

# UNIT I

## Digital Computers

### Introduction

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

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

In contrast to the common decimal numbers that employ the base 10 system, binary numbers use a base 2 system with two digits: 0 and 1. The decimal equivalent of a binary number can be found by expanding it into a power series with a base of 2. For example, the binary number 1001011 represents a quantity that can be converted to a decimal number by multiplying each bit by the base 2 raised to an integer power as follows:

$$1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 75$$

The seven bits 1001011 represent a binary number whose decimal equivalent is 75. However, this same group of seven bits represents the letter K when used in conjunction with a binary code for the letters of the alphabet. It may also represent a control code for specifying some decision logic in a particular digital computer. In other words, groups of bits in a digital computer are used to represent many different things. This is similar to the concept that the same letters of an alphabet are used to construct different languages, such as English and French.

A computer system is sometimes subdivided into two functional entities: hardware and software. The hardware of the computer consists of all the electronic components and electromechanical devices that comprise the physical entity of the device. Computer software consists of the instructions and data that the computer manipulates to perform various data-processing tasks.

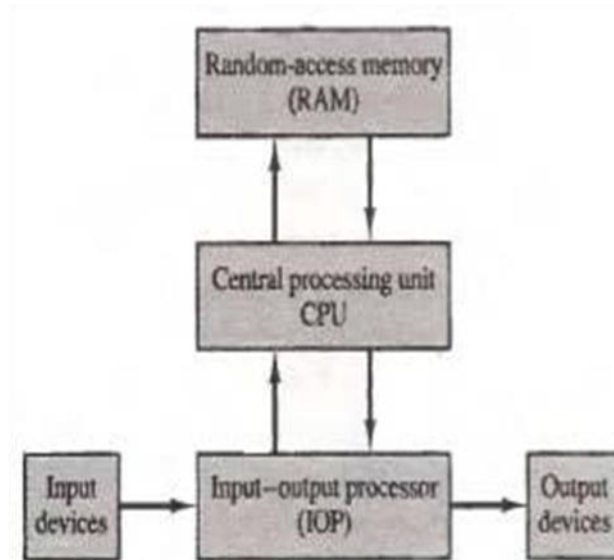
### **Program**

A sequence of instructions for the computer is called a **program**. The data that are manipulated by the program constitute the **data base**. A computer system is composed of its hardware and the system software available for its use. The system software of a computer consists of a collection of programs whose purpose is to make more effective use of the computer.

The programs included in a systems software package are referred to as the **operating system**. They are distinguished from application programs written by the user for the purpose of solving particular problems. For example, a high-level language program written by a user to solve particular data-processing needs is an application program, but the compiler that translates the high-level language program to machine language is a system program.

## computer hardware

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



The hardware of the computer is usually divided into three major parts, as shown in Fig. The central processing unit (CPU) contains an arithmetic and logic unit for manipulating data, a number of registers for storing data, and control circuits for fetching and executing instructions. The memory of a computer contains storage for instructions and data. It is called a random access memory (RAM) because the CPU can access any location in memory at random and retrieve the binary information within a fixed interval of time. The input and output processor (IOP) contains electronic circuits for communicating and controlling the transfer of information between the computer and the outside world. The input and output devices connected to the computer include keyboards, printers, terminals, magnetic disk drives, and other communication devices.

### Computer Organization

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**

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**

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.

Two basic types of computer architecture are von Neumann architecture and Harvard architecture. Von Neumann architecture describes the framework, or structure, that a computer's hardware, programming, and data should follow.

Von Neumann envisioned the structure of a computer system as being composed of the following components

- 1) The central arithmetic unit, which today is called the arithmetic logic unit (ALU). This unit performs the computer's computational and logical functions.
- 2) Memory: more specifically the computer's main or fast, memory such as random access memory (RAM).
- 3) A control unit that directs other components of the computer to perform certain actions, such as directing the fetching of data or instructions from memory to be processed by the ALU.
- 4) Man machine interfaces; i.e., input output devices such as a keyboard for input and display monitor for output.



The instructions used to manipulate that data, should be stored together in the same memory area of the computer and instructions are carried out sequentially, one instruction at a time. The sequential execution of programming imposes a sort of speed limit on program execution, since only one instruction at a time can be handled by the computer's processor. It means that the CPU can be either reading an instruction or so reading /writing data from /to the memory. Both cannot occur at the same time since the instructions and data use the same signal pathways and memory.

The Harvard architecture uses the physically separate storage and signal pathways for their instructions and data. In a computer with Harvard architecture the CPU can read both an instruction and data from memory at the same time.

An example of computer architecture based on the von Neumann architecture is the desktop personal computer. Micro controller based computer system and DSP (digital system processor) based computer system are examples for Harvard architecture.

## REGISTER TRANSFER AND MICROOPERATIONS

- ✓ Register Transfer Language
- ✓ Register Transfer
- ✓ Bus And Memory Transfers
- ✓ Types of Micro-operations
- ✓ Arithmetic Micro-operations
- ✓ Logic Micro-operations
- ✓ Shift Micro-operations
- ✓ Arithmetic Logic Shift Unit

### BASIC DEFINITIONS:

- A digital system is an interconnection of digital hardware modules.
- The modules are registers, decoders, arithmetic elements, and control logic.
- The various modules are interconnected with common data and control paths to form a digital computer system.
- Digital modules are best defined by the registers they contain and the operations that are performed on the data stored in them.
- The operations executed on data stored in registers are called *microoperations*.
- A *microoperation* is an elementary operation performed on the information stored in one or more registers.
- The result of the operation may replace the previous binary information of a register or may be transferred to another register.
- Examples of microoperations are shift, count, clear, and load.
- The internal hardware organization of a digital computer is best defined by specifying:
  1. The set of registers it contains and their function.
  2. The sequence of microoperations performed on the binary information stored in the registers.
  3. The control that initiates the sequence of microoperations.

### REGISTER TRANSFER LANGUAGE:

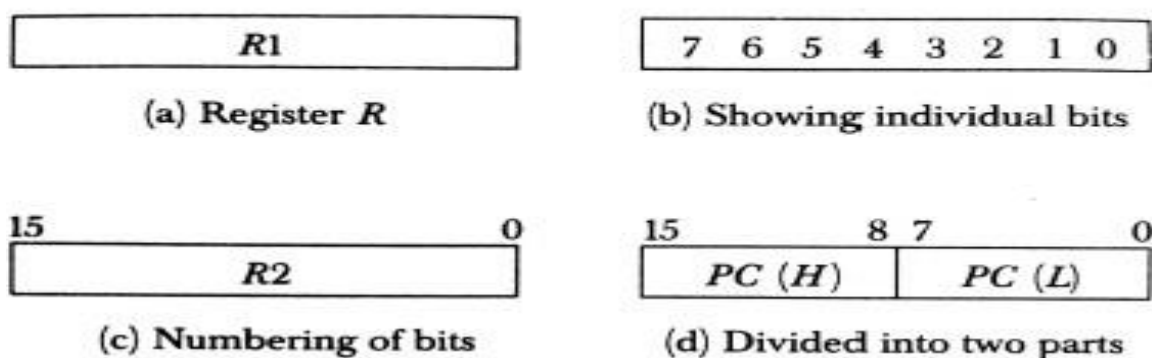
- The symbolic notation used to describe the micro-operation transfer among registers is called RTL (Register Transfer Language).
- The use of **symbols** instead of a **narrative explanation** provides an organized and concise manner for listing the micro-operation sequences in registers and the control functions that initiate them.

- A register transfer language is a system for expressing in symbolic form the microoperation sequences among the registers of a digital module.
- It is a convenient tool for describing the internal organization of digital computers in concise and precise manner.

### Registers:

- Computer registers are designated by upper case letters (and optionally followed by digits or letters) to denote the function of the register.
- For example, the register that holds an address for the memory unit is usually called a memory address register and is designated by the name **MAR**.
- Other designations for registers are **PC** (for program counter), **IR** (for instruction register, and **R1** (for processor register).
- The individual flip-flops in an n-bit register are numbered in sequence from 0 through n-1, starting from 0 in the rightmost position and increasing the numbers toward the left.
- Figure 4-1 shows the representation of registers in block diagram form.

**Figure 4-1** Block diagram of register.



- The most common way to represent a register is by a rectangular box with the name of the register inside, as in Fig. 4-1(a).
- The individual bits can be distinguished as in (b).
- The numbering of bits in a 16-bit register can be marked on top of the box as shown in (c).
- 16-bit register is partitioned into two parts in (d). Bits 0 through 7 are assigned the symbol L (for low byte) and bits 8 through 15 are assigned the symbol H (for high byte).
- The name of the 16-bit register is *PC*. The symbol *PC (0-7)* or *PC (L)* refers to the low-order byte and *PC (8-15)* or *PC (H)* to the high-order byte.

### Register Transfer:

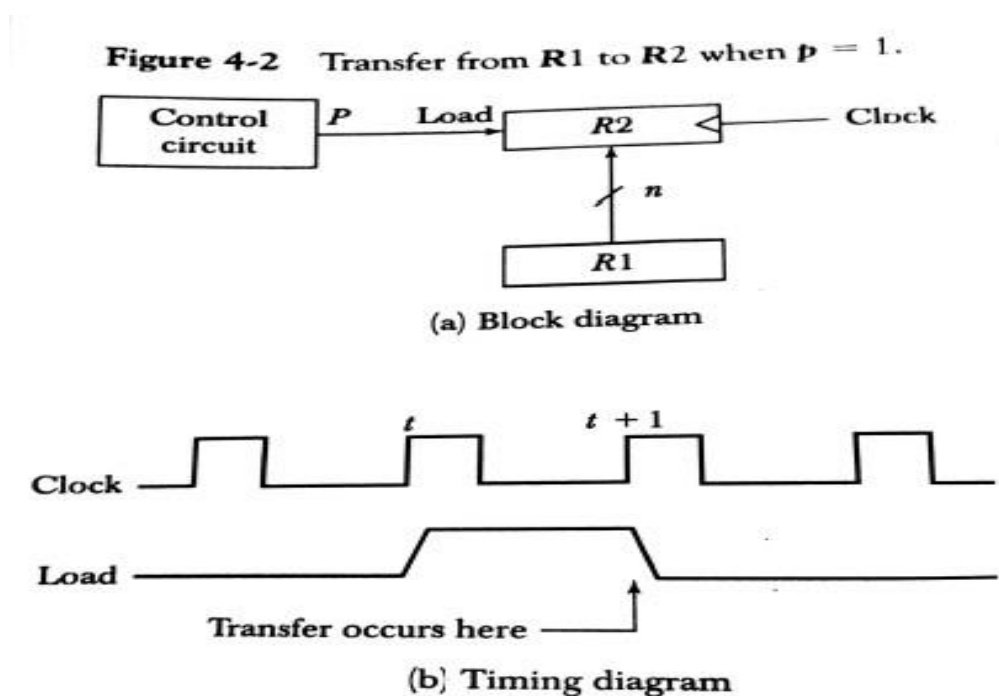
- Information transfer from one register to another is designated in symbolic form by means of a *replacement operator*.
- The statement **R2 ← R1** denotes a transfer of the content of register R1 into register R2.
- It designates a replacement of the content of R2 by the content of R1.
- By definition, the content of the source register R 1 does not change after the transfer.
- If we want the transfer to occur only under a predetermined control condition then it can be shown by an if-then statement.

**if (P=1) then R2 ← R1**

- P is the control signal generated by a control section.
- We can separate the control variables from the register transfer operation by specifying a **Control Function**.
- Control function is a Boolean variable that is equal to 0 or 1.
- control function is included in the statement as

**P: R2 ← R1**

- Control condition is terminated by a colon implies transfer operation be executed by the hardware only if P=1.
- Every statement written in a register transfer notation implies a hardware construction for implementing the transfer.
- Figure 4-2 shows the block diagram that depicts the transfer from R1 to R2.



- The n outputs of register R1 are connected to the n inputs of register R2.
- The letter n will be used to indicate any number of bits for the register. It will be replaced by an actual number when the length of the register is known.
- Register R2 has a load input that is activated by the control variable P.
- It is assumed that the control variable is synchronized with the same clock as the one applied to the register.
- As shown in the timing diagram, P is activated in the control section by the rising edge of a clock pulse at time t.
- The next positive transition of the clock at time t + 1 finds the load input active and the data inputs of R2 are then loaded into the register in parallel.

- P may go back to 0 at time  $t+1$ ; otherwise, the transfer will occur with every clock pulse transition while P remains active.
- Even though the control condition such as P becomes active just after time  $t$ , the actual transfer does not occur until the register is triggered by the next positive transition of the clock at time  $t+1$ .
- The basic symbols of the register transfer notation are listed in below table

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

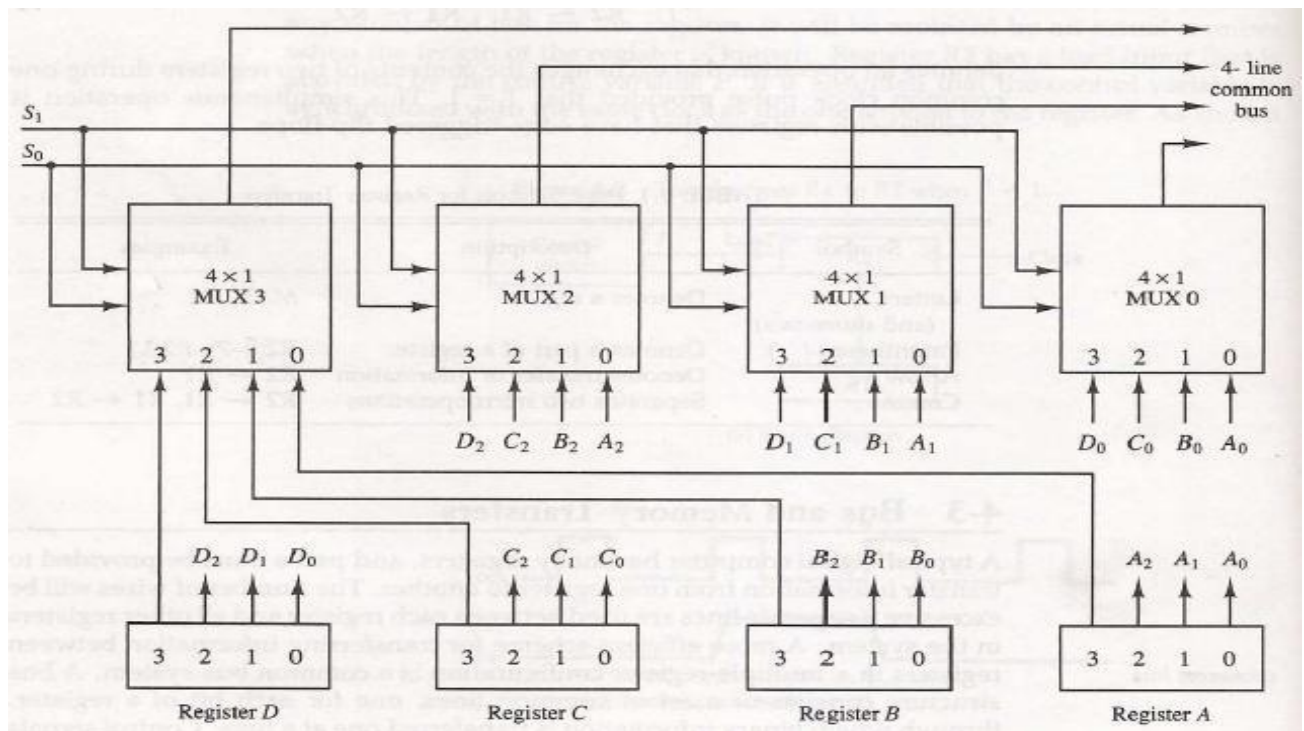
- A comma is used to separate two or more operations that are executed at the same time.
- The statement  
**T :  $R2 \leftarrow R1, R1 \leftarrow R2$**  (exchange operation)  
denotes an operation that exchanges the contents of two registers during one common clock pulse provided that  $T=1$ .

### **Bus and Memory Transfers:**

- A more efficient scheme for transferring information between registers in **a multiple-register configuration** is a **Common Bus System**.
- A common bus consists of a set of common lines, one for each bit of a register.
- Control signals determine which register is selected by the bus during each particular register transfer.
- Different ways of constructing a Common Bus System
  - ✓ Using Multiplexers
  - ✓ Using Tri-state Buffers

### **Common bus system is with multiplexers:**

- The multiplexers select the source register whose binary information is then placed on the bus.
- The construction of a bus system for four registers is shown in below Figure.



- The bus consists of four 4 x 1 multiplexers each having four data inputs, 0 through 3, and two selection inputs,  $S_1$  and  $S_0$ .
- For example, output 1 of register A is connected to input 0 of MUX 1 because this input is labelled  $A_1$ .
- The diagram shows that the bits in the same significant position in each register are connected to the data inputs of one multiplexer to form one line of the bus.
- Thus MUX 0 multiplexes the four 0 bits of the registers, MUX 1 multiplexes the four 1 bits of the registers, and similarly for the other two bits.
- The two selection lines  $S_1$  and  $S_0$  are connected to the selection inputs of all four multiplexers.
- The selection lines choose the four bits of one register and transfer them into the four-line common bus.
- When  $S_1S_0 = 00$ , the 0 data inputs of all four multiplexers are selected and applied to the outputs that form the bus.
- This causes the bus lines to receive the content of register A since the outputs of this register are connected to the 0 data inputs of the multiplexers.
- Similarly, register B is selected if  $S_1S_0 = 01$ , and so on.
- Table 4-2 shows the register that is selected by the bus for each of the four possible binary value of the selection lines.

$S_1$	$S_0$	Register selected
0	0	A
0	1	B
1	0	C
1	1	D

- In general a bus system has
  - ✓ multiplex "k" Registers

- ✓ each register of “n” bits
- ✓ to produce “n-line bus”
- ✓ no. of multiplexers required = n
- ✓ size of each multiplexer = k x 1

➤ When the bus is included in the statement, the register transfer is symbolized as follows:

**BUS ← C, R1 ← BUS**

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

**R1 ← C**

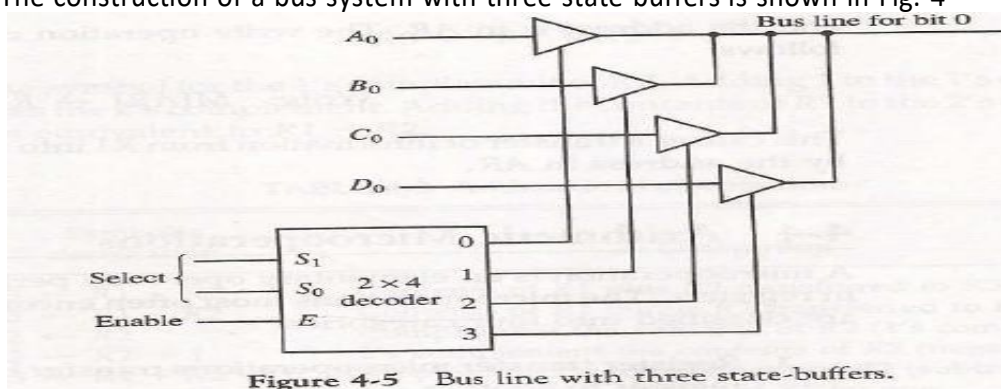
### Three-State Bus Buffers:

- A bus system can be constructed with three-state gates instead of multiplexers.
- A three-state gate is a digital circuit that exhibits three states.
- Two of the states are signals equivalent to logic 1 and 0 as in a conventional gate.
- The third state is a *high-impedance state*.
- The high-impedance state behaves like an open circuit, which means that the output is disconnected and does not have logic significance.
- Because of this feature, a large number of three-state gate outputs can be connected with wires to form a common bus line without endangering loading effects.
- The graphic symbol of a three-state buffer gate is shown in Fig. 4-4.

**Figure 4-4** Graphic symbols for three-state buffer.



- It is distinguished from a normal buffer by having both a normal input and a control input.
- The control input determines the output state. When the control input is equal to 1, the output is enabled and the gate behaves like any conventional buffer, with the output equal to the normal input.
- When the control input is 0, the output is disabled and the gate goes to a high-impedance state, regardless of the value in the normal input.
- The construction of a bus system with three-state buffers is shown in Fig. 4



**Figure 4-5** Bus line with three state-buffers.

- The outputs of four buffers are connected together to form a single bus line.
- The control inputs to the buffers determine which of the four normal inputs will communicate with the bus line.
- No more than one buffer may be in the active state at any given time. The connected buffers must be controlled so that only one three-state buffer has access to the bus line while all other buffers are maintained in a high impedance state.
- One way to ensure that no more than one control input is active at any given time is to use a decoder, as shown in the diagram.
- When the enable input of the decoder is 0, all of its four outputs are 0, and the bus line is in a high-impedance state because all four buffers are disabled.
- When the enable input is active, one of the three-state buffers will be active, depending on the binary value in the select inputs of the decoder.

## Memory Transfer:

- The transfer of information from a memory word to the outside environment is called a *read* operation.
- The transfer of new information to be stored into the memory is called a *write* operation.
- A memory word will be symbolized by the letter M.
- The particular memory word among the many available is selected by the memory address during the transfer.
- It is necessary to specify the address of M when writing memory transfer operations.
- This will be done by enclosing the address in square brackets following the letter M.
- Consider a memory unit that receives the address from a register, called the address register, symbolized by AR.
- The data are transferred to another register, called the data register, symbolized by DR.
- The read operation can be stated as follows:

**Read: DR ← M [AR]**

- This causes a transfer of information into DR from the memory word M selected by the address in AR.
- The write operation transfers the content of a data register to a memory word M selected by the address. Assume that the input data are in register R1 and the address is in AR.
- The write operation can be stated as follows:

**Write: M [AR] ← R1**

## Types of Micro-operations:

- Register Transfer Micro-operations: Transfer binary information from one register to another.
- Arithmetic Micro-operations: Perform arithmetic operation on numeric data stored in registers.
- Logical Micro-operations: Perform bit manipulation operations on data stored in registers.
- Shift Micro-operations: Perform shift operations on data stored in registers.
- Register Transfer Micro-operation doesn't change the information content when the binary information moves from source register to destination register.



- Other three types of micro-operations change the information content during the transfer.

### Arithmetic Micro-operations:

- The basic arithmetic micro-operations are
  - Addition
  - Subtraction
  - Increment
  - Decrement
  - Shift
- The arithmetic Micro-operation defined by the statement below specifies the add micro-operation.

$$R3 \leftarrow R1 + R2$$

- It states that the contents of R1 are added to contents of R2 and sum is transferred to R3.
- To implement this statement hardware requires 3 registers and digital component that performs addition
- Subtraction is most often implemented through complementation and addition.
- The subtract operation is specified by the following statement

$$R3 \leftarrow R1 + \overline{R2} + 1$$

- instead of minus operator, we can write as
- $\overline{R2}$  is the symbol for the 1's complement of R2
- Adding 1 to 1's complement produces 2's complement
- Adding the contents of R1 to the 2's complement of R2 is equivalent to  $R1 - R2$ .

### **Binary Adder:**

- Digital circuit that forms the arithmetic sum of 2 bits and the previous carry is called **FULL ADDER**.
- Digital circuit that generates the arithmetic sum of 2 binary numbers of any lengths is called **BINARY ADDER**.
- Figure 4-6 shows the interconnections of four full-adders (FA) to provide a 4-bit binary adder.

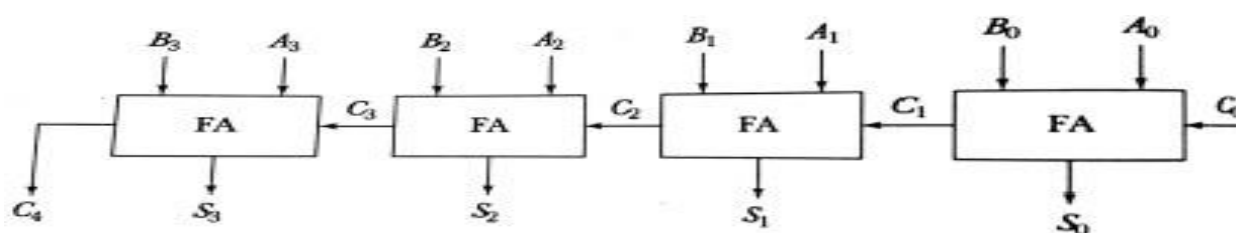
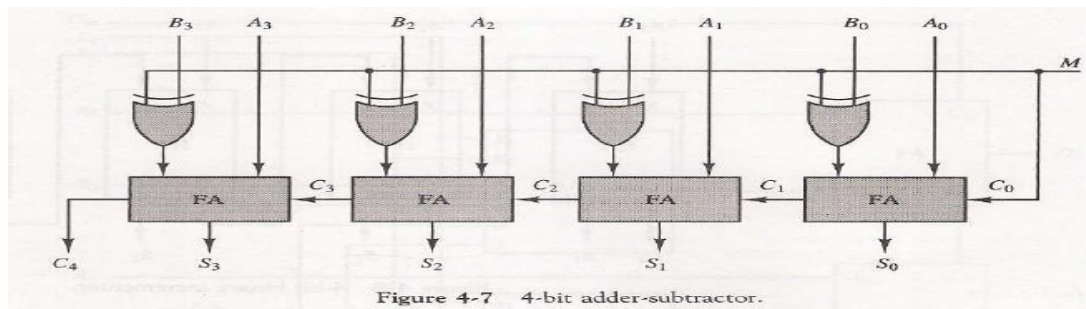


Figure 4-6 4-bit binary adder.

- The augends bits of A and the addend bits of B are designated by subscript numbers from right to left, with subscript 0 denoting the low-order bit.
- The carries are connected in a chain through the full-adders. The input carry to the binary adder is  $C_0$  and the output carry is  $C_4$ . The S outputs of the full-adders generate the required sum bits.
- An n-bit binary adder requires n full-adders.

## Binary Adder – Subtractor:

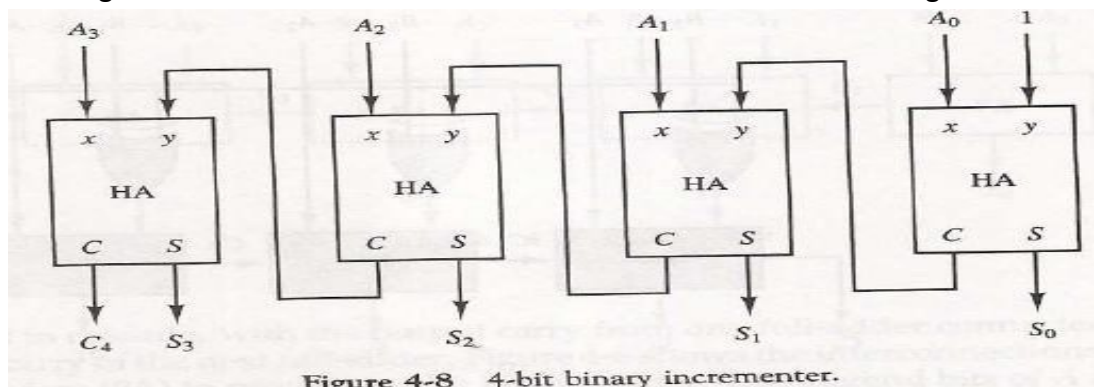
- The addition and subtraction operations can be combined into one common circuit by including an exclusive-OR gate with each full-adder.
- A 4-bit adder-subtractor circuit is shown in Fig. 4-7.



- The mode input  $M$  controls the operation. When  $M = 0$  the circuit is an adder and when  $M = 1$  the circuit becomes a subtractor.
- Each exclusive-OR gate receives input  $M$  and one of the inputs of  $B$
- When  $M = 0$ , we have  $B \text{ xor } 0 = B$ . The full-adders receive the value of  $B$ , the input carry is 0, and the circuit performs  $A$  plus  $B$ .
- When  $M = 1$ , we have  $B \text{ xor } 1 = B'$  and  $C_0 = 1$ .
- The  $B$  inputs are all complemented and a 1 is added through the input carry.
- The circuit performs the operation  $A$  plus the 2's complement of  $B$ .

## Binary Incrementer:

- The increment microoperation adds one to a number in a register.
- For example, if a 4-bit register has a binary value 0110, it will go to 0111 after it is incremented.
- This can be accomplished by means of half-adders connected in cascade.
- The diagram of a 4-bit 'combinational circuit incrementer is shown in Fig. 4-8.

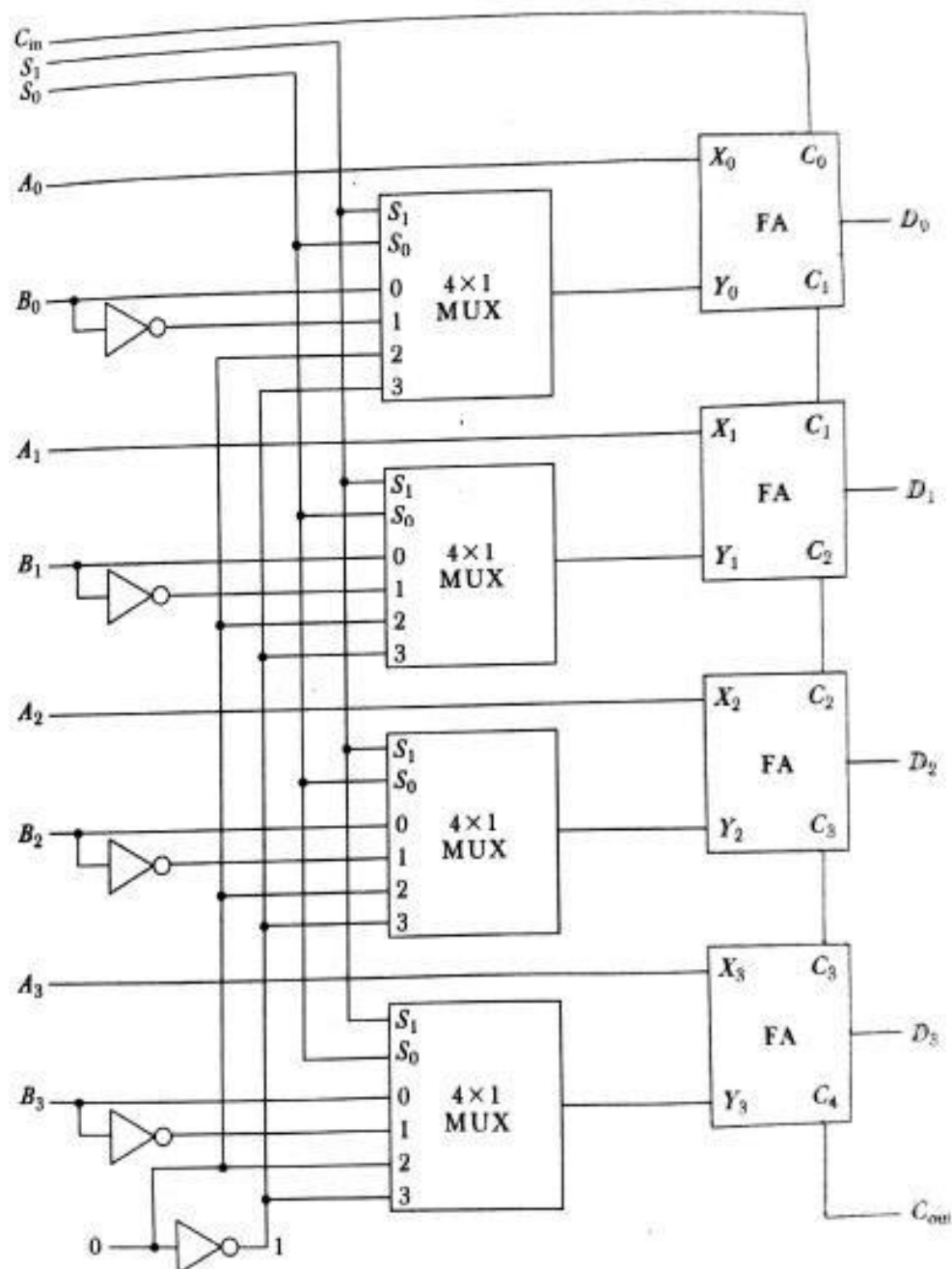


- One of the inputs to the least significant half-adder (HA) is connected to logic-1 and the other input is connected to the least significant bit of the number to be incremented.
- The output carry from one half-adder is connected to one of the inputs of the next-higher-order half-adder.
- The circuit receives the four bits from  $A_0$  through  $A_3$ , adds one to it, and generates the incremented output in  $S_0$  through  $S_3$ .
- The output carry  $C_4$  will be 1 only after incrementing binary 1111. This also causes outputs  $S_0$  through  $S_3$  to go to 0.

- The circuit of Fig. 4-8 can be extended to an  $n$ -bit binary incrementer by extending the diagram to include  $n$  half-adders.
- The least significant bit must have one input connected to logic-1. The other inputs receive the number to be incremented or the carry from the previous stage.

### Arithmetic Circuit:

- The basic component of an arithmetic circuit is the parallel adder.
- By controlling the data inputs to the adder, it is possible to obtain different types of arithmetic operations.
- The diagram of a 4-bit arithmetic circuit is shown in Fig. 4-9. It has four full-adder circuits that constitute the 4-bit adder and four multiplexers for choosing different operations.



- There are two 4-bit inputs  $A$  and  $B$  and a 4-bit output  $D$ .
- The four inputs from  $A$  go directly to the  $X$  inputs of the binary adder.
- Each of the four inputs from  $B$  are connected to the data inputs of the multiplexers.
- The multiplexers data inputs also receive the complement of  $B$ .
- The other two data inputs are connected to logic-0 and logic-1.
- The four multiplexers are controlled by two selection inputs  $S_1$  and  $S_0$ . The input carry  $C_{in}$ , goes to the carry input of the FA in the least significant position. The other carries are connected from one stage to the next.
- By controlling the value of  $Y$  with the two selection inputs  $S_1$  and  $S_0$  and making  $C_{in}$  equal to 0 or 1, it is possible to generate the eight arithmetic microoperations listed in Table 44.

**TABLE 4-4 Arithmetic Circuit Function Table**

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

#### **Addition:**

- When  $S_1S_0 = 00$ , the value of  $B$  is applied to the  $Y$  inputs of the adder.
  - ✓ If  $C_{in} = 0$ , the output  $D = A + B$ .
  - ✓ If  $C_{in} = 1$ , output  $D = A + B + 1$ .
- Both cases perform the add microoperation with or without adding the input carry.

#### **Subtraction:**

- When  $S_1S_0 = 01$ , the complement of  $B$  is applied to the  $Y$  inputs of the adder.
  - ✓ If  $C_{in} = 1$ , then  $D = A + \bar{B} + 1$ . This produces  $A$  plus the 2's complement of  $B$ , which is equivalent to a subtraction of  $A - B$ .
  - ✓ When  $C_{in} = 0$  then  $D = A + \bar{B}$ . This is equivalent to a subtract with borrow, that is,  $A - B - 1$ .

#### **Increment:**

- When  $S_1S_0 = 10$ , the inputs from  $B$  are neglected, and instead, all 0's are inserted into the  $Y$  inputs. The output becomes  $D = A + 0 + C_{in}$ . This gives  $D = A$  when  $C_{in} = 0$  and  $D = A + 1$  when  $C_{in} = 1$ .
- In the first case we have a direct transfer from input  $A$  to output  $D$ .
- In the second case, the value of  $A$  is incremented by 1.

**Decrement:**

- When  $S_1S_0 = 11$ , all 1's are inserted into the Y inputs of the adder to produce the decrement operation  $D = A - 1$  when  $C_{in} = 0$ .
- This is because a number with all 1's is equal to the 2's complement of 1 (the 2's complement of binary 0001 is 1111). Adding a number A to the 2's complement of 1 produces  $F = A + 2$ 's complement of 1 =  $A - 1$ . When  $C_{in} = 1$ , then  $D = A - 1 + 1 = A$ , which causes a direct transfer from input A to output D.

**Logic Micro-operations:**

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

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

- It specifies a logic microoperation to be executed on the individual bits of the registers provided that the control variable  $P = 1$ .

**List of Logic Microoperations:**

- There are 16 different logic operations that can be performed with two binary variables.
- They can be determined from all possible truth tables obtained with two binary variables as shown in Table 4-5.

**TABLE 4-5** Truth Tables for 16 Functions of Two Variables

<i>x</i>	<i>y</i>	$F_0$	$F_1$	$F_2$	$F_3$	$F_4$	$F_5$	$F_6$	$F_7$	$F_8$	$F_9$	$F_{10}$	$F_{11}$	$F_{12}$	$F_{13}$	$F_{14}$	$F_{15}$
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

- The 16 Boolean functions of two variables  $x$  and  $y$  are expressed in algebraic form in the first column of Table 4-6.
- The 16 logic microoperations are derived from these functions by replacing variable  $x$  by the binary content of register A and variable  $y$  by the binary content of register B.
- The logic micro-operations listed in the second column represent a relationship between the binary content of two registers A and B.



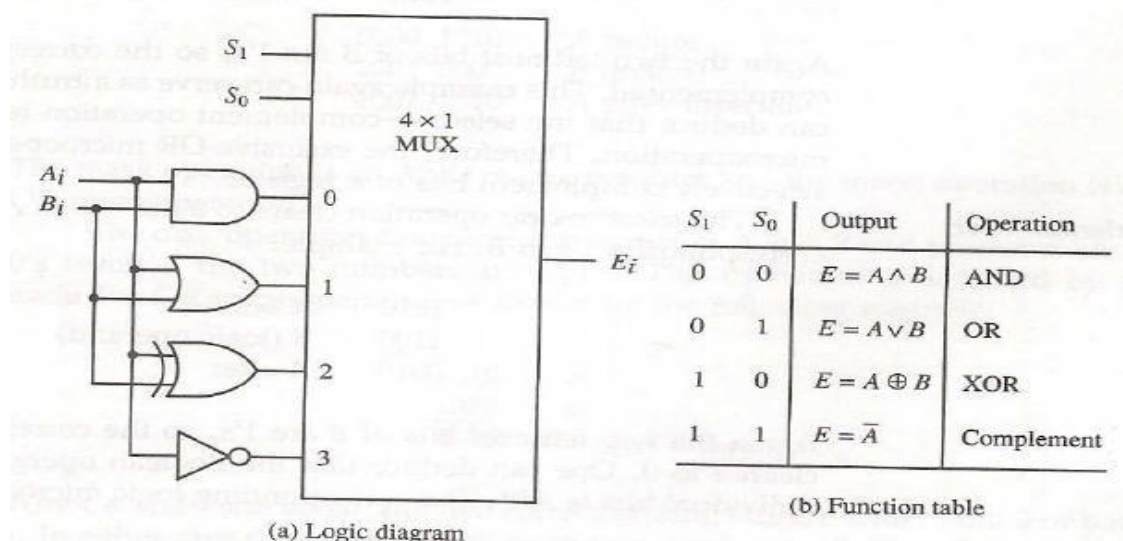
TABLE 4-6 Sixteen Logic Microoperations

Boolean function	Microoperation	Name
$F_0 = 0$	$F \leftarrow 0$	Clear
$F_1 = xy$	$F \leftarrow A \wedge B$	AND
$F_2 = xy'$	$F \leftarrow A \wedge \bar{B}$	
$F_3 = x$	$F \leftarrow A$	Transfer $A$
$F_4 = x'y$	$F \leftarrow \bar{A} \wedge B$	
$F_5 = y$	$F \leftarrow B$	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
$F_{10} = y'$	$F \leftarrow \bar{B}$	Complement $B$
$F_{11} = x + y'$	$F \leftarrow A \vee \bar{B}$	
$F_{12} = x'$	$F \leftarrow \bar{A}$	Complement $A$
$F_{13} = x' + y$	$F \leftarrow \bar{A} \vee B$	
$F_{14} = (xy)'$	$F \leftarrow \bar{A} \wedge \bar{B}$	NAND
$F_{15} = 1$	$F \leftarrow \text{all 1's}$	Set to all 1's

#### Hardware Implementation:

- The hardware implementation of logic microoperations requires that logic gates be inserted for each bit or pair of bits in the registers to perform the required logic function.
- Although there are 16 logic microoperations, most computers use only four--AND, OR, XOR (exclusive-OR), and complement from which all others can be derived.
- Figure 4-10 shows one stage of a circuit that generates the four basic logic microoperations.
- It consists of four gates and a multiplexer. Each of the four logic operations is generated through a gate that performs the required logic.
- The outputs of the gates are applied to the data inputs of the multiplexer. The two selection inputs  $S_1$  and  $S_0$  choose one of the data inputs of the multiplexer and direct its value to the output.

Figure 4-10 One stage of logic circuit.



### Some Applications:

- Logic micro-operations are very useful for manipulating individual bits or a portion of a word stored in a register.
- They can be used to change bit values, delete a group of bits or insert new bits values into a register.
- The following example shows how the bits of one register (designated by A) are manipulated by logic microoperations as a function of the bits of another register (designated by B).
- Selective set
  - ✓ The *selective-set* operation sets to 1 the bits in register A where there are corresponding 1's in register B. It does not affect bit positions that have 0's in B. The following numerical example clarifies this operation:

1010	A before
<u>1100</u>	B (logic operand)
1110	A after

- ✓ The OR microoperation can be used to selectively set bits of a register.

- Selective complement
  - ✓ The *selective-complement* operation complements bits in A where there are corresponding 1's in B. It does not affect bit positions that have 0's in B. For example:

1010	A before
<u>1100</u>	B (logic operand)
0110	A after

- ✓ The exclusive-OR microoperation can be used to selectively complement bits of a register.

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

1010	A before
<u>1100</u>	B (logic operand)
0010	A after

- ✓ The corresponding logic microoperation is  $A \leftarrow A \wedge \bar{B}$

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

0110	1010	A before
<u>0000</u>	1111	B (mask)
0000	1010	A after masking

- Insert
  - ✓ The *insert* operation inserts a new value into a group of bits. This is done by first masking the bits and then ORing them with the required value.

- ✓ For example, suppose that an A register contains eight bits, 0110 1010. To replace the four leftmost bits by the value 1001 we first mask the four unwanted bits:

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

and then insert the new value:

0000 1010	A before
1001 0000	B (insert)
1001 1010	A after insertion

- ✓ The mask operation is an AND microoperation and the insert operation is an OR microoperation.

#### ➤ Clear

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

1010	A
1010	B
0000	$A \leftarrow A \oplus B$

### **Shift Microoperations:**

- Shift microoperations are used for serial transfer of data.
- The contents of a register can be shifted to the left or the right.
- During a shift-left operation the serial input transfers a bit into the rightmost position.
- During a shift-right operation the serial input transfers a bit into the leftmost position.
- There are three types of shifts: logical, circular, and arithmetic.
- The symbolic notation for the shift microoperations is shown in Table 4-7.

**TABLE 4-7** Shift Microoperations

Symbolic designation	Description
$R \leftarrow shl R$	Shift-left register <i>R</i>
$R \leftarrow shr R$	Shift-right register <i>R</i>
$R \leftarrow cil R$	Circular shift-left register <i>R</i>
$R \leftarrow cir R$	Circular shift-right register <i>R</i>
$R \leftarrow ashl R$	Arithmetic shift-left <i>R</i>
$R \leftarrow ashr R$	Arithmetic shift-right <i>R</i>

#### ➤ **Logical Shift:**

- A *logical* shift is one that transfers 0 through the serial input.
- The symbols *shl* and *shr* for logical shift-left and shift-right microoperations.



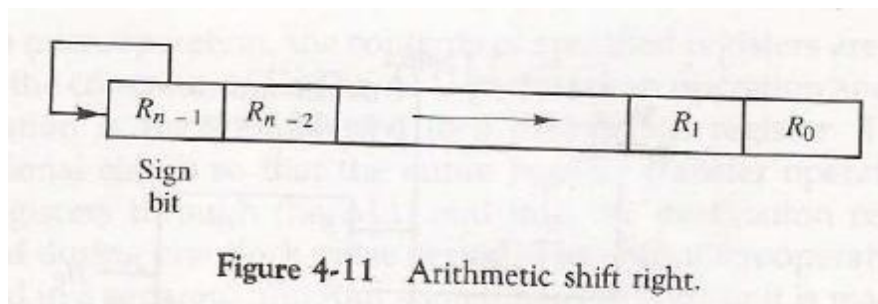
- The microoperations that specify a 1-bit shift to the left of the content of register R and a 1-bit shift to the right of the content of register R shown in table 4.7.
- The bit transferred to the end position through the serial input is assumed to be 0 during a logical shift.

➤ **Circular Shift:**

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

➤ **Arithmetic Shift:**

- An *arithmetic shift* is a microoperation that shifts a signed binary number to the left or right.
- An arithmetic shift-left multiplies a signed binary number by 2.
- An arithmetic shift-right divides the number by 2.
- Arithmetic shifts must leave the sign bit unchanged because the sign of the number remains the same when it is multiplied or divided by 2.



**Hardware Implementation:**

- A combinational circuit shifter can be constructed with multiplexers as shown in Fig. 4-12.
- The 4-bit shifter has four data inputs,  $A_0$  through  $A_3$ , and four data outputs,  $H_0$  through  $H_3$ .
- There are two serial inputs, one for shift left ( $I_L$ ) and the other for shift right ( $I_R$ ).
- When the selection input  $S=0$  the input data are shifted right (down in the diagram).
- When  $S = 1$ , the input data are shifted left (up in the diagram).
- The function table in Fig. 4-12 shows which input goes to each output after the shift.
- A shifter with  $n$  data inputs and outputs requires  $n$  multiplexers.
- The two serial inputs can be controlled by another multiplexer to provide the three possible types of shifts.

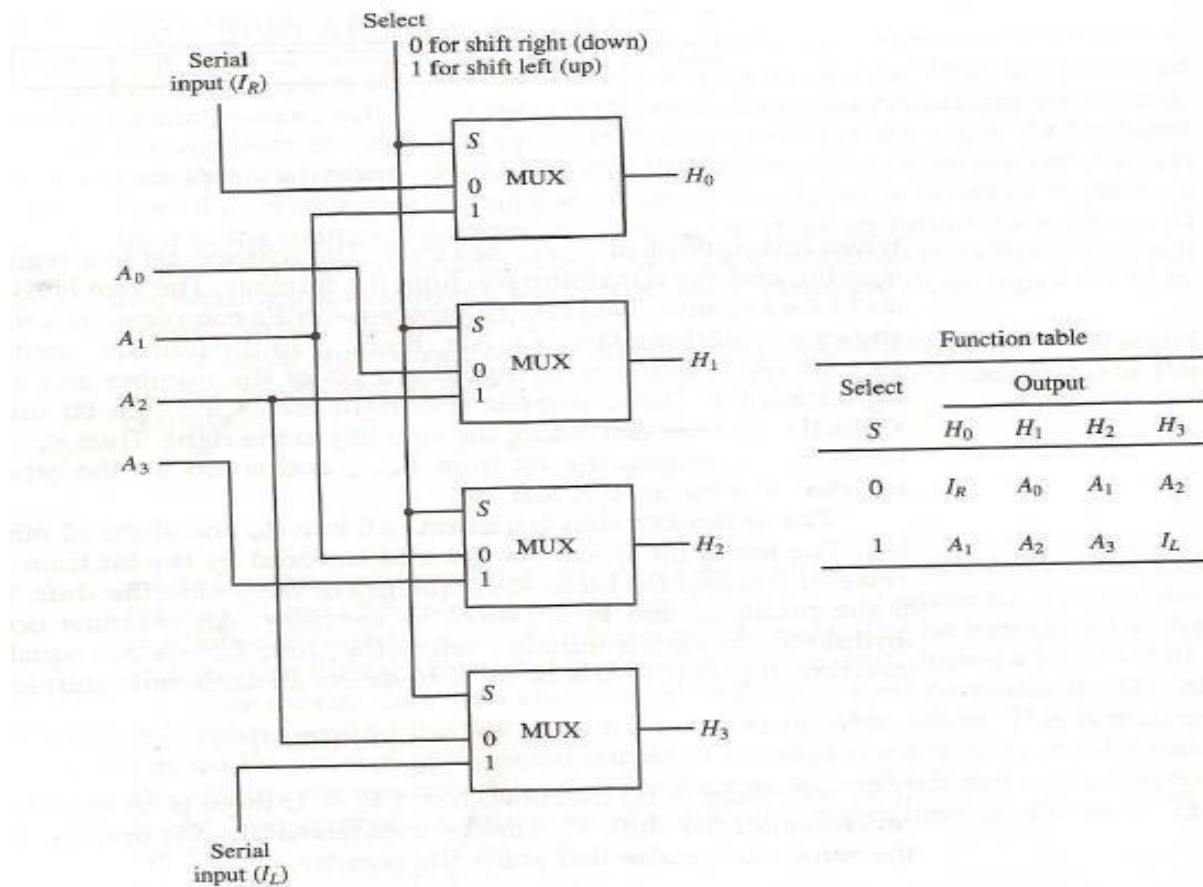
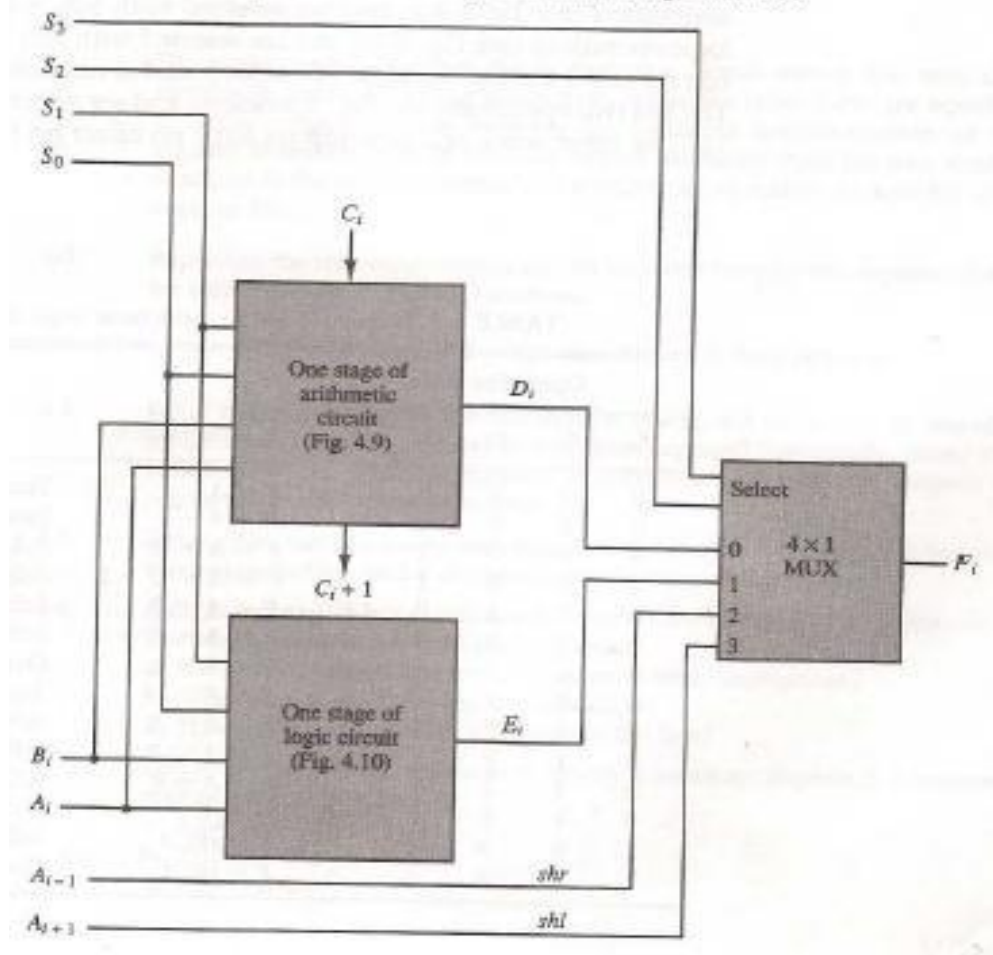


Figure 4-12 4-bit combinational circuit shifter.

### Arithmetic Logic Shift Unit:

- Instead of having individual registers performing the microoperations directly, computer systems employ a number of storage registers connected to a common operational unit called an arithmetic logic unit, abbreviated ALU.
- The ALU is a combinational circuit so that the entire register transfer operation from the source registers through the ALU and into the destination register can be performed during one clock pulse period.
- The shift microoperations are often performed in a separate unit, but sometimes the shift unit is made part of the overall ALU.
- The arithmetic, logic, and shift circuits introduced in previous sections can be combined into one ALU with common selection variables. One stage of an arithmetic logic shift unit is shown in Fig. 4-13.
- Particular microoperation is selected with inputs  $S_1$  and  $S_0$ . A 4 x 1 multiplexer at the output chooses between an arithmetic output in  $D_i$  and a logic output in  $E_i$ .
- The data in the multiplexer are selected with inputs  $S_3$  and  $S_2$ . The other two data inputs to the multiplexer receive inputs  $A_{i-1}$  for the shift-right operation and  $A_{i+1}$  for the shift-left operation.
- The circuit whose one stage is specified in Fig. 4-13 provides eight arithmetic operation, four logic operations, and two shift operations.
- Each operation is selected with the five variables  $S_3, S_2, S_1, S_0$  and  $C_{in}$ .
- The input carry  $C_{in}$  is used for selecting an arithmetic operation only.

Figure 4-13 One stage of arithmetic logic shift unit.



- Table 4-8 lists the 14 operations of the ALU. The first eight are arithmetic operations and are selected with  $S_3S_2 = 00$ .
- The next four are logic and are selected with  $S_3S_2 = 01$ .
- The input carry has no effect during the logic operations and is marked with don't-care x's.
- The last two operations are shift operations and are selected with  $S_3S_2 = 10$  and  $11$ .
- The other three selection inputs have no effect on the shift.

TABLE 4-8 Function Table for 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	x	$F = A \wedge B$	AND
0	1	0	1	x	$F = A \vee B$	OR
0	1	1	0	x	$F = A \oplus B$	XOR
0	1	1	1	x	$F = \overline{A}$	Complement A
1	0	x	x	x	$F = shr A$	Shift right A into F
1	1	x	x	x	$F = shl A$	Shift left A into F

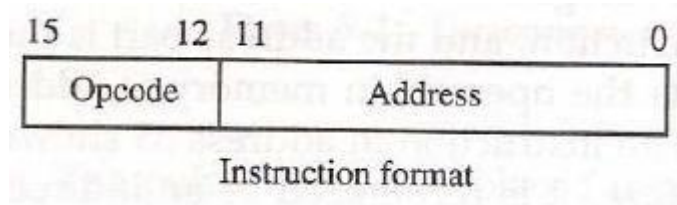
## BASIC COMPUTER ORGANIZATION AND DESIGN

### CONTENTS:

- ✓ Instruction Codes
- ✓ Computer Registers
- ✓ Computer Instructions
- ✓ Timing And Control
- ✓ Instruction Cycle
- ✓ Register – Reference Instructions
- ✓ Memory – Reference Instructions
- ✓ Input – Output And Interrupt

### 1. Instruction Codes:

- The organization of the computer is defined by its internal registers, the timing and control structure, and the set of instructions that it uses.
- Internal organization of a computer is defined by the sequence of micro-operations it performs on data stored in its registers.
- Computer can be instructed about the specific sequence of operations it must perform.
- User controls this process by means of a Program.
- **Program:** set of instructions that specify the operations, operands, and the sequence by which processing has to occur.
- **Instruction:** a binary code that specifies a sequence of micro-operations for the computer.
- 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. – *Instruction Cycle*
- **Instruction Code:** group of bits that instruct the computer to perform specific operation.
- Instruction code is usually divided into two parts: Opcode and address(operand)

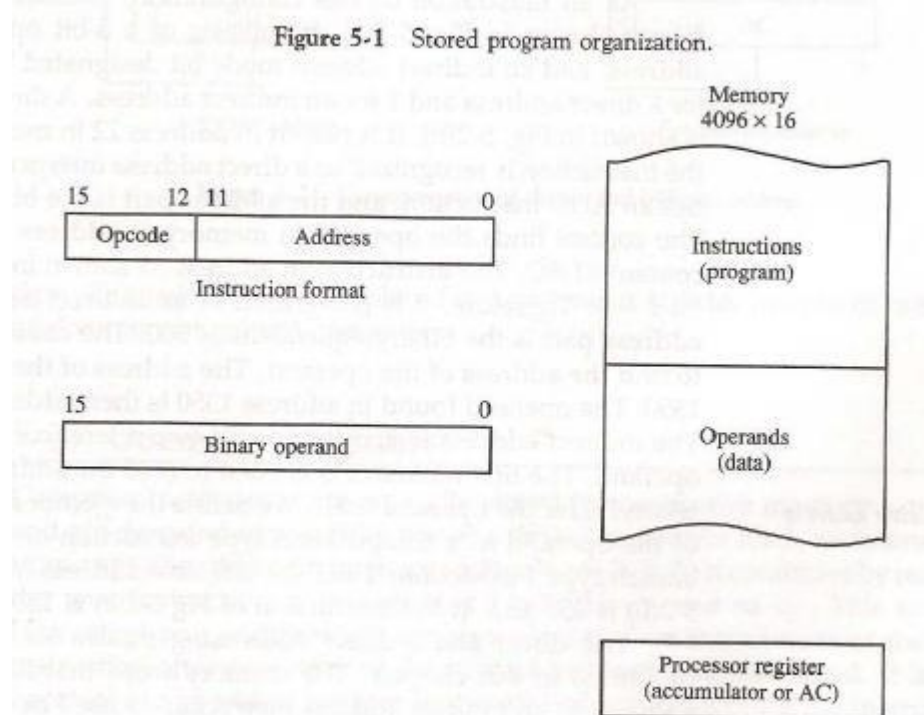


- Operation Code (opcode):
  - ✓ group of bits that define the operation
  - ✓ Eg: add, subtract, multiply, shift, complement.
  - ✓ No. of bits required for opcode depends on no. of operations available in computer.
  - ✓  $n$  bit opcode  $\geq 2^n$  (or less) operations
- Address (operand):
  - ✓ specifies the location of operands (registers or memory words)
  - ✓ Memory words are specified by their address
  - ✓ Registers are specified by their  $k$ -bit binary code

- ✓ k-bit address  $\geq 2^k$  registers

## Stored Program Organization:

- The ability to store and execute instructions is the most important property of a general-purpose computer. That type of stored program concept is called stored program organization.
- The simplest way to organize a computer is to have one processor register and an instruction code format with two parts. The first part specifies the operation to be performed and the second specifies an address.
- The below figure shows the stored program organization



- Instructions are stored in one section of memory and data in another.
- For a memory unit with 4096 words we need 12 bits to specify an address since  $2^{12} = 4096$ .
- If we store each instruction code in one 16-bit memory word, we have available four bits for the operation code (abbreviated opcode) to specify one out of 16 possible operations, and 12 bits to specify the address of an operand.
- **Accumulator (AC):**
  - ✓ Computers that have a single-processor register usually assign to it the name accumulator and label it AC.
  - ✓ The operation is performed with the memory operand and the content of AC.

## Addressing of Operand:

- Sometimes convenient to use the address bits of an instruction code not as an address but as the actual operand.
- When the second part of an instruction code specifies an operand, the instruction is said to have an **immediate operand**.
- When the second part specifies the address of an operand, the instruction is said to have a **direct address**.
- When second part of the instruction designate an address of a memory word in which the address of the operand is found such instruction have **indirect address**.
- One bit of the instruction code can be used to distinguish between a direct and an indirect address.
- The instruction code format shown in Fig. 5-2(a). It consists of a 3-bit operation code, a 12-bit

address, and an indirect address mode bit designated by I. The mode bit is 0 for a direct address and 1 for an indirect address.

- A direct address instruction is shown in Fig. 5-2(b).
- It is placed in address 22 in memory. The I bit is 0, so the instruction is recognized as a direct address instruction. The opcode specifies an ADD instruction, and the address part is the binary equivalent of 457.
- The control finds the operand in memory at address 457 and adds it to the content of AC.
- The instruction in address 35 shown in Fig. 5-2(c) has a mode bit I = 1.
- Therefore, it is recognized as an indirect address instruction.
- The address part is the binary equivalent of 300. The control goes to address 300 to find the address of the operand. The address of the operand in this case is 1350.
- The operand found in address 1350 is then added to the content of AC.
- The **effective address** to be the address of the operand in a computation-type instruction or the target address in a branch-type instruction.
- Thus the effective address in the instruction of Fig. 5-2(b) is 457 and in the instruction of Fig 5-2(c) is 1350.

## 2. Computer Registers:

- What is the need for computer registers?
  - ✓ The need of the registers in computer for
    - Instruction sequencing needs a counter to calculate the address of the next instruction after execution of the current instruction is completed (**PC**).
    - Necessary to provide a register in the control unit for storing the instruction code after it is read from memory (**IR**).
    - Needs processor registers for manipulating data (AC and TR) and a register for holding a memory address (**AR**).
- The above requirements dictate the register configuration shown in Fig. 5-3.
- The registers are also listed in Table 5.1 together with a brief description of their function and the number of bits that they contain.



TABLE 5-1 List of Registers for the Basic Computer

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

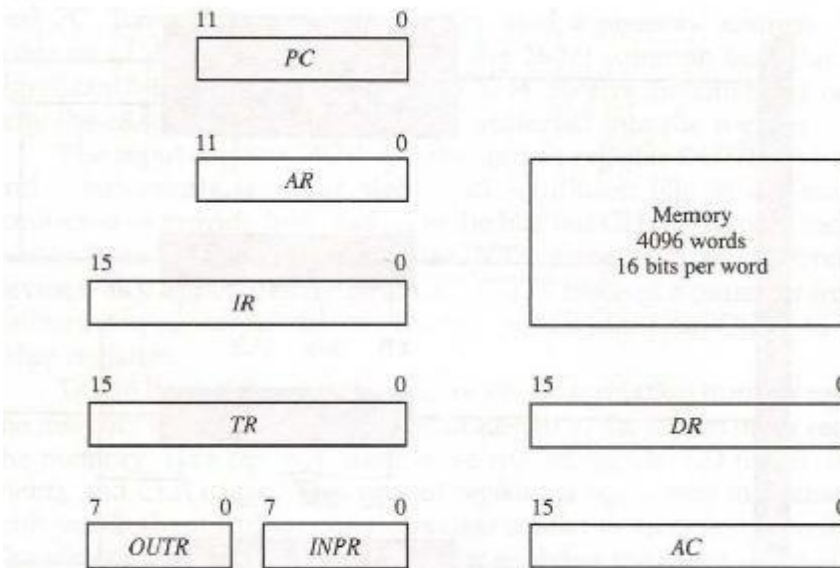


Figure 5-3 Basic computer registers and memory.

- The *data register (DR)* holds the operand read from memory.
- The *accumulator (AC)* register is a general purpose processing register.
- The instruction read from memory is placed in the *instruction register (IR)*.
- The *temporary register (TR)* is used for holding temporary data during the processing.
- The *memory address register (AR)* has 12 bits since this is the width of a memory address.
- The *program counter (PC)* also has 12 bits and it holds the address of the next instruction to be read from memory after the current instruction is executed.
- Two registers are used for input and output.
  - The *input register (INPR)* receives an 8-bit character from an input device.
  - The *output register (OUTR)* holds an 8-bit character for an output device.

### Common Bus System:

- The basic computer has eight registers, a memory unit, and a control unit
- Paths must be provided to transfer information from one register to another and between memory and registers.
- A more efficient scheme for transferring information in a system with many registers is to use a common bus.
- The connection of the registers and memory of the basic computer to a common bus system is shown in Fig. 5-4.
- The outputs of seven registers and memory are connected to the common bus.

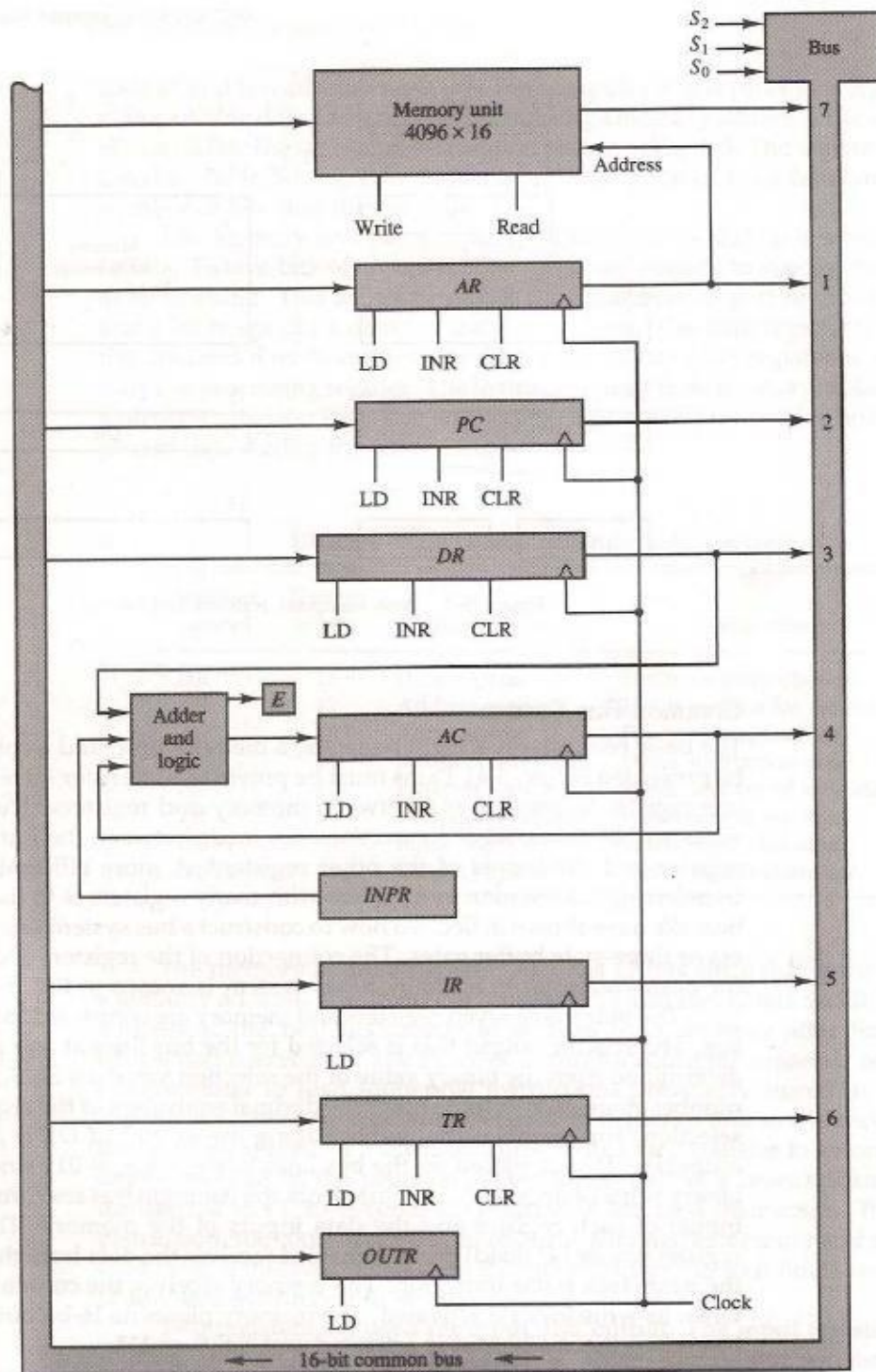


Figure 5-4 Basic computer registers connected to a common bus.

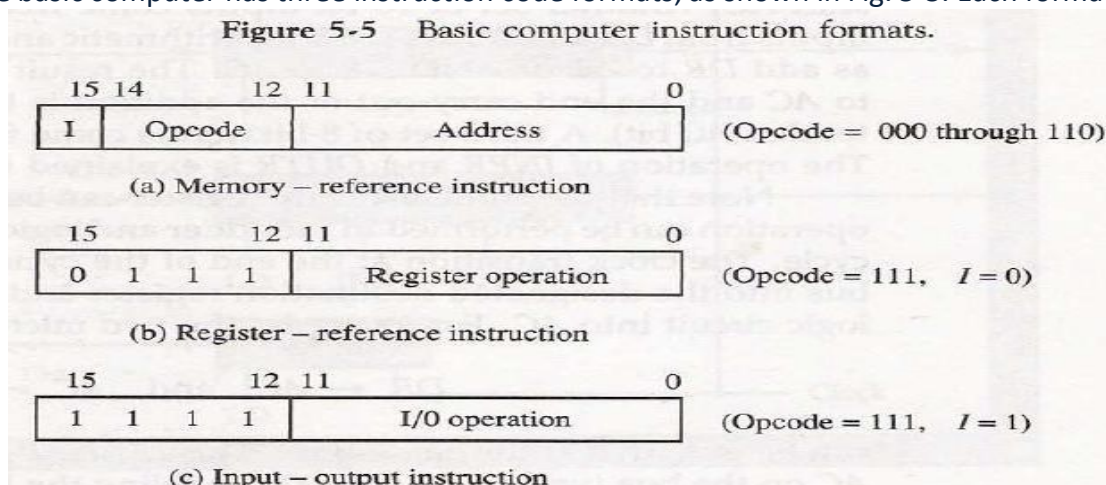
- The specific output that is selected for the bus lines at any given time is determined from the binary value of the selection variables  $S_2$ ,  $S_1$ , and  $S_0$ .
- The number along each output shows the decimal equivalent of the required binary selection.
- For example, the number along the output of DR is 3. The 16-bit outputs of DR are placed on the bus lines when  $S_2S_1S_0 = 011$ .
- The lines from the common bus are connected to the inputs of each register and the data inputs of the memory.



- The particular register whose LD (load) input is enabled receives the data from the bus during the next clock pulse transition.
- The memory receives the contents of the bus when its write input is activated.
- The memory places its 16-bit output onto the bus when the read input is activated and  $S_2S_1S_0 = 111$ .
- Two registers, AR and PC, have 12 bits each since they hold a memory address.
- When the contents of AR or PC are applied to the 16-bit common bus, the four most significant bits are set to 0's.
- When AR or PC receives information from the bus, only the 12 least significant bits are transferred into the register.
- The input register INPR and the output register OUTR have 8 bits each.
- They communicate with the eight least significant bits in the bus.
- INPR is connected to provide information to the bus but OUTR can only receive information from the bus.
- This is because INPR receives a character from an input device which is then transferred to AC.
- OUTR receives a character from AC and delivers it to an output device.
- Five registers have three control inputs: LD (load), INR (increment), and CLR (clear).
- This type of register is equivalent to a binary counter with parallel load and synchronous clear.
- Two registers have only a LD input.
- The input data and output data of the memory are connected to the common bus, but the memory address is connected to AR.
- Therefore, AR must always be used to specify a memory address.
- The 16 inputs of AC come from an adder and logic circuit. This circuit has three sets of inputs.
  - One set of 16-bit inputs come from the outputs of AC.
  - Another set of 16-bit inputs come from the data register DR.
  - The result of an addition is transferred to AC and the end carry-out of the addition is transferred to flip-flop E (extended AC bit).
  - A third set of 8-bit inputs come from the input register INPR.
- The content of any register can be applied onto the bus and an operation can be performed in the adder and logic circuit during the same clock cycle.
- For example, the two microoperations  $DR \leftarrow AC$  and  $AC \leftarrow DR$  can be executed at the same time.
- This can be done by placing the content of AC on the bus (with  $S_2S_1S_0 = 100$ ), enabling the LD (load) input of DR, transferring the content of DR through the adder and logic circuit into AC, and enabling the LD (load) input of AC, all during the same clock cycle.

### 3. Computer Instructions:

- The basic computer has three instruction code formats, as shown in Fig. 5-5. Each format has 16 bits.



- 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 I.
- I is equal to 0 for direct address and to 1 for indirect address.
- The register-reference instructions are recognized by the operation code 1.11 with a 0 in the leftmost bit (bit 15) of the instruction.
- A register-reference instruction specifies an operation on the AC register. So an operand from memory is not needed. Therefore, the other 12 bits are used to specify the operation to be executed.
- An input—output instruction does not need a reference to memory and is recognized by the operation code 111 with a 1 in the leftmost bit of the instruction.
- The remaining 12 bits are used to specify the type of input—output operation.
- The instructions for the computer are listed in Table 5-2.

**TABLE 5-2 Basic Computer Instructions**

Symbol	Hexadecimal code		Description
	<i>I</i> = 0	<i>I</i> = 1	
AND	0xxx	8xxx	AND memory word to <i>AC</i>
ADD	1xxx	9xxx	Add memory word to <i>AC</i>
LDA	2xxx	Axxx	Load memory word to <i>AC</i>
STA	3xxx	Bxxx	Store content of <i>AC</i> 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 <i>AC</i>
CLE	7400		Clear <i>E</i>
CMA	7200		Complement <i>AC</i>
CME	7100		Complement <i>E</i>
CIR	7080		Circulate right <i>AC</i> and <i>E</i>
CIL	7040		Circulate left <i>AC</i> and <i>E</i>
INC	7020		Increment <i>AC</i>
SPA	7010		Skip next instruction if <i>AC</i> positive
SNA	7008		Skip next instruction if <i>AC</i> negative
SZA	7004		Skip next instruction if <i>AC</i> zero
SZE	7002		Skip next instruction if <i>E</i> is 0
HLT	7001		Halt computer
INP	F800		Input character to <i>AC</i>
OUT	F400		Output character from <i>AC</i>
SKI	F200		Skip on input flag
SKO	F100		Skip on output flag
ION	F080		Interrupt on
IOF	F040		Interrupt off

- The symbol designation is a three-letter word and represents an abbreviation intended for programmers and users.
- The hexadecimal code is equal to the equivalent hexadecimal number of the binary code used for the instruction.

### Instruction Set Completeness:

- A computer should have a set of instructions so that the user can construct machine language programs to evaluate any function.
- The set of instructions are said to be complete if the computer includes a sufficient number of instructions in each of the following categories:
  - Arithmetic, logical, and shift instructions
  - Data Instructions (for moving information to and from memory and processor registers)
  - Program control or Branch
  - Input and output instructions
- There is one arithmetic instruction, ADD, and two related instructions, complement AC(CMA) and increment AC(INC). With these three instructions we can add and subtract binary numbers when negative numbers are in signed-2's complement representation.
- The circulate instructions, CIR and CIL; can be used for arithmetic shifts as well as any other type of shifts desired.
- There are three logic operations: AND, complement AC (CMA), and clear AC(CLA). The AND and complement provide a NAND operation.
- Moving information from memory to AC is accomplished with the load AC (LDA) instruction. Storing information from AC into memory is done with the store AC (STA) instruction.
- The branch instructions BUN, BSA, and ISZ, together with the four skip instructions, provide capabilities for program control and checking of status conditions.
- The input (INP) and output (OUT) instructions cause information to be transferred between the computer and external devices.

### 4. Timing and Control:

- The timing for all registers in the basic computer is controlled by a master clock generator.
- The clock pulses are applied to all flip-flops and registers in the system, including the flip-flops and registers in the control unit.
- The clock pulses do not change the state of a register unless the register is enabled by a control signal.
- The control signals are generated in the control unit and provide control inputs for the multiplexers in the common bus, control inputs in processor registers, and microoperations for the accumulator.
- There are two major types of control organization:
  - *Hardwired control*
  - *Microprogrammed control*
- The differences between hardwired and microprogrammed control are

Hardwired control	Microprogrammed control
✓ The control logic is implemented with gates, flip-flops, decoders, and other digital circuits.	✓ The control information is stored in a control memory. The control memory is programmed to initiate the required sequence of microoperations.
✓ The advantage that it can be optimized to produce a fast mode of operation.	✓ Compared with the hardwired control operation is slow.
✓ Requires changes in the wiring among the various components if the design has to be modified or changed.	✓ Required changes or modifications can be done by updating the microprogram in control memory.



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

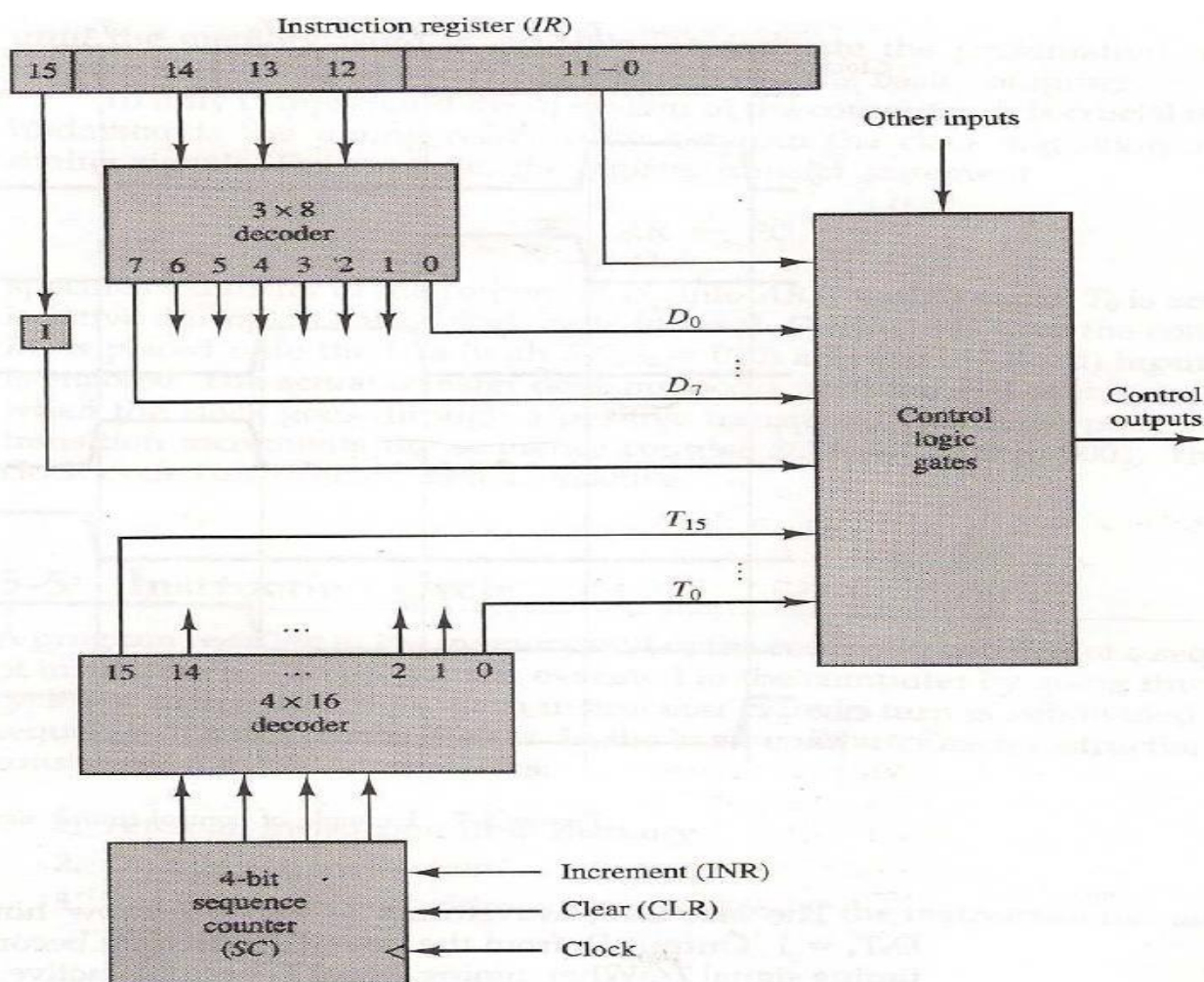


Figure 5-6 Control unit of basic computer.

- The outputs of the counter are decoded into 16 timing signals  $T_0$  through  $T_{15}$ .
- The sequence counter SC can be incremented or cleared synchronously.
- The counter is incremented to provide the sequence of timing signals out of the 4 x 16 decoder.
- As an example, consider the case where SC is incremented to provide timing signals  $T_0$ ,  $T_1$ ,  $T_2$ ,  $T_3$  and  $T_4$  in sequence. At time  $T_4$ , SC is cleared to 0 if decoder output  $D_3$  is active.
- This is expressed symbolically by the statement

$$D_3T_4: SC \leftarrow 0$$

- The timing diagram of Fig. 5-7 shows the time relationship of the control signals.

- The sequence counter *SC* responds to the positive transition of the clock.
- Initially, the CLR input of *SC* is active. The first positive transition of the clock clears *SC* to 0, which in turn activates the timing signal  $T_0$  out of the decoder.  $T_0$  is active during one clock cycle.
- *SC* is incremented with every positive clock transition, unless its CLR input is active.
- This produces the sequence of timing signals  $T_0, T_1, T_2, T_3, T_4$  and so on, as shown in the diagram.
- The last three waveforms in Fig.5-7 show how *SC* is cleared when  $D_3T_4 = 1$ .
- Output  $D_3$  from the operation decoder becomes active at the end of timing signal  $T_2$ .
- When timing signal  $T_4$  becomes active, the output of the AND gate that implements the control function  $D_3T_4$  becomes active.
- This signal is applied to the CLR input of *SC*. On the next positive clock transition (the one marked  $T_4$  in the diagram) the counter is cleared to 0.
- This causes the timing signal  $T_0$  to become active instead of  $T_5$  that would have been active if *SC* were incremented instead of cleared.

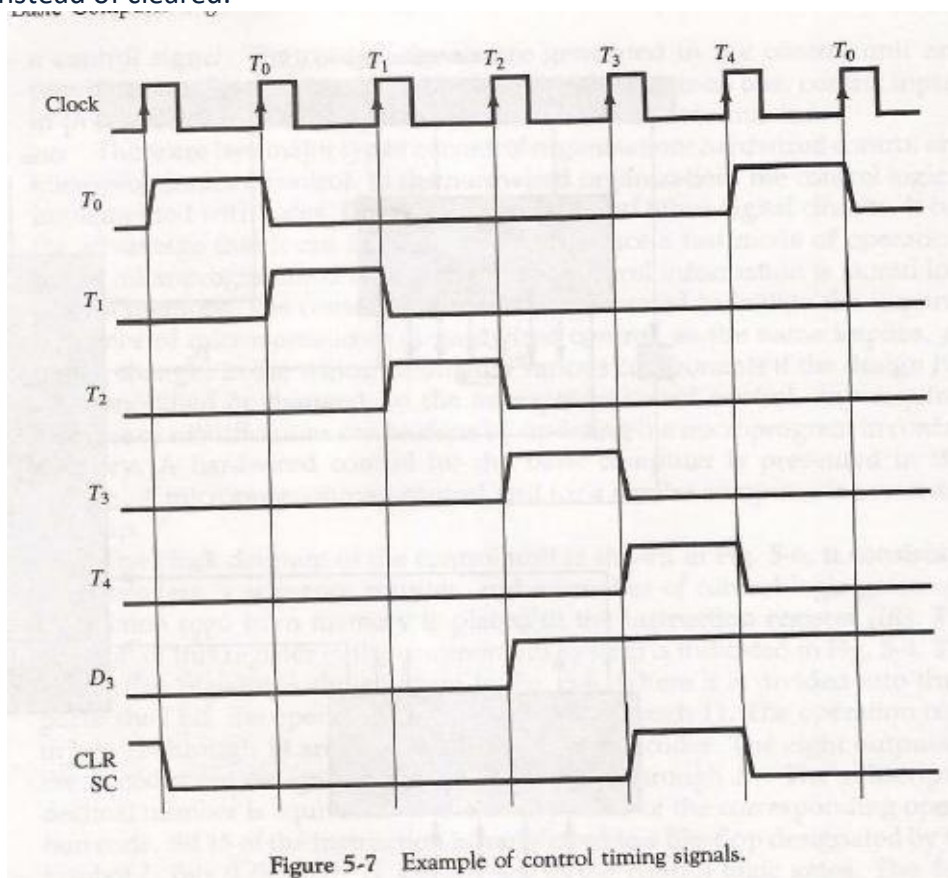


Figure 5-7 Example of control timing signals.

## 5. Instruction Cycle:

- A program residing in the memory unit of the computer consists of a sequence of instructions.
- The program is executed in the computer by going through a cycle for each instruction.
- Each instruction cycle in turn is subdivided into a sequence of sub cycles or phases.
- In the basic computer each instruction cycle consists of the following phases:
  1. Fetch an instruction from memory.
  2. Decode the instruction.
  3. Read the effective address from memory if the instruction has an indirect address.
  4. Execute the instruction.
- Upon the completion of step 4, the control goes back to step 1 to fetch, decode, and execute the next instruction.

## Fetch and Decode:

- Initially, the program counter PC is loaded with the address of the first instruction in the program.
- The sequence counter SC is cleared to 0, providing a decoded timing signal  $T_0$ .
- The microoperations for the fetch and decode phases can be specified by the following register transfer statements.

$T_0: AR \leftarrow PC$

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

$T_2: D_0, \dots, D_7 \leftarrow \text{Decode } IR(12-14), AR \leftarrow IR(0-11), I \leftarrow IR(15)$

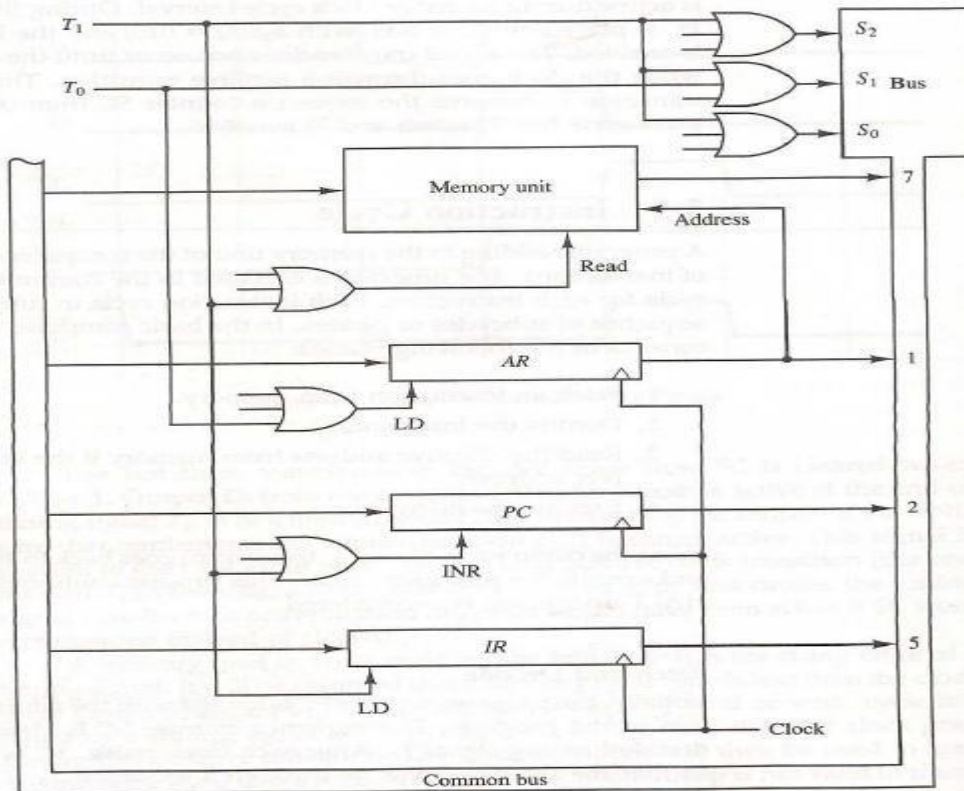


Figure 5-8 Register transfers for the fetch phase.

- Figure 5-8 shows how the first two register transfer statements are implemented in the bus system.
- To provide the data path for the transfer of  $PC$  to  $AR$  we must apply timing signal  $T_0$  to achieve the following connection:
  - Place the content of  $PC$  onto the bus by making the bus selection inputs  $S_2, S_1, S_0$  equal to 010.
  - Transfer the content of the bus to  $AR$  by enabling the  $LD$  input of  $AR$ .
- In order to implement the second statement it is necessary to use timing signal  $T_1$  to provide the following connections in the bus system.
  - Enable the read input of memory.
  - Place the content of memory onto the bus by making  $S_2S_1S_0=111$ .
  - Transfer the content of the bus to  $IR$  by enabling the  $LD$  input of  $IR$ .
  - Increment  $PC$  by enabling the  $INR$  input of  $PC$ .
- Multiple input OR gates are included in the diagram because there are other control functions that will initiate similar operations.

### Determine the Type of Instruction:

- The timing signal that is active after the decoding is  $T_3$ .
- During time  $T_3$ , the control unit determine the type of instruction that was read from the memory.
- The flowchart of fig.5-9 shows the initial configurations for the instruction cycle and also how the control determines the instruction cycle type after the decoding.
- Decoder output  $D_7$  is equal to 1 if the operation code is equal to binary 111.
- If  $D_7=1$ , the instruction must be a register-reference or input-output type.
- If  $D_7 = 0$ , the operation code must be one of the other seven values 000 through 110, specifying a memory-reference instruction.
- Control then inspects the value of the first bit of the instruction, which is now available in flip-flop I.
- If  $D_7 = 0$  and  $I = 1$ , indicates a memory-reference instruction with an indirect address. So it is then necessary to read the effective address from memory.
- If  $D_7 = 0$  and  $I = 0$ , indicates a memory-reference instruction with a direct address.
- If  $D_7 = 1$  and  $I = 0$ , indicates a register-reference instruction.
- If  $D_7 = 01$  and  $I = 1$ , indicates an input-output instruction.
- The three instruction types are subdivided into four separate paths.
- The selected operation is activated with the clock transition associated with timing signal  $T_3$ .
- This can be symbolized as follows:



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

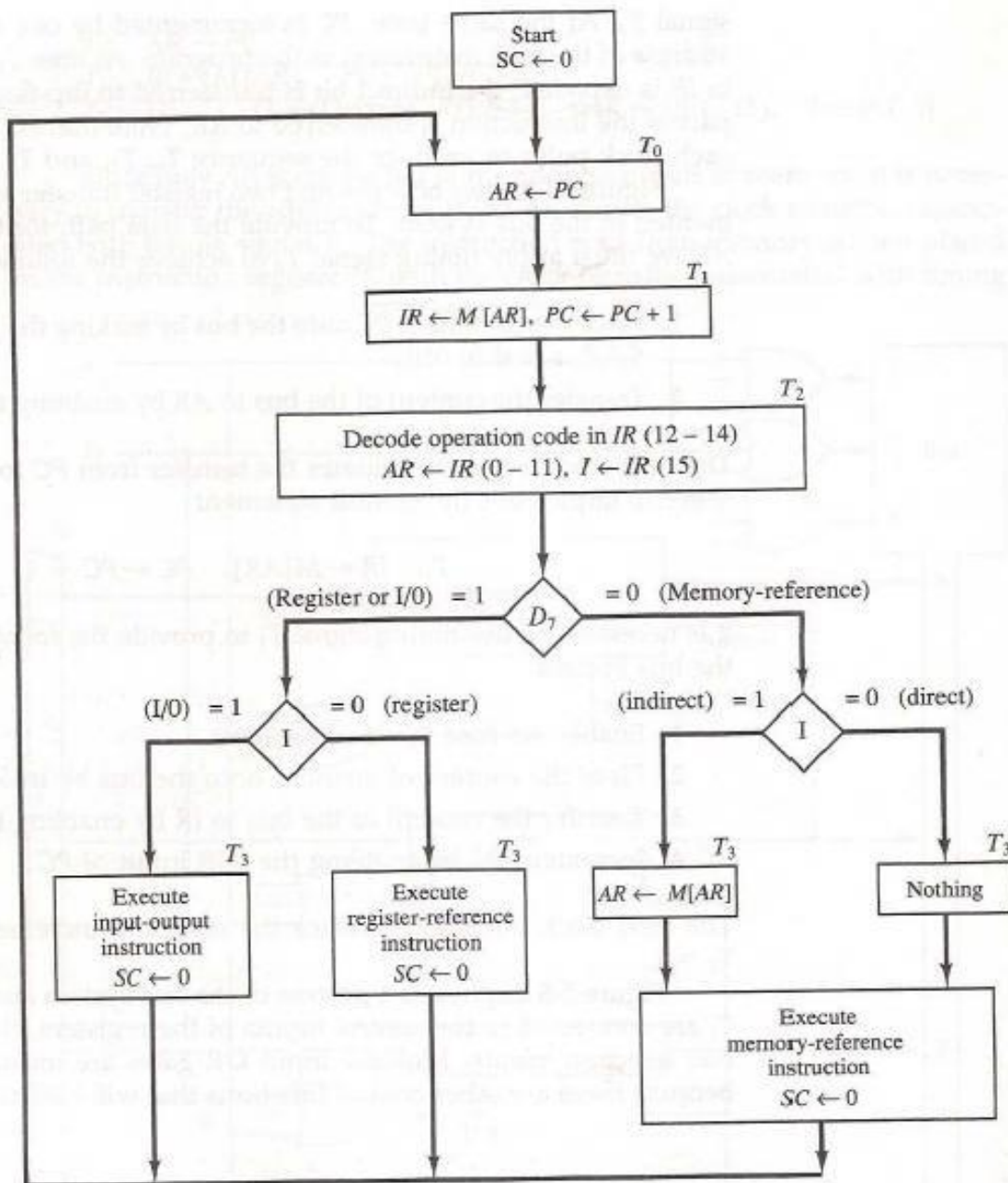


Figure 5-9 Flowchart for instruction cycle (initial configuration).

### Register-Reference Instructions:

- Register-reference instructions are recognized by the control when  $D_7 = 1$  and  $I = 0$ .
- These instructions use bits 0 through 11 of the instruction code to specify one of 12 instructions.
- These 12 bits are available in IR (0-11).
- The control functions and microoperations for the register-reference instructions are listed in Table 5-3.
- These instructions are executed with the clock transition associated with timing variable  $T_3$ .
- Control function needs the Boolean relation  $D_7 I' T_3$ , which we designate for convenience by the symbol  $r$ .

By assigning the symbol  $B_i$  to bit  $i$  of  $IR$ , all control functions can be simply denoted by  $rB_i$ .

TABLE 5-3 Execution of Register-Reference Instructions

$D_7I'T_3 = r$ (common to all register-reference instructions)		
$IR(i) = B_i$ [bit in $IR(0-11)$ that specifies the operation]		
	$r$ :	$SC \leftarrow 0$
CLA	$rB_{11}$ :	$AC \leftarrow 0$
CLE	$rB_{10}$ :	$E \leftarrow 0$
CMA	$rB_9$ :	$AC \leftarrow \overline{AC}$
CME	$rB_8$ :	$E \leftarrow \overline{E}$
CIR	$rB_7$ :	$AC \leftarrow \text{shr } AC, AC(15) \leftarrow E, E \leftarrow AC(0)$
CIL	$rB_6$ :	$AC \leftarrow \text{shl } AC, AC(0) \leftarrow E, E \leftarrow AC(15)$
INC	$rB_5$ :	$AC \leftarrow AC + 1$
SPA	$rB_4$ :	If $(AC(15) = 0)$ then $(PC \leftarrow PC + 1)$
SNA	$rB_3$ :	If $(AC(15) = 1)$ then $(PC \leftarrow PC + 1)$
SZA	$rB_2$ :	If $(AC = 0)$ then $PC \leftarrow PC + 1$
SZE	$rB_1$ :	If $(E = 0)$ then $(PC \leftarrow PC + 1)$
HLT	$rB_0$ :	$S \leftarrow 0$ ( $S$ is a start-stop flip-flop)
		Clear $SC$
		Clear $AC$
		Clear $E$
		Complement $AC$
		Complement $E$
		Circulate right
		Circulate left
		Increment $AC$
		Skip if positive
		Skip if negative
		Skip if $AC$ zero
		Skip if $E$ zero
		Halt computer

- For example, the instruction CLA has the hexadecimal code 7800, which gives the binary equivalent 0111 1000 0000 0000. The first bit is a zero and is equivalent to  $I'$ .
- The next three bits constitute the operation code and are recognized from decoder output  $D_7$ .
- Bit 11 in  $IR$  is 1 and is recognized from  $B_{11}$ . The control function that initiates the microoperation for this instruction is  $D_7I'T_3 B_{11} = rB_{11}$ .
- The execution of a register-reference instruction is completed at time  $T_3$ .
- The sequence counter  $SC$  is cleared to 0 and the control goes back to fetch the next instruction with timing signal  $T_0$ .
- The first seven register-reference instructions perform clear, complement, circular shift, and increment microoperations on the  $AC$  or  $E$  registers.
- The next four instructions cause a skip of the next instruction in sequence when a stated condition is satisfied. The skipping of the instruction is achieved by incrementing  $PC$  once again.
- The condition control statements must be recognized as part of the control conditions.
- The  $AC$  is positive when the sign bit in  $AC(15) = 0$ ; it is negative when  $AC(15) = 1$ . The content of  $AC$  is zero ( $AC = 0$ ) if all the flip-flops of the register are zero.

The HLT instruction clears a start-stop flip-flop  $S$  and stops the sequence counter from counting.

## 6. Memory-Reference Instructions:

- Table 5-4 lists the seven memory-reference instructions.
- The decoded output  $D_i$  for  $i = 0, 1, 2, 3, 4, 5$ , and 6 from the operation decoder that belongs to each instruction is included in the table.
- The effective address of the instruction is in the address register  $AR$  and was placed there during timing signal  $T_2$  when  $I = 0$ , or during timing signal  $T_3$  when  $I = 1$ .
- The execution of the memory-reference instructions starts with timing signal  $T_4$ .
- The symbolic description of each instruction is specified in the table in terms of register transfer notation.

**TABLE 5-4** Memory-Reference Instructions

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

### AND to AC:

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

$$\begin{array}{ll} D_0T_4: & DR \leftarrow M[AR] \\ D_0T_5: & AC \leftarrow AC \wedge DR, \quad SC \leftarrow 0 \end{array}$$

### ADD to AC:

- This instruction adds the content of the memory word specified by the effective address to the value of AC.
- The sum is transferred into AC and the output carry  $C_{out}$  is transferred to the E (extended accumulator) flip-flop.
- The microoperations needed to execute this instruction are

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

### LDA: Load to AC

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

$$\begin{array}{ll} D_2T_4: & DR \leftarrow M[AR] \\ D_2T_5: & AC \leftarrow DR, \quad SC \leftarrow 0 \end{array}$$

### **STA:** Store AC

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

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

### **BUN:** Branch Unconditionally

- This instruction transfers the program to the instruction specified by the effective address.
- The BUN instruction allows the programmer to specify an instruction out of sequence and we say that the program branches (or jumps) unconditionally.
- The instruction is executed with one microoperation:

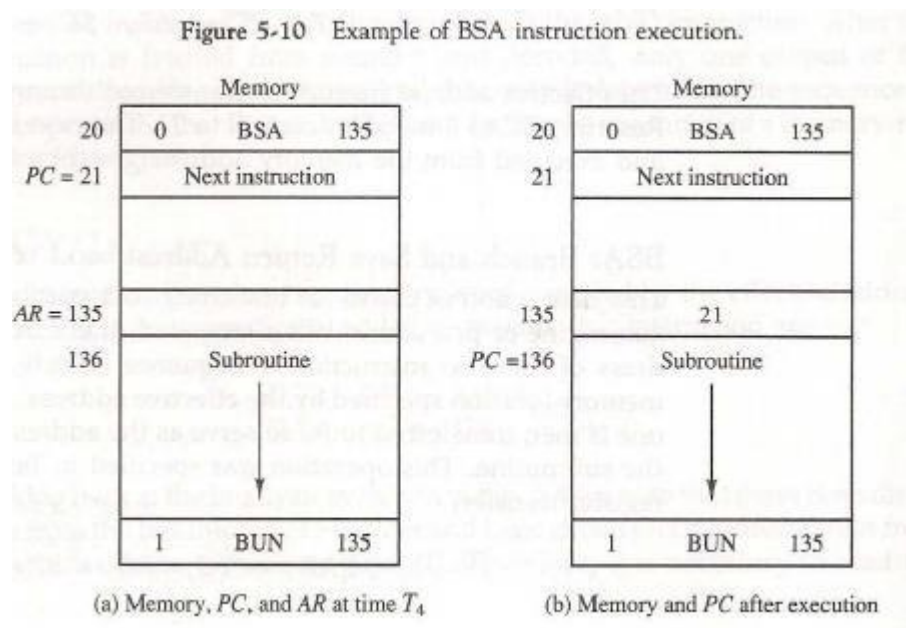
$D_4T_4: PC \leftarrow AR, SC \leftarrow 0$

### **BSA:** Branch and Save Return Address

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

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

- A numerical example that demonstrates how this instruction is used with a subroutine is shown in Fig. 5-10.





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

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

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

$$\begin{aligned} D_5T_4: & \quad M[AR] \leftarrow PC, \quad AR \leftarrow AR + 1 \\ D_5T_5: & \quad PC \leftarrow AR, \quad SC \leftarrow 0 \end{aligned}$$

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

- This instruction increments the word specified by the effective address, and if the incremented value is equal to 0, *PC* is incremented by 1 to skip the next instruction in the program.
- Since it is not possible to increment a word inside the memory, it is necessary to read the word into *DR*, increment *DR*, and store the word back into memory.
- This is done with the following sequence of microoperations:

$$\begin{aligned} D_6T_4: & \quad DR \leftarrow M[AR] \\ D_6T_5: & \quad DR \leftarrow DR + 1 \\ D_6T_6: & \quad M[AR] \leftarrow DR, \quad \text{if } (DR = 0) \text{ then } (PC \leftarrow PC + 1), \quad SC \leftarrow 0 \end{aligned}$$

### **Control Flowchart:**

- A flowchart showing all microoperations for the execution of the seven memory-reference instructions is shown in Fig. 5.11.

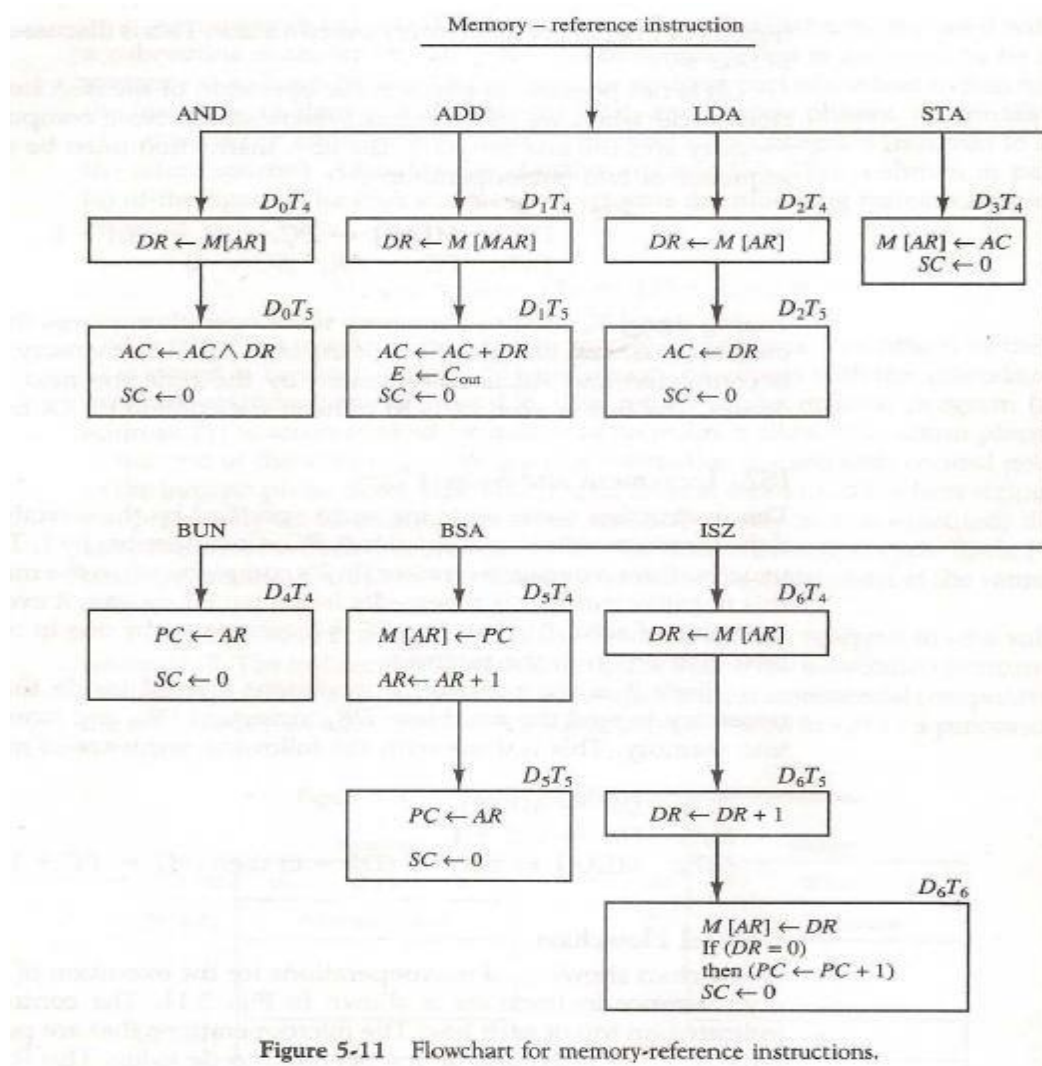


Figure 5-11 Flowchart for memory-reference instructions.

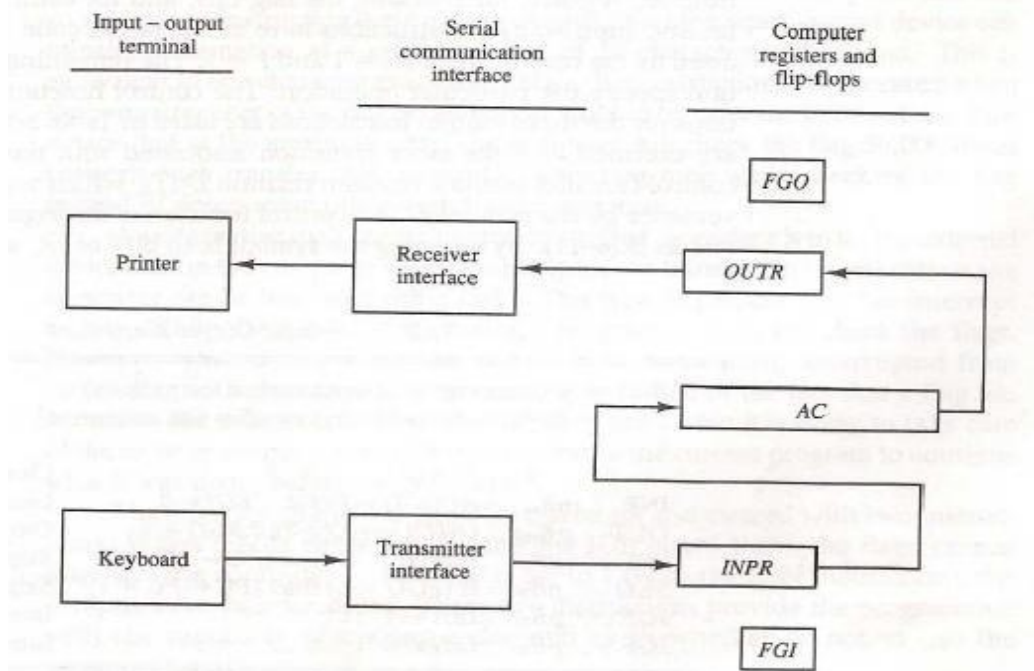
## 7. Input-Output and Interrupt:

- Instructions and data stored in memory must come from some input device.
- Computational results must be transmitted to the user through some output device.
- To demonstrate the most basic requirements for input and output communication, we will use as an illustration a terminal unit with a keyboard and printer.

### Input-Output Configuration:

- The terminal sends and receives serial information.
- Each quantity of information has eight bits of an alphanumeric code.
- The serial information from the keyboard is shifted into the input register *INPR*.
- The serial information for the printer is stored in the output register *OUTR*.
- These two registers communicate with a communication interface serially and with the AC in parallel.
- The input—output configuration is shown in Fig. 5-12.

Figure 5-12 Input-output configuration.



- The input register INPR consists of eight bits and holds alphanumeric input information.
- The 1-bit input flag *FGI* is a control flip-flop.
- The flag bit is set to 1 when new information is available in the input device and is cleared to 0 when the information is accepted by the computer.
- The output register OUTR works similarly but the direction of information flow is reversed.
- Initially, the output flag *FGO* is set to 1.
- The computer checks the flag bit; if it is 1, the information from AC is transferred in parallel to OUTR and FGO is cleared to 0.
- The output device accepts the coded information, prints the corresponding character, and when the operation is completed, it sets FGO to 1.

### Input-Output Instructions:

- Input and output instructions are needed for transferring information to and from AC register, for checking the flag bits, and for controlling the interrupt facility.
- Input-output instructions have an operation code 1111 and are recognized by the control when D7 = 1 and I = 1.
- The remaining bits of the instruction specify the particular operation.
- The control functions and microoperations for the input-output instructions are listed in Table 5-5.



TABLE 5-5 Input-Output Instructions

$D_7IT_3 = p$ (common to all input-output instructions)			
$IR(i) = B_i$ [bit in $IR(6-11)$ that specifies the instruction]			
	$p$ :	$SC \leftarrow 0$	Clear $SC$
INP	$pB_{11}$ :	$AC(0-7) \leftarrow INPR, FGI \leftarrow 0$	Input character
OUT	$pB_{10}$ :	$OUTR \leftarrow AC(0-7), FGO \leftarrow 0$	Output character
SKI	$pB_9$ :	If $(FGI = 1)$ then $(PC \leftarrow PC + 1)$	Skip on input flag
SKO	$pB_8$ :	If $(FGO = 1)$ then $(PC \leftarrow PC + 1)$	Skip on output flag
ION	$pB_7$ :	$IEN \leftarrow 1$	Interrupt enable on
IOF	$pB_6$ :	$IEN \leftarrow 0$	Interrupt enable off

- These instructions are executed with the clock transition associated with timing signal  $T_3$ .
- Each control function needs a Boolean relation  $D_7IT_3$ , which we designate for convenience by the symbol  $p$ .
- The control function is distinguished by one of the bits in  $IR(6-11)$ .
- By assigning the symbol  $B_i$  to bit  $i$  of  $IR$ , all control functions can be denoted by  $pB_i$  for  $i = 6$  through 11.
- The sequence counter  $SC$  is cleared to 0 when  $p = D_7IT_3 = 1$ .
- The last two instructions set and clear an interrupt enable flip-flop  $IEN$ .

### Program Interrupt:

- The computer keeps checking the flag bit, and when it finds it set, it initiates an information transfer.
- The difference of information flow rate between the computer and that of the input—output device makes this type of transfer inefficient.
- An alternative to the programmed controlled procedure is to let the external device inform the computer when it is ready for the transfer.
- In the meantime the computer can be busy with other tasks. This type of transfer uses the interrupt facility.
- While the computer is running a program, it does not check the flags.
- When a flag is set, the computer is momentarily interrupted from the current program.
- The computer deviates momentarily from what it is doing to perform of the input or output transfer.
- It then returns to the current program to continue what it was doing before the interrupt.
- The interrupt enable flip-flop  $IEN$  can be set and cleared with two instructions.
  - When  $IEN$  is cleared to 0 (with the IOF instruction), the flags cannot interrupt the computer.
  - When  $IEN$  is set to (with the ION instruction), the computer can be interrupted.
- The way that the interrupt is handled by the computer can be explained by means of the flowchart of Fig. 5-13.
- An interrupt flip-flop  $R$  is included in the computer. When  $R = 0$ , the computer goes through an instruction cycle.
- During the execute phase of the instruction cycle  $IEN$  is checked by the control.
- If it is 0, it indicates that the programmer does not want to use the interrupt, so control continues with the next instruction cycle.
- If  $IEN$  is 1, control checks the flag bits. If both flags are 0, it indicates that neither the input nor the output registers are ready for transfer of information. In this case, control continues with the next instruction cycle.
- If either flag is set to 1 while  $IEN = 1$ , flip-flop  $R$  is set to 1. At the end of the execute phase, control checks the value of  $R$ , and if it is equal to 1, it goes to an interrupt cycle instead of an instruction cycle.

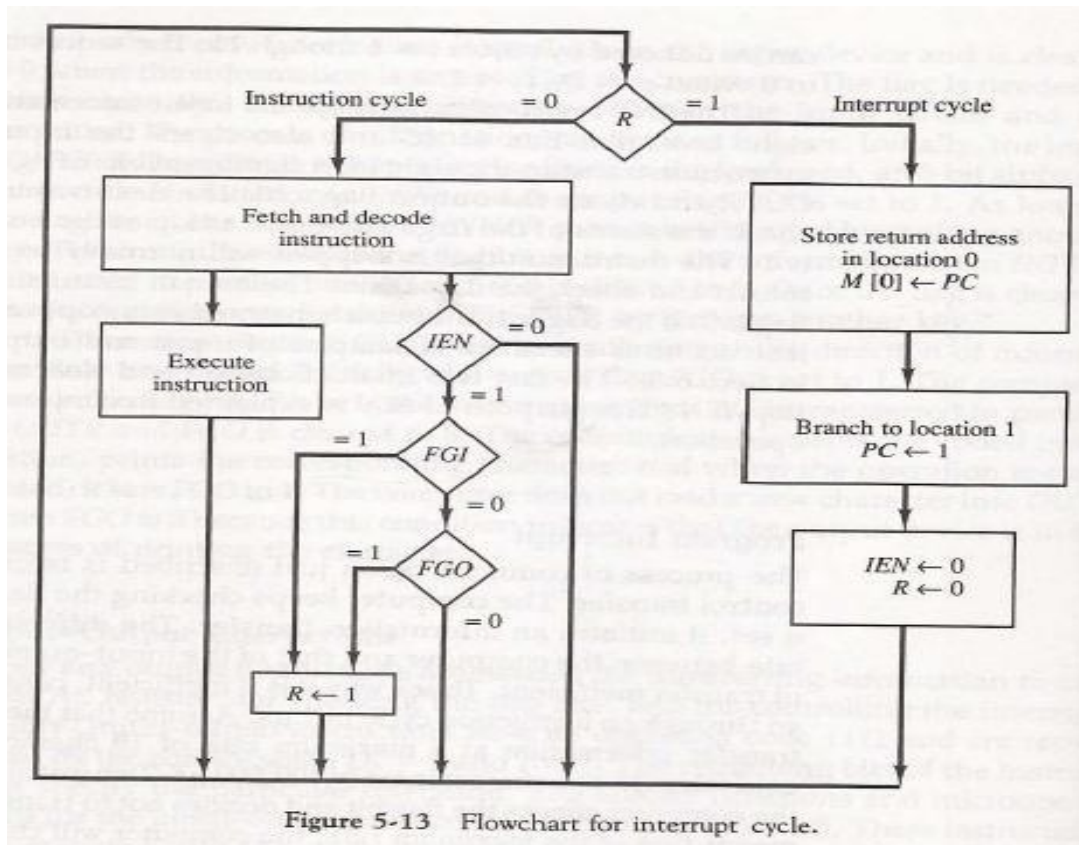
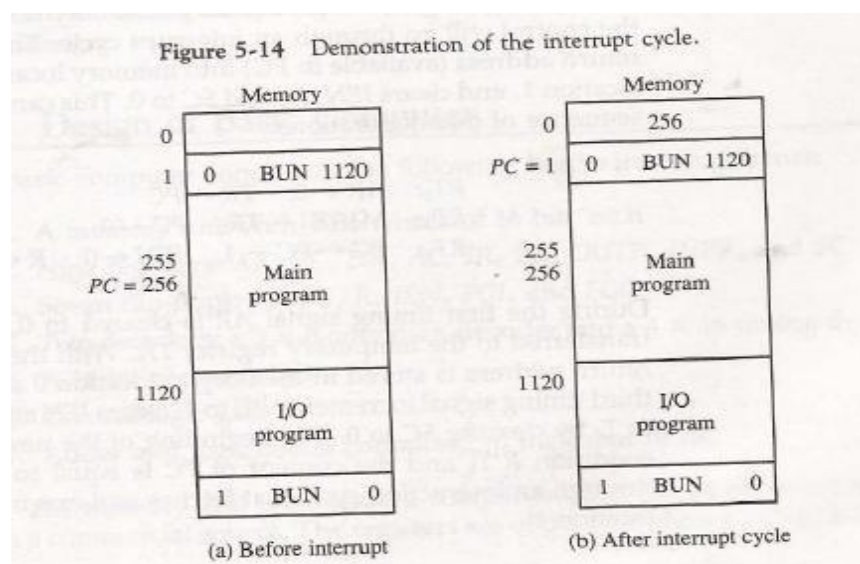


Figure 5-13 Flowchart for interrupt cycle.

### Interrupt cycle:

- The interrupt cycle is a hardware implementation of a branch and save return address operation.
- The return address available in PC is stored in a specific location.
- This location may be a processor register, a memory stack, or a specific memory location.
- An example that shows what happens during the interrupt cycle is shown in Fig. 5-14.



- When an interrupt occurs and  $R$  is set to 1 while the control is executing the instruction at address 255.
- At this time, the return address 256 is in  $PC$ .
- The programmer has previously placed an input—output service program in memory starting from address 1120 and a BUN 1120 instruction at address 1. This is shown in Fig. 5.14(a).
- When control reaches timing signal  $T_0$  and finds that  $R = 1$ , it proceeds with the interrupt cycle.
- The content of  $PC$  (256) is stored in memory location 0,  $PC$  is set to 1, and  $R$  is cleared to 0.
- The branch instruction at address 1 causes the program to transfer to the input—output service program at address 1120.
- This program checks the flags, determines which flag is set, and then transfers the required input or output information.
- Once this is done, the instruction ION is executed to set IEN to 1 (to enable further interrupts), and the program returns to the location where it was interrupted.
- This is shown in Fig. 5-14(b).

## UNIT-II

### MICRO PROGRAMMED CONTROL

#### SYLLABUS:

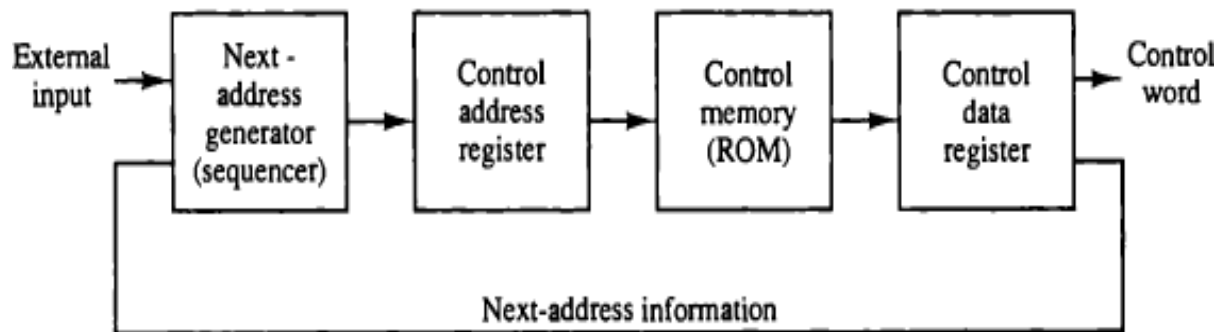
**MICRO PROGRAMMED CONTROL: Control Memory, Address Sequencing, Micro program Example, Design of Control Unit.**

**Central Processing Unit: General Register Organization, Instruction Formats, Addressing modes, Data Transfer and Manipulation, Program Control.**

#### CONTROL MEMORY:

- The function of the control unit in a digital computer is to initiate sequence of micro operation.
- When the control signals are generated by hardware using conventional logic design techniques, the control unit is said to be **“Hardwired”**.
- A control unit whose binary control variables (1's and 0's) are stored in memory is called a **“Micro programmed control unit”**. Each word in control memory contains within it a micro instruction.
- The micro instruction specifies **one or more** micro operations for the system. A sequence of micro instructions constitutes a micro program.
- A computer that employs a micro programmed control unit will have two separate memories. A **main memory** and **control memory**.
- The **main memory** is available to the user for storing the programs. The contents of main memory may alter when the data are manipulated and every time that the program is changed. The user's program in main memory consists of machine instructions and data.
- The control memory holds a fixed micro program that cannot alter by the occasional user. The micro program consists of micro instruction that specify various internal control signals for execution of register micro operations.
- Each machine instruction imitates a series of micro instruction in control memory. These micro instructions generate the micro operation to **fetch the instruction from main memory**, to **evaluate the effective address**, to **execute the operation specified by the instruction** and to **return control to the fetch phase** in order to repeat the cycle for the next instruction.

Figure 7-1 Microprogrammed control organization.



- In the figure, the control memory is assumed to be **ROM**, within which all control information is permanently stored.
- The **control memory address register** specifies the address of the micro instruction.
- The **control data register** (“**pipeline register**”) holds the micro instruction and read from memory.
- The micro instruction contains a control word that specifies one or more micro operations for the data processor. Once these operations are executed, the control must determine the next address.
- The location of the next micro instruction may be the one next in sequence, or it may be located somewhere else in the control memory
- The next address generated is sometimes called an **micro program sequencer**, as it determines the **address sequence** that is read from control memory.
- The function of a micro program sequencer are
  - ❖ Loading an initial address to start the operations.
  - ❖ Incrementing the control address register by one.
  - ❖ Loading into the control address register an address from control memory.
  - ❖ Transferring an external address

## ADDRESS SEQUENCING:

- Micro instructions are stored in control memory in groups, with each group specifying a “routine”. Each computer instruction has its own micro program routine in control memory to generate the micro operations that execute the instruction.
- Once the required routine is reached, the micro instruction that executes the instruction may be sequenced by incrementing the control address register.
- When the execution of the instruction is completed, control must return to the fetch routine.

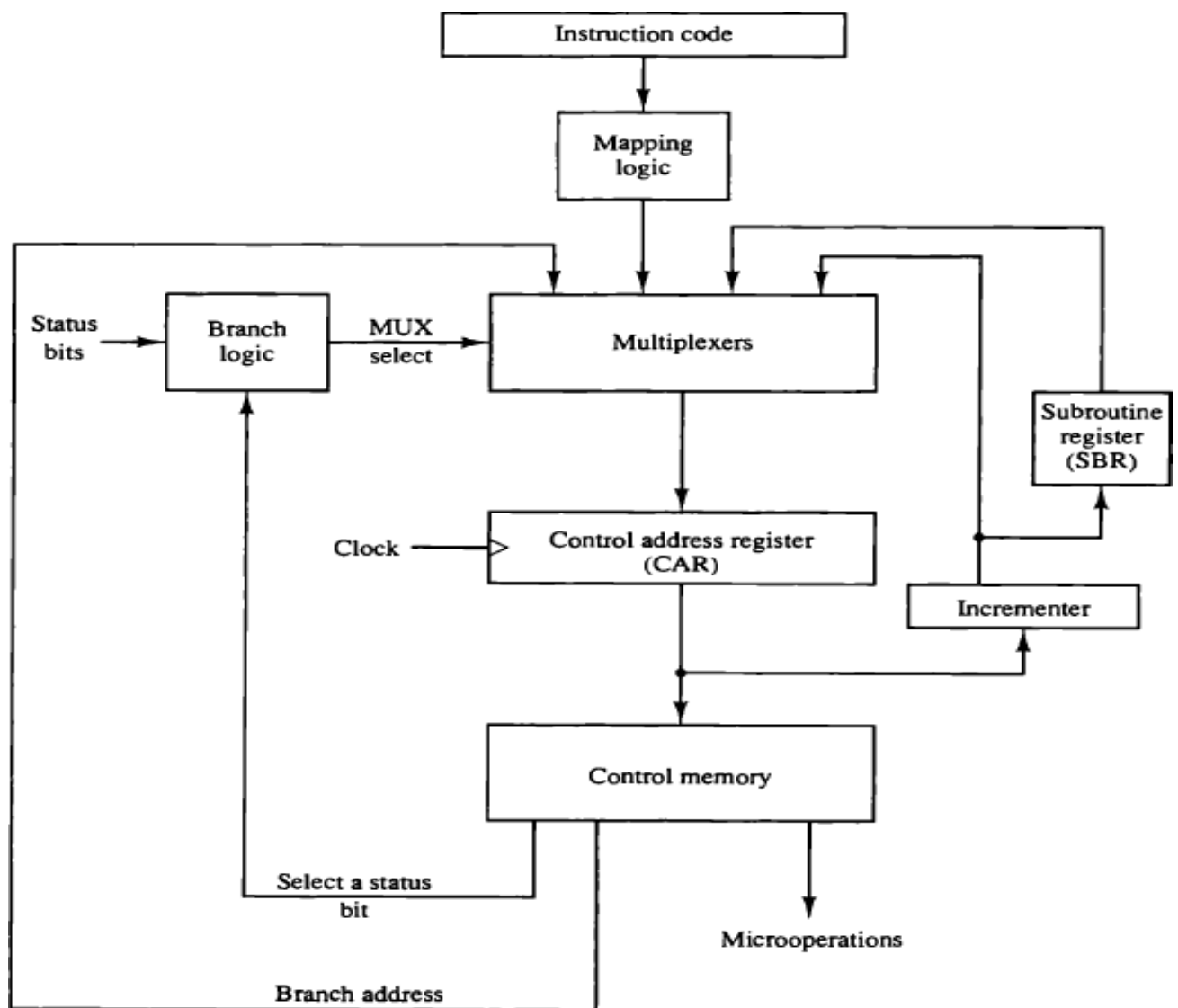


Figure 7-2 Selection of address for control memory.



- The micro instruction in control memory contains a set of bits to initiate micro operation in computer registers and other bits to specify the method by which the next address is obtained.

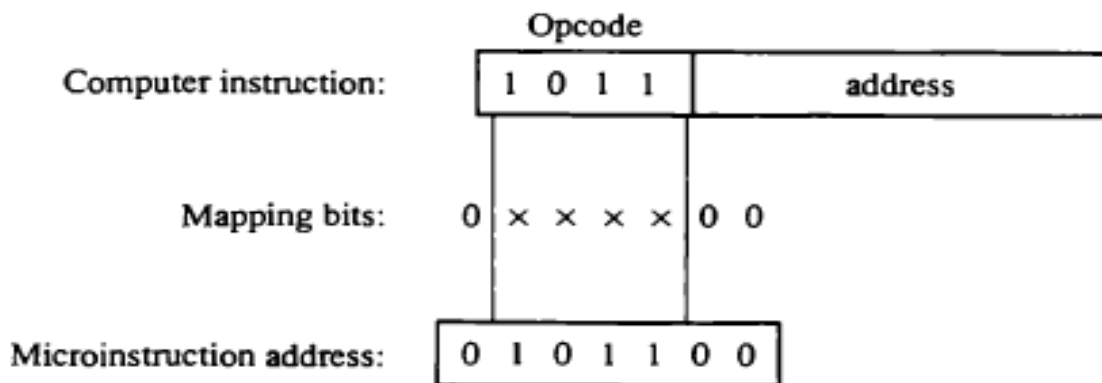
The address sequence capabilities required in a control memory are:

1. A **mapping process** from the bits of the instruction to an address for control memory.
  2. **Incrementing** of the control address Register
  3. **Unconditional branch or conditional branch**, depending on states bit conditions.
  4. A facility for **subroutine call and return**.
- 
- The diagram shows **4** different paths from which the control address register (CAR) receives the address.
    1. A **mapping procedure** is a rule that transforms the instruction code into a control memory address.
    2. The **incrementer** increments the contents of the control address register by one to select the next micro instruction in sequence.
    3. **Branching** is achieved by specifying the branch address in one of the fields of the micro instruction.
    4. Micro program that use subroutines must have a provision for storing the return address in a SBR during a **subroutine call** and restoring the address during a subroutine return.
  - The status conditions are special bits in the system that provides parameter information such as the carry out of an adder, the sign bit of a number, the mode bits of an instruction, and input or output status conditions.
  - The branch logic hardware may be implemented in a variety of ways. The simplest way is to test the specified condition and branch to the indicated address if the condition is met; otherwise the address register is incremented. This can be implemented with a multiplexer.
  - A 1 output in the multiplexer generates a control signal to transfer the branch address from the micro instruction into the CAR.
  - A 0 output in the multiplexer causes the address register to be incremented.

### Mapping of instruction:

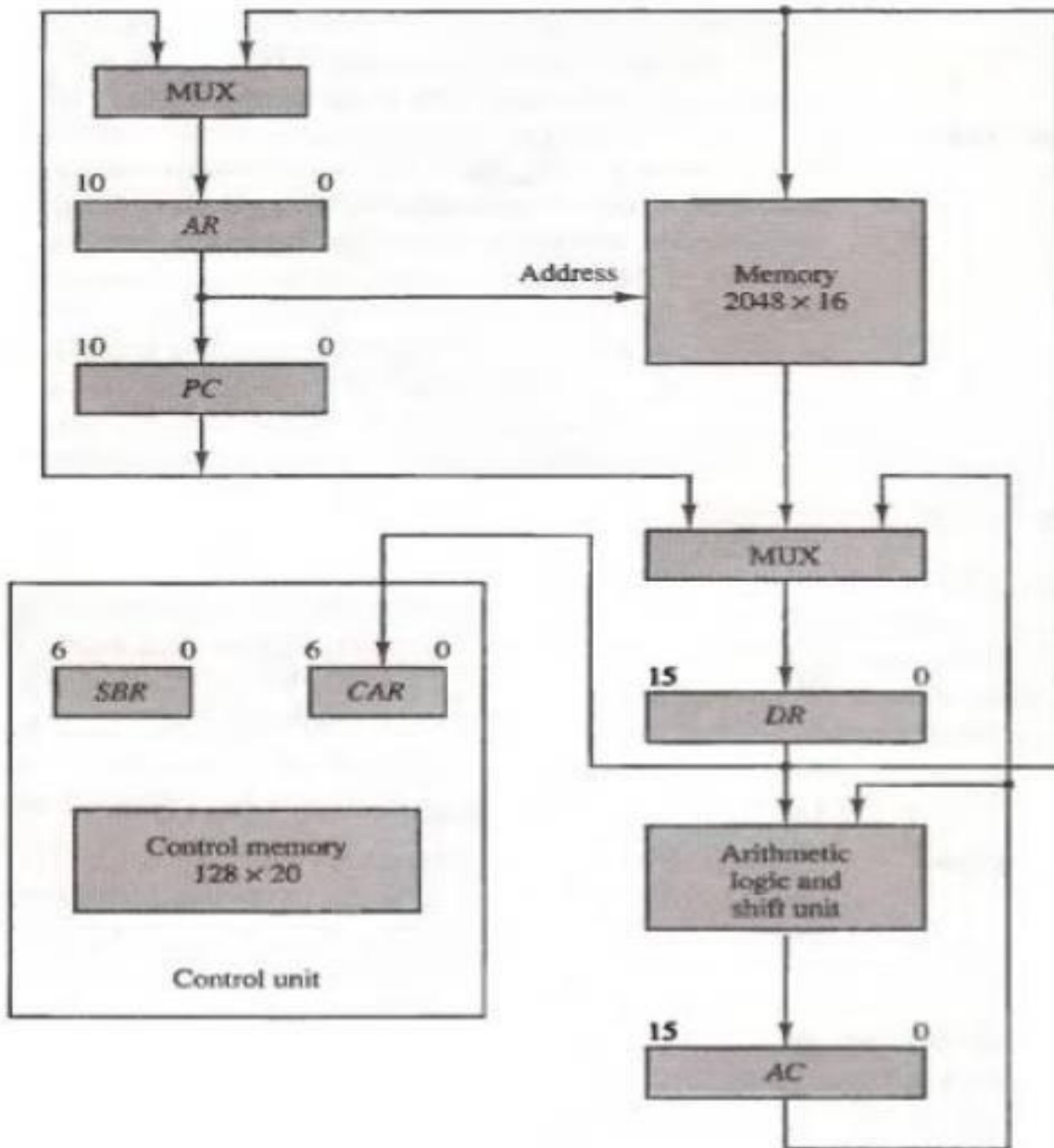
- The transformation from the instruction code bits to an address in control memory where the routine is located is referred to as a **mapping process**.
- The figure has an operation code of 4 bits which can specify up to 16 distinct instructions. Assume further that the control memory has 128 words, requiring an address of 7 bits. For each operation code there exists a micro program routine in control memory that executes the instruction.
- One simple mapping process that converts the 4 bit operation code to a 7 bit address for control memory. The mapping consists of placing a 0 in the MSB of the address, transferring the 4 opcode bits, and clearing the two LSB of the CAR.

**Figure 7-3 Mapping from instruction code to microinstruction address.**



### MICROPROGRAM EXAMPLE:

- Once the configuration of computer and its micro programmed control unit is established, the designers task is to generate the micro code for the control memory.



**Figure 7-4** Computer hardware configuration.

## COMPUTER CONFIGURATION:

- The diagram consists of two memory units, a main memory for storing instructions and data, and a control memory for storing the micro program.
- Four registers are associated with the processor unit and two with the control unit. The processor registers are **PC, AR, DR, and AC**. The control unit has **CAR** and **SBR**.
- The transfer of information among the registers in the processors is done through multiplexers rather than a common bus.
  - ❖ DR can receive information from AC, PC or memory.
  - ❖ AR can receive information from PC or DR.
  - ❖ PC can receive information from only AR.
- The arithmetic, logic, and shift unit performs micro operations with data from AC and DR and places the result in AC. Memory receives its address from AR. Input data written to memory come from DR, data read from memory can go only to DR.
- The instruction format consists of three fields.
- I-indirect address, opcode-operation code.

**Figure 7-5 Computer instructions.**



**(a) Instruction format**

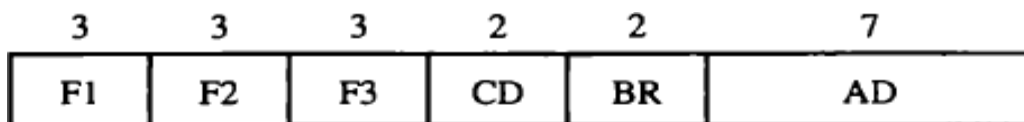
Symbol	Opcode	Description
ADD	0000	$AC \leftarrow AC + M[EA]$
BRANCH	0001	If $(AC < 0)$ then $(PC \leftarrow EA)$
STORE	0010	$M[EA] \leftarrow AC$
EXCHANGE	0011	$AC \leftarrow M[EA], M[EA] \leftarrow AC$

**EA is the effective address**

**(b) Four computer instructions**

- The **ADD** instruction adds the content of the operand found in the effective address to the content of AC.
- The **BRANCH** instruction causes a branch to the effective address, if the operand in AC is negative. The program proceeds with the next consecutive instruction if AC is not negative
- The **STORE** instruction transfers the content of AC into the memory word specified by the effective address.
- The **EXCHANGE** instruction swaps the data between AC and memory word specified by the effective address.

#### MICRO INSTRUCTION FORMAT:



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

**CD: Condition for branching**

**BR: Branch field**

**AD: Address field**

**Figure 7-6 Microinstruction code format (20 bits).**

- The address field is 7 bits wide since the control memory has  $128 = 2^7$  words.
- The micro operations are sub divided into three fields of three bits each. The three bits in each field are encoded to specify 7 distinct micro operations. This gives total of 21 micro operations.
- No more than 3 micro operations can be chosen for microinstruction, one from each field. If fewer than three micro operations are used, one or more of the fields will use the binary code 000 for no operation.
- A micro instruction can specify two simultaneous micro operations from f2 and f3 and none from f1.

$DR \leftarrow M[AR]$                       with f2=100

$PC \leftarrow PC+1$                       with f3=101

The nine bits of the micro operation fields will then be 000 100 101.

**TABLE 7-1** Symbols and Binary Code for Microinstruction Fields

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

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

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



- It is important to realize that two or more conflicting micro operations cannot be specified simultaneously.

**Ex:** a micro operation field **010 001 000** has no meaning, because it specifies the operation to **clear AC to 0** and **subtract DR from AC** at the same time.

**Note:** all transfer type micro operation symbol uses 5 letters. The first two letters designates the source register, the third letter is always a T, and the last 2 letter designates the designation of the register.

**Ex:** The micro operation that specifies the transfer  $AC \leftarrow DR$  (F1=100) has the symbol DRTAC, which stands for a transfer from DR to AC.

**CONDITION FIELD:** It consists of two bits which are encoded to specify four status bit conditions.

- The first condition is always 1, so that a reference to CD=00 (or the symbol **U**) will always find the condition to be true. When this condition is used in conjunction with BR field it provides an unconditional branch operation.
- The **indirect bit I** is available from bit 15 of DR after an instruction is read from memory.
- The **sign bit S** of AC provides the next status bit.
- The zero value, symbolized by **Z**, is a binary variable whose value equal to 1 if all bits in AC are equal to zero.
- We will use the symbols **U , I , S , Z** for the four status bits when we write micro program in symbolic form.

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

**BRANCH FIELD:** it is used in conjunction with the address field AD, to choose the address of the next micro instruction.

- When BR=00, the control program jump (jmp) operation (which is similar to branch) and when BR=01, it performs a call to sub routine (call) operation.
- The two operations are identical except that a call micro instruction stores the return address in the sub routine register SBR.
- The jump and call operations depends on the value of CD field. If the status bit conditions specified the CD field is equal to 1, the next address in the AD field is transferred to control address register CAR. Otherwise, CAR is incremented by 1.
- The return from sub routine is accomplished with a BR field equal to 10. This causes the transfer of the return address from SBR to CAR.
- The mapping from the operation code bits of the instruction to an address for CAR is accomplished when the BR field is equal to 11.

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

#### Symbolic micro instructions:

- ➔ Each line of the assembly language micro program defines a symbolic micro instruction. Each symbolic micro instruction is divided into **five** fields. Label, micro operations, CD, BR and AD.
1. **Label:** The Label field may be empty or it may specify a symbolic address. A Label is terminated with a colon (:).

2. **Micro operations:** This field consists of one, two, or three symbols, separated by commas.  
There may be no more than one symbol from each F field. The NOP is used when the micro instruction has no micro operations. This will be translated by the assembler to nine zeros.
3. **CD:** The CD field has one of the letters U, I, S or Z.
4. **BR:** The BR field contains one of the 4 symbols (JMP, CALL, RET, MAP).
5. **AD:** The AD field specifies a value for the address field of the micro instruction in one of three possible ways.
  - (a) With a symbolic address, which must also appear as a label.
  - (b) With the symbol **NEXT** to designate the next address in sequence.
  - (c) When the BR field contains a **RET** or **MAP** symbol, the AD field is left empty and is converted to seven zeros by the assembler.

**ORG:** ORG defines the origin, or first address of a micro program routine.

The symbol ORG 64 informs the assembler to place the next micro instruction in control memory at decimal address 64, which is equivalent binary address 1000000.

### **Fetch Routine:**

→ The control memory has 128 words, and each word contains 20 bits. To micro program the control memory, it is necessary to determine the bit values of each of the 128 words. The first 64 words (address 0 to 63) are to be occupied by the routines for the 16 instructions. The last 64 words may be used for any other purpose. A convenient starting location for the fetch routine is address 64.

The micro instructions needed for the fetch routine are

$$AR \leftarrow PC$$

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

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

The address of the instruction is transferred from PC to AR and the instruction is then read from memory into DR. since no instruction register is available, the instruction code remains in DR .The address part is transferred to AR and then control is transferred to one of 16 routines by mapping the operation code part of the instruction from DR into CAR. The fetch routine needs three micro instructions, which are placed in control memory at addresses 64, 65, 66.

```

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

```

The translation of the symbolic micro program to binary produces the following binary micro program.

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

#### Symbolic micro program:

- If the instruction is an ADD instruction whose operation code is 0000, the MAP micro instruction will transfer to CAR the address 0000000, which is the start address for the ADD routine in control memory.
- The first address for the BRANCH and STORE routines are 0 000100 (Decimal 4) and 0 0010 00 (Decimal 8), respectively. The first address for the other 13 routines are at address values 12, 16, 20,..... 60. This gives four words in control memory for each routine.
- The first micro instruction in the ADD routine calls subroutine INDRCT, Conditioned on states bit I. If I=1, a branch to INDRCT occurs and the return address is stored in SBR. The INDRCT has two micro instructions.

INDRCT:	READ	U	JMP	NEXT
	DRTAR	U	RET	

#### Indirect Address:

It considers the address part of the instruction as the address is stored rather than the address of the operand. The memory has to be accessed to get the effective address, which is then transferred to AR. The problem from subroutine (RET) transfers the address from SBR to CAR, thus returning to the second micro instruction of the ADD routine.

**TABLE 7-2** Symbolic Microprogram (Partial)

Label	Microoperations	CD	BR	AD
ADD:	ORG 0			
	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ADD	U	JMP	FETCH
BRANCH:	ORG 4			
	NOP	S	JMP	OVER
	NOP	U	JMP	FETCH
	OVER:	I	CALL	INDRCT
STORE:	ARTPC	U	JMP	FETCH
	ORG 8			
	NOP	I	CALL	INDRCT
	ACTDR	U	JMP	NEXT
EXCHANGE:	WRITE	U	JMP	FETCH
	ORG 12			
	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
INDRCT:	ACTDR, DRTAC	U	JMP	NEXT
	WRITE	U	JMP	FETCH
	ORG 64			
	PCTAR	U	JMP	NEXT
INDRCT:	READ, INCPC	U	JMP	NEXT
	DRTAR	U	MAP	
	READ	U	JMP	NEXT
	DRTAR	U	RET	

**ADD:** The execution of the ADD instruction is carried out by the micro instruction at the addresses 1 and 2. The first micro instruction reads the operand from memory into DR. The second micro instruction performs an ADD micro operation with the content of DR and AC and then jumps back to the beginning of the fetch routine.

**BRANCH:** The branch instruction should cause a branch to the effective address if  $AC < 0$ . The AC will be less than zero if its sign is negative, which is detected from status bit S being a 1.

- If s is equal to 1, the first JMP micro instruction transfers control to location OVER. The micro instruction at this location calls the INDRCT subroutine if  $I=1$ .
- The effective address is then transferred from AR to PC and the micro program jumps back to the fetch routine.

**STORE:** The store routine again uses the INDRCT subroutine if  $I=1$ . The content of AC is transferred into DR. A memory write operation is initiated to store the content of DR in a location specified by the effective address.

**EXCHANGE:** The exchange routine reads the operand from the effective address and places it in DR. The contents of DR and AC are interchanged in the third micro instruction. The original content of AC that is now in DR is stored back in the memory.

**Note:** Address 3 is not used. We have to specify all 0's in the word. Since, this location will never be used. The control will be jump to address which is the beginning of the fetch routine.



**TABLE 7-3** Binary Microprogram for Control Memory (Partial)

Micro Routine	Address		Binary Microinstruction					
	Decimal	Binary	F1	F2	F3	CD	BR	AD
ADD	0	0000000	000	000	000	01	01	1000011
	1	0000001	000	100	000	00	00	0000010
	2	0000010	001	000	000	00	00	1000000
	3	0000011	000	000	000	00	00	1000000
BRANCH	4	0000100	000	000	000	10	00	0000110
	5	0000101	000	000	000	00	00	1000000
	6	0000110	000	000	000	01	01	1000011
	7	0000111	000	000	110	00	00	1000000
STORE	8	0001000	000	000	000	01	01	1000011
	9	0001001	000	101	000	00	00	0001010
	10	0001010	111	000	000	00	00	1000000
	11	0001011	000	000	000	00	00	1000000
EXCHANGE	12	0001100	000	000	000	01	01	1000011
	13	0001101	001	000	000	00	00	0001110
	14	0001110	100	101	000	00	00	0001111
	15	0001111	111	000	000	00	00	1000000
FETCH	64	1000000	110	000	000	00	00	1000001
	65	1000001	000	100	101	00	00	1000010
	66	1000010	101	000	000	00	11	0000000
INDRCT	67	1000011	000	100	000	00	00	1000100
	68	1000100	101	000	000	00	10	0000000

## **DESIGN OF CONTROL UNIT:**

- The bits of the micro instruction are usually divided into fields, with each field defining a distinct, separate function. Each field requires a decoder produces the corresponding control signals.
- This method reduces the size of the micro instruction bits but requires additional hardware external to the control memory.
- It also increases the delay time of the control signals because they must propagate through the decoding circuits.
- The nine bits of the micro operation field are divided into three subfields of three bits each. The control memory output of each subfield must be decoded to provide the distinct micro operations. The outputs of the decoders are connected to the appropriate inputs in the processor unit.
- Each of the three fields of the micro instruction presently available in the output of control memory are decoded with a 3X8 decoder to provide 8 outputs.
- When  $F_1=101$ , the next clock pulse transition transfers the content of DR (0-11) to AR (DRTAR). Similarly, when  $F_1=110$ , there is a transfer from PC to AR (PCTAR).
- The outputs 5 and 6 of decoder  $F_1$  are connected to the load input of AR so that when either one of these outputs is active, information from the multipliers is transferred to AR.
- The multiplexers select the information from DR when the output 5 is active and from PC when output 5 is inactive. The transfer into AR occurs with a clock pulse transition only when output 5 or output 6 of the decoder is active.
- The other outputs of the decoders that initiate transfers between registers must be connected in a similar fashion.
- The outputs of the decoders are connected to the arithmetic logic shift unit in order to perform the AND, ADD and DRTAC operations.
- The other outputs of the decoders that are associated with an AC operation must also be connected to the arithmetic logic shift unit in a similar fashion.

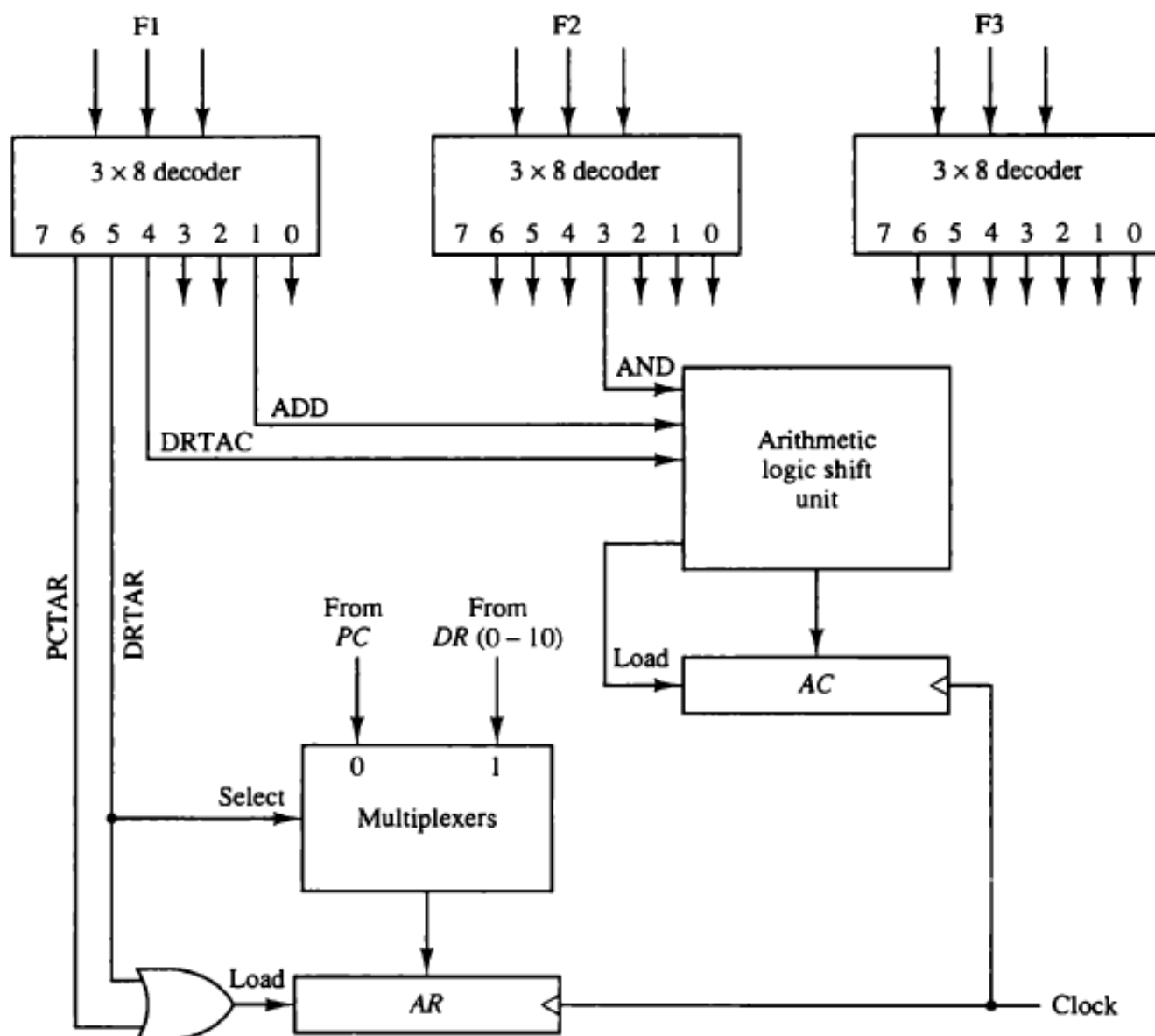


Figure 7-7 Decoding of microoperation fields.

## MICRO PROGRAM SEQUENCER:

- The basic components of a micro programmed control unit are the control memory and the circuits that select the next address. The address sequencer part is called **micro program sequencer**.
- The purpose of a micro program sequencer is to present an address to the memory so that a micro instruction may be read and executed.
- The Next-address logic of the sequencer determines the specific address to be loaded into the control address register.
- The control memory is included in the diagram to show the interaction between sequencer and the memory attached to it. There are two multiplexers in the circuit.
- The **first multiplexer mux1** selects an address from one of four sources and routine it into a control address register CAR.
- The **second multiplexer mux2** tests the values selected status bit and the result of the test is applied to an input logic.
- The output from CAR provides the address for the control memory. The CAR is incremented and applied to one of the multiplexer inputs and the SBR (subroutine register).
- The other three inputs to multiplexer number 1 come from the address field of the present micro instruction, from output of SBR, and from an external source that maps the instruction.
- The CD (condition) field of the micro instruction selects one of the status bits in the second multiplexer. If the bit selected is equal to 1, the T (test) variable is equal to 1, otherwise it is equal to 0.
- The T value together with the two bits from the BR field goes to an input logic circuit. The input logic in a particular sequencer will determine the type of operations that are available in the unit.
- Typical sequencer operations are: increment, branch or jump, call and return from subroutine, load an external address, push or pop the stack, and other address sequencing operations.
- The input logic circuit has three inputs,  $I_0$ ,  $I_1$ , and T and three outputs  $S_0$ ,  $S_1$ , and L. Variables  $S_0$  and  $S_1$  select one of the source address for CAR, Variable L enables the load input in SBR.

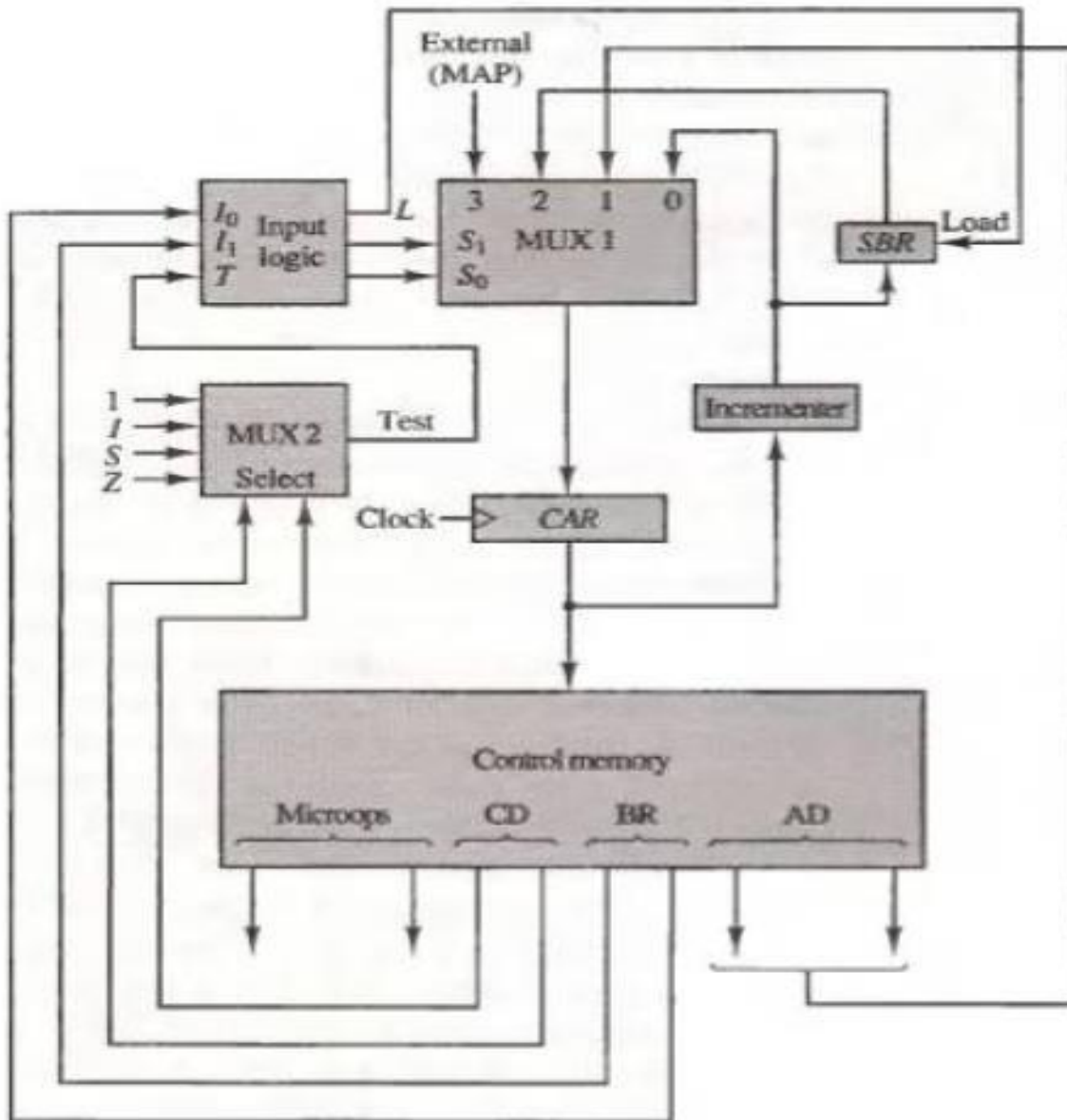


Figure 7-8 Microprogram sequencer for a control memory.

- The binary values of the two selection variables determine the path in the multiplexer. For example: with  $S_1S_0=10$ , multiplexer input number 2 is selected and establishes a transfer path from SBR to CAR. Each of the four inputs as well as the output of MUX1 contains a 7-bit address.
- The inputs  $I_1$  and  $I_0$  are identical to the bit values in the BR field. The bit values for  $S_1$  and  $S_0$  are determined from the stated function and the path in the multiplexer that establishes the required transfer.

- The SBR is load with the incremented value of CAR during a call micro instruction (BR=01) provided that the status bit condition is satisfied (T=1).
- The truth table can be used to obtain the simplified Boolean functions for the input logic circuit.

$$S_1 = I_1$$

$$S_0 = I_1 I_0 + I_1^1 T$$

$$L = I_1^1 I_0 T$$

**TABLE 7-4 Input Logic Truth Table for Microprogram Sequencer**

BR Field		Input $I_1$ $I_0$ $T$			MUX 1 $S_1$ $S_0$		Load <i>SBR</i> $L$
0	0	0	0	0	0	0	0
0	0	0	0	1	0	1	0
0	1	0	1	0	0	0	0
0	1	0	1	1	0	1	1
1	0	1	0	×	1	0	0
1	1	1	1	×	1	1	0

## COMPARISON BETWEEN HARDWIRED CONTROL AND MICRO PROGRAMMED CONTROL UNIT

### Hardwired Control Unit:

- Gates, Flip-Flops, decoders, and other digital circuits are used to implement the control logic.
- Design modifications are done among different components by making changes in wiring.
- Most of the RISC's architecture use hardwired.
- A hardwired is organized by using multiple wires.

**Micro Programmed Control Unit:**

- To implement the control logic, a sequence of micro operations is initiated by programming the control memory which contains control information.
- Modifications are done among various components by updating control memory with a micro program.
- Most of the RISC architecture does not make use of micro programmed control.
- A micro program is organized by using sequence of micro instructions.

<b>ATTRIBUTE</b>	<b>HARDWIRED CONTROL</b>	<b>MICRO PROGRAMMED CONTROL</b>
<b>Speed</b>	Fast	Slow
<b>Control functions</b>	Implemented in hardware	Implemented in software
<b>Flexibility</b>	Not flexible, to accommodate new system specifications or new instructions.	More flexible, to accommodate new specification or new instructions redesign is required.
<b>Ability to handle large / complex instruction sets</b>	Somewhat difficult	Easier
<b>Ability to support operating systems and diagnostic features</b>	Difficult	Easy
<b>Design process</b>	Somewhat complicated	Orderly systematic
<b>Applications</b>	Many RISC processors	Mainframes, some microprocessors.
<b>Instruction set size</b>	Usually under 100 instructions	Usually over 100instructions
<b>ROM size</b>	--	20 – 400 bit micro instructions
<b>Chip area efficiency</b>	Uses least area	Uses more area



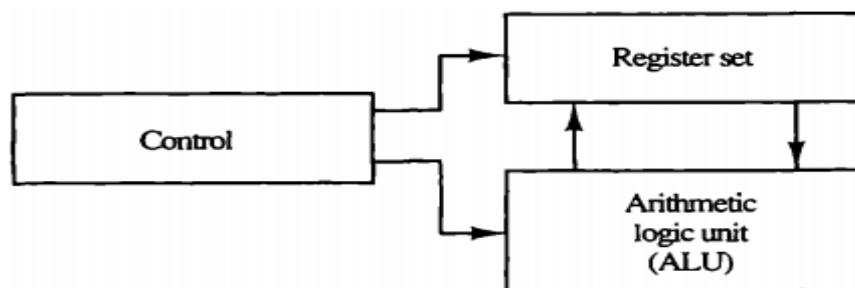
## Central Processing Unit – Introduction

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

- The CPU is made up of three major parts, as shown in Fig. 1. The register set stores intermediate data used during the execution of the instructions. The arithmetic logic unit (ALU) performs the required microoperations for executing the instructions. The control unit supervises the transfer of information among the registers and instructs the ALU as to which operation to perform. The CPU performs a variety of functions dictated by the type of instructions that are incorporated in the computer.

Computer architecture is sometimes defined as the computer structure and behaviour as seen by the programmer that uses machine language instructions. This includes the instruction formats, addressing modes, the instruction set, and the general organization of the CPU registers.

One boundary where the computer designer and the computer programmer see the same machine is the part of the CPU associated with the instruction set. From the designer's point of view, the computer instruction set provides the specifications for the design of the CPU.



**Figure 1** Major components of CPU.

**The design of a CPU** is a task that in large part involves choosing the hardware for implementing the machine instructions. **The user** who programs the computer in machine or assembly language must be aware of the register set, the memory structure, the type of data supported by the instructions, and the function that each instruction performs.

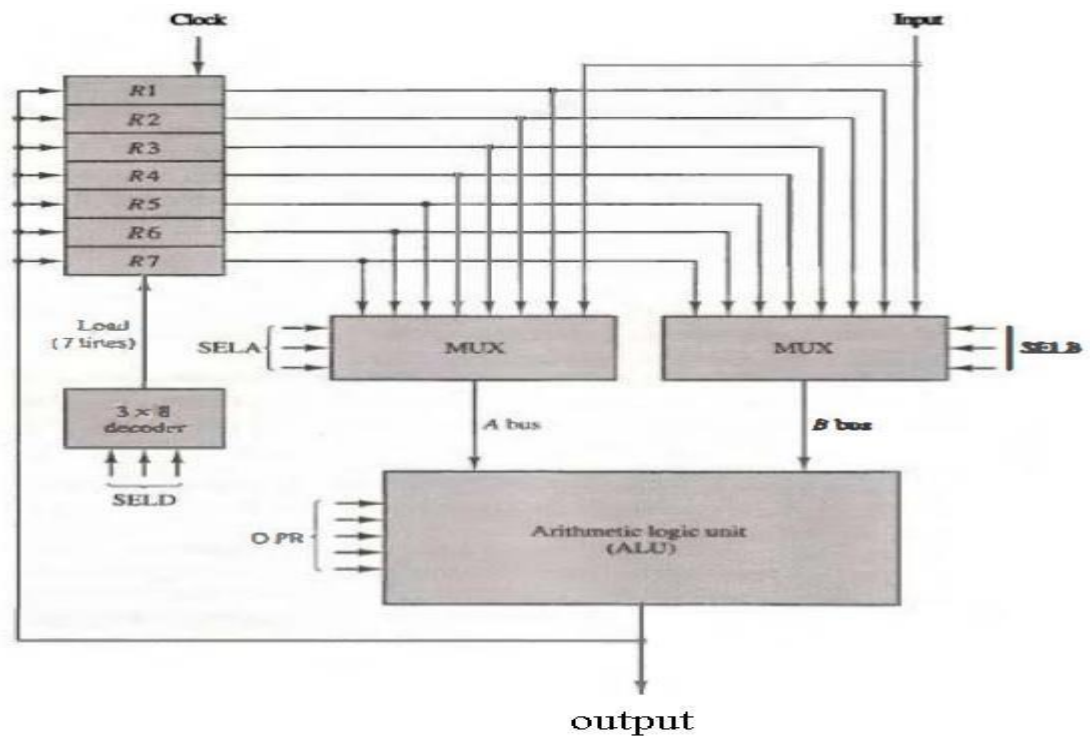
## General Register Organization

- Memory locations are needed for storing pointers, counters, return addresses, temporary results, and partial products during multiplication. Having to refer to memory locations for such applications is time consuming because memory access is the most time-consuming, operation in a computer. It is more convenient and more efficient to store these intermediate values in processor registers.

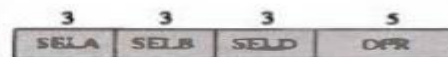
- When a large number of registers are included in the CPU, it is most efficient to connect them through a common bus system. The registers communicate with each other not only for direct data transfers, but also while performing various microoperations. Hence it is necessary to provide a common unit that can perform all the arithmetic, logic, and shift microoperations in the processor.

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

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



(a) Block diagram



(b) Control word

Figure 2 Register set with ALU.

- **The control unit** that operates the CPU bus system directs the information flow through the registers and ALU by selecting the various components in the system. For example, to perform the operation  $R1 \leftarrow R2 + R3$
- **the control must** provide binary selection variables to the following selector inputs:
  1. **MUX A selector (SELA):** to place the content of R2 into bus A.
  2. **MUX B selector (SELB):** to place the content of R 3 into bus B.
  3. **ALU operation selector (OPR):** to provide the arithmetic addition  $A + B$ .
  4. **Decoder destination selector (SELD):** to transfer the content of the output bus into R1.

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

### Control Word:

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

- **It consists of four fields.** Three fields contain three bits each, and one field has five bits.
- **The three bits of SELA** select a source register for the A input of the ALU. The three bits of SELB select a register for the B input of the ALU.
- **The three bits of SELD** select a destination register using the decoder and its seven load outputs.
- **The five bits of OPR** select one of the operations in the ALU.
- **The 14-bit control word** when applied to the selection inputs specify a particular microoperation.
- **The encoding of the register** selections is specified in Table 1.

**TABLE 1** Encoding of Register Selection Fields

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

- **The 3-bit binary code** listed in the first column of the table specifies the binary code for each of the three fields. **The register** selected by fields SELA, SELB, and SELD is the one whose decimal number is equivalent to the binary number in the code. When SELA or SELB is 000, the corresponding multiplexer selects the external input data.

**When SELD = 000**, no destination register is selected but the contents of the output bus are available in the external output. The ALU provides arithmetic and logic operations. **In addition**, the CPU must provide shift operations. The shifter may be placed in the input of the ALU to provide a preshift capability, or at the output of the ALU to provide post shifting capability. In some cases, the shift operations are included with the ALU.

**The function table for this ALU** is listed in Table 8. The encoding of the ALU operations for the CPU is specified in Table 2. The OPR field has five bits and each operation is designated with a symbolic name.

**TABLE 2** Encoding of ALU Operations

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

#### Examples of Microoperations

- **A control word of 14 bits** is needed to specify a microoperation in the CPU. The control word for a given microoperation can be derived from the selection variables.
- **For example**, the subtract microoperation given by the statement  $R1 \leftarrow R2 - R3$  specifies R2 for the A input of the ALU, R3 for the B input of the ALU, R1 for the destination register, and an ALU operation to subtract A - B. **Thus the control word** is
- specified by the four fields and the corresponding binary value for each field is obtained

from the encoding listed in Tables 1 and 2. **The binary control word** for the subtract microoperation is 010 011 001 00101 and is obtained as follows:

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

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

**TABLE 3** Examples of Microoperations for the CPU

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

- A **register** can be cleared to 0 with an exclusive-OR operation. This is because  $x \oplus x = 0$
- **It is apparent** from these examples that many other micro operations can be generated in the CPU. **The most efficient way** to generate control words with a large number of bits is to store them in a memory unit. A **memory unit** that stores control words is referred to as a control memory.
- **By reading consecutive control words from memory**, it is possible to initiate the desired sequence of micro operations for the CPU. **This type of control** is referred to as micro programmed control.

### 1. Instruction Formats:

- The format of an instruction is usually depicted in a rectangular box symbolizing the bits of the instruction as they appear in memory words or in a control register.
- The bits of the instruction are divided into groups called fields.
- The most common fields found in instruction formats are:
  1. An operation code field that specifies the operation to be performed
  2. An address field that designates a memory address or a processor register.
  3. A mode field that specifies the way the operand or the effective address is determined.
- Computers may have instructions of several different lengths containing varying number of addresses.
- The number of address fields in the instruction format of a computer depends on the internal organization of its registers.

- Most computers fall into one of three types of CPU organizations:
  1. Single accumulator organization.
  2. General register organization.
  3. Stack organization.

### Single Accumulator Organization:

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

### General register organization:

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

### Stack organization:

- ✓ The stack-organized CPU has PUSH and POP instructions which require an address field.
- ✓ Thus the instruction **PUSH X** will push the word at address X to the top of the stack.
- ✓ The stack pointer is updated automatically.
- ✓ Operation-type instructions do not need an address field in stack-organized computers.
- ✓ This is because the operation is performed on the two items that are on top of the stack.
- ✓ The instruction **ADD** in a stack computer consists of an operation code only with no address field.
- ✓ This operation has the effect of popping the two top numbers from the stack, adding the numbers, and pushing the sum into the stack.
- ✓ There is no need to specify operands with an address field since all operands are implied to be in the stack.

- Most computers fall into one of the three types of organizations.
- Some computers combine features from more than one organizational structure.
- The influence of the number of addresses on computer programs, we will evaluate the arithmetic statement

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



- Using zero, one, two, or three address instructions and using the symbols ADD, SUB, MUL and DIV for four arithmetic operations; MOV for the transfer type operations; and LOAD and STORE for transfer to and from memory and AC register.
- Assuming that the operands are in memory addresses A, B, C, and D and the result must be stored in memory address X and also the CPU has general purpose registers R1, R2, R3 and R4.

### Three Address Instructions:

- ✓ Three-address instruction formats can use each address field to specify either a processor register or a memory operand.
- ✓ The program assembly language that evaluates  $X = (A+B) * (C+D)$  is shown below, together with comments that explain the register transfer operation of each instruction.

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

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

### Two Address Instructions:

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

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

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

### One Address Instructions:

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

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

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

### Zero Address Instructions:

- ✓ A stack-organized computer does not use an address field for the instructions ADD and MUL.
- ✓ The PUSH and POP instructions, however, need an address field to specify the operand that communicates with the stack.
- ✓ The following program shows how  $X = (A+B) * (C+D)$  will be written for a stack-organized computer. (TOS stands for top of stack).

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

- ✓ To evaluate arithmetic expressions in a stack computer, it is necessary to convert the expression into reverse Polish notation.
- ✓ The name "zero-address" is given to this type of computer because of the absence of an address field in the computational instructions.

### RISC Instructions:

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

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

- ✓ The load instructions transfer the operands from memory to CPU register.
- ✓ The add and multiply operations are executed with data in the register without accessing memory.
- ✓ The result of the computations is then stored memory with a store in instruction.

### 3. Addressing Modes

- ☐ The way the operands are chosen during program execution is dependent on the addressing mode of the instruction.
- ☐ Computers use addressing mode techniques for the purpose of accommodating one or both of the following provisions:
  - o To give programming versatility to the user by providing such facilities as pointers to memory, counters for loop control, indexing of data, and program relocation.
  - o To reduce the number of bits in the addressing field of the instruction
- ☐ Most addressing modes modify the address field of the instruction; there are two modes that need no address field at all. These are *implied* and *immediate* modes.

#### Implied Mode:

- ✓ In this mode the operands are specified implicitly in the definition of the instruction.
- ✓ For example, the instruction "complement accumulator" is an implied-mode instruction because the operand in the accumulator register is implied in the definition of the instruction.
- ✓ All register reference instructions that use an accumulator are implied mode instructions.
- ✓ Zero address in a stack organization computer is implied mode instructions.

#### Immediate Mode:

- ✓ In this mode the operand is specified in the instruction itself.
- ✓ In other words an immediate-mode instruction has an operand rather than an address field.
- ✓ Immediate-mode instructions are useful for initializing registers to a constant value.
- ☐ The address field of an instruction may specify either a memory word or a processor register.
- ☐ When the address specifies a processor register, the instruction is said to be in the register mode.

### Register Mode:

- ✓ In this mode the operands are in registers that reside within the CPU.
- ✓ The particular register is selected from a register field in the instruction.

### Register Indirect Mode:

- ✓ In this mode the instruction specifies a register in CPU whose contents give the address of the operand in memory.
- ✓ In other words, the selected register contains the address of the operand rather than the operand itself.
- ✓ The advantage of a register indirect mode instruction is that the address field of the instruction uses few bits to select a register than would have been required to specify a memory address directly.

### Auto-increment or Auto-Decrement Mode:

- ✓ This is similar to the register indirect mode except that the register is incremented or decremented after (or before) its value is used to access memory.
- The address field of an instruction is used by the control unit in the CPU to obtain the operand from memory.
- Sometimes the value given in the address field is the address of the operand, but sometimes it is just an address from which the address of the operand is calculated.
- The basic two mode of addressing used in CPU are *direct* and *indirect* address mode.

### Direct Address Mode:

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

### Indirect Address Mode:

- ✓ In this mode the address field of the instruction gives the address where the effective address is stored in memory.
  - ✓ Control fetches the instruction from memory and uses its address part to access memory again to read the effective address.
- ☐ A few addressing modes require that the address field of the instruction be added to the content of a specific register in the CPU.
  - ☐ The effective address in these modes is obtained from the following computation:

$$\text{Effective address} = \text{address part of instruction} + \text{content of CPU register}$$

- ☐ The CPU register used in the computation may be the program counter, an index register, or a base register.
- ☐ We have a different addressing mode which is used for a different application.

### Relative Address Mode:

- ✓ In this mode the content of the program counter is added to the address part of the instruction in order to obtain the effective address.

### Indexed Addressing Mode:

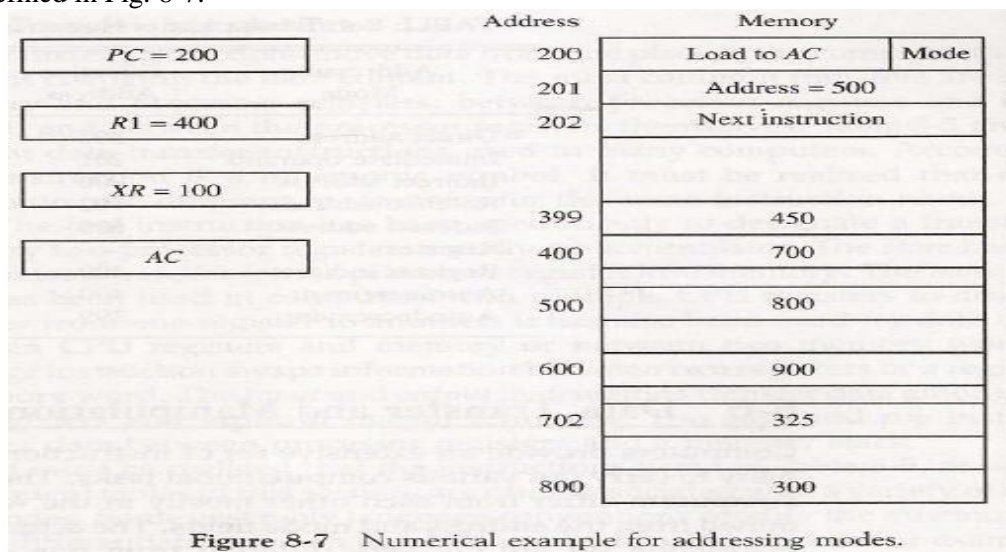
- ✓ In this mode the content of an index register is added to the address part of the instruction to obtain the effective address.
- ✓ An index register is a special CPU register that contains an index value.

### Base Register Addressing Mode:

- ✓ In this mode the content of a base register is added to the address part of the instruction to obtain the effective address.
- ✓ This is similar to the indexed addressing mode except that the register is now called a base register instead of an index register.

## Numerical Example:

- To show the differences between the various modes, we will show the effect of the addressing modes on the instruction defined in Fig. 8-7.



- The two-word instruction at address 200 and 201 is a "load to AC" instruction with an address field equal to 500.
- The first word of the instruction specifies the operation code and mode, and the second word specifies the address part.
- PC has the value 200 for fetching this instruction. The content of processor register R1 is 400, and the content of an index register XR is 100.
- AC receives the operand after the instruction is executed.
- In the **direct address mode** the effective address is the address part of the instruction 500 and the operand to be loaded into AC is 500.
- In the **immediate mode** the second word of the instruction is taken as the operand rather than an address, so 500 is loaded into AC.
- In the **indirect mode** the effective address is stored in memory at address 500. Therefore, the effective address is 800 and the operand is 300.
- In the **relative mode** the effective address is  $500 + 202 = 702$  and the operand is 325. (the value in PC after the fetch phase and during the execute phase is 202.)
- In the **index mode** the effective address is  $XR + 500 = 100 + 500 = 600$  and the operand is 900.
- In the **register mode** the operand is in R1 and 400 is loaded into AC.
- In the **register indirect mode** the effective address is 400, equal to the content of R1 and the operand loaded into AC is 700.
- The **auto-increment mode** is the same as the register indirect mode except that R1 is incremented to 401 after the execution of the instruction.
- The **auto-decrement mode** decrements R1 to 399 prior to the execution of the instruction. The operand loaded into AC is now 450.
- Table 8-4 lists the values of the effective address and the operand loaded into AC for the nine addressing modes.

TABLE 8-4 Tabular List of Numerical Example		
Addressing Mode	Effective Address	Content of AC
Direct address	500	800
Immediate operand	201	500
Indirect address	800	300
Relative address	702	325
Indexed address	600	900
Register	—	400
Register indirect	400	700
Autoincrement	400	700
Autodecrement	399	450



#### 4. Data Transfer and Manipulation:

- Most computer instructions can be classified into three categories:

1. Data transfer instructions
2. Data manipulation instructions
3. Program control instructions

##### Data Transfer Instructions:

- Data transfer instructions move data from one place in the computer to another without changing the data content.
- The most common transfers are between memory and processor registers, between processor registers and input or output, and between the processor registers themselves.
- Table 8-5 gives a list of eight data transfer instructions used in many computers.

TABLE 8-5 Typical Data Transfer Instructions

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

- The **load** instruction has been used mostly to designate a transfer from memory to a processor register, usually an accumulator.
- The **store** instruction designates a transfer from a processor register into memory.
- The **move** instruction has been used in computers with multiple CPU registers to designate a transfer from one register to another and also between CPU registers and memory or between two memory words.
- The **exchange** instruction swaps information between two registers or a register and a memory word.
- The **input** and **output** instructions transfer data among processor registers and input or output terminals.
- The **push** and **pop** instructions transfer data between processor registers and a memory stack.
- Different computers use different mnemonics symbols for differentiate the addressing modes.
- As an example, consider the **load to accumulator** instruction when used with eight different addressing modes.
- Table 8-6 shows the recommended assembly language convention and actual transfer accomplished in each case

TABLE 8-6 Eight Addressing Modes for the Load Instruction

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

- ADR stands for an address.
- NBA a number or operand.
- X is an index register.
- The @ character symbolizes an indirect addressing.
- The \$ character before an address makes the address relative to the program counter PC.
- The # character precedes the operand in an immediate-mode instruction.
- R1 is a processor register.
- AC is the accumulator register.

- An indexed mode instruction is recognized by a register that placed in parentheses after the symbolic address.
- The register mode is symbolized by giving the name of a processor register.
- In the register indirect mode, the name of the register that holds the memory address is enclosed in parentheses.
- The auto-increment mode is distinguished from the register indirect mode by placing a plus after the parenthesized register. The auto-decrement mode would use a minus instead.

### Data Manipulation Instructions:

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

#### 1. Arithmetic instructions

- ✓ The four basic arithmetic operations are addition, subtraction, multiplication and division.
- ✓ Most computers provide instructions for all four operations.
- ✓ Some small computers have only addition and possibly subtraction instructions. The multiplication and division must then be generated by mean software subroutines.
- ✓ A list of typical arithmetic instructions is given in Table 8-7.

**TABLE 8-7 Typical Arithmetic Instructions**

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

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

#### 2. Logical and bit manipulation instructions

- ✓ Logical instructions perform binary operations on strings of bits store, registers.
- ✓ They are useful for manipulating individual bits or a group of that represent binary-coded information.
- ✓ The logical instructions consider each bit of the operand separately and treat it as a Boolean variable.
- ✓ By proper application of the logical instructions it is possible to change bit values, to clear a group of bits, or to insert new bit values into operands stored in register memory words.
- ✓ Some typical logical and bit manipulation instructions are listed in Table 8-8.

**TABLE 8-8 Typical Logical and Bit Manipulation Instructions**

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

- ✓ The clear instruction causes the specified operand to be replaced by 0's.
- ✓ The complement instruction produces the 1's complement by inverting all bits of the operand.
- ✓ The AND, OR, and XOR instructions produce the corresponding logical operations on individual bits of the operands.
- ✓ The logical instructions can also be used to performing bit manipulation operations.
- ✓ There are three bit manipulation operations possible: a selected bit can cleared to 0, or can be set to 1, or can be complemented.
  - The AND instruction is used to clear a bit or a selected group of bits of an operand.
  - The OR instruction is used to set a bit or a selected group of bits of an operand.
  - Similarly, the XOR instruction is used to selectively complement bits of an operand.
- ✓ Other bit manipulations instructions are included in above table perform the operations on individual bits such as a carry can be cleared, set, or complemented.
- ✓ Another example is a flip-flop that controls the interrupt facility and is either enabled or disabled by means of bit manipulation instructions.

### 3. Shift Instructions:

- ✓ Shifts are operations in which the bits of a word are moved to the left or right.
- ✓ The bit shifted in at the end of the word determines the type of shift used.
- ✓ Shift instructions may specify logical shifts, arithmetic shifts, or rotate-type operations.
- ✓ In either case the shift may be to the right or to the left.
- ✓ Table 8-9 lists four types of shift instructions.

**TABLE 8-9 Typical Shift Instructions**

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

- ✓ The logical shift inset to the end bit position.
- ✓ The end position is the leftmost bit position for shift rights the rightmost bit position for the shift left.
- ✓ Arithmetic shifts usually conform to the rules for signed-2's complement numbers.
- ✓ The arithmetic shift-right instruction must preserve the sign bit in the leftmost position.
- ✓ The sign bit is shifted to the right together with the rest of the number, but the sign bit itself remains unchanged.
- ✓ This is a shift-right operation with the end bit remaining the same.
- ✓ The arithmetic shift-left instruction inserts 0 to the end position and is identical to the logical shift-instruction.



- ✓ The rotate instructions produce a circular shift. Bits shifted out at one of the word are not lost as in a logical shift but are circulated back into the other end.
- ✓ The rotate through carry instruction treats a carry bit as an extension of the register whose word is being rotated.
- ✓ Thus a rotate-left through *carry* instruction transfers the carry bit into the rightmost bit position of the register, transfers the leftmost bit position into the carry, and at the same time, shift the entire register to the left.

## 5. Program Control:

- Program control instructions specify conditions for altering the content of the program counter.
- The change in value of the program counter as a result of the execution of a program control instruction causes a break in the sequence of instruction execution.
- This instruction provides control over the flow of program execution and a capability for branching to different program segments.
- Some typical program control instructions are listed in Table 8.10.

TABLE 8-10 Typical Program Control Instructions

Name	Mnemonic
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RET
Compare (by subtraction)	CMP
Test (by ANDing)	TST

- Branch and jump instructions may be conditional or unconditional.
- An unconditional branch instruction causes a branch to the specified address without any conditions.
- The conditional branch instruction specifies a condition such as branch if positive or branch if zero.
- The skip instruction does not need an address field and is therefore a zero-address instruction.
- A conditional skip instruction will skip the next instruction if the condition is met. This is accomplished by incrementing program counter.
- The call and return instructions are used in conjunction with subroutines.
- The compare instruction forms a subtraction between two operands, but the result of the operation not retained. However, certain status bit conditions are set as a result of operation.
- Similarly, the test instruction performs the logical AND of two operands and updates certain status bits without retaining the result or changing the operands.

## Status Bit Conditions:

- The ALU circuit in the CPU have status register for storing the status bit conditions.
- Status bits are also called *condition-code* bits or *flag* bits.
- Figure 8-8 shows block diagram of an 8-bit ALU with a 4-bit status register.

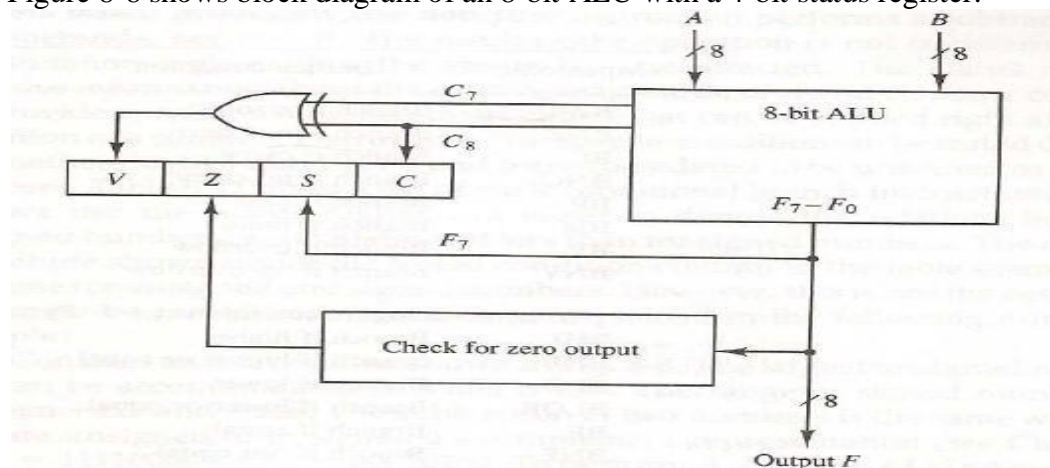


Figure 8-8 Status register bits.

- The four status bits are symbolized by C, S, Z, and V. The bits are set or cleared as a result of an operation performed in the ALU.
  - Bit C (carry) is set to 1 if the end carry  $C_8$  is 1. It is cleared to 0 if the carry is 0.
  - S (sign) is set to 1 if the highest-order bit  $F_7$  is 1. It is set to 0 if the bit is 0.
  - Bit Z (zero) is set to 1 if the output of the ALU contains all 0's. It is clear to 0 otherwise. In other words,  $Z = 1$  if the output is zero and  $Z = 0$  if the output is not zero.
  - Bit V (overflow) is set to 1 if the exclusive-OR of the last two carries equal to 1, and cleared to 0 otherwise.
- The above status bits are used in conditional jump and branch instructions.

### Subroutine Call and Return:

- A subroutine is self contained sequence of instructions that performs a given computational task.
- The most common names used are call subroutine, jump to subroutine, branch to subroutine, or branch and save return address.
- A subroutine is executed by performing two operations
  - (1) The address of the next instruction available in the program counter (the return address) is stored in a temporary location so the subroutine knows where to return
  - (2) Control is transferred to the beginning of the subroutine.
- The last instruction of every subroutine, commonly called *return from subroutine*, transfers the return address from the temporary location in the program counter.
- Different computers use a different temporary location for storing the return address.
- The most efficient way is to store the return address in a memory stack.
- The advantage of using a stack for the return address is that when a succession of subroutines is called, the sequential return addresses can be pushed into the stack.
- A subroutine call is implemented with the following microoperations:

$SP \leftarrow SP - 1$	Decrement stack pointer
$M[SP] \leftarrow PC$	Push content of PC onto the stack
$PC \leftarrow \text{effective address}$	Transfer control to the subroutine

- The instruction that returns from the last subroutine is implemented by the microoperations:

$PC \leftarrow M[SP]$	Pop stack and transfer to PC
$SP \leftarrow SP + 1$	Increment stack pointer

### Program Interrupt:

- Program interrupt refers to the transfer of program control from a currently running program to another service program as a result of an external or internal generated request.
- The interrupt procedure is similar to a subroutine call except for three variations:
  - The interrupt is initiated by an internal or external signal.
  - Address of the interrupt service program is determined by the hardware.
  - An interrupt procedure usually stores all the information rather than storing only PC content.

### Types of interrupts:

- ✓ There are three major types of interrupts that cause a break in the normal execution of a program.
- ✓ They can be classified as
  - **External interrupts:**
    - These come from input—output (I/O) devices, from a timing device, from a circuit monitoring the power supply, or from any other external source.

- Ex: I/O device requesting transfer of data, I/O device finished transfer of data, elapsed time of an event, or power failure.
- **Internal interrupts:**
  - These arise from illegal or erroneous use of an instruction or data.
  - Internal interrupts are also called *traps*.
  - Ex: interrupts caused by internal error conditions are register overflow, attempt to divide by zero, an invalid operation code, stack overflow, and protection violation.
- ✓ Internal and external interrupts are initiated from signals that occur in hardware of CPU.
- **Software interrupts**
  - A software interrupt is initiated by executing an instruction.
  - Software interrupt is a special call instruction that behaves like an interrupt rather than a subroutine call.

## CHAPTER THREE

# Data Representation

### IN THIS CHAPTER

- 3-1 Data Types
- 3-2 Complements
- 3-3 Fixed-Point Representation
- 3-4 Floating-Point Representation
- 3-5 Other Binary Codes
- 3-6 Error Detection Codes

### 3-1 Data Types

---

Binary information in digital computers is stored in memory or processor registers. Registers contain either data or control information. Control information is a bit or a group of bits used to specify the sequence of command signals needed for manipulation of the data in other registers. Data are numbers and other binary-coded information that are operated on to achieve required computational results. In this chapter we present the most common types of data found in digital computers and show how the various data types are represented in binary-coded form in computer registers.

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

## Number Systems

**radix** A number system of *base*, or *radix*,  $r$  is a system that uses distinct symbols for  $r$  digits. Numbers are represented by a string of digit symbols. To determine the quantity that the number represents, it is necessary to multiply each digit by an integer power of  $r$  and then form the sum of all weighted digits. For example, the decimal number system in everyday use employs the radix 10 system. The 10 symbols are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. The string of digits 7245 is interpreted to represent the quantity

$$7 \times 10^3 + 2 \times 10^2 + 4 \times 10^1 + 5 \times 10^0$$

that is, 7 hundreds, plus 2 tens, plus 4 units, plus 5 tenths. Every decimal number can be similarly interpreted to find the quantity it represents.

**decimal** The *binary* number system uses the radix 2. The two digit symbols used are 0 and 1. The string of digits 101101 is interpreted to represent the quantity

$$1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 45$$

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

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

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

$$\begin{aligned}(736.4)_8 &= 7 \times 8^2 + 3 \times 8^1 + 6 \times 8^0 + 4 \times 8^{-1} \\ &= 7 \times 64 + 3 \times 8 + 6 \times 1 + 4/8 = (478.5)_{10}\end{aligned}$$

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

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

**conversion** Conversion from decimal to its equivalent representation in the radix  $r$  system is carried out by separating the number into its *integer* and *fraction* parts and

converting each part separately. The conversion of a decimal integer into a base  $r$  representation is done by successive divisions by  $r$  and accumulation of the remainders. The conversion of a decimal fraction to radix  $r$  representation is accomplished by successive multiplications by  $r$  and accumulation of the integer digits so obtained. Figure 3-1 demonstrates these procedures.

The conversion of decimal 41.6875 into binary is done by first separating the number into its integer part 41 and fraction part .6875. The integer part is converted by dividing 41 by  $r = 2$  to give an integer quotient of 20 and a remainder of 1. The quotient is again divided by 2 to give a new quotient and remainder. This process is repeated until the integer quotient becomes 0. The coefficients of the binary number are obtained from the remainders with the first remainder giving the low-order bit of the converted binary number.

The fraction part is converted by multiplying it by  $r = 2$  to give an integer and a fraction. The new fraction (*without* the integer) is multiplied again by 2 to give a new integer and a new fraction. This process is repeated until the fraction part becomes zero or until the number of digits obtained gives the required accuracy. The coefficients of the binary fraction are obtained from the integer digits with the first integer computed being the digit to be placed next to the binary point. Finally, the two parts are combined to give the total required conversion.

### Octal and Hexadecimal Numbers

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

Figure 3-1 Conversion of decimal 41.6875 into binary.

Integer = 41	Fraction = 0.6875
41	0.6875
20   1	<u>        </u> 2
10   0	1.3750
5   0	<u>        </u> x 2
2   1	0.7500
1   0	<u>        </u> x 2
0   1	1.5000
	<u>        </u> x 2
	1.0000
	<u>        </u>
$(41)_{10} = (101001)_2$	$(0.6875)_{10} = (0.1011)_2$
$(41.6875)_{10} = (101001.1011)_2$	

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

Figure 3-2 Binary, octal, and hexadecimal conversion.

register as composed of 16 binary storage cells, with each cell capable of holding either a 1 or a 0. Suppose that the bit configuration stored in the register is as shown in Fig. 3-2. Since a binary number consists of a string of 1's and 0's, the 16-bit register can be used to store any binary number from 0 to  $2^{16} - 1$ . For the particular example shown, the binary number stored in the register is the equivalent of decimal 44899. Starting from the low-order bit, we partition the register into groups of three bits each (the sixteenth bit remains in a group by itself). Each group of three bits is assigned its octal equivalent and placed on top of the register. The string of octal digits so obtained represents the octal equivalent of the binary number.

Conversion from binary to hexadecimal is similar except that the bits are divided into groups of four. The corresponding hexadecimal digit for each group of four bits is written as shown below the register of Fig. 3-2. The string of hexadecimal digits so obtained represents the hexadecimal equivalent of the binary number. The corresponding octal digit for each group of three bits is easily remembered after studying the first eight entries listed in Table 3-1. The correspondence between a hexadecimal digit and its equivalent 4-bit code can be found in the first 16 entries of Table 3-2.

TABLE 3-1 Binary-Coded Octal Numbers


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



Table 3-1 lists a few octal numbers and their representation in registers in binary-coded form. The binary code is obtained by the procedure explained above. Each octal digit is assigned a 3-bit code as specified by the entries of the first eight digits in the table. Similarly, Table 3-2 lists a few hexadecimal numbers and their representation in registers in binary-coded form. Here the binary code is obtained by assigning to each hexadecimal digit the 4-bit code listed in the first 16 entries of the table.

Comparing the binary-coded octal and hexadecimal numbers with their binary number equivalent we find that the bit combination in all three representations is exactly the same. For example, decimal 99, when converted to binary, becomes 1100011. The binary-coded octal equivalent of decimal 99 is 001 100 011 and the binary-coded hexadecimal of decimal 99 is 0110 0011. If we neglect the leading zeros in these three binary representations, we find that their bit combination is identical. This should be so because of the straightforward conversion that exists between binary numbers and octal or hexadecimal. The point of all this is that a string of 1's and 0's stored in a register could represent a binary number, but this same string of bits may be interpreted as holding an octal number in binary-coded form (if we divide the bits in groups of three) or as holding a hexadecimal number in binary-coded form (if we divide the bits in groups of four).

TABLE 3-2 Binary-Coded Hexadecimal Numbers

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

The registers in a digital computer contain many bits. Specifying the content of registers by their binary values will require a long string of binary digits. It is more convenient to specify content of registers by their octal or hexadecimal equivalent. The number of digits is reduced by one-third in the octal designation and by one-fourth in the hexadecimal designation. For example, the binary number 1111 1111 1111 has 12 digits. It can be expressed in octals as 7777 (four digits) or in hexadecimal as FFF (three digits). Computer manuals invariably choose either the octal or the hexadecimal designation for specifying contents of registers.

## Decimal Representation

The binary number system is the most natural system for a computer, but people are accustomed to the decimal system. One way to solve this conflict is to convert all input decimal numbers into binary numbers, let the computer perform all arithmetic operations in binary and then convert the binary results back to decimal for the human user to understand. However, it is also possible for the computer to perform arithmetic operations directly with decimal numbers provided they are placed in registers in a coded form. Decimal numbers enter the computer usually as binary-coded alphanumeric characters. These codes, introduced later, may contain from six to eight bits for each decimal digit. When decimal numbers are used for internal arithmetic computations, they are converted to a binary code with four bits per digit.

A binary code is a group of  $n$  bits that assume up to  $2^n$  distinct combinations of 1's and 0's with each combination representing one element of the set that is being coded. For example, a set of four elements can be coded by a 2-bit code with each element assigned one of the following bit combinations; 00, 01, 10, or 11. A set of eight elements requires a 3-bit code, a set of 16 elements requires a 4-bit code, and so on. A binary code will have some unassigned bit combinations if the number of elements in the set is not a multiple power of 2. The 10 decimal digits form such a set. A binary code that distinguishes among 10 elements must contain at least four bits, but six combinations will remain unassigned. Numerous different codes can be obtained by arranging four bits in 10 distinct combinations. The bit assignment most commonly used for the decimal digits is the straight binary assignment listed in the first 10 entries of Table 3-3. This particular code is called *binary-coded decimal* and is commonly referred to by its abbreviation BCD. Other decimal codes are sometimes used and a few of them are given in Sec. 3-5.

It is very important to understand the difference between the *conversion* of decimal numbers into binary and the *binary coding* of decimal numbers. For example, when *converted* to a binary number, the decimal number 99 is represented by the string of bits 1100011, but when represented in BCD, it becomes 1001 1001. The *only* difference between a decimal number represented by the familiar digit symbols 0, 1, 2, . . . , 9 and the BCD symbols 0001, 0010, . . . , 1001 is in the symbols used to represent the digits—the number itself is exactly the

*binary code*

**BCD**

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

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

same. A few decimal numbers and their representation in BCD are listed in Table 3-3.

### Alphanumeric Representation

Many applications of digital computers require the handling of data that consist not only of numbers, but also of the letters of the alphabet and certain special characters. An *alphanumeric character set* is a set of elements that includes the 10 decimal digits, the 26 letters of the alphabet and a number of special characters, such as \$, +, and =. Such a set contains between 32 and 64 elements (if only uppercase letters are included) or between 64 and 128 (if both uppercase and lowercase letters are included). In the first case, the binary code will require six bits and in the second case, seven bits. The standard alphanumeric binary code is the ASCII (American Standard Code for Information Interchange), which uses seven bits to code 128 characters. The binary code for the uppercase letters, the decimal digits, and a few special characters is listed in Table 3-4. Note that the decimal digits in ASCII can be converted to BCD by removing the three high-order bits, 011. A complete list of ASCII characters is provided in Table 11-1.

Binary codes play an important part in digital computer operations. The codes must be in binary because registers can only hold binary information. One must realize that binary codes merely change the symbols, not the meaning of the discrete elements they represent. The operations specified for digital

character

ASCII

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

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

computers must take into consideration the meaning of the bits stored in registers so that operations are performed on operands of the same type. In inspecting the bits of a computer register at random, one is likely to find that it represents some type of coded information rather than a binary number.

Binary codes can be formulated for any set of discrete elements such as the musical notes and chess pieces and their positions on the chessboard. Binary codes are also used to formulate instructions that specify control information for the computer. This chapter is concerned with *data* representation. Instruction codes are discussed in Chap. 5.

## 3-2 Complements

Complements are used in digital computers for simplifying the subtraction operation and for logical manipulation. There are two types of complements for each base  $r$  system: the  $r$ 's complement and the  $(r - 1)$ 's complement.

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

### $(r - 1)$ 's Complement

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

For binary numbers,  $r = 2$  and  $r - 1 = 1$ , so the 1's complement of  $N$  is  $(2^n - 1) - N$ . Again,  $2^n$  is represented by a binary number that consists of a 1 followed by  $n$  0's.  $2^n - 1$  is a binary number represented by  $n$  1's. For example, with  $n = 4$ , we have  $2^4 = (10000)_2$  and  $2^4 - 1 = (1111)_2$ . Thus the 1's complement of a binary number is obtained by subtracting each digit from 1. However, the subtraction of a binary digit from 1 causes the bit to change from 0 to 1 or from 1 to 0. Therefore, the 1's complement of a binary number is formed by changing 1's into 0's and 0's into 1's. For example, the 1's complement of 1011001 is 0100110 and the 1's complement of 0001111 is 1110000.

The  $(r - 1)$ 's complement of octal or hexadecimal numbers are obtained by subtracting each digit from 7 or F (decimal 15) respectively.

### $(r)$ 's Complement

The  $r$ 's complement of an  $n$ -digit number  $N$  in base  $r$  is defined as  $r^n - N$  for  $N \neq 0$  and 0 for  $N = 0$ . Comparing with the  $(r - 1)$ 's complement, we note that the  $r$ 's complement is obtained by adding 1 to the  $(r - 1)$ 's complement since  $r^n - N = [(r^n - 1) - N] + 1$ . Thus the 10's complement of the decimal 2389 is  $7610 + 1 = 7611$  and is obtained by adding 1 to the 9's complement value. The 2's complement of binary 101100 is  $010011 + 1 = 010100$  and is obtained by adding 1 to the 1's complement value.

Since  $10^n$  is a number represented by a 1 followed by  $n$  0's, then  $10^n - N$ , which is the 10's complement of  $N$ , can be formed also by leaving all least significant 0's unchanged, subtracting the first nonzero least significant digit from 10, and then subtracting all higher significant digits from 9. The 10's complement of 246700 is 753300 and is obtained by leaving the two zeros unchanged, subtracting 7 from 10, and subtracting the other three digits from 9. Similarly, the 2's complement can be formed by leaving all least significant 0's and the first 1 unchanged, and then replacing 1's by 0's and 0's by 1's in all other higher significant bits. The 2's complement of 1101100 is 0010100 and is obtained by leaving the two low-order 0's and the first 1 unchanged, and then replacing 1's by 0's and 0's by 1's in the other four most significant bits.

9's complement

1's complement

10's complement

2's complement

In the definitions above it was assumed that the numbers do not have a radix point. If the original number  $N$  contains a radix point, it should be removed temporarily to form the  $r$ 's or  $(r - 1)$ 's complement. The radix point is then restored to the complemented number in the same relative position. It is also worth mentioning that the complement of the complement restores the number to its original value. The  $r$ 's complement of  $N$  is  $r^n - N$ . The complement of the complement is  $r^n - (r^n - N) = N$  giving back the original number.

### Subtraction of Unsigned Numbers

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

The subtraction of two  $n$ -digit unsigned numbers  $M - N$  ( $N \neq 0$ ) in base  $r$  can be done as follows:

1. Add the minuend  $M$  to the  $r$ 's complement of the subtrahend  $N$ . This performs  $M + (r^n - N) = M - N + r^n$ .
2. If  $M \geq N$ , the sum will produce an end carry  $r^n$  which is discarded, and what is left is the result  $M - N$ .
3. If  $M < N$ , the sum does not produce an end carry and is equal to  $r^n - (N - M)$ , which is the  $r$ 's complement of  $(N - M)$ . To obtain the answer in a familiar form, take the  $r$ 's complement of the sum and place a negative sign in front.

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

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

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

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

*subtraction*

*end carry*

There is no end carry

Answer is negative 59282 = 10's complement of 40718

Since we are dealing with unsigned numbers, there is really no way to get an unsigned result for the second example. When working with paper and pencil, we recognize that the answer must be changed to a signed negative number. When subtracting with complements, the negative answer is recognized by the absence of the end carry and the complemented result.

Subtraction with complements is done with binary numbers in a similar manner using the same procedure outlined above. Using the two binary numbers  $X = 1010100$  and  $Y = 1000011$ , we perform the subtraction  $X - Y$  and  $Y - X$  using 2's complements:

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

There is no end carry

Answer is negative 0010001 = 2's complement of 1101111

### 3-3 Fixed-Point Representation

Positive integers, including zero, can be represented as unsigned numbers. However, to represent negative integers, we need a notation for negative values. In ordinary arithmetic, a negative number is indicated by a minus sign and a positive number by a plus sign. Because of hardware limitations, computers must represent everything with 1's and 0's, including the sign of a number. As a consequence, it is customary to represent the sign with a bit placed in the leftmost position of the number. The convention is to make the sign bit equal to 0 for positive and to 1 for negative.

#### *binary point*

In addition to the sign, a number may have a binary (or decimal) point. The position of the binary point is needed to represent fractions, integers, or mixed integer-fraction numbers. The representation of the binary point in a register is complicated by the fact that it is characterized by a position in the register. There are two ways of specifying the position of the binary point in a register: by giving it a fixed position or by employing a floating-point representation. The fixed-point method assumes that the binary point is always



fixed in one position. The two positions most widely used are (1) a binary point in the extreme left of the register to make the stored number a fraction, and (2) a binary point in the extreme right of the register to make the stored number an integer. In either case, the binary point is not actually present, but its presence is assumed from the fact that the number stored in the register is treated as a fraction or as an integer. The floating-point representation uses a second register to store a number that designates the position of the decimal point in the first register. Floating-point representation is discussed further in the next section.

### Integer Representation

#### *signed numbers*

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

1. Signed-magnitude representation
2. Signed-1's complement representation
3. Signed 2's complement representation

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

In signed-magnitude representation      1 0001110

In signed-1's complement representation    1 1110001

In signed-2's complement representation    1 1110010

The signed-magnitude representation of -14 is obtained from +14 by complementing only the sign bit. The signed-1's complement representation of -14 is obtained by complementing all the bits of +14, including the sign bit. The signed-2's complement representation is obtained by taking the 2's complement of the positive number, including its sign bit.

The signed-magnitude system is used in ordinary arithmetic but is awkward when employed in computer arithmetic. Therefore, the signed-complement is normally used. The 1's complement imposes difficulties because it

has two representations of 0 (+0 and -0). It is seldom used for arithmetic operations except in some older computers. The 1's complement is useful as a logical operation since the change of 1 to 0 or 0 to 1 is equivalent to a logical complement operation. The following discussion of signed binary arithmetic deals exclusively with the signed-2's complement representation of negative numbers.

### Arithmetic Addition

The addition of two numbers in the signed-magnitude system follows the rules of ordinary arithmetic. If the signs are the same, we add the two magnitudes and give the sum the common sign. If the signs are different, we subtract the smaller magnitude from the larger and give the result the sign of the larger magnitude. For example,  $(+25) + (-37) = -(37 - 25) = -12$  and is done by subtracting the smaller magnitude 25 from the larger magnitude 37 and using the sign of 37 for the sign of the result. This is a process that requires the comparison of the signs and the magnitudes and then performing either addition or subtraction. (The procedure for adding binary numbers in signed-magnitude representation is described in Sec. 10-2.) By contrast, the rule for adding numbers in the signed-2's complement system does not require a comparison or subtraction, only addition and complementation. The procedure is very simple and can be stated as follows: Add the two numbers, including their sign bits, and discard any carry out of the sign (leftmost) bit position. Numerical examples for addition are shown below. Note that negative numbers must initially be in 2's complement and that if the sum obtained after the addition is negative, it is in 2's complement form.

#### 2's complement addition

+6	00000110	-6	11111010
+13	00001101	+13	00001101
+19	00010011	+7	00000111
<hr/>			
+6	00000110	-6	11111010
-13	11110011	-13	11110011
-7	11111001	-19	11101101

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

The complement form of representing negative numbers is unfamiliar to people used to the signed-magnitude system. To determine the value of a negative number when in signed-2's complement, it is necessary to convert it to a positive number to place it in a more familiar form. For example, the signed binary number 11111001 is negative because the leftmost bit is 1. Its 2's complement is 00000111, which is the binary equivalent of +7. We therefore recognize the original negative number to be equal to -7.

**2's complement  
subtraction****Arithmetic Subtraction**

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

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

$$(\pm A) - (+B) = (\pm A) + (-B)$$

$$(\pm A) - (-B) = (\pm A) + (+B)$$

But changing a positive number to a negative number is easily done by taking its 2's complement. The reverse is also true because the complement of a negative number in complement form produces the equivalent positive number. Consider the subtraction of  $(-6) - (-13) = +7$ . In binary with eight bits this is written as  $11111010 - 11110011$ . The subtraction is changed to addition by taking the 2's complement of the subtrahend  $(-13)$  to give  $(+13)$ . In binary this is  $11111010 + 00001101 = 100000111$ . Removing the end carry, we obtain the correct answer  $00000111 (+7)$ .

It is worth noting that binary numbers in the signed-2's complement system are added and subtracted by the same basic addition and subtraction rules as unsigned numbers. Therefore, computers need only one common hardware circuit to handle both types of arithmetic. The user or programmer must interpret the results of such addition or subtraction differently depending on whether it is assumed that the numbers are signed or unsigned.

**Overflow****overflow**

When two numbers of  $n$  digits each are added and the sum occupies  $n + 1$  digits, we say that an overflow occurred. When the addition is performed with paper and pencil, an overflow is not a problem since there is no limit to the width of the page to write down the sum. An overflow is a problem in digital computers because the width of registers is finite. A result that contains  $n + 1$  bits cannot be accommodated in a register with a standard length of  $n$  bits. For this reason, many computers detect the occurrence of an overflow, and when it occurs, a corresponding flip-flop is set which can then be checked by the user.

The detection of an overflow after the addition of two binary numbers depends on whether the numbers are considered to be signed or unsigned. When two unsigned numbers are added, an overflow is detected from the end carry out of the most significant position. In the case of signed numbers, the leftmost bit always represents the sign, and negative numbers are in 2's

complement form. When two signed numbers are added, the sign bit is treated as part of the number and the end carry does not indicate an overflow.

An overflow cannot occur after an addition if one number is positive and the other is negative, since adding a positive number to a negative number produces a result that is smaller than the larger of the two original numbers. An overflow may occur if the two numbers added are both positive or both negative. To see how this can happen, consider the following example. Two signed binary numbers, +70 and +80, are stored in two 8-bit registers. The range of numbers that each register can accommodate is from binary +127 to binary -128. Since the sum of the two numbers is +150, it exceeds the capacity of the 8-bit register. This is true if the numbers are both positive or both negative. The two additions in binary are shown below together with the last two carries.

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

Note that the 8-bit result that should have been positive has a negative sign bit and the 8-bit result that should have been negative has a positive sign bit. If, however, the carry out of the sign bit position is taken as the sign bit of the result, the 9-bit answer so obtained will be correct. Since the answer cannot be accommodated within 8 bits, we say that an overflow occurred.

#### *flow detection*

An overflow condition can be detected by observing the carry into the sign bit position and the carry out of the sign bit position. If these two carries are not equal, an overflow condition is produced. This is indicated in the examples where the two carries are explicitly shown. If the two carries are applied to an exclusive-OR gate, an overflow will be detected when the output of the gate is equal to 1.

### Decimal Fixed-Point Representation

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

0100 0011 1000 0101

By representing numbers in decimal we are wasting a considerable amount of storage space since the number of bits needed to store a decimal number in a binary code is greater than the number of bits needed for its

equivalent binary representation. Also, the circuits required to perform decimal arithmetic are more complex. However, there are some advantages in the use of decimal representation because computer input and output data are generated by people who use the decimal system. Some applications, such as business data processing, require small amounts of arithmetic computations compared to the amount required for input and output of decimal data. For this reason, some computers and all electronic calculators perform arithmetic operations directly with the decimal data (in a binary code) and thus eliminate the need for conversion to binary and back to decimal. Some computer systems have hardware for arithmetic calculations with both binary and decimal data.

The representation of signed decimal numbers in BCD is similar to the representation of signed numbers in binary. We can either use the familiar signed-magnitude system or the signed-complement system. The sign of a decimal number is usually represented with four bits to conform with the 4-bit code of the decimal digits. It is customary to designate a plus with four 0's and a minus with the BCD equivalent of 9, which is 1001.

The signed-magnitude system is difficult to use with computers. The signed-complement system can be either the 9's or the 10's complement, but the 10's complement is the one most often used. To obtain the 10's complement of a BCD number, we first take the 9's complement and then add one to the least significant digit. The 9's complement is calculated from the subtraction of each digit from 9.

The procedures developed for the signed-2's complement system apply also to the signed-10's complement system for decimal numbers. Addition is done by adding all digits, including the sign digit, and discarding the end carry. Obviously, this assumes that all negative numbers are in 10's complement form. Consider the addition  $(+375) + (-240) = +135$  done in the signed-10's complement system.

$$\begin{array}{r}
 0\ 375\ (0000\ 0011\ 0111\ 0101)_{\text{BCD}} \\
 +9\ 760\ (\underline{1001\ 0111\ 0110\ 0000})_{\text{BCD}} \\
 \hline
 0\ 135\ (0000\ 0001\ 0011\ 0101)_{\text{BCD}}
 \end{array}$$

The 9 in the leftmost position of the second number indicates that the number is negative. 9760 is the 10's complement of 0240. The two numbers are added and the end carry is discarded to obtain +135. Of course, the decimal numbers inside the computer must be in BCD, including the sign digits. The addition is done with BCD adders (see Fig. 10-18).

The subtraction of decimal numbers either unsigned or in the signed-10's complement system is the same as in the binary case. Take the 10's complement of the subtrahend and add it to the minuend. Many computers have special hardware to perform arithmetic calculations directly with decimal numbers in BCD. The user of the computer can specify by programmed instructions that the arithmetic operations be performed with decimal numbers directly without having to convert them to binary.

### 3-4 Floating-Point Representation

*mantissa*

*exponent*

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

<i>Fraction</i>	<i>Exponent</i>
+0.6132789	+04

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

Floating-point is always interpreted to represent a number in the following form:

$$m \times r^e$$

Only the mantissa  $m$  and the exponent  $e$  are physically represented in the register (including their signs). The radix  $r$  and the radix-point position of the mantissa are always assumed. The circuits that manipulate the floating-point numbers in registers conform with these two assumptions in order to provide the correct computational results.

A floating-point binary number is represented in a similar manner except that it uses base 2 for the exponent. For example, the binary number +1001.11 is represented with an 8-bit fraction and 6-bit exponent as follows:

<i>Fraction</i>	<i>Exponent</i>
01001110	000100

*fraction*

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

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

*normalization*

A floating-point number is said to be *normalized* if the most significant digit of the mantissa is nonzero. For example, the decimal number 350 is normalized but 00035 is not. Regardless of where the position of the radix point is assumed to be in the mantissa, the number is normalized only if its leftmost digit is nonzero. For example, the 8-bit binary number 00011010 is not normal-

ized because of the three leading 0's. The number can be normalized by shifting it three positions to the left and discarding the leading 0's to obtain 11010000. The three shifts multiply the number by  $2^3 = 8$ . To keep the same value for the floating-point number, the exponent must be subtracted by 3. Normalized numbers provide the maximum possible precision for the floating-point number. A zero cannot be normalized because it does not have a nonzero digit. It is usually represented in floating-point by all 0's in the mantissa and exponent.

Arithmetic operations with floating-point numbers are more complicated than arithmetic operations with fixed-point numbers and their execution takes longer and requires more complex hardware. However, floating-point representation is a must for scientific computations because of the scaling problems involved with fixed-point computations. Many computers and all electronic calculators have the built-in capability of performing floating-point arithmetic operations. Computers that do not have hardware for floating-point computations have a set of subroutines to help the user program scientific problems with floating-point numbers. Arithmetic operations with floating-point numbers are discussed in Sec. 10-5.

### 3-5 Other Binary Codes

---

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

#### Gray Code

Digital systems can process data in discrete form only. Many physical systems supply continuous output data. The data must be converted into digital form before they can be used by a digital computer. Continuous, or analog, information is converted into digital form by means of an analog-to-digital converter. The reflected binary or *Gray code*, shown in Table 3-5, is sometimes used for the converted digital data. The advantage of the Gray code over straight binary numbers is that the Gray code changes by only one bit as it sequences from one number to the next. In other words, the change from any number to the next in sequence is recognized by a change of only one bit from 0 to 1 or from 1 to 0. A typical application of the Gray code occurs when the analog data are represented by the continuous change of a shaft position. The shaft is partitioned into segments with each segment assigned a number. If adjacent segments are made to correspond to adjacent Gray code numbers, ambiguity is reduced when the shaft position is in the line that separates any two segments.

Gray code counters are sometimes used to provide the timing sequences

## UNIT III

**Data Representation:** Data types, Complements, Fixed Point Representation, Floating Point Representation.

**Computer Arithmetic:** Addition and Subtraction, Multiplication Algorithms, Division Algorithms, Floating – Point Arithmetic Operations, Decimal Arithmetic Unit, Decimal Arithmetic Operations.



# Computer Arithmetic

## ADDITION AND SUBTRACTION:

### Addition and Subtraction with Signed–Magnitude Data:

Designate the magnitude of the two numbers by  $A$  and  $B$ .

When the signed numbers are added or subtracted, we find that there are eight different conditions to consider, depending on the sign of the numbers and the operation performed. These conditions are listed in the first column of the table. The other columns in the table show the actual operation to be performed with the *magnitude* of the numbers. The last column is needed to prevent a negative Zero.

**TABLE 10-1** Addition and Subtraction of Signed-Magnitude Numbers

Operation	Add Magnitudes	Subtract Magnitudes		
		When $A > B$	When $A < B$	When $A = B$
$(+A) + (+B)$	$+(A + B)$			
$(+A) + (-B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(-A) + (+B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$
$(-A) + (-B)$	$-(A + B)$			
$(+A) - (+B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(+A) - (-B)$	$+(A + B)$			
$(-A) - (+B)$	$-(A + B)$			
$(-A) - (-B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$

**Addition algorithm:** when the signs of  $A$  and  $B$  are identical, add the two magnitudes and attach the sign of  $A$  to the result. When the signs of  $A$  and  $B$  are different, compare the magnitudes and subtract the smaller number from the larger.

**Subtraction algorithm:** when the signs of  $A$  and  $B$  are different, add the two magnitudes and attach the sign of  $A$  to the result. When the signs of  $A$  and  $B$  are identical, compare the magnitudes and subtract the smaller number from the larger.

## Hardware Implementation:

Let **A** and **B** be two registers that hold the magnitudes of the numbers.

**A<sub>s</sub>** and **B<sub>s</sub>** be two flip flops that hold the corresponding signs.

**M** and **S** be the mode control and sum of adder.

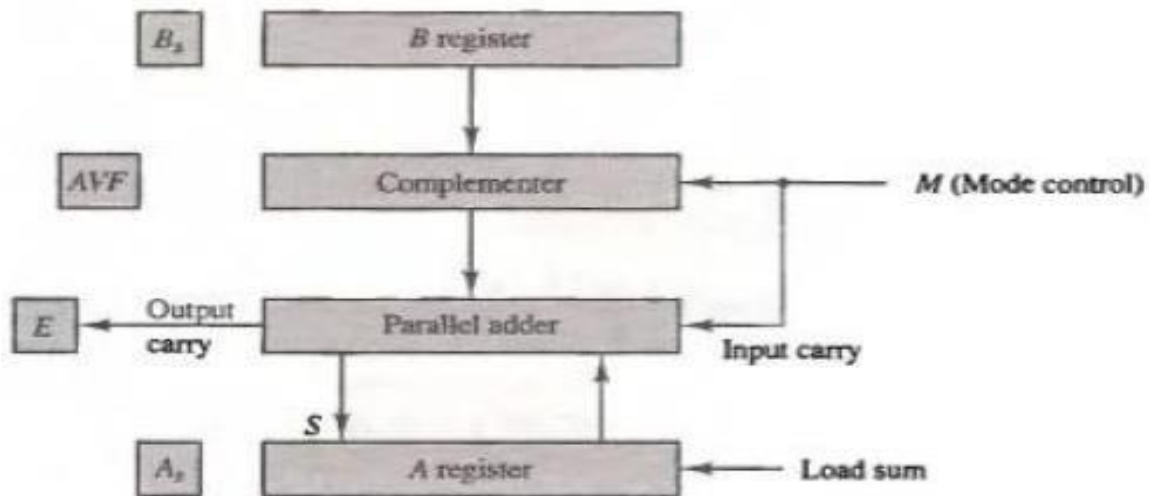
The result of the operation may be transferred to a third register: however, a saving is achieved if the result is transferred into A and A<sub>s</sub>. Thus A and A<sub>s</sub> together form an accumulator register.

Consider now the hardware implementation of the algorithms above.

1. A parallel-adder is needed to perform the micro operations **A+B**.
2. A comparator circuit is needed to establish if **A>B**, **A=B**, or **A<B**.
3. Two parallel-subtractor circuits are needed to perform the micro operations **A-B** and **B-A**.

The sign relationship can be determined from an exclusive-OR gate with A<sub>s</sub> and B<sub>s</sub> as inputs.

- Subtraction is done by adding A to the 2's complement of B. The output carry is transferred to flip-flop E, where it can be checked to determine the relative magnitudes of the two numbers. The add-overflow flip-flop AVF holds the overflow bit when A and B are added.
- The addition of A plus B is done through the parallel adder. The S (sum) output of the adder is applied to the input of the A register.
- The complementer provides an output of B or the complement of B depending on the state of the mode of control M.
- The complementer consists of exclusive-OR gates and the parallel adder consists of full-adder circuits as shown in fig. The M signal is also applied to the input carry of the adder.



**Figure 10-1** Hardware for signed-magnitude addition and subtraction.

- When  $M=0$ , the output of  $B$  is transferred to the adder, the input carry is 0, and the output of the adder is equal to the sum  $A+B$ .
- When  $M=1$ , the 1's complement of  $B$  is applied to the adder, the input carry is 1, and output  $S=A+\bar{B}+1$ . This is equal to  $A$  plus the 2's complement of  $B$ , Which is equivalent to the subtraction  $A-B$ .

#### Hardware Algorithm:

- The two signs  $A_s$  and  $B_s$  are compared by an exclusive-OR gate. If the output of the gate is 0, the signs are identical; if it is 1, the signs are different.
- For an **add** operation, identical signs dictate that magnitudes be added, different signs dictate that the magnitudes be subtracted.
- For **subtract** operation, identical signs dictate that magnitudes be subtracted, different signs dictate that the magnitudes be added.
- The magnitudes are added with a micro operation  $EA \leftarrow A+B$ , Where  $EA$  is a Register that combines  $E$  and  $A$ . The carry in  $E$  after the addition constitutes an overflow if it is equal to 1. The value of  $E$  is transferred into the add-overflow flip-flop  $AVF$ .

- The magnitudes are subtracted by adding A to the 2's complement of B. No overflow can occur if the numbers are subtracted so AVF is cleared to 0.
- A **1 in E** indicates that  $A \geq B$  and the number in A is the correct result. If this is Zero, the sign  $A_s$  must be made positive to avoid a negative Zero.
- A **0 in E** indicates that  $A < B$ . For this case it is necessary to take the 2's complement of the value in A. This operation can be done with one micro operation  $A \leftarrow \bar{A} + 1$ .

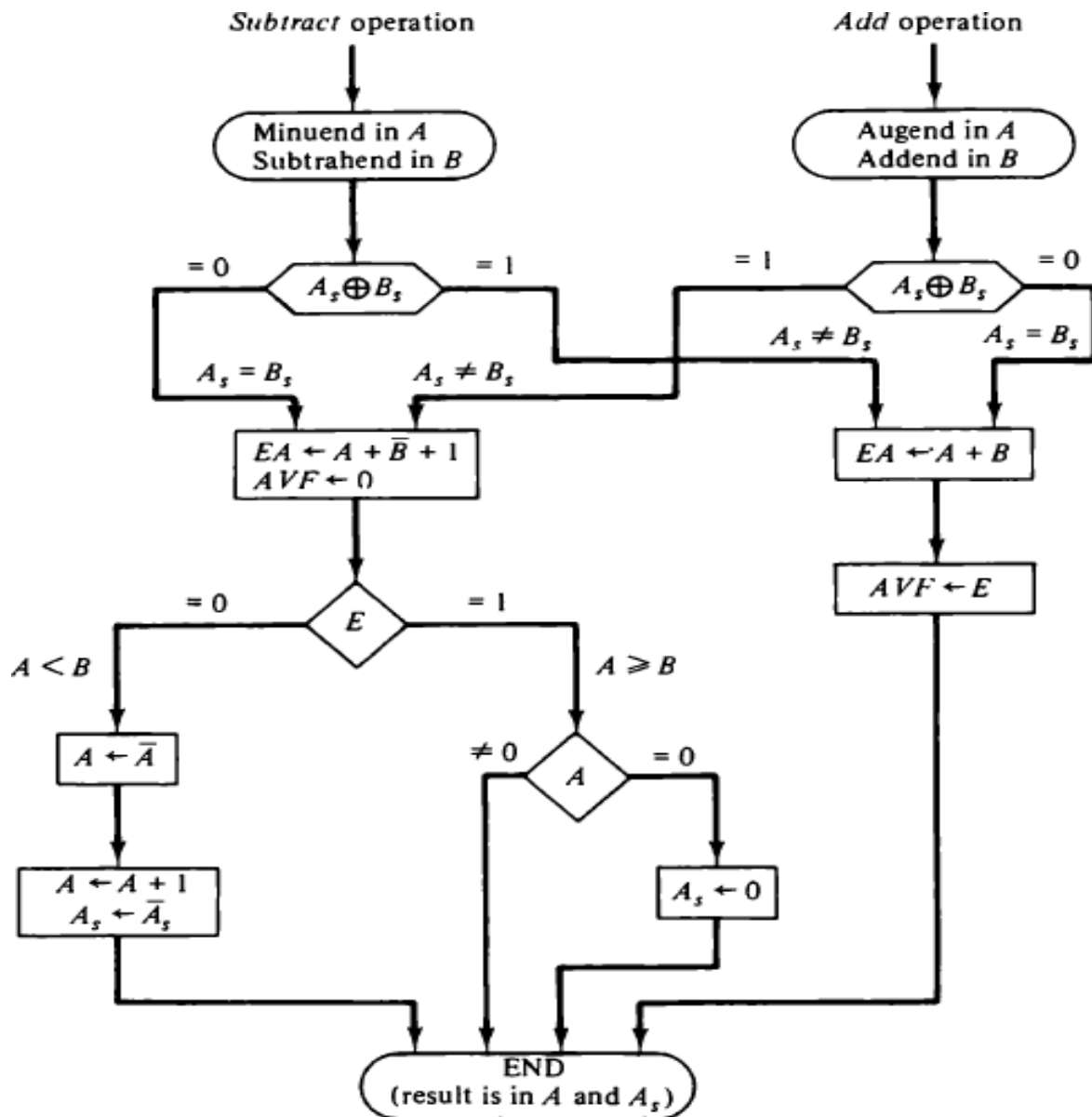


Figure 10-2 Flowchart for add and subtract operations.

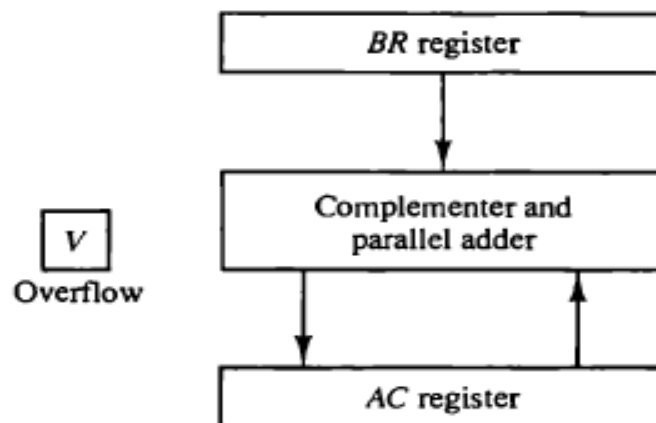
### Addition and Subtraction with Signed-2's :

- The leftmost bit of a binary number represents the sign bit: 0 for positive and 1 for negative.
- If the sign bit is 1, the entire number is represented in 2's complement form.

**Ex:** +33 is Represented as 00100001 and -33 as 11011111. Note that 11011111 is the 2's complement of 00100001.

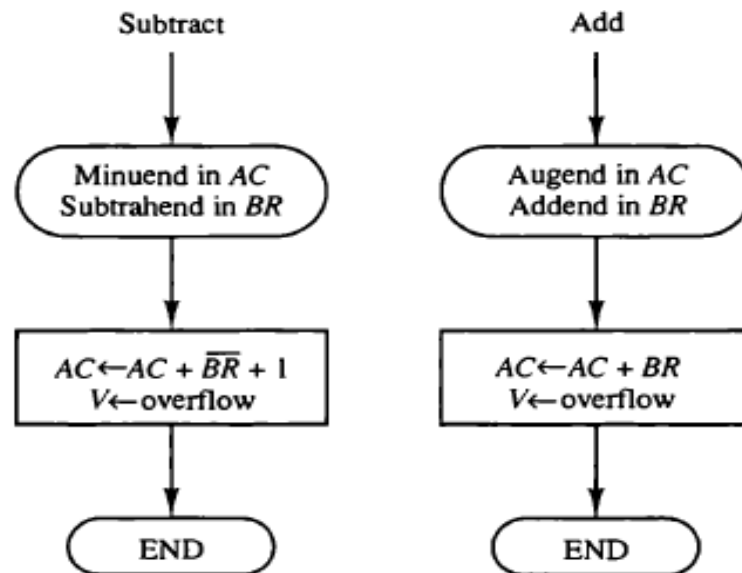
- The **addition** of two numbers in signed-2's complement form consists of adding the numbers with sign bits treated the same as the others bits of the number. A carry-out of the sign-bit position is discarded.
- The **subtraction** consists of first taking the 2's complement of the subtrahend and then adding it to the minuend.

**Figure 10-3** Hardware for signed-2's complement addition and subtraction.



- When two numbers of  $n$  digits each are added and the sum occupies  $n+1$  digits, we say that an overflow occurred.
- An **overflow** can be detected by inspecting the last two carries out of the addition. When the two carries are applied to an exclusive-OR gate, the overflow is detected when the output of the gate is equal to 1.

- We name the A register AC (accumulator) and the B register BR. The leftmost bit in AC and BR represent the sign bits of the numbers. The two sign bits are added or subtracted together with the other bits the complementer and parallel adder.
- The overflow flip-flop V is set to 1 if there is an overflow. The output carry in this case is discarded.
- The sum is obtained by adding the contents of AC and BR (including their sign bits). The overflow bit V is set to 1 if the exclusive-OR of the last two carries is 1, and it is cleared to 0 otherwise.
- The subtraction operation is accomplished by adding the content of AC to the 2's complement of BR. Taking the 2's complement of BR has the effect of changing a positive number to negative and vice-versa.



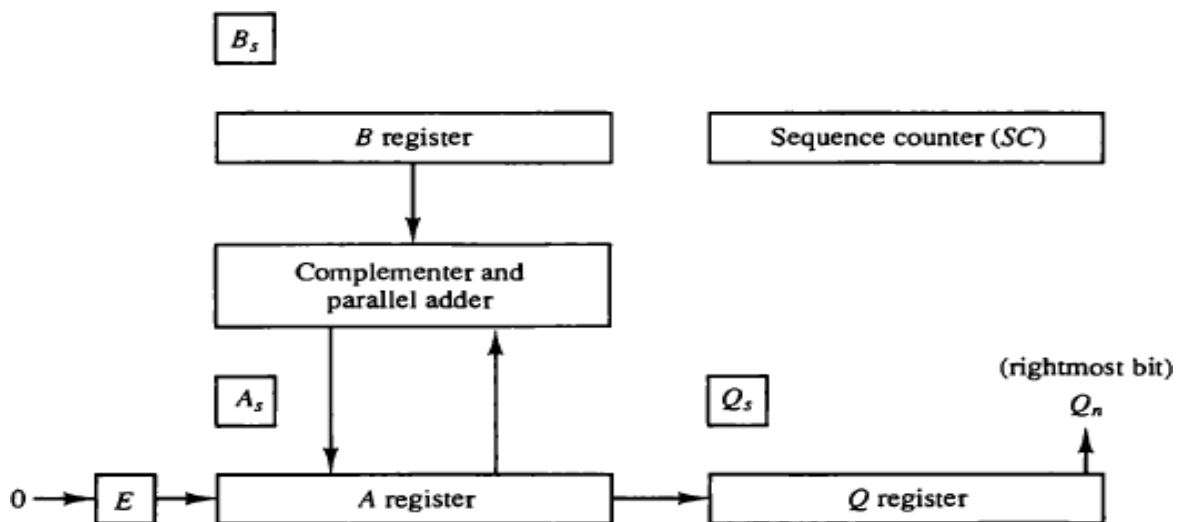
**Figure 10-4** Algorithm for adding and subtracting numbers in signed-2's complement representation.

### MULTIPLICATION ALGORITHM:

$$\begin{array}{r} 23 \quad 10111 \quad \text{Multiplicand} \\ 19 \quad \times 10011 \quad \text{Multiplier} \\ \hline 10111 \\ 10111 \\ 00000 \quad + \\ 00000 \\ 10111 \\ \hline 437 \quad 110110101 \quad \text{Product} \end{array}$$

The process consists of looking at successive bits of the multiplier, least significant bit first. If the multiplier bit is a 1, the multiplicand is copied down; otherwise, zeros are copied down. The numbers copied down in successive lines are shifted one position to the left from the previous number. Finally, the numbers are added and their sum forms the product.

### Hardware Implementation for Signed–Magnitude Data:



The hardware for multiplication consists of the equipment shown in fig. plus two more registers. These registers together with registers **A** and **B** are shown in fig.

The multiplicand is in register **B** and its sign in  $B_s$ .

The multiplier is stored in the **Q** register and its sign in  $Q_s$ .

- The sequence counter **SC** is initially set to a number equal to the number of bits in the multiplier. The counter is decremented by 1 after forming each partial product. When the content of the counter reaches Zero, the product is formed and the process stops.
- The sum of A and B forms a partial product which is transferred to the EA register. Both partial product and multiplier are shifted to the right. This shift Will be denoted by the statement **shr EAQ** to designate the right shift depicted in fig.
- The least significant bit of A is shifted into the most significant position of Q, The bit from E is shifted into the most significant position of A, and 0 is shifted into E.
- After the shift, one bit of the partial product is shifted into Q, pushing the multiplier bits one position to the right. In this manner, the rightmost flip–flop in register Q, designated by  $Q_n$ , will hold the bit of the multiplier.

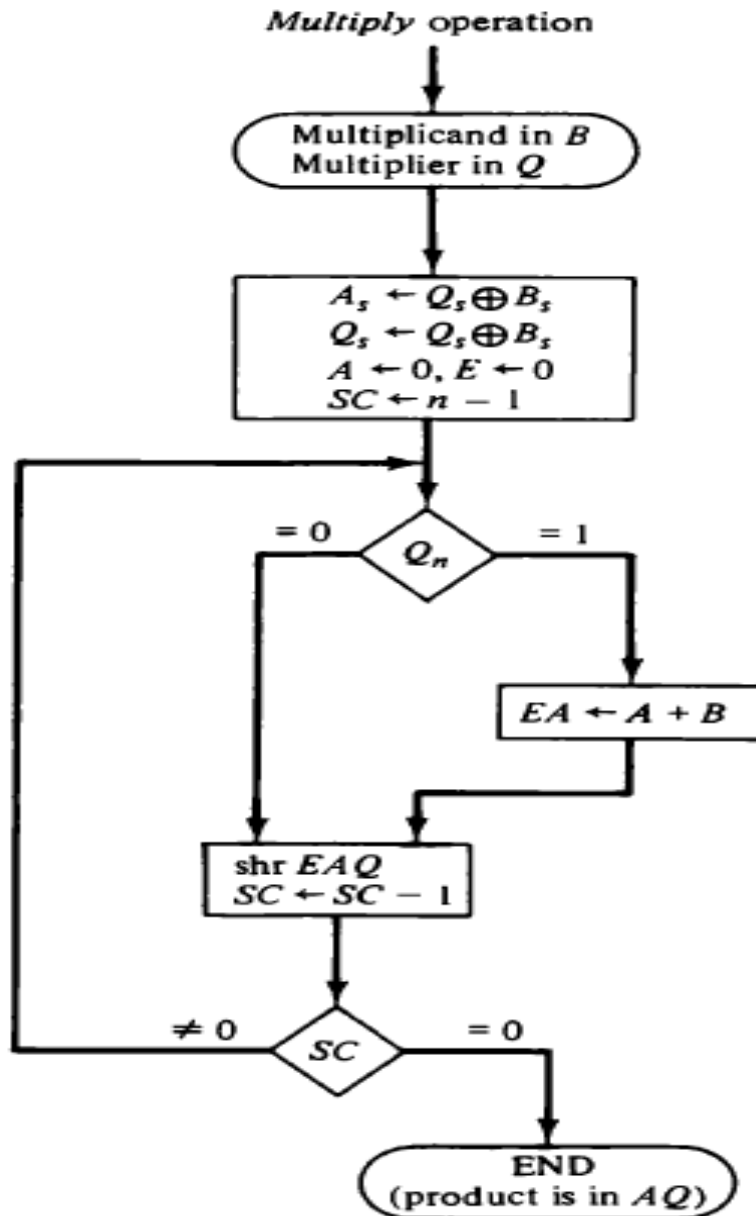
#### **Hardware Algorithm:**

- Initially, the multiplicand is in B and the multiplier in Q. Their corresponding signs are in  $B_s$  and  $Q_s$  respectively. The signs are compared, and both A and Q are set to correspond to the sign of the product since a double–length product will be stored in registers A and Q.
- Registers A and E are cleared and the sequence counter SC is set to a number equal to the number of bits of the multiplier.
- We are assuming here that operands are transferred to registers from a memory unit that has words of n bits. Since an operand must be stored with its sign, one bit of the word will be occupied by the sign and the magnitude will consist of n-1 bits.
- After the initialization, the low–order bit of the multiplier in  $Q_n$  is tested. If it is a 1, the multiplicand in B is added to the present partial product in A. If it is a 0, nothing is done. Register EAQ is then shifted once to the right to form the new partial product.
- The sequence counter is decremented by 1 and its new value checked. If it is not equal to Zero, the process is repeated and a new partial product formed.



- The process stops when  $SC=0$ . Note that the partial product formed in A is shifted into Q one bit at a time and eventually replaces the multiplier. The final product is available in both A and Q, with A holding the most significant bits and Q holding the least significant bits.

**Figure 10-6** Flowchart for multiply operation.



EX:  $B = +23$  (10111)

$Q = +19$  (10011)

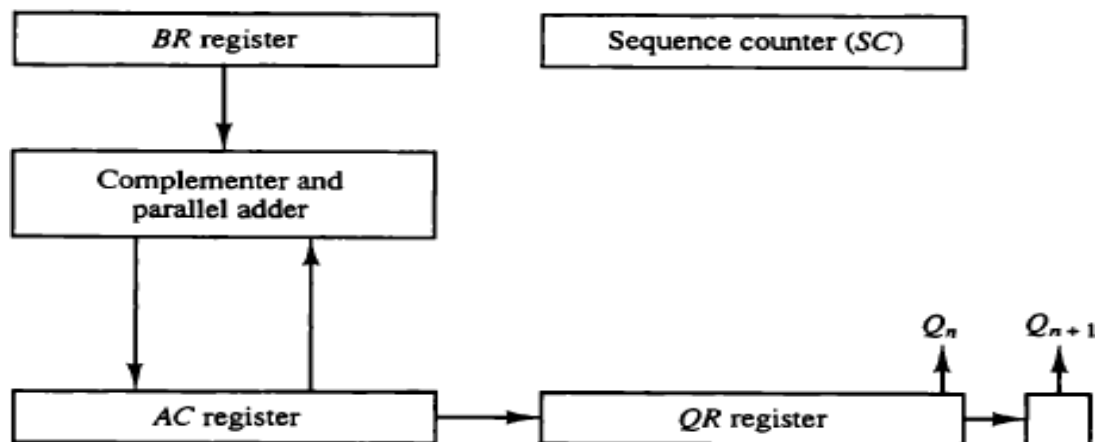
TABLE 10-2 Numerical Example for Binary Multiplier

Multiplicand $B = 10111$	$E$	$A$	$Q$	$SC$
Multiplier in $Q$	0	00000	10011	101
$Q_n = 1$ ; add $B$		<u>10111</u>		
First partial product	0	10111		
Shift right $EAQ$	0	01011	11001	100
$Q_n = 1$ ; add $B$		<u>10111</u>		
Second partial product	1	00010		
Shift right $EAQ$	0	10001	01100	011
$Q_n = 0$ ; shift right $EAQ$	0	01000	10110	010
$Q_n = 0$ ; shift right $EAQ$	0	00100	01011	001
$Q_n = 1$ ; add $B$		<u>10111</u>		
Fifth partial product	0	11011		
Shift right $EAQ$	0	01101	10101	000
Final product in $AQ = 0110110101$				

### BOOTH'S MULTIPLICATION ALGORITHM:

Booth Multiplication gives a procedure for multiplying binary integers in signed-2's complement representation.

Figure 10-7 Hardware for Booth algorithm.



**BR ---** is used to store multiplicand

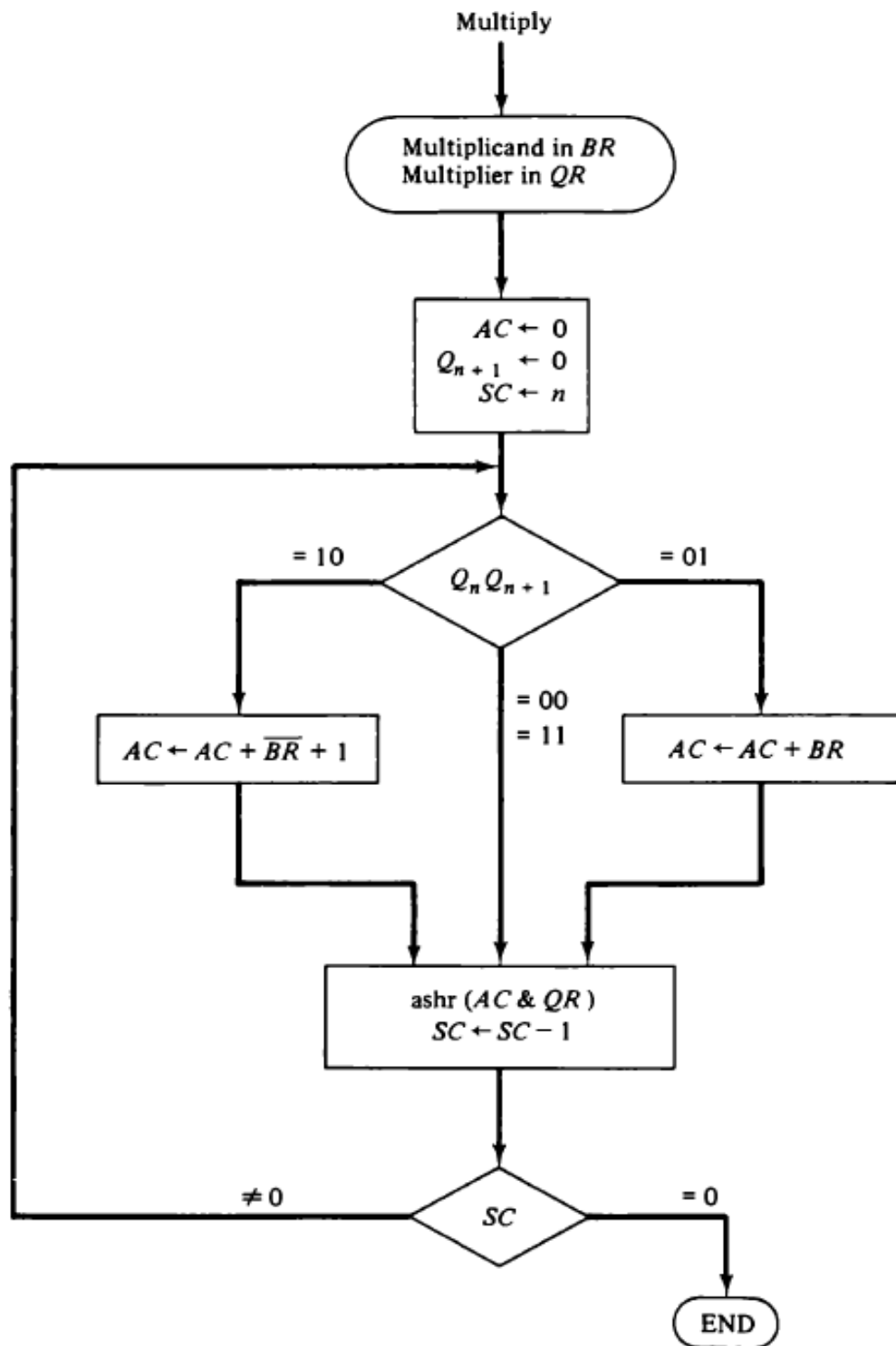
**QR ---** is used to store multiplier

**At the end, AC & QR ---** are used store result.

- The algorithm works for positive or negative multipliers in 2's complement representation. This is because a negative multiplier ends with a string of 1's and the last operation will be a subtraction of the appropriate weight.

**Example:** A multiplier equal to -14 is represented in 2's complement as 110010 and is treated  $-2^4+2^2-2^1=-14$ .

- AC and the appended bit  $Q_{n+1}$  are initially cleared to 0 and the sequence counter SC is set to a number n equal to the number of bits in the multiplier.
- The two bits of the multiplier in  $Q_n$  and  $Q_{n+1}$  are equal to 10, this requires a subtraction of the multiplicand from the partial product in AC.
- If the two bits are equal to 01, this requires addition of the multiplicand from the partial product in AC.
- When the two bits are equal, the partial product does not change.
- An overflow cannot occur because the addition and subtraction of the multiplicand follow each other.
- As a consequence, the two numbers that are added always have opposite signs, a condition that excludes an overflow.
- The next step is to shift right the partial product and the multiplier (including bit  $Q_{n+1}$ ). This is an arithmetic shift right (ashr) operation which shifts AC and QR to the right and leaves the sign bit in AC unchanged.
- The sequence counter is decremented and the computational loop is repeated n times.



**Figure 10-8** Booth algorithm for multiplication of signed-2's complement numbers.

EX:  $BR = -9$  (10111)       $\overline{BR} + 1 = 01001$

$Q = -13$  (10011)

**TABLE 10-3** Example of Multiplication with Booth Algorithm

$Q_n Q_{n+1}$	$BR = 10111$ $\overline{BR} + 1 = 01001$	$AC$	$QR$	$Q_{n+1}$	$SC$
	Initial	00000	10011	0	101
1 0	Subtract $BR$	<u>01001</u> 01001			
	ashr	00100	11001	1	100
1 1	ashr	00010	01100	1	011
0 1	Add $BR$	<u>10111</u> 11001			
	ashr	11100	10110	0	010
0 0	ashr	11110	01011	0	001
1 0	Subtract $BR$	<u>01001</u> 00111			
	ashr	00011	10101	1	000

### ARRAY MULTIPLIER:

This is a fast way of multiplying two numbers since all it takes is the time for the signals to propagate through the gates that form the multiplication array.

An array multiplier requires a large numbers of gates, and for this reason it was not economical until the development of integrated circuits.

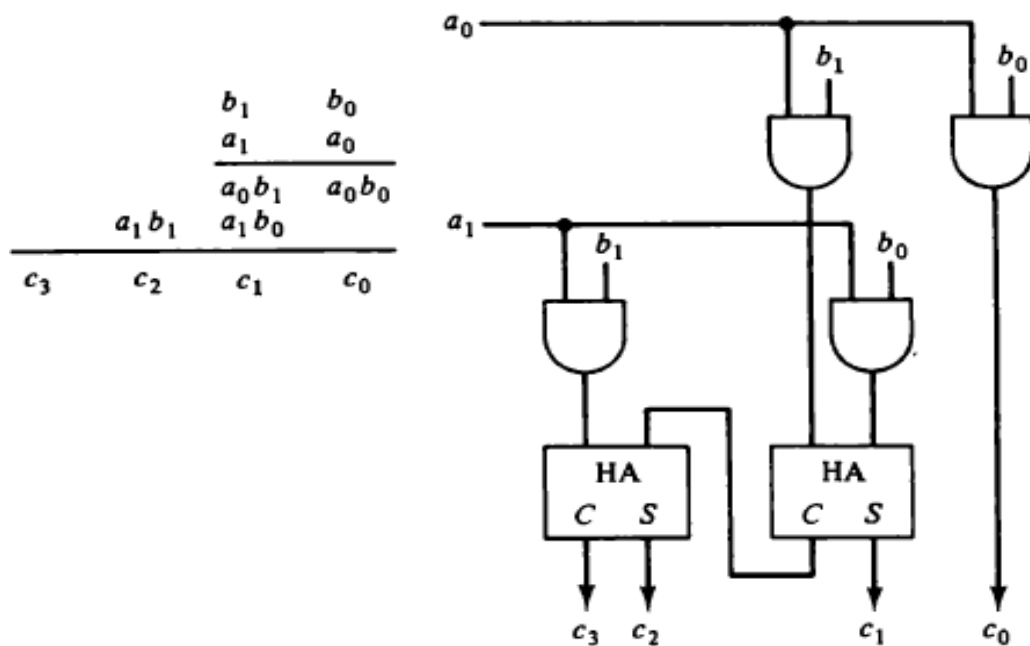
**To see how an array multiplier can be implemented with a combinational circuit:**

- Consider the multiplication of the two 2-bit numbers as shown in figure.
- The multiplicand bits are  $b_1$  and  $b_0$ , the multiplier bits are  $a_1$  and  $a_0$ , and the product is  $c_3c_2c_1c_0$ . The first partial product is formed by multiplying  $a_0$  by  $b_1b_0$ . The multiplication of two bits such as  $a_0$  and  $b_0$  produces a 1 if both bits are 1; otherwise, it produces a 0. This is

identical to an AND operation and can be implemented with an AND gate. As shown in the diagram, the first partial product is formed by means of two AND gates.

- The second partial product is formed by multiplying  $a_1$  by  $b_1b_0$  and is shifted one position to the left.
- The two partial products are added with two half adder (HA) circuits. Usually, there are more bits in the partial products and it will be necessary to use full –adders to produce the sum.
- Note that the least significant bit of the product does not have to go through an adder since it is formed by the output of the first AND gate.
- Consider a multiplier circuit that multiplies a binary number of four bits with a number of three bits. Let the multiplicand be represented by  $b_3b_2b_1b_0$  and the multiplier by  $a_2a_1a_0$ . Since  $k=4$  and  $j=3$ , we need 12 AND gates and two 4–bit adders to produce a product of seven bits. The logic diagram of the multiplier shown in fig.

**Figure 10-9** 2-bit by 2-bit array multiplier.



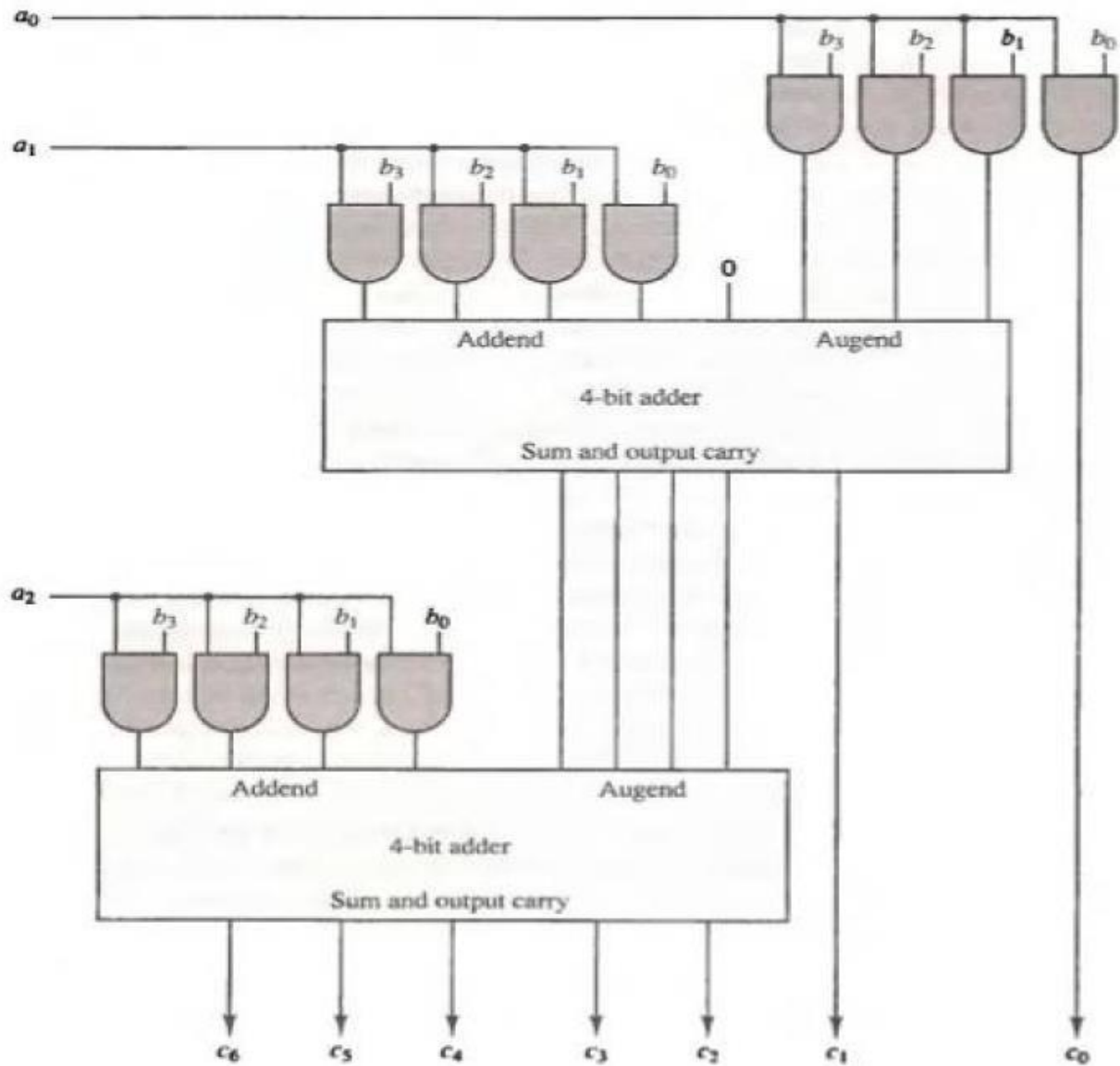


Figure 10-10 4-bit by 3-bit array multiplier.

## DIVISION ALGORITHM:

**AQ** --- is used to store dividend

**B** --- is used to store divisor

At the end, **Q** --- is used to store quotient

**A** --- is used to store remainder.

**Figure 10-11** Example of binary division.

Divisor:	11010	Quotient = <i>Q</i>
<i>B</i> = 10001	$\overline{)0111000000}$	Dividend = <i>A</i>
	01110	5 bits of <i>A</i> < <i>B</i> , quotient has 5 bits
	011100	6 bits of <i>A</i> ≥ <i>B</i>
	<u>-10001</u>	Shift right <i>B</i> and subtract; enter 1 in <i>Q</i>
	-010110	7 bits of remainder ≥ <i>B</i>
	<u>--10001</u>	Shift right <i>B</i> and subtract; enter 1 in <i>Q</i>
	--001010	Remainder < <i>B</i> ; enter 0 in <i>Q</i> ; shift right <i>B</i>
	---010100	Remainder ≥ <i>B</i>
	<u>----10001</u>	Shift right <i>B</i> and subtract; enter 1 in <i>Q</i>
	----000110	Remainder < <i>B</i> ; enter 0 in <i>Q</i>
	-----00110	Final remainder

## Hardware Implementation and Algorithm:

### Divide Overflow:

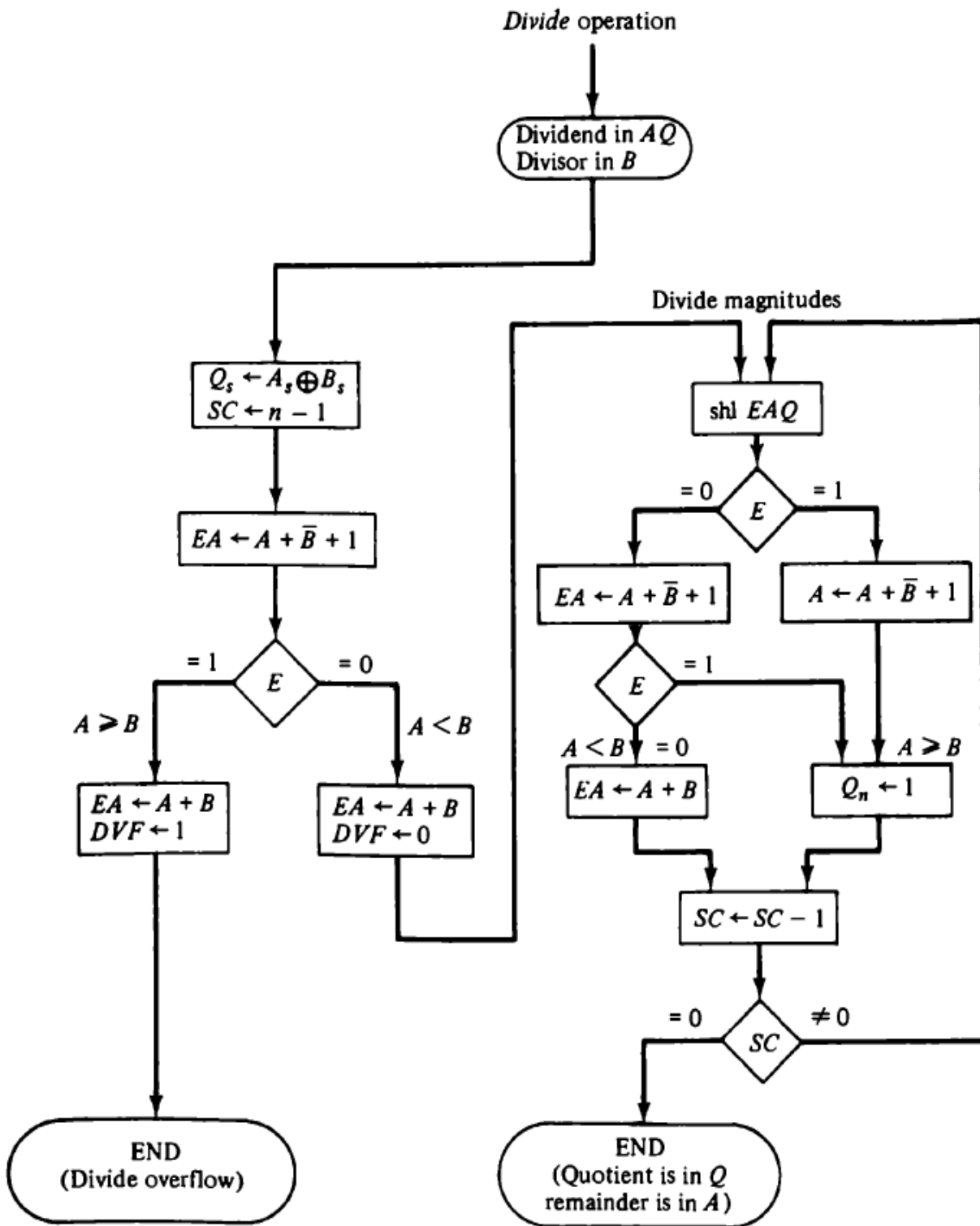
A **divide-overflow** condition occurs if the high-order half bits of the dividend constitute a number greater than or equal to the divisor. Another problem associated with division is the fact that a division by zero must be avoided. Overflow condition is usually detected when a special flip-flop is set. We will call it a divide-overflow flip-flop and label it DVF.

- The dividend is in A and Q and the divisor in B. The sign of the result is transferred into  $Q_s$  to be part of the quotient. A constant is set into the sequence counter SC to specify the number of bits in the quotient.



- A **divide–overflow** condition is tested by subtracting the divisor in B from half of the bits of the dividend stored in A.
  - If  $A \geq B$ , the divide–overflow flip–flop DVF is set and the operation is terminated prematurely.
  - If  $A < B$ , no divide overflow occurs so the value of the dividend is restored by adding B to A.
- Register EAQ is now shifted to the left with 0 inserted into  $Q_n$  and the previous value of E lost. The divisor is stored in the B register and the double–length dividend is stored in registers A and Q.
- The dividend is shifted to the left and the divisor is subtracted by adding its 2's complement value.
- The information about the relative magnitude is available in E.
  - If  $E=1$ , it signifies that  $A \geq B$ . A quotient bit 1 is inserted into  $Q_n$  and the partial remainder is shifted to the left to repeat the process.
  - If  $E=0$ , it signifies that  $A < B$  so the quotient in  $Q_n$  remains a 0 (inserted during the shift). The value of B is then added to restore the partial remainder in A to its previous value.
- The partial remainder is shifted to the left and the process is repeated again until all five quotient bits are formed. Note that while the partial remainder is shifted left, the quotient is in Q and the final remainder is in A.

Figure 10-13 Flowchart for divide operation.



**EX:**  $AQ$  --- is used to store dividend ( 01110 00000 = 448 )

$B$  --- is used to store divisor ( 10001 = 17 )

$Q$  --- is used to store quotient ( 11010 = 26 )

$A$  --- is used to store remainder. ( 00110 = 6 )

**Divisor  $B = 10001$ ,**

**$\overline{B} + 1 = 01111$**

	<u><math>E</math></u>	<u><math>A</math></u>	<u><math>Q</math></u>	<u><math>SC</math></u>
Dividend:		01110	00000	5
shl $EAQ$	0	11100	00000	
add $\overline{B} + 1$		<u>01111</u>		
$E = 1$	1	01011		
Set $Q_n = 1$	1	01011	00001	4
shl $EAQ$	0	10110	00010	
Add $\overline{B} + 1$		<u>01111</u>		
$E = 1$	1	00101		
Set $Q_n = 1$	1	00101	00011	3
shl $EAQ$	0	01010	00110	
Add $\overline{B} + 1$		<u>01111</u>		
$E = 0$ ; leave $Q_n = 0$	0	11001	00110	
Add $B$		<u>10001</u>		
				2
Restore remainder	1	01010		
shl $EAQ$	0	10100	01100	
Add $\overline{B} + 1$		<u>01111</u>		
$E = 1$	1	00011		
Set $Q_n = 1$	1	00011	01101	1
shl $EAQ$	0	00110	11010	
Add $\overline{B} + 1$		<u>01111</u>		
$E = 0$ ; leave $Q_n = 0$	0	10101	11010	
Add $B$		<u>10001</u>		
Restore remainder	1	00110	11010	0
Neglect $E$				
Remainder in $A$ :		00110		
Quotient in $Q$ :			11010	

**Figure 10-12** Example of binary division with digital hardware.

## FLOATING-POINT ARITHMETIC OPERATIONS:

A floating-point number in computer registers contains of two parts: a mantissa 'm' and an exponent 'e'. The two parts represent a number obtained from multiplying 'm' times a radix 'r' raised to the value of e; thus

$$m \times r^e$$

The mantissa may be a fraction or an integer. The location of the radix point and the value of the radix 'r' are assumed and are not included in the registers.

For example, assume a fraction representation and a radix 10. The decimal number 537.25 is represented in a register with m=53725 and e=3 and is interpreted to represent the floating-point number

$$.53275 \times 10^3$$

Adding or subtracting two numbers requires first an alignment of the radix point since the exponent parts must be made equal before adding or subtracting the mantissas. The alignment is done by shifting one mantissa while it's exponent is adjusted until it is equal to the other exponent. Consider the sum of the following floating-point numbers.

$$\begin{array}{r} .5372400 \times 10^2 \\ + .1580000 \times 10^{-1} \end{array}$$

It is necessary that the two exponents be equal before the mantissas can be added. We can either shift the first number three positions to the left, or shift the second number three positions to the right. When the mantissas are stored in registers, shifting to the left causes a loss of least significant digits. The second method is preferable because it only reduces the accuracy, while the first method may cause an error. The usual alignment procedure is to shift the mantissa that has the smaller exponent to the right by a number of places equal to the difference between the exponents. After this is done, the mantissas can be added.

$$\begin{array}{r} .5372400 \times 10^2 \\ + .0001580 \times 10^2 \\ \hline .5373980 \times 10^2 \end{array}$$

When two normalized mantissas are added, the sum may contain an overflow digit. An **overflow** can be corrected easily by shifting the sum once to the right and incrementing the exponent. When two numbers are subtracted, the result may contain most significant zeros as shown in the following example.

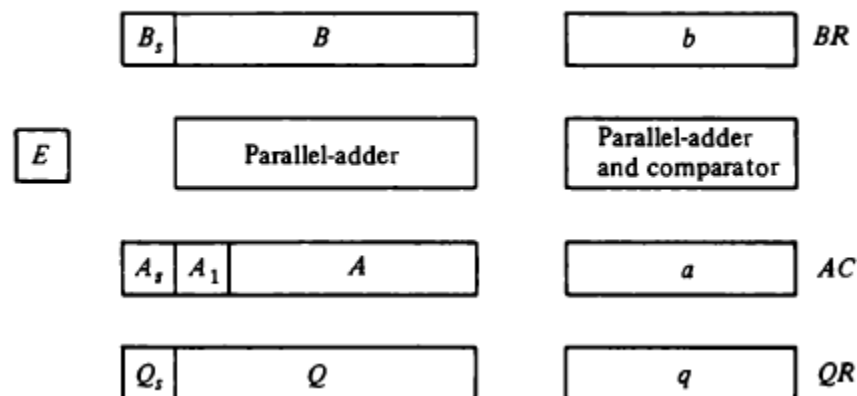
$$\begin{array}{r} .56780 \times 10^5 \\ - .56430 \times 10^5 \\ \hline .00350 \times 10^5 \end{array}$$

A floating point number that has a 0 in the most significant position of the mantissa is said to have an **underflow**. To normalize a number that contains an underflow, it is necessary to shift the mantissa to the left and decrement the exponent until a nonzero digit appears in the first position. In the example above, it is necessary to shift left twice to obtain  $.3500 \times 10^3$

Floating-point multiplication and division do not require an alignment of the mantissas. The product can be formed by multiplying the two mantissas and adding the exponents. Division is accomplished by dividing the mantissas and subtracting the exponents.

**Register Configuration:** There are three registers, BR, AC, and QR. Each register is subdivided into two parts. mantissa and exponent.

**Figure 10-14** Registers for floating-point arithmetic operations.



- It is assumed that each floating-point number has a mantissa in signed-magnitude representation and a biased exponent. Thus the AC has a mantissa whose sign is in  $A_s$ , and a magnitude that is in  $A$ .

- The exponent is in the part of the register denoted by the lowercase letter symbol  $a$ .
- The diagram shows explicitly the most significant bit of  $A$ , labeled by  $A_1$ . The bit in this position must be a 1(one) for the number to be normalized. Note that the symbol  $AC$  represents the entire register, that is, the concatenation of  $A_s$ ,  $A$ , and  $a$ .
- Similarly, register  $BR$  is subdivided into  $B_s$ ,  $B$ , and  $b$ , and  $QR$  into  $Q_s$ ,  $Q$ , and  $q$ .
- A parallel adder adds the two mantissas and transfers the sum into  $A$  and the carry into  $E$ .

### **Addition & Subtraction:**

During addition and subtraction, the two floating-point operands are in  $AC$  and  $BR$ . The sum or difference is formed in the  $AC$ . The algorithm can be divided into four consecutive parts:

1. Check for zeros.
  2. Align the mantissas.
  3. Add or subtract the mantissas.
  4. Normalize the result.
- If  $BR$  is equal to zero, the operation is terminated, with the value in the  $AC$  being the result. If  $AC$  is equal to zero, we transfer the content of  $BR$  into  $AC$  and complement its sign if the numbers are to be subtracted. If neither number is equal to zero, we proceed to align the mantissas.
  - If the two exponents are equal, we go to perform the arithmetic operation. If the exponents are not equal, the mantissa having the smaller exponent is shifted to the right and its exponent incremented. This process is repeated until the two exponents are equal.
  - The addition and subtraction of the two mantissas is identical to the fixed-point addition and subtraction algorithm presented in fig.

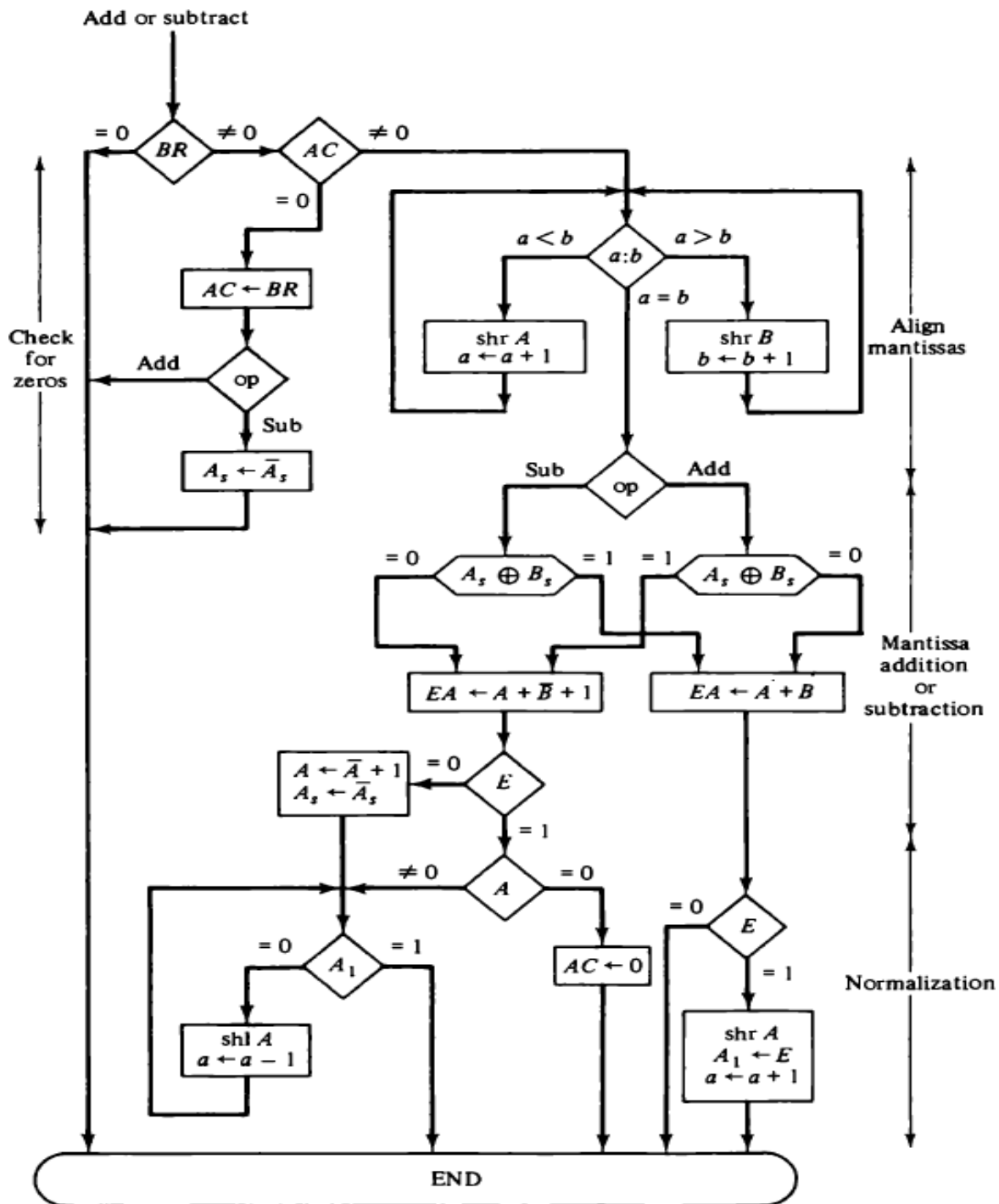


Figure 10-15 Addition and subtraction of floating-point numbers.

- The magnitude part is added or subtracted depending on the operation and signs of the two mantissas.

- If an overflow occurs when the magnitudes are added, it is transferred into flip-flop E. If E is equal to '1', the bit is transferred into A1 and all other bits of A are shifted right. The exponent must be incremented to maintain the correct number.
- If the magnitudes were subtracted, the result may be zero or may have an underflow. If the mantissa is zero, the entire floating-point number in the AC is made zero. Otherwise, the mantissa must have at least one bit that is equal to '1'. The mantissa has an underflow if the most significant bit in position A1 is 0. In that case, the mantissa is shifted left and the exponent decremented. The bit A1 is checked again and the process is repeated until it is equal to 1. When A1=1, the mantissa is normalized and the operation is completed.

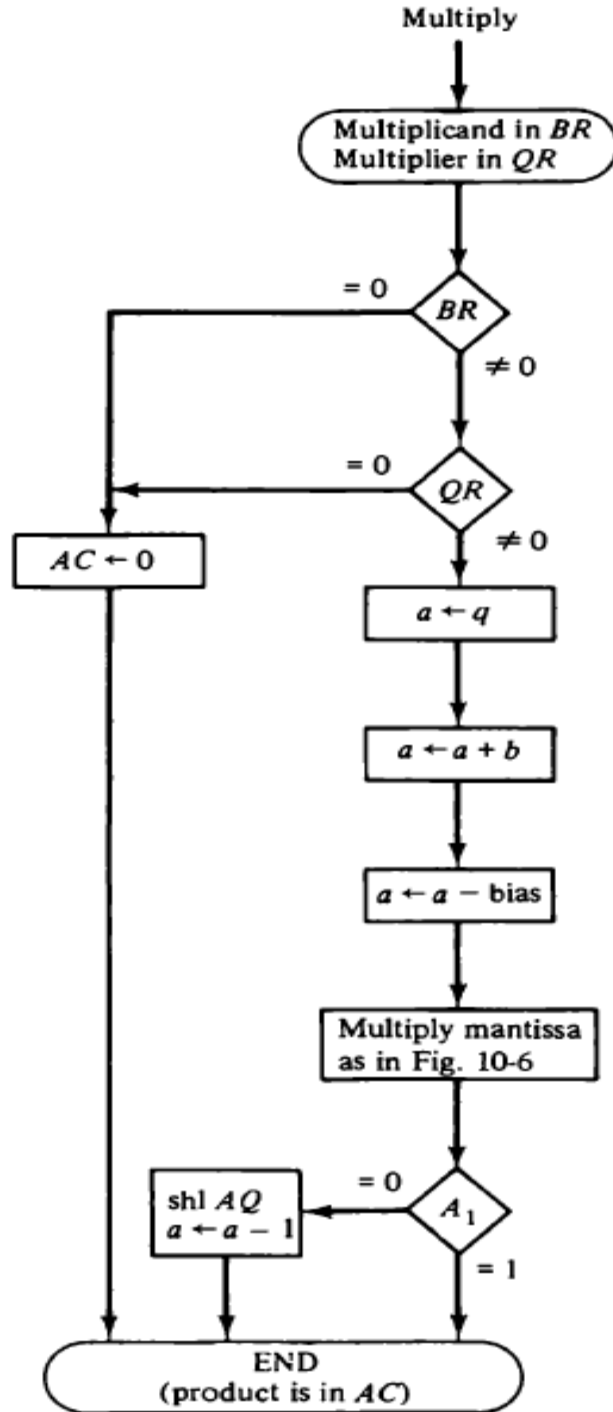
### **Multiplication:**

The multiplication of two floating-point numbers requires that we multiply the mantissas and add the exponents. No comparison of exponents or alignment of mantissas is necessary. The multiplication is subdivided into four parts:

1. Check for zeros.
  2. Add the exponents.
  3. Multiply the mantissas.
  4. Normalize the product.
- The exponent of the multiplier is in q and the adder is between exponents a and b. It is necessary to transfer the exponents from q to a, add the two exponents, and transfer the sum into a.
  - Since both exponents are biased by the addition of a constant, the exponent sum will have double this bias. The correct biased exponent for the product is obtained by subtracting the bias number from the sum.



**Figure 10-16** Multiplication of floating-point numbers.



- The product may have an underflow, so the most significant bit in A is checked. If it is a 1, the product is already normalized. If it is a 0, the mantissa in AQ is shifted left and the exponent decremented. Note that only one normalization shift is necessary.

## Division:

Floating-point division requires that the exponents be subtracted and the mantissas divided. The division algorithm is subdivided into five parts:

1. Check for zeros.
  2. Initialize registers and evaluate the sign.
  3. Align the dividend.
  4. Subtract the exponents.
  5. Divide the mantissas.
- The two operands are checked for zero. If the divisor is zero, it indicates an attempt to divide by zero, which is an illegal operation. The operation is terminated with an error message.
  - If the operands are not zero, we proceed to determine the sign of the quotient and store it in Q. The sign of the dividend in  $A_s$ , is left unchanged to be the sign of the remainder. The Q register is cleared and the sequence counter SC is set to a number equal to the number of bits in the quotient.
  - If  $A \geq B$ , it is necessary to shift A once to the right and increment the dividend exponent. Since both operands are normalized, this alignment ensures that  $A < B$ .
  - The divisor exponent is subtracted from the dividend exponent. Since both exponents were originally biased, the subtraction operation gives the difference without the bias. The bias is then added and the result transferred into q because the quotient is formed in QR.

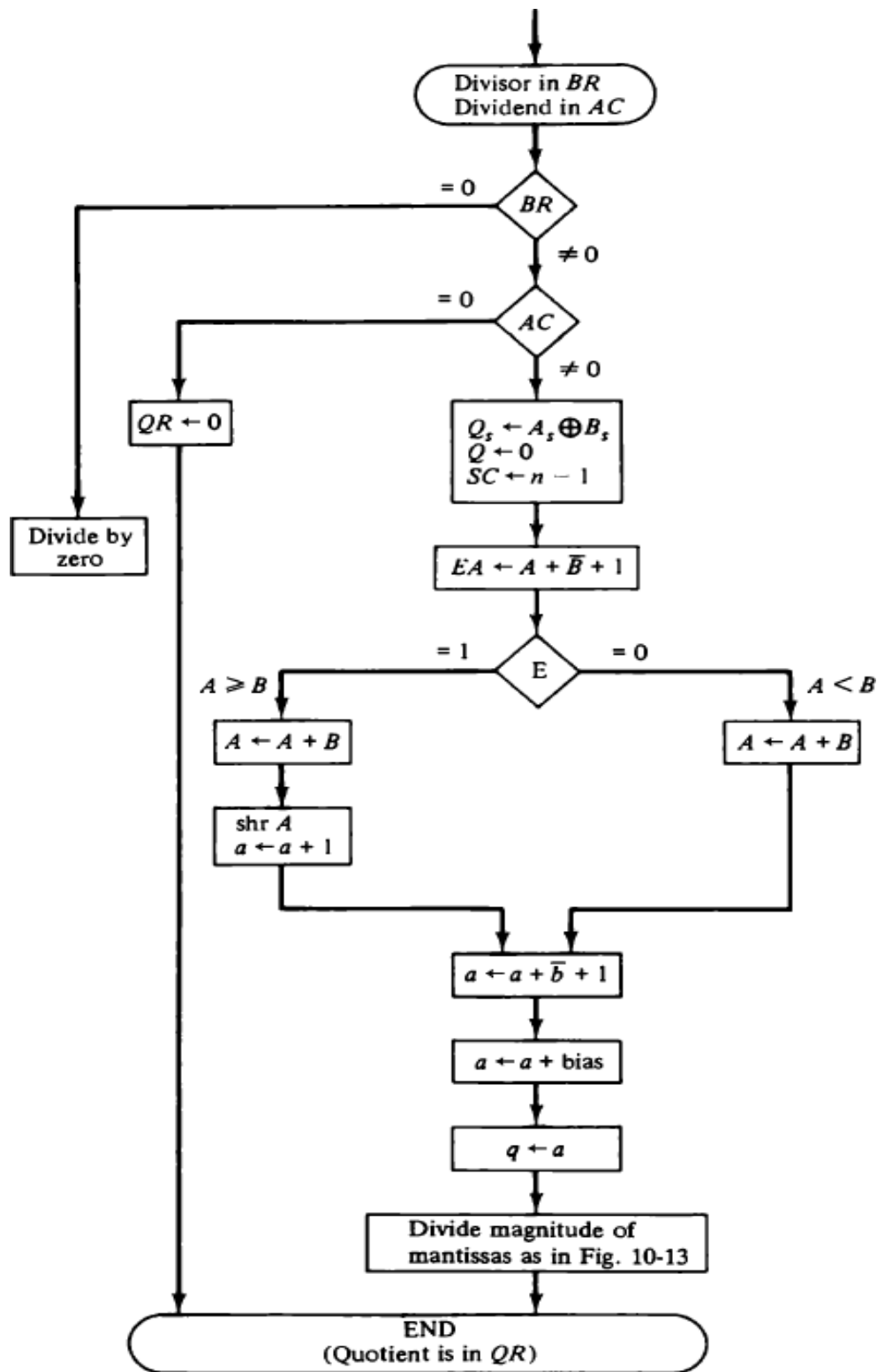


Figure 10-17 Division of floating-point numbers.

- The magnitudes of the mantissas are divided as in the fixed-point case. After the operation, the mantissa quotient resides in Q and the remainder in A. The floating-point quotient is already normalized and resides in QR. The exponent of the remainder should be the same as the exponent of the dividend. The binary point for the remainder mantissa lies (n-1) positions to the left of A1. The remainder can be converted to a normalized fraction by subtracting n-1 from the dividend exponent and by shift and decrement until the bit in A1 is equal 1.

## **Decimal Arithmetic Unit**

The user of a computer prepares data with decimal numbers and receives results in decimal form. A CPU with an arithmetic logic unit can perform arithmetic microoperations with binary data. To perform arithmetic operations with decimal data, it is necessary to convert the input decimal numbers to binary, to perform all calculations with binary numbers, and to convert the results into decimal. This may be an efficient method in applications requiring a large number of calculations and a relatively smaller amount of input and output data. When the application calls for a large amount of input-output and a relatively smaller number of arithmetic calculations, it becomes convenient to do the internal arithmetic directly with the decimal numbers. Computers capable of performing decimal arithmetic must store the decimal data in binary coded form. The decimal numbers are then applied to a decimal arithmetic unit capable of executing decimal arithmetic microoperations. Electronic calculators invariably use an internal decimal arithmetic unit since inputs and outputs are frequent. There does not seem to be a reason for converting the keyboard input numbers to binary and again converting the displayed results to decimal, since this process requires special circuits and also takes a longer time to execute. Many computers have hardware for arithmetic calculations with both binary and decimal data. Users can specify by programmed instructions whether they want the computer to perform calculations with binary or decimal data.

A decimal arithmetic unit is a digital function that performs decimal microoperations. It can add or subtract decimal numbers, usually by forming the 9's or 10's complement of the subtrahend. The unit accepts coded decimal numbers and generates results in the same adopted binary code. A single-stage decimal arithmetic unit consists of nine binary input variables and five binary output variables, since a minimum of four bits is required to represent each coded decimal digit. Each stage must have four inputs for the augend digit, four inputs for the addend digit, and an input-carry. The outputs include four terminals for the sum digit and one for the

output-carry. Of course, there is a wide variety of possible circuit configurations dependent on the code used to represent the decimal digits.

## BCD Adder

Consider the arithmetic addition of two decimal digits in BCD, together with a possible carry from a previous stage. Since each input digit does not exceed 9, the output sum cannot be greater than  $9 + 9 + 1 = 19$ , the 1 in the sum being an input-carry. Suppose that we apply two BCD digits to a 4-bit binary adder. The adder will form the sum in binary and produce a result that may range from 0 to 19. These binary numbers are listed in Table 10-4 and are labeled by symbols K, Z<sub>8</sub>, z<sub>4</sub>, Z<sub>2</sub>, and Z<sub>1</sub>. K is the carry and the subscripts under the letter Z represent the weights 8, 4, 2, and 1 that can be assigned to the four bits in the BCD code. The first column in the table lists the binary sums as they appear in the outputs of a 4-bit binary adder. The output sum of two decimal numbers must be represented in BCD and should appear in the form listed in the second column of the table. The problem is to find a simple rule by which the binary number in the first column can be converted to the correct BCD digit representation of the number in the second column. In examining the contents of the table, it is apparent that when the binary sum is equal to or less than 1001, the corresponding BCD number is identical

**TABLE 10-4** Derivation of BCD Adder

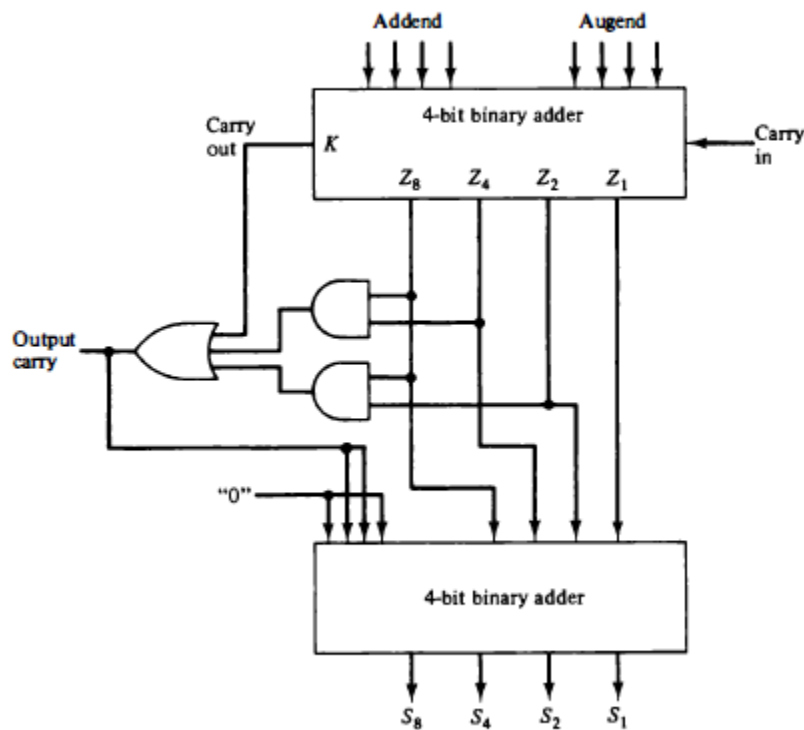
Binary Sum					BCD Sum					Decimal
K	Z <sub>8</sub>	Z <sub>4</sub>	Z <sub>2</sub>	Z <sub>1</sub>	C	S <sub>8</sub>	S <sub>4</sub>	S <sub>2</sub>	S <sub>1</sub>	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

and therefore no conversion is needed. When the binary sum is greater than 1001, we obtain a non valid BCD representation. The addition of binary 6 (0110) to the binary sum converts it to the correct BCD representation and also produces an output-carry as required. One method of adding decimal numbers in BCD would be to employ one 4-bit binary adder and perform the arithmetic operation one digit at a time. The low-order pair of BCD digits is first added to produce a binary sum. If the result is equal or greater than 1010, it is corrected by adding 0110 to the binary sum. This second operation will automatically produce an output-carry for the next pair of significant digits. The next higher-order pair of digits, together with the input-carry, is then added to produce their binary sum. If this result is equal to or greater than 1010, it is corrected by adding 0110. The procedure is repeated until all decimal digits are added. The logic circuit that detects the necessary correction can be derived from the table entries. It is obvious that a correction is needed when the binary sum has an output carry  $K = 1$ . The other six combinations from 1010 to 1 1 1 1 that need a correction have a 1 in position  $Z_8$ . To distinguish them from binary 1000 and 1001 which also have a 1 in position  $z$ , we specify further that either  $z$ , or  $z$ , must have a 1. The condition for a correction and an output-carry can be expressed by the Boolean function

$$C = K + Z_8 Z_4 + z, z,$$

When  $C = 1$ , it is necessary to add 0110 to the binary sum and provide an output-carry for the next stage. A BCD adder is a circuit that adds two BCD digits in parallel and produces a sum digit also in BCD. A BCD adder must include the correction logic in its internal construction. To add 0110 to the binary sum, we use a second 4-bit binary adder as shown in Fig. 10-18. The two decimal digits, together with the input-carry, are first added in the top 4-bit binary adder to produce the binary sum. When the output-carry is equal to 0, nothing is added to the binary sum. When it is equal to 1, binary 0110 is added to the binary sum through the bottom 4-bit binary adder. The output-carry generated from the bottom binary adder may be ignored, since it supplies information already available in the output-carry terminal. A decimal parallel-adder that adds  $n$  decimal digits needs  $n$  BCD adder stages with the output-carry from one stage connected to the input-carry of the next-higher-order stage. To achieve shorter propagation delays, BCD adders include the necessary circuits for carry look-ahead. Furthermore, the adder circuit for the correction does not need all four full-adders, and this circuit can be optimized.

Figure 10-18 Block diagram of BCD adder.



## BCD Subtraction

A straight subtraction of two decimal numbers will require a subtractor circuit that will be somewhat different from a BCD adder. It is more economical to perform the subtraction by taking the 9's or 10's complement of the subtrahend and adding it to the minuend. Since the BCD is not a self-complementing code, the 9's complement cannot be obtained by complementing each bit in the code. It must be formed by a circuit that subtracts each BCD digit from 9.

The 9's complement of a decimal digit represented in BCD may be obtained by complementing the bits in the coded representation of the digit provided a correction is included. There are two possible correction methods. In the first method, binary 1010 (decimal 10) is added to each complemented digit and the carry discarded after each addition. In the second method, binary 0110 (decimal 6) is added before the digit is complemented. As a numerical illustration, the 9's complement of BCD 0111 (decimal 7) is computed by first complementing each bit to obtain 1000. Adding binary 1010 and discarding the carry, we obtain 0010 (decimal 2). By the second method, we add 0110 to 0111 to obtain 1101. Complementing each bit, we obtain the required result of 0010. Complementing each bit of a 4-bit binary number  $N$  is

identical to the subtraction of the number from 1111 (decimal 15). Adding the binary equivalent of decimal 10 gives  $15 - N + 10 = 9 - N + 16$ . But 16 signifies the carry that is discarded, so the result is  $9 - N$  as required. Adding the binary equivalent of decimal 6 and then complementing gives  $15 - (N + 6) = 9 - N$  as required.

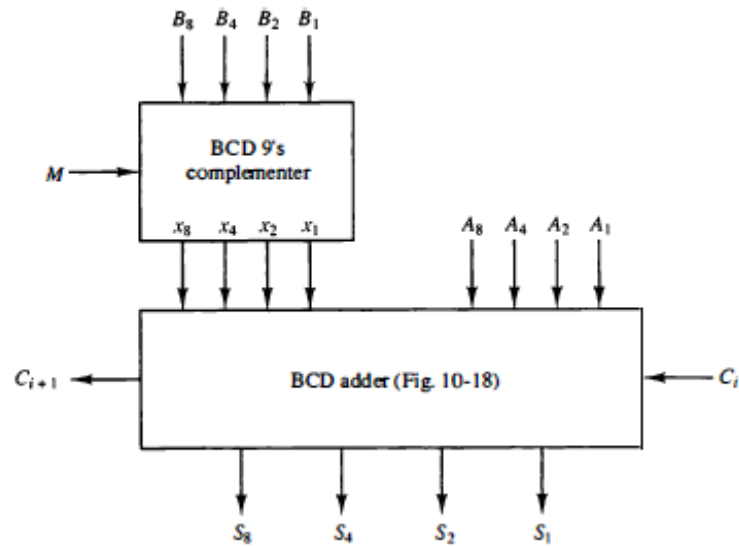
The 9's complement of a BCD digit can also be obtained through a combinational circuit. When this circuit is attached to a BCD adder, the result is a BCD adder/subtractor. Let the subtrahend (or addend) digit be denoted by the four binary variables  $B_8, B_4, B_2$ , and  $B_1$ . Let  $M$  be a mode bit that controls the add/subtract operation. When  $M = 0$ , the two digits are added; when  $M = 1$ , the digits are subtracted. Let the binary variables  $x_8, x_4, x_2$ , and  $x_1$  be the outputs of the 9's complementer circuit. By an examination of the truth table for the circuit, it may be observed (see Prob. 10-30) that  $B_1$  should always be complemented;  $B_2$  is always the same in the 9's complement as in the original digit;  $x_4$  is 1 when the exclusive-OR of  $B_2$  and  $B_4$  is 1; and  $x_8$  is 1 when  $B_8 B_4 B_2 = 000$ . The Boolean functions for the 9's complementer circuit are

$$\begin{aligned} x_1 &= B_1 M' + B_1' M & x_4 &= B_4 M' + (B_4' B_2 + B_4 B_2') M \\ x_2 &= B_2 & x_8 &= B_8 M' + B_8' B_4' B_2' M \end{aligned}$$

From these equations we see that  $x = B$  when  $M = 0$ . When  $M = 1$ , the  $x$  outputs produce the 9's complement of  $B$ . One stage of a decimal arithmetic unit that can add or subtract two BCD digits is shown in Fig. 10-19. It consists of a BCD adder and a 9's complementer. The mode  $M$  controls the operation of the unit. With  $M = 0$ , the  $S$  outputs form the sum of  $A$  and  $B$ . With  $M = 1$ , the  $S$  outputs form the sum of  $A$  plus the 9's complement of  $B$ . For numbers with  $n$  decimal digits we need  $n$  such stages. The output carry  $C_{n+1}$  from one stage must be connected to the input carry  $C_n$  of the next-higher-order stage. The best way to subtract the two decimal numbers is to let  $M = 1$  and apply a 1 to the input carry  $C_1$  of the first stage. The outputs will form the sum of  $A$  plus the 10's complement of  $B$ , which is equivalent to a subtraction operation if the carry-out of the last stage is discarded.



Figure 10-19 One stage of a decimal arithmetic unit.



## Decimal Arithmetic Operations

The algorithms for arithmetic operations with decimal data are similar to the algorithms for the corresponding operations with binary data. In fact, except for a slight modification in the multiplication and division algorithms, the same flowcharts can be used for both types of data provided that we interpret the microoperation symbols properly. Decimal numbers in BCD are stored in computer registers in groups of four bits. Each 4-bit group represents a decimal digit and must be taken as a unit when performing decimal microoperations. For convenience, we will use the same symbols for binary and decimal arithmetic microoperations but give them a different interpretation. As shown in Table 10-5, a bar over the register letter symbol denotes the 9's complement of the decimal number stored in the register. Adding 1 to the 9's complement produces the 10's complement. Thus, for decimal numbers, the symbol  $A \leftarrow A + \bar{B} + 1$  denotes a transfer of the decimal sum formed by adding the original content A to the 10's complement of B. The use of identical symbols for the 9's complement and the 1's complement may be confusing if both types of data are employed in the same system. If this is the case, it may be better to adopt a different symbol for the 9's complement. If only one type of data is being considered, the symbol would apply to the type of data used. Incrementing or decrementing a register is the same for binary and decimal except for the number of states that the register is

allowed to have. A binary counter goes through 16 states, from 0000 to 1111, when incremented. A decimal counter goes through 16 states from 0000 to 1111 and back to 0000, since 9 is the last count. Similarly, a binary counter sequences from 1111 to 0000 when decremented. A decimal counter goes from 1001 to 0000. A decimal shift right or left is preceded by the letter d to indicate a shift over the four bits that hold the decimal digits. As a numerical illustration consider a register A holding decimal 7860 in BCD. The bit pattern of the 12 flip-flops is

0111 1000 0110 0000

The microoperation dshr A shifts the decimal number one digit to the right to give 0786. This shift is over the four bits and changes the content of the register into

0000 0111 1000 0110

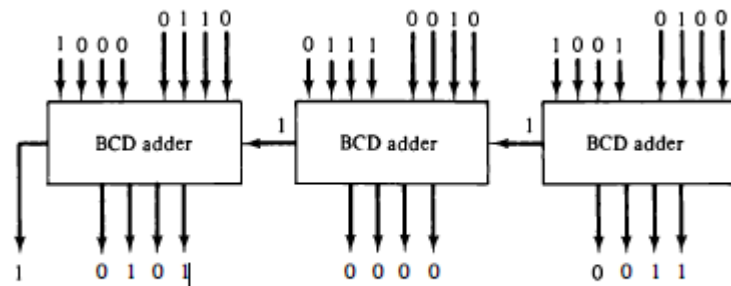
**TABLE 10-5** Decimal Arithmetic Microoperation Symbols

Symbolic Designation	Description
$A \leftarrow A + B$	Add decimal numbers and transfer sum into A
$\bar{B}$	9's complement of B
$A \leftarrow A + \bar{B} + 1$	Content of A plus 10's complement of B into A
$Q_L \leftarrow Q_L + 1$	Increment BCD number in $Q_L$
dshr A	Decimal shift-right register A
dshl A	Decimal shift-left register A

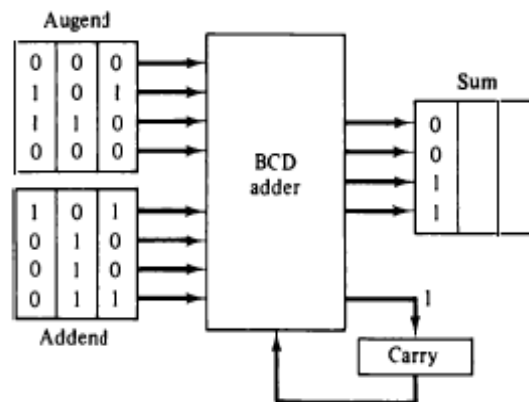
## Addition and Subtraction

The algorithm for addition and subtraction of binary signed-magnitude numbers applies also to decimal signed-magnitude numbers provided that we interpret the microoperation symbols in the proper manner. Similarly, the algorithm for binary signed-2's complement numbers applies to decimal signed-10's complement numbers. The binary data must employ a binary adder and a complementer. The decimal data must employ a decimal arithmetic unit capable of adding two BCD numbers and forming the 9's complement of the subtrahend as shown in Fig. 10-19. Decimal data can be added in three different ways, as shown in Fig. 10-20. The parallel method uses a decimal arithmetic unit composed of as many BCD adders as there are digits in the number. The sum is formed in parallel and requires only one microoperation. In the digit-serial bit-parallel method, the digits are applied to a single BCD adder serially, while the bits of each coded digit are transferred in parallel. The sum is formed by shifting the decimal

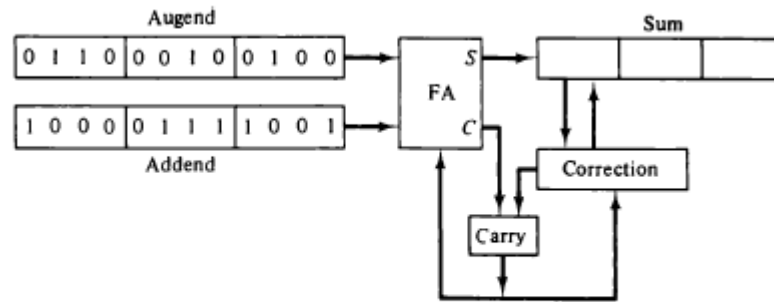
numbers through the BCD adder one at a time. For  $k$  decimal digits, this configuration requires  $k$  microoperations, one for each decimal shift. In the all serial adder, the bits are shifted one at a time through a full-adder. The binary sum formed after four shifts must be corrected into a valid BCD digit. This correction, discussed in Sec. 10-6, consists of checking the binary sum. If it is greater than or equal to 1010, the binary sum is corrected by adding to it 0110 and generating a carry for the next pair of digits. The parallel method is fast but requires a large number of adders. The digit-serial bit-parallel method requires only one BCD adder, which is shared by all the digits. It is slower than the parallel method because of the time required to shift the digits. The all serial method requires a minimum amount of equipment but is very slow.



(a) Parallel decimal addition:  $624 + 879 = 1503$



(b) Digit-serial, bit-parallel decimal addition



(c) All serial decimal addition

Figure 10-20 Three ways of adding decimal numbers.

## Multiplication

The multiplication of fixed-point decimal numbers is similar to binary except for the way the partial products are formed. A decimal multiplier has digits that range in value from 0 to 9, whereas a binary multiplier has only 0 and 1 digits. In the binary case, the multiplicand is added to the partial product if the multiplier bit is 1. In the decimal case, the multiplicand must be multiplied by the digit multiplier and the result added to the partial product. This operation can be accomplished by adding the multiplicand to the partial product a number of times equal to the value of the multiplier digit. The registers organization for the decimal multiplication is shown in Fig. 10-21 . We are assuming here four-digit numbers, with each digit occupying four bits, for a total of 16 bits for each number. There are three registers, A, B, and Q, each having a corresponding sign flip-flop A, B, and Q, . The decimal multiplication algorithm is shown in Fig. 10-22. Initially, the entire A register and B, are cleared and the sequence counter SC is set to a number k equal to the number of digits in the multiplier. The low-order digit of the multiplier in Q, is checked. If it is not equal to 0, the multiplicand in B is added to the partial product in A once and QL is decremented. QL is checked again and the process is repeated until it is equal to 0. In this way, the multiplicand in B is added to the partial product a number of times equal to the multiplier digit. Any temporary overflow digit will reside in A, and can range in value from 0 to 9. Next, the partial product and the multiplier are shifted once to the right. This places zero in A, and transfers the next multiplier quotient into QL. The process is then repeated k times to form a double-length product in AQ .

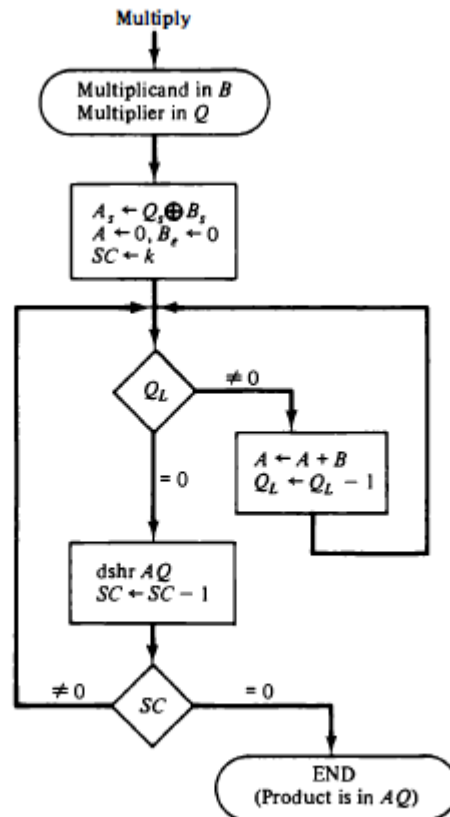


Figure 10-22 Flowchart for decimal multiplication.

## Division

Decimal division is similar to binary division except of course that the quotient digits may have any of the 10 values from 0 to 9. In the restoring division method, the divisor is subtracted from the dividend or partial remainder as many times as necessary until a negative remainder results. The correct remainder is then restored by adding the divisor. The digit in the quotient reflects the number of subtractions up to but excluding the one that caused the negative difference.

The decimal division algorithm is shown in Fig. 10-23. It is similar to the algorithm with binary data except for the way the quotient bits are formed. The dividend (or partial remainder) is shifted to the left, with its most significant digit placed in A, . The divisor is then subtracted by adding its 10's complement value. Since B, is initially cleared, its complement value is 9 as required. The carry in E determines the relative magnitude of A and B. If E = 0, it signifies that A < B. In this case the divisor is added to restore the partial remainder and QL stays at 0 (inserted there during the shift). If E = 1, it signifies that A ≥ B. The quotient digit in QL is

incremented once and the divisor subtracted again. This process is repeated until the subtraction results in a negative difference which is recognized by E being 0. When this occurs, the quotient digit is not incremented but the divisor is added to restore the positive remainder. In this way, the quotient digit is made equal to the number of times that the partial remainder "goes" into the divisor. The partial remainder and the quotient bits are shifted once to the left and the process is repeated k times to form k quotient digits. The remainder is then found in register A and the quotient is in register Q. The value of E is neglected.

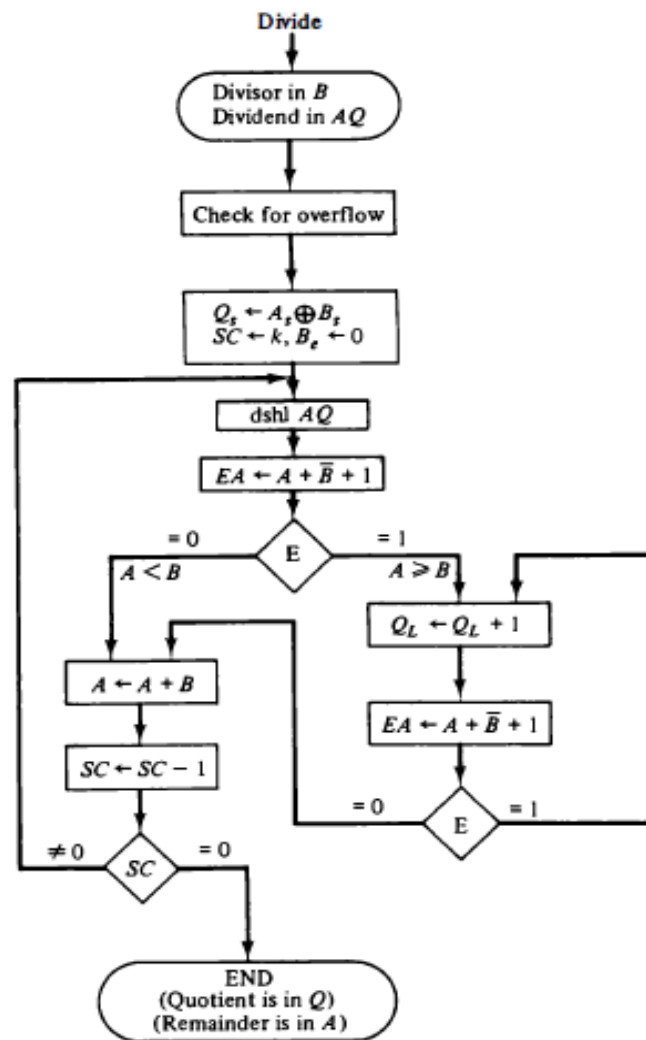


Figure 10-23 Flowchart for decimal division.



## UNIT IV

### INPUT-OUTPUT ORGANIZATION:

**Input-Output Interface, Asynchronous data transfer, Modes of Transfer, Priority Interrupt, Direct memory Access**

### PERIPHERAL DEVICES:

Input and output devices attached to the computer are also called **peripherals**.

**Ex:** keyboards, display units, and printers.

Peripherals that provide auxiliary storage for the system are **magnetic disks** and **tapes**. Peripherals are electromechanical and electromagnetic devices of some complexity.

**Monitor and keyboard:** There are different types of video monitors, but the most popular use a cathode ray tube (CRT). The CRT contains an electronic gun that sends an electronic beam to a phosphorescent screen in front of the tube. The beam can be deflected horizontally and vertically.

A characteristic feature of display devices is a cursor that marks the position in the screen where the next character will be inserted. The cursor can be moved to any position in the screen, to a single character, the beginning of a word, or to any line. Edit keys add or delete information based on the cursor position. The display terminal can operate in a single-character mode where the computer simultaneously.

**Printer:** Printer provides permanent record on paper of computer output data or text.

There are **three** basic types of character printers:

1. Daisy wheel
2. dot matrix, and
3. laser printers.

**Daisy wheel:** The daisywheel printer contains a wheel with the characters placed along the circumference. To print a character, the wheel rotates to the proper position and an energized magnet then presses the letter against the ribbon.



**dot matrix:** The dot matrix printer contains a set of dots along the printing mechanism.

**Example:** A 5\*7 dot matrix printer that prints 80 character per line has seven horizontal lines, each consisting of  $5*80=400$  dots.

**laser printers:** The laser printer uses a rotating photographic drum that is used to imprint the character images.

**Magnetic tape:** Magnetic tapes are used mostly for storing files of data. It is one of the cheapest and slowest methods for storage and has the advantage that tapes can be removed when not in use.

**Magnetic disks:** Magnetic disks have achieved by moving a read-write mechanism to attract in the magnetized surface. Disks are used mostly for bulk storage of programs and data.

## **INPUT – OUTPUT INTERFACE:**

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

The major differences are:

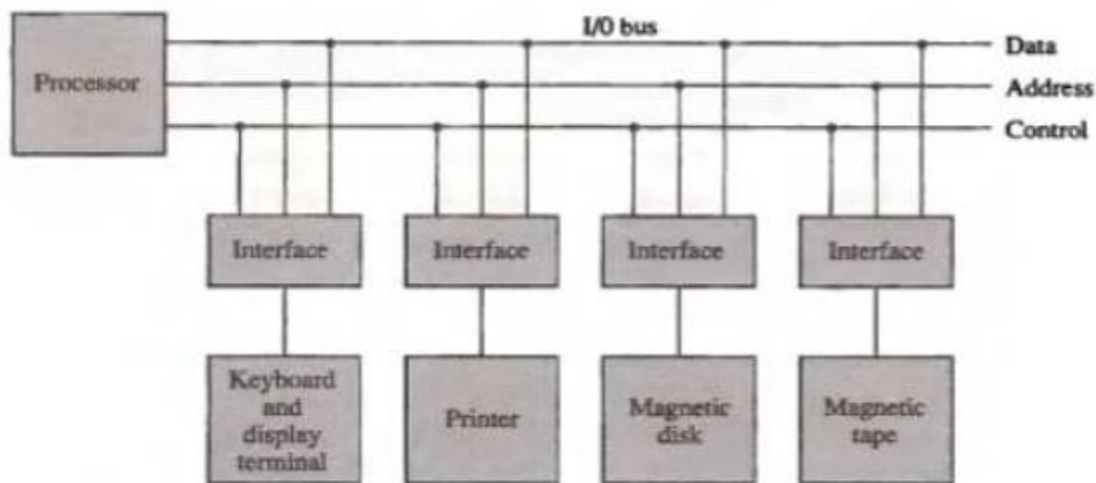
1. Peripherals are electromechanical and electromagnetic devices and their manner of operation is different from the operation of the CPU and memory, which are electronic devices. Therefore, a conversion of a signal values may be required.
2. The **data transfer rate** of peripherals is usually slower than the transfer rate of the CPU, and consequently, a synchronization mechanism may be needed.
3. **Data codes and formats** in peripherals differ from the word format in the CPU and memory.
4. The **operating modes** of peripherals are different from each other and each must be controlled so as not to disturb the operation of other peripherals connected to the CPU.

To resolve these differences, computer systems include special hardware components between the CPU and peripherals to supervise and synchronize all input and output transfers. These components are called **interface units**.

### **I/O Bus and Interface Modules:**

- The I/O bus consists of data lines, address lines, and control lines.
- The magnetic disk, printer, and terminal are employed in practically any general-purpose computer. The magnetic tape is used in some computers for backup storage.
- Each peripheral device has associated with it an interface unit.
- Each interface decodes the address and control received from the I/O bus, interprets them for the peripheral and processor.
- Each peripheral has its own controller that operates the particular electromechanical device.

**Figure 11-1** Connection of I/O bus to input-output devices.



- The I/O bus from the processor is attached to all peripheral interfaces. To communicate with a particular device, the processor places a device address on bus, an address decoder that monitors the address lines.

- When the interface detects its own address, it activates the path between the bus lines and the device that it controls. All peripherals whose address does not correspond to the address in the bus are disabled by their interface.

**I/O Commands:** There are **four** types of commands that an interface may receive. They are classified as control, status, data output, and data input.

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

**Example:** A magnetic tape unit may be instructed to backspace the tape by one record, to rewind the tape, or to start the tape moving in the forward direction.

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

**Example:** The computer may wish to check the status of the peripheral before a transfer is initiated.

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

4. **Input data:** The data input command is the opposite of the data output. In this case the interface receives an item of data from the peripheral and places it in its buffer register.

**I/O versus Memory Bus:** There are three ways that computer buses can be used to communicate with memory and I/O:

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

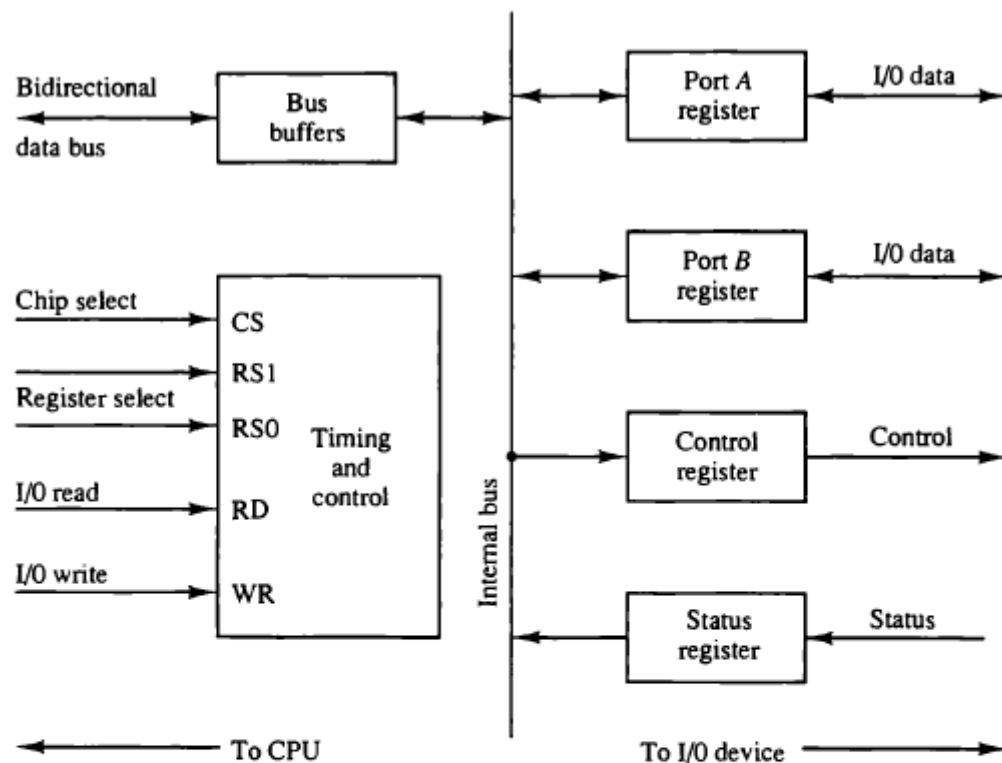
**Example of I/O Interface:**

- It consists of two data registers called **ports**, a control register, a status register, bus buffers, and timing and control circuits.

- The interface communicates with the CPU through the data bus. The chip select and register select inputs determine the address assigned to the interface.
- I/O read and write are two control lines that specify an input or output, respectively.
- The four registers communicate directly with the I/O device attached to the interface.
- The I/O data to and from the device can be transferred into either port A or port B.
- The interface may operate with an output device or with an input device, or with a device that requires both input and output.
- A magnetic disk unit transfers data in both directions but not at the same time, so the interface can use bidirectional lines.
- Control is sent to the control register, status information is received from the status register, and data are transferred to and from ports is always via the common data bus.
- The **control register** receives control information from the CPU. By loading appropriate bits into the control register, the interface and the I/O device attached to it can be placed in a variety of operating modes.
- Port A may be defined as an input port and port B as an output port.
- The bits in the status register are used for status conditions and for recording errors that may occur during the data transfer.

**Example:**

- A status bit may indicate that port A has received a new data item from the I/O device.
- Another bit in the status register may indicate that a parity error has occurred during the transfer.
- The circuit enables the chip select (CS) input when the interface is selected by the address bus. The two register select inputs **RS1** and **RS0** are usually connected to the two least significant lines of the address bus.



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

**Figure 11-2** Example of I/O interface unit.

- These two inputs select one of the four registers in the interface.
- The content of the selected register is transfer into the CPU via the data bus when the I/O read signal is enabled.
- The CPU transfers binary information into the selected register via the data bus when the I/O write input is enabled.

**I/O mapping:** there are two types

**Memory-mapped I/O:**

- Single address space for both memory and I/O devices  
**disadvantage** – uses up valuable memory address space
- I/O module registers treated as memory addresses
- Same machine instructions used to access both memory and I/O devices  
**advantage** – allows for more efficient programming
- Single read line and single write lines needed
- Commonly used

**Isolated I/O:**

- Separate address space for both memory and I/O devices
- Separate memory and I/O select lines needed
- Small number of I/O instructions
- Commonly used

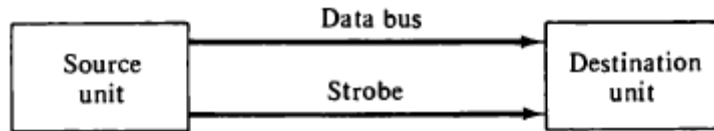
**ASYNCHRONOUS DATA TRANSFER:**

- If the registers in the interface share a common clock with the CPU registers, the transfer between the two units is said to be **synchronous**.
- In most cases, the internal timing in each unit is independent from the other in that each uses its own private clock for internal registers. It is called **asynchronous**.
- Asynchronous data transfer between two independent units requires that control signals be transmitted between the communicating units.

There are **two** methods are used to supply control signals.

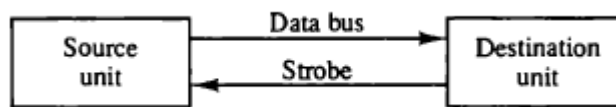
1. Strobe control
2. Handshaking

**Strobe Control:** A strobe pulse supplied by one of the units to indicate to the other unit when the transfer has to occur. The strobe may be activated by either the source or the destination unit. Figure (a) shows a **source-initiated transfer**. The data bus carries the binary information from source unit to the destination unit. The strobe is single line that informs the destination unit when a valid data word is available in the bus.



(a) Block diagram

The following figure shows a **data transfer initiated by the destination unit**. In this case the destination unit activates the strobe pulse, informing the source to provide the data. The source unit responds by placing the requested binary information on the data bus. The data must be valid and remain in the bus long enough for the destination unit to accept it.



**Disadvantage:** The source unit that initiates the transfer has no way of knowing whether the destination unit has actually received the data item that was placed in the bus.

**Handshaking:** The handshake method solves this problem (disadvantage of strobe control) by introducing a second control signal that provides a reply to the unit that initiates the transfer.

The following figure shows the **data transfer procedure when initiated by the source**.

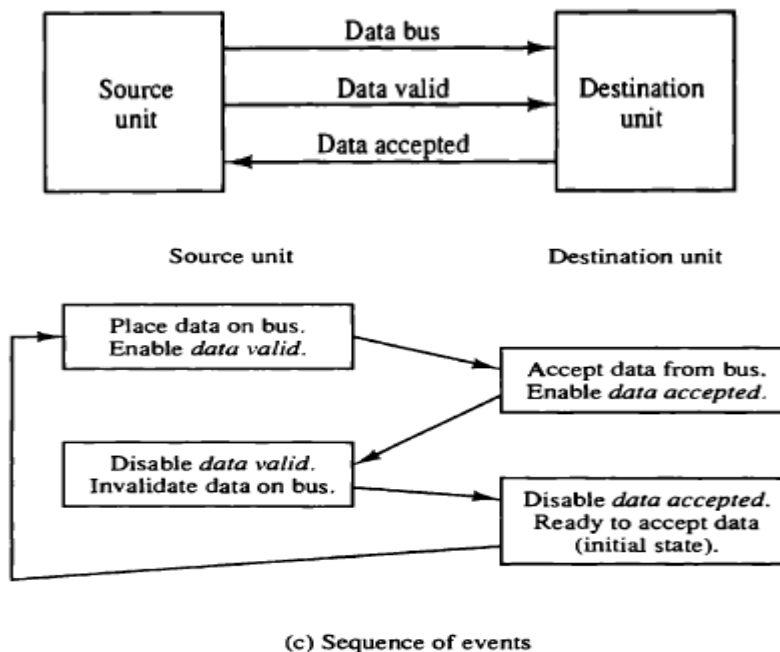
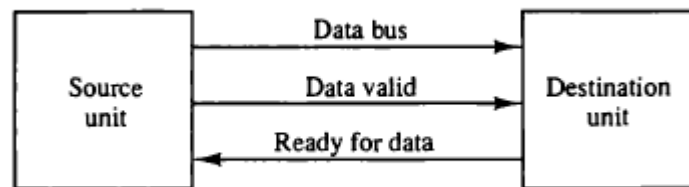


Figure 11-5 Source-initiated transfer using handshaking.

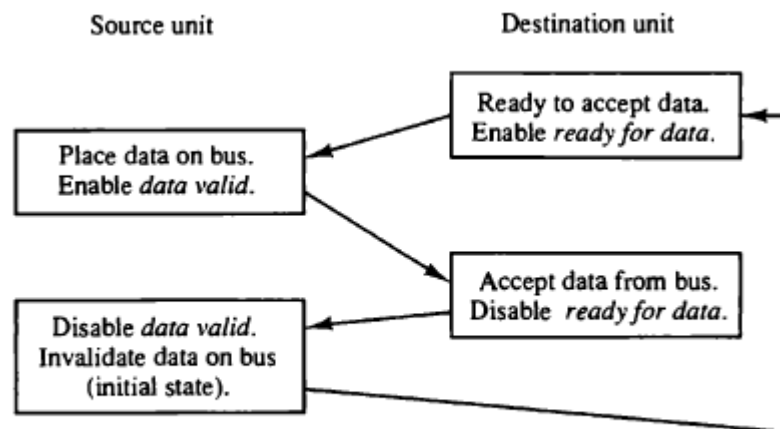
The two handshaking lines are data valid, which is generated by the source unit, and data accepted, generated by the destination unit. The source unit initiates the transfer by placing the data on the bus and enabling its data valid signal. The data accepted signal is activated by the destination unit after it accepts the data from the bus. The source unit then disables its data valid signal, which invalidates the data on the bus. The destination unit then disables its data accepted signal and the system goes into its initial state. The source does not send the next data item until after the destination unit shows its readiness to accept new data by disabling its data accepted signal.

The source unit in this case does not place data on the bus until after it receives the ready for data signal from the destination unit. From there on, the handshaking procedure follows the same pattern as in the source-initiated case. (see the below figure)

**Figure 11-6** Destination-initiated transfer using handshaking.



(a) Block diagram



(c) Sequence of events

**Timeout:** if one unit is faulty, the data transfer will not be completed. Such an error can be detected by means of a timeout mechanism, which produces an alarm if the data transfer is not completed within a predetermined time. The timeout is implemented by means of an internal clock that starts counting time when the unit enables one of its handshaking control signals.

#### ASYNCHRONOUS SERIAL TRANSFER:



The transfer of data between two units may be done in parallel or serial.

**Parallel data transmission:** In parallel data transmission, each bit of the message has its own path and the total message is transmitted at the same time. This means that an n-bit message must be transmitted through n separate conductor paths.

**Advantage:** it is faster and it is used for short distances and where speed is important.

**Disadvantage:** it requires many wires.

**Serial data transmission:** In serial data transmission, each bit in the message is sent in sequence one at a time. This method requires the use of one pair of conductors or one conductor and a common ground.

**Advantage:** it is less expensive since it requires only one pair of conductors. It is used for long distances.

**Disadvantage:** Serial transmission is slower.

Serial transmission can be **synchronous** or **asynchronous**.

**Synchronous:** In synchronous transmission, the two units share a common clock frequency and bits are transmitted continuously at the rate dictated by the clock pulses.

**Asynchronous:** In asynchronous transmission, binary information is sent only when it is available and the line remains idle when there is no information to be transmitted.

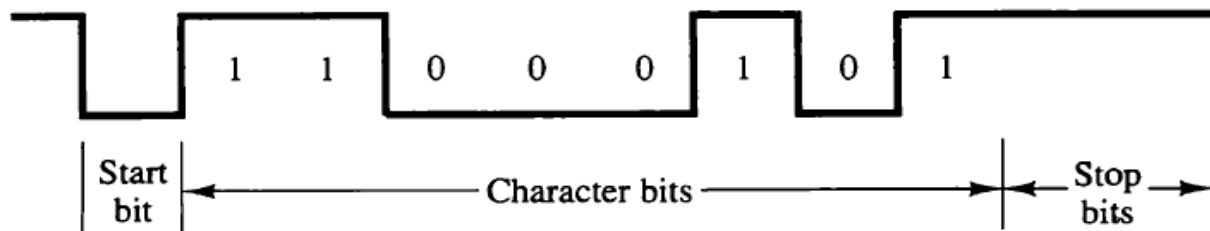
A serial asynchronous data transmission technique used in many interactive terminals employs special bits that are inserted at both ends of the character code. With this technique, each character consists of three parts: a **start bit**, the **character bits** and **stop bits**.

**Start bit:** start bit is always a 0 and is used to indicate the beginning of a character.

**Stop bit:** The stop bit is always a 1 and is used to indicate the beginning of a character.

An example of this format is shown in fig.11-7.

**Figure 11-7 Asynchronous serial transmission.**



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

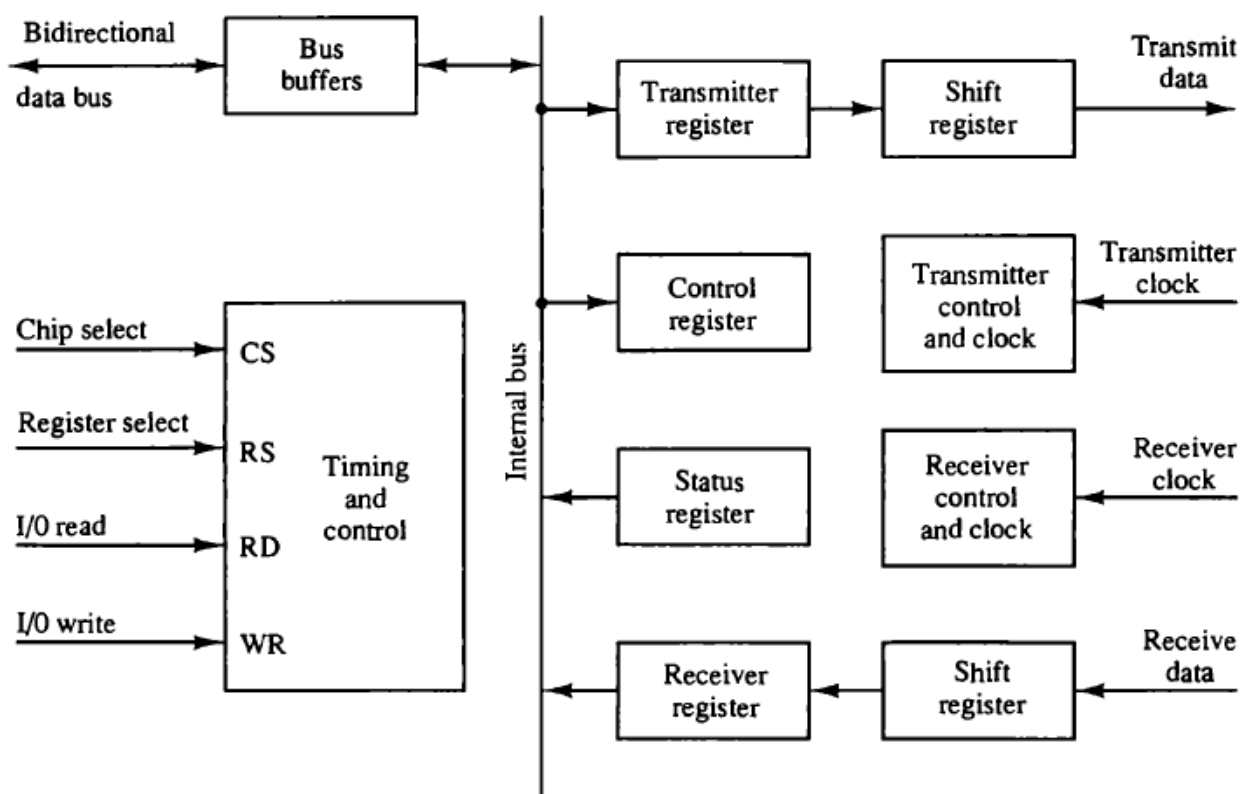
1. When a character is not being sent, the line is kept in the 1-state.
2. The initiation of a character transmission is detected from the start bit, which is always 0.
3. The character bits always follow the start bit.
4. After the last bit of the character is transmitted, a stop bit is detected when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

Using these rules, the receiver can detect the start bit when the line goes from 1 to 0.

### **Asynchronous Communication Interface:**

- The interface is initialized for a particular mode of transfer by means of a control byte that is loaded into its control register.
- The transmitter register accepts a data byte from the CPU through data bus. This byte is transferred to a shift register for serial transmission.
- The receiver portion receives serial information into another shift register and when a complete data byte is accumulated, it is transferred to the receiver register.
- The CPU can select the receiver register to read the byte through the data bus.
- The CPU can read the status register to check the status of the flag bits and to determine if any errors have occurred.
- The chip select and the read & write control lines communicate with the CPU.

- The chip select (CS) input is used to select the interface through the address bus.
- The register select (RS) is associated with the read (RD) and write (WR) controls.
- Two registers are write-only and two are read-only. The register selected is a function of the RS value and the RD and WR status, as listed in the table accompanying the diagram.



CS	RS	Operation	Register selected
0	x	x	None: data bus in high-impedance
1	0	WR	Transmitter register
1	1	WR	Control register
1	0	RD	Receiver register
1	1	RD	Status register

- Two bits in the status register are used as flags. One bit is used to indicate whether the transmitter register is empty and another bit is used to indicate whether the receiver register is full.

- The CPU reads the status register and checks the flag to see if the transmitter register is empty. If it is empty, the CPU transfers a character to the transmitter register and the interface clears the flag to mark the register full. The first bit in the transmitter shift register is set to 0 to generate a start bit.
- The flag in the status register is set to indicate that the receiver register is full. The CPU reads the status register and checks the flag, and if set, reads the data from the receiver register.
- The CPU can read the status register at any time to check if any errors have occurred. There are three possible errors.

1. Parity error
2. Framing error
3. Overrun error

1. **Parity error:** Parity error occurs if the number of 1's in the received data is not correct parity.

2. **Framing error:** Framing error occurs if the right number of stop bits is not detected at end of the received character.

3. **Overrun error:** An Overrun error occurs if the CPU does not read the character from the receiver register before the next one becomes available in the shift register.

### **First-In, First-Out Buffer: (FIFO buffer)**

**Definition:** A first-in, first-out (FIFO) buffer is a memory unit that stores information in such a manner that the item first in is the item first out. A FIFO buffer comes with separate input and output terminals.

- The important feature of this buffer is that it can input data and output data at two different rates and the output data are always in the same order in which the data entered the buffer.
- FIFO buffer can be useful in some or all the applications when data are transferred asynchronously.
- It consists of four 4-bit registers **RI**, **I=1,2,3,4**, and a control register with flip-flops **F<sub>i</sub>**, **i=1,2,3,4**, one for each register.

- The FIFO can store four words of four bits each. The number of bits per word can be increased by increasing the number of bits in each register and the number of words can be increased by increasing the number of registers.
- A flip-flop  $F_i$  in the control register that  $I$  set to 1 indicates that a 4-bit data word is stored in the corresponding register  $R_i$ . A 0 in  $F_i$  indicates that the corresponding register does not contain valid data.
- Whenever the  $F_i$  bit of the control register is set ( $F_i=1$ ) and the  $F_{i+1}$  bit is reset ( $F_{i+1}^1 = 1$ ), a clock is generated causing register  $R_{(I+1)}$  to accept the data from register  $R_i$ .
- Data are inserted into the buffer provided that the *input ready* signal is enabled. This occurs when the first control flip-flop  $F_1$  is reset, indicating that register  $R_1$  is empty.
- Data are loaded from the input lines by enabling the clock in  $R_1$  through the *insert* control line. The same clock sets  $F_1$ , which disables the *input ready* control, indicating that the FIFO is now busy and unable to accept more data.
- The ripple-through process begins provided that  $R_2$  is empty. The data in  $R_1$  are transferred into  $R_2$  and  $F_1$  is cleared. This enables the *input ready* line, indicating that the inputs are now available for another data word.

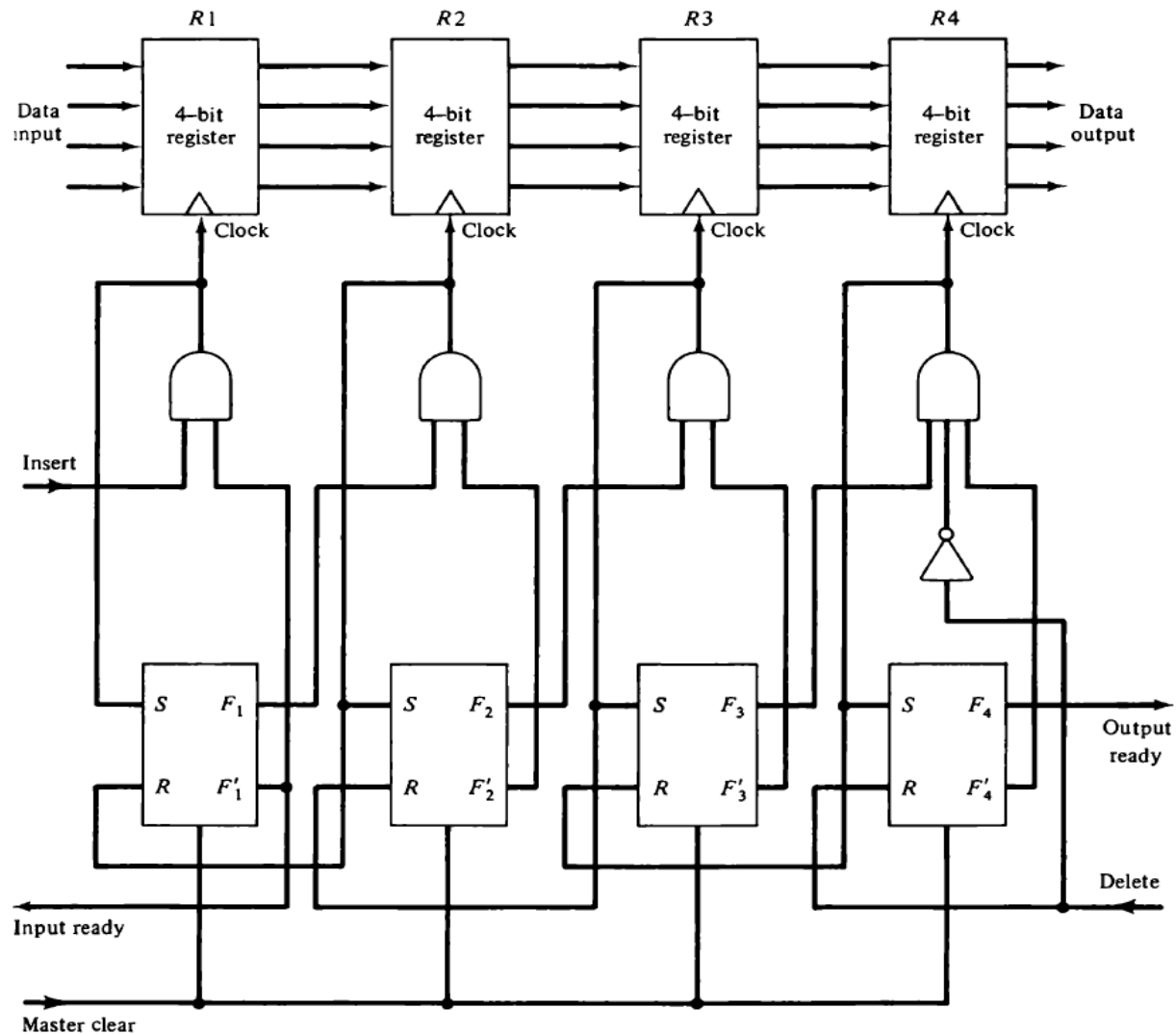


Figure 11-9 Circuit diagram of  $4 \times 4$  FIFO buffer.

- If the FIFO is full,  $F_1$  remains set and the *input ready* line stays in the 0 state. Note that the two control lines *input ready* and *insert* constitute a destination-initiated pair of handshake lines.
- The *output ready* control line is enabled when the last control flip-flop  $F_4$  is set, indicating that there are valid data in the output register  $R_4$ .
- The output data from  $R_4$  are accepted by a destination unit, which then enables the *delete* control signal. This resets  $F_4$ , causing *output ready* to disable, indicating that the data on the out are no longer valid. Only after the *delete* signal goes back to 0 can the data from  $R_3$  and  $F_4$  will remain in the reset state.

## MODES OF TRANSFER:

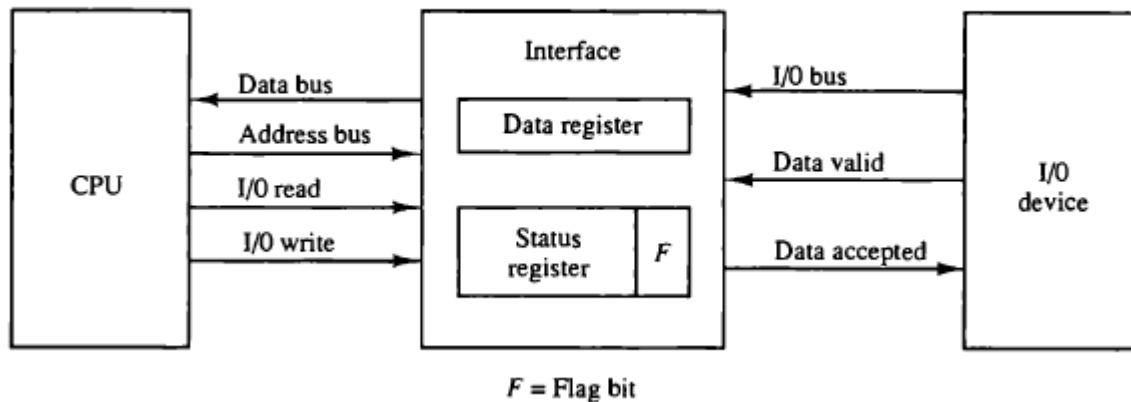
Data transfer to and from peripherals may be handled in one of three possible modes:

1. Programmed I/O
2. Interrupt-initiated I/O
3. Direct memory access (DMA)

### Programmed I/O:

The programmed I/O method is particularly useful in small low-speed computers. An example of data transfer from an I/O device through an interface into the CPU is shown in fig.

**Figure 11-10** Data transfer from I/O device to CPU.



The device transfers bytes of data one at a time as they are available. When a byte of data is available, the device places it in the I/O bus and enables its data valid line.

The interface accepts the byte into data register and enables the data accepted line. The interface sets a bit (*F* or “flag” bit) in the status register.

The device can now disable the data valid line, but it will not transfer another byte until the data accepted line is disabled by the interface.

A program is written for the computer to check the flag in the status register to determine if a byte has been placed in the data register by the I/O device. This is done by reading the status register into a CPU register and checking the value of the flag bit.

If the flag is equal to 1, the CPU reads the data from the data register. The flag bit is then cleared to 0 by either the CPU or the interface. Once the flag is cleared, the interface disables the data accepted line and the device can then transfer the next data byte.

The transfer of each byte requires **three** instructions:

1. Read the status register.
2. Check the status of the flag bit and branch to step 1 if not set or to step 3 if set.
3. Read the data register.

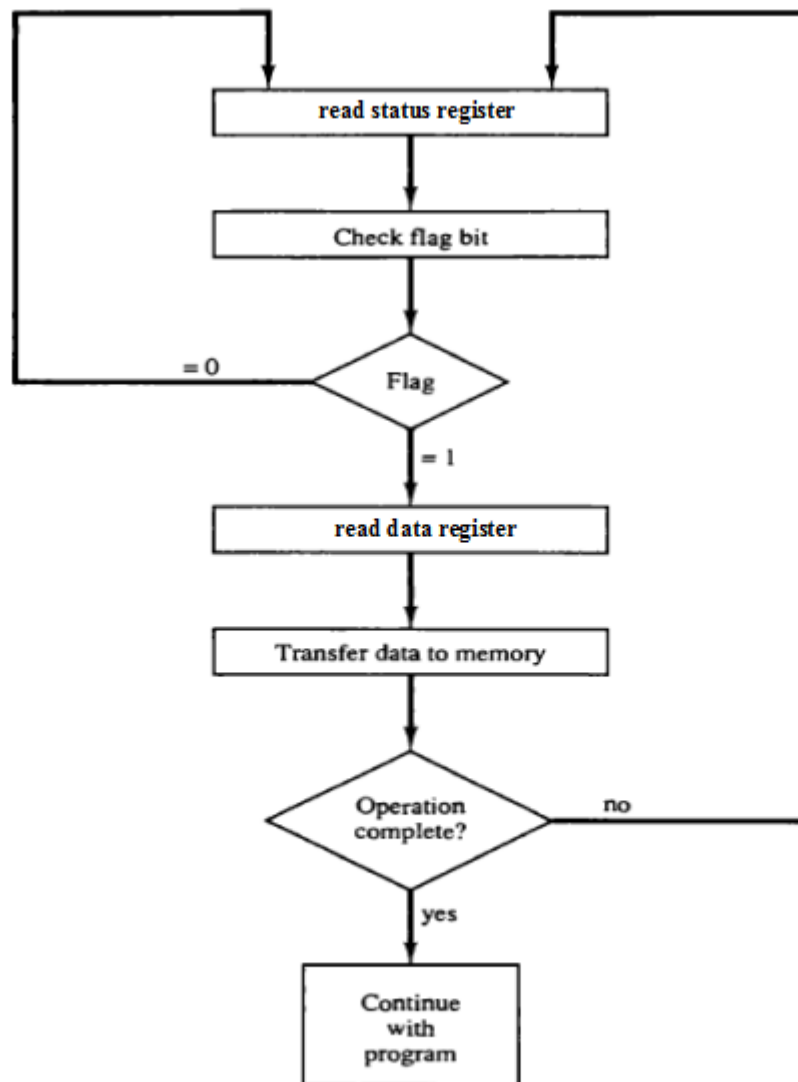


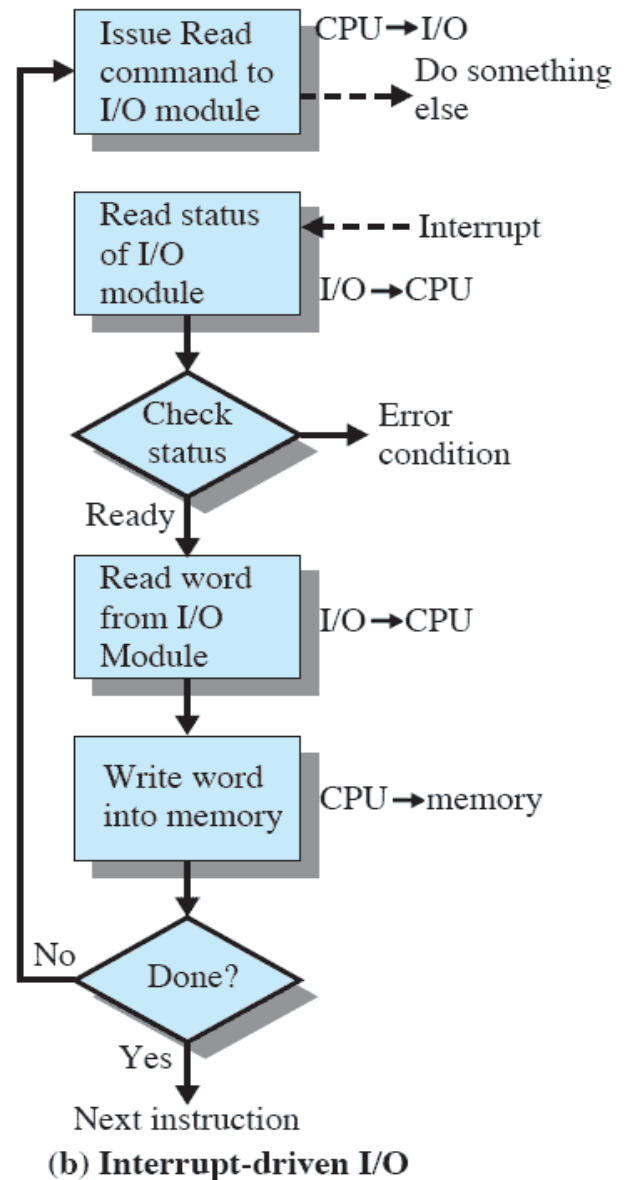
Figure 11-11 Flowchart for CPU program to input data.



## Interrupt initiated I/O:

**Why:** Programmed I/O is a time consuming process, so in order to keep CPU on a busy state introduce an interrupt and special commands to inform the interface to issue an interrupt request signal to CPU, when data are available from the device. In the meantime, the CPU can proceed to execute another program.

- When the device is ready for data transfer, it generates an interrupt request to the computer.
- When there is an external interrupt signal is generated, the CPU will stop the execution of the original program and branches to service program to process the I/O transfer and then returns back to the original program.
- The CPU responds to the interrupt signal by storing the return addresses from program counter into a memory stack and then control branches to a service routine that processes the required I/O transfer.



The way the processor chooses the branch address of the service routine varies from one unit to another. There are two methods for using this:

1. Vectored interrupt
2. Non vectored interrupt

In a **vectored interrupt**, the source that interrupts supplies the branch information to the computer. This information is called the **interrupt vector**.

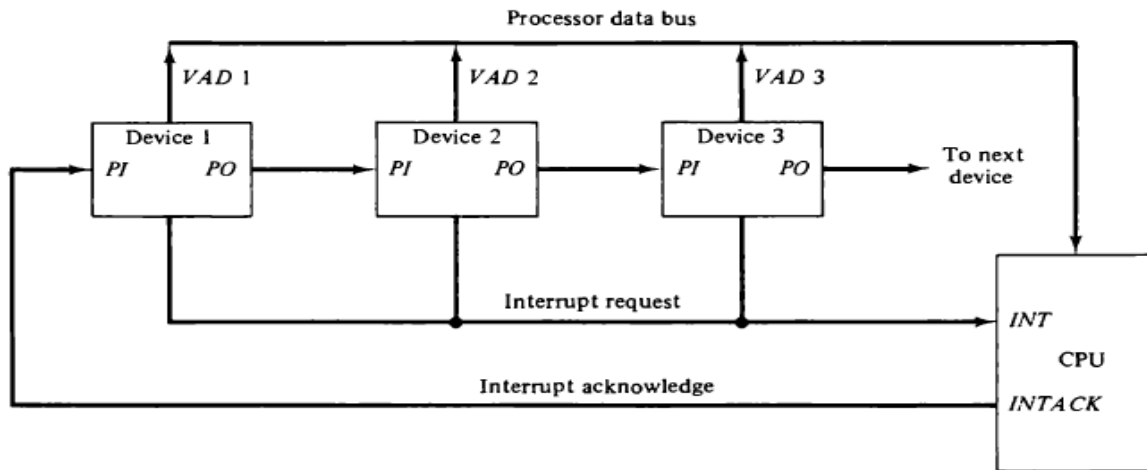
In a **Non vectored interrupt**, the branch address is assigned to fixed location in memory.

### **PRIORITY INTERRUPT:**

- A priority interrupt is a system that establishes a priority over the various sources to determine which condition is to be serviced first when two or more requests arrive simultaneously.
- Devices with high-speed transfers such as magnetic disks are given high priority, and slow devices such as keyboards receive low priority.
- A **polling** procedure is used to identify the highest-priority source.

### **Daisy-Chaining Priority:**

- The daisy-chaining method of establishing priority consists of a serial connection of all devices that request an interrupt.
- The device with the highest priority is placed in the first position, followed by lower-priority devices up to the device with the lowest priority, which is placed last in the chain.
- The interrupt request line is common to all devices and forms a wired logic connection.
- If any device has its interrupt signal in the low-level state, the interrupt line goes to the low-level state and enables the interrupt input in the CPU.
- When no interrupts are pending, the interrupt line stays in the high-level state and no interrupts are recognized by the CPU.
- The CPU responds to an interrupt request by enabling the interrupt acknowledge line. This signal is received by device 1 at its PI (priority in) input.
- The acknowledge signal passes on to the next device through the PO ( priority output) output only if device 1 is not requesting an interrupt.
- If device 1 has a pending interrupt, it blocks the acknowledge signal from the next device by placing a 0 in the PO output. It then proceeds to insert its own interrupt vector address (VAD) into the data bus for the CPU to use during the interrupt cycle.



**Figure 11-12** Daisy-chain priority interrupt.

- A device with a 0 in its PI input generates a 0 in its PO output to inform the next-lower priority device that the acknowledge signal has been blocked.
- A device that is requesting an interrupt and has a 1 in its PI input will interrupt the acknowledge signal by placing a 0 in its PO output.
- If the device does not have pending interrupts, it transmits the acknowledge signal to the next device by placing a 1 in its PO output. Thus the device with PI=1 and PO=0 is the one with the highest priority that is requesting an interrupt, and this device places its VAD on the data bus.
- The daisy chain arrangement gives the highest priority to the device that receives the interrupt acknowledge signal from the CPU.
- The device sets its RF flip-flop when it wants to interrupt the CPU.
- If PO=0, both PO and the enable line to VAD are equal to 0, irrespective of the value of RF.
- If PI=1 and RF=0, then PO=1 and vector address is disabled. This condition passes the acknowledge signal to the next device through PO. The device is active when PI=1 and RF=1.

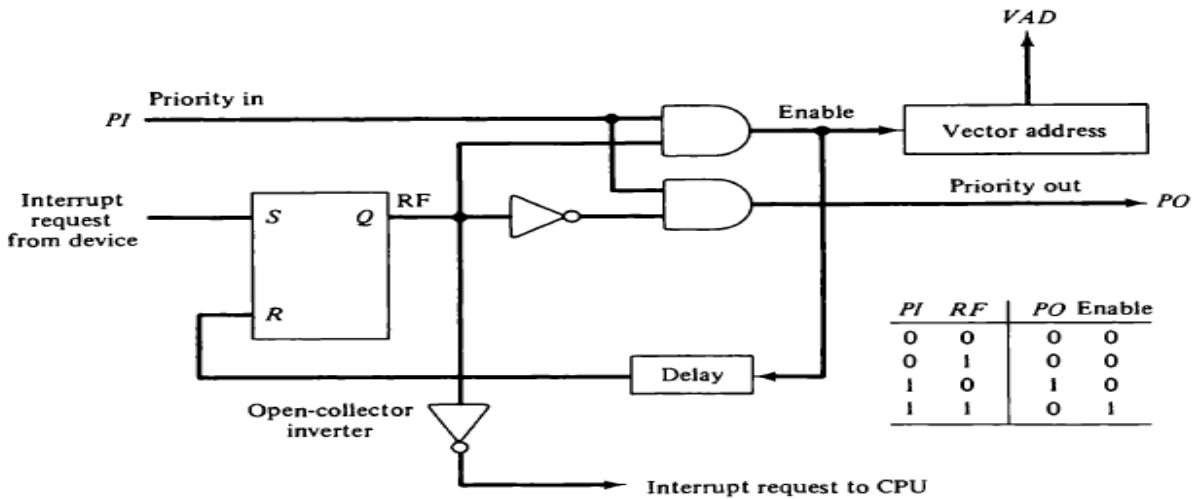


Figure 11-13 One stage of the daisy-chain priority arrangement.

### Parallel Priority Interrupt:

- The parallel priority interrupt method uses a interrupt register whose bits are set separately by the interrupt signal from each device. Priority is establishing according to the position of the bits in the register.
- Mask register purpose is to control the status of each interrupt request. The mask register can be programmed to disable lower-priority interrupts while a higher-priority device is being serviced. It can also provide a facility that allows a high-priority device to interrupt the CPU while a lower-priority device is being serviced.

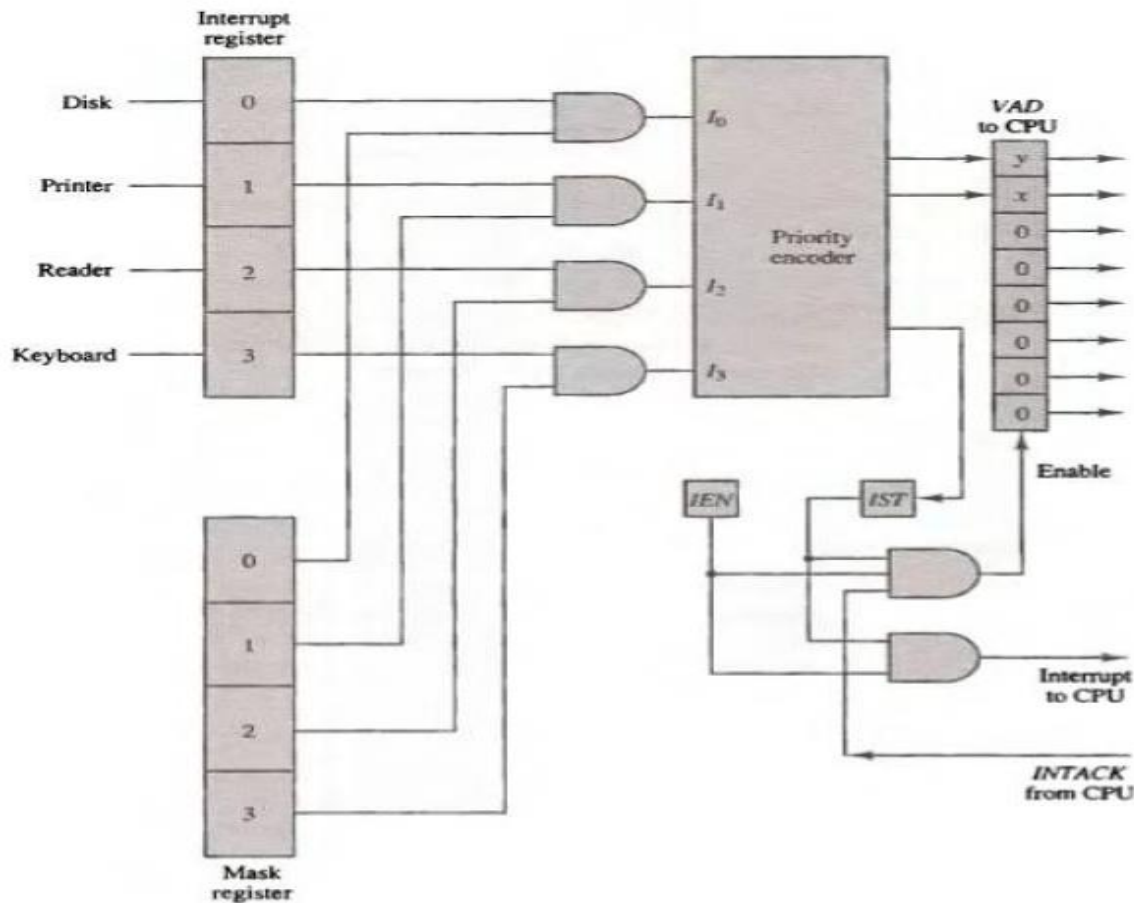


Figure 11-14 Priority interrupt hardware.

- Each interrupt bit and its corresponding mask bit are applied to an AND gate to produce the four inputs to a priority encoder. In this way, an interrupt is recognized only if its corresponding mask bit is set to 1 by the program.
- If IEN set to 1, indicates that the interrupt facility will be used while the current program is running. If both IEN and IST are equal to 1, the CPU goes to an interrupt cycle.
- The priority encoder generates two bits of the vector address, which is transferred to the CPU.

### Priority Encoder:

The x's in the table designate don't-care conditions.

- Input  $I_0$  has the highest priority; so regardless of the values of other inputs, when this input is 1, the output generates an output  $xy=00$ .

- $I_1$  has next priority level. The output is 01 if  $I_1=1$  provided that  $I_0=0$ , regardless of the values of the other two lower-priority inputs.
- The output for  $I_2$  is generated only if higher-priority inputs are 0, and so on down the priority level.

**TABLE 11-2 Priority Encoder Truth Table**

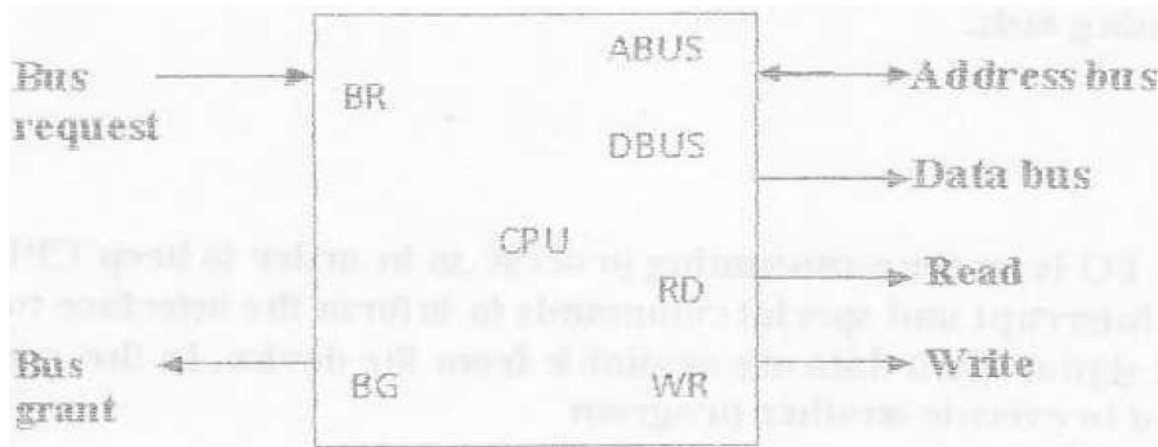
Inputs				Outputs			Boolean functions
$I_0$	$I_1$	$I_2$	$I_3$	$x$	$y$	$IST$	
1	×	×	×	0	0	1	$x = I_0' I_1'$ $y = I_0' I_1 + I_0' I_2'$ $(IST) = I_0 + I_1 + I_2 + I_3$
0	1	×	×	0	1	1	
0	0	1	×	1	0	1	
0	0	0	1	1	1	1	
0	0	0	0	×	×	0	

- The interrupt status  $IST$  is set only when one or more inputs are equal to 1. If all inputs are 0,  $IST$  is cleared to 0 and the other outputs of the encoder are not used, so they are marked with don't-care conditions. This is because the vector address is not transferred to the CPU when  $IST=0$ .

### **DIRECT MEMORY ACCESS (DMA):**

**Definition:** Removing the CPU from the path and letting the peripheral device manage the memory buses directly would improve the speed of transfer. This transfer technique is called “**direct memory access (DMA)**”.

- During DMA transfer, the CPU is idle and has no control of the memory buses. A DMA controller takes over the buses to manage the transfer directly between the I/O device and memory.
- The *bus request* (BR) input is used by the DMA controller to request the CPU to relinquish control of the buses.



- When this input is active, the CPU terminates the execution of the current instruction and places the address bus, the data bus, and the read and write lines into a high-impedance state.
- The CPU activates the *bus grant* (BG) output to inform the external DMA that the buses are in the high-impedance state.
- The DMA that originated the bus request can now take control of the buses to conduct memory transfers without processor intervention.
- When the DMA terminates the transfers, it disables the bus request line. The CPU disables the bus grant, takes control of the buses, and returns to its normal operation. The transfer can be made in several ways.

**DMA burst transfer:** A block sequence consisting of a number of memory words is transferred in a continuous burst while the DMA controller is master of the memory buses.

**cycle stealing:** It allows the DMA controller to transfer one data word at a time, after which it must return control of the buses to the CPU.

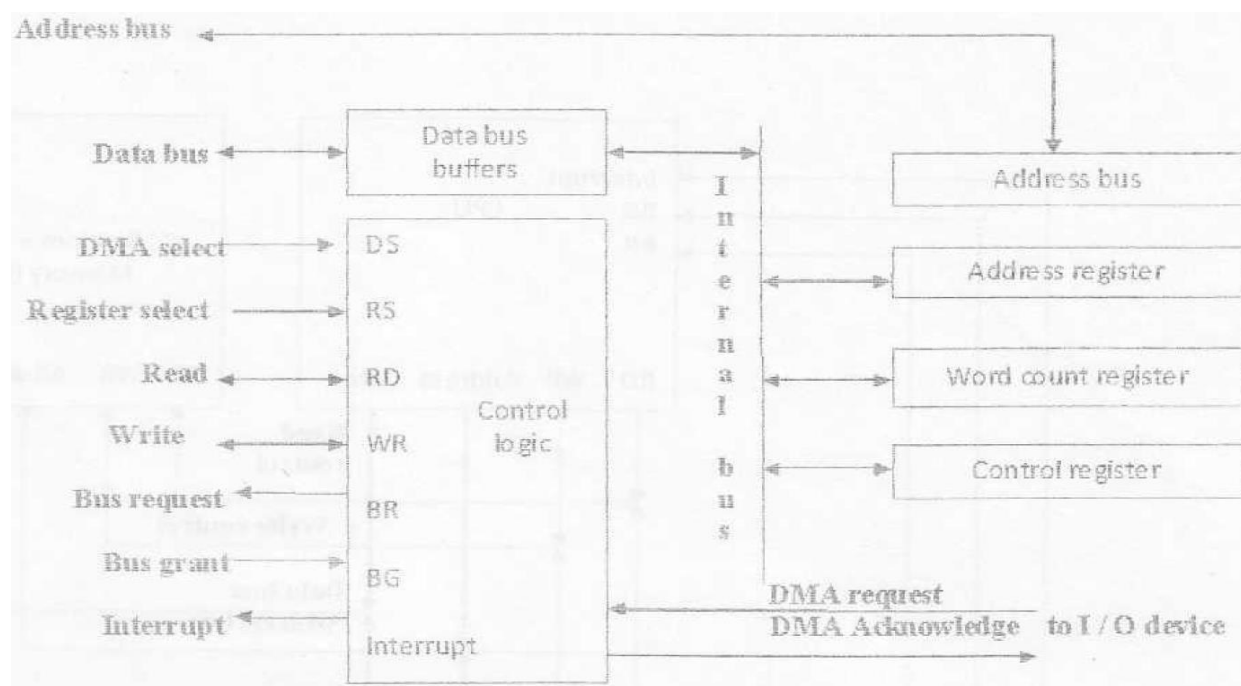
#### **DMA Controller:**

- The unit communicates with the CPU via the data bus and control lines.
- The registers in the DMA are selected by the CPU through the address bus by enabling the DS (DMA select) and RS (register select) inputs. The RD (read) and WR (write) inputs are bidirectional.

- The DMA controller has three registers: an address register, a word count register, and a control register.
- The **address register** contains an address to specify the desired location in memory. The address bits go through bus buffers into the address bus. The address register is incremented after each word that is transferred to memory.
- The **word count register** holds the number of words to be transferred. This register is decremented by one after each word transfer and internally tested for zero.
- The **control register** specifies the mode of transfer.

The CPU initializes the DMA by sending the following information through the data bus:

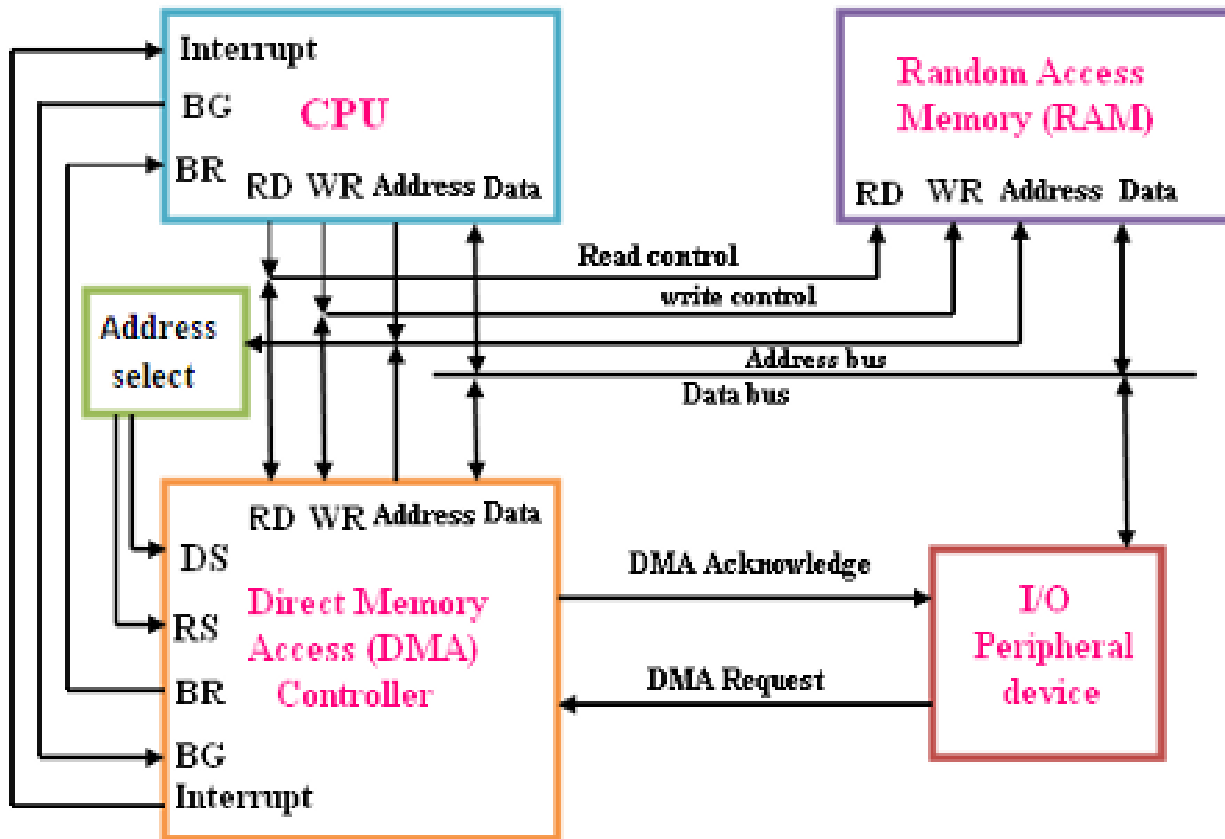
1. The **starting address** of the memory block, where data are available (for read) or where data are to be stored (for write).
2. The **word count**, which is the number of words in the memory block.
3. Control to specify the mode of transfer such as **read** or **write**.
4. A control to start the DMA transfer.





## DMA

### Transfer:



- When the peripheral device sends a DMA request, the DMA controller activates the BR line, informing the CPU to relinquish the buses.
- The CPU responds with its BG line, informing the DMA that its buses are disabled.
- The DMA then puts the current value of its address register into the address bus, initiates the RD or WR signal, and sends a DMA acknowledge to the peripheral device.
- When the peripheral device receives a DMA acknowledge, it puts a word in the data bus (for write) or receives a word from the data bus (for read).
- For each word that is transferred, the DMA increments its addresses register and decrements its word count register.
- If the word count register reaches zero, the DMA stops any further transfer and removes its bus request.

- It is used for fast transfer of information between magnetic disks and memory. It is also useful for updating the display in an interactive terminal.

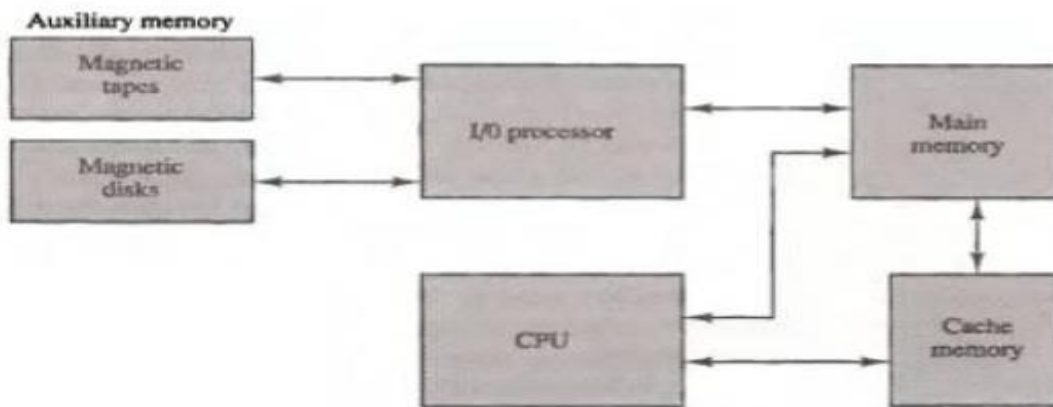
## MEMORY ORGANIZATION:

**Memory hierarchy, Main memory, Auxiliary memory, Associative memory, Cache memory**

### BASIC CONCEPTS: **Memory Hierarchy**

- The memory unit is needed for storing programs and data. The memory unit that communicates directly with the CPU is called the **main memory**.
- Devices that provide backup storage are called “**auxiliary memory**”. The most common auxiliary memory devices used in computer systems are **magnetic disks** and **tapes**. They are used for storing system programs, large data files and other backup information.
- Only programs and data currently needed by the processor reside in main memory. Other information is stored in auxiliary memory & transferred to main memory when needed.
- The main memory occupies a central position by being able to communicate directly with the CPU and with auxiliary memory devices through an I/O processor.
- When programs are not residing in main memory which are needed by the CPU are brought from auxiliary memory. Programs are not currently needed in main memory are transferred into auxiliary memory to provide space for currently used programs and data.
- A special very high speed memory called a **cache** is sometimes used to increase the speed of processing by making current programs and data available to the CPU at a rapid rate.

Figure 12-1 Memory hierarchy in a computer system.



- The cache memory is employed in computer systems to compensate for the speed differential between main memory access time and processor logic.
- The cache is used for storing segments of programs currently being executed in the CPU and temporary data frequently needed in the present calculations.
- The storage capacity of the memory increases, the cost per bit for storing binary information decreases and the access time of the memory becomes longer.
- The auxiliary memory has a large storage capacity, is relatively inexpensive, but has low access speed compared to main memory.
- The cache memory is very small, relatively expensive, and has very high access speed. Thus the memory access speed increases, so does its relative cost.
- The overall goal of using a memory hierarchy is to obtain the highest possible average access speed while minimizing the total cost of the entire memory system.

### Multi-programming:

- Many operating systems are designed to enable the CPU to process a no of independent programs concurrently. This is called “**multiprogramming**” refers to the existence of two or more programs in different parts of the memory hierarchy at the same time.
- In multiprogramming system, when one program is waiting for input or output transfer, there is another program ready to utilize the CPU.

- Computer programs are sometimes too long to be accommodated in the total space available in the main memory. A computer system uses many programs and all the programs cannot reside in main memory at all times.
- A program with its data normally resides in auxiliary memory. When the program or a segment of the program is to be executed, it is transferred to main memory to be executed by the CPU.
- The part of the computer system that supervises the flow of information between auxiliary memory and main memory is called the **memory management system**.

## MAIN MEMORY:

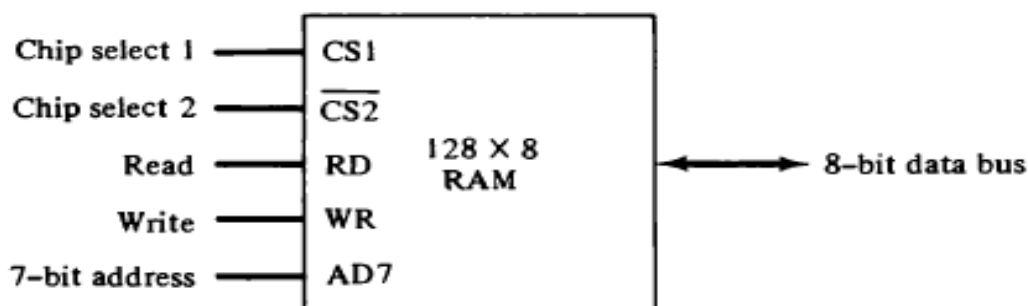
- The main memory is a relatively large and fast memory used to store programs and data during the computer operation.
- Main memory is made up of RAM integrated circuit chips, but a portion of the memory may be constructed with ROM chips.

## SEMICONDUCTOR RAM MEMORIES:

- RAM is used to designate read/write to distinguish it from a ROM.
- RAM is used for storing the bulk amount of the programs and data that are subject to change.
- The RAM portion of main memory is needed for storing an initial program called a **bootstrap loader**. The Bootstrap loader is a program whose function is to start the computer software operating when power is turned on.
- RAM is volatile; its contents are destroyed when power is turned off.
- A RAM chips is better suited for communication with the CPU. The feature is a bidirectional data that allows the transfer of data either from memory to CPU during a read operation or from CPU to memory during a write operation.

- When CS1=1, and  $\overline{CS2}$ =0, the memory can be placed in a write or read mode.
- The RD & WR signals control the memory operation as well as the bus buffers associated with the bidirectional data bus.
  - When the WR input is enabled, the memory stores a byte from the data bus into a location specified by the address input lines.
  - When the RD input is enabled, the content of the selected byte is placed into the data bus.

**Figure 12-2 Typical RAM chip.**



**(a) Block diagram**

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

**(b) Function table**

- The capacity of the memory is 128 words of 8 bits (one byte) per word. This requires a 7-bit address and an 8-bit bidirectional data bus.
- The read and write inputs specify the memory operation and the two chip select (CS) control inputs are for enabling the chip only when it is selected by the microprocessor.
- The read and write inputs are sometimes combined into one line labeled R/W. When the chip is selected, the two binary states in this line specify the two operations of read or write.

- Integrated RAM chips are available in two possible operating modes, **static** and **dynamic**.

#### **Static RAM:**

- The static RAM consists essentially of internal flip-flops that store the binary information.
- The static RAM is easier to use and has shorter read and write cycles.

#### **Dynamic RAM:**

- The dynamic RAM stores the binary information in the form of electric charges that are applied to capacitors. The capacitors are provided inside the chip by MOS transistors. The stored charges on the capacitors tend to discharge with time and the capacitors must be periodically recharged by refreshing the dynamic memory.
- The dynamic RAM offers reduced power consumption and large storage capacity in a single memory chip.

<b>Static RAM</b>	<b>Dynamic RAM</b>
1. The static RAM consists essentially of internal flip-flops that store the binary information.	The dynamic RAM stores the binary information in the form of electric charges that are applied to capacitors. The capacitors are provided inside the chip by MOS transistors. The stored charges on the capacitors tend to discharge with time and the capacitors must be periodically recharged by refreshing the dynamic memory.
2. The static RAM is easier to use	The dynamic RAM offers reduced power consumption
3. It has shorter read and write cycles.	It has large storage capacity in a single memory chip.

## READ-ONLY MEMORIES (ROM CHIPS):

- ROM is used for storing programs that are permanently resided in the computer and that don't change.
- The contents of ROM remain unchanged after power is turned off and on again. (non volatile).
- A ROM can only read, the data bus only in an output made.
- For the same size chip, It is possible to have more bits of ROM than of RAM, because the internal binary cells in ROM occupy less space than in RAM. For this reason, the diagram specifies a 512 byte ROM, while the RAM has 512bytes.
- The 9-address lines specify any one of the 512bytes stored in it. The two chips select inputs must be CS1=1 &  $\overline{CS2}=0$  for the unit operate. Otherwise, the data bus is in a high-impedance state.

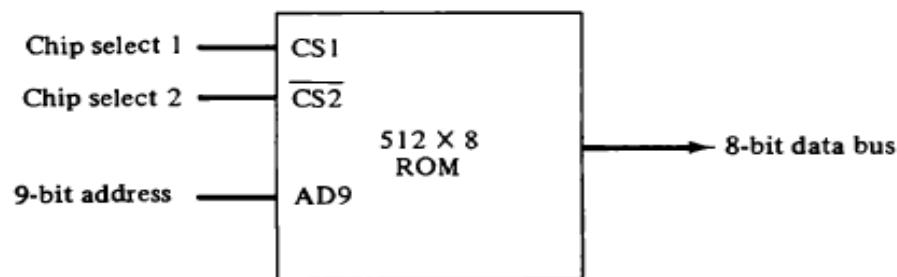


Figure 12-3 Typical ROM chip.

## Differences between RAM and ROM

RAM	ROM
1. RAM is used to designate read/write to distinguish it from a ROM.	ROM is used to designate read operation only.
2. RAM is used for storing the bulk of the programs and data that are subject to change.	ROM is used for storing programs that are permanently resided in the computer and that don't change.

3. RAM is volatile; its contents are destroyed when power is turned off.	ROM is nonvolatile; its contents are not destroyed when power is turned off.
4. It has a bidirectional data bus.	It has a unidirectional data bus
5. The RAM portion of main memory is needed for storing an initial program called a <b>bootstrap loader</b> .	

### MEMORY ADDRESS MAP:

- The addressing of memory can be establish by the means of table that specifies the memory address assigned to each clip. This is called **memory address map**.
- A computer system needs 512 bytes of RAM and 512 bytes of ROM. There are 16 lines in the address bus, the table shows only 10 lines because the other 6 or not used in this example, and are assumed to be zero.
- The small x's under the address bus lines designates those lines that must be connected to the address inputs in the each chip.
- The RAM chips have 128 bytes and need 7 address lines. The ROM chip has 512 bytes and needs 9 address lines.
- The x's are always assign to the low – order bus lines 1 through 7 for the RAM and lines 1 through 9 for the ROM. It is necessary to distinguish between 4 RAM chips by assigning to each a different address.
- When line 10 is 0, the CPU selects a RAM, and
- when this line is equal to 1, it selects the ROM
- The table clearly shows that the 9 low-order bus lines constitute a memory space for RAM equal to  $2^9=512$  bytes.



**TABLE 12-1 Memory Address Map for Microprocomputer**

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

- The first hexadecimal digit represent lines 13 to 16 and is always 0.

The next hexadecimal represent lines 9 to 12, but lines 11 and 12 are always 0.

These x's represent a binary number that can range from all-0's to an all -1's value.

#### **Memory connection to cpu:**

- RAM AND ROM chips are connected to CPU through the data and address buses. This configuration gives a memory capacity of 512 bytes of RAM and 512 bytes of ROM.
- Each RAM receives the 7 low –order bytes of the address bus to select the one of 128 possible bytes. The particular RAM chip select is determined from lines 8 and 9 in the address bus. This is done through a 2x4 decoder whose output goes to the CS1 input in each RAM chip.
- Thus, when address lines 8 and 9 are equal to 00, the first RAM chip is selected and when 01, the second RAM chip is selected and so on. The RD &WR output from the microprocessor are applied to the inputs of each RAM chip.
- The selection between RAM & ROM is achieved through bus line 10. The RAM's are selected when the bit in this line is 0, and the ROM when the bit is 1.
- The other chip selected input in the ROM is connected to the RD control line for the ROM chip to be enabled only during a read operation.

- Address bus lines 1 to 9 are applied to the input address of ROM without going through the decoder. This assigns address 0 to 511 to RAM and 512 to 1023 to ROM.
- The data bus of the ROM has the only an output capability, where as the data bus connected to RAM's can transfer information in the both directions.

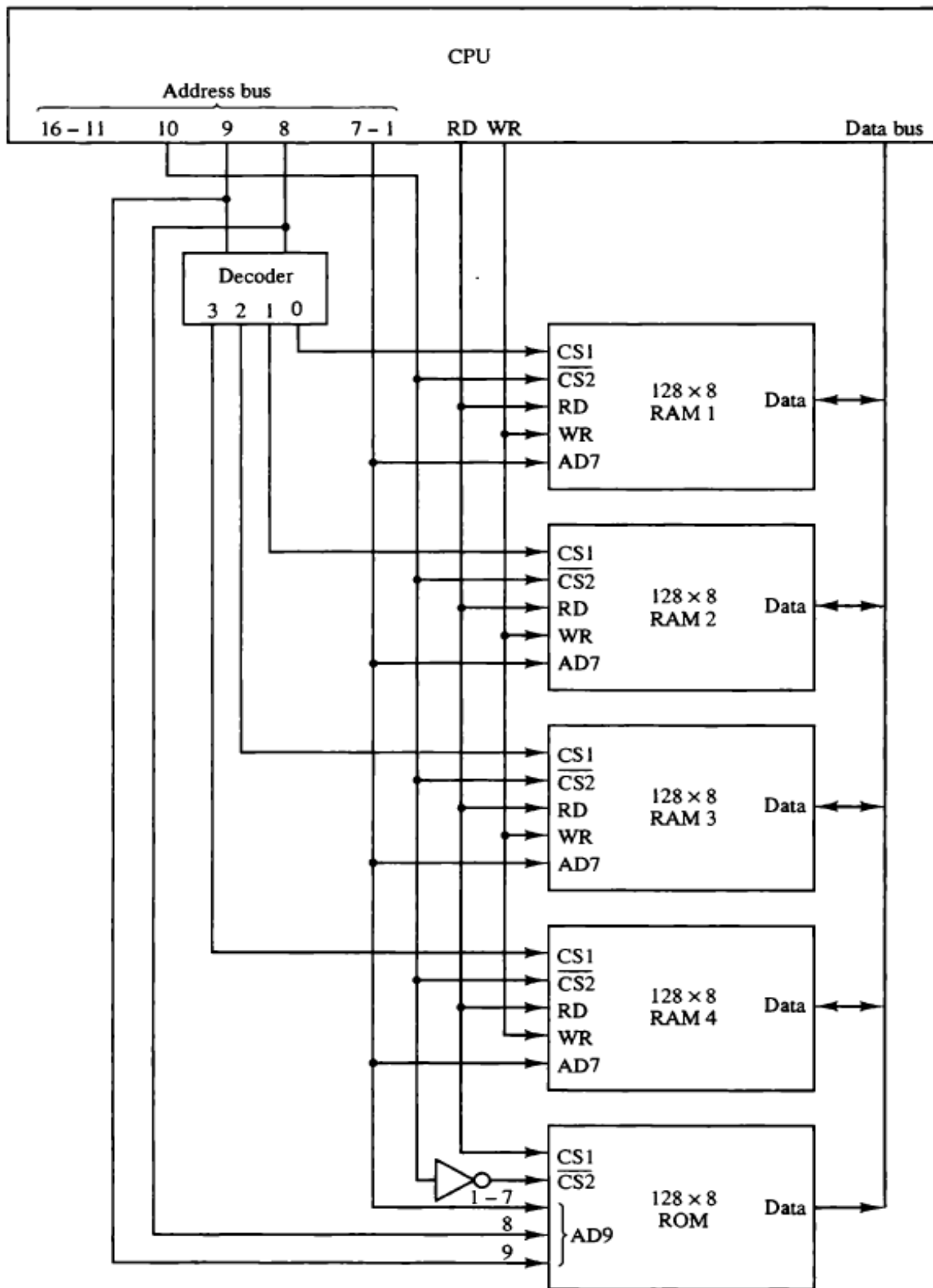


Figure 12-4 Memory connection to the CPU.

## **Types of ROM:**

### **PROM: Programmable Read-Only Memory**

- It is a memory chip on which data can be written only once. Once a program has been written onto a **PROM**, it remains there forever.
- Unlike **RAM**, **PROMs** retain their contents when the computer is turned off.
- The difference between a **PROM** and a **ROM** (read-only memory) is that a **PROM** is manufactured as blank memory, whereas a **ROM** is programmed during the manufacturing process.
- To write data onto a **PROM** chip, you need a special device called a **PROM programmer** or **PROM burner**. The process of programming a PROM is sometimes called **burning the PROM**.

### **EPROM: Erasable Programmable Read-Only Memory**

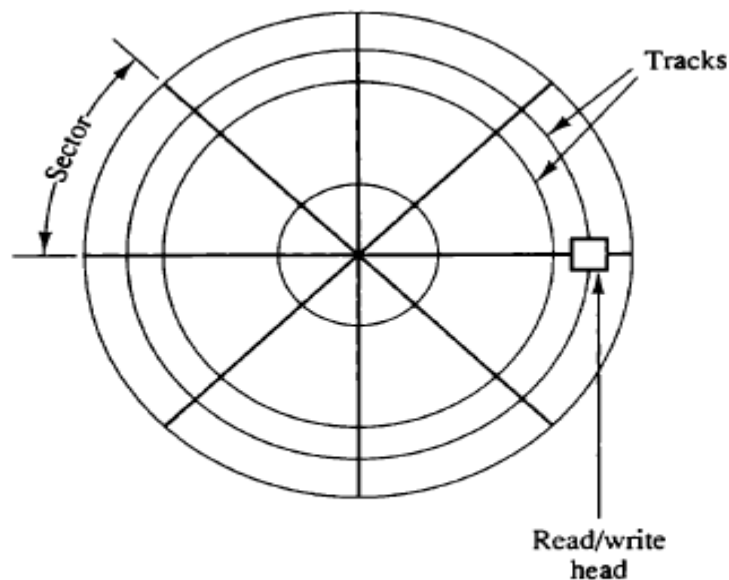
- **EPROM** is a special type of memory that retains its contents until it is exposed to ultraviolet light.
- The ultraviolet light clears its contents, making it possible to reprogram the memory.
- To write to and erase an **EPROM**, you need a special device called a **PROM programmer** or **PROM burner**.

### **EEPROM: Electrically Erasable Programmable Read-Only Memory**

- **EEPROM** is a special type of **PROM** that can be erased by exposing it to an electrical charge.
- Like other types of **PROM**, **EEPROM** retains its contents even when the power is turned off. Also like other types of **ROM**, **EEPROM** is not as fast as **RAM**.

## AUXILIARY MEMORY: (SECONDARY STORAGE)

- The most common auxiliary memory devices used in the computer systems are **magnetic discs** and **tapes**. The most important characteristics of any devices are its access mode, access time, transfer data, capacity and cost.
- The average time required to reach a storage location in memory and obtain its contents is called the **access time**.
- In disks and tapes, the access time consists of **seek time** required to position the read-write head to a location and a **transfer time** required to transfer data to or from the device.
- Because the seek time is usually much longer than the transfer time, auxiliary storage is organized in records or blocks.



**Figure 12-5 Magnetic disk.**

- A record is specified number of characters or words. Reading or writing is always done on entire records. The transfer rate is the number of characters or words that the devices can transfer for second, after it has been positioned at the beginning of the record.
- The recording surface rotates at uniform speed and is not started or stopped during access operations. bits are recorded as magnetic spots on the surface as it passes a stationary mechanism called a **write head**.
- Stored bits are detected by change in the magnetic field produced by the record spot on the surface as it passes through the read head.

- Bits are stored in the magnetized surface in spots along concentric circles called **tracks**. The tracks are commonly divided into sections called **sectors**. In the most systems, the min quantity of information which can be transferred is a sector.
- The track address bits are used to move the head into the specified track position before reading or writing.
- After the read/write heads are positioned in the specified track, the system has to wait until the rotating disk reaches the specified sector under the R/w head.
- Information transfer is very fast once the beginning of the sector has been reached. Disks may have multiple heads and simultaneous transfer of bits from several tracks at the same time.
- If bits are recorded with equal density, some tracks will contain more recorded bits than others. To make all the records in a sector of all lengths, some disks use as a variable recording density with higher density on tracks nearer the center than on tracks near the circumference.
- Disks that are permanently attached to the unit assemble and cannot be removed by the occasional user called **hard disk**. A disk drive with removable disk is called a **floppy disk**.
- The disks used with a floppy disk drive are small removable disks made of plastic coated with magnetic recording material.

#### **Magnetic tape:**

- A magnetic Tape transport consists of the electrical, mechanical, & electronic components to provide the parts & control mechanisms for a magnetic tape unit.
- The tape itself is a strip of plastic coated with a magnetic recording medium for 9 bits are recorded simultaneously to form a character together with a parity bit.
- Read/Write heads are mounted one in each track so that data can be recorded and read as as a sequence of characters.
- Magnetic tape units can be stopped, started to more forward or in reverse or can be rewound.

#### **RAID: Redundancy Array of Independent Disks**

- This is an additional disk which can improve system performance.
- In RAID, we use an array of disks, these disks can operate independently.
- Multiple i/o requests can be handled in parallel if the requested data is distributed across multiple.

#### **Benefits:**

- RAID technology prevents data loss due to disk failure.

- RAID technology can be implemented in hardware or software.

**Applications:** Servers make use of raid technology.

**RAID levels:** RAID0, RAID1, RAID2, RAID3, RAID4, RAID5, RAID6.

**Common characteristics:**

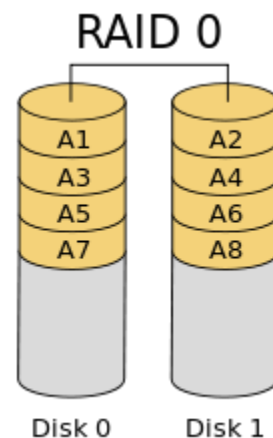
- A set of physical disk drives
- The operating system views these separating disks as a single logical disk.
- Redundant disk capacity is used to store parity information which can help in recovering data in case of disk failure.

**RAID level 0:**

- It divides data into block units & writes them across a no of disks. This is called data stripping.
- There is no parity checking of data so, if one drive gets corrupted then all the data would be lost. Thus RAID 0 does not support data recovery.

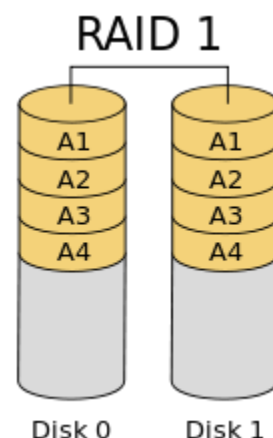
**Advantages:**

- It increases speed
- Implementation is easy
- No overhead of parity calculation



**RAID LEVEL 1:**

- Same data is placed on multiple disks, it is also called **data mirroring**.
- A read request can be executed by either of the two disks.
- A write request means that both the disks must be updated. This can be done in parallel.
- Recovery from failure is simple. If one drive fails we just have to



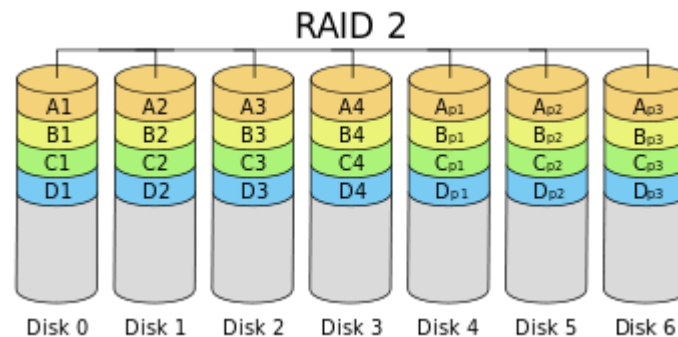
access data from the second drive.

- RAID 1 is used to store systems software.

**Disadvantage:** since data is duplicated, storage costs increase.

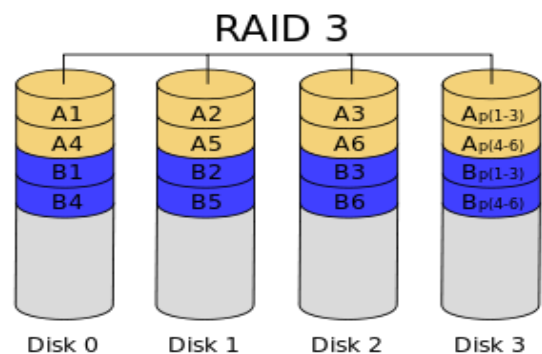
### RAID LEVEL2:

- In RAID2 mechanism, all disks participate in the execution of every i/o request.
- Data stripping is used.
- Error correcting code is also calculated & stored with data.
- not implemented in practice due to high costs & overheads.



### RAID LEVEL 3:

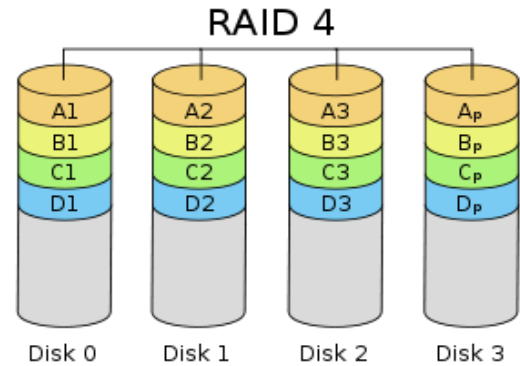
- Data is divided into byte units & written across multiple disk drives.
- Parity information is stored for each disk section & written to a dedicated parity drive.
- Thus disks can be accessed in parallel.
- Data can be transmitted in bulk. Thus high speed data transmission is possible.
- In case of drive failure, the parity drive is accessed & data is reconstructed from the remaining devices.





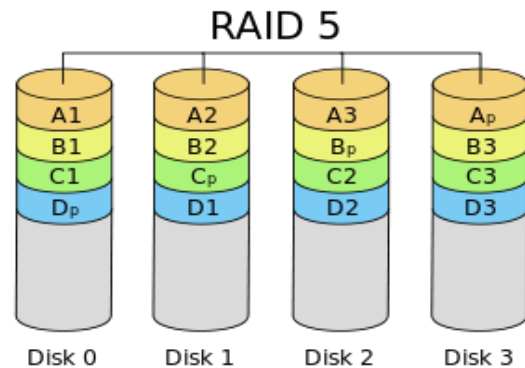
#### RAID LEVEL 4:

- Data stripping size is large and the parity bits are stored in the corresponding strip on the parity disk.
- For each read/write operation parity bits are checked for data reliability



#### RAID LEVEL 5:

The data bits are organized in a similar fashion to RAID level 4. The difference is that RAID 5 distributes the parity strips across all disks.



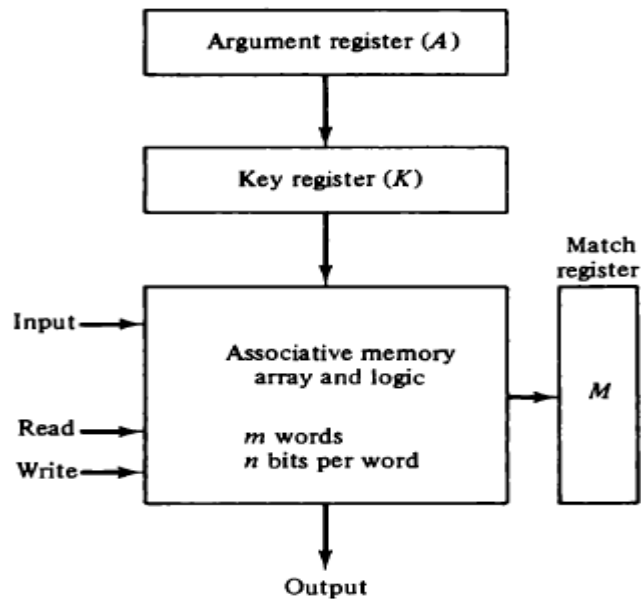
#### ASSOCIATE MEMORY:

- The time required to find an item stored in memory can be reduced considered if stored data can be identified for access by content of the data itself rather than by an address. A memory unit accessed by its content is called an **associative memory** or **content addressable memory (CAM)**.
- An Associate memory is more expensive than a RAM, because each cell must have storage capacity. Associate memories are used in application where the search time is very critical and must be very short.

#### Hardware organization:

The block diagram of associate memory is shown in fig. It consists of a memory array and logic form words and n bits per word.

**Figure 12-6** Block diagram of associative memory.



- The argument register A and Key register K each have n bits, one for each bit of a word. The matches register M has m bits, one for each memory word.
- Each word in memory is compared in parallel with the content of the argument register.
- The words that match the bits of the argument register set a corresponding bit in the match register.
- After the matching process, those bits in the register that have been set indicate the fact that their corresponding word has been matched.
- The key register provides a mask for choosing a particular field or key in the argument word. The entire argument is compared with each memory word if the key register contains all 1's otherwise, only those bits in the argument that have 1's in their corresponding position of the key register are compared.

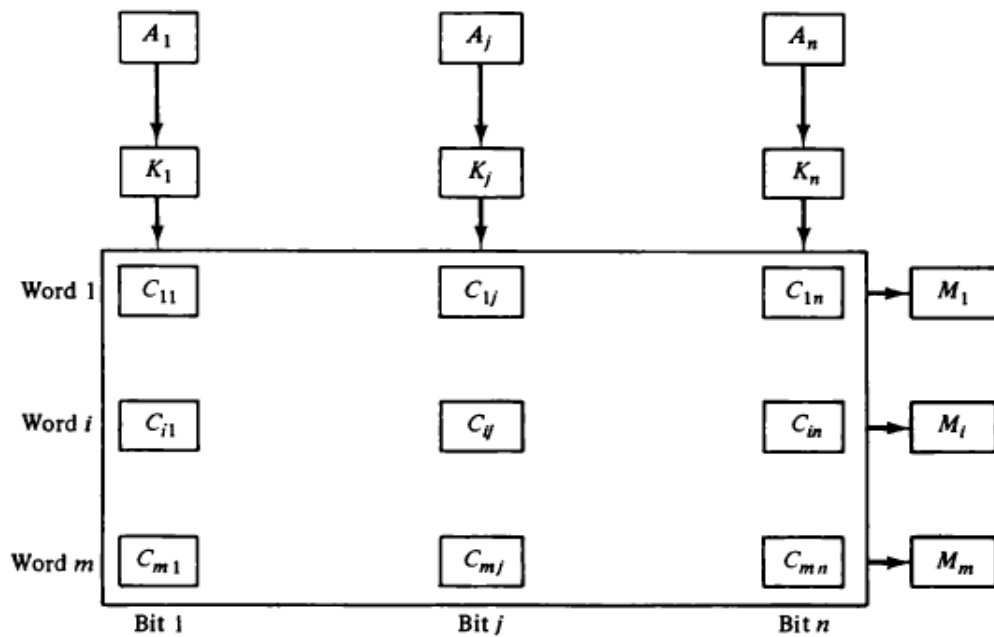
**example:**

<b>A</b>	<b>101 111100</b>	
<b>K</b>	<b>111 000000</b>	
<b>Word 1</b>	<b>100 111100</b>	<b>no match</b>
<b>Word 2</b>	<b>101 000001</b>	<b>match</b>

- Only the three left most bits of A are compared with memory words because k has 1's in three positions.
- Word2 matches the unmasked argument field because the three left most bits of the argument and word are equal.

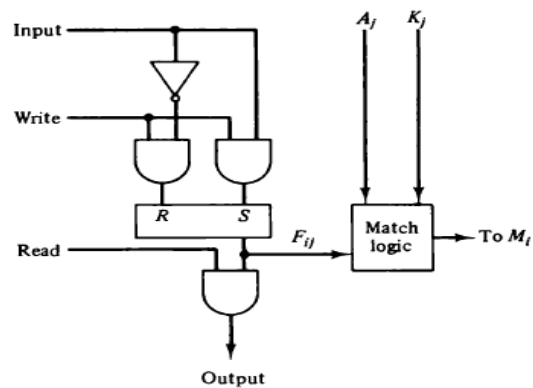
- The cells in the array are marked by the letter **C** with two subscripts. The first subscript gives the word number and the second specifies the bit position in the word.
- Thus cell  $C_{ij}$  is the cell for bit  $j$  in the word  $i$ . A bit  $A_j$  in the argument register is compared with all the bits in column  $j$  of the array provided that  $k_j=1$ . This is done for all columns  $j=1,2,\dots,n$ .
- If a match occurs between all the unmasked bits of the arguments and the bits in the word  $i$ , the corresponding bit  $M_i$  in the match register is set to 1.
- If one or more unmasked bits of the argument and the word don't match  $M_i$  is cleared to 0.

Figure 12-7 Associative memory of  $m$  word,  $n$  cells per word.



- The diagram consists of a flip-flop storage elements  $F_{ij}$  and circuits for reading, writing, and matching the cell. The input bit is transferred into the storage cell during a write operation.
- The bit stored is read out during read operation. The match logic compares the content of the storage cell with the corresponding unmasked bit of argument and provides an output for the decision logic that sets the bit in  $M_i$ .

Figure 12-8 One cell of associative memory.



**Match logic:**

Match logic for each word can be derived from the comparison Algorithm for two binary numbers.

First, we neglect the key bits and compare the argument in A if  $A_j = F_{ij}$  for  $j=1,2,\dots,n$ .

Two bits are equal if they are both 1 or both 0. The equality of two bits can be expressed logically by the Boolean function.

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

Where  $x_j = 1$  if the pair of bits in position  $j$  are equal;

$$x_j = 0 \text{ otherwise}$$

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

$$M_i = x_1 x_2 x_3 \dots \dots \dots n.$$

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

We now include the **key bit**  $K_j$  in the comparison Logic.

If  $K_j = 0$ , the corresponding bits of  $A_j$  and  $F_{ij}$  need no comparison

If  $K_j = 1$ , they must be compared.

The requirement is achieved by ORing each term with  $K_j^1$

$$\begin{aligned} x_j + k_j^1 &= x_j & \text{if } k_j=1 \\ &= 1 & \text{if } k_j=0 \end{aligned}$$

When  $k_j = 1$ , we have  $k_j' = 0$  and  $x_j + 0 = x_j$

When  $k_j = 0$ , we have  $k_j' = 1$  and  $x_j + 1 = 1$ .

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

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

$$M_i = (x_1 + k_1') (x_2 + k_2') (x_3 + k_3') \dots \dots \dots (x_n + k_n')$$

A match will occur and  $M_i$  will be equal to 1 if all terms are equal to 1. If we substitute the original definition of  $x_j$ , the Boolean function is

$$M_i = \prod_{j=1}^n (A_j F_{ij} + A_j' F_{ij}' + K_j')$$

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

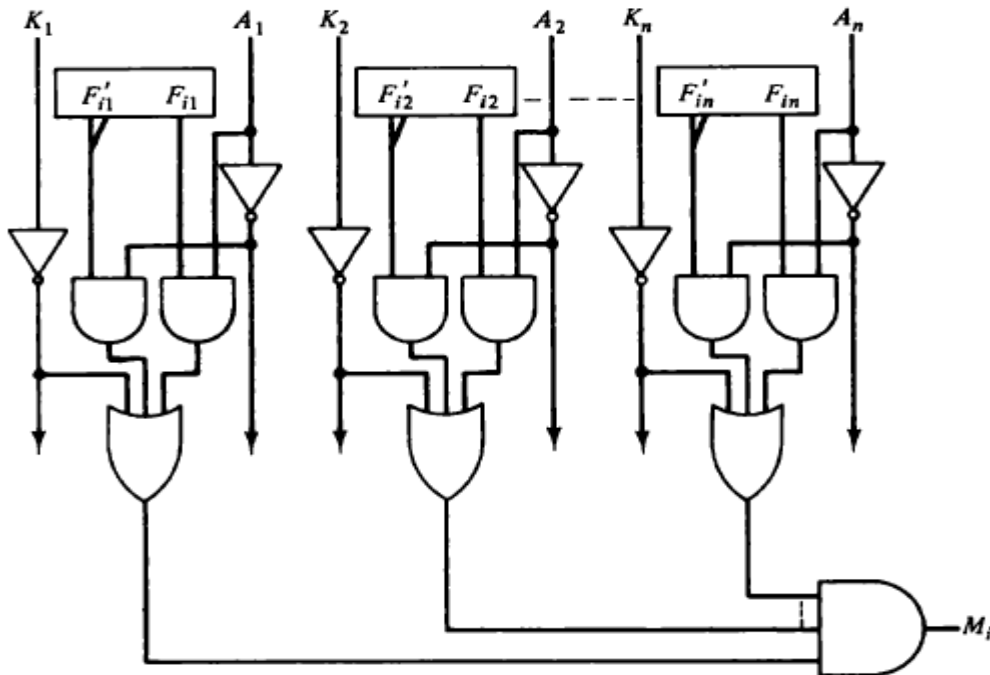


Figure 12-9 Match logic for one word of associative memory.

**Note:**  $M_i$  will be logic 1 if a match occurs and 0 if no matches occurs.

**Read operation:** If more than one word in memory matches the un masked argument field, all the matched words will have 1's in the corresponding bit position of the match register. It is then necessary to scan the bits of the match register one at a time. The matched words are read in sequence by applying a Read signal to each word line whose corresponding  $M_i$  bit is a 1.

#### Write operation:

If unwanted words have to be deleted and new words inserted one at a time, there is a need for special register to distinguish between active and inactive words. This register sometimes called a **Tag register** would have a many bits as there are words in memory.

For every active word stored in memory, the corresponding bit in Tag register is set to 1. A word is deleted from memory by clearing its tag bit to 0. If the Tag register is cleared to 0, then it is used to specify an empty location.

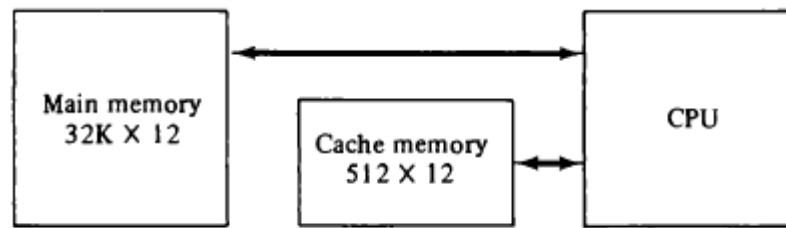
### **CACHE MEMORY:**

- If the active portions of the program and the data are placed in a fast small memory, the average memory access time can be reduced, thus reducing the total execution time of the program. Such a fast small memory is called **cache memory**. The cache memory access time is less than access time of main memory by a factor 5 to 11.
- The cache is the only a small fraction of the size of main memory. It is placed between the CPU and main memory.
- When the CPU needs to access memory, the cache is examined. If the word is found in the cache, it is read from the fast memory. If the word addressed by the CPU is not found in the cache, the main memory is accessed to read the word.
- The performance of cache memory is frequently measured in terms of quantity called **hit ratio**. When the CPU refers to memory and finds the word in cache, it is said to produce a **hit**. If the word is not found in cache, it is in main memory and it comes a **miss**.
- The ratio of the no of hits divided by the total CPU references to memory (hits plus misses) is the **hit ratio**.
- The average memory access time of a computer system can be improved considerably by use of a cache. If the hit ratio is high enough so that most of the time the CPU accesses the cache instead of main memory, the average access time is closer to the access time of the fast cache memory.
- The Transformation of data from main memory to cache memory is referred to as mapping process. Three types of mapping procedures.

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

### **Example for three types of mapping:**

The main memory can store 32k words of 12 bits each. The cache is capable of storing 512 of these words at any given time. For each word stored in cache, there is a duplicate copy in main memory.



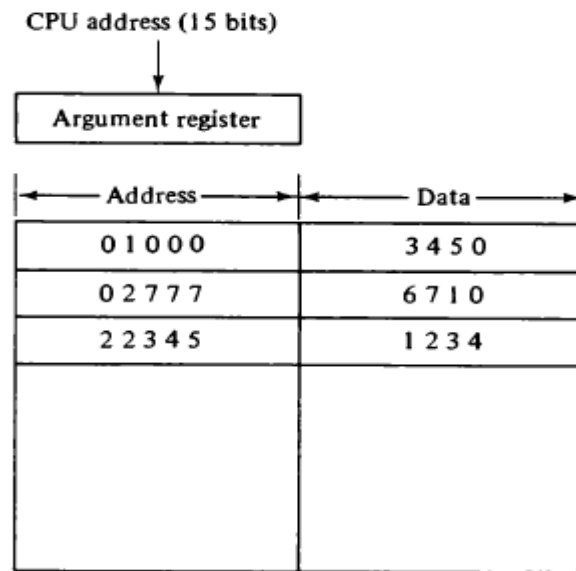
**Figure 12-10** Example of cache memory.

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

#### **Associative mapping:**

- The fastest and most flexible cache organization uses an associative memory. The associative memory stores both address and content (data) of the memory word. This permits any location in cache to store any word from main memory.
- The diagram shows 3 words presently stored in the cache. The address value of 15 bits is shown as 5-digit octal number and its corresponding 12-bit word is shown as a 4-digit octal number.
- A CPU address of 15 bits is placed in the argument register and the associative memory is searched for matching address. If the address is found, the corresponding 12-bit data is read and sent to the CPU.

**Figure 12-11** Associative mapping cache (all numbers in octal).



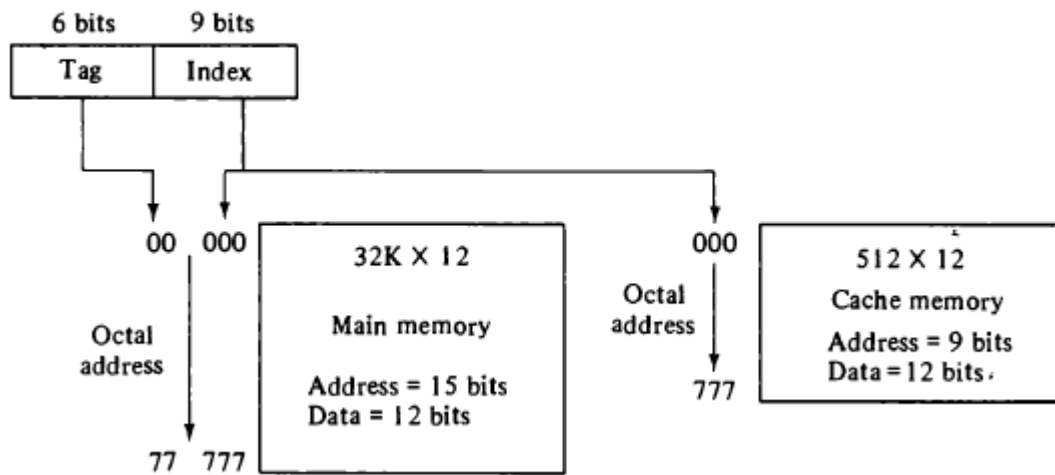
- If no match occurs, the main memory is accessed for the word. The address data pair is then transferred to the associative cache memory.
- If the cache is full, an address-data pair must be displaced to make room for a pair that is needed and not presently in the cache. The decision as to what pair is replaced is determined from the replacement algorithm that the designer chooses for the cache. For example first-in-first- out (FIFO) replacement policy.

**Direct mapping:**

- The CPU address of 15 bits is divided into two fields. The 9 least significant bits constitute the index field and the remaining 6 bits from the tag field.
- The main memory needs an address that includes both the tag and the indexed bits. The no of bits in the indexed field is equal to the number of address bits required to access the cache memory.



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



- For example, there are  $2^k$  words in the cache memory and  $2^n$  words in main memory. The  $n$ -bit address is divided into two fields:  $k$  bits for the **indexed** field and  $n-k$  bits for **tag** field.
- The direct mapping cache organization uses the  $n$ -bit address to access the main memory and the  $k$ -bit index to access cache. The internal organization of the words in the cache memory is shown in fig.
- Each word in cache consists of the data word and its associated tag. When a new word is first brought into the cache, the tag bits are stored along side the data bits.
- When the CPU generates a memory request, the indexed field is used for the address to access the cache. The tag field of the CPU address is compared with the tag in the word read from the cache.
- If the two tags match, there is a hit and desired data word is in cache. If there is no match, there is a miss and the required word is read from main memory. It is then stored in the cache together with the new tag, replacing the previous value.

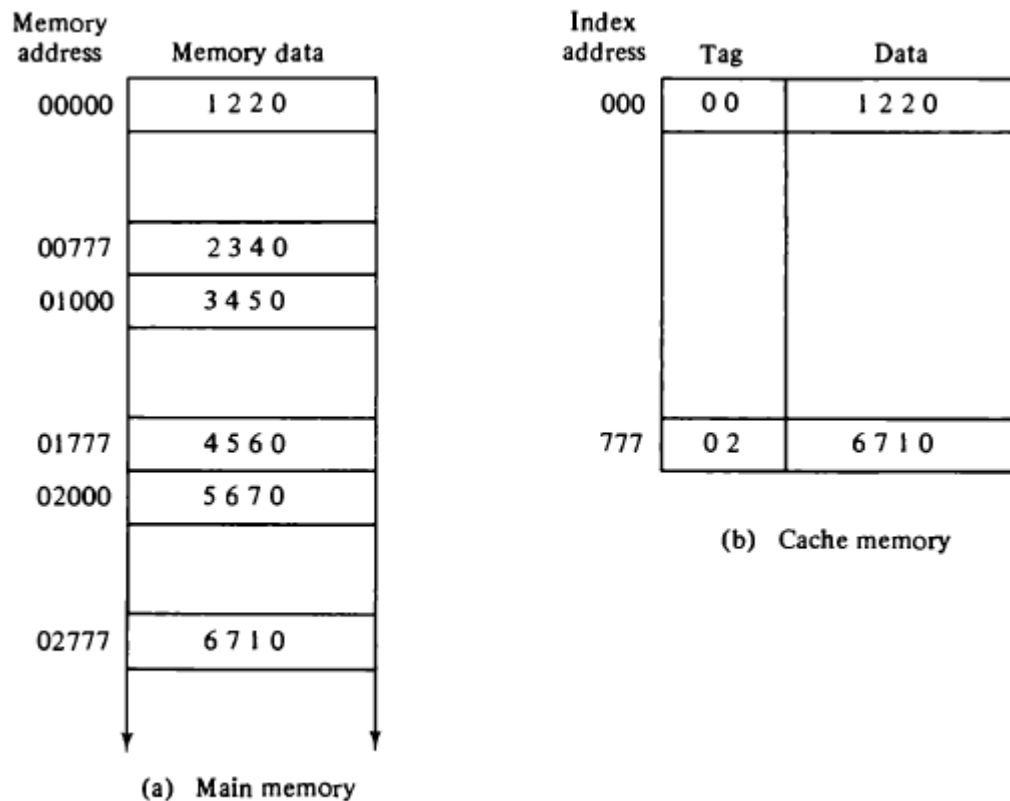


Figure 12-13 Direct mapping cache organization.

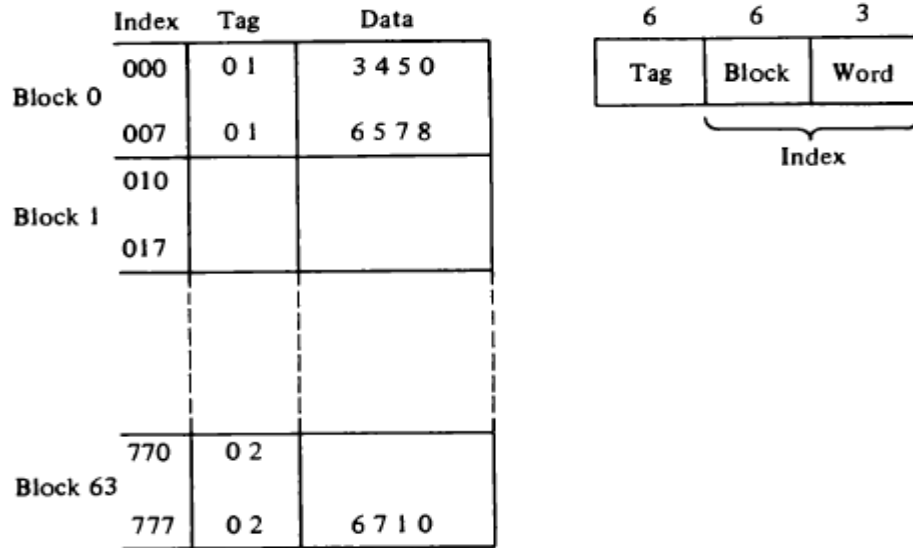
- The word at address 0 is presently stored in the cache (index=000, tag=00, data=1200).
- Suppose that the CPU now wants to access the word at address 0200. The index address is 000, so it is used to access the cache. The two tags are then compared. The cache tag is 00 but the address tag is 02, Which does not produce a match.
- Therefore, the main memory is accessed and the data word 5670 is transferred to the CPU. The cache word at index address 000 is then replaced with a tag of 0.2 and data of 5670.

#### Disadvantage:

The hit ratio can drop considerably if two or more words whose addresses have the same index but different tags are accessed repeatedly.

#### Direct mapping cache with block size of 8 words:

- The index field is now divided into two fields: **block** field and **word** field. In a 512-word cache, there are 64-blocks of 8-words each, since  $64 \times 8 = 512$ .
- The block number is specified with a 6-bit field and the word within the block is specified with 3-bit field. The tag field store with in the cache this common to all 8-words of the same block.



**Figure 12-14** Direct mapping cache with block size of 8 words.

- Every time a miss occurs, an entire block of 8-words must be transferred from main memory to cache memory. Although this takes extra time, the hit ratio will likely improve with a larger block size because the sequential nature of computer programs.

#### Set associative mapping:

- **Why:** The disadvantage of direct mapping is that two words with the same index in their address but with different tag values cannot reside in cache memory at the same time.
- The set-Associative mapping is an improvement over the direct mapping organization in that each word of cache can store two or more words of memory under the same index address.
- Each data word is stored together with its tag and the number of tag-data items in one word of cache is said to form **set**.

#### Example:

Each index address refers to two data words and their associated tags. Each tag requires 6 bits and each data word has 12 bits, so the word length is  $2(6+12) = 36$  bits.

An index address of 9 bits can accommodate 512 words. Thus the size of cache memory is 512x36. It can accommodate 1024 words of main memory since each word of cache contains two data words.

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

**Figure 12-15** Two-way set-associative mapping cache.

- The words stored at addresses 01000 and 02000 of main memory at index addresses 000. Similarly, the words at addresses 02777 and 00777 are stored in cache at index address 777.
- When the CPU generates a memory request, the index value of the address is used to access the cache. The tag field of the CPU address is then compared with both tags in the cache to determine if a match occurs.
- The comparison logic is done by an associative search of the tags in the set similar to an associative memory search. Thus the name “**set-associative**”.
- The hit ratio will improve as the set size increases because more words with the same index but different tags can reside in cache.
- An increase in the set size increases the number of bits in words of cache and requires more complex comparison logic.
- When a miss occurs in a set-associative cache and the set is full, it is necessary to replace one of the tag-data items with a new value. The most common Replacement Algorithms (FIFO, LRU) are used.
- The FIFO procedure selects for replacement the item that has been in the set the longest.

- The LRU (Least recently used) selects for replacement the item that has been least recently used by the CPU. Both FIFO and LRU can be implemented by adding a few extra bits in each word of cache.

### **Writing into cache:**

When the CPU finds a word in cache during a read operation, the main memory is not involved in the transfer. However, if the operation is a write, there are two ways that the system can proceed.

### **Write –Through method:**

The simplest and most commonly used procedure is to update main memory with every memory write operation, with the cache memory being updated in parallel if it contains the word at the specified address. This is called **write-through method**.

**Advantage:** Main memory always contains the same data as the cache. This characteristic is important in systems with direct memory access transfers.

**Write – back:** Only the cache location is updated during a write operation. The location is then marked by a flag so that later when the word is removed from the cache it is copied into main memory.

# UNIT V

**Reduced Instruction Set Computer:** CISC Characteristics, RISC Characteristics. Pipeline and Vector Processing: Parallel Processing, Pipelining, Arithmetic Pipeline, Instruction Pipeline, RISC Pipeline, Vector Processing, Array Processor.

**Multi Processors:** Characteristics of Multiprocessors, Interconnection Structures, Interprocessor arbitration, Interprocessor communication and synchronization, Cache Coherence.

## **Reduced Instruction Set Computer:**

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

## **CISC Characteristics:**

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

## **RISC Characteristics:**

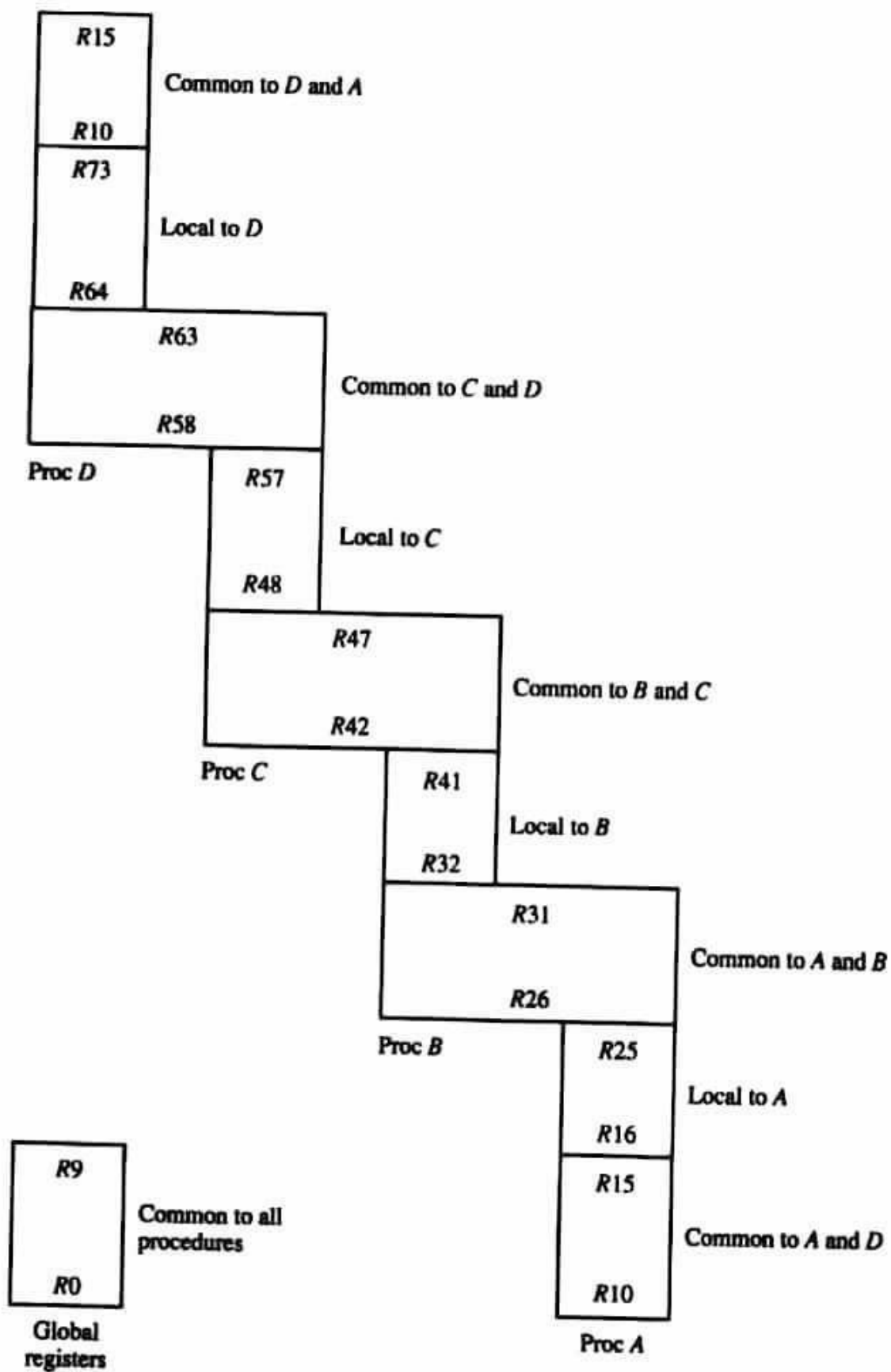
- ✓ Relatively few instructions
- ✓ Relatively few addressing modes
- ✓ Memory access limited to load and store instructions
- ✓ All operations done within the registers of the CPU
- ✓ Fixed-length, easily decoded instruction format
- ✓ Single-cycle instruction execution
- ✓ Hardwired rather than microprogrammed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

### **EFFICIENT PIPELINING:-**

- Efficient pipelining , as well as a few other characteristics, are sometimes attributed to RISC , although they may exist in non-RISC architecture as well. Other characteristics attributed to RISC architecture are :
  1. A relatively large number of registers in the processor unit.
  2. Use of overlapped register windows to speed-up procedure call and return.
  3. Efficient instruction pipeline.
  4. Compiler supports for efficient translation of high-level language programs into machine-level language programs.

### **OVERLAPPED REGISTER WINDOWS:-**

- Procedure call and return occurs quite in high-level programming languages.
- When translated into machine language, a procedure calls a subroutine to execute the body of the process.
- The concept of overlapped register windows is illustrated in figure 8-9. The system has a total of 74 registers, registers R0 through R9 are global registers that hold the parameters shared by all procedures.
- The other 64 registers are divided into 4 windows to accommodate procedures A, B, C and D.
- Each registers windows contains of 10 local registers and two sets of six registers common to adjacent windows.
- In general , the organization of the register windows will have the following relationships:
  - a. Number of global registers = G
  - b. Number of local registers in each window = L
  - c. Number of registers common to two windows = C
  - d. Number of windows = W



**Figure 8-9** Overlapped register windows.



- The number of registers available for each windows is calculated as follows :

$$\text{Windows size} = L + 2C + G$$

- The total number of registers needed in the processor are :

$$\text{Windows size} = (L + C)W + G$$

- let's observe an example

We have  $G = 10$ ,  $L = 10$ ,  $C = 6$  and  $W = 4$ .

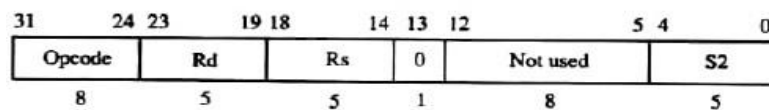
The window size is  $10 + 12 + 10 = 32$  registers.

The registers file consists of  $(10 + 6) \times 4 + 10 = 74$  registers.

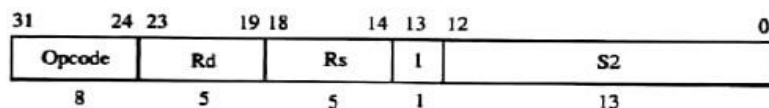
### **BERKELY RISC I:-**

- The Berkeley RISC I is a 32-bit integrated circuit CPU, it supports the 32-bit addresses and either 8-, 16- or 32-bit data.
- It has a 32-bit instruction format and a total of 31 instructions.

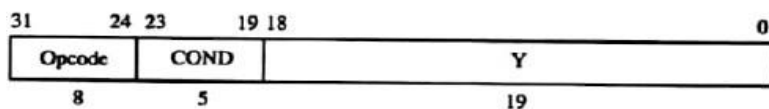
**Figure 8-10 Berkeley RISC I instruction formats.**



(a) Register mode: (S2 specifies a register)



(b) Register-immediate mode: (S2 specifies an operand)



(c) PC relative mode:

- There are 3 basic addressing modes, they are :-
  1. Register address
  2. Immediate operand
  3. Relative to the PC addressing for branch instructions.

- The 31 instructions of RSIC I are listed in below table. They have been grouped into 3 categories.
- Data manipulation instructions perform the arithmetic, logic and shift operations.
- The symbols under this opcode and operands columns are used when writing assembly language program.
- Note that all instructions have 3 operands by the number sign #.
- Consider, for example, the ADD instruction and how it can be used to perform a variety of operations.

---

ADD R22, R21, R23	$R23 \leftarrow R22 + R21$
ADD R22, #150, R23	$R23 \leftarrow R22 + 150$
ADD R0, R21, R22	$R22 \leftarrow R21$ (Move)
ADD R0, #150, R22	$R22 \leftarrow 150$ (Load immediate)
ADD R22, #1, R22	$R22 \leftarrow R22 + 1$ (Increment)

- By using register R0, which always contains 0's, it is possible to transfer the contents of one register or a constant into another register.
- The increment operation is accomplished by adding a constant 1 to the register.
- The data instructions consist of six load instructions, three store instructions and two instructions that transfer the program status word PSW.
- The register that holds PSW contains the status of the CPU and includes the program counter, the status bits from the ALU, pointers are used in conjunction with the register windows and the other information that determines the state of CPU.

TABLE 8-12 Instruction Set of Berkeley RISC I

Opcode	Operands	Register Transfer	Description
<b>Data manipulation instructions</b>			
ADD	$R_s, S2, R_d$	$R_d \leftarrow R_s + S2$	Integer add
ADDC	$R_s, S2, R_d$	$R_d \leftarrow R_s + S2 + \text{carry}$	Add with carry
SUB	$R_s, S2, R_d$	$R_d \leftarrow R_s - S2$	Integer subtract
SUBC	$R_s, S2, R_d$	$R_d \leftarrow R_s - S2 - \text{carry}$	Subtract with carry
SUBR	$R_s, S2, R_d$	$R_d \leftarrow S2 - R_s$	Subtract reverse
SUBCR	$R_s, S2, R_d$	$R_d \leftarrow S2 - R_s - \text{carry}$	Subtract with carry
AND	$R_s, S2, R_d$	$R_d \leftarrow R_s \wedge S2$	AND
OR	$R_s, S2, R_d$	$R_d \leftarrow R_s \vee S2$	OR
XOR	$R_s, S2, R_d$	$R_d \leftarrow R_s \oplus S2$	Exclusive-OR
SLL	$R_s, S2, R_d$	$R_d \leftarrow R_s \text{ shifted by } S2$	Shift-left
SRL	$R_s, S2, R_d$	$R_d \leftarrow R_s \text{ shifted by } S2$	Shift-right logical
SRA	$R_s, S2, R_d$	$R_d \leftarrow R_s \text{ shifted by } S2$	Shift-right arithmetic
<b>Data transfer instructions</b>			
LDL	$(R_s)S2, R_d$	$R_d \leftarrow M[R_s + S2]$	Load long
LDSU	$(R_s)S2, R_d$	$R_d \leftarrow M[R_s + S2]$	Load short unsigned
LDSS	$(R_s)S2, R_d$	$R_d \leftarrow M[R_s + S2]$	Load short signed
LDBU	$(R_s)S2, R_d$	$R_d \leftarrow M[R_s + S2]$	Load byte unsigned
LDBS	$(R_s)S2, R_d$	$R_d \leftarrow M[R_s + S2]$	Load byte signed
LDHI	$R_d, Y$	$R_d \leftarrow Y$	Load immediate high
STL	$R_d, (R_s)S2$	$M[R_s + S2] \leftarrow R_d$	Store long
STS	$R_d, (R_s)S2$	$M[R_s + S2] \leftarrow R_d$	Store short
STB	$R_d, (R_s)S2$	$M[R_s + S2] \leftarrow R_d$	Store byte
GETPSW	$R_d$	$R_d \leftarrow PSW$	Load status word
PUTPSW	$R_d$	$PSW \leftarrow R_d$	Set status word
<b>Program control instructions</b>			
JMP	COND, $S2(R_s)$	$PC \leftarrow R_s + S2$	Conditional jump
JMPR	COND, $Y$	$PC \leftarrow PC + Y$	Jump relative
CALL	$R_d, S2(R_s)$	$R_d \leftarrow PC$ $PC \leftarrow R_s + S2$ $CWP \leftarrow CWP - 1$	Call subroutine and change window
CALLR	$R_d, Y$	$R_d \leftarrow PC$ $PC \leftarrow PC + Y$ $CWP \leftarrow CWP - 1$	Call relative and change window
RET	$R_d, S2$	$PC \leftarrow R_d + S2$ $CWP \leftarrow CWP + 1$	Return and change window
CALLINT	$R_d$	$R_d \leftarrow PC$ $CWP \leftarrow CWP - 1$	Disable interrupts
RETINT	$R_d, S2$	$PC \leftarrow R_d + S2$ $CWP \leftarrow CWP + 1$	Enable interrupts
GTLPC	$R_d$	$R_d \leftarrow PC$	Get last PC



Scanned with  
CamScanner

- The load and store instructions move data between a register and memory.
- The load instructions accommodate the signed and unsigned data of 8bits or 16 bits.

- The long word instructions operate on the 32-bit data , although that data appears to be register plus displacement addressing mode in data transferring instructions.
- The following are examples

<b>LDL (R22)#150,R5</b>	<b><math>RS \leftarrow M[R22] + 150</math></b>
<b>LDL (R22)#0,R5</b>	<b><math>RS \leftarrow M[R22]</math></b>
<b>LDL (R0)#500,R5</b>	<b><math>RS \leftarrow M[500]</math></b>

- The effective address in the first is evaluated from the contents of the registers R22 , plus a displacement of 150.
- The second instruction uses a 0 displacement , which reduces it to a register indirect mode.

## CHAPTER NINE

# Pipeline and Vector Processing

### IN THIS CHAPTER

- 9.1 Parallel Processing
- 9.2 Pipelining
- 9.3 Arithmetic Pipeline
- 9.4 Instruction Pipeline
- 9.5 RISC Pipeline
- 9.6 Vector Processing
- 9.7 Array Processors

## 9-1 Parallel Processing

Parallel processing is a term used to denote a large class of techniques that are used to provide simultaneous data-processing tasks for the purpose of increasing the computational speed of a computer system. Instead of processing each instruction sequentially as in a conventional computer, a parallel processing system is able to perform concurrent data processing to achieve faster execution time. For example, while an instruction is being executed in the ALU, the next instruction can be read from memory. The system may have two or more ALUs and be able to execute two or more instructions at the same time. Furthermore, the system may have two or more processors operating concurrently. The purpose of parallel processing is to speed up the computer processing capability and increase its throughput, that is, the amount of processing that can be accomplished during a given interval of time. The amount of hardware increases with parallel processing, and with it, the cost of the system increases. However, technological developments have reduced hardware costs to the point where parallel processing techniques are economically feasible.

Parallel processing can be viewed from various levels of complexity. At the lowest level, we distinguish between parallel and serial operations by the type of registers used. Shift registers operate in serial fashion one bit at a time,

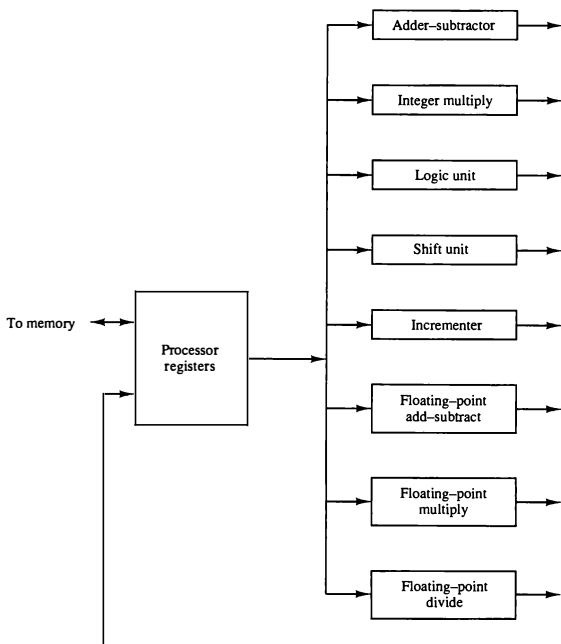
*throughput*

while registers with parallel load operate with all the bits of the word simultaneously. Parallel processing at a higher level of complexity can be achieved by having a multiplicity of functional units that perform identical or different operations simultaneously. Parallel processing is established by distributing the data among the multiple functional units. For example, the arithmetic, logic, and shift operations can be separated into three units and the operands diverted to each unit under the supervision of a control unit.

**multiple functional units**

Figure 9-1 shows one possible way of separating the execution unit into eight functional units operating in parallel. The operands in the registers are applied to one of the units depending on the operation specified by the instruc-

Figure 9-1 Processor with multiple functional units.



tion associated with the operands. The operation performed in each functional unit is indicated in each block of the diagram. The adder and integer multiplier perform the arithmetic operations with integer numbers. The floating-point operations are separated into three circuits operating in parallel. The logic, shift, and increment operations can be performed concurrently on different data. All units are independent of each other, so one number can be shifted while another number is being incremented. A multifunctional organization is usually associated with a complex control unit to coordinate all the activities among the various components.

There are a variety of ways that parallel processing can be classified. It can be considered from the internal organization of the processors, from the interconnection structure between processors, or from the flow of information through the system. One classification introduced by M. J. Flynn considers the organization of a computer system by the number of instructions and data items that are manipulated simultaneously. The normal operation of a computer is to fetch instructions from memory and execute them in the processor. The sequence of instructions read from memory constitutes an *instruction stream*. The operations performed on the data in the processor constitutes a *data stream*. Parallel processing may occur in the instruction stream, in the data stream, or in both. Flynn's classification divides computers into four major groups as follows:

Single instruction stream, single data stream (SISD)

Single instruction stream, multiple data stream (SIMD)

Multiple instruction stream, single data stream (MISD)

Multiple instruction stream, multiple data stream (MIMD)

SISD represents the organization of a single computer containing a control unit, a processor unit, and a memory unit. Instructions are executed sequentially and the system may or may not have internal parallel processing capabilities. Parallel processing in this case may be achieved by means of multiple functional units or by pipeline processing.

*SIMD*

SIMD represents an organization that includes many processing units under the supervision of a common control unit. All processors receive the same instruction from the control unit but operate on different items of data. The shared memory unit must contain multiple modules so that it can communicate with all the processors simultaneously. MISD structure is only of theoretical interest since no practical system has been constructed using this organization. MIMD organization refers to a computer system capable of processing several programs at the same time. Most multiprocessor and multi-computer systems can be classified in this category.

*MIMD*

Flynn's classification depends on the distinction between the performance of the control unit and the data-processing unit. It emphasizes the be-

havioral characteristics of the computer system rather than its operational and structural interconnections. One type of parallel processing that does not fit Flynn's classification is pipelining. The only two categories used from this classification are SIMD array processors discussed in Sec. 9-7, and MIMD multiprocessors presented in Chap. 13.

In this chapter we consider parallel processing under the following main topics:

1. Pipeline processing
2. Vector processing
3. Array processors

Pipeline processing is an implementation technique where arithmetic suboperations or the phases of a computer instruction cycle overlap in execution. Vector processing deals with computations involving large vectors and matrices. Array processors perform computations on large arrays of data.

## 9-2 Pipelining

Pipelining is a technique of decomposing a sequential process into suboperations, with each subprocess being executed in a special dedicated segment that operates concurrently with all other segments. A pipeline can be visualized as a collection of processing segments through which binary information flows. Each segment performs partial processing dictated by the way the task is partitioned. The result obtained from the computation in each segment is transferred to the next segment in the pipeline. The final result is obtained after the data have passed through all segments. The name "pipeline" implies a flow of information analogous to an industrial assembly line. It is characteristic of pipelines that several computations can be in progress in distinct segments at the same time. The overlapping of computation is made possible by associating a register with each segment in the pipeline. The registers provide isolation between each segment so that each can operate on distinct data simultaneously.

Perhaps the simplest way of viewing the pipeline structure is to imagine that each segment consists of an input register followed by a combinational circuit. The register holds the data and the combinational circuit performs the suboperation in the particular segment. The output of the combinational circuit in a given segment is applied to the input register of the next segment. A clock is applied to all registers after enough time has elapsed to perform all segment activity. In this way the information flows through the pipeline one step at a time.

*an example*

The pipeline organization will be demonstrated by means of a simple



example. Suppose that we want to perform the combined multiply and add operations with a stream of numbers.

$$A_i * B_i + C_i \quad \text{for } i = 1, 2, 3, \dots, 7$$

Each suboperation is to be implemented in a segment within a pipeline. Each segment has one or two registers and a combinational circuit as shown in Fig. 9-2. R1 through R5 are registers that receive new data with every clock pulse. The multiplier and adder are combinational circuits. The suboperations performed in each segment of the pipeline are as follows:

$R1 \leftarrow A_i$	$R2 \leftarrow B_i$	Input $A_i$ and $B_i$
$R3 \leftarrow R1 * R2$	$R4 \leftarrow C_i$	Multiply and input $C_i$
$R5 \leftarrow R3 + R4$		Add $C_i$ to product

The five registers are loaded with new data every clock pulse. The effect of each clock is shown in Table 9-1. The first clock pulse transfers  $A_1$  and  $B_1$  into R1 and

Figure 9-2 Example of pipeline processing.

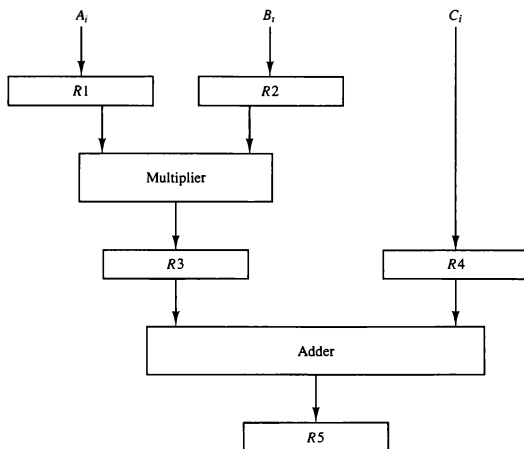


TABLE 9-1 Content of Registers in Pipeline Example

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	$A_1$	$B_1$	—	—	—
2	$A_2$	$B_2$	$A_1 * B_1$	$C_1$	—
3	$A_3$	$B_3$	$A_2 * B_2$	$C_2$	$A_1 * B_1 + C_1$
4	$A_4$	$B_4$	$A_3 * B_3$	$C_3$	$A_2 * B_2 + C_2$
5	$A_5$	$B_5$	$A_4 * B_4$	$C_4$	$A_3 * B_3 + C_3$
6	$A_6$	$B_6$	$A_5 * B_5$	$C_5$	$A_4 * B_4 + C_4$
7	$A_7$	$B_7$	$A_6 * B_6$	$C_6$	$A_5 * B_5 + C_5$
8	—	—	$A_7 * B_7$	$C_7$	$A_6 * B_6 + C_6$
9	—	—	—	—	$A_7 * B_7 + C_7$

R2. The second clock pulse transfers the product of R1 and R2 into R3 and  $C_1$  into R4. The same clock pulse transfers  $A_2$  and  $B_2$  into R1 and R2. The third clock pulse operates on all three segments simultaneously. It places  $A_3$  and  $B_3$  into R1 and R2, transfers the product of R1 and R2 into R3, transfers  $C_2$  into R4, and places the sum of R3 and R4 into R5. It takes three clock pulses to fill up the pipe and retrieve the first output from R5. From there on, each clock produces a new output and moves the data one step down the pipeline. This happens as long as new input data flow into the system. When no more input data are available, the clock must continue until the last output emerges out of the pipeline.

### General Considerations

Any operation that can be decomposed into a sequence of suboperations of about the same complexity can be implemented by a pipeline processor. The technique is efficient for those applications that need to repeat the same task many times with different sets of data. The general structure of a four-segment pipeline is illustrated in Fig. 9-3. The operands pass through all four segments in a fixed sequence. Each segment consists of a combinational circuit  $S_i$  that performs a suboperation over the data stream flowing through the pipe. The segments are separated by registers  $R_i$  that hold the intermediate results between the stages. Information flows between adjacent stages under the control of a common clock applied to all the registers simultaneously. We define a *task* as the total operation performed going through all the segments in the pipeline.

*task*

*space-time diagram*

The behavior of a pipeline can be illustrated with a *space-time* diagram. This is a diagram that shows the segment utilization as a function of time. The space-time diagram of a four-segment pipeline is demonstrated in Fig. 9-4. The horizontal axis displays the time in clock cycles and the vertical axis gives the

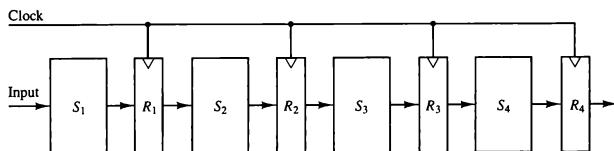


Figure 9-3 Four-segment pipeline.

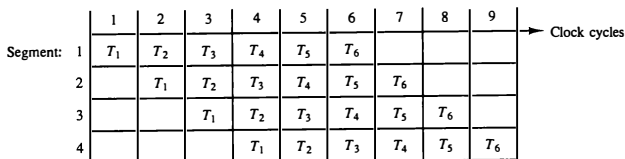
segment number. The diagram shows six tasks  $T_1$  through  $T_6$  executed in four segments. Initially, task  $T_1$  is handled by segment 1. After the first clock, segment 2 is busy with  $T_1$ , while segment 1 is busy with task  $T_2$ . Continuing in this manner, the first task  $T_1$  is completed after the fourth clock cycle. From then on, the pipe completes a task every clock cycle. No matter how many segments there are in the system, once the pipeline is full, it takes only one clock period to obtain an output.

Now consider the case where a  $k$ -segment pipeline with a clock cycle time  $t_p$  is used to execute  $n$  tasks. The first task  $T_1$  requires a time equal to  $kt_p$  to complete its operation since there are  $k$  segments in the pipe. The remaining  $n - 1$  tasks emerge from the pipe at the rate of one task per clock cycle and they will be completed after a time equal to  $(n - 1)t_p$ . Therefore, to complete  $n$  tasks using a  $k$ -segment pipeline requires  $k + (n - 1)$  clock cycles. For example, the diagram of Fig. 9-4 shows four segments and six tasks. The time required to complete all the operations is  $4 + (6 - 1) = 9$  clock cycles, as indicated in the diagram.

Next consider a nonpipeline unit that performs the same operation and takes a time equal to  $t_n$  to complete each task. The total time required for  $n$  tasks is  $nt_n$ . The speedup of a pipeline processing over an equivalent nonpipeline processing is defined by the ratio

$$S = \frac{nt_n}{(k + n - 1)t_p}$$

Figure 9-4 Space-time diagram for pipeline.



speedup

As the number of tasks increases,  $n$  becomes much larger than  $k - 1$ , and  $k + n - 1$  approaches the value of  $n$ . Under this condition, the speedup becomes

$$S = \frac{t_n}{t_p}$$

If we assume that the time it takes to process a task is the same in the pipeline and nonpipeline circuits, we will have  $t_n = kt_p$ . Including this assumption, the speedup reduces to

$$S = \frac{kt_p}{t_p} = k$$

This shows that the theoretical maximum speedup that a pipeline can provide is  $k$ , where  $k$  is the number of segments in the pipeline.

To clarify the meaning of the speedup ratio, consider the following numerical example. Let the time it takes to process a suboperation in each segment be equal to  $t_p = 20$  ns. Assume that the pipeline has  $k = 4$  segments and executes  $n = 100$  tasks in sequence. The pipeline system will take  $(k + n - 1)t_p = (4 + 99) \times 20 = 2060$  ns to complete. Assuming that  $t_n = kt_p = 4 \times 20 = 80$  ns, a nonpipeline system requires  $nt_p = 100 \times 80 = 8000$  ns to complete the 100 tasks. The speedup ratio is equal to  $8000/2060 = 3.88$ . As the number of tasks increases, the speedup will approach 4, which is equal to the number of segments in the pipeline. If we assume that  $t_n = 60$  ns, the speedup becomes  $60/20 = 3$ .

To duplicate the theoretical speed advantage of a pipeline process by means of multiple functional units, it is necessary to construct  $k$  identical units that will be operating in parallel. The implication is that a  $k$ -segment pipeline processor can be expected to equal the performance of  $k$  copies of an equivalent nonpipeline circuit under equal operating conditions. This is illustrated in Fig. 9-5, where four identical circuits are connected in parallel. Each  $P$  circuit performs the same task of an equivalent pipeline circuit. Instead of operating with the input data in sequence as in a pipeline, the parallel circuits accept four input data items simultaneously and perform four tasks at the same time. As far as the speed of operation is concerned, this is equivalent to a four segment pipeline. Note that the four-unit circuit of Fig. 9-5 constitutes a single-instruction multiple-data (SIMD) organization since the same instruction is used to operate on multiple data in parallel.

There are various reasons why the pipeline cannot operate at its maximum theoretical rate. Different segments may take different times to complete their suboperation. The clock cycle must be chosen to equal the time delay of the segment with the maximum propagation time. This causes all other segments to waste time while waiting for the next clock. Moreover, it is not always

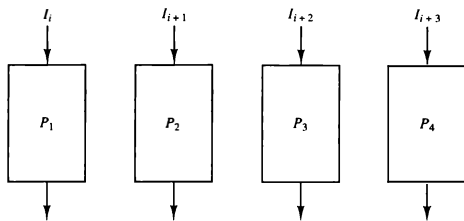


Figure 9-5 Multiple functional units in parallel.

correct to assume that a nonpipe circuit has the same time delay as that of an equivalent pipeline circuit. Many of the intermediate registers will not be needed in a single-unit circuit, which can usually be constructed entirely as a combinational circuit. Nevertheless, the pipeline technique provides a faster operation over a purely serial sequence even though the maximum theoretical speed is never fully achieved.

There are two areas of computer design where the pipeline organization is applicable. An *arithmetic pipeline* divides an arithmetic operation into suboperations for execution in the pipeline segments. An *instruction pipeline* operates on a stream of instructions by overlapping the fetch, decode, and execute phases of the instruction cycle. The two types of pipelines are explained in the following sections.

## 9-3 Arithmetic Pipeline

Pipeline arithmetic units are usually found in very high speed computers. They are used to implement floating-point operations, multiplication of fixed-point numbers, and similar computations encountered in scientific problems. A pipeline multiplier is essentially an array multiplier as described in Fig. 10-10, with special adders designed to minimize the carry propagation time through the partial products. Floating-point operations are easily decomposed into suboperations as demonstrated in Sec. 10-5. We will now show an example of a pipeline unit for floating-point addition and subtraction.

The inputs to the floating-point adder pipeline are two normalized floating-point binary numbers.

$$X = A \times 2^a$$

$$Y = B \times 2^b$$

$A$  and  $B$  are two fractions that represent the mantissas and  $a$  and  $b$  are the exponents. The floating-point addition and subtraction can be performed in four segments, as shown in Fig. 9-6. The registers labeled  $R$  are placed between the segments to store intermediate results. The suboperations that are performed in the four segments are:

1. Compare the exponents.
2. Align the mantissas.
3. Add or subtract the mantissas.
4. Normalize the result.

This follows the procedure outlined in the flowchart of Fig. 10-15 but with some variations that are used to reduce the execution time of the suboperations. The exponents are compared by subtracting them to determine their difference. The larger exponent is chosen as the exponent of the result. The exponent difference determines how many times the mantissa associated with the smaller exponent must be shifted to the right. This produces an alignment of the two mantissas. It should be noted that the shift must be designed as a combinational circuit to reduce the shift time. The two mantissas are added or subtracted in segment 3. The result is normalized in segment 4. When an overflow occurs, the mantissa of the sum or difference is shifted right and the exponent incremented by one. If an underflow occurs, the number of leading zeros in the mantissa determines the number of left shifts in the mantissa and the number that must be subtracted from the exponent.

The following numerical example may clarify the suboperations performed in each segment. For simplicity, we use decimal numbers, although Fig. 9-6 refers to binary numbers. Consider the two normalized floating-point numbers:

$$X = 0.9504 \times 10^3$$

$$Y = 0.8200 \times 10^2$$

The two exponents are subtracted in the first segment to obtain  $3 - 2 = 1$ . The larger exponent 3 is chosen as the exponent of the result. The next segment shifts the mantissa of  $Y$  to the right to obtain

$$X = 0.9504 \times 10^3$$

$$Y = 0.0820 \times 10^3$$

This aligns the two mantissas under the same exponent. The addition of the two mantissas in segment 3 produces the sum

$$Z = 1.0324 \times 10^3$$

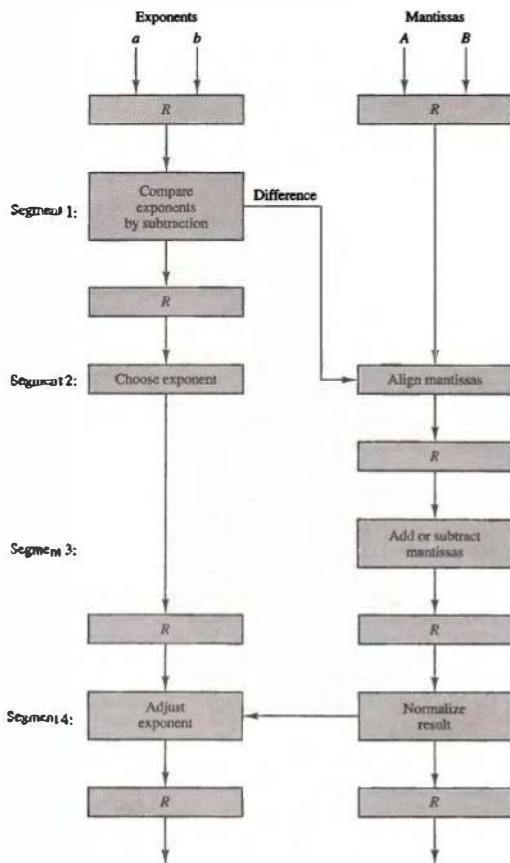


Figure 9-6 Pipeline for floating-point addition and subtraction.

The sum is adjusted by normalizing the result so that it has a fraction with a nonzero first digit. This is done by shifting the mantissa once to the right and incrementing the exponent by one to obtain the normalized sum.

$$Z = 0.10324 \times 10^4$$

The comparator, shifter, adder-subtractor, incrementer, and decrements in the floating-point pipeline are implemented with combinational circuits. Suppose that the timedelays of the four segments are  $t_1 = 60$  ns,  $t_2 = 70$  ns,  $t_3 = 100$  ns,  $t_4 = 80$  ns, and the interface registers have a delay of  $t_r = 10$  ns. The clock cycle is chosen to be  $t_p = t_3 + t_r = 110$  ns. An equivalent nonpipeline floating-point adder-subtractor will have a delay time  $t_n = t_1 + t_2 + t_3 + t_4 + t_r = 320$  ns. In this case the pipelined adder has a speedup of  $320/110 = 2.9$  over the nonpipelined adder.

## 9-4 Instruction Pipeline

Pipeline processing can occur not only in the data stream but in the instruction stream as well. An instruction pipeline reads consecutive instructions from memory while previous instructions are being executed in other segments. This causes the instruction fetch and execute phases to overlap and perform simultaneous operations. One possible digression associated with such a scheme is that an instruction may cause a branch out of sequence. In that case the pipeline must be emptied and all the instructions that have been read from memory after the branch instruction must be discarded.

Consider a computer with an instruction fetch unit and an instruction execution unit designed to provide a two-segment pipeline. The instruction fetch segment can be implemented by means of a first-in, first-out (FIFO) buffer. This is a type of unit that forms a queue rather than a stack. Whenever the execution unit is not using memory, the control increments the program counter and uses its address value to read consecutive instructions from memory. The instructions are inserted into the FIFO buffer so that they can be executed on a first-in, first-out basis. Thus an instruction stream can be placed in a queue, waiting for decoding and processing by the execution segment. The instruction stream queuing mechanism provides an efficient way for reducing the average access time to memory for reading instructions. Whenever there is space in the FIFO buffer, the control unit initiates the next instruction fetch phase. The buffer acts as a queue from which control then extracts the instructions for the execution unit.

Computers with complex instructions require other phases in addition to the fetch and execute to process an instruction completely. In the most general case, the computer needs to process each instruction with the following sequence of steps.

*instruction cycle*



1. Fetch the instruction from memory.
2. Decode the instruction.
3. Calculate the effective address.
4. Fetch the operands from memory.
5. Execute the instruction.
6. Store the result in the proper place.

There are certain difficulties that will prevent the instruction pipeline from operating at its maximum rate. Different segments may take different times to operate on the incoming information. Some segments are skipped for certain operations. For example, a register mode instruction does not need an effective address calculation. Two or more segments may require memory access at the same time, causing one segment to wait until another is finished with the memory. Memory access conflicts are sometimes resolved by using two memory buses for accessing instructions and data in separate modules. In this way, an instruction word and a data word can be read simultaneously from two different modules.

The design of an instruction pipeline will be most efficient if the instruction cycle is divided into segments of equal duration. The time that each step takes to fulfill its function depends on the instruction and the way it is executed.

### Example: Four-Segment Instruction Pipeline

Assume that the decoding of the instruction can be combined with the calculation of the effective address into one segment. Assume further that most of the instructions place the result into a processor register so that the instruction execution and storing of the result can be combined into one segment. This reduces the instruction pipeline into four segments.

Figure 9-7 shows how the instruction cycle in the CPU can be processed with a four-segment pipeline. While an instruction is being executed in segment 4, the next instruction in sequence is busy fetching an operand from memory in segment 3. The effective address may be calculated in a separate arithmetic circuit for the third instruction, and whenever the memory is available, the fourth and all subsequent instructions can be fetched and placed in an instruction FIFO. Thus up to four suboperations in the instruction cycle can overlap and up to four different instructions can be in progress of being processed at the same time.

Once in a while, an instruction in the sequence may be a program control type that causes a branch out of normal sequence. In that case the pending operations in the last two segments are completed and all information stored in the instruction buffer is deleted. The pipeline then restarts from the new address stored in the program counter. Similarly, an interrupt request, when acknowledged, will cause the pipeline to empty and start again from a new address value.

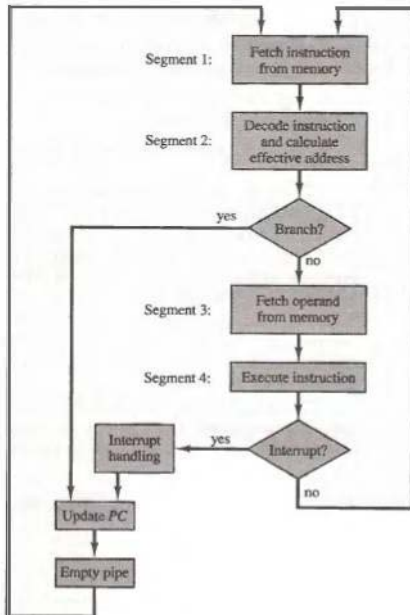


Figure 9-7 Four-segment CPU pipeline.

Figure 9-8 shows the operation of the instruction pipeline. The time in the horizontal axis is divided into steps of equal duration. The four segments are represented in the diagram with an abbreviated symbol.

1. FI is the segment that fetches an instruction.
2. DA is the segment that decodes the instruction and calculates the effective address.
3. FO is the segment that fetches the operand.
4. EX is the segment that executes the instruction.

It is assumed that the processor has separate instruction and data memories so that the operation in FI and FO can proceed at the same time. In the absence

Step:		1	2	3	4	5	6	7	8	9	10	11	12	13
Instruction:	1	FI	DA	FO	EX									
	2		FI	DA	FO	EX								
(Branch)	3			FI	DA	FO	EX							
	4				FI	-	-	FI	DA	FO	EX			
	5					-	-	-	FI	DA	FO	EX		
	6									FI	DA	FO	EX	
	7										FI	DA	FO	EX

Figure 9-8 Timing of instruction pipeline.

of a branch instruction, each segment operates on different instructions. Thus, in step 4, instruction 1 is being executed in segment EX; the operand for instruction 2 is being fetched in segment FO; instruction 3 is being decoded in segment DA; and instruction 4 is being fetched from memory in segment FI.

Assume now that instruction 3 is a branch instruction. As soon as this instruction is decoded in segment DA in step 4, the transfer from FI to DA of the other instructions is halted until the branch instruction is executed in step 6. If the branch is taken, a new instruction is fetched in step 7. If the branch is not taken, the instruction fetched previously in step 4 can be used. The pipeline then continues until a new branch instruction is encountered.

Another delay may occur in the pipeline if the EX segment needs to store the result of the operation in the data memory while the FO segment needs to fetch an operand. In that case, segment FO must wait until segment EX has finished its operation.

In general, there are three major difficulties that cause the instruction pipeline to deviate from its normal operation.

1. *Resource conflicts* caused by access to memory by two segments at the same time. Most of these conflicts can be resolved by using separate instruction and data memories.
2. *Data dependency* conflicts arise when an instruction depends on the result of a previous instruction, but this result is not yet available.
3. *Branch difficulties* arise from branch and other instructions that change the value of PC.

### Data Dependency

A difficulty that may cause a degradation of performance in an instruction pipeline is due to possible collision of data or address. A collision occurs when

an instruction cannot proceed because previous instructions did not complete certain operations. A data dependency occurs when an instruction needs data that are not yet available. For example, an instruction in the FO segment may need to fetch an operand that is being generated at the same time by the previous instruction in segment EX. Therefore, the second instruction must wait for data to become available by the first instruction. Similarly, an address dependency may occur when an operand address cannot be calculated because the information needed by the addressing mode is not available. For example, an instruction with register indirect mode cannot proceed to fetch the operand if the previous instruction is loading the address into the register. Therefore, the operand access to memory must be delayed until the required address is available. Pipelined computers deal with such conflicts between data dependencies in a variety of ways.

#### *hardware interlocks*

The most straightforward method is to insert *hardware interlocks*. An interlock is a circuit that detects instructions whose source operands are destinations of instructions farther up in the pipeline. Detection of this situation causes the instruction whose source is not available to be delayed by enough clock cycles to resolve the conflict. This approach maintains the program sequence by using hardware to insert the required delays.

#### *operand forwarding*

Another technique called *operand forwarding* uses special hardware to detect a conflict and then avoid it by routing the data through special paths between pipeline segments. For example, instead of transferring an ALU result into a destination register, the hardware checks the destination operand, and if it is needed as a source in the next instruction, it passes the result directly into the ALU input, bypassing the register file. This method requires additional hardware paths through multiplexers as well as the circuit that detects the conflict.

#### *delayed load*

A procedure employed in some computers is to give the responsibility for solving data conflicts problems to the compiler that translates the high-level programming language into a machine language program. The compiler for such computers is designed to detect a data conflict and reorder the instructions as necessary to delay the loading of the conflicting data by inserting no-operation instructions. This method is referred to as *delayed load*. An example of delayed load is presented in the next section.

## Handling of Branch Instructions

One of the major problems in operating an instruction pipeline is the occurrence of branch instructions. A branch instruction can be conditional or unconditional. An unconditional branch always alters the sequential program flow by loading the program counter with the target address. In a conditional branch, the control selects the target instruction if the condition is satisfied or the next sequential instruction if the condition is not satisfied. As mentioned previously, the branch instruction breaks the normal sequence of the instruction stream, causing difficulties in the operation of the instruction pipeline.

Pipelined computers employ various hardware techniques to minimize the performance degradation caused by instruction branching.

One way of handling a conditional branch is to prefetch the target instruction in addition to the instruction following the branch. Both are saved until the branch is executed. If the branch condition is successful, the pipeline continues from the branch target instruction. An extension of this procedure is to continue fetching instructions from both places until the branch decision is made. At that time control chooses the instruction stream of the correct program flow.

Another possibility is the use of a *branch target buffer* or BTB. The BTB is an associative memory (see Sec. 12-4) included in the fetch segment of the pipeline. Each entry in the BTB consists of the address of a previously executed branch instruction and the target instruction for that branch. It also stores the next few instructions after the branch target instruction. When the pipeline decodes a branch instruction, it searches the associative memory BTB for the address of the instruction. If it is in the BTB, the instruction is available directly and prefetch continues from the new path. If the instruction is not in the BTB, the pipeline shifts to a new instruction stream and stores the target instruction in the BTB. The advantage of this scheme is that branch instructions that have occurred previously are readily available in the pipeline without interruption.

A variation of the BTB is the *loop buffer*. This is a small very high speed register file maintained by the instruction fetch segment of the pipeline. When a program loop is detected in the program, it is stored in the loop buffer in its entirety, including all branches. The program loop can be executed directly without having to access memory until the loop mode is removed by the final branching out.

Another procedure that some computers use is *branch prediction*. A pipeline with branch prediction uses some additional logic to guess the outcome of a conditional branch instruction before it is executed. The pipeline then begins prefetching the instruction stream from the predicted path. A correct prediction eliminates the wasted time caused by branch penalties.

A procedure employed in most RISC processors is the *delayed branch*. In this procedure, the compiler detects the branch instructions and rearranges the machine language code sequence by inserting useful instructions that keep the pipeline operating without interruptions. An example of delayed branch is the insertion of a no-operation instruction after a branch instruction. This causes the computer to fetch the target instruction during the execution of the no-operation instruction, allowing a continuous flow of the pipeline. An example of delayed branch is presented in the next section.

## 9-5 RISC Pipeline

The reduced instruction set computer (RISC) was introduced in Sec. 8-8. Among the characteristics attributed to RISC is its ability to use an efficient instruction pipeline. The simplicity of the instruction set can be utilized to

implement an instruction pipeline using a small number of suboperations, with each being executed in one clock cycle. Because of the fixed-length instruction format, the decoding of the operation can occur at the same time as the register selection. All data manipulation instructions have register-to-register operations. Since all operands are in registers, there is no need for calculating an effective address or fetching of operands from memory. Therefore, the instruction pipeline can be implemented with two or three segments. One segment fetches the instruction from program memory, and the other segment executes the instruction in the ALU. A third segment may be used to store the result of the ALU operation in a destination register.

The data transfer instructions in RISC are limited to load and store instructions. These instructions use register indirect addressing. They usually need three or four stages in the pipeline. To prevent conflicts between a memory access to fetch an instruction and to load or store an operand, most RISC machines use two separate buses with two memories: one for storing the instructions and the other for storing the data. The two memories can sometimes operate at the same speed as the CPU clock and are referred to as cache memories (see Sec. 12-6).

As mentioned in Sec. 8-8, one of the major advantages of RISC is its ability to execute instructions at the rate of one per clock cycle. It is not possible to expect that every instruction be fetched from memory and executed in one clock cycle. What is done, in effect, is to start each instruction with each clock cycle and to pipeline the processor to achieve the goal of single-cycle instruction execution. The advantage of RISC over CISC (complex instruction set computer) is that RISC can achieve pipeline segments, requiring just one clock cycle, while CISC uses many segments in its pipeline, with the longest segment requiring two or more clock cycles.

Another characteristic of RISC is the support given by the compiler that translates the high-level language program into machine language program. Instead of designing hardware to handle the difficulties associated with data conflicts and branch penalties, RISC processors rely on the efficiency of the compiler to detect and minimize the delays encountered with these problems. The following examples show how a compiler can optimize the machine language program to compensate for pipeline conflicts.

### Example: Three-Segment Instruction Pipeline

A typical set of instructions for a RISC processor are listed in Table 8-12. We see from this table that there are three types of instructions. The data manipulation instructions operate on data in processor registers. The data transfer instructions are load and store instructions that use an effective address obtained from the addition of the contents of two registers or a register and a displacement constant provided in the instruction. The program control instructions use register values and a constant to evaluate the branch address, which is transferred to a register or the program counter PC.

*single-cycle  
instruction  
execution*

*compiler support*

Now consider the hardware operation for such a computer. The control section fetches the instruction from program memory into an instruction register. The instruction is decoded at the same time that the registers needed for the execution of the instruction are selected. The processor unit consists of a number of registers and an arithmetic logic unit (ALU) that performs the necessary arithmetic, logic, and shift operations. A data memory is used to load or store the data from a selected register in the register file. The instruction cycle can be divided into three suboperations and implemented in three segments:

- I: Instruction fetch
- A: ALU operation
- E: Execute instruction

The I segment fetches the instruction from program memory. The instruction is decoded and an ALU operation is performed in the A segment. The ALU is used for three different functions, depending on the decoded instruction. It performs an operation for a data manipulation instruction, it evaluates the effective address for a load or store instruction, or it calculates the branch address for a program control instruction. The E segment directs the output of the ALU to one of three destinations, depending on the decoded instruction. It transfers the result of the ALU operation into a destination register in the register file, it transfers the effective address to a data memory for loading or storing, or it transfers the branch address to the program counter.

### Delayed Load

Consider now the operation of the following four instructions:

1. LOAD:  $R1 \leftarrow M[\text{address } 1]$
2. LOAD:  $R2 \leftarrow M[\text{address } 2]$
3. ADD:  $R3 \leftarrow R1 + R2$
4. STORE:  $M[\text{address } 3] \leftarrow R3$

If the three-segment pipeline proceeds without interruptions, there will be a data conflict in instruction 3 because the operand in  $R2$  is not yet available in the A segment. This can be seen from the timing of the pipeline shown in Fig. 9-9(a). The E segment in clock cycle 4 is in a process of placing the memory data into  $R2$ . The A segment in clock cycle 4 is using the data from  $R2$ , but the value in  $R2$  will not be the correct value since it has not yet been transferred from memory. It is up to the compiler to make sure that the instruction following the load instruction uses the data fetched from memory. If the compiler cannot find a useful instruction to put after the load, it inserts a no-op (no-operation) instruction. This is a type of instruction that is fetched from

Clock cycles:	1	2	3	4	5	6
1. Load R1	I	A	E			
2. Load R2		I	A	E		
3. Add R1 + R2			I	A	E	
4. Store R3				I	A	E

(a) Pipeline timing with data conflict

Clock cycle:	1	2	3	4	5	6	7
1. Load R1	I	A	E				
2. Load R2		I	A	E			
3. No-operation			I	A	E		
4. Add R1 + R2				I	A	E	
5. Store R3					I	A	E

(b) Pipeline timing with delayed load

Figure 9-9 Three-segment pipeline timing.

memory but has no operation, thus wasting a clock cycle. This concept of delaying the use of the data loaded from memory is referred to as *delayed load*.

Figure 9-9(b) shows the same program with a no-op instruction inserted after the load to R2 instruction. The data is loaded into R2 in clock cycle 4. The add instruction uses the value of R2 in step 5. Thus the no-op instruction is used to advance one clock cycle in order to compensate for the data conflict in the pipeline. (Note that no operation is performed in segment A during clock cycle 4 or segment E during clock cycle 5.) The advantage of the delayed load approach is that the data dependency is taken care of by the compiler rather than the hardware. This results in a simpler hardware segment since the segment does not have to check if the content of the register being accessed is currently valid or not.

### Delayed Branch

It was shown in Fig. 9-8 that a branch instruction delays the pipeline operation until the instruction at the branch address is fetched. Several techniques for reducing branch penalties were discussed in the preceding section. The method used in most RISC processors is to rely on the compiler to redefine the branches so that they take effect at the proper time in the pipeline. This method is referred to as *delayed branch*.



The compiler for a processor that uses delayed branches is designed to analyze the instructions before and after the branch and rearrange the program sequence by inserting useful instructions in the delay steps. For example, the compiler can determine that the program dependencies allow one or more instructions preceding the branch to be moved into the delay steps after the branch. These instructions are then fetched from memory and executed through the pipeline while the branch instruction is being executed in other segments. The effect is the same as if the instructions were executed in their original order, except that the branch delay is removed. It is up to the compiler to find useful instructions to put after the branch instruction. Failing that, the compiler can insert no-op instructions.

An example of delayed branch is shown in Fig. 9-10. The program for this example consists of five instructions:

```
Load from memory to R1  
Increment R2  
Add R3 to R4  
Subtract R5 from R6  
Branch to address X
```

In Fig. 9-10(a) the compiler inserts two no-op instructions after the branch. The branch address *X* is transferred to *PC* in clock cycle 7. The fetching of the instruction at *X* is delayed by two clock cycles by the no-op instructions. The instruction at *X* starts the fetch phase at clock cycle 8 after the program counter *PC* has been updated.

The program in Fig. 9-10(b) is rearranged by placing the add and subtract instructions after the branch instruction instead of before as in the original program. Inspection of the pipeline timing shows that *PC* is updated to the value of *X* in clock cycle 5, but the add and subtract instructions are fetched from memory and executed in the proper sequence. In other words, if the load instruction is at address 101 and *X* is equal to 350, the branch instruction is fetched from address 103. The add instruction is fetched from address 104 and executed in clock cycle 6. The subtract instruction is fetched from address 105 and executed in clock cycle 7. Since the value of *X* is transferred to *PC* with clock cycle 5 in the E segment, the instruction fetched from memory at clock cycle 6 is from address 350, which is the instruction at the branch address.

## 9-6 Vector Processing

---

There is a class of computational problems that are beyond the capabilities of a conventional computer. These problems are characterized by the fact that they require a vast number of computations that will take a conventional computer days or even weeks to complete. In many science and engineering

Clock cycles:	1	2	3	4	5	6	7	8	9	10
1. Load	I	A	E							
2. Increment		I	A	E						
3. Add			I	A	E					
4. Subtract				I	A	E				
5. Branch to X					I	A	E			
6. No-operation						I	A	E		
7. No-operation							I	A	E	
8. Instruction in X								I	A	E

(a) Using no-operation instructions

Clock cycles:	1	2	3	4	5	6	7	8
1. Load	I	A	E					
2. Increment		I	A	E				
3. Branch to X			I	A	E			
4. Add				I	A	E		
5. Subtract					I	A	E	
6. Instruction in X						I	A	E

(b) Rearranging the instructions

Figure 9-10 Example of delayed branch.

applications, the problems can be formulated in terms of vectors and matrices that lend themselves to vector processing.

#### *applications*

Computers with vector processing capabilities are in demand in specialized applications. The following are representative application areas where vector processing is of the utmost importance.

Long-range weather forecasting

Petroleum explorations

Seismic data analysis

Medical diagnosis

Aerodynamics and space flight simulations

Artificial intelligence and expert systems  
 Mapping the human genome  
 Image processing

Without sophisticated computers, many of the required computations cannot be completed within a reasonable amount of time. To achieve the required level of high performance it is necessary to utilize the fastest and most reliable hardware and apply innovative procedures from vector and parallel processing techniques.

### Vector Operations

Many scientific problems require arithmetic operations on large arrays of numbers. These numbers are usually formulated as vectors and matrices of floating-point numbers. A vector is an ordered set of a one-dimensional array of data items. A vector  $V$  of length  $n$  is represented as a row vector by  $V = [V_1 \ V_2 \ V_3 \ \cdots \ V_n]$ . It may be represented as a column vector if the data items are listed in a column. A conventional sequential computer is capable of processing operands one at a time. Consequently, operations on vectors must be broken down into single computations with subscripted variables. The element  $V_i$  of vector  $V$  is written as  $V(I)$  and the index  $I$  refers to a memory address or register where the number is stored. To examine the difference between a conventional scalar processor and a vector processor, consider the following Fortran DO loop:

```
DO 20 I = 1, 100
  C(I) = B(I) + A(I)
```

This is a program for adding two vectors  $A$  and  $B$  of length 100 to produce a vector  $C$ . This is implemented in machine language by the following sequence of operations.

```
Initialize I = 0
20 Read A(I)
  Read B(I)
  Store C(I) = A(I) + B(I)
  Increment I = I + 1
  If I ≤ 100 go to 20
Continue
```

This constitutes a program loop that reads a pair of operands from arrays  $A$  and  $B$  and performs a floating-point addition. The loop control variable is then updated and the steps repeat 100 times.

A computer capable of vector processing eliminates the overhead associated with the time it takes to fetch and execute the instructions in the program

loop. It allows operations to be specified with a single vector instruction of the form

$$C(1:100) = A(1:100) + B(1:100)$$

The vector instruction includes the initial address of the operands, the length of the vectors, and the operation to be performed, all in one composite instruction. The addition is done with a pipelined floating-point adder similar to the one shown in Fig. 9-6.

A possible instruction format for a vector instruction is shown in Fig. 9-11. This is essentially a three-address instruction with three fields specifying the base address of the operands and an additional field that gives the length of the data items in the vectors. This assumes that the vector operands reside in memory. It is also possible to design the processor with a large number of registers and store all operands in registers prior to the addition operation. In that case the base address and length in the vector instruction specify a group of CPU registers.

### Matrix Multiplication

Matrix multiplication is one of the most computational intensive operations performed in computers with vector processors. The multiplication of two  $n \times n$  matrices consists of  $n^2$  inner products or  $n^3$  multiply-add operations. An  $n \times m$  matrix of numbers has  $n$  rows and  $m$  columns and may be considered as constituting a set of  $n$  row vectors or a set of  $m$  column vectors. Consider, for example, the multiplication of two  $3 \times 3$  matrices  $A$  and  $B$ .

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

The product matrix  $C$  is a  $3 \times 3$  matrix whose elements are related to the elements of  $A$  and  $B$  by the inner product:

$$c_{ij} = \sum_{k=1}^3 a_{ik} \times b_{kj}$$

For example, the number in the first row and first column of matrix  $C$  is calculated by letting  $i = 1, j = 1$ , to obtain

$$c_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31}$$

Figure 9-11 Instruction format for vector processor.

Operation code	Base address source 1	Base address source 2	Base address destination	Vector length
----------------	-----------------------	-----------------------	--------------------------	---------------

This requires three multiplications and (after initializing  $c_{11}$  to 0) three additions. The total number of multiplications or additions required to compute the matrix product is  $9 \times 3 = 27$ . If we consider the linked multiply-add operation  $c + a \times b$  as a cumulative operation, the product of two  $n \times n$  matrices requires  $n^3$  multiply-add operations. The computation consists of  $n^2$  inner products, with each inner product requiring  $n$  multiply-add operations, assuming that  $c$  is initialized to zero before computing each element in the product matrix.

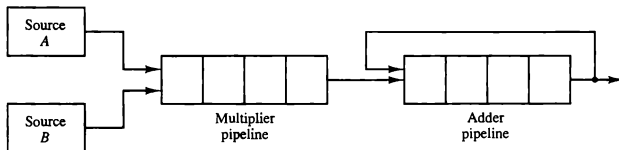
In general, the inner product consists of the sum of  $k$  product terms of the form

$$C = A_1 B_1 + A_2 B_2 + A_3 B_3 + A_4 B_4 + \cdots + A_k B_k$$

In a typical application  $k$  may be equal to 100 or even 1000. The inner product calculation on a pipeline vector processor is shown in Fig. 9-12. The values of  $A$  and  $B$  are either in memory or in processor registers. The floating-point multiplier pipeline and the floating-point adder pipeline are assumed to have four segments each. All segment registers in the multiplier and adder are initialized to 0. Therefore, the output of the adder is 0 for the first eight cycles until both pipes are full.  $A_i$  and  $B_i$  pairs are brought in and multiplied at a rate of one pair per cycle. After the first four cycles, the products begin to be added to the output of the adder. During the next four cycles 0 is added to the products entering the adder pipeline. At the end of the eighth cycle, the first four products  $A_1 B_1$  through  $A_4 B_4$  are in the four adder segments, and the next four products,  $A_5 B_5$  through  $A_8 B_8$ , are in the multiplier segments. At the beginning of the ninth cycle, the output of the adder is  $A_1 B_1$  and the output of the multiplier is  $A_5 B_5$ . Thus the ninth cycle starts the addition  $A_1 B_1 + A_5 B_5$  in the adder pipeline. The tenth cycle starts the addition  $A_2 B_2 + A_6 B_6$ , and so on. This pattern breaks down the summation into four sections as follows:

$$\begin{aligned} C = & A_1 B_1 + A_5 B_5 + A_9 B_9 + A_{13} B_{13} + \cdots \\ & + A_2 B_2 + A_6 B_6 + A_{10} B_{10} + A_{14} B_{14} + \cdots \\ & + A_3 B_3 + A_7 B_7 + A_{11} B_{11} + A_{15} B_{15} + \cdots \\ & + A_4 B_4 + A_8 B_8 + A_{12} B_{12} + A_{16} B_{16} + \cdots \end{aligned}$$

Figure 9-12 Pipeline for calculating an inner product.

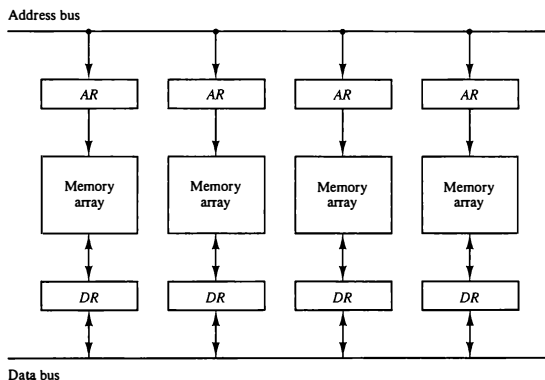


When there are no more product terms to be added, the system inserts four zeros into the multiplier pipeline. The adder pipeline will then have one partial product in each of its four segments, corresponding to the four sums listed in the four rows in the above equation. The four partial sums are then added to form the final sum.

### Memory Interleaving

Pipeline and vector processors often require simultaneous access to memory from two or more sources. An instruction pipeline may require the fetching of an instruction and an operand at the same time from two different segments. Similarly, an arithmetic pipeline usually requires two or more operands to enter the pipeline at the same time. Instead of using two memory buses for simultaneous access, the memory can be partitioned into a number of modules connected to a common memory address and data buses. A memory module is a memory array together with its own address and data registers. Figure 9-13 shows a memory unit with four modules. Each memory array has its own address register *AR* and data register *DR*. The address registers receive information from a common address bus and the data registers communicate with a bidirectional data bus. The two least significant bits of the address can be used to distinguish between the four modules. The modular system permits one module to initiate a memory access while other modules are in the process of reading or writing a word and each module can honor a memory request independent of the state of the other modules.

Figure 9-13 Multiple module memory organization.



The advantage of a modular memory is that it allows the use of a technique called *interleaving*. In an interleaved memory, different sets of addresses are assigned to different memory modules. For example, in a two-module memory system, the even addresses may be in one module and the odd addresses in the other. When the number of modules is a power of 2, the least significant bits of the address select a memory module and the remaining bits designate the specific location to be accessed within the selected module.

A modular memory is useful in systems with pipeline and vector processing. A vector processor that uses an  $n$ -way interleaved memory can fetch  $n$  operands from  $n$  different modules. By staggering the memory access, the effective memory cycle time can be reduced by a factor close to the number of modules. A CPU with instruction pipeline can take advantage of multiple memory modules so that each segment in the pipeline can access memory independent of memory access from other segments.

### Supercomputers

A commercial computer with vector instructions and pipelined floating-point arithmetic operations is referred to as a *supercomputer*. Supercomputers are very powerful, high-performance machines used mostly for scientific computations. To speed up the operation, the components are packed tightly together to minimize the distance that the electronic signals have to travel. Supercomputers also use special techniques for removing the heat from circuits to prevent them from burning up because of their close proximity.

The instruction set of supercomputers contains the standard data transfer, data manipulation, and program control instructions of conventional computers. This is augmented by instructions that process vectors and combinations of scalars and vectors. A supercomputer is a computer system best known for its high computational speed, fast and large memory systems, and the extensive use of parallel processing. It is equipped with multiple functional units and each unit has its own pipeline configuration. Although the supercomputer is capable of general-purpose applications found in all other computers, it is specifically optimized for the type of numerical calculations involving vectors and matrices of floating-point numbers.

Supercomputers are not suitable for normal everyday processing of a typical computer installation. They are limited in their use to a number of scientific applications, such as numerical weather forecasting, seismic wave analysis, and space research. They have limited use and limited market because of their high price.

A measure used to evaluate computers in their ability to perform a given number of floating-point operations per second is referred to as *flops*. The term *megaflops* is used to denote million flops and *gigaflops* to denote billion flops. A typical supercomputer has a basic cycle time of 4 to 20 ns. If the processor can calculate a floating-point operation through a pipeline each cycle time, it will have the ability to perform 50 to 250 megaflops. This rate would be

sustained from the time the first answer is produced and does not include the initial setup time of the pipeline.

The first supercomputer developed in 1976 is the Cray-1 supercomputer. It uses vector processing with 12 distinct functional units in parallel. Each functional unit is segmented to process the incoming data through a pipeline. All the functional units can operate concurrently with operands stored in the large number of registers (over 150) in the CPU. A floating-point operation can be performed on two sets of 64-bit operands during one clock cycle of 12.5 ns. This gives a rate of 80 megaflops during the time that the data are processed through the pipeline. It has a memory capacity of 4 million 64-bit words. The memory is divided into 16 banks, with each bank having a 50-ns access time. This means that when all 16 banks are accessed simultaneously, the memory transfer rate is 320 million words per second. Cray research extended its supercomputer to a multiprocessor configuration called Cray X-MP and Cray Y-MP. The new Cray-2 supercomputer is 12 times more powerful than the Cray-1 in vector processing mode.

Another early model supercomputer is the Fujitsu VP-200. It has a scalar processor and a vector processor that can operate concurrently. Like the Cray supercomputers, a large number of registers and multiple functional units are used to enable register-to-register vector operations. There are four execution pipelines in the vector processor, and when operating simultaneously, they can achieve up to 300 megaflops. The main memory has 32 million words connected to the vector registers through load and store pipelines. The VP-200 has 83 vector instructions and 195 scalar instructions. The newer VP-2600 uses a clock cycle of 3.2 ns and claims a peak performance of 5 gigaflops.

## 9-7 Array Processors

---

An array processor is a processor that performs computations on large arrays of data. The term is used to refer to two different types of processors. An *attached array processor* is an auxiliary processor attached to a general-purpose computer. It is intended to improve the performance of the host computer in specific numerical computation tasks. An *SIMD array processor* is a processor that has a single-instruction multiple-data organization. It manipulates vector instructions by means of multiple functional units responding to a common instruction. Although both types of array processors manipulate vectors, their internal organization is different.

### Attached Array Processor

An attached array processor is designed as a peripheral for a conventional host computer, and its purpose is to enhance the performance of the computer by providing vector processing for complex scientific applications. It achieves



high performance by means of parallel processing with multiple functional units. It includes an arithmetic unit containing one or more pipelined floating-point adders and multipliers. The array processor can be programmed by the user to accommodate a variety of complex arithmetic problems.

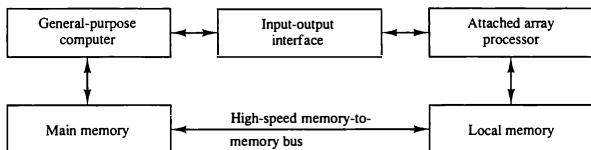
Figure 9-14 shows the interconnection of an attached array processor to a host computer. The host computer is a general-purpose commercial computer and the attached processor is a back-end machine driven by the host computer. The array processor is connected through an input-output controller to the computer and the computer treats it like an external interface. The data for the attached processor are transferred from main memory to a local memory through a high-speed bus. The general-purpose computer without the attached processor serves the users that need conventional data processing. The system with the attached processor satisfies the needs for complex arithmetic applications.

Some manufacturers of attached array processors offer a model that can be connected to a variety of different host computers. For example, when attached to a VAX 11 computer, the FSP-164/MAX from Floating-Point Systems increases the computing power of the VAX to 100 megaflops. The objective of the attached array processor is to provide vector manipulation capabilities to a conventional computer at a fraction of the cost of supercomputers.

### SIMD Array Processor

An SIMD array processor is a computer with multiple processing units operating in parallel. The processing units are synchronized to perform the same operation under the control of a common control unit, thus providing a single instruction stream, multiple data stream (SIMD) organization. A general block diagram of an array processor is shown in Fig. 9-15. It contains a set of identical processing elements (PEs), each having a local memory  $M$ . Each processor element includes an ALU, a floating-point arithmetic unit, and working registers. The master control unit controls the operations in the processor elements. The main memory is used for storage of the program. The function of the master control unit is to decode the instructions and determine how the instruction is to be executed. Scalar and program control instructions are

Figure 9-14 Attached array processor with host computer.



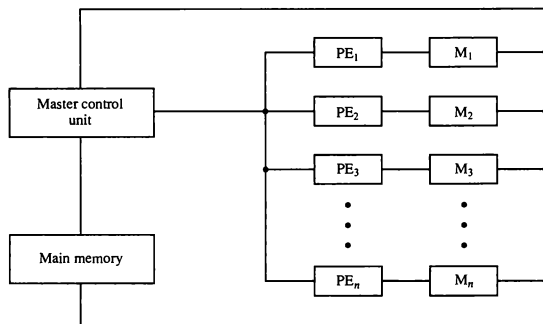


Figure 9-15 SIMD array processor organization.

directly executed within the master control unit. Vector instructions are broadcast to all PEs simultaneously. Each PE uses operands stored in its local memory. Vector operands are distributed to the local memories prior to the parallel execution of the instruction.

Consider, for example, the vector addition  $C = A + B$ . The master control unit first stores the  $i$ th components  $a_i$  and  $b_i$  of  $A$  and  $B$  in local memory  $M_i$  for  $i = 1, 2, 3, \dots, n$ . It then broadcasts the floating-point add instruction  $c_i = a_i + b_i$  to all PEs, causing the addition to take place simultaneously. The components of  $c_i$  are stored in fixed locations in each local memory. This produces the desired vector sum in one add cycle.

Masking schemes are used to control the status of each PE during the execution of vector instructions. Each PE has a flag that is set when the PE is active and reset when the PE is inactive. This ensures that only those PEs that need to participate are active during the execution of the instruction. For example, suppose that the array processor contains a set of 64 PEs. If a vector length of less than 64 data items is to be processed, the control unit selects the proper number of PEs to be active. Vectors of greater length than 64 must be divided into 64-word portions by the control unit.

The best known SIMD array processor is the ILLIAC IV computer developed at the University of Illinois and manufactured by the Burroughs Corp. This computer is no longer in operation. SIMD processors are highly specialized computers. They are suited primarily for numerical problems that can be expressed in vector or matrix form. However, they are not very efficient in other types of computations or in dealing with conventional data-processing programs.

# Multiprocessors

## IN THIS CHAPTER

- 13-1 Characteristics of Multiprocessors
- 13-2 Interconnection Structures
- 13-3 Interprocessor Arbitration
- 13-4 Interprocessor Communication and Synchronization
- 13-5 Cache Coherence

### 13-1 Characteristics of Multiprocessors

A multiprocessor system is an interconnection of two or more CPUs with memory and input-output equipment. The term "processor" in *multiprocessor* can mean either a central processing unit (CPU) or an input-output processor (IOP). However, a system with a single CPU and one or more IOPs is usually not included in the definition of a multiprocessor system unless the IOP has computational facilities comparable to a CPU. As it is most commonly defined, a multiprocessor system implies the existence of multiple CPUs, although usually there will be one or more IOPs as well. As mentioned in Sec. 9-1, multiprocessors are classified as multiple instruction stream, multiple data stream (MIMD) systems.

There are some similarities between multiprocessor and multicomputer systems since both support concurrent operations. However, there exists an important distinction between a system with multiple computers and a system with multiple processors. Computers are interconnected with each other by means of communication lines to form a *computer network*. The network consists of several autonomous computers that may or may not communicate with each other. A multiprocessor system is controlled by one operating system that provides interaction between processors and all the components of the system cooperate in the solution of a problem.

**MIMD**

*microprocessor**VLSI*

Although some large-scale computers include two or more CPUs in their overall system, it is the emergence of the microprocessor that has been the major motivation for multiprocessor systems. The fact that microprocessors take very little physical space and are very inexpensive brings about the feasibility of interconnecting a large number of microprocessors into one composite system. Very-large-scale integrated circuit technology has reduced the cost of computer components to such a low level that the concept of applying multiple processors to meet system performance requirements has become an attractive design possibility.

Multiprocessing improves the reliability of the system so that a failure or error in one part has a limited effect on the rest of the system. If a fault causes one processor to fail, a second processor can be assigned to perform the functions of the disabled processor. The system as a whole can continue to function correctly with perhaps some loss in efficiency.

The benefit derived from a multiprocessor organization is an improved system performance. The system derives its high performance from the fact that computations can proceed in parallel in one of two ways.

1. Multiple independent jobs can be made to operate in parallel.
2. A single job can be partitioned into multiple parallel tasks.

An overall function can be partitioned into a number of tasks that each processor can handle individually. System tasks may be allocated to special-purpose processors whose design is optimized to perform certain types of processing efficiently. An example is a computer system where one processor performs the computations for an industrial process control while others monitor and control the various parameters, such as temperature and flow rate. Another example is a computer where one processor performs high-speed floating-point mathematical computations and another takes care of routine data-processing tasks.

Multiprocessing can improve performance by decomposing a program into parallel executable tasks. This can be achieved in one of two ways. The user can explicitly declare that certain tasks of the program be executed in parallel. This must be done prior to loading the program by specifying the parallel executable segments. Most multiprocessor manufacturers provide an operating system with programming language constructs suitable for specifying parallel processing. The other, more efficient way is to provide a compiler with multiprocessor software that can automatically detect parallelism in a user's program. The compiler checks for *data dependency* in the program. If a program depends on data generated in another part, the part yielding the needed data must be executed first. However, two parts of a program that do not use data generated by each can run concurrently. The parallelizing compiler checks the entire program to detect any possible data dependencies. Those that have no data dependency are then considered for concurrent scheduling on different processors.

*tightly coupled*

Multiprocessors are classified by the way their memory is organized. A multiprocessor system with common shared memory is classified as a *shared-memory* or *tightly coupled multiprocessor*. This does not preclude each processor from having its own local memory. In fact, most commercial tightly coupled multiprocessors provide a cache memory with each CPU. In addition, there is a global common memory that all CPUs can access. Information can therefore be shared among the CPUs by placing it in the common global memory.

*loosely coupled*

An alternative model of microprocessor is the *distributed-memory* or *loosely coupled* system. Each processor element in a loosely coupled system has its own private local memory. The processors are tied together by a switching scheme designed to route information from one processor to another through a message-passing scheme. The processors relay program and data to other processors in packets. A packet consists of an address, the data content, and some error detection code. The packets are addressed to a specific processor or taken by the first available processor, depending on the communication system used. Loosely coupled systems are most efficient when the interaction between tasks is minimal, whereas tightly coupled systems can tolerate a higher degree of interaction between tasks.

## 13-2 Interconnection Structures

---

The components that form a multiprocessor system are CPUs, IOPs connected to input-output devices, and a memory unit that may be partitioned into a number of separate modules. The interconnection between the components can have different physical configurations, depending on the number of transfer paths that are available between the processors and memory in a shared memory system or among the processing elements in a loosely coupled system. There are several physical forms available for establishing an interconnection network. Some of these schemes are presented in this section:

1. Time-shared common bus
2. Multiport memory
3. Crossbar switch
4. Multistage switching network
5. Hypercube system

### Time-Shared Common Bus

A common-bus multiprocessor system consists of a number of processors connected through a common path to a memory unit. A time-shared common bus for five processors is shown in Fig. 13-1. Only one processor can communicate with the memory or another processor at any given time. Transfer

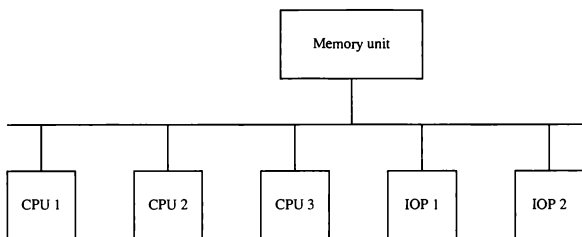


Figure 13-1 Time-shared common bus organization.

operations are conducted by the processor that is in control of the bus at the time. Any other processor wishing to initiate a transfer must first determine the availability status of the bus, and only after the bus becomes available can the processor address the destination unit to initiate the transfer. A command is issued to inform the destination unit what operation is to be performed. The receiving unit recognizes its address in the bus and responds to the control signals from the sender, after which the transfer is initiated. The system may exhibit transfer conflicts since one common bus is shared by all processors. These conflicts must be resolved by incorporating a bus controller that establishes priorities among the requesting units.

A single common-bus system is restricted to one transfer at a time. This means that when one processor is communicating with the memory, all other processors are either busy with internal operations or must be idle waiting for the bus. As a consequence, the total overall transfer rate within the system is limited by the speed of the single path. The processors in the system can be kept busy more often through the implementation of two or more independent buses to permit multiple simultaneous bus transfers. However, this increases the system cost and complexity.

A more economical implementation of a dual bus structure is depicted in Fig. 13-2. Here we have a number of local buses each connected to its own local memory and to one or more processors. Each local bus may be connected to a CPU, an IOP, or any combination of processors. A system bus controller links each local bus to a common system bus. The I/O devices connected to the local IOP, as well as the local memory, are available to the local processor. The memory connected to the common system bus is shared by all processors. If an IOP is connected directly to the system bus, the I/O devices attached to it may be made available to all processors. Only one processor can communicate with the shared memory and other common resources through the system bus at any given time. The other processors are kept busy communicating with their local memory and I/O devices. Part of the local memory may be designed

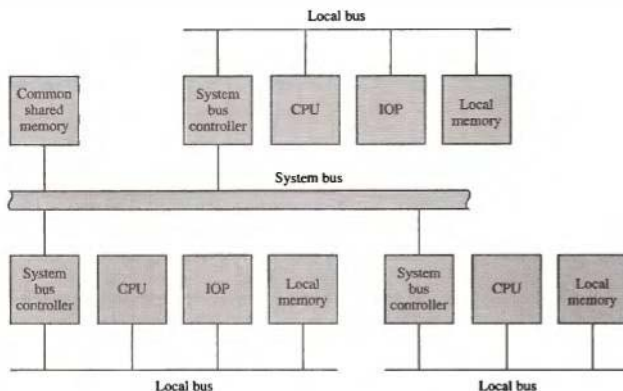


Figure 13-2 System bus structure for multiprocessors.

as a cache memory attached to the CPU (see Sec. 12-6). In this way, the average access time of the local memory can be made to approach the cycle time of the CPU to which it is attached.

### Multipoint Memory

A multipoint memory system employs separate buses between each memory module and each CPU. This is shown in Fig. 13-3 for four CPUs and four memory modules (MMs). Each processor bus is connected to each memory module. A processor bus consists of the address, data, and control lines required to communicate with memory. The memory module is said to have four ports and each port accommodates one of the buses. The module must have internal control logic to determine which port will have access to memory at any given time. Memory access conflicts are resolved by assigning fixed priorities to each memory port. The priority for memory access associated with each processor may be established by the physical port position that its bus occupies in each module. Thus CPU 1 will have priority over CPU 2, CPU 2 will have priority over CPU 3, and CPU 4 will have the lowest priority.

The advantage of the multipoint memory organization is the high transfer rate that can be achieved because of the multiple paths between processors and memory. The disadvantage is that it requires expensive memory control logic and a large number of cables and connectors. As a consequence, this intercon-

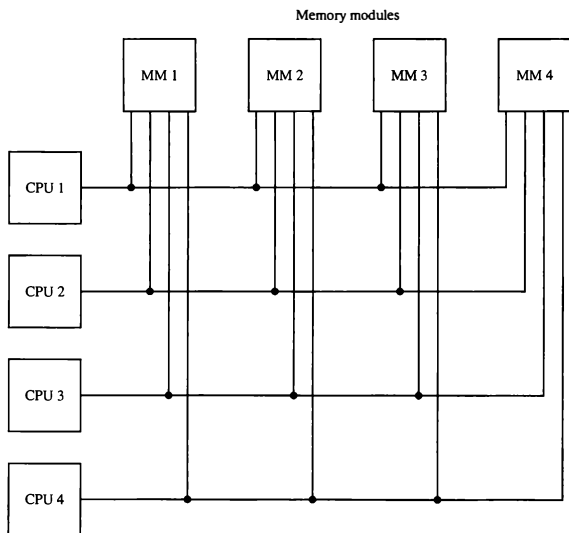


Figure 13-3 Multiport memory organization.

nection structure is usually appropriate for systems with a small number of processors.

### Crossbar Switch

The crossbar switch organization consists of a number of crosspoints that are placed at intersections between processor buses and memory module paths. Figure 13-4 shows a crossbar switch interconnection between four CPUs and four memory modules. The small square in each crosspoint is a switch that determines the path from a processor to a memory module. Each switch point has control logic to set up the transfer path between a processor and memory. It examines the address that is placed in the bus to determine whether its particular module is being addressed. It also resolves multiple requests for access to the same memory module on a predetermined priority basis.

Figure 13-5 shows the functional design of a crossbar switch connected to one memory module. The circuit consists of multiplexers that select the data,



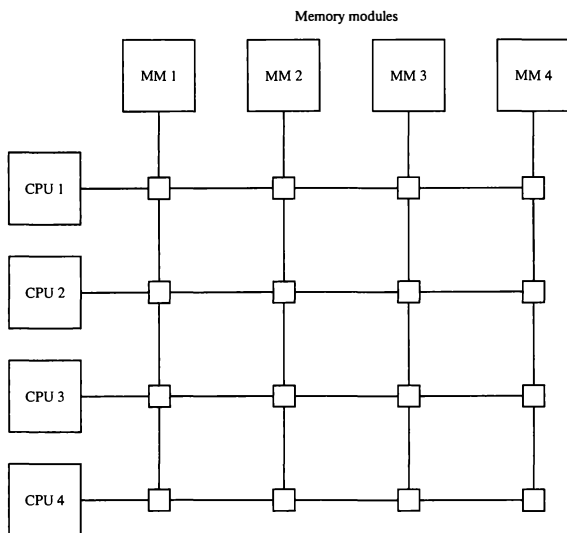
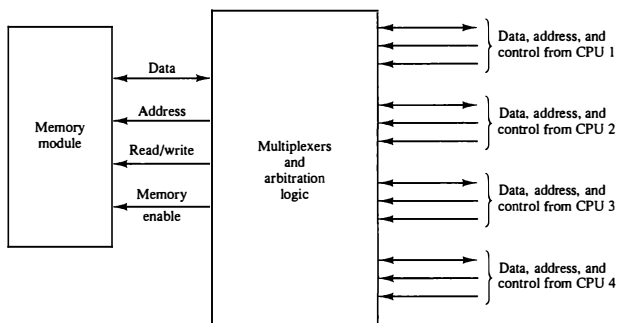


Figure 13-4 Crossbar switch.

Figure 13-5 Block diagram of crossbar switch.



address, and control from one CPU for communication with the memory module. Priority levels are established by the arbitration logic to select one CPU when two or more CPUs attempt to access the same memory. The multiplexers are controlled with the binary code that is generated by a priority encoder within the arbitration logic.

A crossbar switch organization supports simultaneous transfers from memory modules because there is a separate path associated with each module. However, the hardware required to implement the switch can become quite large and complex.

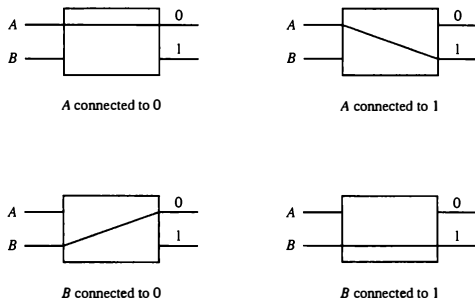
### Multistage Switching Network

#### *interchange switch*

The basic component of a multistage network is a two-input, two-output interchange switch. As shown in Fig. 13-6, the  $2 \times 2$  switch has two inputs labeled  $A$  and  $B$ , and two outputs, labeled 0 and 1. There are control signals (not shown) associated with the switch that establish the interconnection between the input and output terminals. The switch has the capability of connecting input  $A$  to either of the outputs. Terminal  $B$  of the switch behaves in a similar fashion. The switch also has the capability to arbitrate between conflicting requests. If inputs  $A$  and  $B$  both request the same output terminal, only one of them will be connected; the other will be blocked.

Using the  $2 \times 2$  switch as a building block, it is possible to build a multistage network to control the communication between a number of sources and destinations. To see how this is done, consider the binary tree shown in Fig. 13-7. The two processors  $P_1$  and  $P_2$  are connected through switches to eight memory modules marked in binary from 000 through 111. The path from source to a destination is determined from the binary bits of the destination.

Figure 13-6 Operation of a  $2 \times 2$  interchange switch.



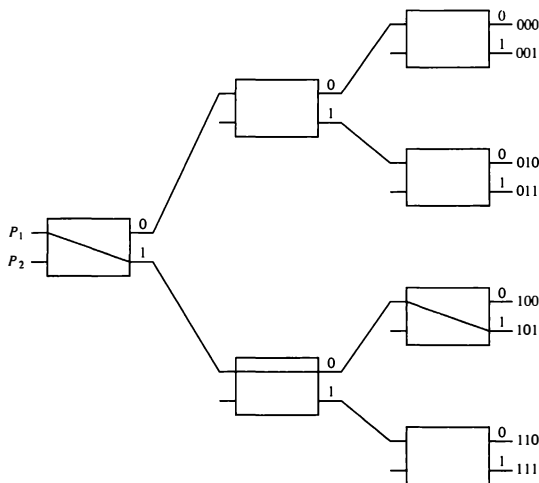


Figure 13-7 Binary tree with  $2 \times 2$  switches.

number. The first bit of the destination number determines the switch output in the first level. The second bit specifies the output of the switch in the second level, and the third bit specifies the output of the switch in the third level. For example, to connect  $P_1$  to memory 101, it is necessary to form a path from  $P_1$  to output 1 in the first-level switch, output 0 in the second-level switch, and output 1 in the third-level switch. It is clear that either  $P_1$  or  $P_2$  can be connected to any one of the eight memories. Certain request patterns, however, cannot be satisfied simultaneously. For example, if  $P_1$  is connected to one of the destinations 000 through 011,  $P_2$  can be connected to only one of the destinations 100 through 111.

Many different topologies have been proposed for multistage switching networks to control processor-memory communication in a tightly coupled multiprocessor system or to control the communication between the processing elements in a loosely coupled system. One such topology is the omega switching network shown in Fig. 13-8. In this configuration, there is exactly one path from each source to any particular destination. Some request patterns, however, cannot be connected simultaneously. For example, any two sources cannot be connected simultaneously to destinations 000 and 001.

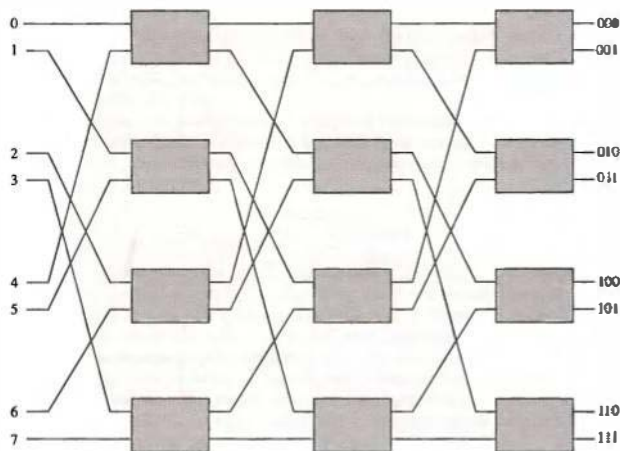


Figure 13-8 8 x 8 omega switching network.

A particular request is initiated in the switching network by the source, which sends a 3-bit pattern representing the destination number. As the binary pattern moves through the network, each level examines a different bit to determine the  $2 \times 2$  switch setting. Level 1 inspects the most significant bit, level 2 inspects the middle bit, and level 3 inspects the least significant bit. When the request arrives on either input of the  $2 \times 2$  switch, it is routed to the upper output if the specified bit is 0 or to the lower output if the bit is 1.

In a tightly coupled multiprocessor system, the source is a processor and the destination is a memory module. The first pass through the network sets up the path. Succeeding passes are used to transfer the address into memory and then transfer the data in either direction, depending on whether the request is a read or a write. In a loosely coupled multiprocessor system, both the source and destination are processing elements. After the path is established, the source processor transfers a message to the destination processor.

### Hypercube Interconnection

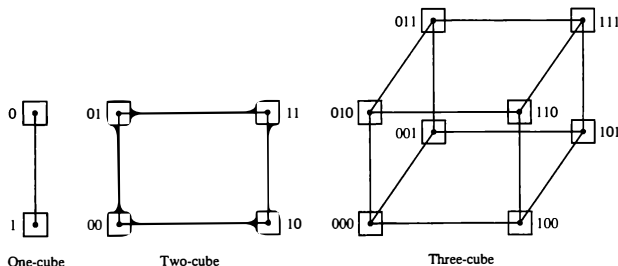
The hypercube or binary  $n$ -cube multiprocessor structure is a loosely coupled system composed of  $N = 2^n$  processors interconnected in an  $n$ -dimensional binary cube. Each processor forms a node of the cube. Although it is customary

to refer to each node as having a processor, in effect it contains not only a CPU but also local memory and I/O interface. Each processor has direct communication paths to  $n$  other neighbor processors. These paths correspond to the edges of the cube. There are  $2^n$  distinct  $n$ -bit binary addresses that can be assigned to the processors. Each processor address differs from that of each of its  $n$  neighbors by exactly one bit position.

Figure 13-9 shows the hypercube structure for  $n = 1, 2$ , and 3. A one-cube structure has  $n = 1$  and  $2^n = 2$ . It contains two processors interconnected by a single path. A two-cube structure has  $n = 2$  and  $2^n = 4$ . It contains four nodes interconnected as a square. A three-cube structure has eight nodes interconnected as a cube. An  $n$ -cube structure has  $2^n$  nodes with a processor residing in each node. Each node is assigned a binary address in such a way that the addresses of two neighbors differ in exactly one bit position. For example, the three neighbors of the node with address 100 in a three-cube structure are 000, 110, and 101. Each of these binary numbers differs from address 100 by one bit value.

Routing messages through an  $n$ -cube structure may take from one to  $n$  links from a source node to a destination node. For example, in a three-cube structure, node 000 can communicate directly with node 001. It must cross at least two links to communicate with 011 (from 000 to 001 to 011 or from 000 to 010 to 011). It is necessary to go through at least three links to communicate from node 000 to node 111. A routing procedure can be developed by computing the exclusive-OR of the source node address with the destination node address. The resulting binary value will have 1 bits corresponding to the axes on which the two nodes differ. The message is then sent along any one of the axes. For example, in a three-cube structure, a message at 010 going to 001 produces an exclusive-OR of the two addresses equal to 011. The message can be sent along the second axis to 000 and then through the third axis to 001.

Figure 13-9 Hypercube structures for  $n = 1, 2, 3$ .



A representative of the hypercube architecture is the Intel iPSC computer complex. It consists of 128 ( $n = 7$ ) microcomputers connected through communication channels. Each node consists of a CPU, a floating-point processor, local memory, and serial communication interface units. The individual nodes operate independently on data stored in local memory according to resident programs. The data and programs to each node come through a message-passing system from other nodes or from a cube manager. Application programs are developed and compiled on the cube manager and then downloaded to the individual nodes. Computations are distributed through the system and executed concurrently.

### 13-3 Interprocessor Arbitration

---

*system bus*

Computer systems contain a number of buses at various levels to facilitate the transfer of information between components. The CPU contains a number of internal buses for transferring information between processor registers and ALU. A memory bus consists of lines for transferring data, address, and read/write information. An I/O bus is used to transfer information to and from input and output devices. A bus that connects major components in a multiprocessor system, such as CPUs, IOPs, and memory, is called a *system bus*. The physical circuits of a system bus are contained in a number of identical printed circuit boards. Each board in the system belongs to a particular module. The board consists of circuits connected in parallel through connectors. Each pin of each circuit connector is connected by a wire to the corresponding pin of all other connectors in other boards. Thus any board can be plugged into a slot in the backplane that forms the system bus.

The processors in a shared memory multiprocessor system request access to common memory or other common resources through the system bus. If no other processor is currently utilizing the bus, the requesting processor may be granted access immediately. However, the requesting processor must wait if another processor is currently utilizing the system bus. Furthermore, other processors may request the system bus at the same time. Arbitration must then be performed to resolve this multiple contention for the shared resources. The arbitration logic would be part of the system bus controller placed between the local bus and the system bus as shown in Fig. 13-2.

#### System Bus

A typical system bus consists of approximately 100 signal lines. These lines are divided into three functional groups: data, address, and control. In addition, there are power distribution lines that supply power to the components. For example, the IEEE standard 796 multibus system has 16 data lines, 24 address lines, 26 control lines, and 20 power lines, for a total of 86 lines.

The data lines provide a path for the transfer of data between processors and common memory. The number of data lines is usually a multiple of 8, with 16 and 32 being most common. The address lines are used to identify a memory address or any other source or destination, such as input or output ports. The number of address lines determines the maximum possible memory capacity in the system. For example, an address of 24 lines can access up to  $2^{24}$  (16 mega) words of memory. The data and address lines are terminated with three-state buffers (see Fig. 4-5). The address buffers are unidirectional from processor to memory. The data lines are bidirectional (see Fig. 12-3), allowing the transfer of data in either direction.

*synchronous bus*

Data transfers over the system bus may be synchronous or asynchronous. In a synchronous bus, each data item is transferred during a time slice known in advance to both source and destination units. Synchronization is achieved by driving both units from a common clock source. An alternative procedure is to have separate clocks of approximately the same frequency in each unit. Synchronization signals are transmitted periodically in order to keep all clocks in the system in step with each other. In an asynchronous bus, each data item being transferred is accompanied by handshaking control signals (see Fig. 11-9) to indicate when the data are transferred from the source and received by the destination.

*asynchronous bus*

The control lines provide signals for controlling the information transfer between units. Timing signals indicate the validity of data and address information. Command signals specify operations to be performed. Typical control lines include transfer signals such as memory read and write, acknowledge of a transfer, interrupt requests, bus control signals such as bus request and bus grant, and signals for arbitration procedures.

Table 13-1 lists the 86 lines that are available in the IEEE standard 796 multibus. It includes 16 data lines and 24 address lines. All signals in the multibus are active or enabled in the low-level state. The data transfer control signals include memory read and write as well as I/O read and write. Consequently, the address lines can be used to address separate memory and I/O spaces. The memory or I/O responds with a transfer acknowledge signal when the transfer is completed. Each processor attached to the multibus has up to eight interrupt request outputs and one interrupt acknowledge input line. They are usually applied to a priority interrupt controller similar to the one described in Fig. 11-21. The miscellaneous control signals provide timing and initialization capabilities. In particular, the bus lock signal is essential for multiprocessor applications. This processor-activated signal serves to prevent other processors from getting hold of the bus while executing a test and set instruction. This instruction is needed for proper processor synchronization (see Sec. 13-4).

The six bus arbitration signals are used for interprocessor arbitration. These signals will be explained later after a discussion of the serial and parallel arbitration procedures.

TABLE 13-1 IEEE Standard 796 Multibus Signals

	Signal name
Data and address	
Data lines (16 lines)	DATA0–DATA15
Address lines (24 lines)	ADRS0–ADRS23
Data transfer	
Memory read	MRDC
Memory write	MWTC
IO read	IORC
IO write	IOWC
Transfer acknowledge	TACK
Interrupt control	
Interrupt request (8 lines)	INT0–INT7
Interrupt acknowledge	INTA
Miscellaneous control	
Master clock	CCLK
System initialization	INIT
Byte high enable	BHEN
Memory inhibit (2 lines)	INH1–INH2
Bus lock	LOCK
Bus arbitration	
Bus request	BREQ
Common bus request	CBRQ
Bus busy	BUSY
Bus clock	BCLK
Bus priority in	BPRN
Bus priority out	BPRO
Power and ground (20 lines)	

Reprinted with permission of the IEEE.

### Serial Arbitration Procedure

Arbitration procedures service all processor requests on the basis of established priorities. A hardware bus priority resolving technique can be established by means of a serial or parallel connection of the units requesting control of the system bus. The serial priority resolving technique is obtained from a daisy-chain connection of bus arbitration circuits similar to the priority interrupt logic presented in Sec. 11-5. The processors connected to the system bus are assigned priority according to their position along the priority control line. The device closest to the priority line is assigned the highest priority. When multiple devices concurrently request the use of the bus, the device with the highest priority is granted access to it.

Figure 13-10 shows the daisy-chain connection of four arbiters. It is assumed that each processor has its own bus arbiter logic with priority-in and priority-out lines. The priority out (*PO*) of each arbiter is connected to the



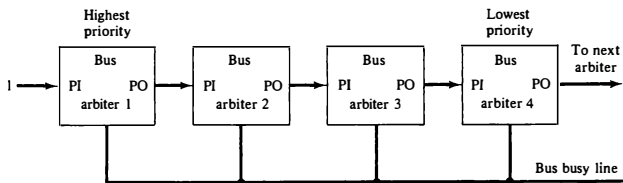


Figure 13-10 Serial (daisy-chain) arbitration.

priority (*PI*) of the next-lower-priority arbiter. The *PI* of the highest-priority unit is maintained at a logic 1 value. The highest-priority unit in the system will always receive access to the system bus when it requests it. The *PO* output for a particular arbiter is equal to 1 if its *PI* input is equal to 1 and the processor associated with the arbiter logic is not requesting control of the bus. This is the way that priority is passed to the next unit in the chain. If the processor requests control of the bus and the corresponding arbiter finds its *PI* input equal to 1, it sets its *PO* output to 0. Lower-priority arbiters receive a 0 in *PI* and generate a 0 in *PO*. Thus the processor whose arbiter has a *PI* = 1 and *PO* = 0 is the one that is given control of the system bus.

A processor may be in the middle of a bus operation when a higher-priority processor requests the bus. The lower-priority processor must complete its bus operation before it relinquishes control of the bus. The bus busy line shown in Fig. 13-10 provides a mechanism for an orderly transfer of control. The busy line comes from open-collector circuits in each unit and provides a wired-OR logic connection. When an arbiter receives control of the bus (because its *PI* = 1 and *PO* = 0) it examines the busy line. If the line is inactive, it means that no other processor is using the bus. The arbiter activates the busy line and its processor takes control of the bus. However, if the arbiter finds the busy line active, it means that another processor is currently using the bus. The arbiter keeps examining the busy line while the lower-priority processor that lost control of the bus completes its operation. When the bus busy line returns to its inactive state, the higher-priority arbiter enables the busy line, and its corresponding processor can then conduct the required bus transfers.

### Parallel Arbitration Logic

The parallel bus arbitration technique uses an external priority encoder and a decoder as shown in Fig. 13-11. Each bus arbiter in the parallel scheme has a bus request output line and a bus acknowledge input line. Each arbiter enables the request line when its processor is requesting access to the system

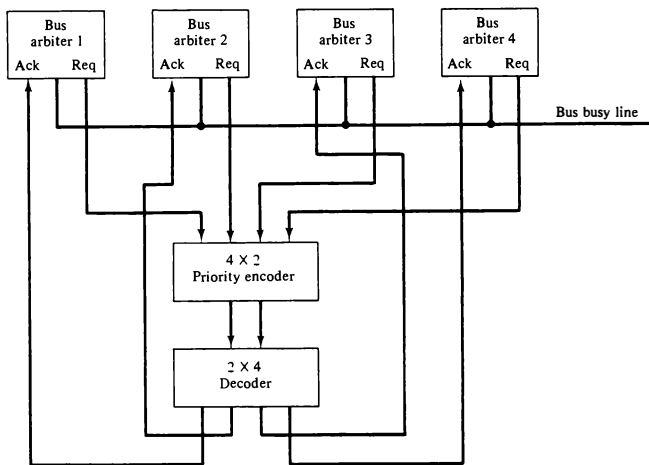


Figure 13-11 Parallel arbitration.

bus. The processor takes control of the bus if its acknowledge input line is enabled. The bus busy line provides an orderly transfer of control, as in the daisy-chaining case.

Figure 13-11 shows the request lines from four arbiters going into a  $4 \times 2$  priority encoder. The output of the encoder generates a 2-bit code which represents the highest-priority unit among those requesting the bus. The truth table of the priority encoder can be found in Table 11-2 (Sec. 11-5). The 2-bit code from the encoder output drives a  $2 \times 4$  decoder which enables the proper acknowledge line to grant bus access to the highest-priority unit.

We can now explain the function of the bus arbitration signals listed in Table 13-1. The bus priority-in BPRN and bus priority-out BPRO are used for a daisy-chain connection of bus arbitration circuits. The bus busy signal BUSY is an open-collector output used to instruct all arbiters when the bus is busy conducting a transfer. The common bus request CBRQ is also an open-collector output that serves to instruct the arbiter if there are any other arbiters of lower-priority requesting use of the system bus. The signals used to construct a parallel arbitration procedure are bus request BREQ and priority-in BPRN,

corresponding to the request and acknowledge signals in Fig. 13-11. The bus clock BCLK is used to synchronize all bus transactions.

### Dynamic Arbitration Algorithms

The two bus arbitration procedures just described use a static priority algorithm since the priority of each device is fixed by the way it is connected to the bus. In contrast, a dynamic priority algorithm gives the system the capability for changing the priority of the devices while the system is in operation. We now discuss a few arbitration procedures that use dynamic priority algorithms.

#### *time slice*

The *time slice* algorithm allocates a fixed-length time slice of bus time that is offered sequentially to each processor, in round-robin fashion. The service given to each system component with this scheme is independent of its location along the bus. No preference is given to any particular device since each is allotted the same amount of time to communicate with the bus.

#### *polling*

In a bus system that uses *polling*, the bus grant signal is replaced by a set of lines called poll lines which are connected to all units. These lines are used by the bus controller to define an address for each device connected to the bus. The bus controller sequences through the addresses in a prescribed manner. When a processor that requires access recognizes its address, it activates the bus busy line and then accesses the bus. After a number of bus cycles, the polling process continues by choosing a different processor. The polling sequence is normally programmable, and as a result, the selection priority can be altered under program control.

#### *LRU*

The *least recently used* (LRU) algorithm gives the highest priority to the requesting device that has not used the bus for the longest interval. The priorities are adjusted after a number of bus cycles according to the LRU algorithm. With this procedure, no processor is favored over any other since the priorities are dynamically changed to give every device an opportunity to access the bus.

#### *FIFO*

In the *first-come, first-serve* scheme, requests are served in the order received. To implement this algorithm, the bus controller establishes a queue arranged according to the time that the bus requests arrive. Each processor must wait for its turn to use the bus on a first-in, first-out (FIFO) basis.

#### *rotating daisy-chain*

The *rotating daisy-chain* procedure is a dynamic extension of the daisy-chain algorithm. In this scheme there is no central bus controller, and the priority line is connected from the priority-out of the last device back to the priority-in of the first device in a closed loop. This is similar to the connections shown in Fig. 13-10 except that the PO output of arbiter 4 is connected to the PI input of arbiter 1. Whichever device has access to the bus serves as a bus controller for the following arbitration. Each arbiter priority for a given bus cycle is determined by its position along the bus priority line from the arbiter

whose processor is currently controlling the bus. Once an arbiter releases the bus, it has the lowest priority.

### 13-4 Interprocessor Communication and Synchronization

---

The various processors in a multiprocessor system must be provided with a facility for communicating with each other. A communication path can be established through common input-output channels. In a shared memory multiprocessor system, the most common procedure is to set aside a portion of memory that is accessible to all processors. The primary use of the common memory is to act as a message center similar to a mailbox, where each processor can leave messages for other processors and pick up messages intended for it.

The sending processor structures a request, a message, or a procedure, and places it in the memory mailbox. Status bits residing in common memory are generally used to indicate the condition of the mailbox, whether it has meaningful information, and for which processor it is intended. The receiving processor can check the mailbox periodically to determine if there are valid messages for it. The response time of this procedure can be time consuming since a processor will recognize a request only when polling messages. A more efficient procedure is for the sending processor to alert the receiving processor directly by means of an interrupt signal. This can be accomplished through a software-initiated interprocessor interrupt by means of an instruction in the program of one processor which when executed produces an external interrupt condition in a second processor. This alerts the interrupted processor of the fact that a new message was inserted by the interrupting processor.

In addition to shared memory, a multiprocessor system may have other shared resources. For example, a magnetic disk storage unit connected to an IOP may be available to all CPUs. This provides a facility for sharing of system programs stored in the disk. A communication path between two CPUs can be established through a link between two IOPs associated with two different CPUs. This type of link allows each CPU to treat the other as an I/O device so that messages can be transferred through the I/O path.

To prevent conflicting use of shared resources by several processors there must be a provision for assigning resources to processors. This task is given to the operating system. 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.

In a master-slave mode, one processor, designated the master, always executes the operating system functions. The remaining processors, denoted as slaves, do not perform operating system functions. If a slave processor needs

an operating system service, it must request it by interrupting the master and waiting until the current program can be interrupted.

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 where every processor may have its own copy of the entire operating system.

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. This type of organization is also referred to as a floating operating system since the routines float from one processor to another and the execution of the routines may be assigned to different processors at different times.

In a loosely coupled multiprocessor system the memory is distributed among the processors and there is no shared memory for passing information. The communication between processors is by means of message passing through I/O channels. The communication is initiated by one processor calling a procedure that resides in the memory of the processor with which it wishes to communicate. When the sending processor and receiving processor name each other as a source and destination, a channel of communication is established. A message is then sent with a header and various data objects used to communicate between nodes. There may be a number of possible paths available to send the message between any two nodes. The operating system in each node contains routing information indicating the alternative paths that can be used to send a message to other nodes. The communication efficiency of the interprocessor network depends on the communication routing protocol, processor speed, data link speed, and the topology of the network.

### Interprocessor Synchronization

The instruction set of a multiprocessor contains basic instructions that are used to implement communication and synchronization between cooperating processes. Communication refers to the exchange of data between different processes. For example, parameters passed to a procedure in a different processor constitute interprocessor communication. Synchronization refers to the special case where the data used to communicate between processors is control information. Synchronization is needed to enforce the correct sequence of processes and to ensure mutually exclusive access to shared writable data.

Multiprocessor systems usually include various mechanisms to deal with the synchronization of resources. Low-level primitives are implemented directly by the hardware. These primitives are the basic mechanisms that enforce mutual exclusion for more complex mechanisms implemented in software. A number of hardware mechanisms for mutual exclusion have been developed. One of the most popular methods is through the use of a binary semaphore.

### Mutual Exclusion with a Semaphore

A properly functioning multiprocessor system must provide a mechanism that will guarantee orderly access to shared memory and other shared resources. This is necessary to protect data from being changed simultaneously by two or more processors. This mechanism has been termed *mutual exclusion*. Mutual exclusion must be provided in a multiprocessor system to enable one processor to exclude or lock out access to a shared resource by other processors when it is in a *critical section*. A critical section is a program sequence that, once begun, must complete execution before another processor accesses the same shared resource.

*critical section*

A binary variable called a *semaphore* is often used to indicate whether or not a processor is executing a critical section. A semaphore is a software-controlled flag that is stored in a memory location that all processors can access. When the semaphore is equal to 1, it means that a processor is executing a critical program, so that the shared memory is not available to other processors. When the semaphore is equal to 0, the shared memory is available to any requesting processor. Processors that share the same memory segment agree by convention not to use the memory segment unless the semaphore is equal to 0, indicating that memory is available. They also agree to set the semaphore to 1 when they are executing a critical section and to clear it to 0 when they are finished.

Testing and setting the semaphore is itself a critical operation and must be performed as a single indivisible operation. If it is not, two or more processors may test the semaphore simultaneously and then each set it, allowing them to enter a critical section at the same time. This action would allow simultaneous execution of critical section, which can result in erroneous initialization of control parameters and a loss of essential information.

*hardware lock*

A semaphore can be initialized by means of a test and set instruction in conjunction with a hardware *lock* mechanism. A hardware lock is a processor-generated signal that serves to prevent other processors from using the system bus as long as the signal is active. The test-and-set instruction tests and sets a semaphore and activates the lock mechanism during the time that the instruction is being executed. This prevents other processors from changing the semaphore between the time that the processor is testing it and the time that it is setting it. Assume that the semaphore is a bit in the least significant position of a memory word whose address is symbolized by SEM. Let the mnemonic TSL designate the “test and set while locked” operation. The instruction

TSL     SEM

will be executed in two memory cycles (the first to read and the second to write) without interference as follows:

$R \leftarrow M[SEM]$	Test semaphore
$M[SEM] \leftarrow 1$	Set semaphore

The semaphore is tested by transferring its value to a processor register  $R$  and then it is set to 1. The value in  $R$  determines what to do next. If the processor finds that  $R = 1$ , it knows that the semaphore was originally set. (The fact that it is set again does not change the semaphore value.) That means that another processor is executing a critical section, so the processor that checked the semaphore does not access the shared memory. If  $R = 0$ , it means that the common memory (or the shared resource that the semaphore represents) is available. The semaphore is set to 1 to prevent other processors from accessing memory. The processor can now execute the critical section. The last instruction in the program must clear location SEM to zero to release the shared resource to other processors.

Note that the lock signal must be active during the execution of the test-and-set instruction. It does not have to be active once the semaphore is set. Thus the lock mechanism prevents other processors from accessing memory while the semaphore is being set. The semaphore itself, when set, prevents other processors from accessing shared memory while one processor is executing a critical section.

## 13-5 Cache Coherence

---

The operation of cache memory is explained in Sec. 12-6. The primary advantage of cache is its ability to reduce the average access time in uniprocessors. When the processor finds a word in cache during a read operation, the main memory is not involved in the transfer. If the operation is to write, there are two commonly used procedures to update memory. In the *write-through* policy, both cache and main memory are updated with every write operation. In the *write-back* policy, only the cache is updated and the location is marked so that it can be copied later into main memory.

In a shared memory multiprocessor system, all the processors share a common memory. In addition, each processor may have a local memory, part or all of which may be a cache. The compelling reason for having separate caches for each processor is to reduce the average access time in each processor. The same information may reside in a number of copies in some caches and main memory. To ensure the ability of the system to execute memory operations correctly, the multiple copies must be kept identical. This requirement imposes a *cache coherence* problem. A memory scheme is *coherent* if the value returned on a load instruction is always the value given by the latest store instruction with the same address. Without a proper solution to the cache coherence problem, caching cannot be used in bus-oriented multiprocessors with two or more processors.

### Conditions for Incoherence

Cache coherence problems exist in multiprocessors with private caches because of the need to share writable data. Read-only data can safely be replicated

without cache coherence enforcement mechanisms. To illustrate the problem, consider the three-processor configuration with private caches shown in Fig. 13-12. Sometime during the operation an element  $X$  from main memory is loaded into the three processors,  $P_1$ ,  $P_2$ , and  $P_3$ . As a consequence, it is also copied into the private caches of the three processors. For simplicity, we assume that  $X$  contains the value of 52. The load on  $X$  to the three processors results in consistent copies in the caches and main memory.

If one of the processors performs a store to  $X$ , the copies of  $X$  in the caches become inconsistent. A load by the other processors will not return the latest value. Depending on the memory update policy used in the cache, the main memory may also be inconsistent with respect to the cache. This is shown in Fig. 13-13. A store to  $X$  (of the value of 120) into the cache of processor  $P_1$  updates memory to the new value in a write-through policy. A write-through policy maintains consistency between memory and the originating cache, but the other two caches are inconsistent since they still hold the old value. In a write-back policy, main memory is not updated at the time of the store. The copies in the other two caches and main memory are inconsistent. Memory is updated eventually when the modified data in the cache are copied back into memory.

Another configuration that may cause consistency problems is a direct memory access (DMA) activity in conjunction with an IOP connected to the system bus. In the case of input, the DMA may modify locations in main memory that also reside in cache without updating the cache. During a DMA output, memory locations may be read before they are updated from the cache when using a write-back policy. I/O-based memory incoherence can be overcome by making the IOP a participant in the cache coherent solution that is adopted in the system.

### Solutions to the Cache Coherence Problem

Various schemes have been proposed to solve the cache coherence problem in shared memory multiprocessors. We discuss some of these schemes briefly here. See references 3 and 10 for more detailed discussions.

A simple scheme is to disallow private caches for each processor and have a shared cache memory associated with main memory. Every data access is made to the shared cache. This method violates the principle of closeness of CPU to cache and increases the average memory access time. In effect, this scheme solves the problem by avoiding it.

For performance considerations it is desirable to attach a private cache to each processor. One scheme that has been used allows only nonshared and read-only data to be stored in caches. Such items are called *cachable*. *Shared writable data are noncachable*. The compiler must tag data as either cachable or noncachable, and the system hardware makes sure that only cachable data are stored in caches. The noncachable data remain in main memory. This method



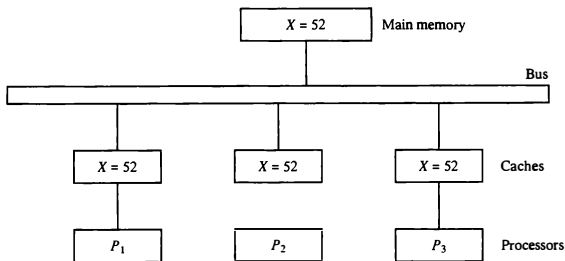
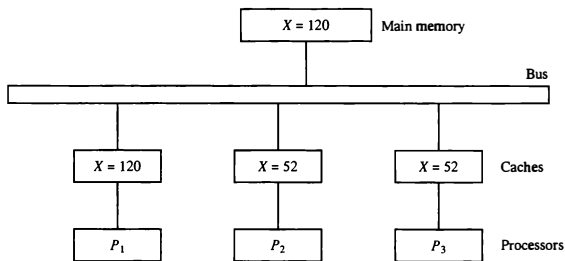
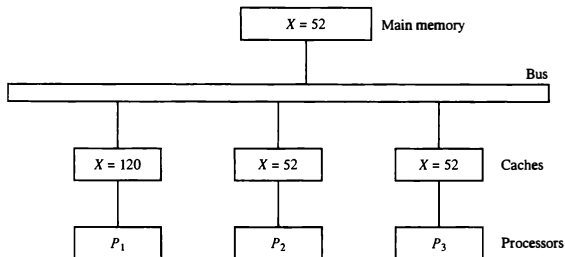


Figure 13-12 Cache configuration after a load on X.

Figure 13-13 Cache configuration after a store to X by processor  $P_1$ .



(a) With write-through cache policy



(b) With write-back cache policy

restricts the type of data stored in caches and introduces an extra software overhead that may degrade performance.

A scheme that allows writable data to exist in at least one cache is a method that employs a *centralized global table* in its compiler. The status of memory blocks is stored in the central global table. Each block is identified as *read-only* (RO) or *read and write* (RW). All caches can have copies of blocks identified as RO. Only one cache can have a copy of an RW block. Thus if the data are updated in the cache with an RW block, the other caches are not affected because they do not have a copy of this block.

The cache coherence problem can be solved by means of a combination of software and hardware or by means of hardware-only schemes. The two methods mentioned previously use software-based procedures that require the ability to tag information in order to disable caching of shared writable data. Hardware-only solutions are handled by the hardware automatically and have the advantage of higher speed and program transparency. In the hardware solution, the cache controller is specially designed to allow it to monitor all bus requests from CPUs and IOPs. All caches attached to the bus constantly monitor the network for possible write operations. Depending on the method used, they must then either update or invalidate their own cache copies when a match is detected. The bus controller that monitors this action is referred to as a *snoopy cache controller*. This is basically a hardware unit designed to maintain a bus-watching mechanism over all the caches attached to the bus.

Various schemes have been proposed to solve the cache coherence problem by means of snoopy cache protocol. The simplest method is to adopt a write-through policy and use the following procedure. All the snoopy controllers watch the bus for memory store operations. When a word in a cache is updated by writing into it, the corresponding location in main memory is also updated. The local snoopy controllers in all other caches check their memory to determine if they have a copy of the word that has been overwritten. If a copy exists in a remote cache, that location is marked invalid. Because all caches snoop on all bus writes, whenever a word is written, the net effect is to update it in the original cache and main memory and remove it from all other caches. If at some future time a processor accesses the invalid item from its cache, the response is equivalent to a cache miss, and the updated item is transferred from main memory. In this way, inconsistent versions are prevented.

**snoopy cache  
controller**

## PROBLEMS

- 13-1. Discuss the difference between tightly coupled multiprocessors and loosely coupled multiprocessors from the viewpoint of hardware organization and programming techniques.