the CPU can be processed with a four-segment pipeline. While an instruction is being executed in segment 4, the next instruction in sequence is busy fetching an operand from memory in segment 3. The effective address may be calculated in a separate arithmetic circuit for the third instruction, and whenever the memory is available, the fourth and all subsequent instructions can be fetched and placed in an instruction FIFO. Thus up to four sub operations in the instruction cycle can overlap and up to four different instructions can be in progress of being processed at the same time.

Once in a while, an instruction in the sequence may be a program control type that causes a branch out of normal sequence. In that case the pending operations in the last two segments are completed and all information stored in the instruction buffer is deleted. The pipeline then restarts from the new address stored in the program counter. Similarly, an interrupt request, when acknowledged, will cause the pipeline to empty and start again from a new address value. Figure 10.14 shows the operation of the instruction pipeline. The time in the horizontal axis is divided into steps of equal duration. The four segments are represented in the diagram with an abbreviated symbol.

1. FI is the segment that fetches an instruction.
2. DA is the segment that decodes the instruction and calculates the effective address.
3. FO is the segment that fetches the operand.
4. EX is the segment that executes the instructions.

It is assumed that the processor has separate instruction and data memories so that the operation in FI and FO can proceed at the same time. In the absence of a branch instruction, each segment operates on different instructions. Thus, in step 4, instruction 1 is being executed in segment EX; the operand for instruction 2 is being fetched in segment FO; instruction 3 is being decoded in segment DA; and instruction 4 is being fetched from memory is segment FI.

Step:		1	2	3	4	5	6	7	8	9	10	11	12	13
Instruction:	1	FI	DA	FO	EX									
	2		FI	DA	FO	EX								
(Branch)	3			FI	DA	FO	EX							
	4				FI	–	–	FI	DA	FO	EX			
	5					–	–	–	FI	DA	FO	EX		
	6									FI	DA	FO	EX	
	7										FI	DA	FO	EX

Figure 10.14 Timing of instruction pipeline

Assume now that instruction 3 is a branch instruction. As soon as this instruction is decoded in segment DA in step 4, the transfer from FI to DA of the other instruction is halted until the branch instruction is executed in step 6. If the branch is taken, a new instruction is not taken, the instruction fetched previously in step 4 can be used. The pipeline then continues until a new branch instruction is encountered.

Another delay may occur in the pipeline if the EX segment needs to store the result of the operation in the data memory while the FO segment needs to fetch an operand. In that case, segment FO must wait until segment EX has finished its operation.

In general, there are three major difficulties that cause the instruction pipeline to deviate from its normal operation.

1. Resource conflicts caused by access to memory by two segments at the same time. Most of these conflicts can be resolved by using separate instruction and data memories.
2. Data dependency conflicts arise when an instruction depends on the result of a previous instruction, but this result is not yet available.
3. Branch difficulties arise from branch and other instructions that change the value of PC.

Data Dependency

A difficulty that may caused a degradation of performance in an instruction pipeline is due to possible collision of data or address. A collision occurs when an instruction cannot proceed because previous instructions did not complete certain operations. A

data dependency occurs when an instruction needs data that are not yet available. For example, an instruction in the FO segment may need to fetch an operand that is being generated at the same time by the previous instruction in segment EX. Therefore, the second instruction must wait for data to become available by the first instruction. Similarly, an address dependency may occur when an operand address cannot be calculated because the information needed by the addressing mode is not available. For example, an instruction with register indirect mode cannot proceed to fetch the operand if the previous instruction is loading the address into the register. Therefore, the operand access to memory must be delayed until the required address is available. Pipelined computers deal with such conflicts between data dependencies in a variety of ways.

The most straightforward method is to insert hardware interlocks. An interlock is a circuit that detects instructions whose source operands are destinations of instructions farther up in the pipeline. Detection of this situation causes the instruction whose source is not available to be delayed by enough clock cycles to resolve the conflict. This approach maintains the program sequence by using hardware to insert the required delays.

Another technique called operand forwarding uses special hardware to detect a conflict and then avoid it by routing the data through special paths into a destination register, the hardware checks the destination operand, and if it is needed as a source in the next instruction, it process the result directly into the ALU input, bypassing the register file. This method requires additional hardware paths through multiplexers as well as the circuit that detects the conflict.

A procedure employed in some computers is to give the responsibility for solving data conflicts problems to the complier that translates the high-level programming language into a machine language program. The complier for such computers is designed to detect a data conflict and reorder the instructions as necessary to delay the loading of the conflicting data by inserting no-operation instructions. This method is referred to as delayed load. An example of delayed load is presented in the next section.

10.3 SUPERSCALAR PROCESSORS

The term *superscalar,* first coined in 1987, refers to a machine that is designedto improve the performance of the execution of scalar instructions. In mostapplications, the bulk of the operations are on scalar quantities. Accordingly, the superscalar approach represents the next step in the evolution of high-performance general-purpose processors. The essence of the superscalar approach is the ability to execute instructions independently and concurrently in different pipelines. The concept can be further exploited by allowing instructions to be executed in an order different from the program order. Figure 10.14 shows, in general terms, the superscalar approach. There are multiple functional units, each of which is implemented as a pipeline, which support parallel execution of several instructions.

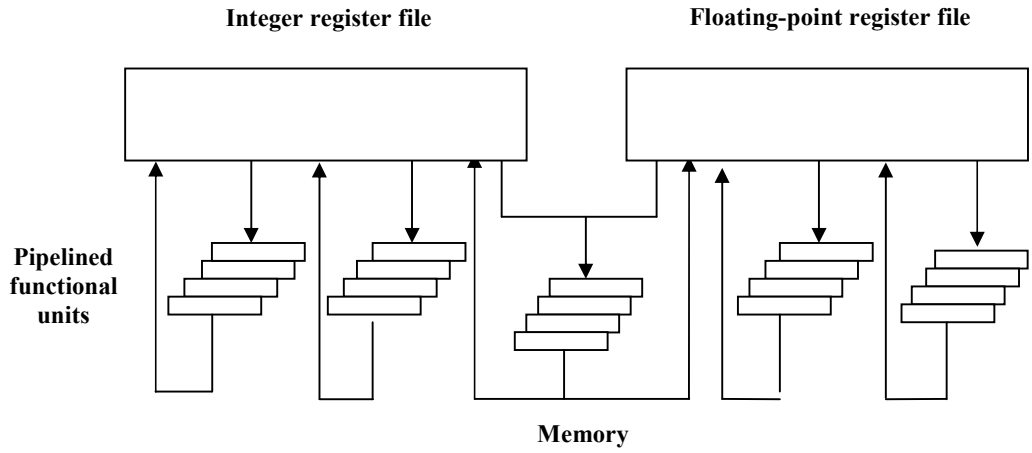


Figure 10.14 General Superscalar Organization

In this example, two integer, two floating-point, and one memory (either load or store) operations can be executing at the same time. Many researchers have investigated superscalar-like processors, and their research indicates that some degree of performance improvement is possible. The differences in the results arise from differences both in the hardware of the simulated machine and in the applications being simulated.

A superscalar processor typically fetches multiple instructions at a time and then attempts to find nearby instructions that are independent of one another and can therefore be executed in parallel. If the input to one instruction depends on the output of a preceding instruction, then the latter instruction cannot complete execution at the same time or before the former instruction. Once such dependencies have been identified, the processor may issue and complete instructions in an order that differs

from that of the original machine code. The processor may eliminate some unnecessary dependencies by the use of additional registers and the renaming of register references in the original code. Whereas pure RISC processors often employ delayed branches to maximize the utilization of the instruction pipeline, this method is less appropriate to a superscalar machine. Instead, most superscalar machines use traditional branch prediction methods to improve efficiency.

A superscalar implementation of a processor architecture is one in which common instructions—integer and floating-point arithmetic, loads, stores, and conditional branches—can be initiated simultaneously and executed independently. Such implementations raise a number of complex design issues related to the instruction pipeline. Superscalar design arrived on the scene hard on the heels of RISC architecture. Although the simplified instruction set architecture of a RISC machine lends itself readily to superscalar techniques, the superscalar approach can be used on either a RISC or CISC architecture. Whereas the gestation period for the arrival of commercial RISC machines from the beginning of true RISC research with the IBM 801 and the Berkeley RISC I was seven or eight years, the first superscalar machines became commercially available within just a year or two of the coining of the term *superscalar*. The superscalar approach has now become the standard method for implementing high-performance microprocessors.

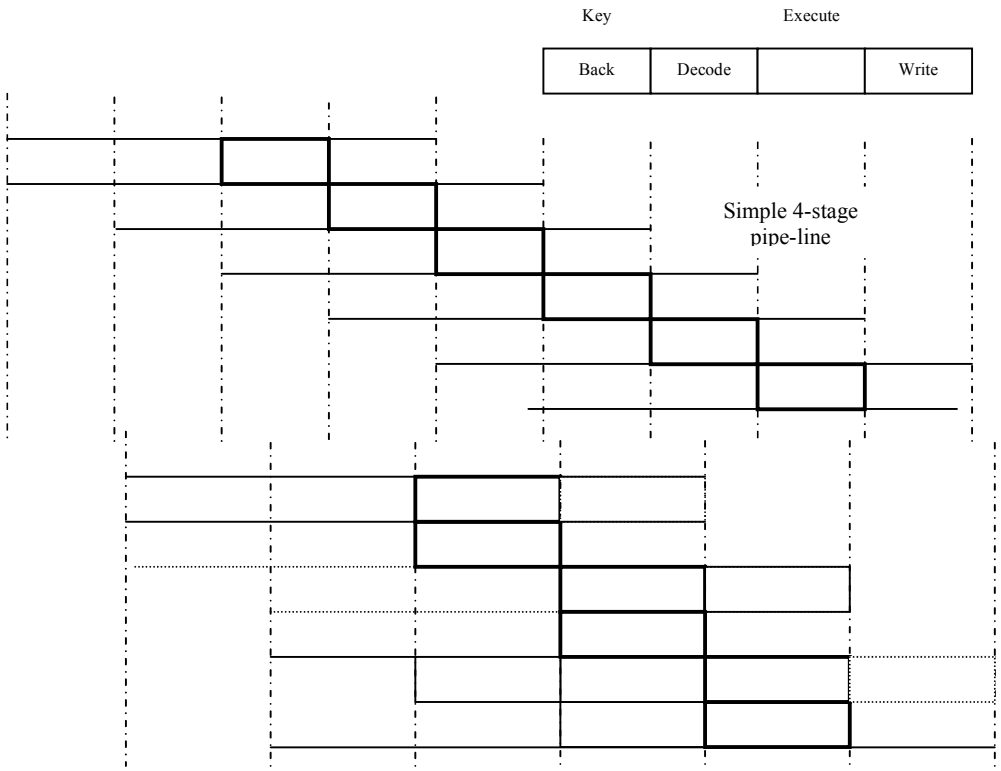


Figure 10.15 (a) Simple 4 stage pipeline (b) Superscalar

Figure 10.15 shows comparison between simple 4 stage pipeline and superscalar. Figure 10.15 (a). The base pipeline issues one instruction per clock cycle and can perform one pipeline stage per clock cycle. Note that although several instructions are executing concurrently, only one instruction is in its execution stage at any one time. Figure 10.15 (b) shows a superscalar implementation capable of executing two instances of each stage in parallel. Higher-degree superscalar implementations are of course possible.

Superscalar Execution

We are now in a position to provide an overview of superscalar execution of programs; this is illustrated in Figure 10.16. The program to be executed consists of a linear sequence of instructions. This is the static program as written by the programmer or generated by the compiler. The instruction fetch process, which includes branch prediction, is used to form a dynamic stream of instructions.

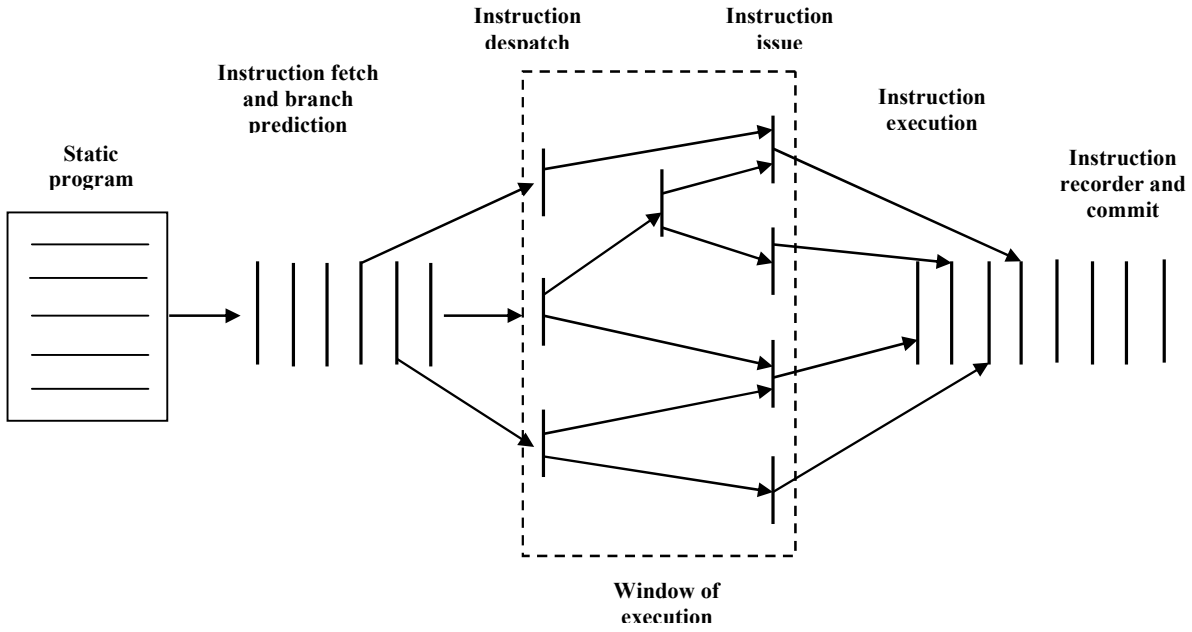


Figure 10.16 Conceptual description of superscalar processing

This stream is examined for dependencies, and the processor may remove artificial dependencies. The processor then dispatches the instructions into a window of execution. In this window, instructions no longer form a sequential stream but are structured according to their true data dependencies. The processor performs the execution stage of each instruction in an order determined by the true data dependencies and hardware resource availability. Finally, instructions are conceptually put back into sequential order and their results are recorded. The final step mentioned in the preceding paragraph is referred to as committing, or retiring, the instruction. This step is needed for the following reason. Because of the use of parallel, multiple pipelines, instructions may complete in an order different from that shown in the static program. Further, the use of branch prediction and speculative execution means that some instructions may complete execution and then must be abandoned because the branch they represent is not taken. Therefore, permanent storage and program-visible registers cannot be updated immediately when instructions complete execution. Results must be held in some sort of temporary storage that is usable by dependent instructions and then made permanent when it is determined that the sequential model would have executed the instruction.

Superscalar Implementation

Based on our discussion so far, we can make some general comments about the processor hardware required for the superscalar approach. Lists of the following key elements:

- Instruction fetch strategies that simultaneously fetch multiple instructions, often by predicting the outcomes of, and fetching beyond, conditional branch instructions. These functions require the use of multiple pipeline fetch and decode stages, and branch prediction logic.
- Logic for determining true dependencies involving register values, and mechanisms for communicating these values to where they are needed during execution.
- Mechanisms for initiating, or issuing, multiple instructions in parallel.
- Resources for parallel execution of multiple instructions, including multiple pipelined functional units and memory hierarchies capable of simultaneously servicing multiple memory references.
- Mechanisms for committing the process state in correct order.

10.4 MULTIPROCESSOR SYSTEMS

A multiprocessor system is an interconnection of two or more CPUs with memory and input—output equipment. The term "processor" in multiprocessor can mean either a central processing unit (CPU) or an input-output processor (IOP). However, a system with a single CPU and one or more IOPs is usually not included in the definition of a multiprocessor system unless *the* IOP has computational facilities comparable to a CPU. As it is most commonly defined, a multiprocessor system implies the existence of multiple CPUs, although usually there will be one or more IOPs as well.

There are some similarities between multiprocessor and multicomputer systems since both support concurrent operations. However, there exists an important distinction between a system with multiple computers and a system with multiple processors. Computers are interconnected with each other by means of

communication lines to form a *computer network. The* network consists of several autonomous computers that may or may not communicate with *each* other. A multiprocessor system is controlled by one operating system that provides interaction between processors and all the components of the system cooperate in the solution of a problem.

Although some large-scale computers include two or more CPUs in their overall system, it is the emergence of the microprocessor that has been the major motivation for multiprocessor systems. The fact that microprocessors take very little physical space and are very inexpensive brings about the feasibility of interconnecting a large number of microprocessors into one composite system. Very-large-scale integrated circuit technology has reduced the cost of computer components to such a low level that the concept of applying multiple processors to meet system performance requirements has become an attractive design possibility.

Multiprocessing improves the reliability of the system so that a failure or error in one part has a limited effect on the rest of the system. If a fault causes one processor to fail, a second processor can be assigned to perform the functions of the disabled processor. The system as a whole can continue to function correctly with perhaps some loss in efficiency.

The benefit derived from a multiprocessor organization is an improved system performance. The system derives its high performance from the fact that computations can proceed in parallel in one of two ways.

1. Multiple independent jobs can be made to operate in parallel.
2. A single job can be partitioned into multiple parallel tasks.

An overall function can be partitioned into a number of tasks that each processor can handle individually. System tasks may be allocated to special purpose processors whose design is optimized to perform certain types of processing efficiently. An example is a computer system where one processor performs the computations for an industrial process control while others monitor and control the various parameters, such as temperature and flow rate. Another

example is a computer where one processor performs high speed floating-point mathematical computations and another takes care of routine data-processing tasks.

Multiprocessing can improve performance by decomposing a program into parallel executable tasks. This can be achieved in one of two ways. The user can explicitly declare that certain tasks of the program be executed in parallel. This must be done prior to loading the program by specifying the parallel executable segments. Most multiprocessor manufacturers provide an operating system with programming language constructs suitable for specifying parallel processing. The other, more efficient way is to provide a compiler with multiprocessor software that can automatically detect parallelism in a user's program. The compiler checks for *data dependency* in the program. If a program depends on data generated in another part, the part yielding the needed data must be executed first. However, two parts of a program that do not use data generated by each can run concurrently. The parallelizing compiler checks the entire program to detect any possible data dependencies. These that have no data dependency are then considered for concurrent scheduling on different processors.

Multiprocessors are classified by the way their memory is organized. A multiprocessor system with common shared memory is classified as a *shared- memory* or *tightly coupled multiprocessor*. This does not preclude each processor from having its own local memory. In fact, most commercial tightly coupled multiprocessors provide a cache memory with each CPU. In addition, there is a global common memory that all CPUs can access. Information can therefore be shared among the CPUs by placing it in the common global memory.

An alternative model of microprocessor is the *distributed-memory* or *loosely coupled* system. Each processor element in a loosely coupled system has its own private local memory. The processors are tied together by a switching scheme designed to route information from one processor to another through a message-passing scheme. The processors relay program and data to other processors in packets. A packet consists of an address, the data content, and some error

detection code. The packets are addressed to a specific processor or taken by the first available processor, depending on the communication system used. Loosely coupled systems are most efficient when the interaction between tasks is minimal, whereas tightly coupled systems can tolerate a higher degree of interaction between tasks.

10.4.1 Interconnection Structures

The components that form a multiprocessor system are CPUs, 10Ps connected to input—output devices, and a memory unit that may be partitioned into a number of separate modules. The interconnection between the components can have different physical configurations, depending on the number of transfer paths that are available between the processors and memory in a shared memory system or among the processing elements in a loosely coupled system. There are several physical forms available for establishing an interconnection network. Some of these schemes are presented in this section:

1. Time-shared common bus
2. Multiport memory
3. Crossbar switch
4. Multistage switching network
5. Hypercube system

Time-Shared Common Bus

A common-bus multiprocessor system consists of a number of processors connected through a common path to a memory unit. A time-shared common bus for five processors is shown in Fig. 10.17.

Only one processor can communicate with the memory or another processor at any given time. Transfer operations are conducted by the processor that is in control of the bus at the time. Any other processor wishing to initiate a transfer must first determine the availability status of the bus, and only after the bus becomes available can the processor address the destination unit to initiate the transfer. A command is issued to inform the destination unit what operation is to be performed. The receiving unit recognizes its address in the bus

and responds to the control signals from the sender, after which the transfer is initiated. The system may exhibit transfer conflicts since one common bus is shared by all processors. These conflicts must be resolved by incorporating a bus controller that establishes priorities among the requesting units.

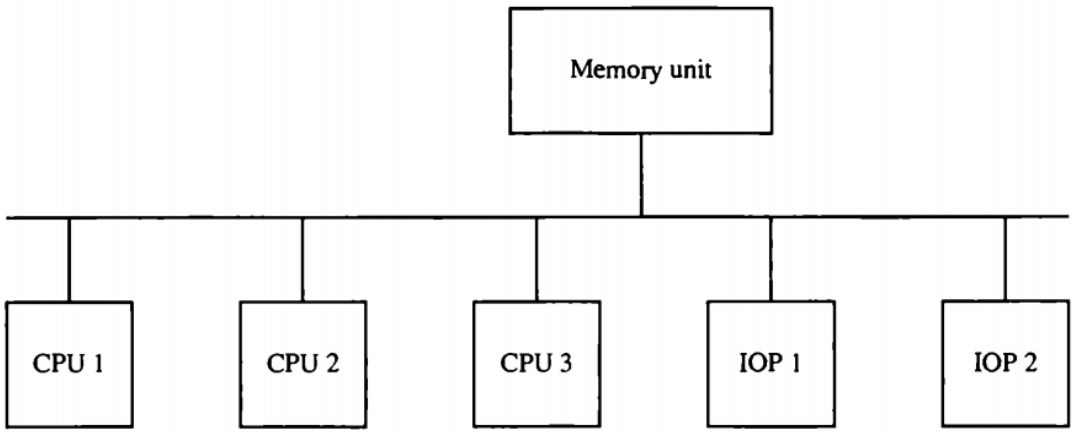


Figure 10.17 Time-shared common bus organization.

A single common-bus system is restricted to one transfer at a time. This means that when one processor is communicating with the memory, all other processors are either busy with internal operations or must be idle waiting for the bus. As a consequence, the total overall transfer rate within the system is limited by the speed of the single path. The processors in the system can be kept busy more often through the implementation of two or more independent buses to permit multiple simultaneous bus transfers. However, this increases the system cost and complexity.

A more economical implementation of a dual bus structure is depicted in Fig. 10.18. Here we have a number of local buses each connected to its own local memory and to one or more processors. Each local bus may be connected to a CPU, an IOP, or any combination of processors. A system bus controller links each local bus to a common system bus. The I/O devices connected to the local IOP, as well as the local memory, are available to the local processor. The memory connected to the common system bus is shared by all

processors. If an IOP is connected directly to the system bus, the I/0 devices attached to it may be made available to all processors. Only one processor can communicate with the shared memory and other common resources through the system bus at any given time.

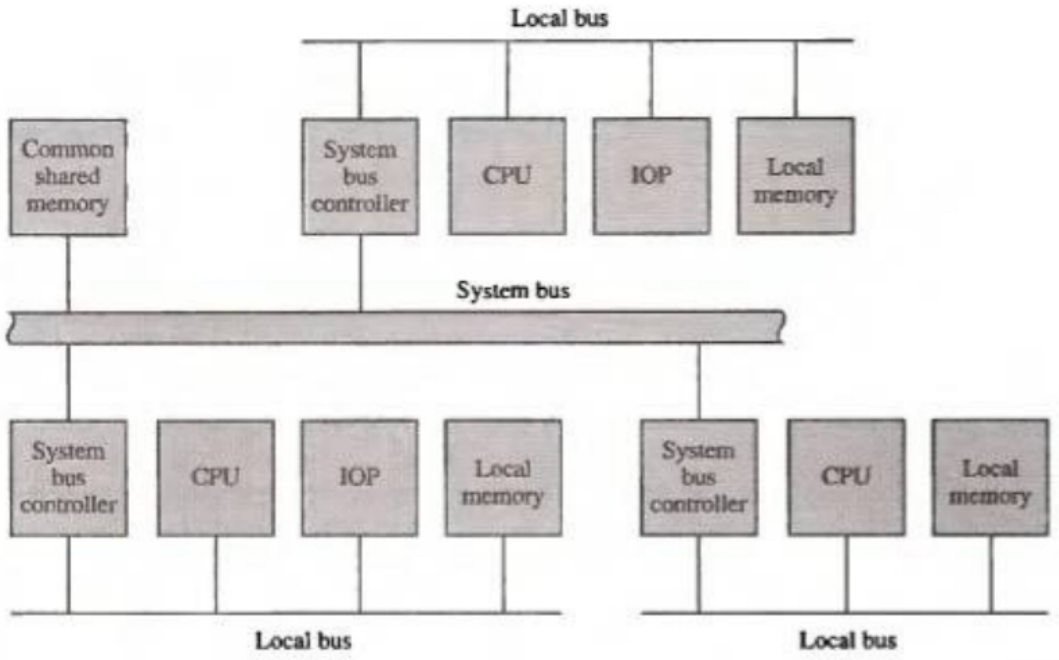


Figure 10.18 System bus structure for multiprocessors.

The other processors are kept busy communicating with their local memory and I/O devices. Part of the local memory may be designed as a cache memory attached to the CPU (see Sec. 12-4 In this way, the average access time of the local memory can be made to approach the cycle time of the CPU to which it is attached.

Multipart Memory

A multiport memory system employs separate buses between each memory module and each CPU. This is shown in Fig. 10.19 for four CPUs and four memory modules (Mb). Each processor bus is connected to each memory module. A processor bus consists of the address, data, and control lines required to communicating with

memory. The memory module is said to have four ports and *each* port accommodates one of the buses. The module must have internal control logic to determine which port will have access to memory at any given time. Memory access conflicts are resolved by assigning fixed priorities to each memory port. The priority for memory access associated with each processor may be established by the physical port position that its bus occupies in each module. Thus CPU 1 will have priority over CPU 2, CPU 2 will have priority over CPU 3, and CPU 4 will have the lowest priority.

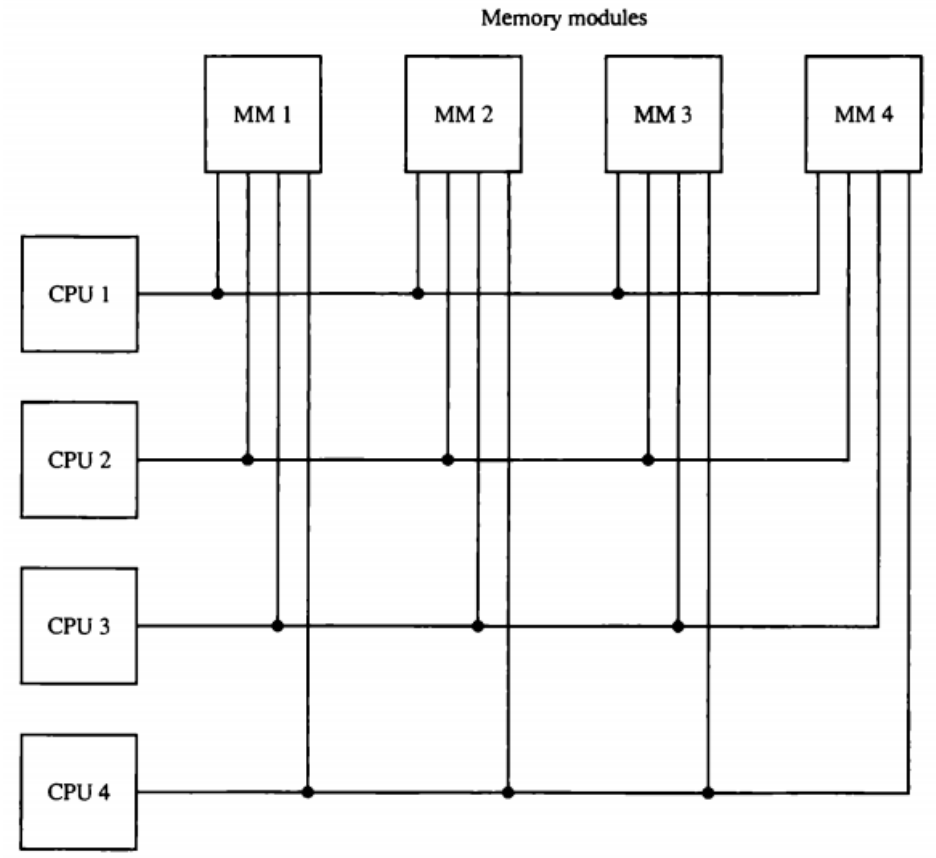


Figure 10.19 Multiport memory organization.

The advantage of the multiport memory organization is the high transfer rate that can be achieved because of the multiple paths between processors and memory. The disadvantage is that it requires expensive memory control logic and a *large* number of cables and connectors. As a consequence, *this* interconnection structure is usually

appropriate for systems with a small number of processors.

Crossbar Switch

The crossbar switch organization consists of a number of crosspoints that are placed at intersections between processor buses and memory module paths. Figure 10.20 shows a crossbar switch interconnection between four CPUs and four memory modules.

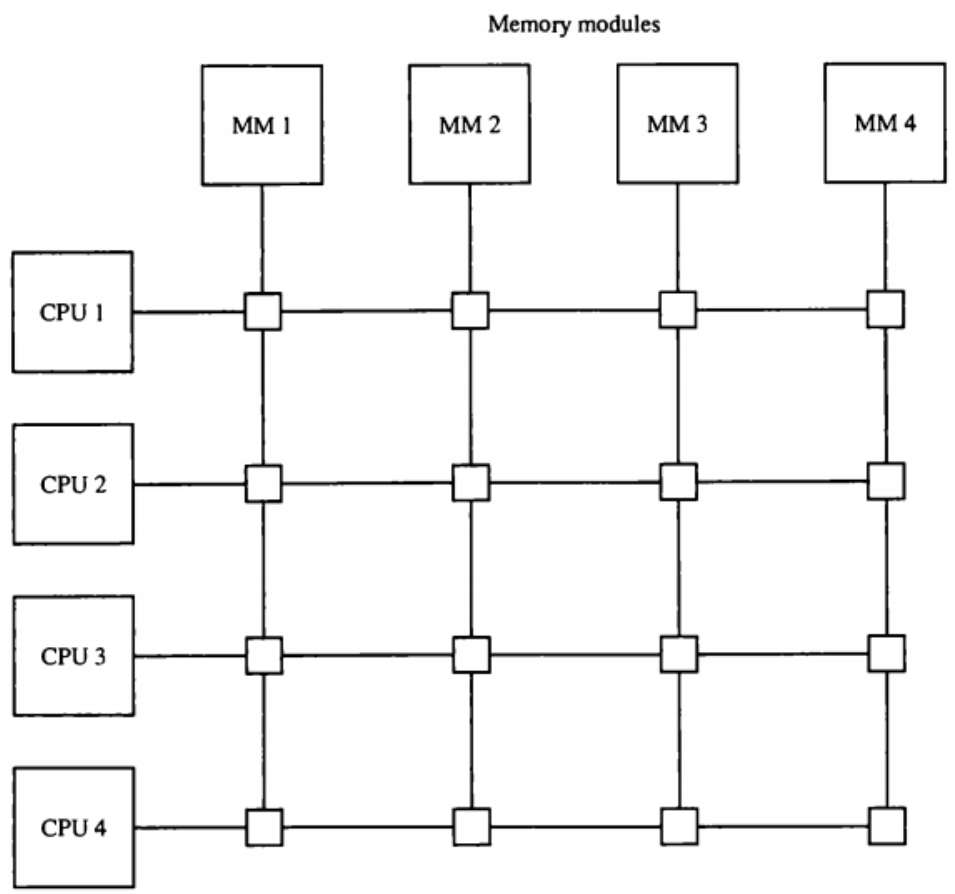


Figure 10.20 Crossbar switch.

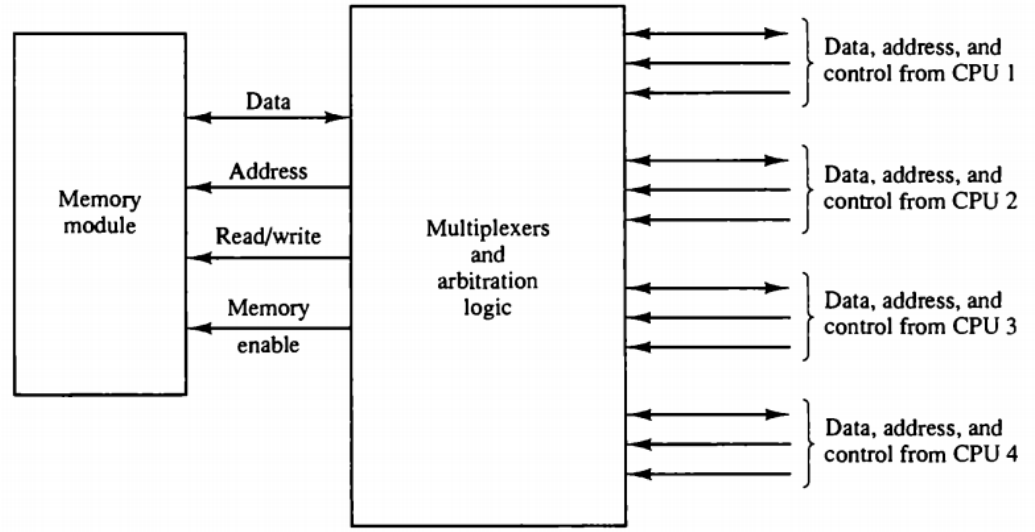
The small square in each crosspoint is a switch that determines the path from a processor to a memory module. Each switch point has control logic to set up the transfer path between a processor and memory. It examines the address that is placed in the bus to determine whether its particular module is being addressed. It also resolves multiple requests for access to the same memory module on a

predetermined priority basis.

Figure 10.21 shows the functional design of a crossbar switch connected to one memory module. The circuit consists of multiplexers that select the data, address, and control from one CPU for communication with the memory module. Priority levels are established by the arbitration logic to select one CPU when two or more CPUs attempt to access the same memory. The multiplexers are controlled with the binary code that is generated by a priority encode within the arbitration logic.

A crossbar switch organization supports simultaneous transfers from memory modules because there is a separate path associated with each module. However, the hardware required to implement the switch can become quite large and complex.

Figure 10.21 Block diagram of crossbar switch.



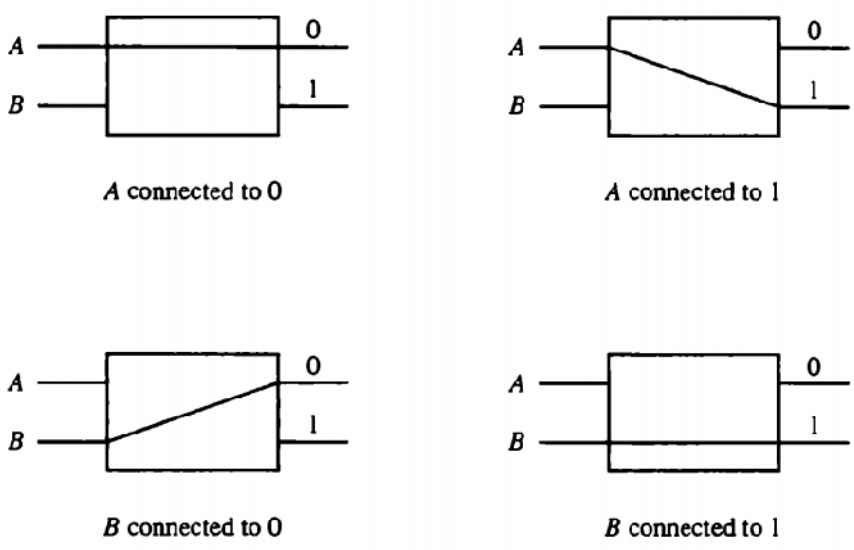
Multistage Switching Network

The basic component of a multistage network is a two-input, two-output interchange switch. As shown in Fig. 10.22, the 2 x 2 switch has two input labeled *A* and *B,* and two outputs, labeled 0 and 1. There are control sign (not shown) associated with the switch that establish the interconnection between the input and output terminals. The switch has the capability connecting input A to either

of the outputs. Terminal *B* of the switch behave in a similar fashion. The switch also has the capability to arbitrate between conflicting requests. If inputs A and *B* both request the same output terminate only one of them will be connected; the other will be blocked.

Using the 2 x 2 switch as a building block, it is possible to build multistage network to control the communication between a number of sours and destinations. To see how this is done, consider the binary tree shown Fig. 10.23. The two processors *Pi* and *P2* are connected through switches to eight memory modules marked in binary from 000 through 111. The path from source to a destination is determined from the binary bits of the destination number.

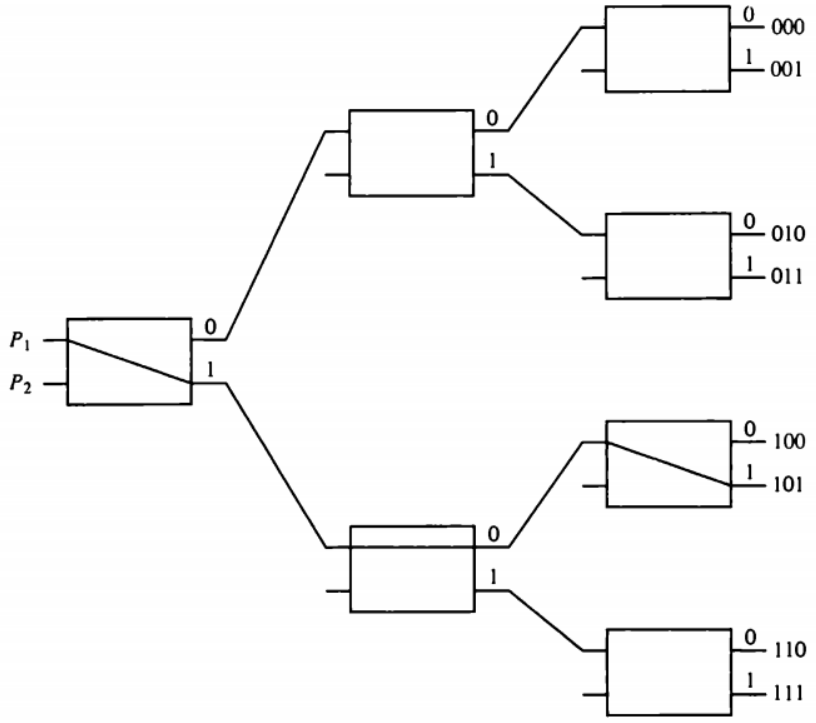
Figure 10.22 Operation of a 2 x 2 interchange switch.



The first bit of the destination number determines the switch output in the first level. The second bit specifies the output of the switch in the second level, and the third bit specifies the output of the switch in the third level. For example, to connect *P,* to memory 101, it is necessary to form a path from 132 to output 1 in the first-level switch, output 0 in the second-level switch, and output 1 in the third-level switch. It is clear that either P, or *P2* can be connected to any one of the eight memories. Certain request patterns, however, cannot be satisfied simultaneously. For example, if PI is connected to one of the

destinations 000 through 011, *P2* can be connected to only one of the destinations 100 through 111.

Figure 10.23 Binary tree with 2 x 2 switches.



Many different topologies have been proposed for multistage switching networks to control processor-memory communication in a tightly coupled multiprocessor system or to control the communication between the processing elements in a loosely coupled system. One such topology is the omega switching network shown in Fig. 10.24. In this configuration, there is exactly one path from each source to any particular destination. Some request patterns, however, cannot be connected simultaneously. For example, any two sources cannot be connected simultaneously to destinations 000 and 001.

A particular request is initiated in the switching network by the source, which sends a 3-bit pattern representing the destination number. As the binary pattern moves through the network, each level examines a different bit to determine the 2 *x* 2 switch setting. Level 1 inspects the most significant bit, level 2 inspects the middle bit, and level 3 inspects the least significant bit. When the request arrives on either input of the 2 x

2 switch, it is routed to the upper output if the specified bit is 0 or to the lower output if *the* bit is 1.

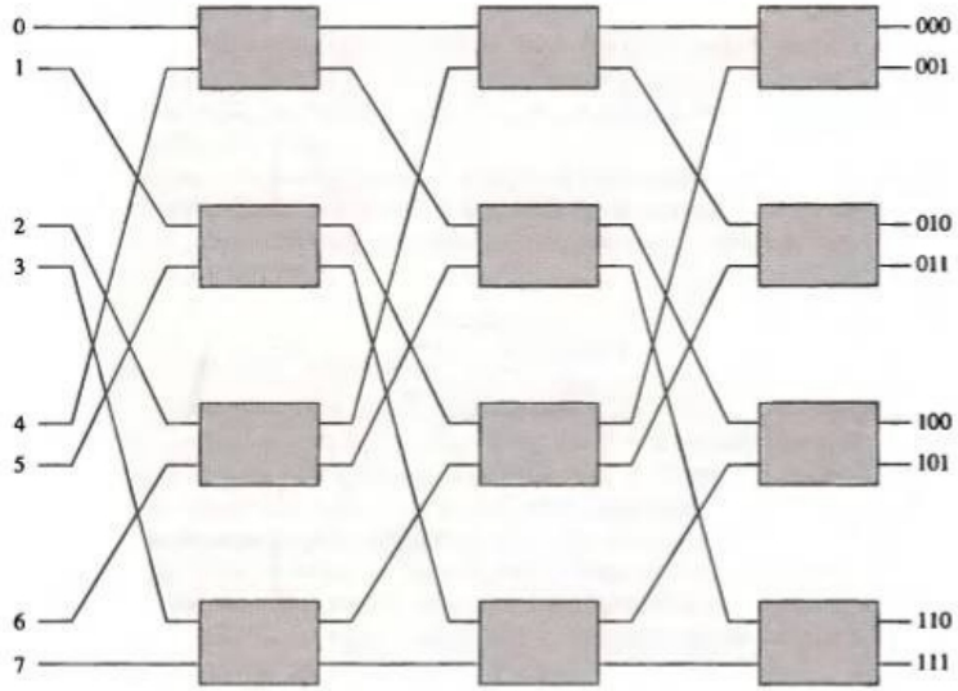


Figure 10.24 8 x 8 omega switching network.

In a tightly coupled multiprocessor system, the source is a processor and the destination is a memory module. The first pass through the network sets up the path. Succeeding passes are used to transfer the address into memory and then transfer the data in either direction, depending on whether the request is a read or a write. In a loosely coupled multiprocessor system, both *the* source and destination are processing elements. After the path is established, the source processor transfers a message to the destination processor.

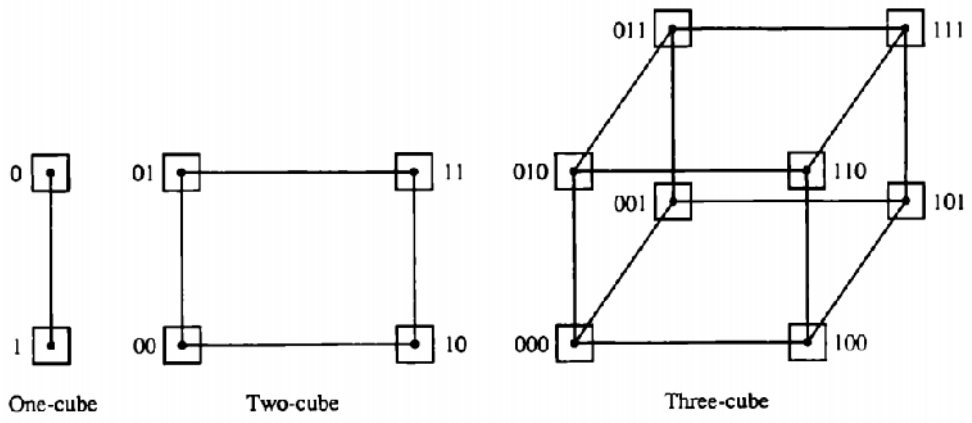
Hypercube interconnection

The hypercube or binary n-cube multiprocessor structure is a loosely coupled system composed of $N = 2$ processors interconnected in an n-dimensional binary cube. Each processor forms a *node* of the cube. Although it is customary to refer to each node as having a processor, in effect it contains not only a CPU but also local memory and I/O interface. Each processor has direct communication paths to n other neighbor

processors. These paths correspond to the *edges* of the cube. There are 2" distinct n-bit binary addresses that can be assigned to the processors. Each processor address differs from that of each of its *n* neighbors by exactly one bit position.

Figure 10.25 shows the hypercube structure for *n = 1,* 2, and 3. A one-cube structure has *n* = 1 and 2" = 2. It contains two processors interconnected by a single path. A two-cube structure has n = 2 and 2" = 4. It contains four nodes interconnected as a square. A three-cube structure has eight nodes interconnected as a cube. An *n* -cube structure has 2" nodes with a processor residing in each node. Each node is assigned a binary address in such a way that the addresses of two neighbors differ in exactly one bit position. For example, the three neighbors of the node with address 100 in a three-cube structure are 000, 110, and 101. Each of these binary numbers differs from address 100 by one bit value.

Figure 10.25 Hypercube structures for n = 1,2,3.



Routing messages through an n-cube structure may take from one to n links from a source node to a destination node. For example, in a three-cube structure, node 000 can communicate directly with node 001. It must cross at least two links to communicate with 011 (from 000 to 001 to 011 or from 000 to 010 to 011). It is necessary to go through at least three links to communicate from node 000 to node 111. A routing procedure can be developed by computing the exclusive-OR

of the source node address with the destination node address. The resulting binary value will have 1 bits corresponding to the axes on which the two nodes differ. The message is then sent along any one of the axes. For example, in a three-cube structure, a message at 010 going to 001 produces an exclusive-OR of the two addresses equal to 011. The message can be sent along the second axis to 000 and then through the third axis to 001.

A representative of the hypercube architecture is the Intel iPSC computer complex. It consists of 128 (n = 7) microcomputers connected through communication channels. Each node consists of a CPU, a floating-point processor, local memory, and serial communication interface units. The individual nodes operate independently on data stored in local memory according to resident programs. The data and programs to each node come through a message-passing system from other nodes or from a cube manager. Application programs are developed and compiled on the cube manager and then downloaded to the individual nodes. Computations are distributed through the system and executed concurrently.

10.5 CHECK YOUR PROGRESS

1. How many types of Pipelining exist?
   a) 2
   b) 3
   c) 4
   d) 5
2. Arithmetic Pipeline is used for?
   a) floating point operations
   b) interger operations
   c) character operations
   d) None of the above
3. Which of the following is not a Pipeline Conflicts?
   a) Timing Variations

b) Branching

c) Load Balancing

d) Data Dependency

4. Which of the following is disadvantage of Pipelining?

a) cycle time of the processor is reduced.

b) The design of pipelined processor is complex and costly to manufacture.

c) The instruction latency is more.

d) Both B and C

5. Which of the following is an advantage of pipelining?

a) Instruction throughput increases.

b) Faster ALU can be designed when pipelining is used.

c) Pipelining increases the overall performance of the CPU.

d) All of the above

10.6 SUMMARY

Pipelining is a technique of decomposing a sequential process into sub-operations, with each sub-process being executed in a special dedicated segment that operates concurrently with all other segments. A pipeline can be visualized as a collection of processing segments through which binary information flows. Pipelining is an implementation technique whereby multiple instructions are overlapped in execution. This is solved without additional hardware but only by letting different parts of the hardware work for different instructions at the same time. To duplicate the theoretical speed advantage of a pipeline process by means of multiple functional units, it is necessary to construct k identical units that will be operating in parallel. The implication is that a k-segment pipeline processor can be expected to equal the performance of k copies of an equivalent non-pipeline circuit under equal operating conditions. There are various reasons why the pipeline cannot operate at its maximum theoretical rate. Different segments may take different times to complete their sub-operation. The clock cycle must be chosen to equal the time delay of the segment with the maximum propagation time. This causes all other segments to waste time while waiting for the next clock. There are two areas of computer design where the pipeline organization is applicable. An arithmetic pipeline divides an arithmetic operation into

sub-operations for execution in the pipeline segments. An instructions pipeline operates on a stream of instructions by overlapping the fetch, decode, and execute phases of the instruction cycle.

A superscalar processor is one in which multiple independent instruction pipelines are used. Each pipeline consists of multiple stages, so that each pipeline can handle multiple instructions at a time. Multiple pipelines introduce a new level of parallelism, enabling multiple streams of instructions to be processed at a time. A superscalar processor exploits what is known as instruction-level parallelism, which refers to the degree to which the instructions of a program can be executed in parallel. A superscalar processor typically fetches multiple instructions at a time and then attempts to find nearby instructions that are independent of one another and can therefore be executed in parallel.

10.7 KEYWORDS

1. **Pipelining:** Pipelining is most suited for tasks in which essentially the same sequence of steps must be repeated many times for different data. a pipeline which breaks a process into N steps could achieve an N-fold increase in processing speed.
2. **Speedup:** speedup ratio is the ratio of time taken to do task by pipeline processor to non pipeline processor.
3. **Arithmetic pipeline:** It divides the arithmetic operation into sub-operations for execution in pipeline segment.
4. **Instruction pipeline:** It operates on stream of instructions by overlapping the fetch, decode and execute phases of instruction cycle.
5. **FIFO Buffer:** Instructions are placed in FIFO buffer, waiting for decoding and processing by execution segment.
6. **Hardware Interlocks:** An interlock is a circuit that detects instructions whose source operands are destinations of instructions farther up in the pipeline.
7. **Operand Forwarding:** It uses special hardware to detect a conflict and then avoid it by routing the data through special paths into a destination register. The hardware checks the destination operand, and if it is needed as a source in the next instruction, it processes the result directly into the ALU input.

8. **Delayed Load:** The complier for some computers is designed to detect a data conflict and reorder the instructions as necessary to delay the loading of the conflicting data by inserting no operation instructions. This technique is applied in most RISC processors.
9. **Branch Target Buffer (BTB):** The BTB is associative memory included in the fetch segment of the pipeline. Each entry in BTB consists of address of previously executed branch instruction and the target instruction for that branch.
10. **Loop Buffer:** It is variation of BTB. This is small very high speed register file maintained by instruction fetch segment of pipeline.
11. **Superscalar Processor:** A superscalar processor is one in which multiple independent instruction pipelines are used. Each pipeline consists of multiple stages, so that each pipeline can handle multiple instructions at a time.

10.8 SELF-ASSESSMENT TEST

1. Define pipelining. What is the benefit of having many pipeline stages in processor?
2. What is difference between superscalar processing and simple pipelining processing?
3. What are the key elements of a superscalar processor organization?
4. In a certain scientific computations it is necessary to perform the arithmetic operation $(A_i+B_i)(C_i+D_i)$ with a stream of numbers. Specify a pipeline configuration to carry out this task. List the contents of all registers in the pipeline for i=1 through
5. Draw a space-timing diagram for a six-segment pipeline showing the time it takes to process eight tasks.
6. Determine the number of clock cycles that it takes to process 200 tasks in a six-segment pipeline.
7. A non-pipeline system takes 50ns to process a task. The same task can be processed in a six-segment pipeline with a clock cycle of 10ns. Determine the speedup ratio of the pipeline for 100 tasks. What is the maximum speed up that can be achieved?
8. Formulate a six-segment instruction pipeline for a computer. Specify the operations to be performed in each segment.

10.9 ANSWERS TO CHECK YOUR PROGRESS

1. A
2. A
3. C
4. D
5. D

10.10 REFERENCES / SUGGESTED READINGS

1. Computer Organization and Architecture, Rajaram & Radhakrishan, PHI.
2. Computer Organization & Architecture: Designing for Performance, Stalling, PHI.
3. Computer Organization and Design, Pal Choudhary, PHI.
4. Computer Systems Organization & Architecture, Carpenelli, Pearson Education.
5. Computer Organization and Architecture, Stalling, Pearson Education.
6. Computer System Architecture, Morris Mano, PHI.
7. Computer Architecture and Organization, McGraw Hill Company, New Delhi. J.P. Hayes.