

Chapter One: Introduction

Objectives

After the completion of this chapter students will be able to:

- ⇒ Understand basic terminology, types of logic gates (**AND**, **OR**, **NOT**, **NAND**, **NOR**, **XOR**)
- ⇒ Understand the basic operations used in computers and other digital systems
- ⇒ Identify basic rules of Boolean algebra, De Morgan's laws

Key concepts

- **Boolean algebra** is digital circuitry in digital computers and other digital system is designed, and its behavior is analyzed, with the use of a mathematical discipline.
- **Logic gate** is A large number of electronic circuits (in computers, control units, and so on) are made up of logic gates.
- When constructing a truth table, the binary values 1 and 0 are used
- A logic circuit whose output depends only on its current inputs is called a **Sequential circuit**. Its operation is fully described by a truth table that lists all possible combinations of inputs and the output values produced by each

Overview

The operation of the digital computer is based on the storage and processing of binary data. Throughout this book, we have assumed the existence of storage elements that can exist in one of two stable states and of circuits that can operate on binary data under the control of control signals to implement the various computer functions. In this appendix, we suggest how these storage elements and circuits can be implemented in digital logic, specifically with combinational and sequential circuits. The appendix begins with a brief review of Boolean algebra, which is the mathematical foundation of digital logic. Next, the concept of a gate is introduced. Finally, combinational and sequential circuits, which are constructed from gates, are described.

1.1 Boolean Algebra

The digital circuitry in digital computers and other digital systems is designed, and its behavior is analyzed, with the use of a mathematical discipline known as Boolean algebra. The name is in honor of an English mathematician George Boole, who proposed the basic principles of this algebra in 1854 in his treatise, *An Investigation of the Laws of Thought on Which to Found the Mathematical Theories of Logic and Probabilities*. In 1938, Claude Shannon, a research assistant in the Electrical Engineering Department at M.I.T., suggested that Boolean algebra could be used to solve problems in relay-switching circuit design.

Boolean algebra turns out to be a convenient tool in two areas:

1. Analysis: It is an economical way of describing the function of digital circuitry.
2. Design: Given a desired function, Boolean algebra can be applied to develop a simplified implementation of that function.

As with any algebra, Boolean algebra makes use of variables and operations. In this case, the variables and operations are logical variables and operations. Thus, a variable may take on the value 1 (TRUE) or 0 (FALSE). The basic logical operations are AND, OR, and NOT, which are symbolically represented by dot, plus sign, and overbar:

$$A \text{ AND } B = A.B$$

$$A \text{ or } B = A+B$$

$$\text{NOT } A = \overline{A}$$

The operation AND produce true (binary value 1) if and only if both of its operands are true. The operation OR yields true if either or both of its operands are true. The unary operation NOT inverts the value of its operand. For example, consider the equation

$$D = A + (B . C)$$

D is equal to 1 if A is 1 or if both B = 0 and C = 1. Otherwise D is equal to 0

Table 1.1 Boolean Operators

P	Q	NOT P(\overline{P})	P AND Q(P.Q)	P OR Q(P+Q)	P NAND Q($\overline{P.Q}$)	P NOR Q($\overline{P+Q}$)	P XOR Q($P \oplus Q$)
0	0	1	0	0	1	1	0
0	1	1	0	1	1	0	1
1	0	0	0	1	1	0	1
1	1	0	1	1	0	0	0

a) Two Input Variables

operation	Expression	Output=1 if
AND	$A.B...$	All of the set (A, B,...) are 1.
OR	$A+B+....$	Any of the set (A, B,...) are 1.
NAND	$\overline{A.B...}$	Any of the set (A, B,...)are 0.
NOR	$\overline{A+B+\dots}$	All of the set (A, B,...) are 0.
XOR	$A\oplus B\oplus....$	The set (A,B,...)contains an odd number of ones

b) Extended to More than Two Inputs (A, B . . .)

Several points concerning the notation are needed. In the absence of parentheses, the AND operation takes precedence over the OR operation. Also, when no ambiguity will occur, the AND operation is represented by simple concatenation instead of the dot operator.

Thus,

$$A + B.C = A + (B.C) = A + BC$$

all mean: Take the AND of B and C; then take the OR of the result and A. Table 1.1a defines the basic logical operations in a form known as a truth table, which lists the value of an operation for every possible combination of values of operands. The table also lists three other useful operators: XOR, NAND, and NOR. The exclusive-or (XOR) of two logical operands is 1 if and only if exactly one of the operands has the value 1. The NAND function is the complement (NOT) of the AND function, and the NOR is the complement of OR:

$$A \text{ NAND } B = \text{NOT } (A \text{ AND } B) = \overline{AB}$$

$$A \text{ NOR } B = \text{NOT } (A \text{ OR } B) = \overline{A+B}$$

As we shall see, these three new operations can be useful in implementing certain digital circuits.

The logical operations, with the exception of NOT, can be generalized to more than two variables, as shown in Table 1.1b.

Table 1.2 Basic Identities of Boolean algebra

Basic postulates		
$A.B = B.A$	$A+B = B+A$	commutative laws
$A.(B+C) = (A.B) + (A.C)$	$A+(B.C) = (A+B).(A+C)$	distributive laws
$1.A = A$	$0+A = A$	identity laws
$A.\bar{A} = 0$	$A+\bar{A} = 1$	inverse elements
Other Identities		
$0.A = 0$	$1+A = 1$	
$A.A = A$	$A+A = A$	
$A.(B.C) = (A.B).C$	$A+(B+C) = (A+B)+C$	associative laws
$\overline{A.B} = \bar{A} + \bar{B}$	$\overline{A+B} = \bar{A}.\bar{B}$	De Morgan's theorem

Table 1.2 summarizes key identities of Boolean algebra. The equations have been arranged in two columns to show the complementary, or dual, nature of the AND and OR operations. There are two classes of identities: basic rules (or *postulates*), which are stated without proof, and other identities that can be derived from the basic postulates.

The postulates define the way in which Boolean expressions are interpreted. One of the two distributive laws is worth noting because it differs from what we would find in ordinary algebra:

$$A + (B \cdot C) = (A + B) \cdot (A + C)$$

The two bottoms most expressions are referred to as DE Morgan's theorem. We can restate them as follows. $A \text{ NOR } B = \bar{A} \text{ AND } \bar{B}$ and $A \text{ NAND } B = \bar{A} \text{ OR } \bar{B}$

⇒ Students are invited to verify the expressions in Table 1.2 by substituting actual values (1s and 0s) for the variables A, B, and C.

1.2 Gates

The fundamental building block of all digital logic circuits is the gate. Logical functions are implemented by the interconnection of gates. A gate is an electronic circuit that produces an output signal that is a simple Boolean operation on its input signals. The basic gates used in digital logic are AND, OR, NOT, NAND, NOR, and XOR. Figure 1.1 depicts these six gates. Each gate is defined in three ways: graphic symbol, algebraic notation, and truth table.

Note that the inversion (NOT) operation is indicated by a circle. Each gate shown in Figure 1.1 has one or two inputs and one output. However, as indicated in Table 11b, all of the gates except NOT can have more than two inputs. Thus $(X + Y + Z)$, can be implemented with a single OR gate with three inputs.

When one or more of the values at the input are changed, the correct output signal appears almost instantaneously, delayed only by the propagation time of signals through the gate (known as the gate delay). In some cases, a gate is implemented with two outputs, one output being the negation of the other output.

Here we introduce a common term: we say that to assert a signal is to cause signal line to make a transition from its **logically false (0) state to its logically true (1) State**. The true (1) state is

either a high or low voltage state, depending on the type of electronic circuitry. Typically, not all gate types are used in implementation. Design and fabrication are simpler if only one or two types of gates are used. Thus, it is important to identify *functionally complete* sets of gates. This means that any Boolean function can be implemented using only the gates in the set. The following are functionally complete sets:

AND, OR, NOT, AND, NOTOR, NOT NAND, NOR



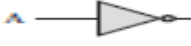


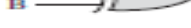
Name	Graphical Symbol	Algebraic Function	Truth Table															
AND		$F = A \cdot B$ or $F = AB$	<table><tr><th>A</th><th>B</th><th>F</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	F	0	0	0	0	1	0	1	0	0	1	1	1
A	B	F																
0	0	0																
0	1	0																
1	0	0																
1	1	1																
OR		$F = A + B$	<table><tr><th>A</th><th>B</th><th>F</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	F	0	0	0	0	1	1	1	0	1	1	1	1
A	B	F																
0	0	0																
0	1	1																
1	0	1																
1	1	1																
NOT		$F = \overline{A}$ or $F = A'$	<table><tr><th>A</th><th>F</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	A	F	0	1	1	0									
A	F																	
0	1																	
1	0																	
NAND		$F = \overline{AB}$	<table><tr><th>A</th><th>B</th><th>F</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	F	0	0	1	0	1	1	1	0	1	1	1	0
A	B	F																
0	0	1																
0	1	1																
1	0	1																
1	1	0																
NOR		$F = \overline{A + B}$	<table><tr><th>A</th><th>B</th><th>F</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	F	0	0	1	0	1	0	1	0	0	1	1	0
A	B	F																
0	0	1																
0	1	0																
1	0	0																
1	1	0																
XOR		$F = A \oplus B$	<table><tr><th>A</th><th>B</th><th>F</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	F	0	0	0	0	1	1	1	0	1	1	1	0
A	B	F																
0	0	0																
0	1	1																
1	0	1																
1	1	0																

Figure 1.1 Basic Logic Gates

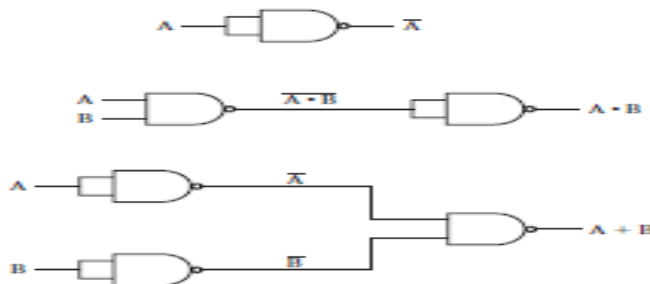


Figure 1.2 The Use of NAND Gates

Complete set, because they represent the three operations of Boolean algebra. For the AND and NOT gates to form a functionally complete set, there must be a way to synthesize the OR operation from the AND and NOT operations. This can be done by applying DE Morgan's theorem: $A+B=\overline{A \cdot B}$ and $A \text{ OR } B = \text{NOT } ((\text{NOT } A) \text{ AND } (\text{NOT } B))$

Similarly, the OR and NOT operations are functionally complete because they can be used to synthesize the AND operation. Figure 1.2 shows how the AND, OR, and NOT functions can be implemented solely with NAND gates, and Figure 1.3 shows the same thing for NOR gates.

For this reason, digital circuits can be, and frequently are, implemented solely with NAND gates or solely with NOR gates.

With gates, we have reached the most primitive circuit level of computer hardware. An examination of the transistor combinations used to construct gates departs from that realm and enters the realm of electrical engineering. For our purposes, however, we are content to describe how gates can be used as building blocks to implement the essential logical circuits of a digital computer.

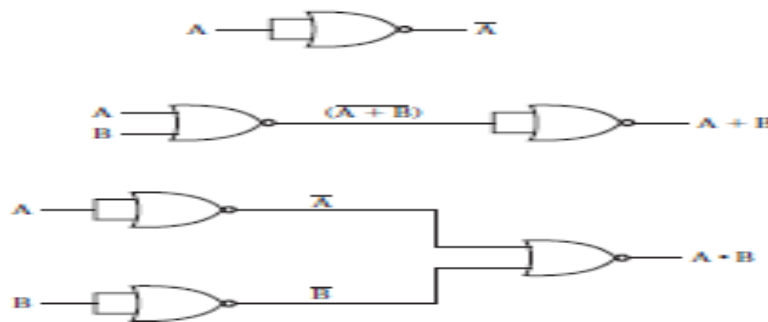


Figure 1.3 NOR Gates

1.3 Combinational Circuits

A combinational circuit is an interconnected set of gates whose output at any time is a function only of the input at that time. As with a single gate, the appearance of the input is followed almost immediately by the appearance of the output, with only gate delays.

In general terms, a combinational circuit consists of n binary inputs and m binary outputs. As with a gate, a combinational circuit can be defined in three ways: Truth table: For each of the 2^n possible combinations of input signals, the binary value of each of the m output signals is listed.

Graphical symbols: The interconnected layout of gates is depicted.

Boolean equations: Each output signal is expressed as a Boolean function of its input signals.

Implementation of Boolean Functions

Any Boolean function can be implemented in electronic form as a network of gates. For any given function, there are a number of alternative realizations. Consider the Boolean function represented by the truth table in Table 1.3. We can express this function by simply itemizing the combinations of values of A , B , and C that cause F to be 1:

$$F = \bar{A}\bar{B}\bar{C} + \bar{A}BC + A\bar{B}\bar{C}$$

There are three combinations of input values that cause F to be 1, and if any one of these combinations occurs, the result is 1. This form of expression, for self-evident reasons, is known as the *sum of products* (SOP) form.

Table 1.3 a Boolean Function of Three Variables

A	B	C	D
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

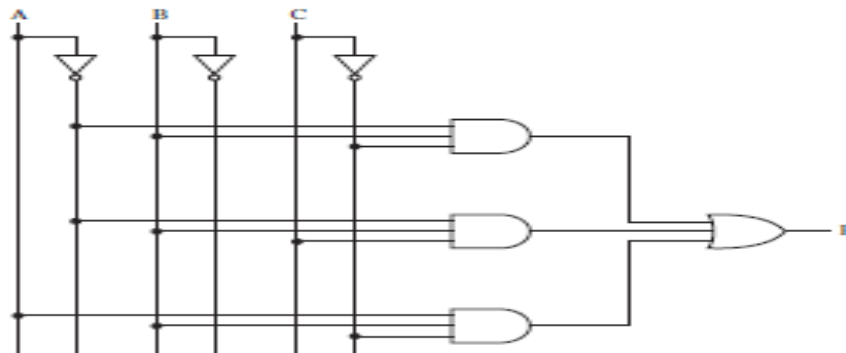


Figure 1.4 Sum-of-Products Implementation of Table 1.3

Figure 1.4 shows a straightforward implementation with AND, OR, and NOT gates. Another form can also be derived from the truth table. The SOP form expresses that the output is 1 if any of the input combinations that produce 1 is true. We can also say that the output is 1 if none of the input combinations that produce 0 is true. Thus,

$$F = (\bar{A}\bar{B}\bar{C}).(\bar{A}\bar{B}C).(\bar{A}B\bar{C}).(\bar{A}BC).(AB\bar{C}).(ABC).(A\bar{B}\bar{C}).(A\bar{B}C).(AB\bar{C}).(ABC).$$

This can be rewritten using a generalization of De Morgan's theorem:

$$(\overline{X.Y.Z}) = \bar{X} + \bar{Y} + \bar{Z} \text{ Thus,}$$

$$F = (\bar{A} + \bar{B} + \bar{C}).(\bar{A} + \bar{B} + C).(\bar{A} + B + \bar{C}).(\bar{A} + B + C).(A + \bar{B} + \bar{C}).(A + \bar{B} + C).$$

$$= (A + B + C).(A + B + \bar{C}).(\bar{A} + B + C).(\bar{A} + B + \bar{C}).(A + \bar{B} + \bar{C}).(A + \bar{B} + C).$$

This is in the *product of sums* (POS) form, which is illustrated in Figure 1.5. For clarity, NOT gates are not shown. Rather, it is assumed that each input signal and its complement are available. This simplifies the logic diagram and makes the inputs to the gates more readily apparent.

Thus, a Boolean function can be realized in either SOP or POS form. At this point, it would seem that the choice would depend on whether the truth table contains more 1s or 0s for the output function: The SOP has one term for each 1, and the POS has one term for each 0. However, there are other considerations:

It is often possible to derive a simpler Boolean expression from the truth table than either SOP or POS.

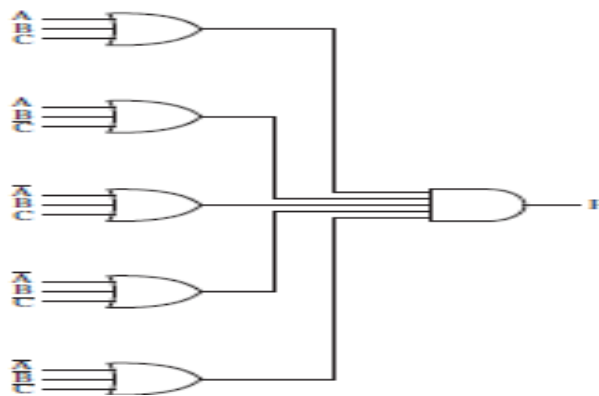


Figure 1.5 Product-of-Sums Implementation of Table 1.3

It may be preferable to implement the function with a single gate type (NAND or NOR).

The significance of the first point is that, with a simpler Boolean expression, fewer gates will be needed to implement the function. Three methods that can be used to achieve simplification are

- ✓ Algebraic simplification
- ✓ Karnaugh maps
- ✓ Quine–McKluskey tables

1.4 Sequential Circuits

Combinational circuits implement the essential functions of a digital computer. However, except for the special case of ROM, they provide no memory or state information, elements also essential to the operation of a digital computer. For the latter purposes, a more complex

form of digital logic circuit is used: the sequential circuit. The current output of a sequential circuit depends not only on the current input, but also on the past history of inputs. Another and generally more useful way to view it is that the current output of a sequential circuit depends on the current input and the current state of that circuit.

In this section, we examine some simple but useful examples of sequential circuits. As will be seen, the sequential circuit makes use of combinational circuits.

1.5 Flip-Flops

The circuit's stored information about the previous history of input is called storage or memory elements. A primitive storage element can be constructed from a small number of gates connecting the outputs back as inputs. These circuits are binary cells capable of storing one bit of information. They have two outputs, one for the normal value and one for the complement value of bit stored in it. Primitive memory elements actually fall into two broad classes: *latches* and *flip-flop*.

If a latch has only data inputs, it is called an *unlocked latch* (or only latch). Level-sensitive latches have an additional enable input, sometimes called the *clock*. Level-sensitive latches continuously sample their inputs when they are enabled. Any change in the level of the input is propagated through to the output. When the enable signal is unasserted, the last value of the inputs determines the state held by the latch.

Flip-flops differ from latches in that their output change only with respect to the clock, whereas latches change output when their inputs change. Flip-flops are characterized on the basis of the clock transition that causes the output change: there are *positive edge-triggered*, *negative edge-triggered*, and *master/slave* flip-flops.

A positive edge-triggered flip-flop samples its inputs on the low-to-high clock transition. A negative edge-triggered flip-flop works in a similar fashion, with the input sampled on the high-to-low clock transition. A master-slave flip-flop is constructed from two stage separate flip-flops. The first stage (first flip-flop) samples the inputs on the rising edge of a clock signal. The second stage transfers them to the output on the falling edge of the clock signal.

These circuits have two additional control inputs. These are *preset* and *clear*, which force the output of the flip-flop or latch to the logic-1 or logic-0 state, respectively, independent of the flip-flop or latch inputs [2].

S-R Latch:

An S-R (Set-Reset) latch is the simplest possible memory element and it is constructed by feeding the outputs of two NOR gates back to the other NOR gates input. The inputs R and S are referred to as the Reset and Set inputs, respectively. To understand the operation of the S-R latch considers the following scenarios:

S=1 and R=0: The output of the bottom NOR gate is equal to zero $Q' = 0$. Hence both inputs to the top NOR gate are equal to zero, thus, $Q = 1$. Hence, the input combination S=1 and R=0 leads to the latch being set to $Q = 1$.

S=0 and R=1: Similar to the arguments above, the outputs become $Q' = 1$ and $Q = 0$.

We say that the latch is reset.

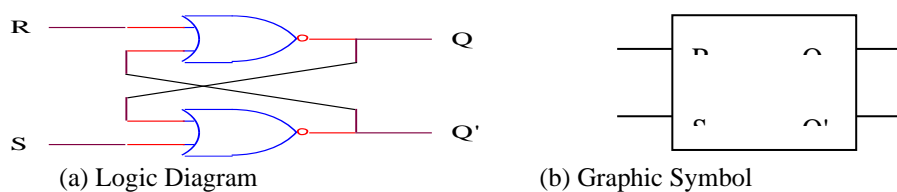
S=0 and R=0: Assume the latch is set ($Q' = 0$ and $Q = 1$), then the output of the top NOR gate remains at $Q = 1$ and the bottom NOR gate stays at $Q' = 0$.

Similarly, when the latch is in a reset state ($Q' = 1$ and $Q = 0$), it will remain there with this input combination. Therefore, with inputs S=0 and R=0, the latch remains in its state.

S=1 and R=1: This input combination must be avoided

The logic diagram and graphic symbol are shown in Figure.4.1.

The following truth table can be summarized the operation of the S-R latch.

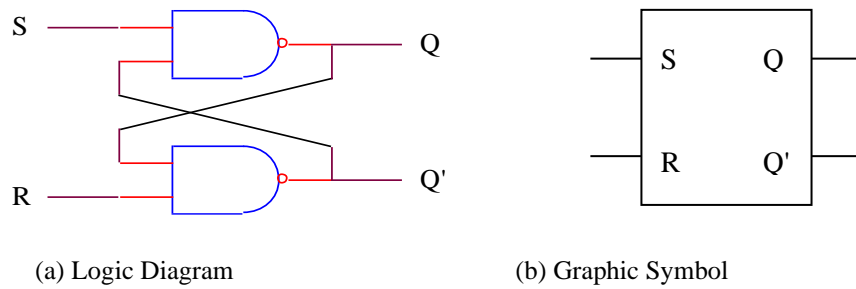


S	R	Q	Q'	Comment
0	0	Q	Q'	Hold State
0	1	0	1	Reset
1	0	1	0	Set
1	1	-	-	Forbidden

(c) Truth table

Figure 1.6 S-R latches with NOR gates

A S-R latch can also be constructed from NAND gates. The graphic symbol, logic diagram, and truth table of the latch are shown in Figure 1.7.



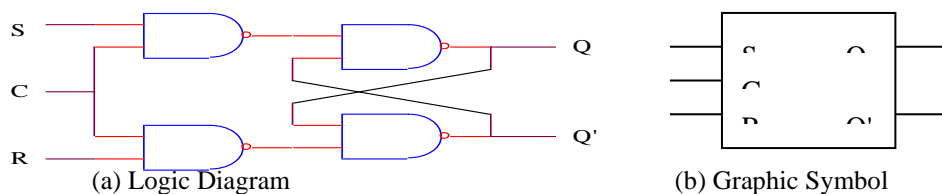
S	R	Q	Q'	Comment
1	1	Q	Q'	Hold State
0	1	1	0	Set
1	0	0	1	Reset
0	0	-	-	Forbidden

(c) Truth table

Figure 1.7 S-R latch with NAND gates.

Level Sensitive (Clock) S-R Latch

The operation of the S-R latch can be modified by providing an additional control input that determines when the state of the circuit is to be changed. The logic diagram, graphic symbol, and truth table of level sensitive S-R latch are shown in Figure 1.8.



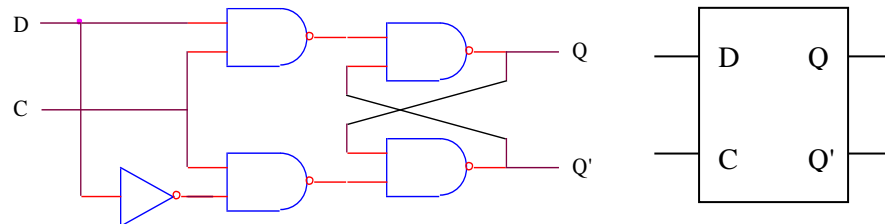
S	R	C	Q	Q'	Comment
0	0	1	Q	Q'	Hold State
0	1	1	0	1	Reset
1	0	1	1	0	Set
1	1	1	-	-	Forbidden
X	X	0	Q	Q'	Hold State

(c) Truth table

Figure: 1.8 Level Sensitive S-R latches with NAND gates

Level Sensitive (Clock) D (Delay) Latch

One way to eliminate the undesirable condition of the indeterminate state in the S-R latch is to ensure that inputs S and R are never equal to 1 at the state time. This is done level sensitive D latch shown in Figure 1.9. The latch has only two inputs: D and C. The D inputs connect directly to the S input and its complement is applied to the R input. The D input is sampled when C is equal to 1. If D is equal to 1, the Q output goes to 1. If D is equal to 0, the Q output goes to 0. If C is equal to 0, the Q output remains in its previous state [1].



(a) Logic Diagram

(b) Graphic Symbol

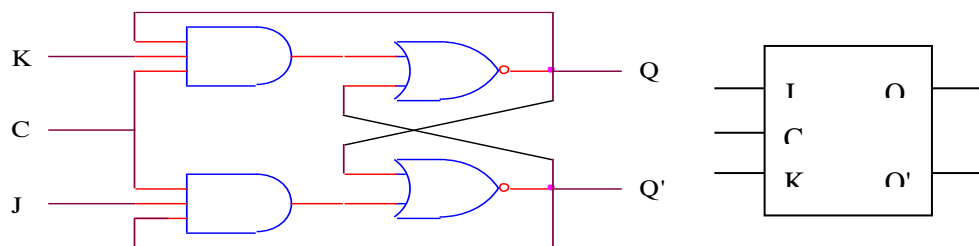
C	D	Q	Q'
1	0	0	1
1	1	1	0
0	X	Q	Q'

(c) Truth table

Figure: 1.9 Level Sensitive D latch with NAND gates

Level Sensitive (Clock) J-K Latch

A level sensitive J-K latch shown in Figure.4.5 is a refinement of the S-R latch in that the indeterminate state of the S-R type is defined in the J-K type. Inputs J and K behave like inputs S and R to set and clear the latch, respectively. The input marked J is for set and the input marked K is for reset. When the both inputs J and K are equal to 1, the latch switches to its complement state, that is, if $Q=1$, it switches to $Q=0$, and vice versa. If the C is equal to 0, the output of the latch remains in its previous state [1].



(a) Logic Diagram

(b) Graphic Symbol

C	J	K	Q	Q'	Comment
1	0	0	Q	Q'	Hold
1	0	1	0	1	Reset
1	1	0	1	0'	Set
1	1	1	Q'	Q	Toggle
0	x	X	Q	Q'	Hold

(c) Truth table

Figure: 1.10 Level Sensitive J-K latches

D Flip-Flop

Positive-Edge Triggered:

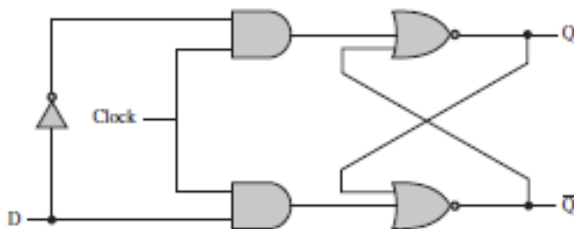


Figure 1.11 D flip flop

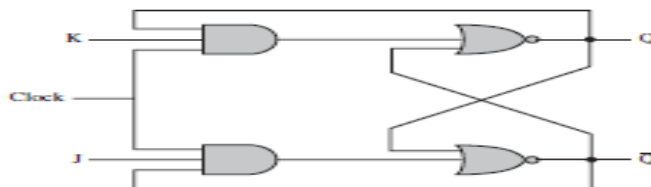


Figure 1.12 J-K flip flops

J-K Flip-Flop

J-K FLIP-FLOP is another useful flip-flop is the J-K flip-flop. Like the S-R flip-flop, it has two inputs. However, in this case all possible combinations of input values are valid. Figure 1.12 shows a gate implementation of the J-K flip-flop.

Note that the first three combinations are the same as for the S-R flip-flop. With no input asserted, the output is stable. If only the J input is asserted, the result is a set function, causing the output to be 1; if only the K input is asserted, the result is are set function, causing the output to be 0. when both J and K are 1, the function performed is referred to as the toggle function: the output is reversed. Thus, if Q is 1, 1 is applied to J and K, and then Q becomes 0.

Summary

Computers are implementations of Boolean logic. Any Boolean functions can be represented as truth tables. Truth tables provide us with a means to express the characteristics of Boolean functions as well as logic circuits. There is a one-to-one correspondence between a Boolean function and its digital representation. From a chip designer's point of view, the two most important factors are **speed and cost: minimizing the circuits** helps to both lower the cost and increase performance. Computer circuits consist of combinational logic circuits and sequential logic circuits. Combinational circuits produce outputs (almost) immediately when their inputs change. Sequential circuits require clocks to control their changes of state.

Combinational logic devices, such as adders, decoders, and multiplexers, produce outputs that are based strictly on the current inputs. The AND, OR, and NOT gates are the building blocks for combinational logic circuits, although universal gates, such as NAND and NOR, could also be used. Sequential logic devices, such as registers, counters, and memory, produce outputs based on the combination of current inputs and the current state of the circuits. These circuits are built using SR, D, and JK flip-flops.

Review Questions & Problems

1. Write short about flip flop
2. Logic gates with a set of input and outputs is arrangement of
(A) Combinational circuit (B) Logic circuit (C) Design circuits (D) Register
3. The circuit used to store one bit of data is known as
(A) Register (B) Encoder (C) Decoder (D) Flip Flop
4. Simplify $A + AB = A + B$
 $= A + (AB) = A + B$
 $AA + A + B = A + B = A + A + B$

References

1. Brown, S., and Rose, S. "Architecture of FPGAs and CPLDs: A Tutorial." *IEEE Design and Test of Computers*, Vol. 13, No. 2, 1996.
2. Farhat, H. *Digital Design and Computer Organization*. Boca Raton: CRC Press, 2004.
3. Gregg, J. *Ones and Zeros: Understanding Boolean algebra, Digital Circuits, and the Logic of Sets*. New York: Wiley, 1998.
4. Leong, p. "Recent Trends in FPGA Architectures and Applications." *Proceedings, 4th IEEE International symposium on Electronic Design, Test, and Applications*, 2008.

CHAPTER TWO: NUMBER SYSTEM AND CODES

Objectives

Upon completion of this chapter, students will be able to:

- ⇒ Convert a number from one number system (decimal, binary, hexadecimal) to its equivalent in one of the other number systems.
- ⇒ Cite the advantages of the hexadecimal number system.
- ⇒ Represent decimal numbers using the BCD code
- ⇒ Understand the difference between BCD and straight binary.
- ⇒ Understand the purpose of alphanumeric codes such as the ASCII code.

Key concepts

- The **binary number system** is a positional system where each binary digit (bit) carries a certain weight based on its position relative to the LSB.
- **Hexadecimal number system** uses base 16. Thus, it has 16 possible digit symbols. It uses the digits 0 through 9 plus the letters A, B, C, D, E, and F as the 16 digit symbols.
- The most widely used alphanumeric code is the American Standard Code for Information Interchange (**ASCII**).

Introduction

Every computer stores numbers, letters, & other special characters in a coded form not as it is. When we type some letters or words the computer translates them in numbers as computers that can understand only numbers. Computer can understand positional number system where there are only few symbols represent different values depending on the position they occupy in the number. The value of each digit in a number can be determined using Digit, the position of the digit in the number and the base of the number system (where the base is defined as the total number of digits available in the number system. there are number systems that are available in computer data representation these are decimal number system, binary number system, octal number system and hexadecimal number system.

2.1 The Decimal Number System

In everyday life we use a system based on decimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) to represent numbers and refer to the system as the decimal system. Consider what the number 83 means. It means eight tens plus three: $83 = (8 * 10) + 3$

The number 4728 means four thousands, seven hundreds, two teens, plus eight:

$$4728 = (4 * 1000) + (7 * 100) + (2 * 10) + 8$$

The decimal system is said to have a **base**, or **radix**, of 10. This means that each digit in the number is multiplied by 10 raised to a power corresponding to that digit's position:

$$83 = (8 * 10^1) + (3 * 10^0)$$

$$4728 = (4 * 10^3) + (7 * 10^2) + (2 * 10^1) + (8 * 10^0)$$

The same principle holds for decimal fractions but negative powers of 10 are used. Thus, the decimal fraction 0.256 stands for 2 tenths plus 5 hundredths plus 6 thousandths:

$$0.256 = (2 * 10^{-1}) + (5 * 10^{-2}) + (6 * 10^{-3})$$

A number with both an integer and fractional part has digits raised to both positive and negative powers of 10:

$$472.256 = (4 * 10^2) + (7 * 10^1) + (2 * 10^0) + (2 * 10^{-1}) + (5 * 10^{-2}) + (6 * 10^{-3})$$

In general, for the decimal representation of $X = \{\dots d_2, d_1, d_0.d_{-1}d_{-2}d_{-3}\}$ the value of $X =$

$$x = \sum_i (d_i \times 10^i)$$

2.2 The Binary System

In the decimal system, 10 different digits are used to represent numbers with a base of 10. In the binary system, we have only two digits, 1 and 0. Thus, numbers in the binary system are represented to the base 2.

To avoid confusion, we will sometimes put a subscript on a number to indicate its base. For example, 83_{10} and 4728_{10} are numbers represented in decimal notation or, more briefly, decimal numbers. The digits 1 and 0 in binary notation have the same meaning as in decimal notation:

$$02 = 010, 12 = 110$$

To represent larger numbers, as with decimal notation, each digit in a binary number has a value depending on its position:

$$102 = (1 * 2^1) + (0 * 2^0) = 210$$

$$112 = (1 * 2^1) + (1 * 2^0) = 310$$

$$1002 = (1 * 2^2) + (0 * 2^1) + (0 * 2^0) = 410$$

and so on. Again, fractional values are represented with negative powers of the radix:

$$1001.101 = 2^3 + 2^0 + 2^{-1} + 2^{-3} = 9.625_{10}$$

In general, for the binary representation of $Y = (\dots b_2, b_1, b_0, b_{-1}, b_{-2}, b_{-3})$ the value of y

$$y = \sum_i (b_i \times 2^i)$$

2.3 Converting Between Binary and Decimal

In fact, we showed several examples in the previous subsection. All that is required is to multiply each binary digit by the appropriate power of 2 and add the results. To convert from decimal to binary, the integer and fractional parts are handled separately.

Integers

For the integer part, recall that in binary notation, an integer represented by

$b_{m-1}, b_{m-2}, \dots, b_2, b_1, b_0$ where $b_i = 0 \text{ or } 1$ has the value

$$(b_{m-1} * 2^{m-1}) + (b_{m-2} * 2^{m-2}) + \dots + (b_1 * 2^1) + b_0$$

Suppose it is required to convert a decimal integer N into binary form. If we divide N by 2, in the decimal system, and obtain a quotient N_1 and a remainder R_0 we may write

$$N = 2 * N_1 + R_0 \text{ where } R_0 = 0 \text{ or } 1$$

Next, we divide the quotient N_1 by 2. Assume that the new quotient is N_2 and the new remainder R_1 . Then $N_1 = 2 * N_2 + R_1$ where $R_1 = 0 \text{ or } 1$ so that

$$N = 2(2N_2 + R_1) + R_0 = (N_2 * 2^2) + (R_1 * 2^1) + R_0$$

$$\text{If next } N_2 = 2N_3 + R_2 \text{ we have}$$

$$N = (N_3 * 2^3) + (R_2 * 2^2) + (R_1 * 2^1) + R_0$$

Because $N > N_1 > N_2 \dots$ continuing this sequence will eventually produce a quotient $N_{m-1} = 1$ (Except for the decimal integers 0 and 1, whose binary equivalents are 0 and 1, respectively) and a remainder R_{m-2} which is 0 or 1. Then

$N = (1 * 2^{m-1}) + (R_{m-2} * 2^{m-2}) + \dots + (R_2 * 2^2) + (R_1 * 2^1) + R_0$ which is the binary form of N . Hence, we convert from base 10 to base 2 by repeated divisions by 2. The remainders and the final quotient, 1, give us, in order of increasing significance, the binary digits of N . Figure 2.1 shows two examples.

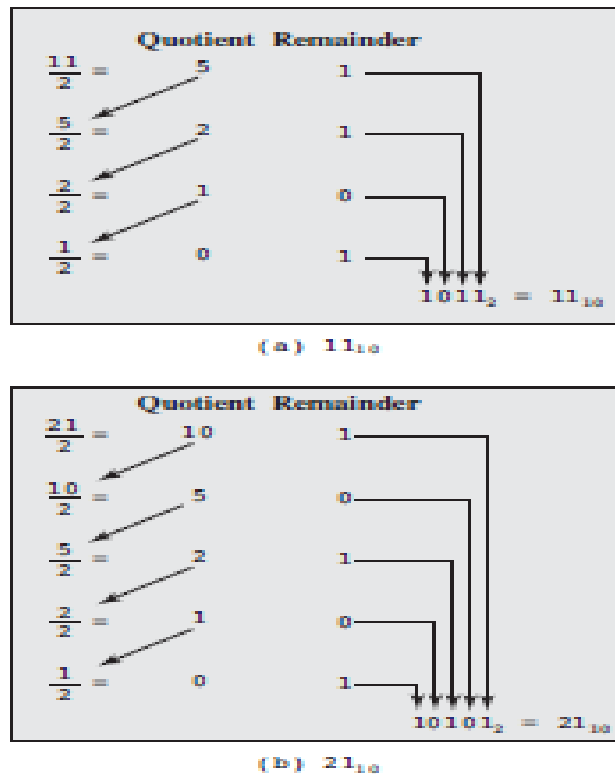


Figure 2.1 Examples of Converting from Decimal Notation to Binary Notation for Integers

Fractions

For the fractional part, recall that in binary notation, a number with a value between 0 and 1 is represented by $0.b_1b_2b_3\dots b_i$ where $b_i = 0$ or 1 . $0.b_1b_2b_3\dots b_i = (b_1 * 2^{-1}) + (b_2 * 2^{-2}) + (b_3 * 2^{-3})\dots$

This can be rewritten as $2^{-1} * (b_1 + 2^{-1} * (b_2 + 2^{-1} * (b_3 + \dots)))$

This expression suggests a technique for conversion. Suppose we want to convert the number F (0 < F < 1) from decimal to binary notation. We know that F can be expressed in the form $F = 2^{-1} * (b_1 + 2^{-1} * (b_2 + 2^{-1} * (b_3 + \dots)))$

If we multiply F by 2, we obtain, $2 * F = b_1 + 2^{-1} * (b_2 + 2^{-1} * (b_3 + \dots))$

From this equation, we see that the integer part of $(2 * F)$, which must be either 0 or 1 because $0 < F < 1$, is simply b_1 . So we can say $(2 * F) = b_1 + F_1$, where $0 < F_1 < 1$ and where $F_1 = 2^{-1} * (b_2 + 2^{-1} * (b_3 + 2^{-1} * (b_4 + \dots)))$

To find b_2 we repeat the process. Therefore, the conversion algorithm involves repeated multiplication by 2. At each step, the fractional part of the number from the previous step is multiplied by 2. The digit to the left of the decimal point in the product will be 0 or 1 and contributes to the binary representation, starting with the most significant digit. The fractional part of the product is used as the multiplicand in the next step.

This process is not necessarily exact; that is, a decimal fraction with a finite number of digits may require a binary fraction with an infinite number of digits. In such cases, the conversion algorithm is usually halted after a pre-specified number of steps, depending on the desired accuracy.

Hexadecimal Notation

Because of the inherent binary nature of digital computer components, all forms of data within computers are represented by various binary codes. However, no matter how convenient the binary system is for computers, it is exceedingly cumbersome for human beings. Consequently, most computer professionals who must spend time working with the actual raw data in the computer prefer a more compact notation.

What notation to use? One possibility is the decimal notation. This is certainly more compact than binary notation, but it is uncomfortable because of the tediousness of converting between base 2 and base 10.

Table 2.1 Examples of Converting from Decimal Notation to Binary Notation for Fractions

Product	Integer Part		0.110011_2
$0.81 \times 2 = 1.62$	1		
$0.62 \times 2 = 1.24$	1		
$0.24 \times 2 = 0.48$	0		
$0.48 \times 2 = 0.96$	0		
$0.96 \times 2 = 1.92$	1		
$0.92 \times 2 = 1.84$	1		

(a) $0.81_{10} = 0.110011_2$ (approximately)

Product	Integer Part		0.01_2
$0.25 \times 2 = 0.5$	0		
$0.5 \times 2 = 1.0$	1		

(b) $0.25_{10} = 0.01_2$ (exactly)

Instead, a notation known as hexadecimal has been adopted. Binary digits are grouped into sets of four. Each possible combination of four binary digits is given a symbol, as follows:

0000 = 0	0100 = 4	1000 = 8	1100 = C
0001 = 1	0101 = 5	1001 = 9	1101 = D
0010 = 2	0110 = 6	1010 = A	1110 = E
0011 = 3	0111 = 7	1011 = B	1111 = F

Because 16 symbols are used, the notation is called hexadecimal, and the 16 symbols are the hexadecimal digits.

$$\begin{aligned}
 2C16 &= (216 * 161) + (C16 * 160) \\
 &= (210 * 161) + (1210 * 160) = 44
 \end{aligned}$$

Table 2.2 Decimal, Binary, and Hexadecimal

Decimal (base 10)	Binary (base 2)	Hexadecimal (base 16)
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F
16	0001 0000	10
17	0001 0001	11
18	0001 0010	12
31	0001 0000	1F
100	0110 0100	64
255	1111 0000	FF
256	0001 0000 0000	100

Hexadecimal notation is used not only for representing integers. It is also used as a concise notation for representing any sequence of binary digits, whether they represent text, numbers, or some other type of data. The reasons for using hexadecimal notation are:

1. It is more compact than binary notation.
2. In most computers, binary data occupy some multiple of 4 bits, and hence some multiple of a single hexadecimal digit.
3. It is extremely easy to convert between binary and hexadecimal.

As an example of the last point, consider the binary string 110111100001. This is equivalent to

$$\begin{array}{ccccccc} 1101 & 1110 & 0001 & = & DE1_{16} \\ D & E & 1 & & \\ - & - & - & - & - \end{array}$$

This process is performed so naturally that an experienced programmer can mentally convert visual representations of binary data to their hexadecimal equivalent without written effort

2.4 Data Types

Data stored in memory is a string of bits (0 or 1).

Byte	8 bit	-128 to 127
Short	16 bit	-32,768 to 32,727
Int	32 bit	2,147,483,648 to 2,147,483,647
Long	64 bit	-9x10 ¹⁸ to 9x10 ¹⁸
Float	32 bit	±10 ⁻⁴⁵ to ±10 ³⁸ , 7 significant digits
Double	64 bit	±10 ⁻³²⁴ to ±10 ³⁰⁸ , 15 significant digits
Char	single characters	'a', 'B', '&', '6'
Boolean	logical values	true, false

2.5 Complements

It is possible to avoid this subtraction process by using a complement representation for negative numbers. This will be discussed specifically for binary fractions, although it is easy to extend the complement techniques to integers and mixed numbers. 1's Complement

For positive values, the ones' complement representation is identical to the binary representation. For instance, the value 75 is represented as $(01001011)_2$ in an 8-bit binary system, as well as $(01001011)_{1s}$ in an 8-bit 1's complement system.

What about negative values? Given a number X which can be expressed as an n -bit binary number, its negated value, $-X$, can be obtained in 1's complement form by this formula:

$$-X = 2^n - X - 1$$

For example, 75 is represented as $(01001011)_2$ in an 8-bit binary system, and hence -75 is represented as $(10110100)_{1s}$ in the 8-bit 1's complement system. (Note that $2^8 - 75 - 1 = 180$ whose binary form is 10110100.) We observe that we can easily derive $-X$ from X in 1's complement by inverting all the bits in the binary representation of X . Note also that the first bit serves very much like a sign bit, indicating that the value is positive (negative) if the first bit is 0 (1). However, the remaining bits do not constitute the magnitude of the number, particularly for negative numbers.

An 8-bit 1's complement representation allows values between -127 (represented as $(10000000)_{1s}$) and $+127$ (represented as $(01111111)_{1s}$) to be represented. There are two representations for zero: $(00000000)_{1s}$ and $(11111111)_{1s}$. In general, an n -bit 1's complement representation has a range $[-(2^{n-1} - 1), 2^{n-1} - 1]$. To negate a value, we invert all the bits. For example, in an 8-bit 1's complement scheme, the value 14 is represented as $(00001110)_{1s}$, therefore -14 is represented as $(11110001)_{1s}$.

2's Complement

The two's complement system share some similarities with the ones' complement system. For positive values, it is the same as the binary representation. The first bit also indicates the sign of the value (0 for positive, 1 for negative). Given a number X which can be expressed as an n -bit binary number, its negated value, $-X$, can be obtained in 2's complement form by this formula:

$$-X = 2^n - X$$

For example, 75 is represented as $(01001011)_2$ in an 8-bit binary system, and hence -75 is

represented as $(10110101)_{2s}$ in the 8-bit 2's complement system. (Note that $2^8 - 75 = 181$ whose binary form is 10110101.) Again, we observe that we can easily derive $-X$ from X in 2's complement by inverting all the bits in the binary representation of X , and then adding one to it.

An 8-bit 2's complement representation allows values between -128 (represented as $(10000000)_{2s}$) and $+127$ (represented as $(01111111)_{2s}$) to be represented, and hence it has a range that is one larger than that of the 1's complement representation. This is due to the fact that there is only one unique representation for zero: $(00000000)_{2s}$. In general, an n -bit 2's complement representation has a range $[-2^{n-1}, 2^{n-1} - 1]$.

To negate a value, we invert all the bits and plus 1. For example, in an 8-bit 2's complement scheme, the value 14 is represented as $(00001110)_{2s}$, therefore -14 is represented as $(11110010)_{2s}$.

2.6 Floating-Point Representation

The simple fixed-point representation allows for a very limited range of values that can be represented. A more versatile scheme, the floating-point representation, overcomes this weakness. The radix point in this format is 'floating', as we adopt the concept of scientific notation. Some examples of values written in the scientific notation are shown below.

Examples:

Decimal values expressed in scientific notation.

$$0.123 \times 10^{30} = 123,000,000,000,000,000,000,000,000$$

$$0.5 \times 10^{-17} = 0.000\ 000\ 000\ 000\ 000\ 005$$

$$-98.765 \times 10^{-7} = -0.000\ 009\ 876\ 5$$

Binary values expressed in scientific notation.

$$101.101 \times 2^{-7} = 0.0000101101$$

$$110.011 \times 2^8 = 11001100000$$

The scientific notation comprises the *mantissa* (also called significant), the *base* (or *radix*) and the *exponent*. Figure 2-2 shows the three components of a decimal number in scientific notation.

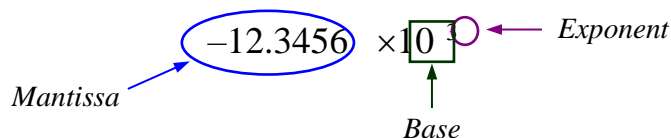


Figure 2.2 Scientific Notation.

These representations: -12.3456×10^3 , -123456×10^{-1} , and -0.00123456×10^7 all represent the same value -12345.6 . To achieve consistency and storage efficiency, we adopt a form where the mantissa is said to be *normalized*. A normalized mantissa is one in which the digit immediately after the radix point is non-zero. Therefore, the normalized form of our example would be -0.123456×10^5 .

The scientific notation is implemented as floating-point representation. The sign-and-magnitude format is usually chosen to represent the mantissa, and the base (radix) is assumed to be two, since numbers are represented in binary in computers. The floating-point format hence consists of three fields: a sign bit, a mantissa field, and an exponent field (the relative positions of the mantissa field and exponent field, as well as their widths, may vary from system to system), as shown in Figure 2-3

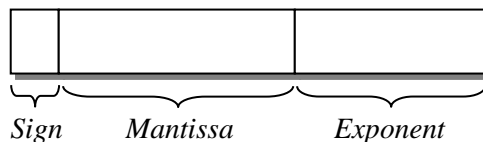


Figure 2.3 A Floating-Point Representation

The mantissa is usually normalized, so the first bit in the mantissa field must be a 1. Indeed, some systems further save on this bit, and store the mantissa only from the second bit onwards. The exponent field may be implemented in different schemes, such as sign-and-magnitude, 1's complement, 2's complement, or excess.

More bits for mantissa offers better precision, while a bigger exponent field provides a larger range of values. There is hence a trade-off between precision and range in determining the widths of these two fields.

The examples below assume a floating-point representation with 1-bit sign, 5-bit normalized mantissa, and 4-bit exponent for the value $(0.1875)_{10}$, under the various schemes for exponent.

Examples:

Convert the given value to binary and express it in scientific notation with normalized mantissa:

$$(0.1875)_{10} = (0.0011)_2 = 0.11 \times 2^{-2}$$

If exponent is in 1's complement format:

0	11000	1101
---	-------	------

If exponent is in 2's complement format:

0	11000	0110
---	-------	------

If exponent is in excess-8 format:

0	11000	1110
---	-------	------

Multiplication on Floating-Point Numbers

The steps for multiplying two numbers A and B in floating-point representation are:

1. Multiply the mantissas of A and B .
2. Add the exponents of A and B .
3. Normalize if necessary.

For example: if $A = (0.11)_2 \times 2^4$ and $B = (0.101)_2 \times 2^{-1}$, then $A \times B = (0.11 \times 0.101) \times 2^{4-1} = (0.01111)_2 \times 2^3 = (0.1111)_2 \times 2^2$. This illustrates the multiplication of two decimal values 12 and 5/16 to obtain 3.75.

Addition on Floating-Point Numbers

The steps for adding two numbers A and B in floating-point representation are:

1. Equalize the exponents of A and B .
2. Add the mantissas of A and B .
3. Normalize if necessary.

For example: if $A = (0.11)_2 \times 2^4$ and $B = (0.101)_2 \times 2^{-1}$, then we convert A to $(11000)_2 \times 2^{-1}$, so $A + B = (11000 + 0.101) \times 2^{-1} = (11000.101)_2 \times 2^{-1} = (0.11000101)_2 \times 2^4$. This illustrates the addition of two decimal values 12 and 5/16 to obtain 12.3125.

2.7 Fixed-Point Representation

In representing real numbers in the computer, the radix point is not explicitly stored. One strategy is to assume a position of the radix point, which divides the field into two portions, one for the integral part and the other the fractional part. This is known as *fixed-point representation*. For example, Figure 2-4 shows the value $(21.75)_{10}$ or $(010101.11)_2$, in an 8-bit sign-and-magnitude scheme with two bits allocated to the fraction part.

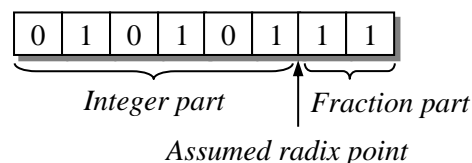


Figure 2.4 An 8-bit Sign-and-Magnitude Fixed-Point Representation

It is apparent that the number of bits allocated to the representation determines the maximum number of values (up to 2^n values for an n -bit representation), the width of the integral portion

determines the range of values, and the width of the fractional portion determines the *precision* of the values. It is the finite size of the fractional portion that results in imprecise representation of certain numeric values, and computational error due to truncation or rounding. Due to the discrete nature of computer, real numbers are only approximated in their computer representation.

2.8 Codes

2.8.1 BCD Code

When numbers, letters, or words are represented by a special group of symbols, we say that they are being encoded, and the group of symbols is called a *code*. Probably one of the most familiar codes is the Morse code, where a series of dots and dashes represents letters of the alphabet. We have seen that any decimal number can be represented by an equivalent binary number. The group of 0s and 1s in the binary number can be thought of as a code representing the decimal number. When a decimal number is represented by its equivalent binary number, we call it **straight binary coding**.

Digital systems all use some form of binary numbers for their internal operation, but the external world is decimal in nature. This means that conversions between the decimal and binary systems are being performed often. We have seen that the conversions between decimal and binary can become long and complicated for large numbers. For this reason, a means of encoding decimal numbers that combines some features of both the decimal and the binary systems is used in certain situations.

Binary-Coded-Decimal Code

If *each* digit of a decimal number is represented by its binary equivalent, the result is a code called binary-coded-decimal (hereafter abbreviated BCD). Since a decimal digit can be as large as 9, four bits are required to code each digit (the binary code for 9 is 1001). To illustrate the BCD code, take a decimal number such as 874. Each *digit* is changed to its binary equivalent as follows:

8	7	4	(decimal)
↓	↓	↓	
1000	0111	0100	(BCD)

As another example, let us change 943 to its BCD-code representation:

9	4	3	(decimal)
↓	↓	↓	
1001	0100	0011	(BCD)

Once again, each decimal digit is changed to its straight binary equivalent. Note that four bits are *always* used for each digit. The BCD code, then, represents each digit of the decimal

number by a four-bit binary number. Clearly only the four-bit binary numbers from 0000 through 1001 are used. The BCD code does not use the numbers 1010, 1011, 1100, 1101, 1110, and 1111. In other words, only 10 of the 16 possible four-bit binary code groups are used. If any of the “forbidden” four-bit numbers ever occurs in a machine using the BCD code, it is usually an indication that an error has occurred.

Example 1: Convert 0110100000111001 (BCD) to its decimal equivalent.

Solution

Divide the BCD number into four-bit groups and convert each to decimal.

$$\begin{array}{ccccccc} \underline{0110} & \underline{1000} & \underline{0011} & \underline{1001} \\ 6 & 8 & 3 & 9 \end{array}$$

Comparison of BCD and Binary

It is important to realize that BCD is not another number system like binary, decimal, and hexadecimal. In fact, it is the decimal system with each digit encoded in its binary equivalent. It is also important to understand that a BCD number is *not* the same as a straight binary number. A straight binary number takes the *complete* decimal number and represents it in binary; the BCD code converts *each* decimal *digit* to binary individually. To illustrate, take the number 137 and compare its straight binary and BCD codes:

$$\begin{array}{ll} 137_{10} = 10001001_2 & \text{(binary)} \\ 137_{10} = 0001\ 0011\ 0111 & \text{(BCD)} \end{array}$$

The BCD code requires 12 bits, while the straight binary code requires only eight bits to represent 137. BCD requires more bits than straight binary to represent decimal numbers of more than one digit because BCD does not use all possible four-bit groups, as pointed out earlier, and is therefore somewhat inefficient. The main advantage of the BCD code is the relative ease of converting to and from decimal. Only the four-bit code groups for the decimal digits 0 through 9 needs to be remembered. This ease of conversion is especially important from a hardware standpoint because in a digital system, it is the logic circuits that perform the conversions to and from decimal.

2.8.2 Alphanumeric Codes

In addition to numerical data, a computer must be able to handle non numerical information. In other words, a computer should recognize codes that represent letters of the alphabet, punctuation marks, and other special characters as well as numbers. These codes are called alphanumeric codes. A complete alphanumeric code would include the 26 lowercase letters, 26 uppercase letters, 10 numeric digits, 7 punctuation marks, and anywhere from 20 to 40 other characters, such as $_$, $/$, $\#$, $\%$, \ast , and so on. We can say that an alphanumeric code represents all of the various characters and functions that are found on a computer keyboard.

2.8.3 ASCII Code

The most widely used alphanumeric code is the American Standard Code for Information Interchange (ASCII). The ASCII (pronounced “askee”) code is a seven-bit code, and so it has $2^7 = 128$ possible code groups. This is more than enough to represent all of the standard keyboard characters as well as the control functions such as the (RETURN) and (LINEFEED) functions. Table 2-4 shows a listing of the standard seven-bit ASCII code. The table gives the hexadecimal and decimal equivalents. The seven-bit binary code for each character can be obtained by converting the hex value to binary.

Table 2. 2 Standard ASCII codes

Character	HEX	Decimal	Character	HEX	Decimal	Character	HEX	Decimal	Character	HEX	Decimal
NUL (null)	0	0	Space	20	32	@	40	64	`	60	96
Start Heading	1	1	!	21	33	A	41	65	a	61	97
Start Text	2	2	"	22	34	B	42	66	b	62	98
End Text	3	3	#	23	35	C	43	67	c	63	99
End Transmit.	4	4	\$	24	36	D	44	68	d	64	100
Enquiry	5	5	%	25	37	E	45	69	e	65	101
Acknowledge	6	6	&	26	38	F	46	70	f	66	102
Bell	7	7	^	27	39	G	47	71	g	67	103
Backspace	8	8	(28	40	H	48	72	h	68	104
Horiz. Tab	9	9)	29	41	I	49	73	i	69	105
Line Feed	A	10	*	2A	42	J	4A	74	j	6A	106
Vert. Tab	B	11	+	2B	43	K	4B	75	k	6B	107
Form Feed	C	12	,	2C	44	L	4C	76	l	6C	108
Carriage Return	D	13	-	2D	45	M	4D	77	m	6D	109
Shift Out	E	14	.	2E	46	N	4E	78	n	6E	110
Shift In	F	15	/	2F	47	O	4F	79	o	6F	111
Data Link Esc	10	16	0	30	48	P	50	80	p	70	112
Direct Control 1	11	17	1	31	49	Q	51	81	q	71	113
Direct Control 2	12	18	2	32	50	R	52	82	r	72	114
Direct Control 3	13	19	3	33	51	S	53	83	s	73	115
Direct Control 4	14	20	4	34	52	T	54	84	t	74	116
Negative ACK	15	21	5	35	53	U	55	85	u	75	117
Synch Idle	16	22	6	36	54	V	56	86	v	76	118
End Trans Block	17	23	7	37	55	W	57	87	w	77	119
Cancel	18	24	8	38	56	X	58	88	x	78	120
End of Medium	19	25	9	39	57	Y	59	89	y	79	121
Substitute	1A	26	:	3A	58	Z	5A	90	z	7A	122
Escape	1B	27	;	3B	59	[5B	91	{	7B	123
Form separator	1C	28	<	3C	60	\	5C	92		7C	124
Group separator	1D	29	=	3D	61]	5D	93	}	7D	125
Record Separator	1E	30	>	3E	62	^	5E	94	~	7E	126
Unit Separator	1F	31	?	3F	63	_	5F	95	Delete	7F	127

Summary

The hexadecimal number system is used in digital systems and computers as an efficient way of representing binary quantities. In conversions between hex and binary, each hex digit corresponds to four bits. Using an N -bit binary number, we can represent decimal values from 0 to .The BCD code for a decimal number is formed by converting each digit of the decimal number to its four-bit binary equivalent. A byte is a string of eight bits. A nibble is four bits. The word size depends on the system. An alphanumeric code is one that uses groups of bits to represent all of the various characters and functions that are part of a typical computer's keyboard. The ASCII code is the most widely used alphanumeric code.

Review Question

1. If the ASCII code for A is 1000001, B is 1000010, and C is 1000011 then the string 100001110000011000010 represents:
A. **CAB** B. BAC C. CCB D. ABC
2. The binary representation of 15 is:
A. 01010 B. **01111** C. 10011 D. 00101
3. Floating point representation is used to store
A. Boolean values B. whole numbers C. **real numbers** D. integers
4. The two's complement representation of -10 is:
A. **11110110** B. 11011001 C. 00001010 D. 11111100

References

1. Benham, J. "A Geometric Approach to Presenting Computer Representations of Integers." *SIGCSE Bulletin*, December 1992.
2. <http://www.williamstallings.com/COA/COA8e.html>; Retrieved on June 5, 2015
3. Stallings, W. *Computer Architecture and Organizations Designing for Performance, Eighth Edition*. Upper Saddle River, NJ: Prentice Hall, 2010.

CHAPTER THREE: COMMON DIGITAL COMPONENTS

Objectives

Upon the completion of this chapter students will be able to

- ⇒ Draw decoder and encoder diagram with different given input
- ⇒ Identify computer memory units
- ⇒ Draw block diagram of RAM and ROM

Key concepts

- **Integrated circuit:** integrated circuit IC (abbreviated IC) is a small silicon semiconductor crystal, called a chip, containing the electronic components for the digital gates.
- **Decoder:** is a combinational circuit that converts binary information from the n coded inputs to a maximum of 2^n unique outputs.
- **Encoder:** is a digital circuit that performs the inverse operation of a decoder. An encoder has 2^n (or less) input lines and n output lines.
- **Register:** is a group of flip-flops with each flip-flop capable of storing one bit of information. An n -bit register has a group of n flip-flops and is capable of storing any binary information of n bits.
- **Memory unit:** is a collection of storage cells together with associated circuits needed to transfer information in and out of storage.

Introduction

The digital computer is a digital system that performs various computational tasks. Digital computers use the binary number system, which has two digits: 0 and 1. A binary digit is called a bit. A computer system is sometimes subdivided into two functional entities: hardware and software. The hardware of the computer consists of all the electronic components and electromechanical devices that comprise the physical entity of the device.

Computer software consists of the instructions and data that the computer manipulates to perform various data-processing tasks. Program is a sequence of instructions for the computer is called a program. The data that are manipulated by the program constitute the data base.

Computer organization is concerned with the way the hardware components operate and the way they are connected together to form the computer system. The various components are

assumed to be in place and the task is to investigate the organizational structure to verify that the computer parts operate as intended.

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

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

3.1 Integrated Circuits

An integrated circuit IC (abbreviated IC) is a small silicon semiconductor crystal, called a chip, containing the electronic components for the digital gates.

3.2 Decoders & Encoders

A decoder is a combinational circuit that converts binary information from the n coded inputs to a maximum of 2^n unique outputs. A decoder has n inputs and m outputs, where $m \leq 2^n$, and are called n -to- m -line decoders.

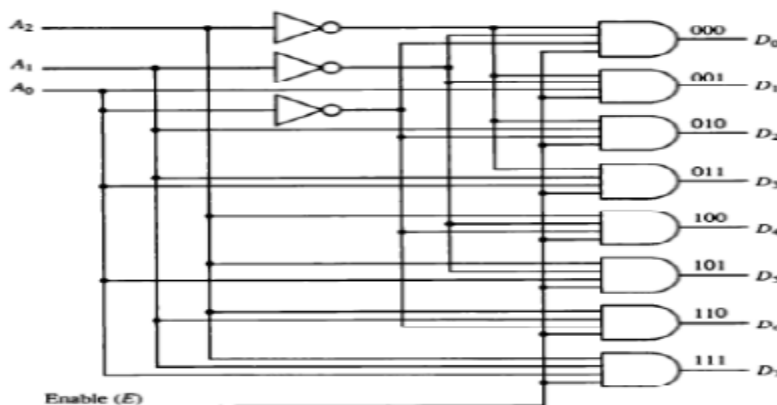


Figure 3.1 3-8 decoders

An encoder is a digital circuit that performs the inverse operation of a decoder. An encoder has 2^n (or less) input lines and n output lines.

Table 3.1 Truth table for 3 to 8 line decoder

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

These conditions can be expressed by the following Boolean functions:

$$A_0 = D_0 + D_3 + D_5 + D_7$$

$$A_1 = D_2 + D_3 + D_6 + D_7$$

$$A_2 = D_4 + D_5 + D_6 + D_7$$

3.3 Multiplexer

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 *input selection lines* determine which input data line is selected for the output.

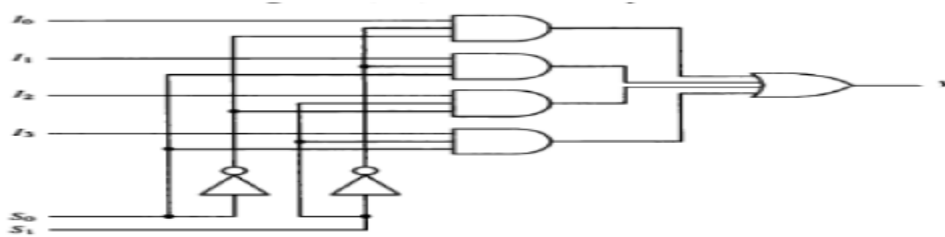


Figure 3.2 multiplexer

3.4 Registers

Register is a group of flip-flops with each flip-flop capable of storing one bit of information. A register may also have combinational gates that Form certain data-processing tasks. The flip-flops hold the data and the gates control when and how new data is transferred into the register and also flip-flops have a common clock input. Ex: 4 bit register

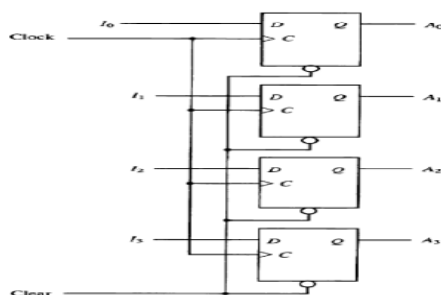


Figure 3.3 four bit register

3.5 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 a clock or an external input and input may occur at uniform intervals of time or randomly. It is used to count the number of occurrences of an event and for generating timing signals to control the sequence of operations.

A counter that follows the binary number sequence is a *binary counter*. An n -bit binary counter is a register of n flip-flops and gates that follow a sequence of states

Consider the sequence 0000, 0001, 0010, 0011, 1000 ..., the lsb (left significant bit) is complemented each count.

Every other bit is complemented if all its lower-order bits are equal to 1. Natural to use either T or JK flip-flops since they both have a complement state

- The counter has an enable input
- Synchronous counters have a regular pattern with a common clock
- The chain of AND gates generate the logic for the flip-flop inputs

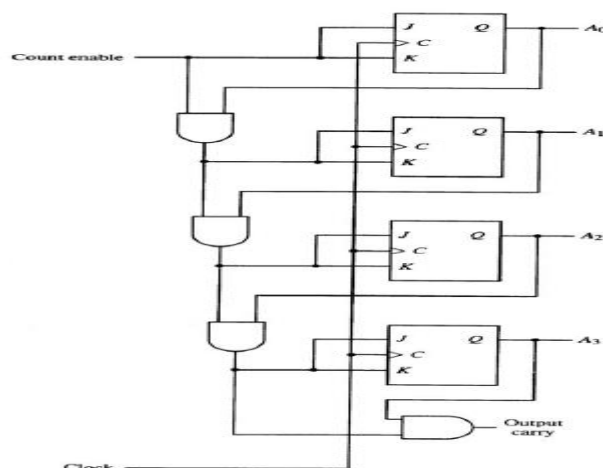


Figure 3.5 4 bit synchronous binary counters

- ✌ Counters often require a parallel load capability to transfer an initial count value
- ✌ These would then need a clear input to reset the initial value
- ✌ An input load control disables the count and allows a transfer of data
- ✌ If the clear and load inputs are both 0, and the count input is 1, then the count proceeds
- ✌ Counters with parallel load are referred to as registers with load and increment operations

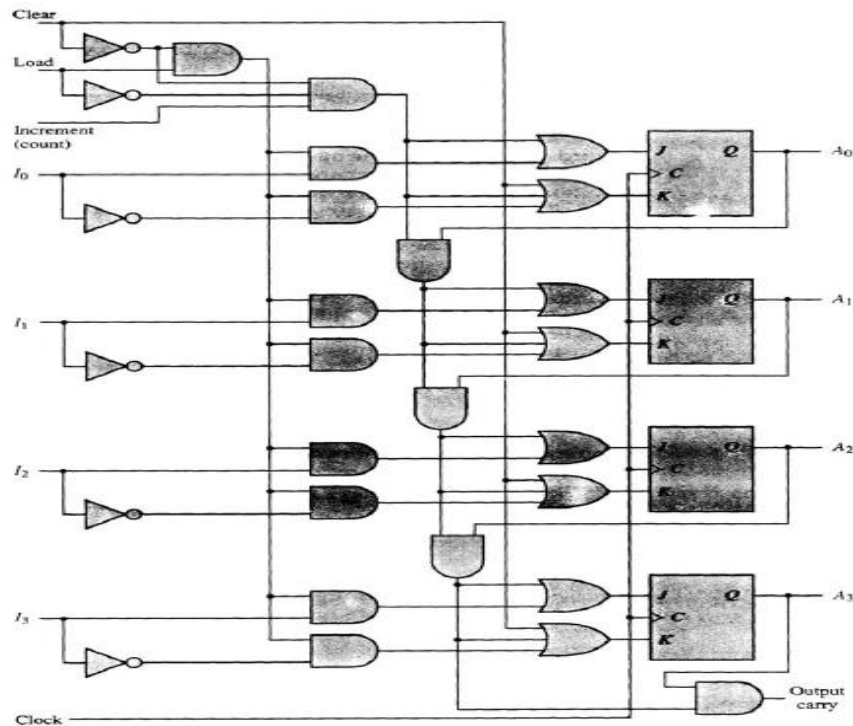


Figure 3.6 4 bit binary counters with parallel load and synchronous clear

3.6 Memory Unit

A memory unit is a collection of storage cells together with associated circuits needed to transfer information in and out of storage. The memory stores binary word information in groups of bits called words.

A word can represent an instruction code or alphanumeric characters; each word in memory is assigned an address from 0 to $2^k - 1$, where k is the number of address lines. Decoder inside the memory accepts an address opens the paths needed to select the bits of the specified word

The memory capacity is stated as the total number of bytes that can be stored and refer to the number of bytes using one of the following:

- ✓ K (kilo) = 2^{10}
- ✓ M (mega) = 2^{20}
- ✓ G (giga) = 2^{30}
- ✓ 64K = 2^{16} , 2M = 2^{21} , and 4G = 2^{32}

In random-access memory (RAM) the memory cells can be accessed for information from any desired random location. The process of locating a word in memory is the same and requires an equal amount of time no matter where the cells are located physically in memory.

Communication between memory and its environment is achieved via data input and output lines, address selections lines, and control lines.

- ⇒ The n data input lines provide the information to be stored in memory
- ⇒ The n data output lines supply the information coming out of memory
- ⇒ The k address lines provide a binary number of k bits that specify a specific word or location

The two control lines specify the direction of transfer – either read or write

Random-Access Memory

In random-access memory (RAM) the memory cells can be accessed for information transfer from any desired random location.

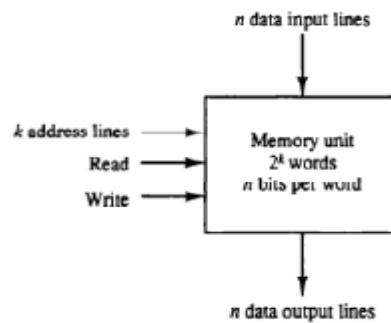


Figure 3.4 block diagram of random access memory (RAM)

Read-Only memory

The name implies a read - only memory (ROM) is a memory unit that performs the read operation only; it does not have a write capability.

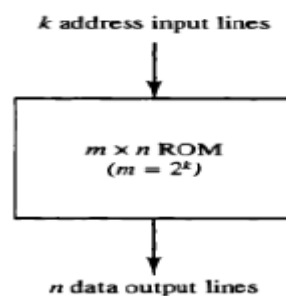


Figure 3.5 block diagram of read only memory (ROM)

Summary

Digital circuit is a circuit where the signal must be one of two discrete levels. Each level is interpreted as one of two different states (for example on/off, 0/1, true/false).digital circuit uses transistors to create logic gates in order to perform Boolean logic. This logic is the foundation of digital electronics and computer processing.

Review question

1. Write down the difference between decoder and encoder?
2. Draw the block diagram of RAM
3. Draw diagram for four bit register

References

1. Charles alexander and matthew sadiku(2004) , "*fundamentals of electric circuit*".Mcgraw-hill
2. John Hayes (1993). "*Introduction to digital logic design*". Addison Wesley.
3. Stallings, W. *Computer Architecture and Organizations Designing for Performance, Eighth Edition*. Upper Saddle River, NJ: Prentice Hall, 2010.

CHAPTER FOUR: REGISTER TRANSFER LANGUAGE AND MICRO OPERATIONS

Objective

Upon the completion of this chapter students will be able to

- ⇒ Describe what is register transfer language
- ⇒ Identify basic symbol for register transfer
- ⇒ Understand the basic difference between logic micro-operations and arithmetic micro operations

Key Concepts

- Combinational and sequential circuits can be used to create simple digital systems.
- Simple digital systems are frequently characterized in terms of the registers they contain, and the operations that they perform.
- Memory (RAM) can be thought as a sequential circuits containing some number of registers.
- Memory is usually accessed in computer systems by putting the desired address in a special register, the Memory Address Register (MAR, or AR)
- Digital systems are composed of modules that are constructed from digital components, such as registers, decoders, arithmetic elements, and control logic

Introduction

Digital system is an interconnection of digital hardware modules that accomplish a specific Information-processing task. The modules are constructed from such digital components as registers, decoders, arithmetic elements, and control logic. The various modules are interconnected with common data and control paths to form a digital computer system. The operations executed on data stored in registers are called micro operations. A micro-operation is an elementary operation performed on the information stored in one or more registers. The internal hardware organization of a digital computer is best defined by specifying:

1. The set of registers it contains and their function.
2. The sequence of micro-operations performed on the binary information stored in the registers.
3. The control that initiates the sequence of micro-operations.

4.1 Registers transfer language

The symbolic notation used to describe the micro-operation transfers register transfer among registers is called a register transfer language. The term “registers transfer” implies the

availability of hardware logic circuits that can perform a stated micro operation and transfer the result of the operation to the same or another register.

Register Transfer

Information transfer from one register to another is designated in symbolic form by means of a replacement operator. The statement $R2 \leftarrow R1$ denotes a transfer of the content of register R1 into register R2. It designates a replacement of the content of R2 by the content of R1. By definition, the content of the source register R1 does not change after the transfer.

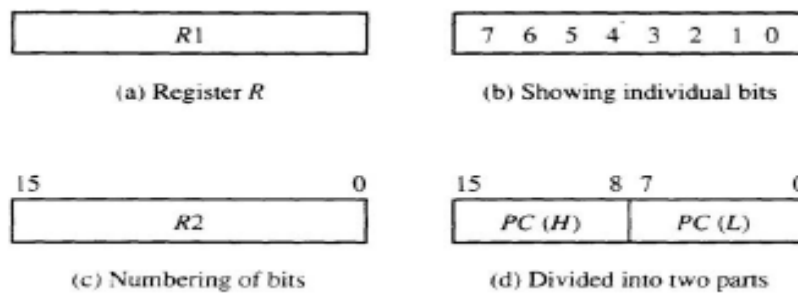


Figure 4.1 block diagram for register

If we want the transfer to occur only under a predetermined control condition. This can be shown by means of an if-then statement. If $(P = 1)$ then $(R2 \leftarrow R1)$

Control function

It is sometimes convenient to separate the control variables from the register transfer operation by specifying a control function. $P: R2 \leftarrow R1$

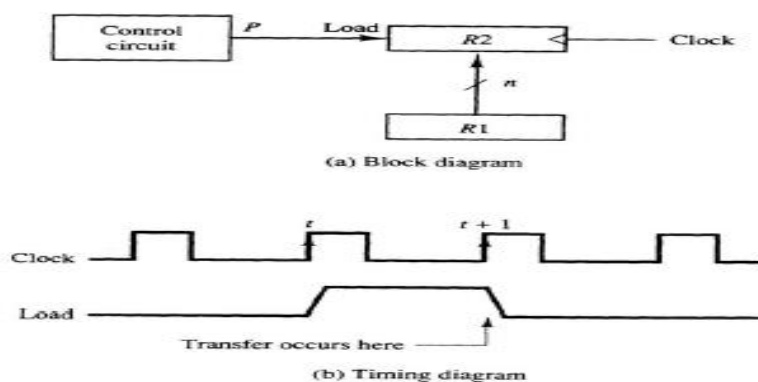


Figure 4.2 transfer from R1 to R2 when $P=1$

It is assumed that all transfers occur during a clock edge transition. All micro-operations

Written on a single line are to be executed at the same time T : $R2 \leftarrow R1, R1 \leftarrow R2$

Symbol	Description	Examples
Letters (and numerals)	Denotes a register	<i>MAR, R2</i>
Parentheses ()	Denotes a part of a register	<i>R2(0-7), R2(L)</i>
Arrow \leftarrow	Denotes transfer of information	<i>R2 \leftarrow R1</i>
Comma ,	Separates two microoperations	<i>R2 \leftarrow R1, R1 \leftarrow R2</i>

Table 4.1 basic symbol for register transfer

4.2 Bus and Memory Transfers

A typical digital computer has many registers, and paths must be provided to transfer information from one register to another. The number of wires will be excessive if separate lines are used between each register and all other registers in the system. A more efficient scheme for transferring information between common bus registers in a multiple-register configuration is a common bus system.

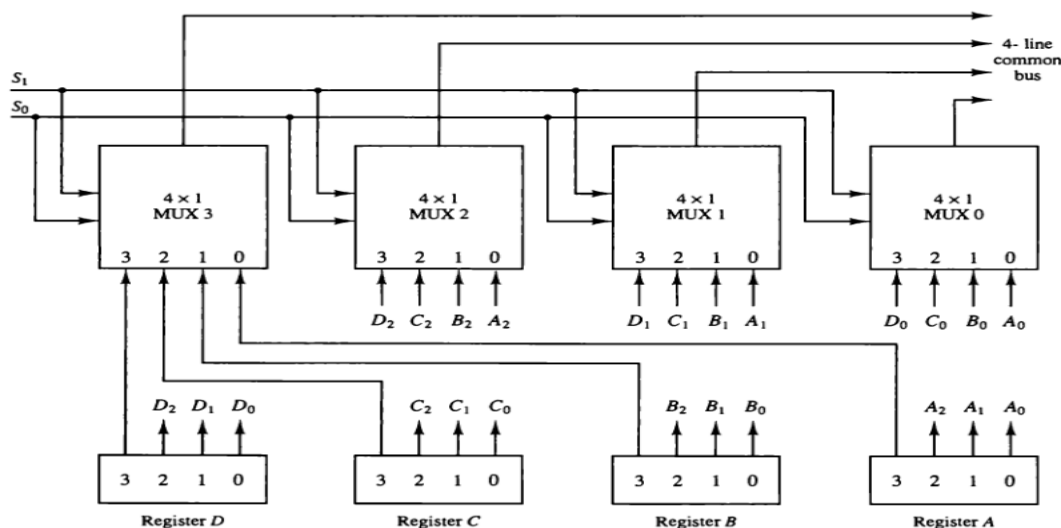


Figure 4.3 bus systems for four register

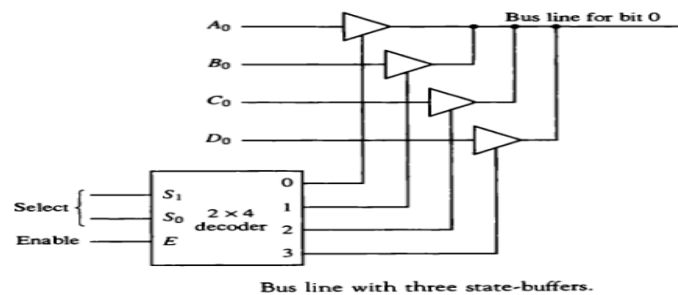


Figure 4.4 bus lines with three –state buffers

4.3 Arithmetic Micro operations

There are four categories of the most common micro-operations:

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

The basic arithmetic micro-operations are addition, subtraction, increment, decrement, and shift.

Example of addition: $R3 \leftarrow R1 + R2$. Subtraction is most often implemented through complementation and addition.

Example of subtraction: $R3 \leftarrow R1 + \overline{R2} + 1$ (strikethrough denotes bar on top – 1's complement of $R2$).

Adding 1 to the 1's complement produces the 2's complement. Adding the contents of $R1$ to the 2's complement of $R2$ is equivalent to subtracting, multiply and divide are not included as micro operations

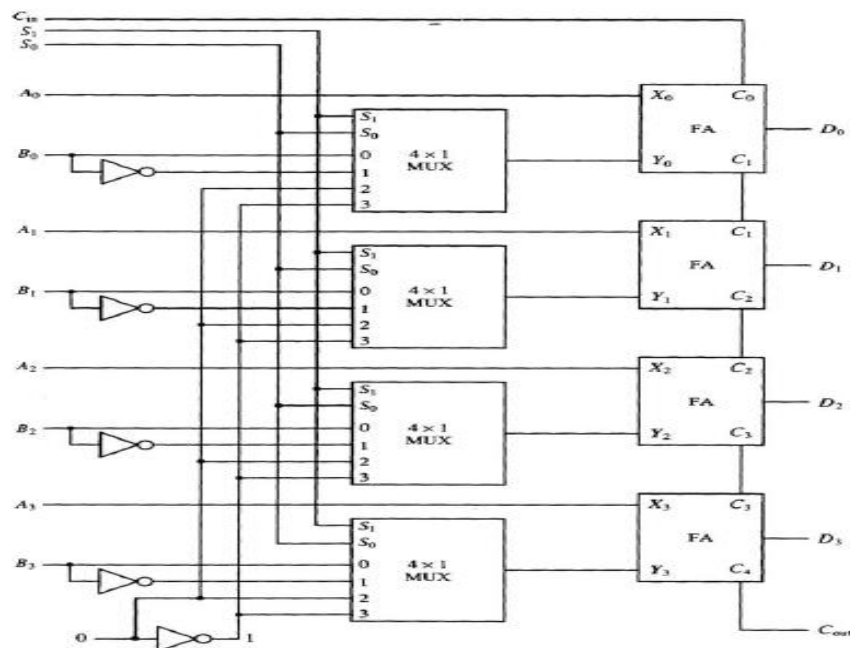


Figure 4.5 four bit arithmetic circuits

Select			Input Y	Output $D = A + Y + C_{in}$	Microoperation
S_1	S_0	C_{in}			
0	0	0	B	$D = A + B$	Add
0	0	1	B	$D = A + B + 1$	Add with carry
0	1	0	\overline{B}	$D = A + \overline{B}$	Subtract with borrow
0	1	1	\overline{B}	$D = A + \overline{B} + 1$	Subtract
1	0	0	0	$D = A$	Transfer A
1	0	1	0	$D = A + 1$	Increment A
1	1	0	1	$D = A - 1$	Decrement A
1	1	1	1	$D = A$	Transfer A

Table 4.2 Arithmetic Circuit Function Table

4.4 Logic Micro operations

Logic operations specify binary operations for strings of bits stored in registers and treat each bit separately. Example: the XOR of R1 and R2 is symbolized by

$$P: R1 \leftarrow R1 \oplus R2$$

Example: $R1 = 1010$ and $R2 = 1100$

1010 Content of R1

1100 Content of R2

0110 Content of R1 after $P = 1$

Symbols used for logical micro-operations:

\Rightarrow OR: \vee

\Rightarrow AND: \wedge

\Rightarrow XOR: \oplus

\Rightarrow The $+$ sign has two different meanings: logical OR and summation. When $+$ is in a micro-operation, then summation.

\Rightarrow When $+$ is in a control function, then OR

Example:

$$P + Q: R1 \leftarrow R2 + R3, R4 \leftarrow R5 \vee R6$$

There are 16 different logic operations that can be performed with two binary variables

x	y	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Table 4.3 Truth table for 16 functions of two variables

Boolean function	Microoperation	Name
$F_0 = 0$	$F \leftarrow 0$	Clear
$F_1 = xy$	$F \leftarrow A \wedge B$	AND
$F_2 = xy'$	$F \leftarrow A \wedge \overline{B}$	
$F_3 = x$	$F \leftarrow A$	Transfer A
$F_4 = x'y$	$F \leftarrow \overline{A} \wedge B$	
$F_5 = y$	$F \leftarrow B$	Transfer B
$F_6 = x \oplus y$	$F \leftarrow A \oplus B$	Exclusive-OR
$F_7 = x + y$	$F \leftarrow A \vee B$	OR
$F_8 = (x + y)'$	$F \leftarrow \overline{A \vee B}$	NOR
$F_9 = (x \oplus y)'$	$F \leftarrow \overline{A \oplus B}$	Exclusive-NOR
$F_{10} = y'$	$F \leftarrow \overline{B}$	Complement B
$F_{11} = x + y'$	$F \leftarrow A \vee \overline{B}$	
$F_{12} = x'$	$F \leftarrow \overline{A}$	Complement A
$F_{13} = x' + y$	$F \leftarrow \overline{A} \vee B$	
$F_{14} = (xy)'$	$F \leftarrow \overline{A \wedge B}$	NAND
$F_{15} = 1$	$F \leftarrow \text{all 1's}$	Set to all 1's

Table 4.4 sixteen micro operations

The hardware implementation of logic micro-operations requires that logic gates be inserted for each bit or pair of bits in the registers. All 16 micro-operations can be derived from using four logic gates.

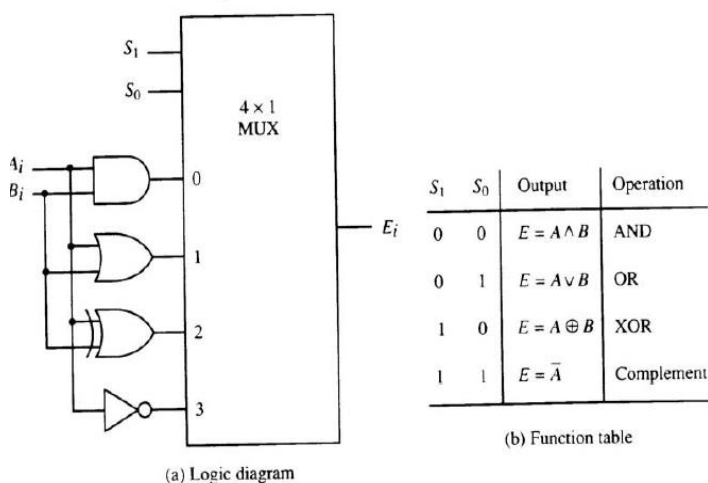


Table 4.6 one stage of logic circuit

4.5 Shift Micro operation:

Shift micro operations are used for serial transfer of data. They are also used in conjunction with arithmetic, logic, and other data-processing operations. There are three types of shifts: logical, circular, and arithmetic.

Symbolic designation	Description
$R \leftarrow \text{shl } R$	Shift-left register R
$R \leftarrow \text{shr } R$	Shift-right register R
$R \leftarrow \text{cil } R$	Circular shift-left register R
$R \leftarrow \text{cir } R$	Circular shift-right register R
$R \leftarrow \text{ashl } R$	Arithmetic shift-left R
$R \leftarrow \text{ashr } R$	Arithmetic shift-right R

Table 4.5 shift micro-operations

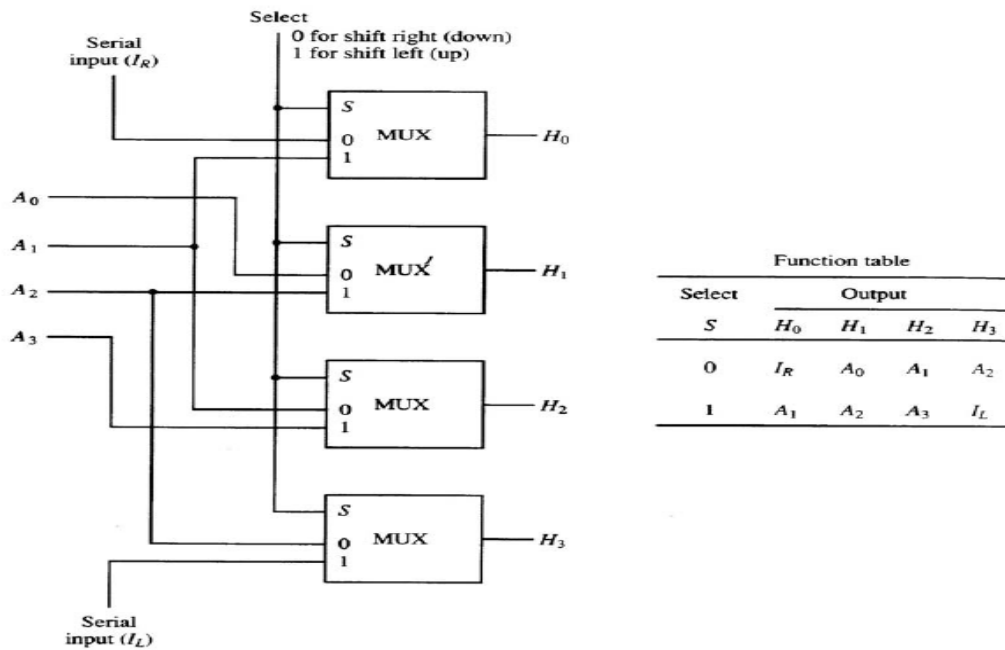


Figure 4.7 Four-bit combinational circuit shifter

4.6 Arithmetic Logic Shift Unit

The *arithmetic logic unit (ALU)* is a common operational unit connected to a number of storage registers. To perform a micro operation; the contents of specified registers are placed in the inputs of the ALU. The ALU performs an operation and the result is then transferred to a destination register.

The ALU is a combinational circuit so that the entire register transfer operation from the source registers through the ALU and into the destination register can be performed during one clock pulse period.

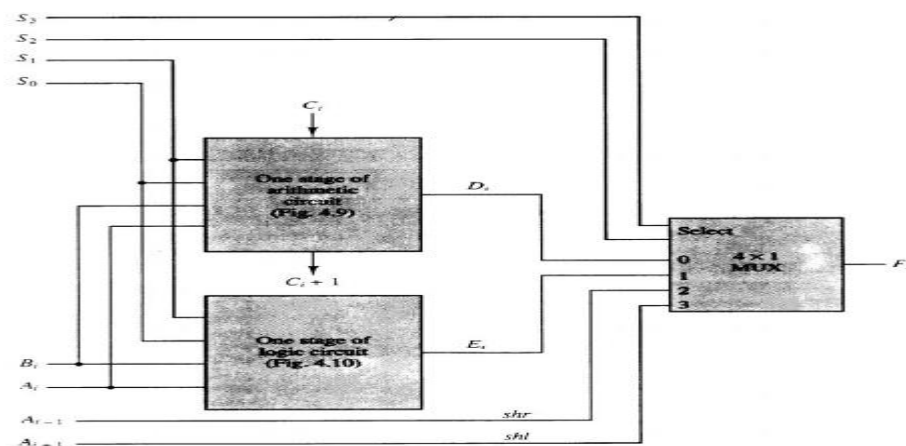


Figure 4.8 one stage of arithmetic logic shift unit

Operation select					Operation	Function
S_3	S_2	S_1	S_0	C_{in}		
0	0	0	0	0	$F = A$	Transfer A
0	0	0	0	1	$F = A + 1$	Increment A
0	0	0	1	0	$F = A + B$	Addition
0	0	0	1	1	$F = A + B + 1$	Add with carry
0	0	1	0	0	$F = A + \overline{B}$	Subtract with borrow
0	0	1	0	1	$F = A + \overline{B} + 1$	Subtraction
0	0	1	1	0	$F = A - 1$	Decrement A
0	0	1	1	1	$F = A$	Transfer A
0	1	0	0	\times	$F = A \wedge B$	AND
0	1	0	1	\times	$F = A \vee B$	OR
0	1	1	0	\times	$F = A \oplus B$	XOR
0	1	1	1	\times	$F = \overline{A}$	Complement A
1	0	\times	\times	\times	$F = \text{shr } A$	Shift right A into F
1	1	\times	\times	\times	$F = \text{shl } A$	Shift left A into F

Table 4.6 function table for arithmetic logic shift unit

Summary

A micro-operation is an elementary operation performed on the information stored in one or more registers. The internal hardware organization of a digital computer is best defined by specifying

- ⇒ The set of registers it contains and their functions
- ⇒ The sequence of micro operations performed on the binary information stored
- ⇒ The control that initiates the sequence of micro operations

The register that holds an address for the memory unit is called MAR. Decoders are used to ensure that no more than one control input is active at any given time. There are four categories of the most common micro operations:

- ⇒ *Register transfer*: transfer binary information from one register to another
- ⇒ *Arithmetic*: perform arithmetic operations on numeric data stored in registers
- ⇒ *Logic*: perform bit manipulation operations on non-numeric data stored in registers
- ⇒ *Shift*: perform shift operations on data stored in registers

Review Questions

1. List and discuss types of micro operation
2. Discuss in detail about register transfer

References

1. Shankersinh VaghelaBapu, **Assistant Professor (I.T.)** Institute of Technology, Gandhinagar
2. Stallings, W. *Computer Architecture and Organizations Designing for Performance, Eighth Edition*. Upper Saddle River, NJ: Prentice Hall, 2010.

CHAPTER FIVE: BASIC COMPUTER ORGANIZATION AND DESIGN

Objective

Upon the completion of this chapter students will be able to

- ⇒ Define what is instruction code
- ⇒ Identify set of instruction to be the instruction is completeness
- ⇒ Describe timing and control
- ⇒ Identify basic unit of control unit
- ⇒ Describe computer registers

Key concepts

- A instruction is a binary code that specifies a sequence of micro operations
- The computer reads each instruction from memory and **places it in a control register**.
- The control then **interprets the binary code** of the instruction and proceeds to **execute it** by issuing a sequence of micro-operations.
- A master clock generator controls the timing for all registers in the basic computer.
- The control logic is implemented with gates, F/Fs, decoders, and other digital circuit.

Introduction

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, operand, and the sequence (*control*) and instruction is a binary code that specifies a sequence of micro-operations. Instruction codes together with data are stored in memory. Instructions are encoded as binary *instruction codes*. Each instruction code contains of a *operation code*, or *opcode*, which designates the overall purpose of the instruction (e.g. add, subtract, move, input, etc.).

5.1 Instruction Code

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 code operation part.

Stored Program Organization

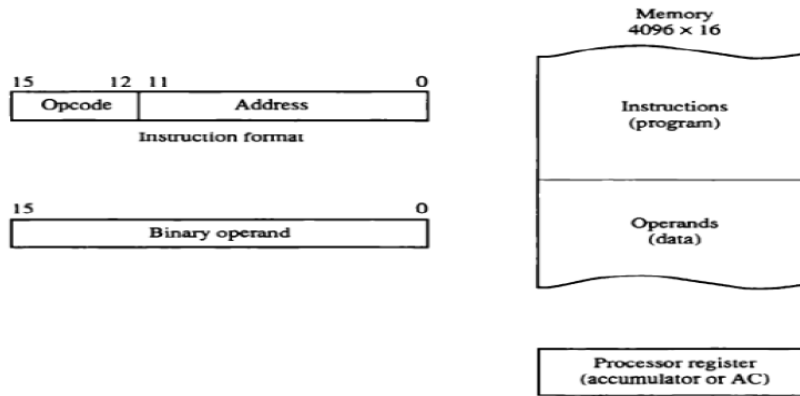


Figure 5.1 stored program organization

5.2 Computer Registers

Computer instructions are normally stored in consecutive memory locations and are executed sequentially one at a time.

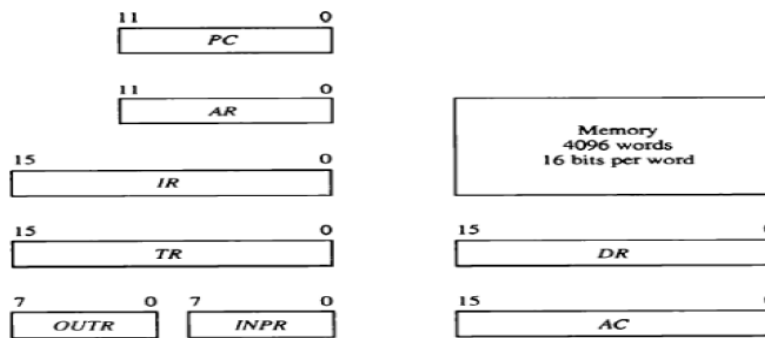


Figure 5.2 basic computer registers and memory

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

Table 5.1 list of registers for basic computer

5.3 Computer Instructions

The operation code (opcode) 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 J.

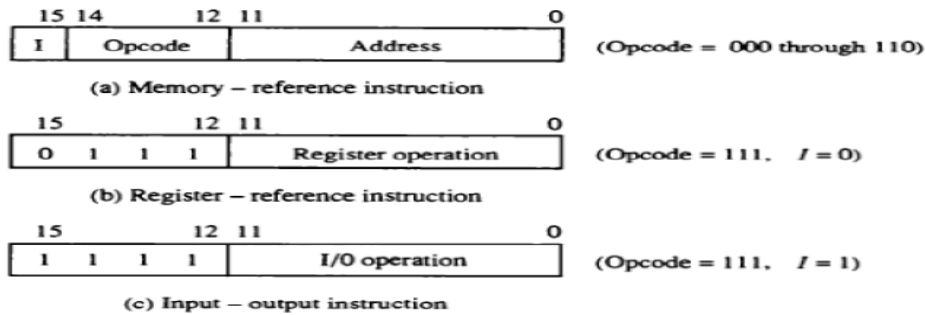


Figure 5.3 basic computer instruction formats

Symbol	Hexadecimal code		Description
	<i>I</i> = 0	<i>I</i> = 1	
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

Table 5.2 basic computer instruction

Instruction Set Completeness

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 status conditions
4. Input and output instructions

5.4 Timing and Control

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 micro operations for the accumulator.

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 micro programmed control, any required control changes or modifications can be done by updating the micro program in control memory. 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.

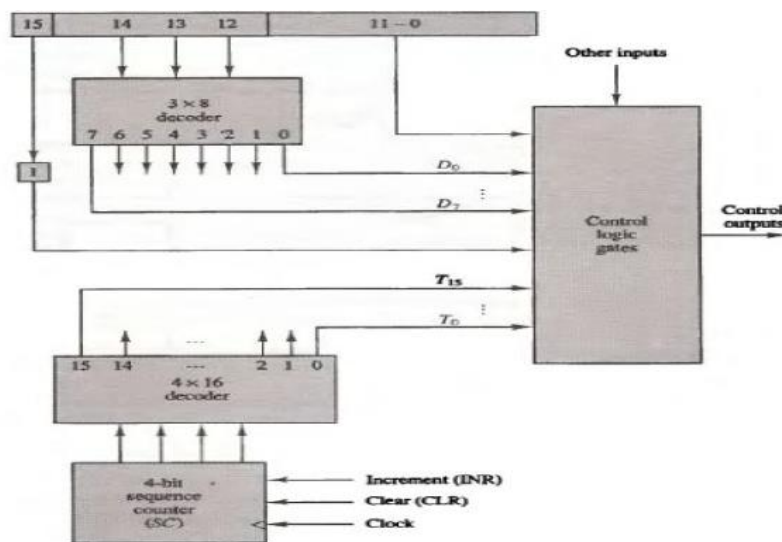


Figure 5.4 control unit of basic computer

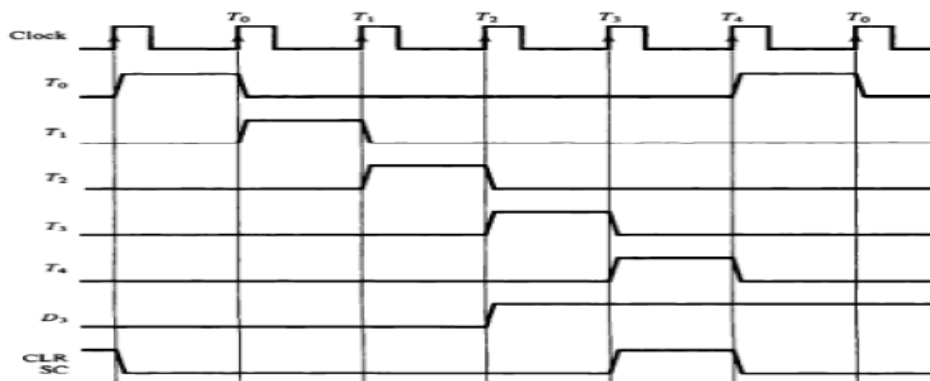


Figure 5.5 example of control timing signal

- Memory reference instruction
- Design of basic computer
- Design of accumulator logic

Summary

Instruction Codes: is the user of a computer can control the process by means of a program and also is a group of bits that instruct the computer to perform a specific operation. A program is a set of instructions that specify the operations, operand, and the sequence (control) instruction is a binary code that specifies a sequence of micro operations. 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 instruction and proceeds to execute it by issuing a sequence of micro operations.

Review questions & Problems

1. Draw the diagram for basic components of control unit
2. List and discuss set of instruction in the computer system
3. Define what is timing and control
4. What is computer register?

References

1. *Computer organization and design, hardware / software interface* David a. Patterson, revised fourth edition
2. *Computer organization and design, hardware / software interface* john l. Hennessy, revised fourth edition
3. *Stallings, W. Computer Architecture and Organizations Designing for Performance, Eighth Edition. Upper Saddle River, NJ: Prentice Hall, 2010.*

CHAPTER SIX: CENTRAL PROCESSING UNIT

Objectives

After the completion of this chapter students will be able to:

- ⇒ Understand the parts of the central processing unit (CPU)
- ⇒ Identify the registers and control units
- ⇒ know how instruction formats are defined
- ⇒ know the design of processor architectures

Key Concepts

- A processor includes both user-visible registers and control/status registers.
- An operand reference in an instruction either contains the actual value of the operand (immediate) or a reference to the address of the operand
- The instruction format defines the layout fields in the instruction.
- Studies of the execution behavior of high-level language programs have provided guidance in designing a new type of processor architecture: the reduced instruction set computer (RISC).

Introduction

This chapter examines the internal structure and function of the processor. The processor consists of registers, the arithmetic and logic unit, the instruction execution unit, a control unit, and the interconnections among these components.

To understand the organization of the processor, let us consider the requirements placed on the processor, the things that it must do:

- **Fetch instruction:** The processor reads an instruction from memory (register, cache, main memory).
- **Interpret instruction:** The instruction is decoded to determine what action is required.
- **Fetch data:** The execution of an instruction may require reading data from memory or an I/O module.
- **Process data:** The execution of an instruction may require performing some arithmetic or logical operation on data.

- **Write data:** The results of an execution may require writing data to memory or an I/O module.

To do these things, it should be clear that the processor needs to store some data temporarily. It must remember the location of the last instruction so that it can know where to get the next instruction. It needs to store instructions and data temporarily while an instruction is being executed. In other words, the processor needs a small internal memory.

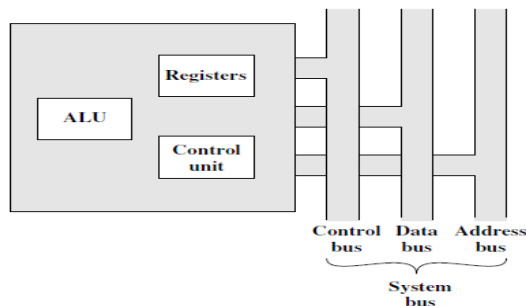


Figure 6.1 CPU with the System Bus

Figure 6.1 is a simplified view of a processor, indicating its connection to the rest of the system via the system bus. It should be understood that the major components of the processor are an *arithmetic and logic unit* (ALU) and a *control unit* (CU). The ALU does the actual computation or processing of data. The control unit controls the movement of data and instructions into and out of the processor and controls the operation of the ALU. In addition, the figure shows a minimal internal memory, consisting of a set of storage locations, called *registers*.

6.1 Register Organizations

A computer system employs a memory hierarchy. At higher levels of the hierarchy, memory is faster, smaller, and more expensive (per bit). Within the processor, there is a set of registers that function as a level of memory above main memory and cache in the hierarchy. The registers in the processor perform two roles:

User-visible registers: Enable the machine- or assembly language programmer to minimize main memory references by optimizing use of registers.

Control and status registers: Used by the control unit to control the operation of the processor and by privileged, operating system programs to control the execution of programs.

There is not a clean separation of registers into these two categories. For example, on some machines the program counter is user visible (e.g., x86), but on many it is not. For purposes of the following discussion, however, we will use these categories.

A user-visible register is one that may be referenced by means of the machine language that the processor executes. We can characterize these in the following categories:

General-purpose registers can be assigned to a variety of functions by the programmer. Sometimes their use within the instruction set is orthogonal to the operation. That is, any general-purpose register can contain the operand for any opcode. This provides true general-purpose register use. Often, however, there are restrictions. For example, there may be dedicated registers for floating-point and stack operations. In some cases, general-purpose registers can be used for addressing functions (e.g., register indirect, displacement). In other cases, there is a partial or clean separation between data registers and address registers.

Data registers may be used only to hold data and cannot be employed in the calculation of an operand address.

Address registers may themselves be somewhat general purpose, or they may be devoted to a particular addressing mode. Examples include the following:

- **Segment pointers:** In a machine with segmented addressing, a segment register holds the address of the base of the segment. There may be multiple registers: for example, one for the operating system and one for the current process.
- **Index registers:** These are used for indexed addressing and may be auto indexed.
- **Stack pointer:** If there is user-visible stack addressing, then typically there is a dedicated register that points to the top of the stack. This allows implicit addressing; that is, push, pop, and other stack instructions need not contain an explicit stack operand.

Another design issue is the number of registers, general purpose or data plus address, to be provided. Again, this affects instruction set design because more registers require more operand specifier bits.

Finally, there is the issue of register length. Registers that must hold addresses obviously must be at least long enough to hold the largest address. Data registers should be able to hold values of most data types. Some machines allow two contiguous registers to be used as one for holding double-length values.

A final category of registers, which is at least partially visible to the user, holds **condition codes** (also referred to as *flags*). Condition codes are bits set by the processor hardware as the result of operations. For example, an arithmetic operation may produce a positive, negative, zero, or overflow result. In addition to the result itself being stored in a register or memory, a condition code is also set. The code may subsequently be tested as part of a conditional branch operation.

Condition code bits are collected into one or more registers. Usually, they form part of a control register. Generally, machine instructions allow these bits to be read by implicit reference, but the programmer cannot alter them. Many processors, including those based on the IA-64 architecture and the MIPS processors do not use condition codes at all. Rather, conditional branch instructions specify a comparison to be made and act on the result of the comparison, without storing a condition code. Table 6.1 lists key advantages and disadvantages of condition codes.

Table 6.1 Condition Codes

Advantages	Disadvantages
<ol style="list-style-type: none"> 1. Because condition codes are set by normal arithmetic and data movement instructions, they should reduce the number of COMPARE and TEST instructions needed. 2. Conditional instructions, such as BRANCH are simplified relative to composite instructions, such as TEST AND BRANCH. 3. Condition codes facilitate multi way branches. For example, a TEST instruction can be followed by two branches, one on less than or equal to zero and one on greater than zero. 	<ol style="list-style-type: none"> 1. Condition codes add complexity, both to the hardware and software. Condition code bits are often modified in different ways by different instructions, making life more difficult for both the micro-programmer and compiler writer. 2. Condition codes are irregular; they are typically not part of the main data path, so they require extra hardware connections. 3. Often condition code machines must add special non-condition-code instructions for special situations anyway, such as bit checking, loop control, and atomic semaphore operations. 4. In a pipelined implementation, condition codes require special synchronization to avoid conflicts.

There are a variety of processor registers that are employed to control the operation of the processor. Most of these, on most machines, are not visible to the user. Some of them may be visible to machine instructions executed in a control or operating system mode.

Of course, different machines will have different register organizations and use different terminology. We list here a reasonably complete list of register types, with a brief description.

Four registers are essential to instruction execution:

- **Program counter (PC):** Contains the address of an instruction to be fetched
- **Instruction register(IR):** Contains the instruction most recently fetched
- **Memory address register (MAR):** Contains the address of a location in memory
- **Memory buffer register (MBR):** Contains a word of data to be written to memory or the word most recently read.

Not all processors have internal registers designated as MAR and MBR, but some equivalent buffering mechanism is needed whereby the bits to be transferred to the system bus are staged and the bits to be read from the data bus are temporarily stored.

Typically, the processor updates the PC after each instruction fetch so that the PC always points to the next instruction to be executed. A branch or skip instruction will also modify the contents of the PC. The fetched instruction is loaded into an IR, where the opcode and operand specifiers are analyzed. Data are exchanged with memory using the MAR and MBR. In a bus-organized system, the MAR connects directly to the address bus, and the MBR connects directly to the data bus. User-visible registers, in turn, exchange data with the MBR.

The four registers just mentioned are used for the movement of data between the processor and memory. Within the processor, data must be presented to the ALU for processing. The ALU may have direct access to the MBR and user-visible registers. Alternatively, there may be additional buffering registers at the boundary to the ALU; these registers serve as input and output registers for the ALU and exchange data with the MBR and user-visible registers.

Many processor designs include a register or set of registers, often known as the *program status word* (PSW), that contain status information. The PSW typically contains condition codes plus other status information. Common fields or flags include the following:

- **Sign:** Contains the sign bit of the result of the last arithmetic operation.
- **Zero:** Set when the result is 0.
- **Carry:** Set if an operation resulted in a carry (addition) into or borrow (subtraction) out of a high-order bit. Used for multiword arithmetic operations.
- **Equal:** Set if a logical compare result is equality.

- **Overflow:** Used to indicate arithmetic overflow.
- **Interrupt Enable/Disable:** Used to enable or disable interrupts.
- **Supervisor:** Indicates whether the processor is executing in supervisor or user mode.

Example: Microprocessor Register Organizations- It is instructive to examine and compare the register organization of comparable systems. In this section, we look at two 16-bit microprocessors that were designed at about the same time: the Motorola MC68000 and the Intel 8086. Figures 6.2a and b depict the register organization of each; purely internal registers, such as a memory address register, are not shown.

The MC68000 partitions its 32-bit registers into eight data registers and nine address registers. The eight data registers are used primarily for data manipulation and are also used in addressing as index registers. The width of the registers allows 8-, 16-, and 32-bit data operations, determined by opcode. The address registers contain 32-bit (no segmentation) addresses; two of these registers are also used as stack pointers, one for users and one for the operating system, depending on the current execution mode. Both registers are numbered 7, because only one can be used at a time. The MC68000 also includes a 32-bit program counter and a 16-bit status register.

The Motorola team wanted a very regular instruction set, with no special purpose registers. A concern for code efficiency led them to divide the registers into two functional components, saving one bit on each register specifier. This seems a reasonable compromise between complete generality and code compaction.

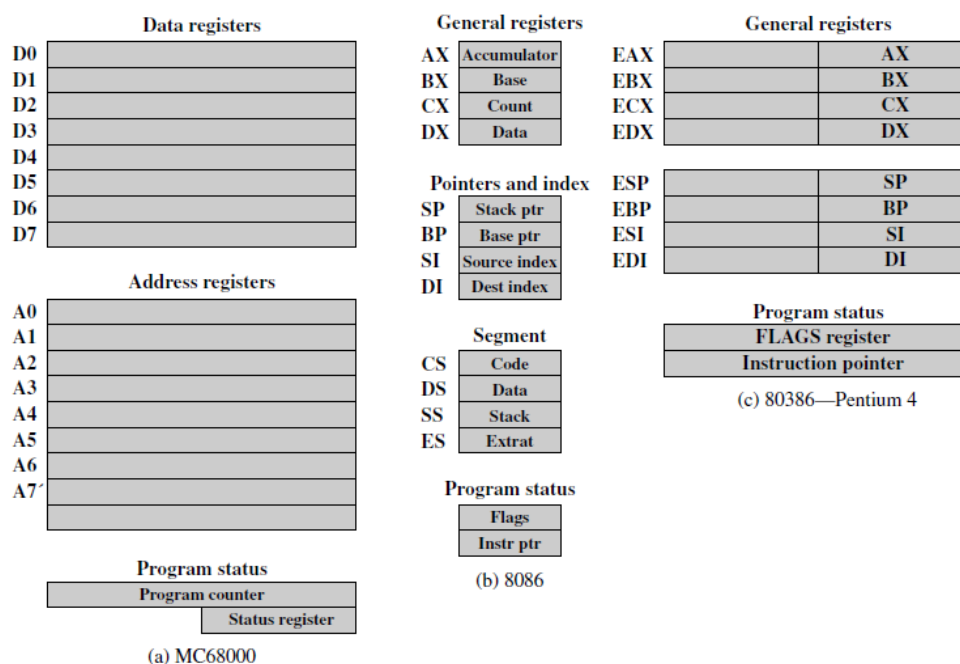


Figure 6.2 Example Microprocessor Register Organizations

The Intel 8086 takes a different approach to register organization. Every register is special purpose, although some registers are also usable as general purpose. The 8086 contains four 16-bit data registers that are addressable on a byte or 16-bit basis, and four 16-bit pointer and index registers. The data registers can be used as general purpose in some instructions. In others, the registers are used implicitly. For example, a multiply instruction always uses the accumulator. The four pointer registers are also used implicitly in a number of operations; each contains a segment offset.

There are also four 16-bit segment registers. Three of the four segment registers are used in a dedicated, implicit fashion, to point to the segment of the current instruction (useful for branch instructions), a segment containing data, and a segment containing a stack, respectively. These dedicated and implicit uses provide for compact encoding at the cost of reduced flexibility. The 8086 also includes an instruction pointer and a set of 1-bit status and control flags.

6.2 Instruction Sets: Addressing Modes and Formats

6.2.1 Addressing Modes

The address field or fields in a typical instruction format are relatively small. We would like to be able to reference a large range of locations in main memory or, for some systems, virtual memory. To achieve this objective, a variety of addressing techniques has been employed. They all involve some trade-off between address range and/or addressing flexibility, on the one hand, and the number of memory references in the instruction and/or the complexity of address calculation, on the other.

In this section, we examine the most common addressing techniques: Immediate, Direct, Indirect, Register, Register indirect, Displacement, Stack

These modes are illustrated in Figure 6.3. In this section, we use the following notation:

(X) = contents of memory location X or register X

EA = actual (effective) address of the location containing the referenced operand

R = contents of an address field in the instruction that refers to a register

A = contents of an address field in the instruction

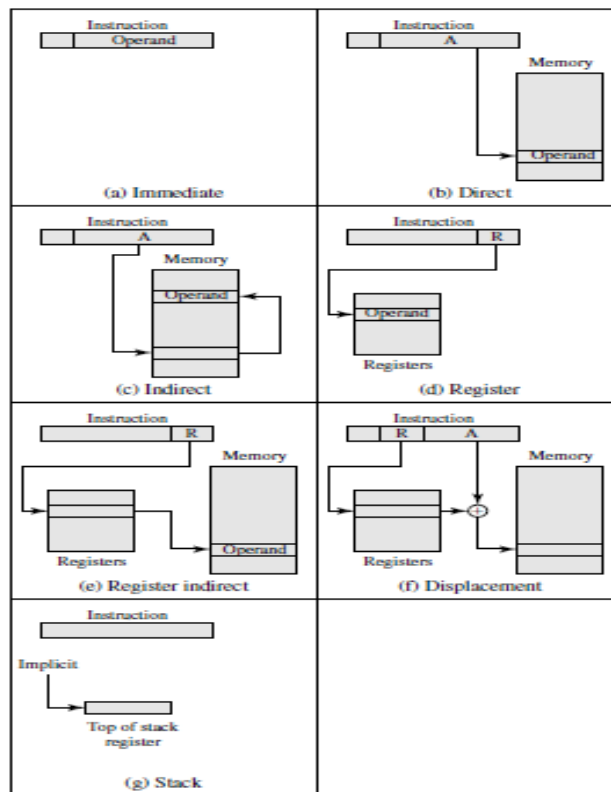


Figure 6.3 Addressing Modes

Table 6.2 Basic Addressing Modes

Mode	Algorithm	Principal Advantage	Principal Disadvantage
Immediate	$\text{Operand} = A$	No memory reference	Limited operand magnitude
Direct	$\text{EA} = A$	Simple	Limited address space
Indirect	$\text{EA} = (A)$	Large address space	Multiple memory references
Register	$\text{EA} = R$	No memory reference	Limited address space
Register indirect	$\text{EA} = (R)$	Large address space	Extra memory reference
Displacement	$\text{EA} = A + (R)$	Flexibility	Complexity
Stack	$\text{EA} = \text{top of stack}$	No memory reference	Limited applicability

Table 6.2 indicates the address calculation performed for each addressing mode. Before beginning this discussion, two comments need to be made. First, virtually all computer architectures provide more than one of these addressing modes. The question arises as to how the processor can determine which address mode is being used in a particular instruction. Several approaches are taken. Often, different opcodes will use different addressing modes. Also, one or more bits in the instruction format can be used as a *mode field*. The value of the mode field determines which addressing mode is to be used.

The second comment concerns the interpretation of the effective address (EA). In a system without virtual memory, the *effective address* will be either a main memory address or a register. In a virtual memory system, the effective address is a virtual address or a register. The

actual mapping to a physical address is a function of the memory management unit (MMU) and is invisible to the programmer.

Immediate Addressing: The simplest form of addressing is immediate addressing, in which the operand value is present in the instruction

$$\text{Operand} = A$$

This mode can be used to define and use constants or set initial values of variables. Typically, the number will be stored in two's complement form; the leftmost bit of the operand field is used as a sign bit. When the operand is loaded into a data register, the sign bit is extended to the left to the full data word size. In some cases, the immediate binary value is interpreted as an unsigned nonnegative integer.

The advantage of immediate addressing is that no memory reference other than the instruction fetch is required to obtain the operand, thus saving one memory or cache cycle in the instruction cycle. The disadvantage is that the size of the number is restricted to the size of the address field, which, in most instruction sets, is small compared with the word length.

Direct Addressing: A very simple form of addressing is direct addressing, in which the address field contains the effective address of the operand:

$$EA = A$$

The technique was common in earlier generations of computers but is not common on contemporary architectures. It requires only one memory reference and no special calculation. The obvious limitation is that it provides only a limited address space.

Indirect Addressing: With direct addressing, the length of the address field is usually less than the word length, thus limiting the address range. One solution is to have the address field refer to the address of a word in memory, which in turn contains a full-length address of the operand. This is known as *indirect addressing*:

$$EA = (A)$$

As defined earlier, the parentheses are to be interpreted as meaning *contents of*. The obvious advantage of this approach is that for a word length of N , an address space of 2^N is now available. The disadvantage is that instruction execution requires two memory references to fetch the operand: one to get its address and a second to get its value.

A rarely used variant of indirect addressing is multilevel or cascaded indirect addressing:

$$EA = (\text{ } \acute{A} (A) \acute{A})$$

In this case, one bit of a full-word address is an indirect flag (I). If the I bit is 0, then the word contains the EA. If the I bit is 1, then another level of indirection is invoked. There does not

appear to be any particular advantage to this approach, and its disadvantage is that three or more memory references could be required to fetch an operand.

Register Addressing: Register addressing is similar to direct addressing. The only difference is that the address field refers to a register rather than a main memory address:

$$EA = R$$

To clarify, if the contents of a register address field in an instruction is 5, then register R5 is the intended address, and the operand value is contained in R5. Typically, an address field that references registers will have from 3 to 5 bits, so that a total of from 8 to 32 general-purpose registers can be referenced. The advantages of register addressing are that (1) only a small address field is needed in the instruction, and (2) no time-consuming memory references are required.

The disadvantage of register addressing is that the address space is very limited. It is up to the programmer or compiler to decide which values should remain in registers and which should be stored in main memory.

Register Indirect Addressing: Just as register addressing is analogous to direct addressing, register indirect addressing is analogous to indirect addressing. In both cases, the only difference is whether the address field refers to a memory location or a register. Thus, for register indirect address,

$$EA = (R)$$

The advantages and limitations of register indirect addressing are basically the same as for indirect addressing. In both cases, the address space limitation (limited range of addresses) of the address field is overcome by having that field refer to a word-length location containing an address.

Displacement Addressing: A very powerful mode of addressing combines the capabilities of direct addressing and register indirect addressing. It is known by a variety of names depending on the context of its use, but the basic mechanism is the same. We will refer to this as *displacement addressing*:

$$EA = A + (R)$$

Displacement addressing requires that the instruction have two address fields, at least one of which is explicit. The value contained in one address field (value = A) is used directly. The other address field, or an implicit reference based on opcode, refers to a register whose contents are added to A to produce the effective address.

We will describe three of the most common uses of displacement addressing:

1. **RELATIVE ADDRESSING** For relative addressing, also called PC-relative addressing, the implicitly referenced register is the program counter (PC). That is, the next instruction address is added to the address field to produce the EA. Typically, the address field is treated as a two's complement number for this operation. Thus, the effective address is a displacement relative to the address of the instruction. Relative addressing exploits the concept of locality.
2. **BASE-REGISTER ADDRESSING** For base-register addressing, the interpretation is the following: The referenced register contains a main memory address, and the address field contains a displacement (usually an unsigned integer representation) from that address. The register reference may be explicit or implicit.
3. **INDEXING** For indexing, the interpretation is typically the following: The address field references a main memory address, and the referenced register contains a positive displacement from that address. Note that this usage is just the opposite of the interpretation for base-register addressing.

Stack Addressing: The final addressing mode that we consider is stack addressing. A stack is a linear array of locations. It is sometimes referred to as a *pushdown list* or *last-in-first-out queue*. The stack is a reserved block of locations. Items are appended to the top of the stack so that, at any given time, the block is partially filled. Associated with the stack is a pointer whose value is the address of the top of the stack. Alternatively, the top two elements of the stack may be in processor registers, in which case the stack pointer references the third element of the stack. The stack pointer is maintained in a register. Thus, references to stack locations in memory are in fact register indirect addresses.

The stack mode of addressing is a form of implied addressing. The machine instructions need not include a memory reference but implicitly operate on the top of the stack.

6.2.2 Instruction Formats

An instruction format defines the layout of the bits of an instruction, in terms of its constituent fields. An instruction format must include an opcode and, implicitly or explicitly, zero or more operands. Each explicit operand is referenced using one of the addressing modes described in Section 6.2.1. The format must, implicitly or explicitly, indicate the addressing mode for each operand. For most instruction sets, more than one instruction format is used.

The design of an instruction format is a complex art, and an amazing variety of designs have been implemented. We examine the key design issues, looking briefly at some designs to illustrate points, and then we examine the x86 and ARM solutions in detail.

Instruction Length: The most basic design issue to be faced is the instruction format length. This decision affects, and is affected by, memory size, memory organization, bus structure, processor complexity, and processor speed. This decision determines the richness and flexibility of the machine as seen by the assembly-language programmer.

The most obvious trade-off here is between the desire for a powerful instruction repertoire and a need to save space. Programmers want more opcodes, more operands, more addressing modes, and greater address range. More opcodes and more operands make life easier for the programmer, because shorter programs can be written to accomplish given tasks. Similarly, more addressing modes give the programmer greater flexibility in implementing certain functions, such as table manipulations and multiple-way branch. And, of course, with the increase in main memory size and the increasing use of virtual memory, programmers want to be able to address larger memory ranges. All of these things (opcodes, operands, addressing modes, address range) require bits and push in the direction of longer instruction lengths. But longer instruction length may be wasteful. A 64-bit instruction occupies twice the space of a 32-bit instruction but is probably less than twice as useful.

Allocation of Bits: We've looked at some of the factors that go into deciding the length of the instruction format. An equally difficult issue is how to allocate the bits in that format. The trade-offs here are complex.

For a given instruction length, there is clearly a trade-off between the number of opcodes and the power of the addressing capability. More opcodes obviously mean more bits in the opcode field. For an instruction format of a given length, this reduces the number of bits available for addressing. There is one interesting refinement to this trade-off, and that is the use of variable-length opcodes. In this approach, there is a minimum opcode length but, for some opcodes, additional operations may be specified by using additional bits in the instruction. For a fixed-length instruction, this leaves fewer bits for addressing. The following interrelated factors go into determining the use of the addressing bits.

- Number of addressing modes
- Number of operands
- Register versus memory
- Number of register sets
- Address range
- Address granularity

Let us briefly look at how two historical machine designs balance these various factors.

PDP-8 One of the simplest instruction designs for a general-purpose computer was for the PDP-8. The PDP-8 uses 12-bit instructions and operates on 12-bit words. There is a single general-purpose register, the accumulator. Despite the limitations of this design, the addressing is quite flexible. Each memory reference consists of 7 bits plus two 1-bit modifiers. The memory is divided into fixed-length pages of $2^7 = 128$ words each. Address calculation is based on references to page 0 or the current page (page containing this instruction) as determined by the page bit. The second modifier bit indicates whether direct or indirect addressing is to be used. These two modes can be used in combination, so that an indirect address is a 12-bit address contained in a word of page 0 or the current page. In addition, 8 dedicated words on page 0 are auto-index “registers.” When an indirect reference is made to one of these locations, pre-indexing occurs.

Figure 6.4 shows the PDP-8 instruction format. There are a 3-bit opcode and three types of instructions. For opcodes 0 through 5, the format is a single-address memory reference instruction including a page bit and an indirect bit. Thus, there are only six basic operations. To enlarge the group of operations, opcode 7 defines a register reference or *microinstruction*. In this format, the remaining bits are used to encode additional operations. In general, each bit defines a specific operation (e.g. clear accumulator), and these bits can be combined in a single instruction. The microinstruction strategy was used as far back as the PDP-1 by DEC and is, in a sense, a forerunner of today’s micro-programmed machines, to be discussed in Part Four. Opcode 6 is the I/O operation; 6 bits are used to select one of 64 devices, and 3 bits specify a particular I/O command.

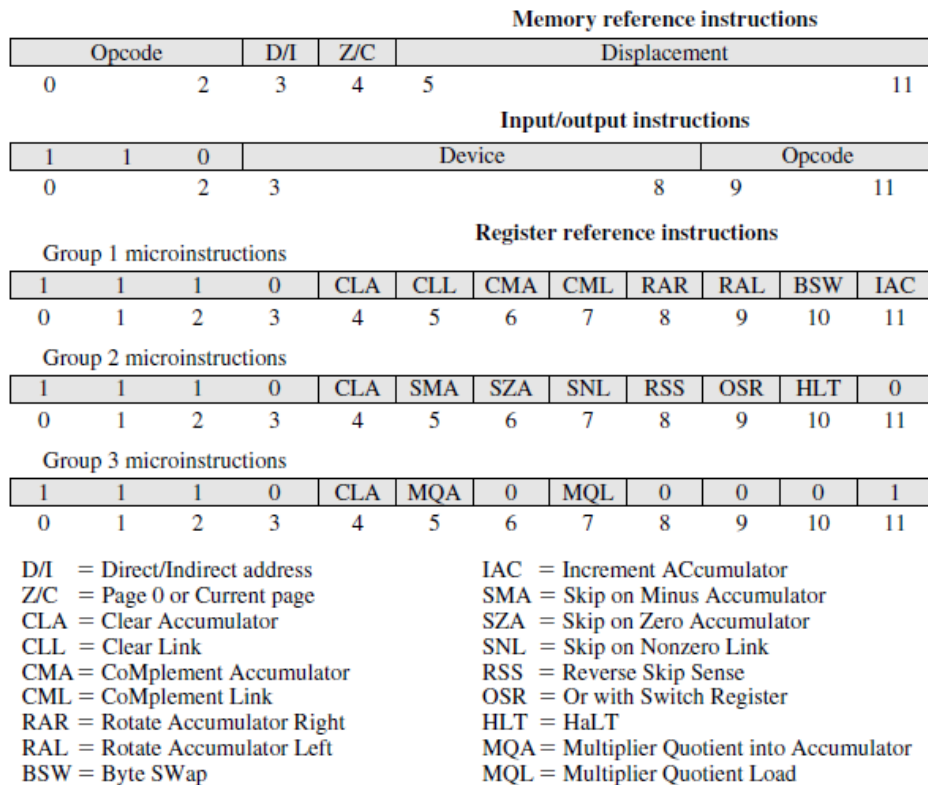


Figure 6.4 PDP-8 Instruction Formats

The PDP-8 instruction format is remarkably efficient. It supports indirect addressing, displacement addressing, and indexing. With the use of the opcode extension, it supports a total of approximately 35 instructions. Given the constraints of a 12-bit instruction length, the designers could hardly have done better.

PDP-10 A sharp contrast to the instruction set of the PDP-8 is that of the PDP-10. The PDP-10 was designed to be a large-scale time-shared system, with an emphasis on making the system easy to program, even if additional hardware expense was involved. Among the design principles employed in designing the instruction set were the following [6]:

Orthogonality: Orthogonality is a principle by which two variables are independent of each other. In the context of an instruction set, the term indicates that other elements of an instruction are independent of (not determined by) the opcode. The PDP-10 designers use the term to describe the fact that an address is always computed in the same way, independent of the opcode.

Completeness: Each arithmetic data type (integer, fixed-point, floating-point) should have a complete and identical set of operations.

Direct addressing: Base plus displacement addressing, which places a memory organization burden on the programmer, was avoided in favor of direct addressing.

Each of these principles advances the main goal of ease of programming.

Variable-Length Instructions: The examples we have looked at so far have used a single fixed instruction length, and we have implicitly discussed trade-offs in that context. But the designer may choose instead to provide a variety of instruction formats of different lengths. This tactic makes it easy to provide a large repertoire of opcodes, with different opcode lengths. Addressing can be more flexible, with various combinations of register and memory references plus addressing modes.

6.3 Characteristics of RISC and CISC

One of the most visible forms of evolution associated with computers is that of programming languages. As the cost of hardware has dropped, the relative cost of software has risen. Along with that, a chronic shortage of programmers has driven up software costs in absolute terms. Thus, the major cost in the life cycle of a system is software, not hardware. Adding to the cost, and to the inconvenience, is the element of unreliability: it is common for programs, both system and application, to continue to exhibit new bugs after years of operation.

The response from researchers and industry has been to develop ever more powerful and complex high-level programming languages. These high-level languages (HLLs) allow the programmer to express algorithms more concisely, take care of much of the detail, and often support naturally the use of structured programming or object-oriented design.

Alas, this solution gave rise to another problem, known as the *semantic gap*, the difference between the operations provided in HLLs and those provided in computer architecture. Symptoms of this gap are alleged to include execution inefficiency, excessive machine program size, and compiler complexity. Designers responded with architectures intended to close this gap. Key features include large instruction sets, dozens of addressing modes, and various HLL statements implemented in hardware. An example of the latter is the CASE machine instruction. Such complex instruction sets are intended to:

- Ease the task of the compiler writer.
- Improve execution efficiency, because complex sequences of operations can be implemented in microcode.
- Provide support for even more complex and sophisticated HLLs.

Table 6.3 Characteristics of Some CISCs, RISCs, and Superscalar Processors

Characteristic	Complex Instruction Set (CISC) Computer			Reduced Instruction Set (RISC) Computer		Superscalar		
	IBM 370/168	VAX 11/780	Intel 80486	SPARC	MIPS R4000	PowerPC	Ultra SPARC	MIPS R10000
Year developed	1973	1978	1989	1987	1991	1993	1996	1996
Number of instructions	208	303	235	69	94	225		
Instruction size (bytes)	2–6	2–57	1–11	4	4	4	4	4
Addressing modes	4	22	11	1	1	2	1	1
Number of general-purpose registers	16	16	8	40–520	32	32	40–520	32
Control memory size (Kbits)	420	480	246	—	—	—	—	—
Cache size (KBytes)	64	64	8	32	128	16–32	32	64

Meanwhile, a number of studies have been done over the years to determine the characteristics and patterns of execution of machine instructions generated from HLL programs. The results of these studies inspired some researchers to look for a different approach: namely, to make the architecture that supports the HLL simpler, rather than more complex.

To understand the line of reasoning of the RISC advocates, we begin with a brief review of instruction execution characteristics. The aspects of computation of interest are as follows:

- **Operations performed:** These determine the functions to be performed by the processor and its interaction with memory.
- **Operands used:** The types of operands and the frequency of their use determine the memory organization for storing them and the addressing modes for accessing them.
- **Execution sequencing:** This determines the control and pipeline organization.

The work that has been done on assessing merits of the RISC approach can be grouped into two categories:

Quantitative: Attempts to compare program size and execution speed of programs on RISC and CISC machines that use comparable technology (see Table 6.3).

Qualitative: Examines issues such as high-level language support and optimum use of VLSI. Most of the work on quantitative assessment has been done by those working on RISC systems, and it has been, by and large, favorable to the RISC approach. Others have examined the issue and come away unconvinced. There are several problems with attempting such comparisons:

- There is no pair of RISC and CISC machines that is comparable in life-cycle cost, level of technology, gate complexity, sophistication of compiler, operating system support, and so on.
- No definitive test set of programs exists. Performance varies with the program.
- It is difficult to sort out hardware effects from effects due to skill in compiler writing.

- Most of the comparative analysis on RISC has been done on “toy” machines rather than commercial products. Furthermore, most commercially available machines advertised as RISC possess a mixture of RISC and CISC characteristics.

Thus, a fair comparison with a commercial, “pure-play” CISC machine (e.g., VAX, Pentium) is difficult.

Summary

A processor includes both user-visible registers and control/status registers. The former may be referenced, implicitly or explicitly, in machine instructions. User-visible registers may be general purpose or have a special use, such as fixed-point or floating-point numbers, addresses, indexes, and segment pointers. Control and status registers are used to control the operation of the processor. One obvious example is the program counter. Another important example is a program status word (PSW) that contains a variety of status and condition bits. These include bits to reflect the result of the most recent arithmetic operation, interrupt enable bits, and an indicator of whether the processor is executing in supervisor or user mode. The instruction format defines the layout fields in the instruction. A wide variety of addressing modes is used in various instruction sets. These include direct (operand address is in address field), indirect (address field points to a location that contains the operand address), register, register indirect, and various forms of displacement, in which a register value is added to an address value to produce the operand address. Instruction format design is a complex undertaking, including such consideration as instruction length, fixed or variable length, number of bits assigned to opcode and each operand reference, and how addressing mode is determined.

Review Questions & Problems

- 1) The floating point format for the PDP-10 processor has an interesting feature: While positive numbers are represented similar to the way most processors represent binary floating point value, negative values are formed by taking the two's complement of the entire 36 bit word. What advantages and disadvantages does this representation have?
- 2) What is the relaxed consistency model in context of register ordering?
- 3) Explain what control hazard is and how to avoid it.
- 4) Briefly give two ways in which loop unrolling can increase performance and one in which it can decrease performance.

- 5) Some RISC architectures require the compiler (or assembly programmer) to guarantee that a register not be accessed for a given number of cycles after it is loaded from memory. Give an advantage and a disadvantage of this design choice.

References

1. Stritter, E., and Gunter, T. "A Microprocessor Architecture for a Changing World: The Motorola 68000." *Computer*, February 1979.
2. Blaauw, G., and Brooks, F. *Computer Architecture: Concepts and Evolution*. Reading, MA: Addison-Wesley, 1997.
3. Patterson, D. "Reduced Instruction Set Computers." *Communications of the ACM*. January 1985.
4. Stallings, W. *Computer Architecture and Organizations Designing for Performance, Eighth Edition*. Upper Saddle River, NJ: Prentice Hall, 2010.

CHAPTER SEVEN: MEMORY ORGANIZATION

Objectives

After the completion of this chapter students will be able to:

- ⇒ Define the computer memory and its organization
- ⇒ Understand the nature of different kinds of memories
- ⇒ Understand the hierarchy of memory
- ⇒ Know the technology of RAID

Key Concepts

- Computer memory is organized into a hierarchy
- As one goes down the memory hierarchy, one finds decreasing cost/bit, increasing capacity, and slower access time
- If the cache is designed properly, then most of the time the processor will request memory words that are already in the cache
- Magnetic disks remain the most important component of external memory
- RAID is a family of techniques for using multiple disks as a parallel array of data storage devices, with redundancy built in to compensate for disk failure

Introduction

Although seemingly simple in concept, computer memory exhibits perhaps the widest range of type, technology, organization, performance, and cost of any feature of a computer system. No one technology is optimal in satisfying the memory requirements for a computer system. As a consequence, the typical computer system is equipped with a hierarchy of memory subsystems, some internal to the system (directly accessible by the processor) and some external (accessible by the processor via an I/O module).

This chapter focuses on internal memory elements, and devoted to external memory. To begin, the first section examines key characteristics of computer memories. The remainder of the chapter examines an essential element of all modern computer systems: cache memory, and main memory, Magnetic disks, RAID Technology, Optical disks and Magnetic Tapes.

7.1 Characteristics of Memory Systems

The complex subject of computer memory is made more manageable if we classify memory systems according to their key characteristics. The most important of these are listed in Table 7.1.

The term **location** in Table 7.1 refers to whether memory is internal and external to the computer. Internal memory is often equated with main memory. But there are other forms of internal memory. The processor requires its own local memory, in the form of registers.

Further, the control unit portion of the processor may also require its own internal memory. Cache is another form of internal memory. External memory consists of peripheral storage devices, such as disk and tape that are accessible to the processor via I/O controllers.

Table 7.1 Key Characteristics of Computer Memory Systems

Location Internal (e.g. processor registers, main memory, cache) External (e.g. optical disks, magnetic disks, tapes)	Performance Access time Cycle time Transfer rate
Capacity Number of words Number of bytes	Physical Type Semiconductor Magnetic Optical Magneto-optical
Unit of Transfer Word Block	Physical Characteristics Volatile/non-volatile Erasable/non-erasable
Access Method Sequential Direct Random Associative	Organization Memory modules

An obvious characteristic of memory is its **capacity**. For internal memory, this is typically expressed in terms of bytes (1 byte 8 bits) or words. Common word lengths are 8, 16, and 32 bits. External memory capacity is typically expressed in terms of bytes.

A related concept is the **unit of transfer**. For internal memory, the unit of transfer is equal to the number of electrical lines into and out of the memory module. This may be equal to the word length, but is often larger, such as 64, 128, or 256 bits. To clarify this point, consider three related concepts for internal memory:

Word: The “natural” unit of organization of memory. The size of the word is typically equal to the number of bits used to represent an integer and to the instruction length. Unfortunately, there are many exceptions. For example, the CRAY C90 (an older model CRAY supercomputer) has a 64-bit word length but uses a 46-bit integer representation. The Intel x86 architecture has a wide variety of instruction lengths, expressed as multiples of bytes, and a word size of 32 bits.

Addressable units: In some systems, the addressable unit is the word. However, many systems allow addressing at the byte level. In any case, the relationship between the length in bits A of an address and the number N of addressable units is $2^A = N$.

Unit of transfer: For main memory, this is the number of bits read out of or written into memory at a time. The unit of transfer need not equal a word or an addressable unit. For external memory, data are often transferred in much larger units than a word, and these are referred to as blocks.

Another distinction among memory types is the **method of accessing** units of data. These include the following:

Sequential access: Memory is organized into units of data, called records. Access must be made in a specific linear sequence. Stored addressing information is used to separate records and assist in the retrieval process. A shared read–write mechanism is used, and this must be moved from its current location to the desired location, passing and rejecting each intermediate record. Thus, the time to access an arbitrary record is highly variable. Tape units are sequential access.

Direct access: As with sequential access, direct access involves a shared read–write mechanism. However, individual blocks or records have a unique address based on physical location. Access is accomplished by direct access to reach general vicinity plus sequential searching, counting, or waiting to reach the final location. Again, access time is variable. Disk units are direct access.

Random access: Each addressable location in memory has a unique, physically wired-in addressing mechanism. The time to access a given location is independent of the sequence of prior accesses and is constant. Thus, any location can be selected at random and directly addressed and accessed. Main memory and some cache systems are random access.

Associative: This is a random access type of memory that enables one to make a comparison of desired bit locations within a word for a specified match, and to do this for all words simultaneously. Thus, a word is retrieved based on a portion of its contents rather than its address. As with ordinary random-access memory, each location has its own addressing mechanism, and retrieval time is constant independent of location or prior access patterns. Cache memories may employ associative access.

From a user's point of view, the two most important characteristics of memory are capacity and **performance**. Three performance parameters are used:

Access time (latency): For random-access memory, this is the time it takes to perform a read or write operation, that is, the time from the instant that an address is presented to the memory to the instant that data have been stored or made available for use. For non-random-access memory, access time is the time it takes to position the read–write mechanism at the desired location.

Memory cycle time: This concept is primarily applied to random-access memory and consists of the access time plus any additional time required before a second access can commence. This additional time may be required for transients to die out on signal lines or to regenerate data if they are read destructively. Note that memory cycle time is concerned with the system bus, not the processor.

Transfer rate: This is the rate at which data can be transferred into or out of a memory unit. For random-access memory, it is equal to $1/(\text{cycle time})$.

For non-random-access memory, the following relationship holds:

$$TN = TA + N/R$$

where,

TN = Average time to read or write N bits

TA = Average access time

N = Number of bits

R = Transfer rate, in bits per second (bps)

A variety of **physical types** of memory have been employed. The most common today are semiconductor memory, magnetic surface memory, used for disk and tape, and optical and magneto-optical.

Several **physical characteristics** of data storage are important. In a volatile memory, information decays naturally or is lost when electrical power is switched off. In a non-volatile memory, information once recorded remains without deterioration until deliberately changed; no electrical power is needed to retain information. Magnetic-surface memories are non-volatile. Semiconductor memory may be either volatile or non-volatile. Non-erasable memory cannot be altered, except by destroying the storage unit. Semiconductor memory of this type is known as *read-only memory* (ROM). Of necessity, a practical non-erasable memory must also be non-volatile.

For random-access memory, the **organization** is a key design issue. By *organization* is meant the physical arrangement of bits to form words.

7.2 The Memory Hierarchy

The design constraints on a computer's memory can be summed up by three questions: How much? How fast? How expensive?

The question of how much is somewhat open ended. If the capacity is there, applications will likely be developed to use it. The question of how fast is, in a sense, easier to answer. To achieve greatest performance, the memory must be able to keep up with the processor. That is, as the processor is executing instructions, we would not want it to have to pause waiting for instructions or operands. The final question must also be considered. For a practical system, the cost of memory must be reasonable in relationship to other components.

As might be expected, there is a trade-off among the three key characteristics of memory: namely, capacity, access time, and cost. A variety of technologies are used to implement memory systems, and across this spectrum of technologies, the following relationships hold:

- Faster access time, greater cost per bit
- Greater capacity, smaller cost per bit
- Greater capacity, slower access time

The dilemma facing the designer is clear. The designer would like to use memory technologies that provide for large-capacity memory, both because the capacity is needed and because the cost per bit is low. However, to meet performance requirements, the designer needs to use expensive, relatively lower-capacity memories with short access times.

The way out of this dilemma is not to rely on a single memory component or technology, but to employ a **memory hierarchy**. A typical hierarchy is illustrated in Figure 7.1. As one goes down the hierarchy, the following occur:

- a. Decreasing cost per bit

- b. Increasing capacity
- c. Increasing access time
- d. Decreasing frequency of access of the memory by the processor

Thus, smaller, more expensive, faster memories are supplemented by larger, cheaper, slower memories. The key to the success of this organization is item (d): decreasing frequency of access. We examine this concept in greater detail when we discuss the cache, later in this chapter.

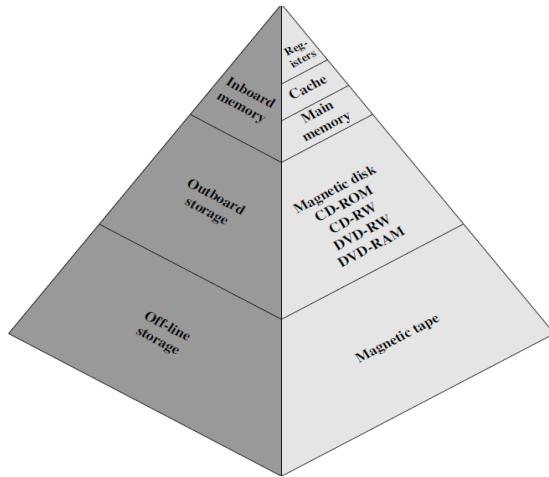


Figure 7.1 The Memory Hierarchy

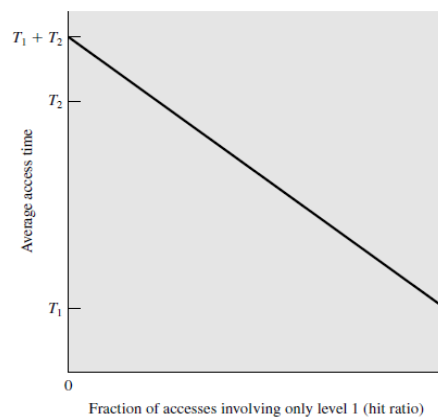


Figure 7.2 Performance of accesses involving only level 1 (hit ratio)

Example 7.1 Suppose that the processor has access to two levels of memory. Level 1 contains 1000 words and has an access time of 0.01 s; level 2 contains 100,000 words and has an access time of 0.1 s. Assume that if a word to be accessed is in level 1, then the processor accesses it directly. If it is in level 2, then the word is first transferred to level 1 and then accessed by the processor. For simplicity, we ignore the time required for the processor to determine whether the word is in level 1 or level 2. Figure 7.2 shows the general shape of the curve that covers this situation. The figure shows the average access time to a two-level memory as a function of the hit ratio H , where H is defined as the fraction of all memory accesses that are found in the faster memory (e.g., the cache), T_1 is the access time to level 1, and T_2 is the access time to level 2. As can be seen, for high percentages of level 1 access, the average total access time is much closer to that of level 1 than that of level 2.

In our example, suppose 95% of the memory accesses are found in the cache. Then the average time to access a word can be expressed as

$$(0.95)(0.01 \mu s) + (0.05)(0.01 \mu s + 0.1 \mu s) = 0.0095 + 0.0055 = 0.015 \mu s$$

The average access time is much closer to 0.01 μs than to 0.1 μs , as desired.

The use of two levels of memory to reduce average access time works in principle, but only if conditions (a) through (d) apply. By employing a variety of technologies, a spectrum of memory systems exists that satisfies conditions (a) through (c). Fortunately, condition (d) is also generally valid.

This principle can be applied across more than two levels of memory, as suggested by the hierarchy shown in Figure 7.1. The fastest, smallest, and most expensive type of memory consists of the registers internal to the processor. Typically, a processor will contain a few dozen such registers, although some machines contain hundreds of registers. Skipping down two levels, main memory is the principal internal memory system of the computer. Each location in main memory has a unique address. Main memory is usually extended with a higher-speed, smaller cache. The cache is not usually visible to the programmer or, indeed, to the processor. It is a device for staging the movement of data between main memory and processor registers to improve performance.

The three forms of memory just described are, typically, volatile and employ semiconductor technology. The use of three levels exploits the fact that semiconductor memory comes in a variety of types, which differ in speed and cost. Data are stored more permanently on external mass storage devices, of which the most common are hard disk and removable media, such as removable magnetic disk, tape, and optical storage. External, non-volatile memory is also referred to as **secondary memory** or **auxiliary memory**. These are used to store program and data files and are usually visible to the programmer only in terms of files and records, as opposed to individual bytes or words. Disk is also used to provide an extension to main memory known as virtual memory.

7.3 Main Memory

In earlier computers, the most common form of random-access storage for computer main memory employed an array of doughnut-shaped ferromagnetic loops referred to as *cores*. Hence, main memory was often referred to as *core*, a term that persists to this day. The advent of, and advantages of, microelectronics has long since vanquished the magnetic core memory. Today, the use of semiconductor chips for main memory is almost universal. Key aspects of this technology are explored in this section.

Organization: The basic element of a semiconductor memory is the memory cell. Although a variety of electronic technologies are used, all semiconductor memory cells share certain properties:

- They exhibit two stable (or semi-stable) states, which can be used to represent binary 1 and 0.
- They are capable of being written into (at least once), to set the state.
- They are capable of being read to sense the state.

Figure 7.3 depicts the operation of a memory cell. Most commonly, the cell has three functional terminals capable of carrying an electrical signal. The select terminal, as the name suggests, selects a memory cell for a read or write operation. The control terminal indicates read or write. For writing, the other terminal provides an electrical signal that sets the state of the cell to 1 or 0. For reading, that terminal is used for output of the cell's state. The details of the internal organization, functioning, and timing of the memory cell depend on the specific integrated circuit technology used and are beyond the scope of this book, except for a brief summary. For our purposes, we will take it as given that individual cells can be selected for reading and writing operations.

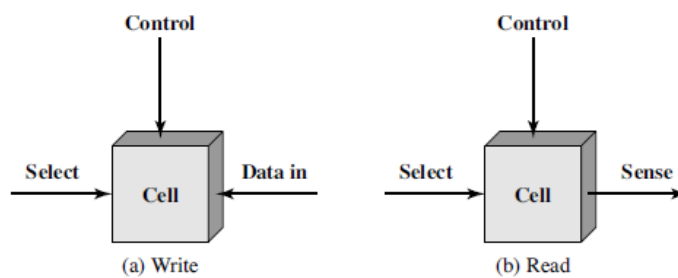


Figure 7.3 Memory Cell Operation

DRAM and SRAM: All of the memory types that we will explore in this section are random access. That is, individual words of memory are directly accessed through wired-in addressing logic. Table 7.2 lists the major types of semiconductor memory. The most common is referred to as *random-access memory* (RAM). This is, of course, a misuse of the term; because all of the types listed in the table are random accesses. One distinguishing characteristic of RAM is that it is possible both to read data from the memory and to write new data into the memory easily and rapidly. Both the reading and writing are accomplished through the use of electrical signals.

Table 7.2 Semiconductor Memory Types

Memory Type	Category	Erasure	Write Mechanism	Volatility
Random-access memory (RAM)	Read-write memory	Electrically, byte-level	Electrically	Volatile
Read-only memory (ROM)	Read-only memory	Not possible	Masks	Nonvolatile
Programmable ROM (PROM)			Electrically	
Erasable PROM (EPROM)	Read-mostly memory	UV light, chip-level		
Electrically Erasable PROM (EEPROM)		Electrically, byte-level		
Flash memory		Electrically, block-level		

The other distinguishing characteristic of RAM is that it is volatile. A RAM must be provided with a constant power supply. If the power is interrupted, then the data are lost. Thus, RAM can be used only as temporary storage. The two traditional forms of RAM used in computers are DRAM and SRAM.

DYNAMIC RAM: RAM technology is divided into two technologies: dynamic and static. A dynamic RAM (DRAM) is made with cells that store data as charge on capacitors. The presence or absence of charge in a capacitor is interpreted as a binary 1 or 0. Because capacitors have a natural tendency to discharge, dynamic RAMs require periodic charge refreshing to maintain data storage. The term *dynamic* refers to this tendency of the stored charge to leak away, even with power continuously applied.

Figure 7.4a is a typical DRAM structure for an individual cell that stores 1 bit. The address line is activated when the bit value from this cell is to be read or written. The transistor acts as a switch that is closed (allowing current to flow) if a voltage is applied to the address line and open (no current flows) if no voltage is present on the address line.

For the write operation, a voltage signal is applied to the bit line; a high voltage represents 1, and a low voltage represents 0. A signal is then applied to the address line, allowing a charge to be transferred to the capacitor.

For the read operation, when the address line is selected, the transistor turns on and the charge stored on the capacitor is fed out onto a bit line and to a sense amplifier. The sense amplifier compares the capacitor voltage to a reference value and determines if the cell contains logic 1 or logic 0. The readout from the cell discharges the capacitor, which must be restored to complete the operation.

Although the DRAM cell is used to store a single bit (0 or 1), it is essentially an analogue device. The capacitor can store any charge value within a range; a threshold value determines whether the charge is interpreted as 1 or 0.

STATIC RAM: In contrast, a static RAM (SRAM) is a digital device that uses the same logic elements used in the processor. In a SRAM, binary values are stored using traditional flip-flop logic-gate configurations. A static RAM will hold its data as long as power is supplied to it.

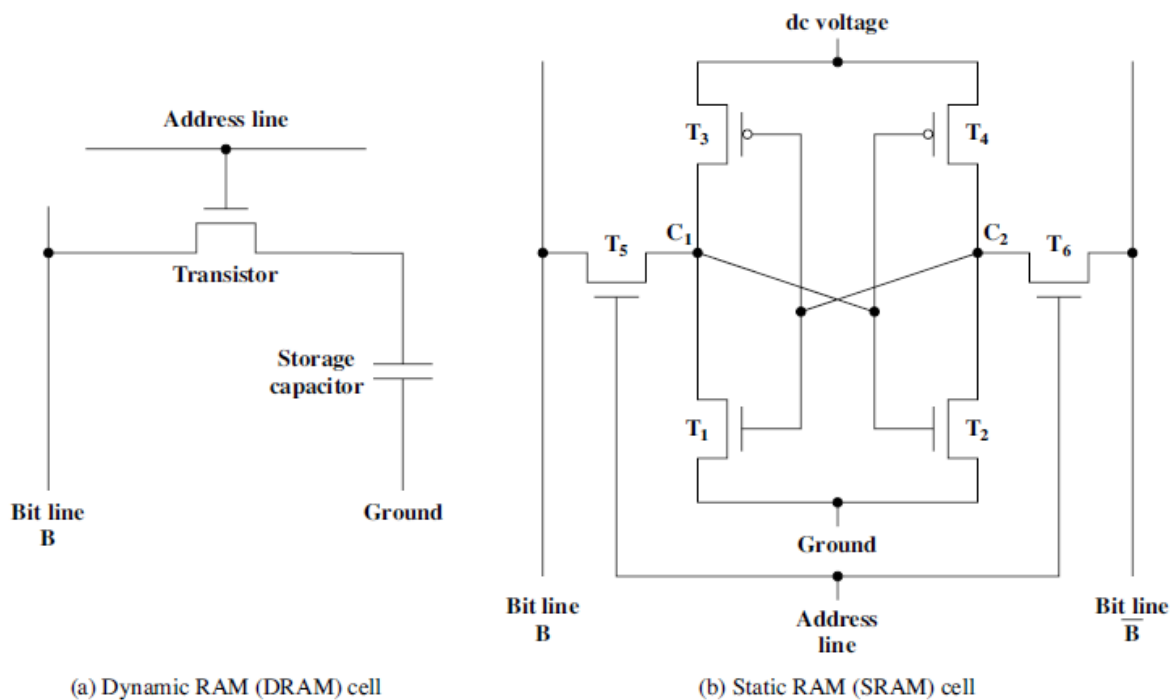


Figure 7.4 Typical Memory Cell Structures

Figure 7.4b is a typical SRAM structure for an individual cell. Four transistors (T1,T2,T3,T4) are cross connected in an arrangement that produces a stable logic state. In logic state 1, point C1 is high and point C2 is low; in this state,T1 and T4 are off and T2 and T3 are on. In logic state 0, point C1 is low and point C2 is high; in this state,T1 and T4 are on and T2 and T3 are off. Both states are stable as long as the direct current (dc) voltage is applied. Unlike the DRAM, no refresh is needed to retain data.

As in the DRAM, the SRAM address line is used to open or close a switch. The address line controls two transistors (T5 and T6).When a signal is applied to this line, the two transistors are switched on, allowing a read or write operation. For a write operation, the desired bit value is applied to line B, while its complement is applied to line. This forces the four transistors (T1, T2, T3, T4) into the proper state. For a read operation, the bit value is read from line B.

SRAM versus DRAM Both static and dynamic RAMs are volatile; that is, power must be continuously supplied to the memory to preserve the bit values. A dynamic memory cell is simpler and smaller than a static memory cell. Thus, a DRAM is more dense (smaller cells = more cells per unit area) and less expensive than a corresponding SRAM. On the other hand, a DRAM requires the supporting refresh circuitry. For larger memories, the fixed cost of the refresh circuitry is more than compensated for by the smaller variable cost of DRAM cells. Thus, DRAMs tend to be favoured for large memory requirements. A final point is that SRAMs are generally somewhat faster than DRAMs. Because of these relative characteristics, SRAM is used for cache memory (both on and off chip), and DRAM is used for main memory.

Types of ROM: As the name suggests, a **read-only memory (ROM)** contains a permanent pattern of data that cannot be changed. A ROM is non-volatile; that is, no power source is required to maintain the bit values in memory. While it is possible to read a ROM, it is not possible to write new data into it. An important application of ROMs is microprogramming, discussed in Part Four. Other potential applications include:

- Library subroutines for frequently wanted functions
- System programs
- Function tables

For a modest-sized requirement, the advantage of ROM is that the data or program is permanently in main memory and need never be loaded from a secondary storage device.

A ROM is created like any other integrated circuit chip, with the data actually wired into the chip as part of the fabrication process. This presents two problems:

- The data insertion step includes a relatively large fixed cost, whether one or thousands of copies of a particular ROM are fabricated.
- There is no room for error. If one bit is wrong, the whole batch of ROMs must be thrown out.

When only a small number of ROMs with particular memory content is needed, a less expensive alternative is the **programmable ROM (PROM)**. Like the ROM, the PROM is non-volatile and may be written into only once. For the PROM, the writing process is performed electrically and may be performed by a supplier or customer at a time later than the original chip fabrication. Special equipment is required for the writing or “programming” process. PROMs provide flexibility and convenience. The ROM remains attractive for high-volume production runs.

Another variation on read-only memory is the read-mostly memory, which is useful for applications in which read operations are far more frequent than write operations but for which non-volatile storage is required. There are three common forms of read-mostly memory: EPROM, EEPROM, and flash memory.

The optically **erasable programmable read-only memory (EPROM)** is read and written electrically, as with PROM. However, before a write operation, all the storage cells must be erased to the same initial state by exposure of the packaged chip to ultraviolet radiation. Erasure is performed by shining an intense ultraviolet light through a window that is designed into the memory chip. This erasure process can be performed repeatedly; each erasure can take as much as 20 minutes to perform. Thus, the EPROM can be altered multiple times and, like the ROM

and PROM, holds its data virtually indefinitely. For comparable amounts of storage, the EPROM is more expensive than PROM, but it has the advantage of the multiple update capability.

A more attractive form of read-mostly memory is **electrically erasable programmable read-only memory** (EEPROM). This is a read-mostly memory that can be written into at any time without erasing prior contents; only the byte or bytes addressed are updated. The write operation takes considerably longer than the read operation, on the order of several hundred microseconds per byte. The EEPROM combines the advantage of non-volatility with the flexibility of being updatable in place, using ordinary bus control, address, and data lines. EEPROM is more expensive than EPROM and also is less dense, supporting fewer bits per chip.

Another form of semiconductor memory is **flash memory** (so named because of the speed with which it can be reprogrammed). First introduced in the mid-1980s, flash memory is intermediate between EPROM and EEPROM in both cost and functionality. Like EEPROM, flash memory uses an electrical erasing technology. An entire flash memory can be erased in one or a few seconds, which is much faster than EPROM. In addition, it is possible to erase just blocks of memory rather than an entire chip. Flash memory gets its name because the microchip is organized so that a section of memory cells are erased in a single action or “flash.” However, flash memory does not provide byte-level erasure. Like EPROM, flash memory uses only one transistor per bit, and so achieves the high density (compared with EEPROM) of EPROM.

7.4 Cache Memory

Cache memory is intended to give memory speed approaching that of the fastest memories available, and at the same time provide a large memory size at the price of less expensive types of semiconductor memories. The concept is illustrated in Figure 7.5a. There is a relatively large and slow main memory together with a smaller, faster cache memory. The cache contains a copy of portions of main memory. When the processor attempts to read a word of memory, a check is made to determine if the word is in the cache. If so, the word is delivered to the processor. If not, a block of main memory, consisting of some fixed number of words, is read into the cache and then the word is delivered to the processor. Because of the phenomenon of locality of reference, when a block of data is fetched into the cache to satisfy a single memory reference, it is likely that there will be future references to that same memory location or to other words in the block.

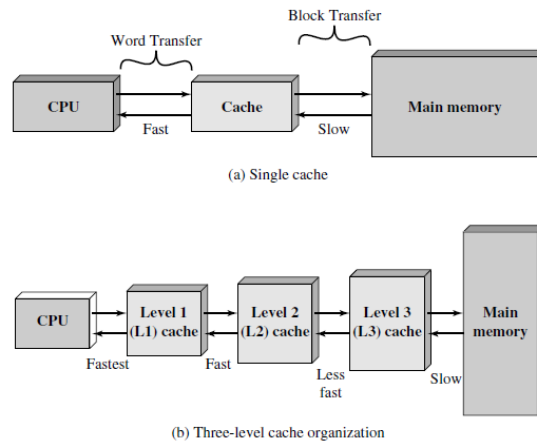


Figure 7.5 Cache and Main Memory

Figure 7.5b depicts the use of multiple levels of cache. The L2 cache is slower and typically larger than the L1 cache, and the L3 cache is slower and typically larger than the L2 cache.

Figure 7.6 depicts the structure of a cache/main-memory system. Main memory consists of up to 2^n addressable words, with each word having a unique n -bit address. For mapping purposes, this memory is considered to consist of a number of fixed length blocks of K words each. That is, there are $M = 2^n/K$ blocks in main memory. The cache consists of m blocks, called **lines**. Each line contains K words, plus a tag of a few bits. Each line also includes control bits (not shown), such as a bit to indicate whether the line has been modified since being loaded into the cache. The length of a line, not including tag and control bits, is the **line size**. The line size may be as small as 32 bits, with each “word” being a single byte; in this case the line size is 4 bytes. The number of lines is considerably less than the number of main memory blocks ($m \ll M$). At any time, some subset of the blocks of memory resides in lines in the cache. If a word in a block of memory is read, that block is transferred to one of the lines of the cache. Because there are more blocks than lines, an individual line cannot be uniquely and permanently dedicated to a particular block. Thus, each line includes a **tag** that identifies which particular block is currently being stored. The tag is usually a portion of the main memory address, as described later in this section.

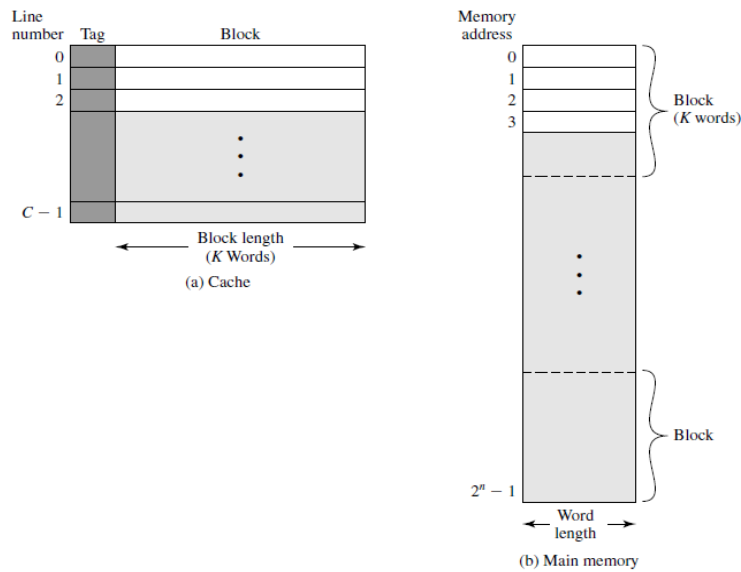


Figure 7.6 Cache/Main Memory Structure

7.4.1 Elements of Cache Design

This section provides an overview of cache design parameters and reports some typical results. We occasionally refer to the use of caches in high-performance computing (HPC). HPC deals with supercomputers and supercomputer software, especially for scientific applications that involve large amounts of data, vector and matrix computation, and the use of parallel algorithms. Cache design for HPC is quite different than for other hardware platforms and applications. Indeed, many researchers have found that HPC applications perform poorly on computer architectures that employ caches. Other researchers have since shown that a cache hierarchy can be useful in improving performance if the application software is tuned to exploit the cache.

Although there are a large number of cache implementations, there are a few basic design elements that serve to classify and differentiate cache architectures. Table 7.3 lists key elements.

Table 7.3 Elements of Cache Design

Cache Addresses	Write Policy
Logical	Write through
Physical	Write back
Cache Size	Write once
Mapping Function	Line Size
Direct	Number of caches
Associative	Single or two level Unified or split
Set Associative	
Replacement Algorithm	
Least recently used (LRU)	
First in first out (FIFO)	
Least frequently used (LFU)	
Random	

Cache Addresses: Almost all non embedded processors, and many embedded processors, support virtual memory. In essence, virtual memory is a facility that allows programs to address memory from a logical point of view, without regard to the amount of main memory physically available. When virtual memory is used, the address fields of machine instructions contain virtual addresses. For reads to and writes from main memory, a hardware memory management unit (MMU) translates each virtual address into a physical address in main memory.

When virtual addresses are used, the system designer may choose to place the cache between the processor and the MMU or between the MMU and main memory. A **logical cache**, also known as a **virtual cache**, stores data using **virtual addresses**. The processor accesses the cache directly, without going through the MMU. A physical cache stores data using main memory **physical addresses**.

One obvious advantage of the logical cache is that cache access speed is faster than for a physical cache, because the cache can respond before the MMU performs an address translation. The disadvantage has to do with the fact that most virtual memory systems supply each application with the same virtual memory address space. That is, each application sees a virtual memory that starts at address 0. Thus, the same virtual address in two different applications refers to two different physical addresses. The cache memory must therefore be completely flushed with each application context switch, or extra bits must be added to each line of the cache to identify which virtual address space this address refers to.

Cache Size: We would like the size of the cache to be small enough so that the overall average cost per bit is close to that of main memory alone and large enough so that the overall average access time is close to that of the cache alone. There are several other motivations for minimizing cache size. The larger the cache, the larger the number of gates involved in addressing the cache. The result is that large caches tend to be slightly slower than small ones—even when built with the same integrated circuit technology and put in the same place on chip and circuit board. The available chip and board area also limits cache size. Because the performance of the cache is very sensitive to the nature of the workload, it is impossible to arrive at a single “optimum” cache size.

7.4.2 Mapping Function

Because there are fewer cache lines than main memory blocks, an algorithm is needed for mapping main memory blocks into cache lines. Further, a means is needed for determining which main memory block currently occupies a cache line. The choice of the mapping function dictates how the cache is organized. Three techniques can be used: direct, associative, and set

associative. We examine each of these in turn. In each case, we look at the general structure and then a specific example.

Example 7.2 For all three cases, the example includes the following elements:

- The cache can hold 64 KBytes.
- Data are transferred between main memory and the cache in blocks of 4 bytes each. This means that the cache is organized as $16K = 2^{14}$ lines of 4 bytes each.
- The main memory consists of 16 Mbytes, with each byte directly addressable by a 24-bit address ($2^{24} = 16M$). Thus, for mapping purposes, we can consider main memory to consist of 4M blocks of 4 bytes each.

DIRECT MAPPING: The simplest technique, known as direct mapping, maps each block of main memory into only one possible cache line. The mapping is expressed as

$$i = j \text{ modulo } m$$

where,

i = cache line number

j = main memory block number

m = number of lines in the cache

Figure 7.7a shows the mapping for the first blocks of main memory. Each block of main memory maps into one unique line of the cache. The next blocks of main memory map into the cache in the same fashion; that is, block B_m of main memory maps into line L_0 of cache, block B_{m+1} maps into line L_1 , and so on.

The mapping function is easily implemented using the main memory address. Figure 7.8 illustrates the general mechanism. For purposes of cache access, each main memory address can be viewed as consisting of three fields. The least significant w bits identify a unique word or byte within a block of main memory; in most contemporary machines, the address is at the byte level. The remaining s bits specify one of the 2^s blocks of main memory. The cache logic interprets these s bits as a tag of $s - r$ bits (most significant portion) and a line field of r bits. This latter field identifies one of the $m = 2^r$ lines of the cache. To summarize,

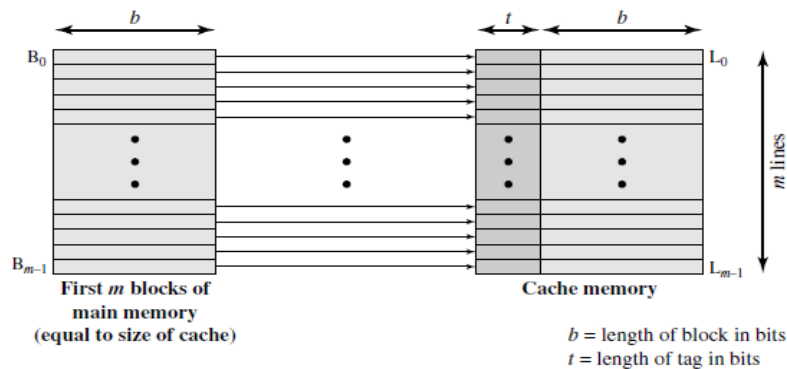
- Address length = $(s + w)$ bits
- Number of addressable units = 2^{s+w} words or bytes
- Block size = line size = 2^w words or bytes
- Number of blocks in main memory = $2^{s+w}/2^w = 2^s$

- Number of lines in cache = $m = 2^r$
- Size of cache = 2^{r+w} words or bytes
- Size of tag = $(s - r)$ bits

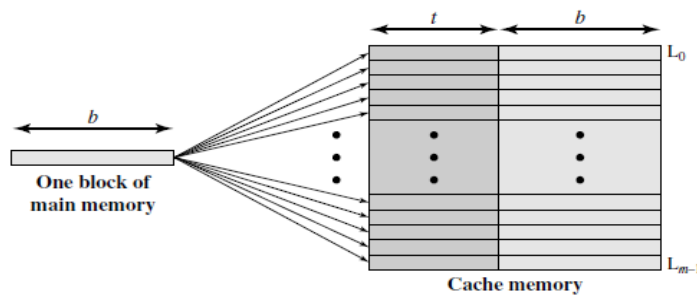
The effect of this mapping is that blocks of main memory are assigned to lines of the cache as follows:

Cache line	Main memory blocks assigned
0	0, m, 2m, ..., $2^s - m$
1	1, m+1, 2m+1, ..., $2^s - m + 1$
⋮	⋮
m-1	m-1, 2m-1, 3m-1, ..., $2^s - 1$

Thus, the use of a portion of the address as a line number provides a unique mapping of each block of main memory into the cache. When a block is actually read into its assigned line, it is necessary to tag the data to distinguish it from other blocks that can fit into that line. The most significant $s - r$ bits serve this purpose.



(a) Direct mapping



(b) Associative mapping

Figure 7.7 Mapping from Main Memory to Cache: Direct and Associative

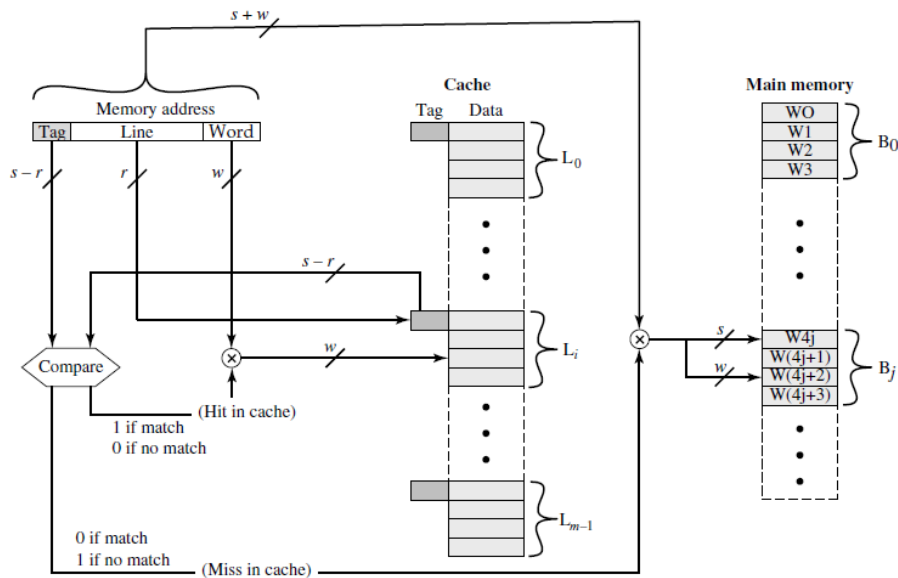


Figure 7.8 Direct-Mapping Cache Organization

Example 7.2a: Figure 7.9 shows our example system using direct mapping. In the example, $m = 16K = 2^{14}$ and $i = j \text{ modulo } 2^{14}$. The mapping becomes

Cache Line	Starting Memory Address of Block
0	000000, 010000, ..., FF0000
1	000004, 010004, ..., FF0004
...	...
$2^{14} - 1$	00FFFC, 01FFFC, ..., FFFFFC

Note that no two blocks that map into the same line number have the same tag number. Thus, blocks with starting addresses 000000, 010000, ..., FF0000 have tag numbers 00,01, ..., FF, respectively.

The direct mapping technique is simple and inexpensive to implement. Its main disadvantage is that there is a fixed cache location for any given block. Thus, if a program happens to reference words repeatedly from two different blocks that map into the same line, then the blocks will be continually swapped in the cache, and the hit ratio will be low (a phenomenon known as *thrashing*).

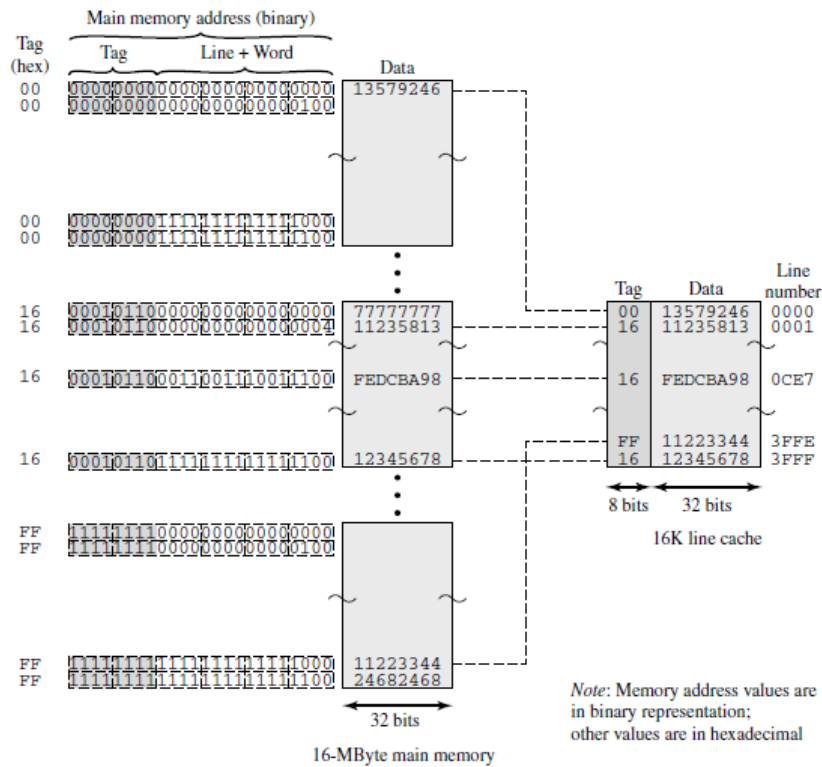


Figure 7.9 Direct Mapping Example

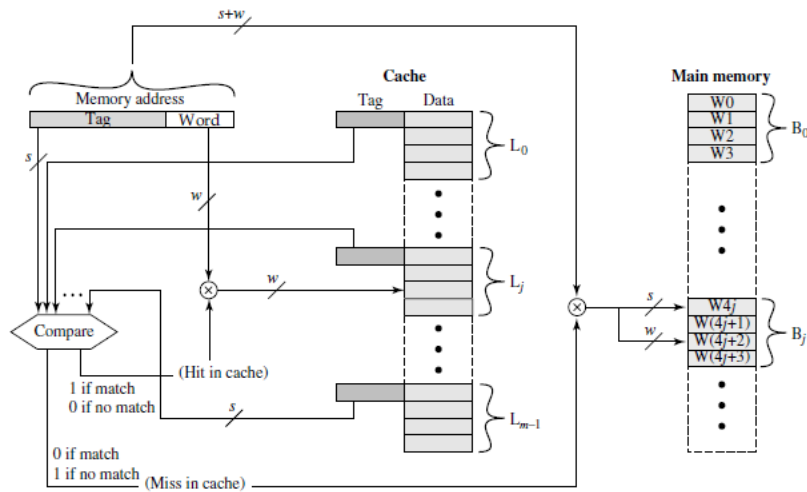


Figure 7.10 Fully Associative Cache Organization

ASSOCIATIVE MAPPING: Associative mapping overcomes the disadvantage of direct mapping by permitting each main memory block to be loaded into any line of the cache (Figure 7.7b). In this case, the cache control logic interprets a memory address simply as a Tag and a Word field. The Tag field uniquely identifies a block of main memory. To determine whether a block is in the cache, the cache control logic must simultaneously examine every line's tag

for a match. Figure 7.10 illustrates the logic. Note that no field in the address corresponds to the line number, so that the number of lines in the cache is not determined by the address format. To summarize,

- Address length = $(s + w)$ bits
- Number of addressable units = 2^{s+w} words or bytes
- Block size = line size = 2^w words or bytes
- Number of blocks in main memory = $2^{s+w} / 2^w = 2^s$
- Number of lines in cache = undetermined
- Size of tag = s bits

Example 7.2b Figure 7.11 shows our example using associative mapping. A main memory address consists of a 22-bit tag and a 2-bit byte number. The 22-bit tag must be stored with the 32-bit block of data for each line in the cache. Note that it is the leftmost (most significant) 22 bits of the address that form the tag. Thus, the 24-bit hexadecimal address 16339C has the 22-bit tag 058CE7. This is easily seen in binary notation:

memory address	0001	0110	0011	0011	1001	1100	(binary)
	1	6	3	3	9	C	(hex)
tag (leftmost 22 bits)	00	0101	1000	1100	1110	0111	(binary)
	0	5	8	C	E	7	(hex)

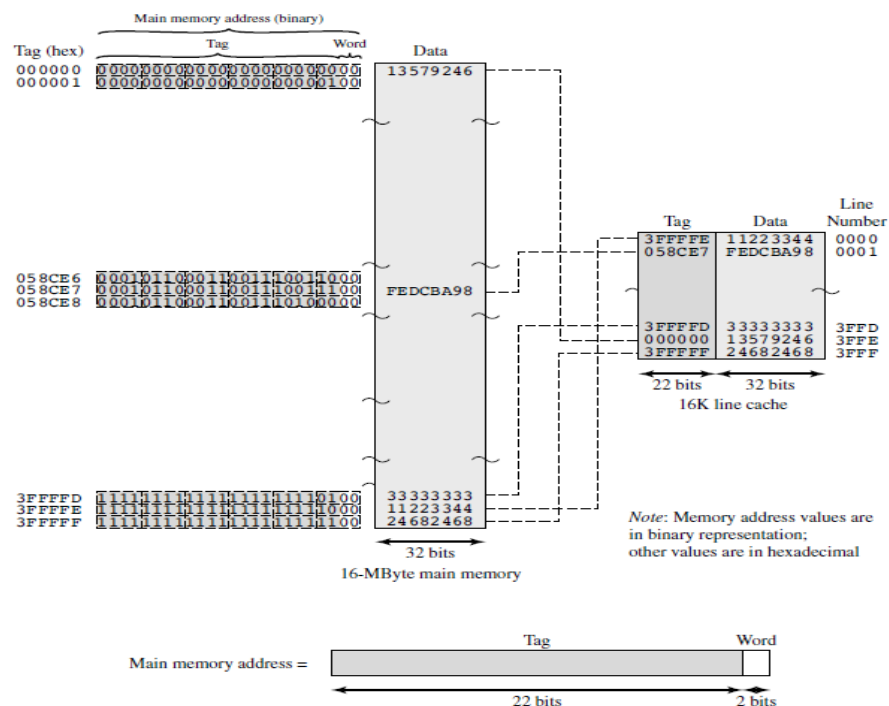


Figure 7.11 Associative Mapping Example

With associative mapping, there is flexibility as to which block to replace when a new block is read into the cache.

SET-ASSOCIATIVE MAPPING Set-associative mapping is a compromise that exhibits the strengths of both the direct and associative approaches while reducing their disadvantages.

In this case, the cache consists of a number sets, each of which consists of a number of lines.

The relationships are

$$m = n \times k$$

$$i = j \text{ modulo } v$$

where

i = cache set number

j = main memory block number

m = number of lines in the cache

v = number of sets

k = number of lines in each set

This is referred to as k-way set-associative mapping. With set-associative mapping, block B_j can be mapped into any of the lines of set j .

As with associative mapping, each word maps into multiple cache lines. For set-associative mapping, each word maps into all the cache lines in a specific set, so that main memory block B_0 maps into set 0, and so on. Thus, the set-associative cache can be physically implemented as associative caches. It is also possible to implement the set-associative cache as k direct mapping caches.

For set-associative mapping, the cache control logic interprets a memory address as three fields: Tag, Set, and Word. The d set bits specify one of $v = 2^d$ sets. The s bits of the Tag and Set fields specify one of the 2^s blocks of main memory.

Replacement Algorithms: Once the cache has been filled, when a new block is brought into the cache, one of the existing blocks must be replaced. For direct mapping, there is only one possible line for any particular block, and no choice is possible. For the associative and set-associative techniques, a replacement algorithm is needed. To achieve high speed, such an algorithm must be implemented in hardware. A number of algorithms have been tried. We mention four of the most common. Probably the most effective is least recently used (LRU): Replace that block in the set that has been in the cache longest with no reference to it. For two-way set associative, this is easily implemented. Each line includes a USE bit. When a line is referenced, its USE bit is set to 1 and the USE bit of the other line in that set is set to 0. When a block is to be read into the set, the line whose USE bit is 0 is used. Because we are assuming

that more recently used memory locations are more likely to be referenced, LRU should give the best hit ratio. LRU is also relatively easy to implement for a fully associative cache. The cache mechanism maintains a separate list of indexes to all the lines in the cache. When a line is referenced, it moves to the front of the list. For replacement, the line at the back of the list is used. Because of its simplicity of implementation, LRU is the most popular replacement algorithm.

Another possibility is first-in-first-out (FIFO): Replace that block in the set that has been in the cache longest. FIFO is easily implemented as a round-robin or circular buffer technique. Still another possibility is least frequently used (LFU): Replace that block in the set that has experienced the fewest references. LFU could be implemented by associating a counter with each line. A technique not based on usage (i.e., not LRU, LFU, FIFO, or some variant) is to pick a line at random from among the candidate lines. Simulation studies have shown that random replacement provides only slightly inferior performance to an algorithm based on usage.

Write Policy: When a block that is resident in the cache is to be replaced, there are two cases to consider. If the old block in the cache has not been altered, then it may be overwritten with a new block without first writing out the old block. If at least one write operation has been performed on a word in that line of the cache, then main memory must be updated by writing the line of cache out to the block of memory before bringing in the new block. A variety of write policies, with performance and economic trade-offs, is possible. There are two problems to contend with. First, more than one device may have access to main memory. For example, an I/O module may be able to read-write directly to memory. If a word has been altered only in the cache, then the corresponding memory word is invalid.

The simplest technique is called **write through**. Using this technique, all write operations are made to main memory as well as to the cache, ensuring that main memory is always valid. Any other processor-cache module can monitor traffic to main memory to maintain consistency within its own cache. The main disadvantage of this technique is that it generates substantial memory traffic and may create a bottleneck. An alternative technique, known as **write back**, minimizes memory writes. With write back, updates are made only in the cache. When an update occurs, a **dirty bit**, or **use bit**, associated with the line is set. Then, when a block is replaced, it is written back to main memory if and only if the dirty bit is set. The problem with write back is that portions of main memory are invalid, and hence accesses by I/O modules can be allowed only through the cache.

7.5 External Memory

This section examines a range of external memory devices and systems. We begin with the most important device, the magnetic disk. Magnetic disks are the foundation of external memory on virtually all computer systems. The next section examines the use of disk arrays to achieve greater performance, looking specifically at the family of systems known as RAID (Redundant Array of Independent Disks). An increasingly important component of many computer systems is external optical memory, and this is examined in the third section. Finally, magnetic tape is described.

7.5.1 Magnetic Disk

A disk is a circular platter constructed of nonmagnetic material, called the substrate, coated with a magnetisable material. Traditionally, the substrate has been an aluminum or aluminum alloy material. More recently, glass substrates have been introduced. The glass substrate has a number of benefits, including the following:

- Improvement in the uniformity of the magnetic film surface to increase disk reliability
- A significant reduction in overall surface defects to help reduce read-write errors
- Ability to support lower fly heights (described subsequently)
- Better stiffness to reduce disk dynamics
- Greater ability to withstand shock and damage

Magnetic Read and Write Mechanisms: Data are recorded on and later retrieved from the disk via a conducting coil named the **head**; in many systems, there are two heads, a read head and a write head. During a read or write operation, the head is stationary while the platter rotates beneath it.

The write mechanism exploits the fact that electricity flowing through a coil produces a magnetic field. Electric pulses are sent to the write head, and the resulting magnetic patterns are recorded on the surface below, with different patterns for positive and negative currents. The write head itself is made of easily magnetisable material and is in the shape of a rectangular doughnut with a gap along one side and a few turns of conducting wire along the opposite side (Figure 7.12). An electric current in the wire induces a magnetic field across the gap, which in turn magnetizes a small area of the recording medium. Reversing the direction of the current reverses the direction of the magnetization on the recording medium.

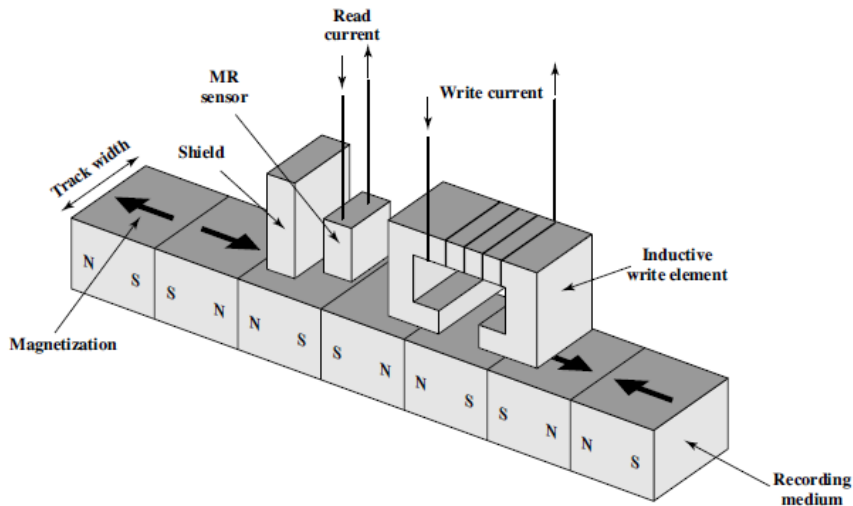


Figure 7.12 Inductive Write/Magneto resistive Read Head

The traditional read mechanism exploits the fact that a magnetic field moving relative to a coil produces an electrical current in the coil. When the surface of the disk passes under the head, it generates a current of the same polarity as the one already recorded. The structure of the head for reading is in this case essentially the same as for writing and therefore the same head can be used for both. Such single heads are used in floppy disk systems and in older rigid disk systems.

Contemporary rigid disk systems use a different read mechanism, requiring a separate read head, positioned for convenience close to the write head. The read head consists of a partially shielded magneto resistive (MR) sensor. The MR material has an electrical resistance that depends on the direction of the magnetization of the medium moving under it.

Data Organization and Formatting: The head is a relatively small device capable of reading from or writing to a portion of the platter rotating beneath it. This gives rise to the organization of data on the platter in a concentric set of rings, called **tracks**. Each track is the same width as the head. There are thousands of tracks per surface.

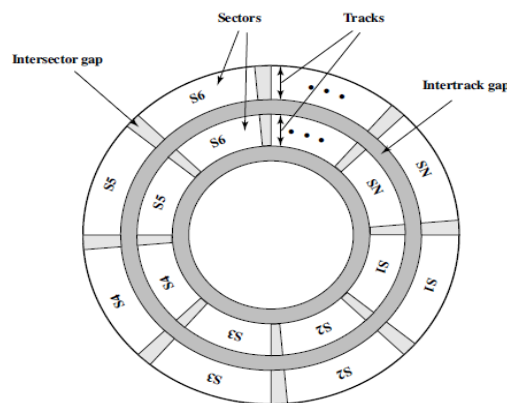


Figure 7.13 Disk Data Layout

Figure 7.13 depicts this data layout. Adjacent tracks are separated by **gaps**. This prevents, or at least minimizes, errors due to misalignment of the head or simply interference of magnetic fields.

Data are transferred to and from the disk in **sectors** (Figure 7.13). There are typically hundreds of sectors per track, and these may be of either fixed or variable length. In most contemporary systems, fixed-length sectors are used, with 512 bytes being the nearly universal sector size. To avoid imposing unreasonable precision requirements on the system, adjacent sectors are separated by intra-track (inter-sector) gaps.

A bit near the centre of a rotating disk travels past a fixed point (such as a read–write head) slower than a bit on the outside. Therefore, some way must be found to compensate for the variation in speed so that the head can read all the bits at the same rate. This can be done by increasing the spacing between bits of information recorded in segments of the disk. The information can then be scanned at the same rate by rotating the disk at a fixed speed, known as the **constant angular velocity (CAV)**.

Because the **density**, in bits per linear inch, increases in moving from the outermost track to the innermost track, disk storage capacity in a straightforward CAV system is limited by the maximum recording density that can be achieved on the innermost track. To increase density, modern hard disk systems use a technique known as **multiple zone recording**, in which the surface is divided into a number of concentric zones (16 is typical). Within a zone, the number of bits per track is constant. Zones farther from the centre contain more bits (more sectors) than zones closer to the centre. This allows for greater overall storage capacity at the expense of somewhat more complex circuitry. As the disk head moves from one zone to another, the length (along the track) of individual bits changes, causing a change in the timing for reads and writes.

Physical Characteristics: Table 7.4 lists the major characteristics that differentiate among the various types of magnetic disks. First, the head may either be fixed or movable with respect to the radial direction of the platter. In a **fixed-head disk**, there is one read-write head per track. All of the heads are mounted on a rigid arm that extends across all tracks; such systems are rare today. In a **movable-head disk**, there is only one read-write head. Again, the head is mounted on an arm. Because the head must be able to be positioned above any track, the arm can be extended or retracted for this purpose.

The disk itself is mounted in a disk drive, which consists of the arm, a spindle that rotates the disk, and the electronics needed for input and output of binary data. A **non-removable disk** is permanently mounted in the disk drive; the hard disk in a personal computer is a non-removable disk. A **removable disk** can be removed and replaced with another disk. The advantage of the

latter type is that unlimited amounts of data are available with a limited number of disk systems. Furthermore, such a disk may be moved from one computer system to another. Floppy disks and ZIP cartridge disks are examples of removable disks. For most disks, the magnetisable coating is applied to both sides of the platter, which is then referred to as **double-sided**. Some less expensive disk systems use **single-sided** disks.

Table 7.4 Physical Characteristics of Disk Systems

Head Motion	Single sided
Fixed head (one per track)	Double sided
Movable head (one per surface)	Platters
Platters	Single platter
Single platter	Multiple platter
Multiple platter	Head Mechanism
Disk Portability	Contact (floppy)
Non removable disk	Fixed gap
Removable disk	Aerodynamic gap (Winchester)
Head Mechanism	
Contact (floppy)	
Fixed gap	
Aerodynamic gap (Winchester)	
Sides	

7.5.2 RAID Technology

As discussed earlier, the rate in improvement in secondary storage performance has been considerably less than the rate for processors and main memory. This mismatch has made the disk storage system perhaps the main focus of concern in improving overall computer system performance. As in other areas of computer performance, disk storage designers recognize that if one component can only be pushed so far, additional gains in performance are to be had by using multiple parallel components. In the case of disk storage, this leads to the development of arrays of disks that operate independently and in parallel. With multiple disks, separate I/O requests can be handled in parallel, as long as the data required reside on separate disks. Further, a single I/O request can be executed in parallel if the block of data to be accessed is distributed across multiple disks.

With the use of multiple disks, there is a wide variety of ways in which the data can be organized and in which redundancy can be added to improve reliability. This could make it difficult to develop database schemes that are usable on a number of platforms and operating systems. Fortunately, industry has agreed on a standardized scheme for multiple-disk database design, known as RAID (Redundant Array of Independent Disks). The RAID scheme consists of seven levels, zero through six. These levels do not imply a hierarchical relationship but designate different design architectures that share three common characteristics:

1. RAID is a set of physical disk drives viewed by the operating system as a single logical drive.
2. Data are distributed across the physical drives of an array in a scheme known as striping, described subsequently.

3. Redundant disk capacity is used to store parity information, which guarantees data recoverability in case of a disk failure.

The details of the second and third characteristics differ for the different RAID levels. RAID 0 and RAID 1 do not support the third characteristic.

The term *RAID* was originally coined in a paper by a group of researchers at the University of California at Berkeley [2][5]. The paper outlined various RAID configurations and applications and introduced the definitions of the RAID levels that are still used. The RAID strategy employs multiple disk drives and distributes data in such a way as to enable simultaneous access to data from multiple drives, thereby improving I/O performance and allowing easier incremental increases in capacity.

The unique contribution of the RAID proposal is to address effectively the need for redundancy. Although allowing multiple heads and actuators to operate simultaneously achieves higher I/O and transfer rates, the use of multiple devices increases the probability of failure. To compensate for this decreased reliability, RAID makes use of stored parity information that enables the recovery of data lost due to a disk failure.

We now examine each of the RAID levels. Table 7.5 provides a rough guide to the seven levels. In the table, I/O performance is shown both in terms of data transfer capacity, or ability to move data, and I/O request rate, or ability to satisfy I/O requests, since these RAID levels inherently perform differently relative to these two metrics.

Table 7.5 RAID Levels

Category	Level	Description	Disks Required	Data Availability	Large I/O Data Transfer Capacity	Small I/O Request Rate
Striping	0	Non-redundant	N	Lower than single disk	Very high	Very high for both read
Mirroring	1	Mirrored	2N	Higher than RAID 2, 3, 4, or 5; lower than	Higher than single disk for read; similar to	Up to twice that of a single disk for read; similar

				RAID 6	single disk for write	to single disk for write
Parallel access	2	Redundant via Hamming Code	N+m	Much higher than single disk; comparable to RAID 3, 4, or 5	Highest of all listed alternatives	Approximately twice that of a single disk
	3	Bit-interleaved parity	N+1	Much higher than single disk; comparable to RAID 2, 4, or 5	Highest of all listed alternatives	Approximately twice that of a single disk
Independent Access	4	Block-interleaved Parity	N+1	Much higher than single disk; comparable to RAID 2, 3, or 5	Similar to RAID 0 for read; significantly lower than single disk for write	Similar to RAID 0 for read; significantly lower than single disk for write
	5	Block-interleaved distributed parity	N+1	Much higher than single disk; comparable to RAID 2, 3, or 4	Similar to RAID 0 for read; lower than single disk for write	Similar to RAID 0 for read; generally lower than single disk for write
	6	Block-interleaved dual distributed parity	N+2	Highest of all listed alternatives	Similar to RAID 0 for read; lower than RAID 5 for write	Similar to RAID 0 for read; significantly lower than RAID 5 for write

N = number of data disks; m proportional to $\log N$

Table 7.6 is a comparative summary of the seven levels.

Level	Advantages	Disadvantages	Applications
⁰	I/O performance is greatly improved by spreading the I/O load across many channels and drives No parity calculation overhead is involved Very simple design Easy to implement	The failure of just one drive will result in all data in an array being lost	Video production and editing Image Editing Pre-press applications Any application requiring high bandwidth
¹	100% redundancy of data means no rebuild is necessary in case of a disk failure, just a copy to the replacement disk Under certain circumstances, RAID 1 can sustain multiple simultaneous drive failures Simplest RAID storage subsystem Design	Highest disk overhead of all RAID types (100%)—inefficient	Accounting Payroll Financial Any application requiring very high availability

²	Extremely high data transfer rates possible The higher the data transfer rate required, the better the ratio of data disks to ECC disks Relatively simple controller design compared to RAID levels 3, 4 & 5	Very high ratio of ECC disks to data disks with smaller word sizes—inefficient Entry level cost very high—requires very high transfer rate requirement to justify	No commercial implementations exist/ not commercially viable
³	Very high read data transfer rate Very high write data transfer rate Disk failure has an insignificant impact on throughput Low ratio of ECC (parity) disks to data disks means high efficiency	Transaction rate equal to that of a single disk drive at best (if spindles are synchronized) Controller design is fairly complex	Video production and live streaming Image editing Video editing Prepress applications Any application requiring high throughput
⁴	Very high Read data transaction rate Low ratio of ECC (parity) disks to data disks means high efficiency	Quite complex controller design Worst write transaction rate and Write aggregate transfer rate Difficult and inefficient data rebuild in the event of disk failure	No commercial implementations exist/ not commercially viable
⁵	Highest Read data transaction rate Low ratio of ECC (parity) disks to data disks means high efficiency Good aggregate transfer rate	Most complex controller design Difficult to rebuild in the event of a disk failure (as compared to RAID level 1)	File and application servers Database servers Web, e-mail, and news servers Intranet servers Most versatile RAID level
⁶	Provides for an extremely high data fault tolerance and can sustain multiple simultaneous drive failures	More complex controller design Controller overhead to compute parity addresses is extremely high	Perfect solution for mission critical applications

7.5.3 Optical Memory

In 1983, one of the most successful consumer products of all time was introduced: the compact disk (CD) digital audio system. The CD is a non-erasable disk that can store more than 60 minutes of audio information on one side. The huge commercial success of the CD enabled the development of low-cost optical-disk storage technology that has revolutionized computer data storage.

Compact Disk: Both the audio CD and the CD-ROM (compact disk read-only memory) share a similar technology. The main difference is that CD-ROM players are more rugged and have error correction devices to ensure that data are properly transferred from disk to computer. Both types of disk are made the same way. The disk is formed from a resin, such as polycarbonate. Digitally recorded information (either music or computer data) is imprinted as a series of microscopic pits on the surface of the polycarbonate. This is done, first of all, with a finely focused, high-intensity laser to create a master disk. The master is used, in turn, to make a die to stamp out copies onto polycarbonate.

The pitted surface is then coated with a highly reflective surface, usually aluminum or gold. This shiny surface is protected against dust and scratches by a topcoat of clear acrylic. Finally, a label can be silkscreened onto the acrylic.

Data on the CD-ROM are organized as a sequence of blocks. It consists of the following fields:

- **Sync:** The sync field identifies the beginning of a block. It consists of a byte of all 0s, 10 bytes of all 1s, and a byte of all 0s.
- **Header:** The header contains the block address and the mode byte. Mode 0 specifies a blank data field; mode 1 specifies the use of an error-correcting code and 2048 bytes of data; mode 2 specifies 2336 bytes of user data with no error-correcting code.
- **Data:** User data.
- **Auxiliary:** Additional user data in mode 2. In mode 1, this is a 288-byte error-correcting code.

Digital Versatile Disk: With the capacious digital versatile disk (DVD), the electronics industry has at last found an acceptable replacement for the analog VHS

video tape. The DVD has replaced the videotape used in video cassette recorders (VCRs) and, more important for this discussion, replace the CD-ROM in personal computers and servers. The DVD takes video into the digital age. It delivers movies with impressive picture quality, and it can be randomly accessed like audio CDs, which DVD machines can also play. Vast volumes of data can be crammed onto the disk, currently seven times as much as a CD-ROM. With DVD's huge storage capacity and vivid quality, PC games have become more realistic and educational software incorporates more video.

A variety of optical-disk systems have been introduced and reviewed below briefly.

CD (Compact Disk)-A non-erasable disk that stores digitized audio information. The standard system uses 12-cm disks and can record more than 60 minutes of uninterrupted playing time.

CD-ROM (Compact Disk Read-Only Memory)-A non-erasable disk used for storing computer data. The standard system uses 12-cm disks and can hold more than 650 Mbytes.

CD-R (CD Recordable)- Similar to a CD-ROM. The user can write to the disk only once.

CD-RW (CD Rewritable)-Similar to a CD-ROM. The user can erase and rewrite to the disk multiple times.

DVD (Digital Versatile Disk)- A technology for producing digitized, compressed representation of video information, as well as large volumes of other digital data. Both 8 and 12 cm diameters are used, with a double-sided capacity of up to 17 Gbytes. The basic DVD is read-only (DVD-ROM).

DVD-R (DVD Recordable)- Similar to a DVD-ROM. The user can write to the disk only once. Only one-sided disks can be used.

DVD-RW (DVD Rewritable)- Similar to a DVD-ROM. The user can erase and rewrite to the disk multiple times. Only one-sided disks can be used.

Blu-Ray DVD (High definition video disk)- Provides considerably greater data storage density than DVD, using a 405-nm(blue-violet) laser. A single layer on a single side can store 25 Gbytes.

7.5.4 Magnetic Tape

Tape systems use the same reading and recording techniques as disk systems. The medium is flexible polyester (similar to that used in some clothing) tape coated with magnetizable material. The coating may consist of particles of pure metal in special binders or vapor-plated metal films. The tape and the tape drive are analogous to a home tape recorder system. Tape widths vary from 0.38 cm (0.15 inch) to 1.27 cm (0.5 inch). Tapes used to be packaged as open reels that have to be threaded through a second spindle for use. Today, virtually all tapes are housed in cartridges.

Data on the tape are structured as a number of parallel tracks running lengthwise. Earlier tape systems typically used nine tracks. This made it possible to store data one byte at a time, with an additional parity bit as the ninth track. This was followed by tape systems using 18 or 36 tracks, corresponding to a digital word or double word. The recording of data in this form is referred to as **parallel recording**. Most modern systems instead use **serial recording**, in which data are laid out as a sequence of bits along each track, as is done with magnetic disks. As with the disk, data are read and written in contiguous blocks, called *physical records*, on a tape. Blocks on the tape are separated by gaps referred to as *inter-record* gaps. As with the disk, the tape is formatted to assist in locating physical records.

The typical recording technique used in serial tapes is referred to as **serpentine recording**. In this technique, when data are being recorded, the first set of bits is

recorded along the whole length of the tape. When the end of the tape is reached, the heads are repositioned to record a new track, and the tape is again recorded on its whole length, this time in the opposite direction. That process continues, back and forth, until the tape is full. To increase speed, the read-write head is capable of reading and writing a number of adjacent tracks simultaneously (typically two to eight tracks). Data are still recorded serially along individual tracks, but blocks in sequence are stored on adjacent tracks.

A tape drive is a *sequential-access* device. If the tape head is positioned at record 1, then to read record N, it is necessary to read physical records 1 through N-1, one at a time. If the head is currently positioned beyond the desired record, it is necessary to rewind the tape certain distance and begin reading forward. Unlike the disk, the tape is in motion only during a read or writes operation.

In contrast to the tape, the disk drive is referred to as a *direct-access* device .A disk drive need not read all the sectors on a disk sequentially to get to the desired one. It must only wait for the intervening sectors within one track and can make successive accesses to any track.

Magnetic tape was the first kind of secondary memory. It is still widely used as the lowest-cost, slowest-speed member of the memory hierarchy.

The dominant tape technology today is a cartridge system known as linear tape-open (LTO). LTO was developed in the late 1990s as an open-source alternative to the various proprietary systems on the market.

Summary

Computer memory is organized into a hierarchy. At the highest level (closest to the processor) are the processor registers. Next comes one or more levels of cache, When

multiple levels are used, they are denoted L1, L2, and so on. Next comes main memory, which is usually made out of dynamic random-access memory (DRAM). All of these are considered internal to the computer system. The hierarchy continues with external memory, with the next level typically being a fixed hard disk, and one or more levels below that consisting of removable media such as optical disks and tape. As one goes down the memory hierarchy, one finds decreasing cost/bit, increasing capacity, and slower access time. It would be nice to use only the fastest memory, but because that is the most expensive memory, we trade off access time for cost by using more of the slower memory. The design challenge is to organize the data and programs in memory so that the accessed memory words are usually in the faster memory. In general, it is likely that most future accesses to main memory by the processor will be to locations recently accessed. So the cache automatically retains a copy of some of the recently used words from the DRAM. Magnetic disks remain the most important component of external memory. Both removable and fixed, or hard, disks are used in systems ranging from personal computers to mainframes and supercomputers. To achieve greater performance and higher availability, servers and larger systems use RAID disk technology. RAID is a family of techniques for using multiple disks as a parallel array of data storage devices, with redundancy built in to compensate for disk failure.

Review Questions & Problems

- 1) Consider a memory of 32 blocks (labelled 0 through 31) and a cache of 8 blocks (labelled 0 through 7). In the questions below, list only correct results.

- a) Under direct mapping, which blocks of memory contend for block 2 of the cache?
- b) Under 4-way set associatively, to which blocks of cache may element 31 of memory go?
- c) In the following sequence of memory block references from the CPU, beginning from an empty cache, on which reference must be cache layout for a direct-mapped and a 4-way set associative cache first differ?

Order of reference	1	2	3	4	5	6	7	8
Block referenced	0	15	18	5	1	13	15	26

- 2) What are the differences between a write-allocate and no-write-allocate policy in a cache?
- 3) A computer has a cache, main memory, and a disk used for virtual memory. An access to the cache takes 10 ns. An access to main memory takes 100 ns. An access to the disk takes 10,000 ns. Suppose the cache hit ratio is 0.9 and the main memory hit ratio is 0.8. What is the effective Access time (EAT) in ns required to access a referenced word on this system?
- 4) The following is an eight bit data word with ECC and one bit parity: 0100101111010. The bits are in the order M8M7M6M5M4M3M2M1C8C4C2C1P. Is this word correct, does it have a one-bit error, or does it have a two-bit error? If it has a one-bit error, correct it.
- 5) For each of the following cases, state whether SRAMs or DRAMs would be more appropriate building blocks for the memory system, and state why. Assume that there is only one level of internal memory.
 - a) A memory system where performance is the most important goal.

- b) A memory system where cost is the most important factor
- 6) What are the advantages and disadvantages of smaller vs. Larger disk sector size?
- 7) A magnetic disk with 5 platters has 2048 tracks/platter, 1024 sectors/track (fixed number of sectors per track), and 512-byte sectors. What is its total capacity?

References

1. Blaauw, G., and Brooks, F. *Computer Architecture: Concepts and Evolution*. Reading, MA: Addison-Wesley, 1997
2. Stallings, W. "Reduced Instruction Set Computer Architecture." *Proceedings of the IEEE*, January 1988.
3. Stallings, W. *Computer Architecture and Organizations Designing for Performance, Eighth Edition*. Upper Saddle River, NJ: Prentice Hall, 2010.
4. Bailey, D. "RISC Microprocessors and Scientific Computing." *Proceedings, Supercomputing '93*, 1993.

CHAPTER EIGHT: INPUT/OUTPUT ORGANIZATION

Objectives

After the completion of this chapter students will be able to:

- ⇒ Understand the I/O architecture and interfaces
- ⇒ Understand the I/O modules and interrupts
- ⇒ Knows the concept of I/O interfacing with the outside world or peripherals

Key Concepts

- The computer system's I/O architecture is its interface to the outside world
- **Programmed I/O**, in which I/O occurs under the direct and continuous control of the program requesting the I/O operation
- **Interrupt-driven I/O**, in which a program issues an I/O command and then continues to execute,
- **Direct memory access (DMA)**, in which a specialized I/O processor takes over control of an I/O operation to move a large block of data.
- Two important examples of external I/O interfaces are **FireWire** and **Infiniband**.

Introduction

In addition to the processor and a set of memory modules, the third key element of a computer system is a set of I/O modules. Each module interfaces to the system bus or

central switch, and controls one or more peripheral devices. An I/O module is not simply a set of mechanical connectors that wire a device into the system bus. Rather, the I/O module contains logic for performing a communication function between the peripheral and the bus.

The reader may wonder why one does not connect peripherals directly to the system bus. The reasons are as follows:

- There are a wide variety of peripherals with various methods of operation. It would be impractical to incorporate the necessary logic within the processor to control a range of devices.
- The data transfer rate of peripherals is often much slower than that of the memory or processor. Thus, it is impractical to use the high-speed system bus to communicate directly with a peripheral.
- On the other hand, the data transfer rate of some peripherals is faster than that of the memory or processor. Again, the mismatch would lead to inefficiencies if not managed properly.
- Peripherals often use different data formats and word lengths than the computer to which they are attached.

Thus, an I/O module is required. This module has two major functions:

- Interface to the processor and memory via the system bus or central switch
- Interface to one or more peripheral devices by tailored data links

8.1 External/ Peripheral Devices

I/O operations are accomplished through a wide assortment of external devices that provide a means of exchanging data between the external environment and the

computer. An external device attaches to the computer by a link to an I/O module. The link is used to exchange control, status, and data between the I/O module and the external device. An external device connected to an I/O module is often referred to as a *peripheral device* or, simply, a *peripheral*.

We can broadly classify external devices into three categories:

- **Human readable:** Suitable for communicating with the computer user
- **Machine readable:** Suitable for communicating with equipment
- **Communication:** Suitable for communicating with remote devices

Examples of human-readable devices are video display terminals (VDTs) and printers. Examples of machine-readable devices are magnetic disk and tape systems, and sensors and actuators, such as are used in a robotics application. Note that we are viewing disk and tape systems as I/O devices in this chapter, whereas in the previous chapter (chapter 7) we viewed them as memory devices. From a functional point of view, these devices are part of the memory hierarchy, and their use is appropriately discussed in Chapter 7. From a structural point of view, these devices are controlled by I/O modules and are hence to be considered in this chapter.

Communication devices allow a computer to exchange data with a remote device, which may be a human-readable device, such as a terminal, a machine-readable device, or even another computer.

In very general terms, the nature of an external device is indicated in Figure 8.1. The interface to the I/O module is in the form of control, data, and status signals. *Control signals* determine the function that the device will perform, such as send data to the I/O module (INPUT or READ), accept data from the I/O module (OUTPUT or WRITE), report status, or perform some control function particular to the device (e.g., position a disk head). *Data* are in the form of a set of bits to be sent to or received

from the I/O module. *Status signals* indicate the state of the device. Examples are READY/NOT-READY to show whether the device is ready for data transfer.

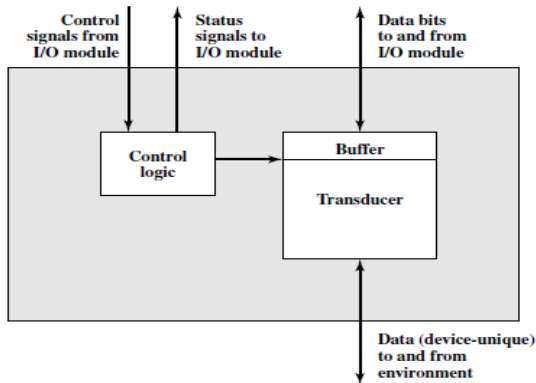


Figure 8.1 Block Diagram of an External Device

Keyboard/Monitor: The most common means of computer/user interaction is a keyboard/monitor arrangement. The user provides input through the keyboard. This input is then transmitted to the computer and may also be displayed on the monitor. In addition, the monitor displays data provided by the computer.

The basic unit of exchange is the character. Associated with each character is a code, typically 7 or 8 bits in length. The most commonly used text code is the International Reference Alphabet (IRA). Each character in this code is represented by a unique 7-bit binary code; thus, 128 different characters can be represented. Characters are of two types: printable and control. Printable characters are the alphabetic, numeric, and special characters that can be printed on paper or displayed on a screen. Some of the control characters have to do with controlling the printing or displaying of characters; an example is carriage return. Other control characters are concerned with communications procedures.

For keyboard input, when the user depresses a key, this generates an electronic signal that is interpreted by the transducer in the keyboard and translated into the bit pattern of the corresponding IRA code. This bit pattern is then transmitted to the I/O module in the computer. At the computer, the text can be stored in the same IRA code. On output, IRA code characters are transmitted to an external device from the I/O module. The transducer at the device interprets this code and sends the required electronic signals to the output device either to display the indicated character or perform the requested control function.

Disk Drive: A disk drive contains electronics for exchanging data, control, and status signals with an I/O module plus the electronics for controlling the disk read/write mechanism. In a fixed-head disk, the transducer is capable of converting between the magnetic patterns on the moving disk surface and bits in the device's buffer (Figure 8.1). A moving-head disk must also be able to cause the disk arm to move radially in and out across the disk's surface.

8.2 Input-Output Interface

Types of Interfaces: The interface to a peripheral from an I/O module must be tailored to the nature and operation of the peripheral. One major characteristic of the interface is whether it is serial or parallel (Figure 8.2). In a **parallel interface**, there are multiple lines connecting the I/O module and the peripheral, and multiple bits are transferred simultaneously, just as all of the bits of a word are transferred simultaneously over the data bus. In a **serial interface**, there is only one line used to transmit data, and bits must be transmitted one at a time. A parallel interface has traditionally been used for higher-speed peripherals, such as tape and disk, while the serial interface has traditionally been used for printers and terminals. With a new

generation of high-speed serial interfaces, parallel interfaces are becoming much less common.

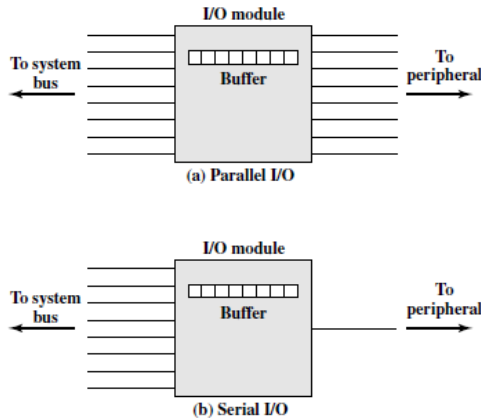


Figure 8.2 Parallel and Serial I/O

In either case, the I/O module must engage in a dialogue with the peripheral. In general terms, the dialogue for a write operation is as follows:

1. The I/O module sends a control signal requesting permission to send data.
2. The peripheral acknowledges the request.
3. The I/O module transfers data (one word or a block depending on the peripheral).
4. The peripheral acknowledges receipt of the data.

A read operation proceeds similarly.

Key to the operation of an I/O module is an internal buffer that can store data being passed between the peripheral and the rest of the system. This buffer allows the I/O

module to compensate for the differences in speed between the system bus and its external lines.

Point-to-Point and Multipoint Configurations: The connection between an I/O module in a computer system and external devices can be either point-to-point or multipoint. A point-to-point interface provides a dedicated line between the I/O module and the external device. On small systems (PCs, workstations), typical point-to-point links include those to the keyboard, printer, and external modem. A typical example of such an interface is the EIA-232 specification [2][5].

Of increasing importance are multipoint external interfaces, used to support external mass storage devices (disk and tape drives) and multimedia devices (CD-ROMs, video, audio). These multipoint interfaces are in effect external buses, and they exhibit the same type of logic as the buses. In this section, we look at two key examples: FireWire and Infiniband.

FireWire Serial Bus: With processor speeds reaching gigahertz range and storage devices holding multiple gigabits, the I/O demands for personal computers, workstations, and servers are formidable. Yet the high-speed I/O channel technologies that have been developed for mainframe and super-computer systems are too expensive and bulky for use on these smaller systems. Accordingly, there has been great interest in developing a high-speed alternative to Small Computer System Interface (SCSI) and other small system I/O interfaces. The result is the IEEE standard 1394, for a High Performance Serial Bus, commonly known as FireWire.

FireWire has a number of advantages over older I/O interfaces. It is very high speed, low cost, and easy to implement. In fact, FireWire is finding favor not only for computer systems, but also in consumer electronics products, such as digital cameras,

DVD players/recorders, and televisions. In these products, FireWire is used to transport video images, which are increasingly coming from digitized sources.

One of the strengths of the FireWire interface is that it uses serial transmission (bit at a time) rather than parallel. Parallel interfaces, such as SCSI, require more wires, which means wider, more expensive cables and wider, more expensive connectors with more pins to bend or break. A cable with more wires requires shielding to prevent electrical interference between the wires. Also, with a parallel interface, synchronization between wires becomes a requirement, a problem that gets worse with increased cable length.

In addition, computers are getting physically smaller even as they expand in computing power and I/O needs. Handheld and pocket-size computers have little room for connectors yet need high data rates to handle images and video. The intent of FireWire is to provide a single I/O interface with a simple connector that can handle numerous devices through a single port, so that the mouse, laser printer, external disk drive, sound, and local area network hook-ups can be replaced with this single connector.

InfiniBand: InfiniBand is a recent I/O specification aimed at the high-end server market. The first version of the specification was released in early 2001 and has attracted numerous vendors. The standard describes architecture and specifications for data flow among processors and intelligent I/O devices. InfiniBand has become a popular interface for storage area networking and other large storage configurations. In essence, InfiniBand enables servers, remote storage, and other network devices to be attached in a central fabric of switches and links. The switch-based architecture can connect up to 64,000 servers, storage systems, and networking devices.

8.3 Priority Interrupts

The problem with programmed I/O is that the processor has to wait a long time for the I/O module of concern to be ready for either reception or transmission of data. The processor, while waiting, must repeatedly interrogate the status of the I/O module. As a result, the level of the performance of the entire system is severely degraded.

An alternative is for the processor to issue an I/O command to a module and then go on to do some other useful work. The I/O module will then interrupt the processor to request service when it is ready to exchange data with the processor. The processor then executes the data transfer, as before, and then resumes its former processing.

Let us consider how this works, first from the point of view of the I/O module. For input, the I/O module receives a READ command from the processor. The I/O module then proceeds to read data in from an associated peripheral. Once the data are in the module's data register, the module signals an interrupt to the processor over a control line. The module then waits until its data are requested by the processor. When the request is made, the module places its data on the data bus and is then ready for another I/O operation.

From the processor's point of view, the action for input is as follows. The processor issues a READ command. It then goes off and does something else (e.g., the processor may be working on several different programs at the same time). At the end of each instruction cycle, the processor checks for interrupts. When the interrupt from the I/O module occurs, the processor saves the context (e.g., program counter and processor registers) of the current program and processes the interrupt. In this case, the processor reads the word of data from the I/O module and stores it in memory. It then restores

the context of the program it was working on (or some other program) and resumes execution.

Interrupt Processing: Let us consider the role of the processor in interrupt-driven I/O in more detail. The occurrence of an interrupt triggers a number of events, both in the processor hardware and in software. Figure 8.3 shows a typical sequence. When an I/O device completes an I/O operation, the following sequence of hardware events occurs:

1. The device issues an interrupt signal to the processor.
2. The processor finishes execution of the current instruction before responding to the interrupt.
3. The processor tests for an interrupt, determines that there is one, and sends an acknowledgment signal to the device that issued the interrupt. The acknowledgment allows the device to remove its interrupt signal.
4. The processor now needs to prepare to transfer control to the interrupt routine. To begin, it needs to save information needed to resume the current program at the point of interrupt. The minimum information required is (a) the status of the processor, which is contained in a register called the program status word (PSW), and (b) the location of the next instruction to be executed, which is contained in the program counter. These can be pushed onto the system control stack.
5. The processor now loads the program counter with the entry location of the interrupt-handling program that will respond to this interrupt. Depending on the computer architecture and operating system design, there may be a single program; one program for each type of interrupt; or one program for each device and each type of interrupt. If there is more than one interrupt-handling

routine, the processor must determine which one to invoke. This information may have been included in the original interrupt signal, or the processor may have to issue a request to the device that issued the interrupt to get a response that contains the needed information.

6. At this point, the program counter and PSW relating to the interrupted program have been saved on the system stack. However, there is other information that is considered part of the “state” of the executing program. In particular, the contents of the processor registers need to be saved, because these registers may be used by the interrupt handler. So, all of these values, plus any other state information, need to be saved. Typically, the interrupt handler will begin by saving the contents of all registers on the stack. Figure 8.3a shows a simple example. In this case, a user program is interrupted after the instruction at location N. The contents of all of the registers plus the address of the next instruction (N+ 1) are pushed onto the stack. The stack pointer is updated to point to the new top of stack, and the program counter is updated to point to the beginning of the interrupt service routine.
7. The interrupt handler next processes the interrupt. This includes an examination of status information relating to the I/O operation or other event that caused an interrupt. It may also involve sending additional commands or acknowledgments to the I/O device.
8. When interrupt processing is complete, the saved register values are retrieved from the stack and restored to the registers (e.g., see Figure 8.3b).
9. The final act is to restore the PSW and program counter values from the stack. As a result, the next instruction to be executed will be from the previously interrupted program.

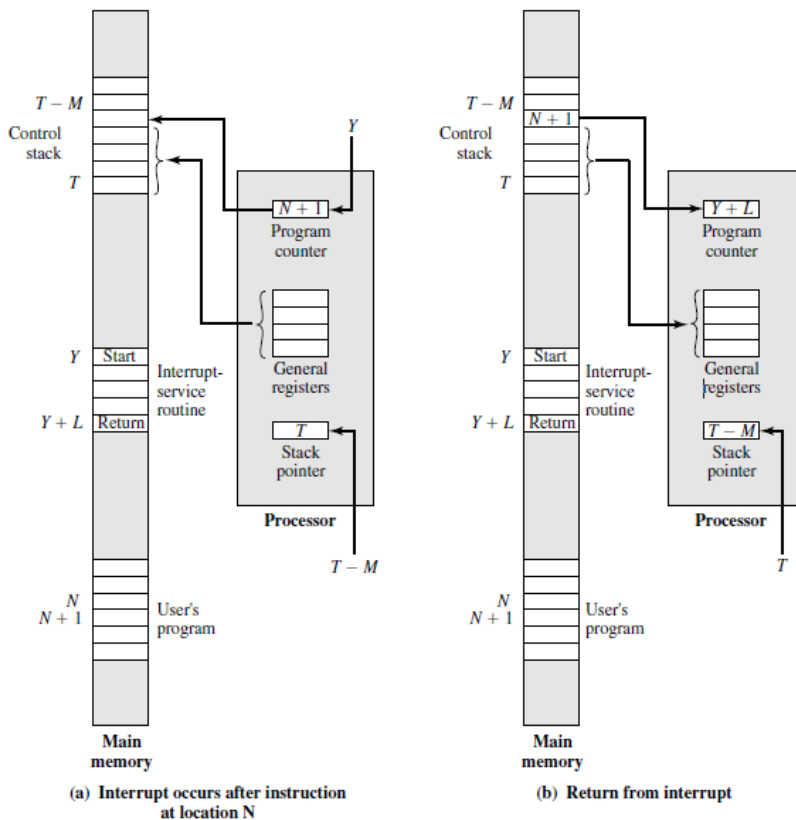


Figure 8.3 Changes in Memory and Registers for an Interrupt

8.4 Direct Memory Access(DMA)

Interrupt-driven I/O, though more efficient than simple programmed I/O, still requires the active intervention of the processor to transfer data between memory and an I/O module, and any data transfer must traverse a path through the processor. Thus, both these forms of I/O suffer from two inherent drawbacks:

1. The I/O transfer rate is limited by the speed with which the processor can test and service a device.

2. The processor is tied up in managing an I/O transfer; a number of instructions must be executed for each I/O transfer.

There is somewhat of a trade-off between these two drawbacks. Consider the transfer of a block of data. Using simple programmed I/O, the processor is dedicated to the task of I/O and can move data at a rather high rate, at the cost of doing nothing else. Interrupt I/O frees up the processor to some extent at the expense of the I/O transfer rate. Nevertheless, both methods have an adverse impact on both processor activity and I/O transfer rate.

When large volumes of data are to be moved, a more efficient technique is required: direct memory access (DMA).

DMA Function: DMA involves an additional module on the system bus. The DMA module (Figure 8.4) is capable of mimicking the processor and, indeed, of taking over control of the system from the processor. It needs to do this to transfer data to and from memory over the system bus. For this purpose, the DMA module must use the bus only when the processor does not need it, or it must force the processor to suspend operation temporarily. The latter technique is more common and is referred to as *cycle stealing*, because the DMA module in effect steals a bus cycle.

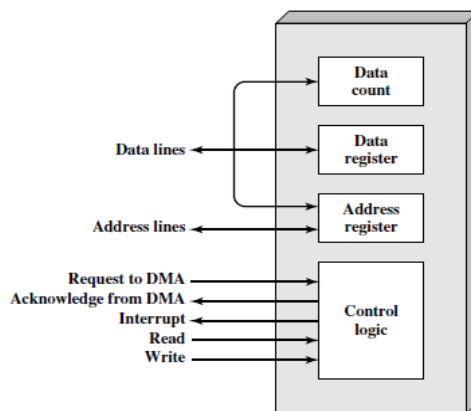


Figure 8.4 Typical DMA Block Diagram

When the processor wishes to read or write a block of data, it issues a command to the DMA module, by sending to the DMA module the following information:

- Whether a read or write is requested, using the read or write control line between the processor and the DMA module
- The address of the I/O device involved, communicated on the data lines
- The starting location in memory to read from or write to, communicated on the data lines and stored by the DMA module in its address register
- The number of words to be read or written, again communicated via the data lines and stored in the data count register

The processor then continues with other work. It has delegated this I/O operation to the DMA module. The DMA module transfers the entire block of data, one word at a time, directly to or from memory, without going through the processor. When the transfer is complete, the DMA module sends an interrupt signal to the processor. Thus, the processor is involved only at the beginning and end of the transfer. The processor is suspended just before it needs to use the bus. The DMA module then transfers one word and returns control to the processor. Note that this is not an interrupt; the processor does not save a context and do something else. Rather, the processor pauses for one bus cycle. The overall effect is to cause the processor to execute more slowly. Nevertheless, for a multiple-word I/O transfer, DMA is far more efficient than interrupt-driven or programmed I/O.

The DMA mechanism can be configured in a variety of ways. The DMA module, acting as a surrogate processor, uses programmed I/O to exchange data between memory and an I/O module through the DMA module. This configuration, while it

may be inexpensive, is clearly inefficient. As with processor-controlled programmed I/O, each transfer of a word consumes two bus cycles.

The number of required bus cycles can be cut substantially by integrating the DMA and I/O functions. The DMA logic may actually be a part of an I/O module, or it may be a separate module that controls one or more I/O modules. This concept can be taken one step further by connecting I/O modules to the DMA module using an I/O bus. This reduces the number of I/O interfaces in the DMA module to one and provides for an easily expandable configuration. In both of these cases, the system bus that the DMA module shares with the processor and memory is used by the DMA module only to exchange data with memory. The exchange of data between the DMA and I/O modules takes place off the system bus.

8.5 I/O Channels and Processors

As computer systems have evolved, there has been a pattern of increasing complexity and sophistication of individual components. Nowhere is this more evident than in the I/O function. We have already seen part of that evolution. The evolutionary steps can be summarized as follows:

1. The CPU directly controls a peripheral device. This is seen in simple microprocessor- controlled devices.
2. A controller or I/O module is added. The CPU uses programmed I/O without interrupts. With this step, the CPU becomes somewhat divorced from the specific details of external device interfaces.
3. The same configuration as in step 2 is used, but now interrupts are employed. The CPU need not spend time waiting for an I/O operation to be performed, thus increasing efficiency.

4. The I/O module is given direct access to memory via DMA. It can now move a block of data to or from memory without involving the CPU, except at the beginning and end of the transfer.
5. The I/O module is enhanced to become a processor in its own right, with a specialized instruction set tailored for I/O. The CPU directs the I/O processor to execute an I/O program in memory. The I/O processor fetches and executes these instructions without CPU intervention. This allows the CPU to specify a sequence of I/O activities and to be interrupted only when the entire sequence has been performed.
6. The I/O module has a local memory of its own and is, in fact, a computer in its own right. With this architecture, a large set of I/O devices can be controlled, with minimal CPU involvement. A common use for such architecture has been to control communication with interactive terminals. The I/O processor takes care of most of the tasks involved in controlling the terminals.

As one proceeds along this evolutionary path, more and more of the I/O function is performed without CPU involvement. The CPU is increasingly relieved of I/O-related tasks, improving performance. With the last two steps (5–6), a major change occurs with the introduction of the concept of an I/O module capable of executing a program. For step 5, the I/O module is often referred to as an *I/O channel*. For step 6, the term *I/O processor* is often used. However, both terms are on occasion applied to both situations. In what follows, we will use the term *I/O channel*.

Characteristics of I/O Channels: The I/O channel represents an extension of the DMA concept. An I/O channel has the ability to execute I/O instructions, which gives it complete control over I/O operations. In a computer system with such devices, the

CPU does not execute I/O instructions. Such instructions are stored in main memory to be executed by a special-purpose processor in the I/O channel itself. Thus, the CPU initiates an I/O transfer by instructing the I/O channel to execute a program in memory. The program will specify the device or devices, the area or areas of memory for storage, priority, and actions to be taken for certain error conditions. The I/O channel follows these instructions and controls the data transfer.

Two types of I/O channels are common, as illustrated in Figure 8.5. A *selector channel* controls multiple high-speed devices and, at any one time, is dedicated to the transfer of data with one of those devices. Thus, the I/O channel selects one device and affects the data transfer. Each device, or a small set of devices, is handled by a *controller*, or I/O module, that is much like the I/O modules we have been discussing. Thus, the I/O channel serves in place of the CPU in controlling these I/O controllers. A *multiplexor channel* can handle I/O with multiple devices at the same time. For low-speed devices, a *byte multiplexor* accepts or transmits characters as fast as possible to multiple devices. For example, the resultant character stream from three devices with different rates and individual streams A1A2A3A4 . . . , B1B2B3B4 . . . , and C1C2C3C4 . . . might be A1B1C1A2C2A3B2C3A4, and so on. For high-speed devices, a *block multiplexor* interleaves blocks of data from several devices.

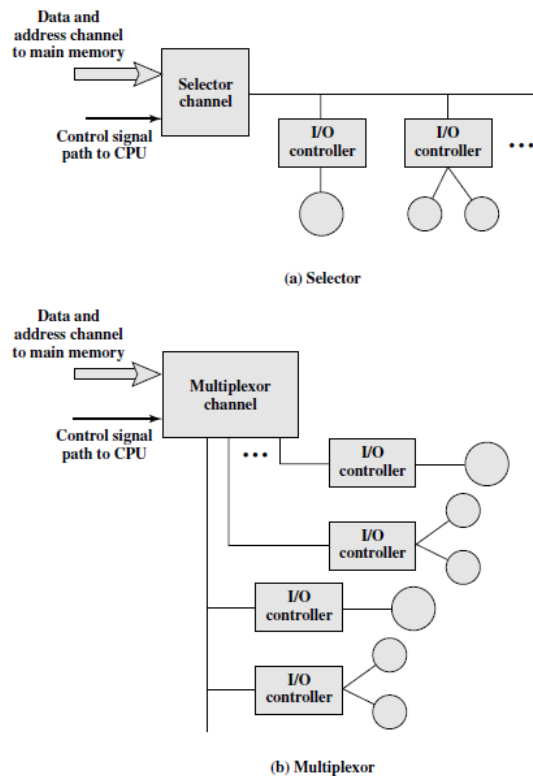


Figure 8.5 I/O Channel Architecture

Summary

The computer system's I/O architecture is its interface to the outside world. This architecture provides a systematic means of controlling interaction with the outside world and provides the operating system with the information it needs to manage I/O activity effectively. There are three principal I/O techniques: **programmed I/O**, in which I/O occurs under the direct and continuous control of the program requesting the I/O operation; **interrupt-driven I/O**, in which a program issues an I/O command and then continues to execute, until it is interrupted by the I/O hardware to signal the end of the I/O operation; and **direct memory access (DMA)**, in which a specialized

I/O processor takes over control of an I/O operation to move a large block of data. Two important examples of external I/O interfaces are **FireWire** and **Infiniband**.

Review Questions

- 1) Some processors use memory mapped I/O where I/O devices are in the same address space as main memory. Others have separate I/O address space and separate instructions. Give some advantages and disadvantages of each.
- 2) Although DMA does not use the CPU, the maximum transfer rate is still limited. Consider reading a block from the disk. Name three factors that might ultimately limit the rate transfer.
- 3) An I/O device transfers 10 MB/s of data into the memory of a processor over the I/O bus, which has a total data transfer capacity of 100 MB/s. The 10 MB/s of the data is transferred as 2500 independent pages of 4 KB each. If the processor operates at 200 MHz, it takes 1000 cycles to initiate a DMA transaction, and 1500 cycles to respond to the device's interrupt when the DMA transfer completes, what fraction of the processor's time is spent handling the data transfer with and without DMA?
- 4) If the normal processor memory read (RD) and write (WR) control outputs are connected to I/O interface adapters, what type of I/O technique is being used?

References

1. Blaauw, G., and Brooks, F. *Computer Architecture: Concepts and Evolution*. Reading, MA: Addison-Wesley, 1997.
2. Stallings, W. *Computer Architecture and Organizations Designing for Performance, Eighth Edition*. Upper Saddle River, NJ: Prentice Hall, 2010.

CHAPTER NINE: PIPELINE AND VECTOR PROCESSING

Objectives

After the completion of this chapter students will be able to:

- ⇒ Understand how multiple processes can be executed in parallel
- ⇒ Understand the organizations of multi-processors

Key Concepts

- The two most common multiple-processor organizations are **symmetric multiprocessors** (SMPs) and clusters
- An SMP consists of multiple similar processors within the same computer
- A special-purpose type of parallel organization is the vector facility

Introduction

Traditionally, the computer has been viewed as a sequential machine. Most computer programming languages require the programmer to specify algorithms as sequences of instructions. Processors execute programs by executing machine instructions in a sequence and one at a time. Each instruction is executed in a sequence of operations (fetch instruction, fetch operands, perform operation, store results).

This view of the computer has never been entirely true. At the micro-operation level, multiple control signals are generated at the same time. Instruction pipelining, at least to the extent of overlapping fetch and execute operations, has been around for a long time. Both of these are examples of performing functions in parallel. This approach is taken further with superscalar organization, which exploits instruction-level parallelism. With a superscalar machine, there are multiple execution units within a single processor, and these may execute multiple instructions from the same program in parallel.

As computer technology has evolved, and as the cost of computer hardware has dropped, computer designers have sought more and more opportunities for parallelism, usually to enhance performance and, in some cases, to increase availability. After an overview, this chapter looks at some of the most prominent approaches to parallel organization. First, we examine symmetric multiprocessors (SMPs), one of the earliest and still the most common example of parallel organization. Another approach to the use of multiple processors that we examine is that of non-uniform memory access (NUMA) machines. The NUMA approach is relatively new and not yet proven in the marketplace, but is often considered as an alternative to the SMP or cluster approach. Finally, this chapter looks at hardware organizational approaches to vector computation. These approaches optimize the ALU for processing vectors or arrays of floating-point numbers. They are common on the class of systems known as *supercomputers*.

9.1 Parallel Processing

A taxonomy first introduced by Flynn is still the most common way of categorizing systems with parallel processing capability. Flynn proposed the following categories of computer systems:

- **Single instruction, single data (SISD) stream:** A single processor executes a single instruction stream to operate on data stored in a single memory. Uniprocessors fall into this category.
- **Single instruction, multiple data (SIMD) stream:** A single machine instruction controls the simultaneous execution of a number of processing elements on a lockstep basis. Each processing element has an associated data memory, so that each instruction is executed on a different set of data by the different processors. Vector and array processors fall into this category.

- **Multiple instruction, single data (MISD) stream:** A sequence of data is transmitted to a set of processors, each of which executes a different instruction sequence. This structure is not commercially implemented.
- **Multiple instruction, multiple data (MIMD) stream:** A set of processors simultaneously execute different instruction sequences on different data sets. SMPs, clusters, and NUMA systems fit into this category.

With the MIMD organization, the processors are general purpose; each is able to process all of the instructions necessary to perform the appropriate data transformation. MIMDs can be further subdivided by the means in which the processors communicate (Figure 9.1). If the processors share a common memory, then each processor accesses programs and data stored in the shared memory, and processors communicate with each other via that memory. The most common form of such system is known as a **symmetric multiprocessor (SMP)**. In an SMP, multiple processors share a single memory or pool of memory by means of a shared bus or other interconnection mechanism; a distinguishing feature is that the memory access time to any region of memory is approximately the same for each processor. A more recent development is the **non-uniform memory access (NUMA)** organization. As the name suggests, the memory access time to different regions of memory may differ for a NUMA processor.

A collection of independent uni-processors or SMPs may be interconnected to form a **cluster**. Communication among the computers is either via fixed paths or via some network facility.

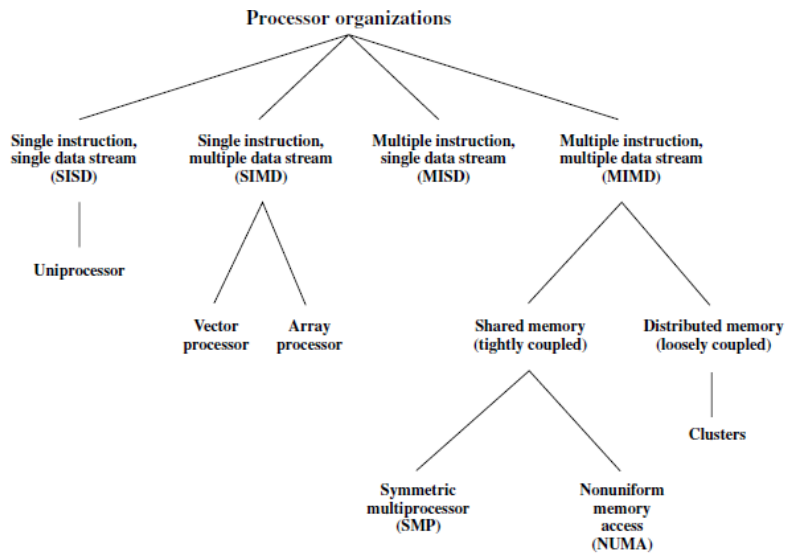


Figure 9.1 A Taxonomy of Parallel Processor Architectures

Until fairly recently, virtually all single-user personal computers and most workstations contained a single general-purpose microprocessor. As demands for performance increase and as the cost of microprocessors continues to drop, vendors have introduced systems with an SMP organization. The term *SMP* refers to a computer hardware architecture and also to the operating system behavior that reflects that architecture. An SMP can be defined as a standalone computer system with the following characteristics:

1. There are two or more similar processors of comparable capability.
2. These processors share the same main memory and I/O facilities and are interconnected by a bus or other internal connection scheme, such that memory access time is approximately the same for each processor.
3. All processors share access to I/O devices, either through the same channels or through different channels that provide paths to the same device.
4. All processors can perform the same functions (hence the term *symmetric*).

5. The system is controlled by an integrated operating system that provides interaction between processors and their programs at the job, task, file, and data element levels.

Points 1 to 4 should be self-explanatory. Point 5 illustrates one of the contrasts with a loosely coupled multiprocessing system, such as a cluster. In the latter, the physical unit of interaction is usually a message or complete file. In an SMP, individual data elements can constitute the level of interaction, and there can be a high degree of cooperation between processes.

The operating system of an SMP schedules processes or threads across all of the processors. An SMP organization has a number of potential advantages over a uni-processor organization, including the following:

Performance: If the work to be done by a computer can be organized so that some portions of the work can be done in parallel, then a system with multiple processors will yield greater performance than one with a single processor of the same type (Figure 9.2).

Availability: In a symmetric multiprocessor, because all processors can perform the same functions, the failure of a single processor does not halt the machine. Instead, the system can continue to function at reduced performance.

Incremental growth: A user can enhance the performance of a system by adding an additional processor.

Scaling: Vendors can offer a range of products with different price and performance characteristics based on the number of processors configured in the system.



Figure 9.2 Multiprogramming and Multiprocessing

It is important to note that these are potential, rather than guaranteed, benefits. The operating system must provide tools and functions to exploit the parallelism in an SMP system.

An attractive feature of an SMP is that the existence of multiple processors is transparent to the user. The operating system takes care of scheduling of threads or processes on individual processors and of synchronization among processors.

9.2 Vector Processing

Although the performance of mainframe general-purpose computers continues to improve relentlessly, there continue to be applications that are beyond the reach of the contemporary mainframe. There is a need for computers to solve mathematical problems of physical processes, such as occur in disciplines including aerodynamics, seismology, meteorology, and atomic, nuclear, and plasma physics.

Typically, these problems are characterized by the need for high precision and a program that repetitively performs floating-point arithmetic operations on large arrays of numbers. Most of these problems fall into the category known as *continuous-field simulation*. In essence, a physical situation can be described by a surface or region in three dimensions (e.g., the flow of air adjacent to the surface of a rocket). This surface is approximated by a grid of points. A set of differential equations defines the physical behavior of the surface at each point. The equations are represented as an array of values and coefficients, and the solution involves repeated arithmetic operations on the arrays of data.

Supercomputers were developed to handle these types of problems. These machines are typically capable of billions of floating-point operations per second. In contrast to mainframes, which are designed for multiprogramming and intensive I/O, the supercomputer is optimized for the type of numerical calculation just described.

The supercomputer has limited use and, because of its price tag, a limited market. Comparatively few of these machines are operational, mostly at research centers and some government agencies with scientific or engineering functions. As with other areas of computer technology, there is a constant demand to increase the performance of the supercomputer. Thus, the technology and performance of the supercomputer continues to evolve.

There is another type of system that has been designed to address the need for vector computation, referred to as the *array processor*. Although a supercomputer is optimized for vector computation, it is a general-purpose computer, capable of handling scalar processing and general data processing tasks. Array processors do not include scalar processing; they are configured as peripheral devices by both mainframe and minicomputer users to run the vectorized portions of programs.

Approaches to Vector Computation: The key to the design of a supercomputer or array processor is to recognize that the main task is to perform arithmetic operations on arrays or vectors of floating-point numbers. In a general-purpose computer, this will require iteration through each element of the array. For example, consider two vectors (one-dimensional arrays) of numbers, A and B. We would like to add these and place the result in C. In the example of Figure 9.3, this requires six separate additions. How could we speed up this computation? The answer is to introduce some form of parallelism.

Several approaches have been taken to achieving parallelism in vector computation. We illustrate this with an example. Consider the vector multiplication $C = A \times B$, where A, B, and C are $N \times N$ matrices. The formula for each element of C is $C_{i,j} = \sum_{k=0}^N a_{i,k} \times b_{k,j}$, where A, B, and C have elements $a_{i,j}$, $b_{i,j}$, and $c_{i,j}$, respectively.

$\begin{bmatrix} 1.5 \\ 7.1 \\ 6.9 \\ 100.5 \\ 0 \\ 59.7 \end{bmatrix}$	+	$\begin{bmatrix} 2.0 \\ 39.7 \\ 1000.003 \\ 11 \\ 21.1 \\ 19.7 \end{bmatrix}$	=	$\begin{bmatrix} 3.5 \\ 46.8 \\ 1006.093 \\ 111.5 \\ 21.1 \\ 79.4 \end{bmatrix}$
A	+	B	=	C

Figure 9.3 Example of Vector Addition

9.3 Pipelining

As computer systems evolve, greater performance can be achieved by taking advantage of improvements in technology, such as faster circuitry. In addition, organizational enhancements to the processor can improve performance. We have already seen some examples of this, such as the use of multiple registers rather than a

single accumulator, and the use of a cache memory. Another organizational approach, which is quite common, is instruction pipelining.

Pipelining Strategy: Instruction pipelining is similar to the use of an assembly line in a manufacturing plant. An assembly line takes advantage of the fact that a product goes through various stages of production. By laying the production process out in an assembly line, products at various stages can be worked on simultaneously. This process is also referred to as *pipelining*, because, as in a pipeline, new inputs are accepted at one end before previously accepted inputs appear as outputs at the other end.

To apply this concept to instruction execution, we must recognize that, in fact, an instruction has a number of stages. Clearly, there should be some opportunity for pipelining.

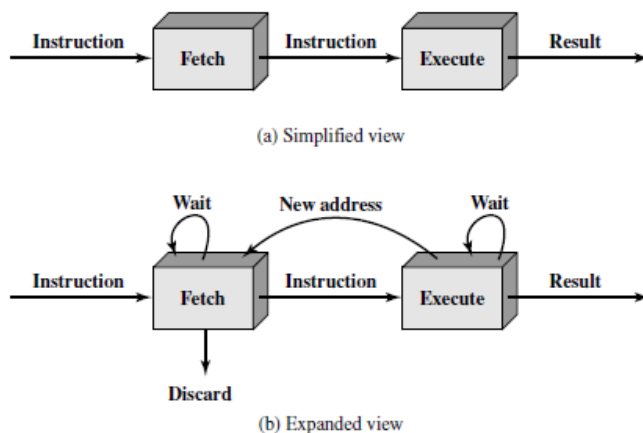


Figure 9.4 Two-Stage Instruction Pipeline

As a simple approach, consider subdividing instruction processing into two stages: fetch instruction and execute instruction. There are times during the execution of an instruction when main memory is not being accessed. This time could be used to fetch

the next instruction in parallel with the execution of the current one. Figure 9.4a depicts this approach. The pipeline has two independent stages. The first stage fetches an instruction and buffers it. When the second stage is free, the first stage passes it the buffered instruction. While the second stage is executing the instruction, the first stage takes advantage of any unused memory cycles to fetch and buffer the next instruction. This is called *instruction pre-fetch* or *fetches overlap*. Note that this approach, which involves instruction buffering, requires more registers. In general, pipelining requires registers to store data between stages.

It should be clear that this process will speed up instruction execution. If the fetch and execute stages were of equal duration, the instruction cycle time would be halved. However, if we look more closely at this pipeline (Figure 9.4b), we will see that this doubling of execution rate is unlikely for two reasons:

1. The execution time will generally be longer than the fetch time. Execution will involve reading and storing operands and the performance of some operation. Thus, the fetch stage may have to wait for some time before it can empty its buffer.
2. A conditional branch instruction makes the address of the next instruction to be fetched unknown. Thus, the fetch stage must wait until it receives the next instruction address from the execute stage. The execute stage may then have to wait while the next instruction is fetched.

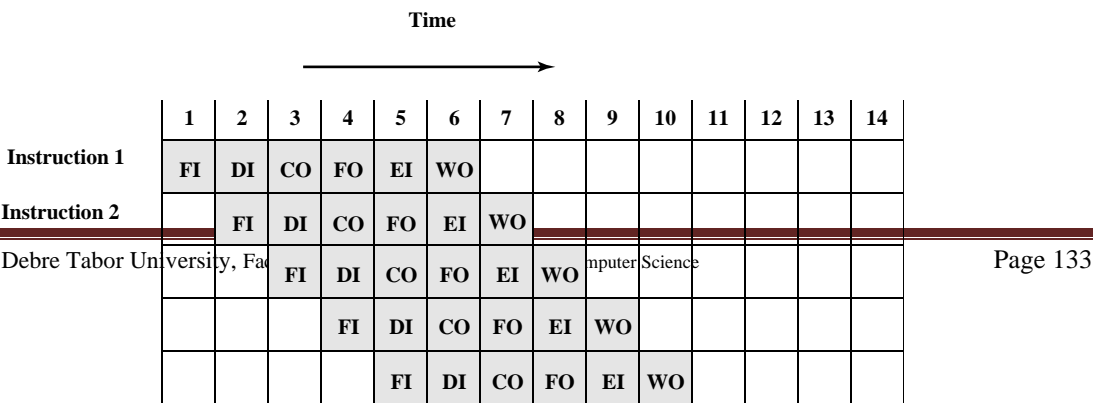
Guessing can reduce the time loss from the second reason. A simple rule is the following: When a conditional branch instruction is passed on from the fetch to the execute stage, the fetch stage fetches the next instruction in memory after the branch instruction. Then, if the branch is not taken, no time is lost. If the branch is taken, the fetched instruction must be discarded and a new instruction fetched.

While these factors reduce the potential effectiveness of the two-stage pipeline, some speedup occurs. To gain further speedup, the pipeline must have more stages. Let us consider the following decomposition of the instruction processing.

- **Fetch instruction (FI):** Read the next expected instruction into a buffer.
- **Decode instruction (DI):** Determine the opcode and the operand specifiers.
- **Calculate operands (CO):** Calculate the effective address of each source operand. This may involve displacement, register indirect, indirect, or other forms of address calculation.
- **Fetch operands (FO):** Fetch each operand from memory. Operands in registers need not be fetched.
- **Execute instruction (EI):** Perform the indicated operation and store the result, if any, in the specified destination operand location.
- **Write operand (WO):** Store the result in memory.

With this decomposition, the various stages will be of more nearly equal duration. the sake of illustration, let us assume equal duration. Using this assumption, Figure 12.10 shows that a six-stage pipeline can reduce the execution time for 9 instructions from 54 time units to 14 time units.

Several comments are in order: The diagram assumes that each instruction goes through all six stages of the pipeline. This will not always be the case. For example, a load instruction does not need the WO stage. However, to simplify the pipeline hardware, the timing is set up assuming that each instruction requires all six stages. Also, the diagram assumes that all of the stages can be performed in parallel. In particular, it is assumed that there are no memory conflicts. For example, the FI,



Instruction 5

Instruction 6

Instruction 7

Instruction 8

Instruction 9

Figure Timing diagram for instruction pipelining operation

FO, and WO stages involve a memory access. The diagram implies that all these accesses can occur simultaneously. Most memory systems will not permit that. However, the desired value may be in cache, or the FO or WO stage may be null. Thus, much of the time, memory conflicts will not slow down the pipeline.

Several other factors serve to limit the performance enhancement. If the six stages are not of equal duration, there will be some waiting involved at various pipeline stages, as discussed before for the two-stage pipeline. Another difficulty is the conditional branch instruction, which can invalidate several instruction fetches. A similar unpredictable event is an interrupt. Figure 12.11 illustrates the effects of the conditional branch, using the same program as Figure 12.10. Assume that instruction 3 is a conditional branch to instruction 15. Until the instruction is executed, there is no way of knowing which instruction will come next. The pipeline, in this example, simply loads the next instruction in sequence (instruction 4) and proceeds. In Figure 12.10, the branch is not taken, and we get the full performance benefit of the enhancement. In Figure 12.11, the branch is taken. This is not determined until the end of time unit 7. At this point, the pipeline must be cleared of instructions that are not useful. During time unit 8, instruction 15 enters the pipeline. No instructions complete during time units 9 through 12; this is the performance penalty incurred because we could not anticipate the branch. Figure 12.12 indicates the logic needed for pipelining to account for branches and interrupts.

Other problems arise that did not appear in our simple two-stage organization. The CO stage may depend on the contents of a register that could be altered by a previous instruction that is still in the pipeline. Other such register and memory conflicts could occur. The system must contain logic to account for this type of conflict. To clarify pipeline operation, it might be useful to look at an alternative depiction. Figures 12.10 and 12.11 show the progression of time horizontally across the

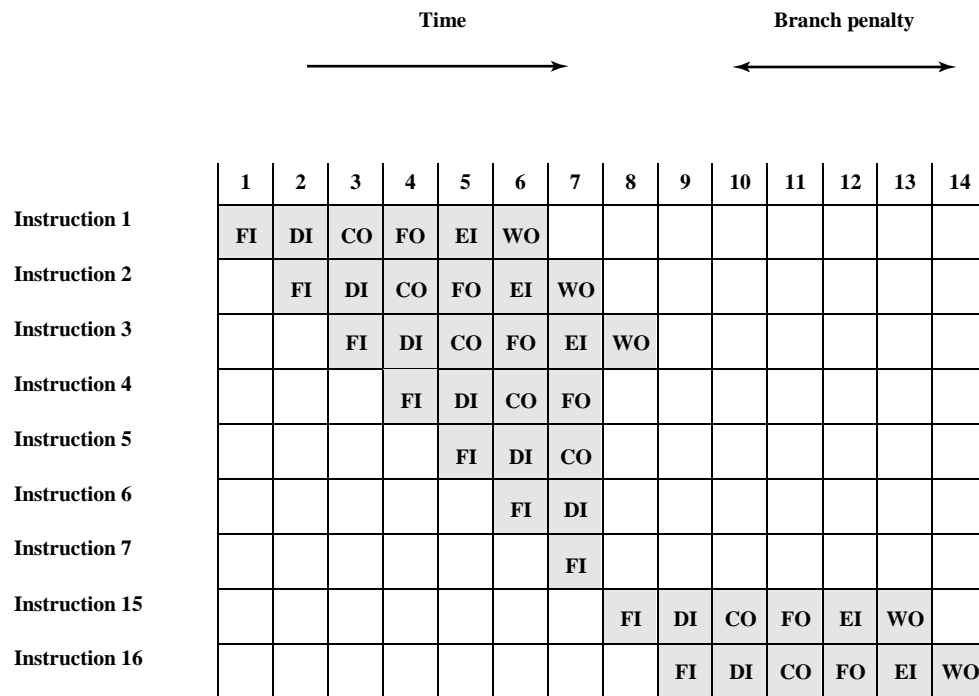
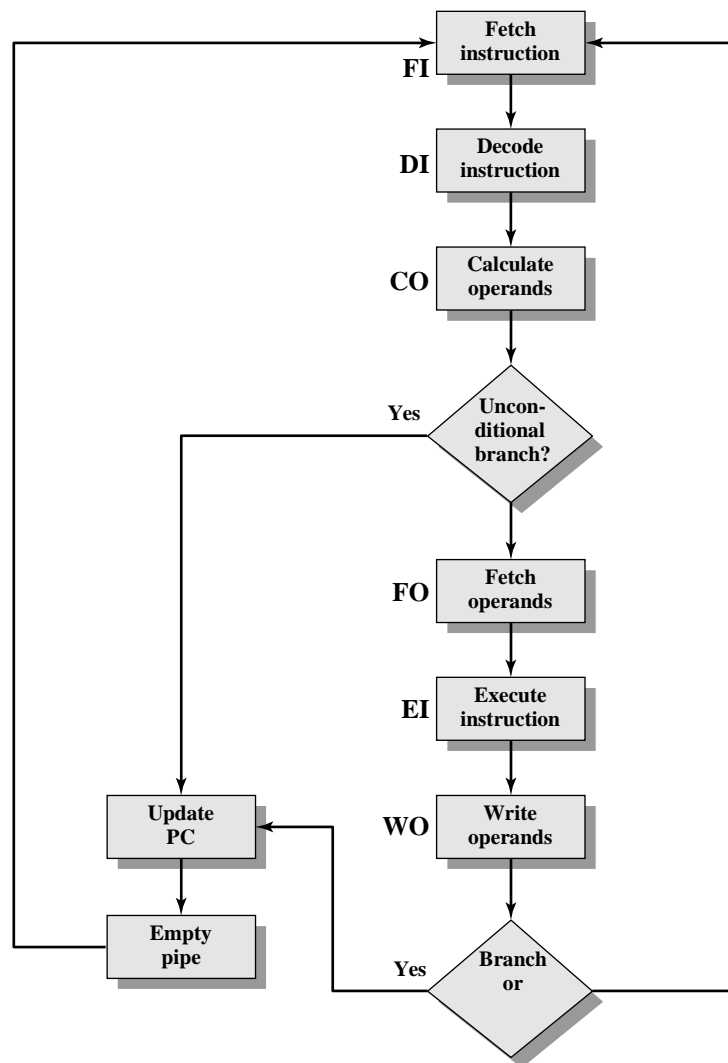


Figure 12.11 The Effect of a Conditional Branch on Instruction Pipeline Operation



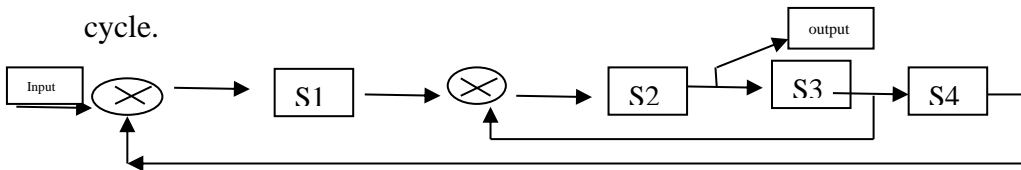
Figures, with each row showing the progress of an individual instruction.

Summary

A traditional way to increase system performance is to use multiple processors that can execute in parallel to support a given work-load. The two most common multiple-processor organizations are symmetric multiprocessors (SMPs) and clusters. More recently, non-uniform memory access (NUMA) systems have been introduced commercially. An SMP consists of multiple similar processors within the same computer, interconnected by a bus or some sort of switching arrangement. The most critical problem to address in an SMP is that of cache coherence. Each processor has its own cache and so it is possible for a given line of data to be present in more than one cache. If such a line is altered in one cache, then both main memory and the other cache have an invalid version of that line. Cache coherence protocols are designed to cope with this problem. A special-purpose type of parallel organization is the vector facility, which is tailored to the processing of vectors or arrays of data.

Review Questions & Problems

- 1) Vectorizing compilers generally detect loops that can be executed on a pipelined vector computer. Are the vectorization algorithms used by vectorizing compilers suitable for MIMD machine parallelization? Justify your answer.
- 2) Consider the following pipelined processor with four stages. This pipeline has a total evaluation time of six clock cycles. All successor stages must be used after each clock cycle.



- a) Specify the reservation table for this pipeline with six columns and four rows.
- b) List the set of forbidden latencies between task initiations.

- c) List all greedy cycles from the state diagram.
- d) Determine the minimal average latency (MAL).

References

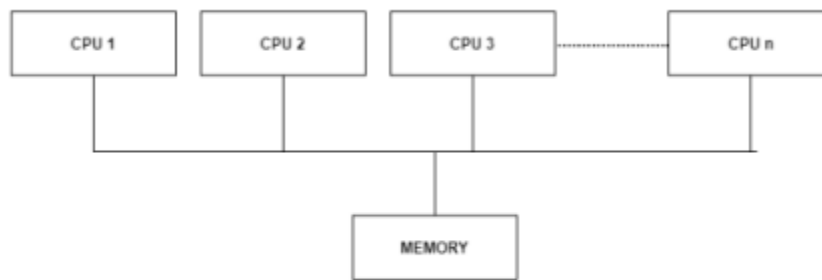
1. Bohr, M. "Silicon Trends and Limits for Advanced Microprocessors." *Communications of the ACM*, March 1998.
2. Stallings, W. *Computer Architecture and Organizations Designing for Performance, Eighth Edition*. Upper Saddle River, NJ: Prentice Hall, 2010.

Chapter 10: multiprocessors

Contents: Multiprocessors: Characteristics of Multiprocessor, Structure of Multiprocessor, Inter processor Arbitration, Inter-Processor Communication and Synchronization. Memory in Multiprocessor System

10.1 Multiprocessor and its characteristics

Most computer systems are single processor systems i.e they only have one processor. However, multiprocessor or parallel systems are increasing in importance nowadays. These systems have multiple processors working in parallel that share the computer clock, memory, bus, peripheral devices etc. An image demonstrating the multiprocessor architecture is:



Multiprocessing Architecture

Types of multiprocessors

There are mainly two types of multiprocessors i.e. symmetric and asymmetric multiprocessors. Details about them are as follows: **Symmetric Multiprocessors** In these types of systems, each processor contains a similar copy of the operating system and they all communicate with each other. All the processors are in a peer-to-peer relationship i.e. no master - slave relationship exists between them. An example of the symmetric multiprocessing system is the Encore version of Unix for the Multimax Computer. **Asymmetric Multiprocessors** In asymmetric systems, each processor is given a predefined task. There is a master processor that gives instruction to all the other processors. Asymmetric multiprocessor system contains a master slave relationship. Asymmetric

multiprocessor was the only type of multiprocessor available before symmetric multiprocessors were created. Now also, this is the cheaper option.

Advantages of Multiprocessor Systems There are multiple advantages to multiprocessor systems. Some of these are:

- More reliable Systems** In a multiprocessor system, even if one processor fails, the system will not halt. This ability to continue working despite hardware failure is known as graceful degradation. For example: If there are 5 processors in a multiprocessor system and one of them fails, then also 4 processors are still working. So the system only becomes slower and does not ground to a halt.
- Enhanced Throughput** If multiple processors are working in tandem, then the throughput of the system increases i.e. number of processes getting executed per unit of time increase. If there are N processors then the throughput increases by an amount just under N .
- More Economic Systems** Multiprocessor systems are cheaper than single processor systems in the long run because they share the data storage, peripheral devices, power supplies etc. If there are multiple processes that share data, it is better to schedule them on multiprocessor systems with shared data than have different computer systems with multiple copies of the data.

Disadvantages of Multiprocessor Systems There are some disadvantages as well to multiprocessor systems. Some of these are:

- Increased Expense** Even though multiprocessor systems are cheaper in the long run than using multiple computer systems, still they are quite expensive. It is much cheaper to buy a simple single processor system than a multiprocessor system.
- Complicated Operating System Required** There are multiple processors in a multiprocessor system that share peripherals, memory etc. So, it is much more complicated to schedule processes and impart resources to processes. Than in single processor systems. Hence, a more complex and complicated operating system is required in multiprocessor systems.

Large Main Memory Required All the processors in the multiprocessor system share the memory. So a much larger pool of memory is required as compared to single processor systems.

Characteristics of multiprocessors

1. A multiprocessor system is an interconnection of two or more CPUs with memory and input-output equipment.
2. The term “processor” in multiprocessor can mean either a central processing unit (CPU) or an input-output processor (IOP).
3. Multiprocessors are classified as multiple instruction stream, multiple data stream (MIMD) systems
4. The similarity and distinction between multiprocessor and multicomputer are ``
 - Similarity
 - Both support concurrent operations
 - Distinction
 - 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.
5. Multiprocessing improves the reliability of the system.
6. The benefit derived from a multiprocessor organization is an improved system performance.
 - Multiple independent jobs can be made to operate in parallel.
 - A single job can be partitioned into multiple parallel tasks.
6. Multiprocessing can improve performance by decomposing a program into parallel executable tasks.
 - 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.

- The other is to provide a compiler with multiprocessor software that can automatically detect parallelism in a user's program.
7. Multiprocessor 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
 - Tolerate a higher degree of interaction between tasks. Each processor element with its own private local memory is classified as a distributed-memory or loosely coupled system.
 - Are most efficient when the interaction between tasks is minimal

10.2 Interconnection structures for multiprocessor

interconnection Structures

1. The components that form a multiprocessor system are CPUs, IOPs connected to input-output devices, and a memory unit.
 2. The interconnection between the components can have different physical configurations, depending on the number of transfer paths that are available
 - o Between the processors and memory in a shared memory system
 - o Among the processing elements in a loosely coupled system
 3. There are several physical forms available for establishing an interconnection network.
 - o Time-shared common bus
 - o Multiport memory
 - o Crossbar switch
 - o Multistage switching network
 - o Hypercube system
- Time Shared Common Bus
1. A common-bus multiprocessor system consists of a number of processors connected through a common path to a memory unit.
 2. Disadvantage:
 - o Only one processor can communicate with the memory or another processor at any given time.
 - o As a consequence, the total overall transfer rate within the system is limited by the speed of the single path
 3. A more economical implementation of a dual bus structure is depicted in Fig. below.
 4. Part of the local memory may be designed as a cache memory attached to the CPU.

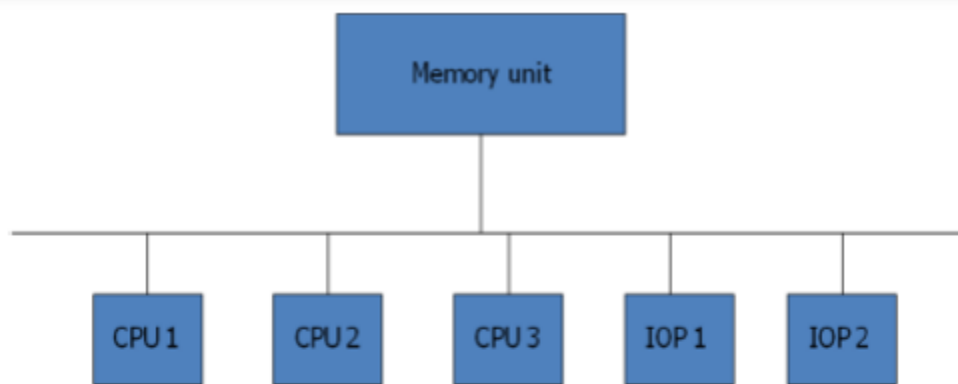


Fig: Time shared common bus organization

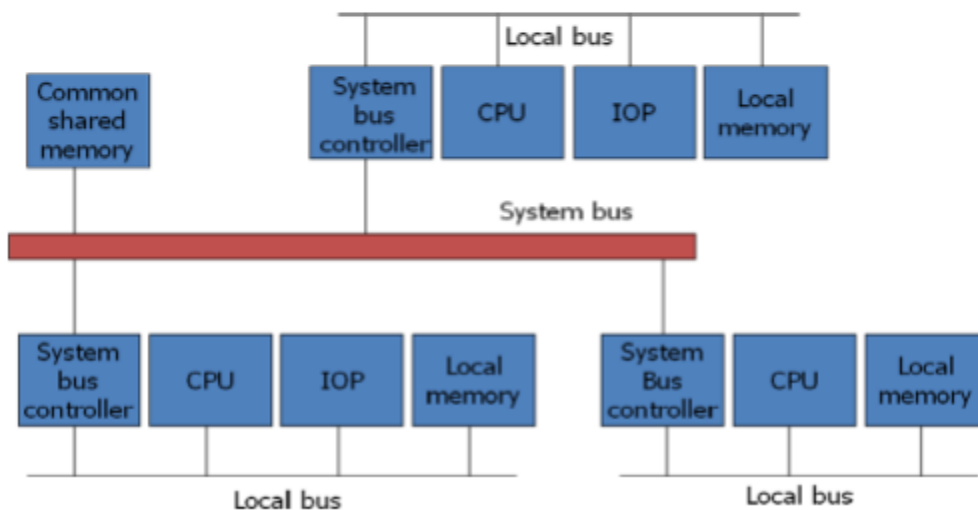


Fig: System bus structure for multiprocessors

Multiport Memory

1. A multiport memory system employs separate buses between each memory module and each CPU.
2. The module must have internal control logic to determine which port will have access to memory at any given time.
3. Memory access conflicts are resolved by assigning fixed priorities to each memory port.
4. Adv.: o The high transfer rate can be achieved because of the multiple paths.

5. Disadv.:

- o It requires expensive memory control logic and a large number of cables and connections

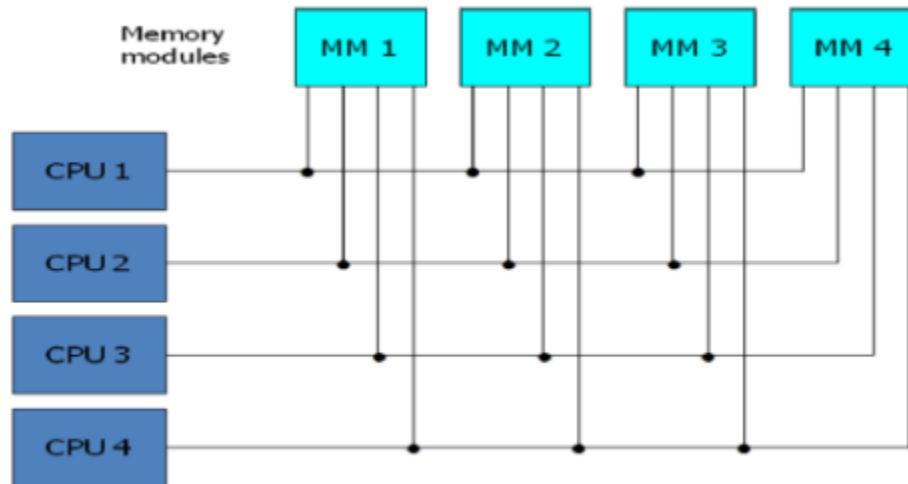


Fig: Multiport memory organization

Crossbar Switch

1. Consists of a number of cross points that are placed at intersections between processor buses and memory module paths.
2. The small square in each cross point is a switch that determines the path from a processor to a memory module.
3. Adv.:
 - o Supports simultaneous transfers from all memory modules
4. Disadv.:
 - o The hardware required to implement the switch can become quite large and complex.
5. Below fig. shows the functional design of a crossbar switch connected to one memory module.

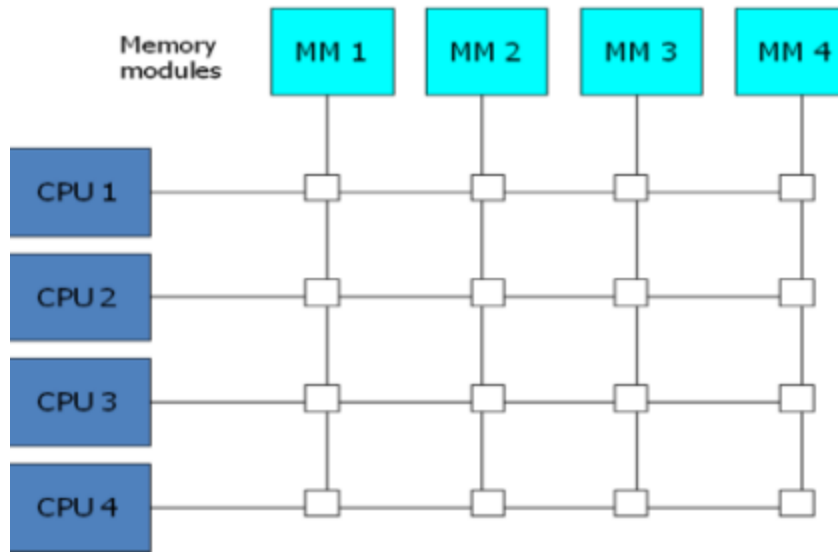


Figure: Crossbar switch

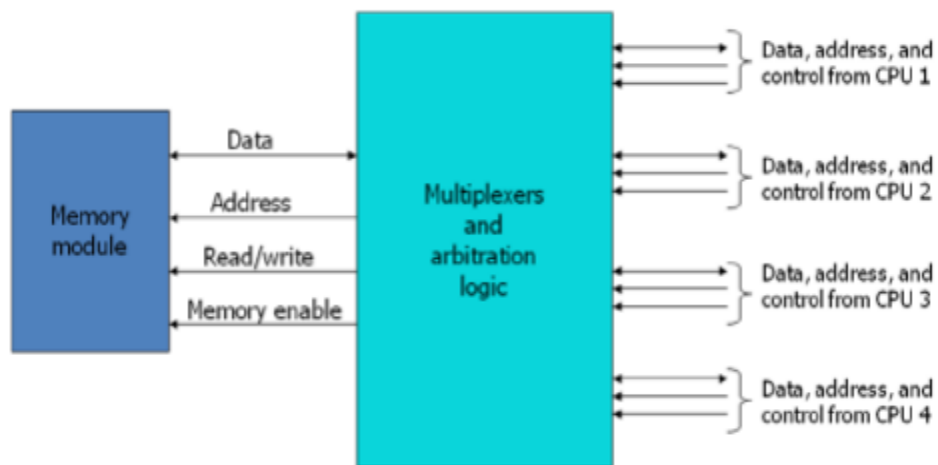


Fig: Block diagram of crossbar switch

10.3 Inter processors communication and synchronization

1. The various processors in a multiprocessor system must be provided with a facility for communicating with each other. o A communication path can be established through a portion of memory or a common input-output channels.
2. The sending processor structures a request, a message, or a procedure, and places it in the memory mailbox. o Status bits residing in common memory o The receiving processor can check the mailbox periodically. o The response time of this procedure can be time consuming.
3. A more efficient procedure is for the sending processor to alert the receiving processor directly by means of an interrupt signal.
4. In addition to shared memory, a multiprocessor system may have other shared resources. e.g., a magnetic disk storage unit.
5. To prevent conflicting use of shared resources by several processors there must be a provision for assigning resources to processors. i.e., operating system.
6. There are three organizations that have been used in the design of operating system for multiprocessors: master-slave configuration, separate operating system, and distributed operating system.
7. In a master-slave mode, one processor, master, always executes the operating system functions.
8. In the separate operating system organization, each processor can execute the operating system routines it needs. This organization is more suitable for loosely coupled systems.
9. In the distributed operating system organization, the operating system routines are distributed among the available processors. However, each particular operating system function is assigned to only one processor at a time. It is also referred to as a floating operating system.

Loosely Coupled System

1. There is no shared memory for passing information.
2. The communication between processors is by means of message passing through I/O channels.
3. The communication is initiated by one processor calling a procedure that resides in the memory of the processor with which it wishes to communicate.
4. The communication efficiency of the inter processor network depends on the communication routing protocol, processor speed, data link speed, and the topology of the network.

Inter process Synchronization

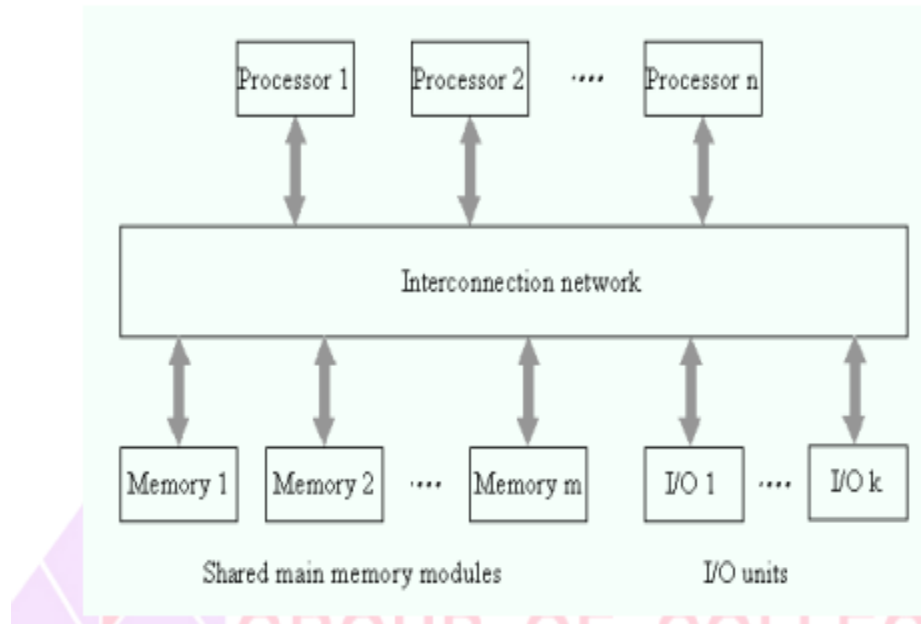
1. The instruction set of a multiprocessor contains basic instructions that are used to implement communication and synchronization between cooperating processes.
 - o Communication refers to the exchange of data between different processes.
 - o Synchronization refers to the special case where the data used to communicate between processors is control information.
2. Synchronization is needed to enforce the correct sequence of processes and to ensure mutually exclusive access to shared writable data.
3. Multiprocessor systems usually include various mechanisms to deal with the synchronization of resources.
 - o Low-level primitives are implemented directly by the hardware.
 - o These primitives are the basic mechanisms that enforce mutual exclusion for more complex mechanisms implemented in software.
 - o A number of hardware mechanisms for mutual exclusion have been developed.
4. A binary semaphore

Mutual Exclusion with Semaphore

1. A properly functioning multiprocessor system must provide a mechanism that will guarantee orderly access to shared memory and other shared resources.
 - o Mutual exclusion: This is necessary to protect data from being changed simultaneously by two or more processors.
 - o Critical section: is a program sequence that must complete execution before another processor accesses the same shared resource.
2. A binary variable called a semaphore is often used to indicate whether or not a processor is executing a critical section.
3. Testing and setting the semaphore is itself a critical operation and must be performed as a single indivisible operation.
4. A semaphore can be initialized by means of a test and set instruction in conjunction with a hardware lock mechanism.
5. The instruction TSL SEM will be executed in two memory cycles (the first to read and the second to write) as follows:
6. Note that the lock signal must be active during the execution of the test-and-set instruction.

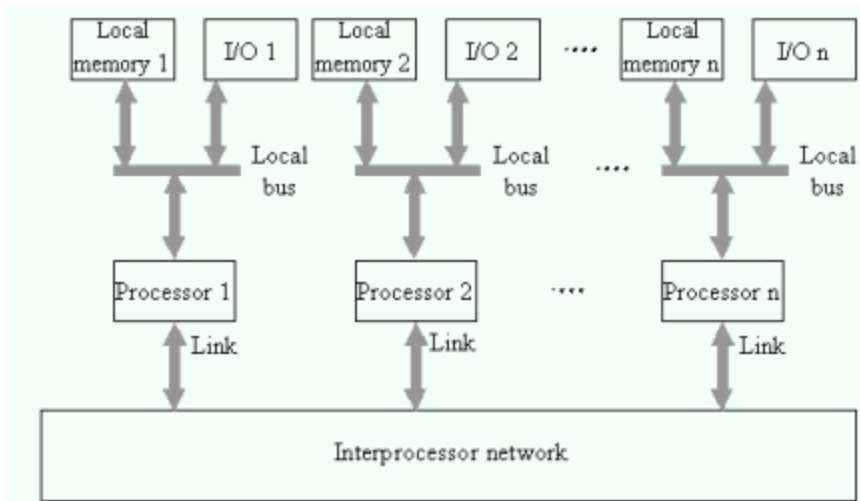
Memory in multiprocessor Systems

Data and code in a parallel program are stored in the main memory accessible for processors of the executive system. Regarding the way in which the main memory is used by processors in a multiprocessor system, we divide parallel systems onto shared memory system and distributed memory systems. See the figures below.



A multiprocessor system with shared memory (tightly coupled system)

In a shared memory system, all processors can access all the main memory address space. Fragments of the address space are usually located in separate memory modules, which are supplied with separate address decoders. Communication between processors (program code fragments) is done by means of shared variables access in the main memory. It is called communication through shared variables. Fetching instructions for execution in processors is also done from a shared memory. The efficiency of accessing memory modules depends on the structure and properties of the interconnection network. This network is a factor, which limitates the memory access throughput for a larger number of processors. It sets a limit on the number of processors in such systems, with which good efficiency of a parallel system is achieved. Multiprocessor systems with shared memory are called tightly coupled systems or multiprocessors. Due to symmetric access of all processors to all memory modules, the computations in such systems are called Symmetric Multiprocessing - SMP.



A multiprocessor system with a distributed memory (loosely coupled system)

In a distributed memory multiprocessor system, each processor has its local memory with the address space available only for this processor. Processors can exchange data through the interconnection network by means of communication through the **message passing**.

The instructions "send message" and "receive message" are used in programs for this purpose. The communication instructions send or receive messages with the use of identifiers of special elements (variables) are called **communication channels**.

The channels represent the use of connections that exist permanently (or are created in the interconnection network) between processors. There exist processors that are specially adapted for sending and receiving messages by the existence of communication links. Communication links can be serial or parallel. The number of communication links in such processors is from 4 to 6 (ex. transputer - 4 serial links, SHARC - a DSP (Data Signal Processor) from Analog Devices - 6 parallel links). Each link is supervised by an independent processor controller that organizes external data transmissions over the link. When a message is sent, it is fetched from the processor main memory. A message received from a link is next sent to the main memory. Multiprocessor systems that have distributed memory are called in the literature **loosely coupled systems**. In such

systems, it is possible to organize many inter-processor connections at the same time. It provides high communication efficiency and, as a consequence, high efficiency of parallel computations in processors (due to distribution of memory accesses), which gives rise to calling computations in such systems the **Massively Parallel Processing - MPP**.

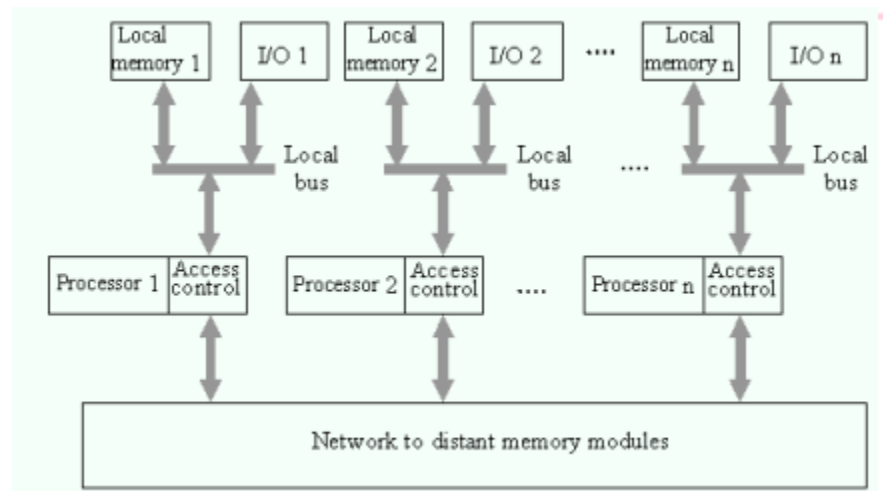
Communication by message passing in such systems can be executed according to the synchronous or **asynchronous communication model**.

In the synchronous communication model, the partner processes (programs) - the sending and the receiving one, get synchronized on communication instructions in a given channel. It means that the sending process can start transmitting data only if the receiving process in another processor has reached execution of the receive instruction in the same channel as the sending one. Since the communication is performed with the use of send and receive instructions in both processors simultaneously, there is no need of buffering of messages, and so, they are sent as if they were sent directly from the main memory of one processor to the memory of the other one. All this is done under supervision of link controllers in both processors.

With the asynchronous communication model, the sending and receiving processes (programs) do not synchronize communication execution in the involved channels. A message is sent to a channel at any time and it is directed to the buffer for messages in a given channel in the controller at the other side of the interconnection between the processors. The receiving process reads the message from the buffer of the given channel at any convenient time.

The third type of multiprocessor systems are systems with the distributed shared memory called also the virtual shared memory. In such systems, which currently show strong development, each processor has a local main memory. However, each memory is placed in a common address space of the entire system. It means that each processor can have access to the local memory of any other processor. In this type of the system, communication between processors is done by accessing shared variables. It involves execution of a simple read or write instruction concerning the shared variables in the memory of another processor. In each processor, a memory interface unit

examinees addresses used in current processor memory access instructions. As a result, it directs instruction execution to the local main memory bus or it sends the address together with the operation code to the local memory interface of another processor. Sending the address and later the data is performed through the network that connects all processors (their local main memory interface units).



A multiprocessor system with the distributed shared memory This type of the system is called in the literature the system with the Scalable Parallel Processing - SPP, since with the increase of the number of processors in the system, used to perform a given parallel program, the parallel program speedup in the system increases proportionally to the increase of the number of processors. Such a property is called parallel system scalability.

Teaching materials

Textbooks

Stallings, William. *Computer organization and architecture: designing for performance*. Pearson Education India,

Reference books

Tanenbaum, Andrew S. *Structured computer organization*. Pearson Education India,