ANNAMACHARYA INSTITUTE OF TECHNOLOGY AND SCIENCES: KADAPA

(AUTONOMOUS)

Utukur(P), C.K. Dinne (V&M), Kadapa, YSR Kadapa Dist., A.P.
Approved by AICTE, New Delhi & Affiliated to JNTUA, Anantapuramu.
Accredited by NAAC with 'A' Grade, Bangalore.

Year & Semester	B. Tech III Year I Semester
Name of the Subject	COMPUTER ARCHITECTURE & ORGANIZATION
Subject Code	23HPE041a
Regulation	HM23

PREPARED BY

Name of the Faculty	Mrs. P. Vijaya
Designation	Assistant Professor
Department	Electronics and Communication Engineering
College Name & Code	Annamacharya Institute of Technology and Sciences: Kadapa & HM

III B.Tech I Sem (E.C.E)

23HPE041a	COMPUTER ARCHITECTURE & ORGANIZATION	L	T	P	C
		3	0	0	3

Course Objectives:

1. To learn the design of various functional units of digital computers and performance issues of computer systems.
2. To understand the basic processing unit and their connections.
3. To get familiar with different types of Data representation and Computer Arithmetic operations.
4. To know about different types of memory and their interconnections.
5. To learn the basics of parallel computing and pipelining.

Course Outcomes: At the end of this course, the students will be able to

1. Learn the design of various functional units of digital computers and performance issues of computer systems.
2. Understand the basic processing unit and their connections.
3. Know about different types of Data representation and Computer Arithmetic operations.
4. Learn about different types of memory and their interconnections.
5. Understand the basics of parallel computing and pipelining.

UNIT I

Digital Computers: Introduction, Block diagram of Digital Computer, Definition of Computer Organization, Computer Design and Computer Architecture.

Register Transfer Language and Micro operations: Register Transfer language, Register Transfer, Bus and memory transfers, Arithmetic Micro operations, logic micro operations, shift micro operations, Arithmetic logic shift unit.

Basic Computer Organization and Design: Instruction codes, Computer Registers Computer instructions, Timing and Control, Instruction cycle, Memory Reference Instructions, Input – Output and Interrupt.

UNIT II

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.

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.

UNIT IV

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

Memory Organization: Memory Hierarchy, Main Memory, Auxiliary memory, Associate Memory, Cache 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, Inter-processor arbitration, Inter-processor communication and synchronization, Cache Coherence.

Textbook:

1. Computer System Architecture – M. Moris Mano, Third Edition, Pearson/PHI.

References:

1. Computer Organization – Car Hamacher, ZvonksVranesic, SafeaZaky, Vth Edition, McGraw Hill.
2. Computer Organization and Architecture – William Stallings Sixth Edition, Pearson/PHI.
3. Structured Computer Organization – Andrew S. Tanenbaum, 4th Edition, PHI/Pearson.

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. ln 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 mi. 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 I. 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 eachbit 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 10010ll 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.

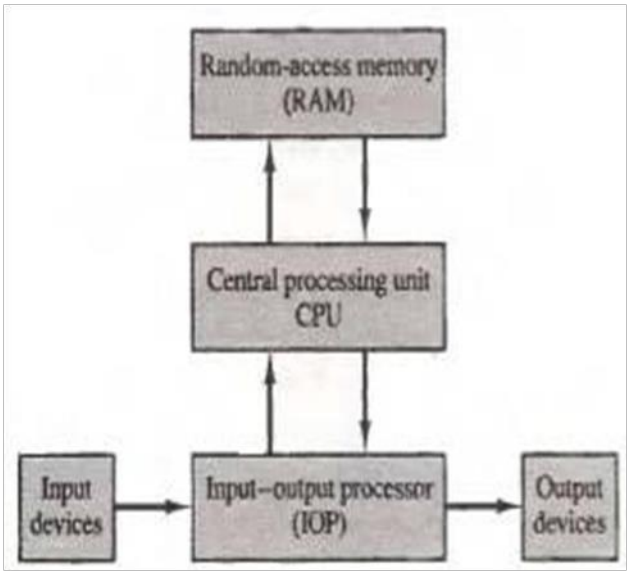
Program

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 dala, a number of registers for storing dala, and control circuits for fetching and executing instuctions. The memory of a computer contains storage for instuctions 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 (lOP) 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

Compute 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 frame work, or structure, that a computer's hardware, programming, and data should follow.

Von Neumann envisioned 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 computers 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 key board for input and display monitor for output.

The instructional 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 sequentially execution of programming imposes a sort of speed limit on program execution, since only one instruction at a time can be handled by the computers 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:

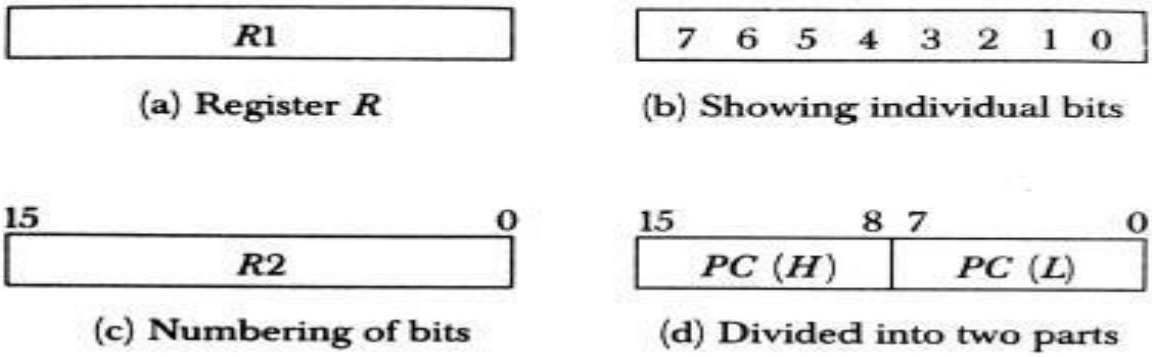
- The symbolic notation used to describe the micro-operation transfer among registers is called RTL (Register Transfer Language).
- The use of ***symbols*** instead of a ***narrative explanation*** provides an organized and concise manner for listing the micro-operation sequences in registers and the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

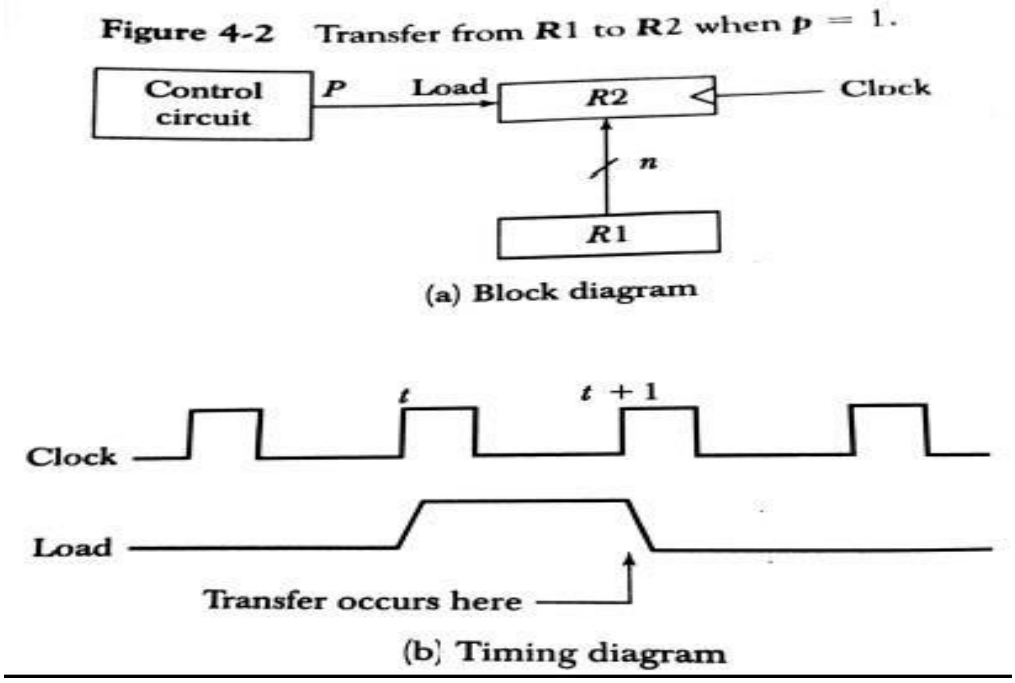


Figure 4-2 Transfer from R1 to R2 when p = 1.

(a) Block diagram

(b) Timing diagram

- 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 <--	Denotes transfer of information	R2 <-- R1
Comma ,	Separates two microoperations	R2 <-- R1, R1 <-- R2

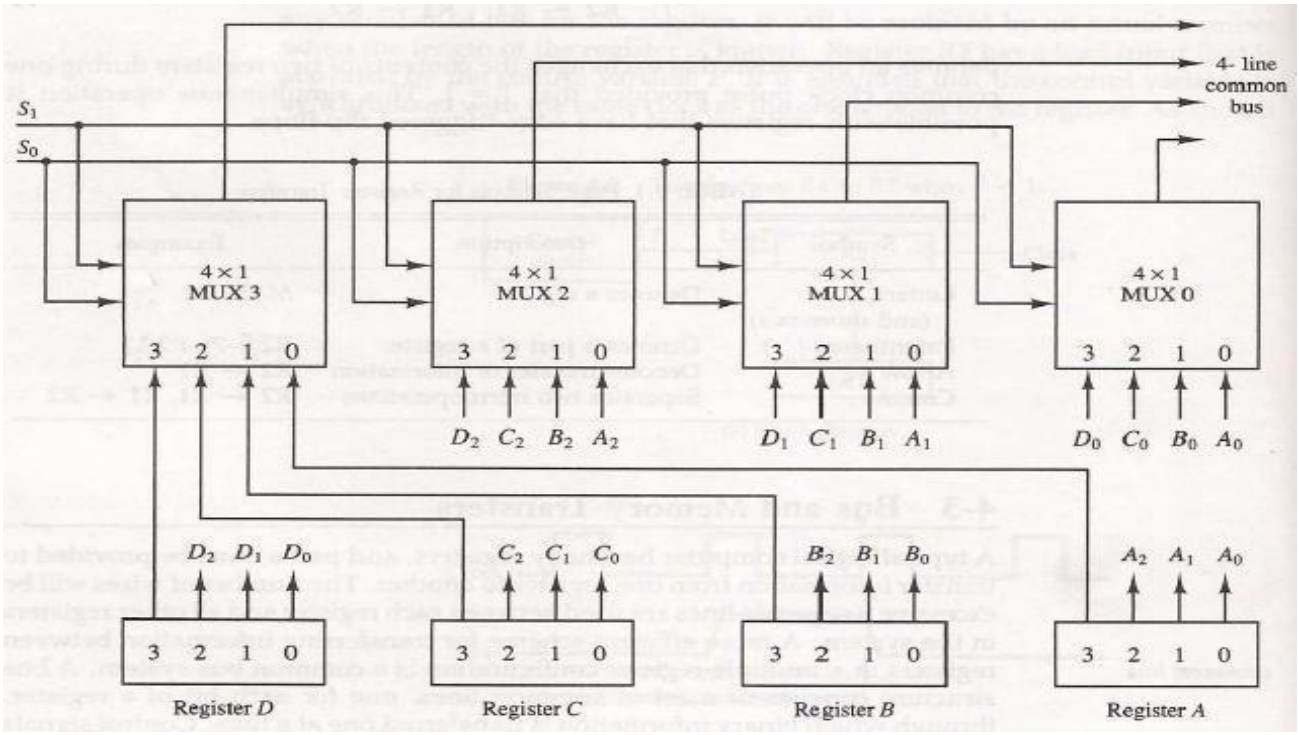
- A comma is used to separate two or more operations that are executed at the same time**.**
- The statement
  T : R2← R1, R1← R2 (exchange operation)
  denotes an operation that exchanges the contents of two rgisters during one common clock pulse provided that T=1.

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 Si and So 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 includes 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

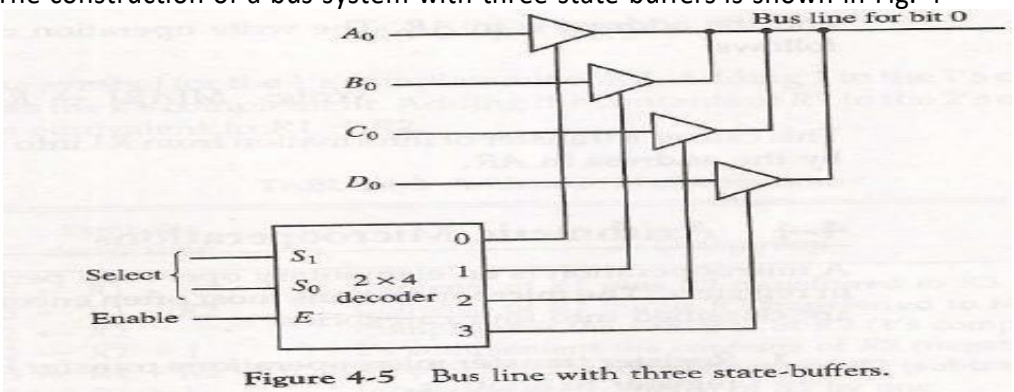


Figure 4-5 Bus line with three state-buffers.

- 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 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 ← 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 ← 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.

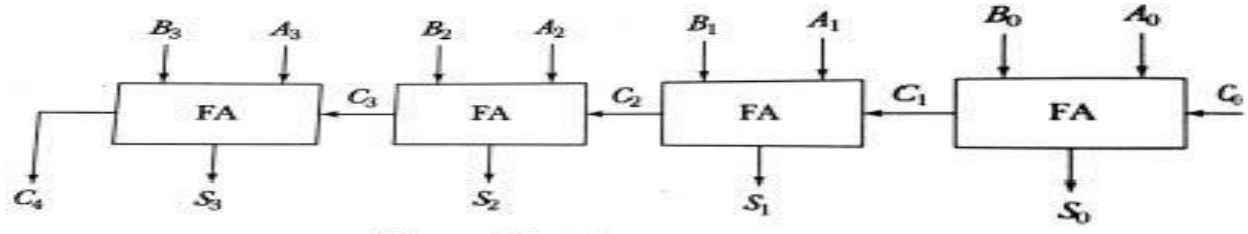


Figure 4-6 4-bit binary adder.

- 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 Co and the output carry is C4. 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.

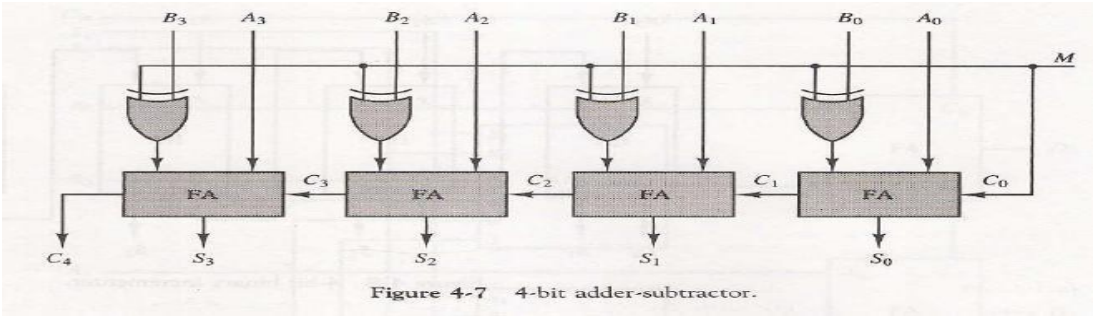


Figure 4-7 4-bit adder-subtractor.

- 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* 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 xor 1 = B'* and Co = 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.

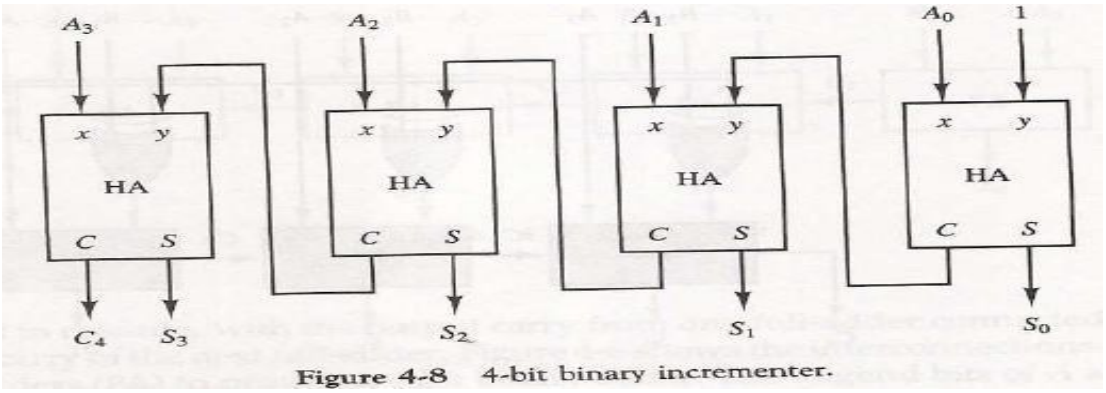


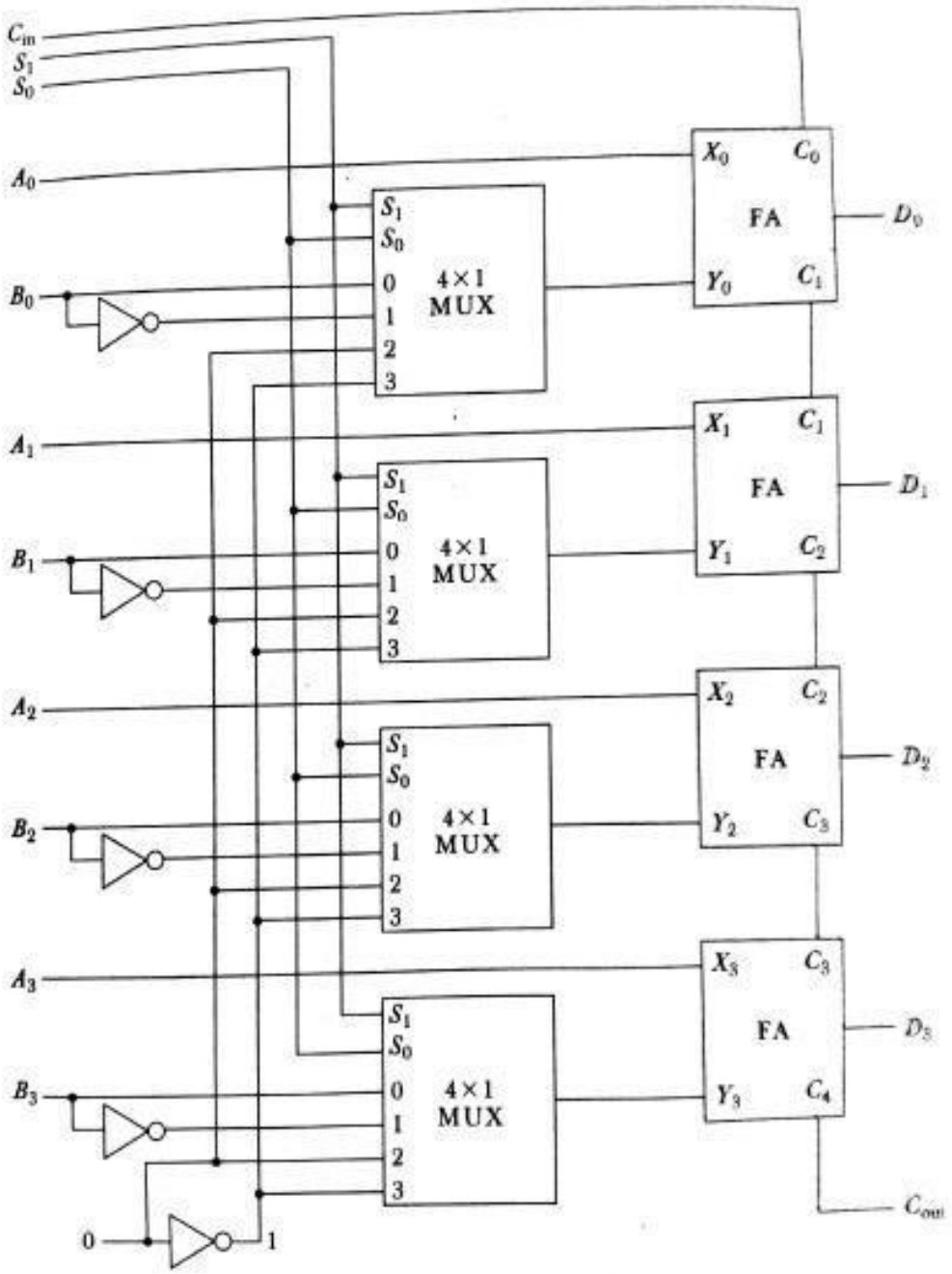
Figure 4-8 4-bit binary incrementer.

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

Addition:

- When S_1S_0= 00, the value of *B* is applied to the Y inputs of the adder.
  - ✓ If Cir, = 0, the output *D =A+B.*
  - ✓ If Cin = 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 + $\overline{B}$ + 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 + $\overline{B}$. 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 l'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 RI and R2 is symbolized by the statement

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

- 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

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

- 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.

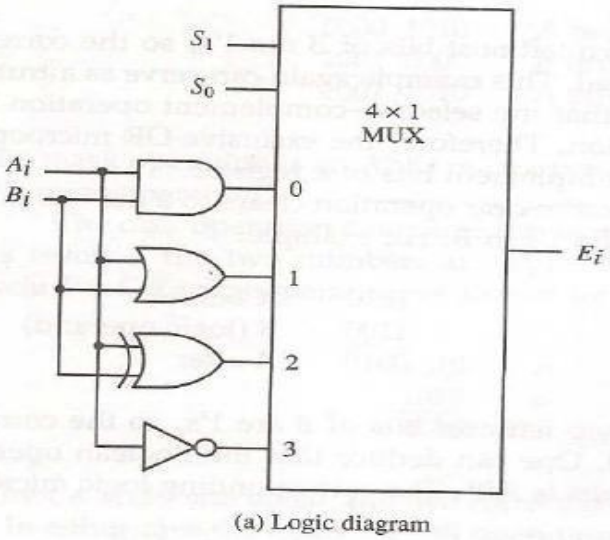
TABLE 4-6 Sixteen Logic Microoperations

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



(a) Logic diagram

S_1	S_0	Output	Operation
0	0	$E = A \wedge B$	AND
0	1	$E = A \vee B$	OR
1	0	$E = A \oplus B$	XOR
1	1	$E = \overline{A}$	Complement

(b) Function table

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 l'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
1100    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
1100    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 l's in B. For example:

```
1010    A before
1100    B (logic operand)
0010    A after
```

  - ✓ The corresponding logic microoperation is $A \leftarrow A \wedge \overline{B}$
- Mask
  - ✓ The *mask* operation is similar to the selective-clear operation except that the bits of A are cleared only where there are corresponding O's in B . The mask operation is an AND micro operation as seen from the following numerical example:

```
0110 1010    A before
0000 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 ← A ⊕ 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
$R \leftarrow \text{shl } R$	Shift-left register R
$R \leftarrow \text{shr } R$	Shift-right register R
$R \leftarrow \text{cil } R$	Circular shift-left register R
$R \leftarrow \text{cir } R$	Circular shift-right register R
$R \leftarrow \text{ashl } R$	Arithmetic shift-left R
$R \leftarrow \text{ashr } R$	Arithmetic shift-right R

- **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.

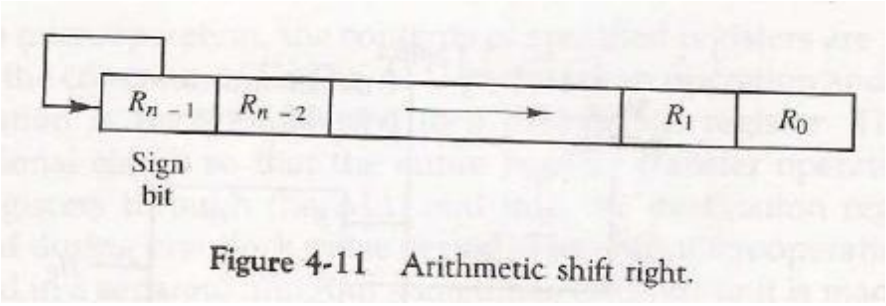
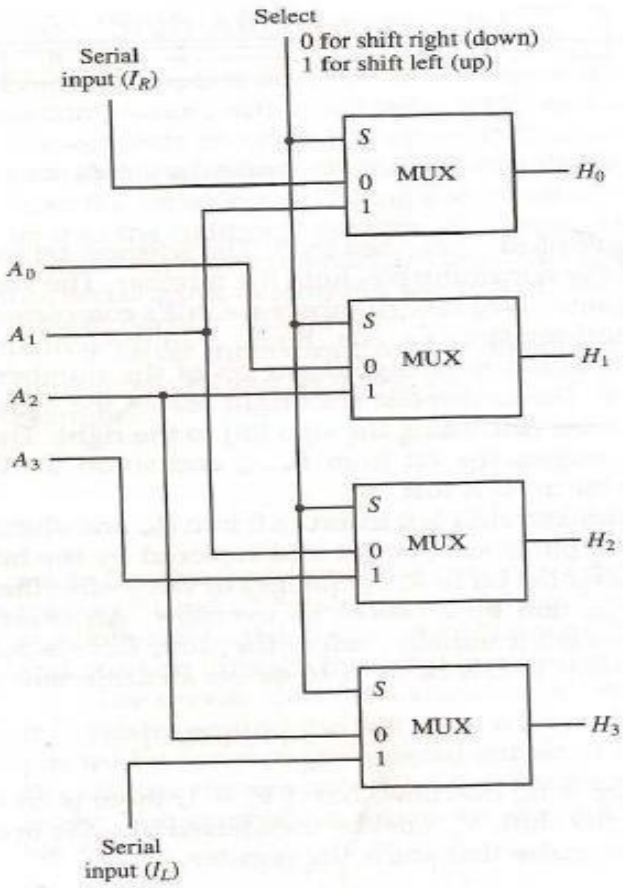


Figure 4-11 Arithmetic shift right.

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.



Function table

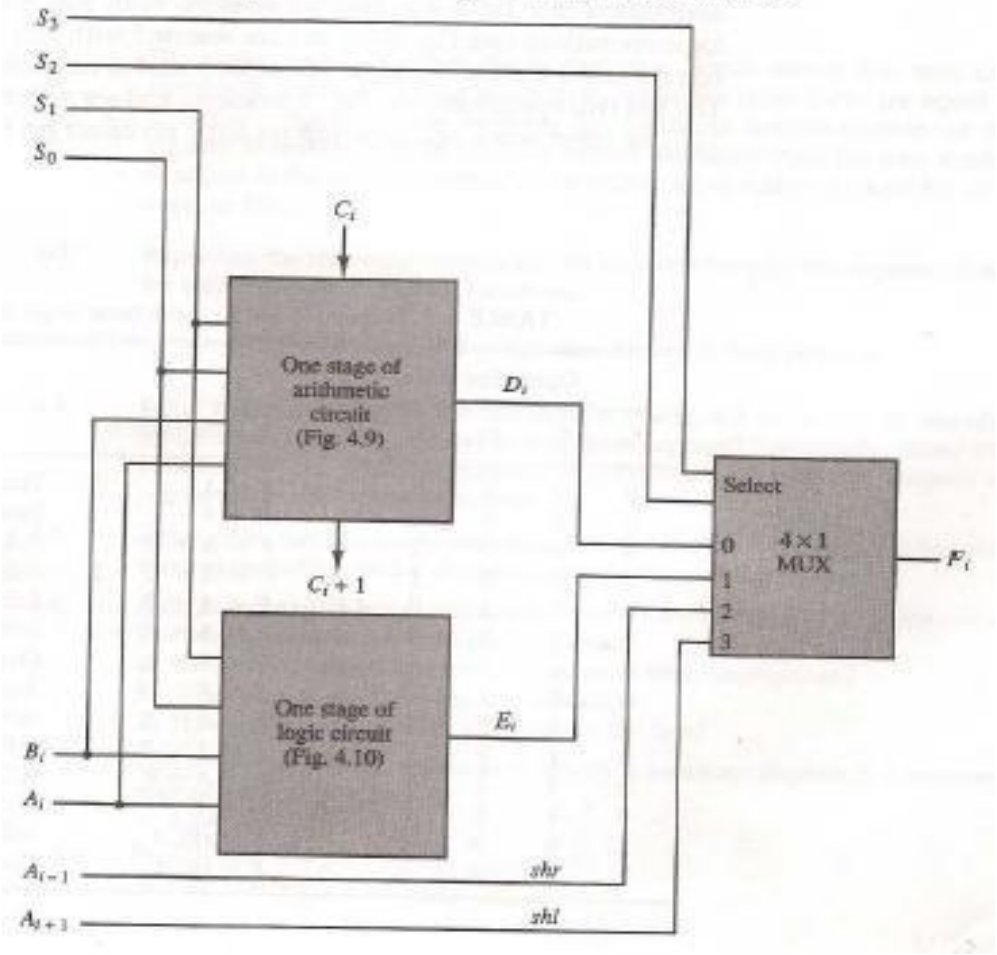
Select	Output			
S	H_0	H_1	H_2	H_3
0	I_R	A_0	A_1	A_2
1	A_1	A_2	A_3	I_L

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						
S_3	S_2	S_1	S_0	C_{in}	Operation	Function
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	×	$F = A \wedge B$	AND
0	1	0	1	×	$F = A \vee B$	OR
0	1	1	0	×	$F = A \oplus B$	XOR
0	1	1	1	×	$F = \overline{A}$	Complement A
1	0	×	×	×	$F =$ shr A	Shift right A into F
1	1	×	×	×	$F =$ shl A	Shift left A into F

UNIT II

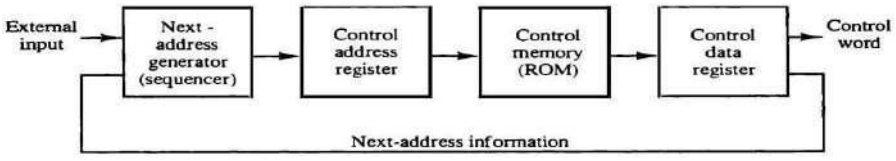
MICRO-PROGRAMMED CONTROL

CONTROL MEMORY

- ✓ The control unit in a digital computer initiates sequences of microoperations
- ✓ The complexity of the digital system is derived from the number of sequences that are performed
- ✓ Two methods of implementing control unit are ***hardwired control*** and ***microprogrammed control***
- ✓ When the control signals are generated by hardware, it is *hardwired.* The design of hardwired control involves the use of fixed instructions; fixed logic blocks of arrays, encoders, decoders etc. The key characteristics of hardwired control logic are high-speed operation, expensive, relatively complex and no flexibility of adding new instructions.
- ✓ The control function that specifies a microoperation is a binary variable. In a bus-oriented system, the control signals that specify microoperations are groups of bits that select the paths in multiplexers, decoders, and ALUs.
- ✓ The control unit initiates a series of sequential steps of microoperations. The control variables can be represented by a string of 1's and 0's called a ***control word.***
- ✓ A ***microprogrammed control unit*** is a control unit whose binary control variables are stored in memory.
- ✓ A sequence of microinstructions constitutes a ***microprogram***
- ✓ The control memory can be a read-only memory
- ✓ *Dynamic* microprogramming permits a microprogram to be loaded and uses a writable control memory
- ✓ A computer with a microprogrammed control unit will have two separate memories:

  main memory and
  control memory
- ✓ The microprogram consists of microinstructions that specify various internal control signals for execution of register microoperations. These microinstructions generate the microoperations to:
  - fetch the instruction from main memory
  - evaluate the effective address
  - execute the operation
  - return control to the fetch phase for the next instruction

Microprogrammed control organization.

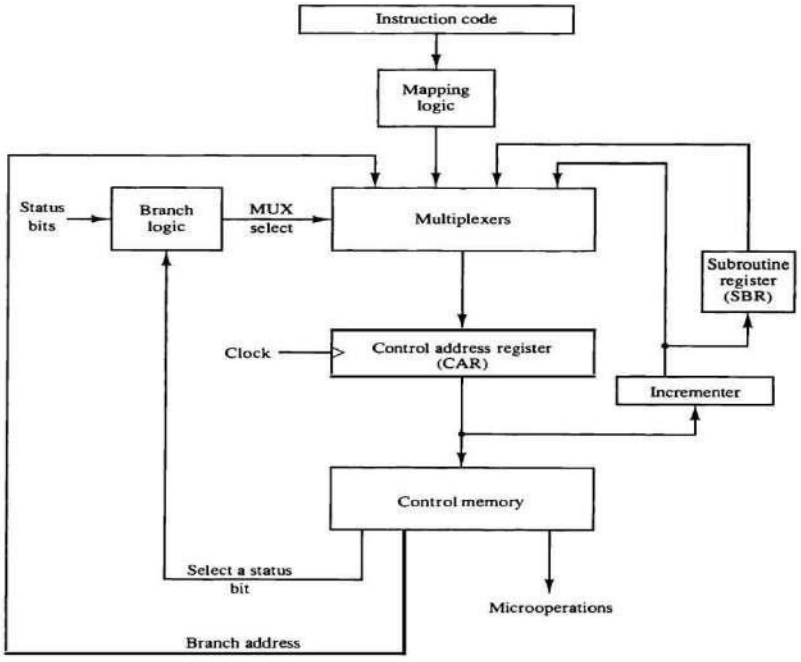


- ✓ The general configuration of a microprogrammed control unit is demonstrated in the block diagram
- ✓ The control memory address register specifies the address of the microinstruction.
- ✓ The control data register holds the microinstruction read from memory.
- ✓ The microinstruction contains a control word that specifies one or more microoperations for the data processor.
- ✓ The location for the next microinstruction may, or may not be the next in sequence. Some bits of the present microinstruction control the generation of the address of the next microinstruction.
- ✓ The next address may also be a function of external input conditions.
- ✓ While the microoperations are being executed, the next address is computed in the next address generator circuit (sequencer) and then transferred into the CAR to read the next microinstructions. Typical functions of a sequencer are:
    - ➢ incrementing the CAR by one
    - ➢ loading into the CAR an address from control memory
    - ➢ transferring an external address
    - ➢ loading an initial address to start the control operations
- ✓ A clock is applied to the CAR and the control word and next-address information are taken directly from the control memory.
- ✓ The address value is the input for the ROM and the control work is the output.No read signal is required for the ROM as in a RAM.
- ✓ The main advantage of the microprogrammed control is that once the hardware configuration is established, there should be no need for hardware or wiring changes. To establish a different control sequence, specify a different set of microinstructions for control memory

ADDRESS SEQUENCING

- ✓ Microinstructions are stored in control memory in groups, with each group specifying a *routine*
- ✓ Each computer instruction has its own microprogram routine to generate the microoperations that execute the instruction.
- ✓ The hardware that controls the address sequencing of the control memory must be capable of sequencing the microinstructions within a routine and be able to branch from one routine to another
- ✓ Steps that the control must undergo during the execution of a single computer instruction:
- ➢ Load an initial address into the CAR when power is turned on in the computer. This address is usually the address of the first microinstruction that activates the instruction fetch routine – IR holds instruction
- ➢ The control memory then goes through the routine to determine the effective address of the operand – AR holds operand address
- ➢ The next step is to generate the microoperations that execute the instruction by considering the opcode and applying a *mapping*
- ➢ After execution, control must return to the fetch routine by executing an unconditional branch
- ➢ In summary, the address sequencing capabilities required in control memory are:

1. Incrementing of the control address register.
2. Unconditional branch or conditional branch, depending on status bit conditions.
3. A mapping process from the bits of the instruction to an address for control memory.
4. A facility for subroutine call and return.



Selection of address for control memory.

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

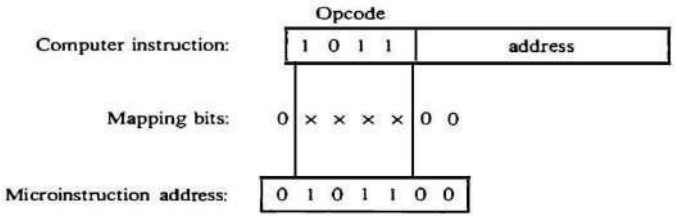
CONDITIONAL BRANCHING

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

MAPPING OF INSTRUCTION

- A special type of branch exists when a microinstruction specifies a branch to the first word in control memory where a microprogram routine is located. The status bits for this type of branch are the bits in the opcode
- Assume an opcode of four bits and a control memory of 128 locations

Mapping from instruction code to microinstruction address.



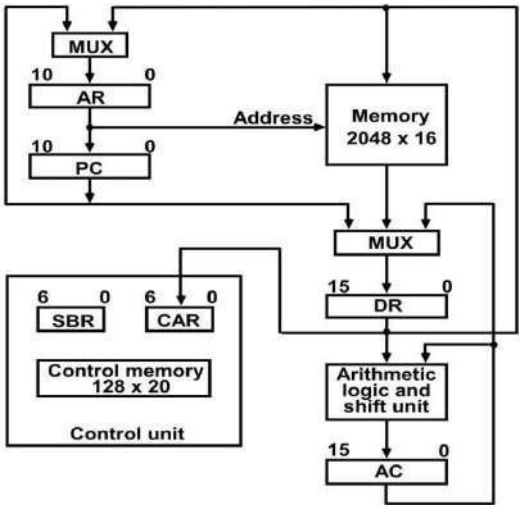
- The mapping process converts the 4-bit opcode to a 7-bit address for control memory
- This provides for each computer instruction a microprogram routine with a capacity of four microinstructions
- The mapping function is sometimes implemented by means of an integrated circuit called programmable logic device or PLD. A PLD is similar to ROM in concept except that it uses AND and OR gates with internal electronic fuses.

SUBROUTINES

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

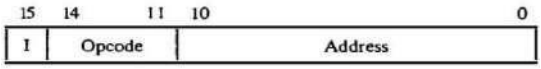
MICROPROGRAM EXAMPLE

- The process of code generation for the control memory is called **microprogramming**
- Two memory units:
  - Main memory – stores instructions and data;
  - Control memory – stores microprogram
- Four processor registers
  - Program counter – PC
  - Address register – AR
  - Data register – DR
  - Accumulator register - AC
- Two control unit registers
  - Control address register – CAR
  - Subroutine register – SBR



- Transfer of information among registers in the processor is through MUXs rather than a bus
- Three fields for an instruction:
  - 1-bit field for indirect addressing
  - 4-bit opcode
  - 11-bit address field

Computer instructions.



(a) Instruction format

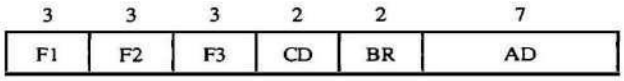
- The example will only consider the following 4 of the possible 16 memory instructions

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

EA is the effective address

(b) Four computer instructions

- The microinstruction format is composed of 20 bits with four parts to it
  - Three fields F1, F2, and F3 specify microoperations for the computer [3 bits each]
  - The CD field selects status bit conditions [2 bits]
  - The BR field specifies the type of branch to be used [2 bits]
  - The AD field contains a branch address [7 bits]



F1, F2, F3: Microoperation fields

CD: Condition for branching

BR: Branch field

AD: Address field

Microinstruction code format (20 bits).

- Each of the three microoperation fields can specify one of seven possibilities
- No more than three microoperations can be chosen for a microinstruction
- If fewer than three are needed, the code 000 = NOP

DR ← M[AR], PC ← PC +1 F2 = 100 and F3 = 101

F1 F2 F3 = 000 100 101

- Five letters to specify a transfer-type microoperation

- First two designate the source register
  - Third is a 'T'
  - Last two designate the destination register

AC ← DR F1 = 100 = DRTAC

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	$AR \leftarrow DR(0\text{–}10)$	DRTAR
110	$AR \leftarrow PC$	PCTAR
111	$M[AR] \leftarrow DR$	WRITE

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

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

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

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

- Each line of an assembly language microprogram defines a symbolic microinstruction and is divided into five parts

  1. The label field may be empty or it may specify a symbolic address. Terminate with a colon
  2. The microoperations field consists of 1-3 symbols, separated by commas. Only one symbol from each F field. If NOP, then translated to 9 zeros
  3. The condition field specifies one of the four conditions
  4. The branch field has one of the four branch symbols
  5. The address field has three formats
     a. A symbolic address – must also be a label
     b. The symbol NEXT to designate the next address in sequence
     c. Empty if the branch field is RET or MAP and is converted to 7 zeros

- The symbol ORG defines the first address of a microprogram routine

THE FETCH ROUTINE

- The control memory has 128 locations, each one is 20 bits
- The first 64 locations are occupied by the routines for the 16 instructions, addresses 0-63
- A convenient starting location for the fetch routine is address 64.
- ORG 64 – places first microinstruction at control memory 1000000
- The fetch routine requires the following three microinstructions (locations 64-66)

  AR ← PC

  DR ← M[AR], PC ← PC +1

  AR ← DR(0-10), CAR(2-5) ← DR(11-14), CAR(0,1,6) ← 0

- The symbolic microprogram for the fetch routine as follows:

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

The translation of the symbolic microprogram to biary produces the following binary microprogram.

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

Horizontal versus vertical microprogramming

Microprogram control units used across various computers can broadly be classified into horizontal and vertical microprogram controls.

Each of these two types of microprogram control has its own advantages and disadvantages.

→ In the microprogrammed control unit, all the control signals associated with microoperations are stored in special memory called control memory.

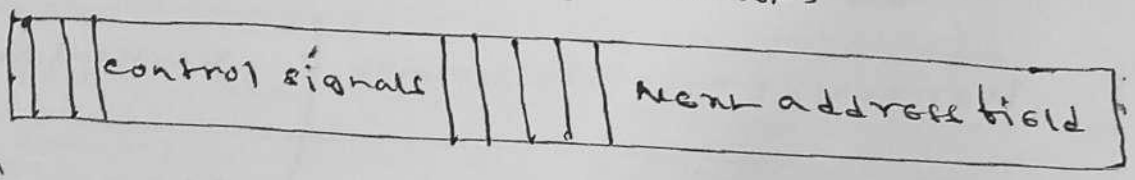
→ set of control signals that cause microoperations to occur is called microinstructions.

→ Microprogrammed control unit can be classified into two types based on the type of control word (microinstructions) stored in the control memory.

① Horizontal micro-programmed control unit

② vertical micro-Programmed control unit

→ In horizontal microprogrammed control unit the control signals are represented in the decoded binary format.



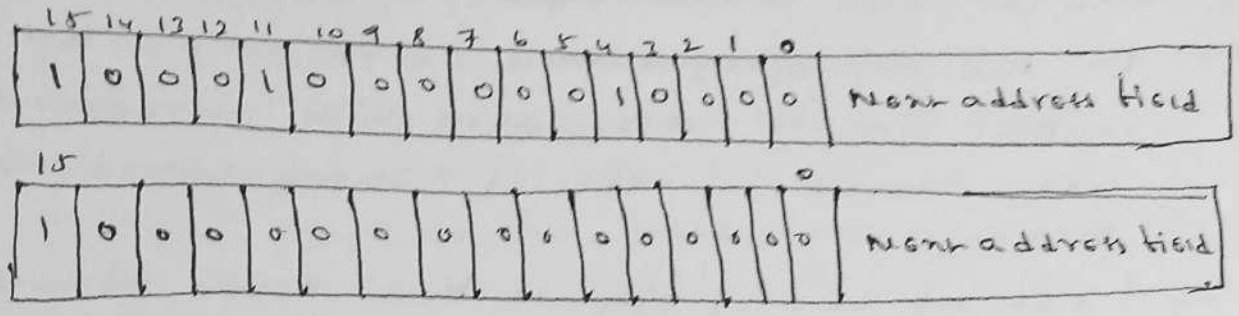
'n' control signals require 'n' bit encoding i.e 1 bit per control signal.

→ In vertical micro-programmed control unit, the control signals are represented in the encoded binary format. 'n' control signals require $\geq \log_2 n$ bit encoding.

Eg: If 16 control signals are needed in a system then $\log_2 16$ bits required to represent control signal part in microinstruction.

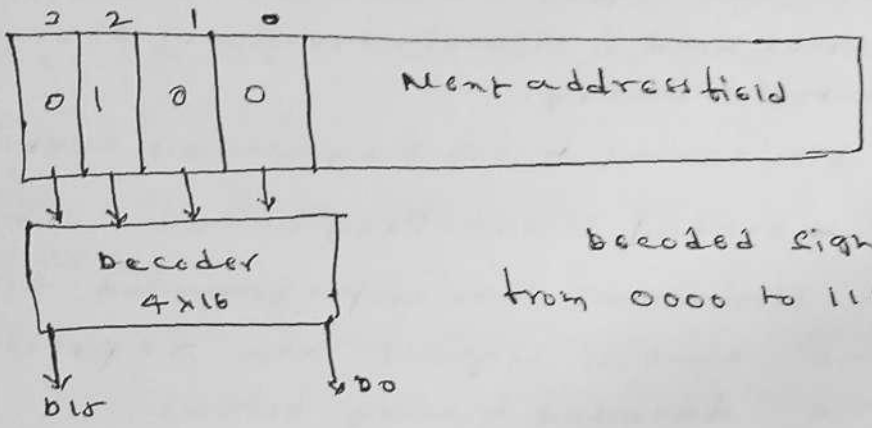
$\log_2 16 = \log_2 2^4 = 4 \log_2 2 = 4$ bits.

Horizontal microinstructions



Longer control word, higher degree of parallelism

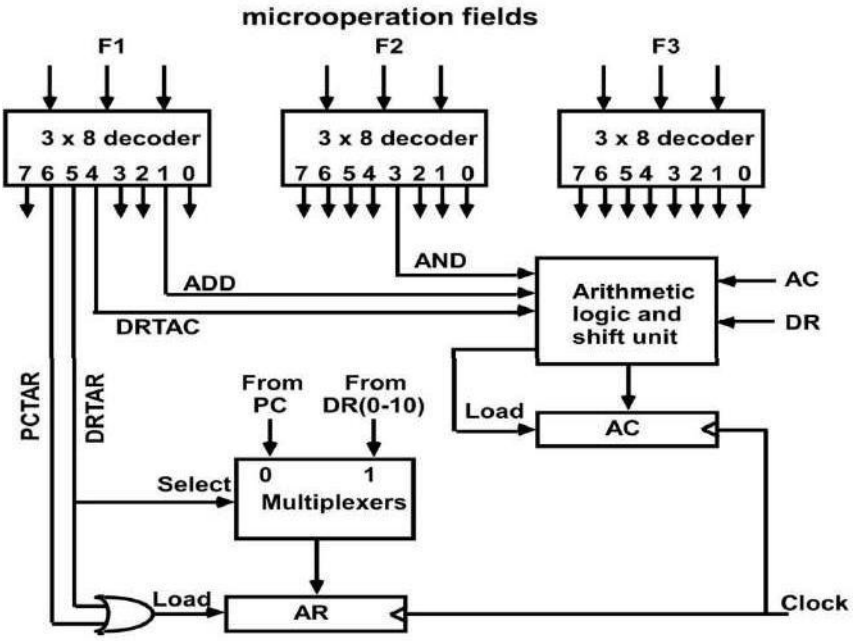
Vertical microinstructions



Horizontal microinstruction	Vertical microinstruction
① Longtery instructions	Shorter instructions
② Degree of parallelism is high	Low degree of parallelism.
③ No external hardware like decoder is not required to generate control signals	Decoder unit is needed to generate control signals.
④ Faster instructions	Comparatively slower instructions.
⑤ Requires large control memory space.	Less control memory space is required.

DESIGN OF CONTROL UNIT

- The bits of the microinstruction are usually divided into fields, with each field defining a distinct, separate function. The various fields encountered in instruction formats provide control bits to initiate microoperations in the system, special bits to specify the way that the next address is to be evaluated, and an address field for branching.
- The control memory output of each subfield must be decoded to provide the distinct microoperations. The outputs of the decoders are connected to the appropriate inputs in the processor unit.

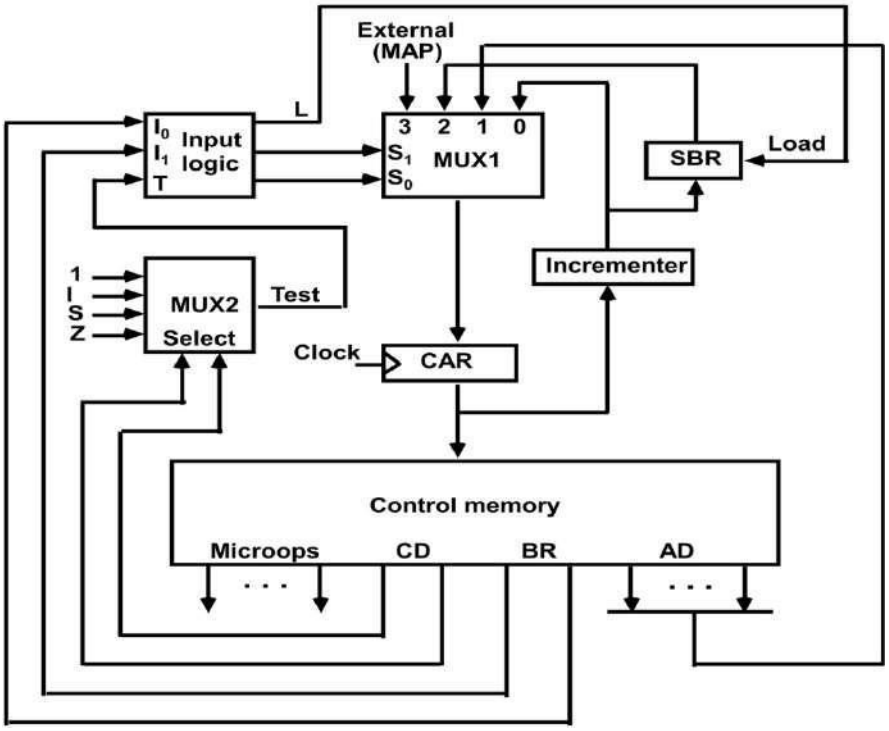


Decoding of Microoperation Fields

- Each of the three fields of the microinstruction presently available in the output of control memory are decoded with a 3 × 8 decoder to provide eight outputs. Each of these outputs must be connected to the proper circuit to initiate the corresponding microoperation
- For example, when F1 = 101 (binary 5), the next clock pulse transition transfers the content of DR (0-10) to AR (symbolized by DRTAR). Similarly, when F1 = 101 (binary 6) there is a transferfrom PC to AR (symbolized by PCTAR).
- The multiplexers select the information from DR when output 5 is active and from PC when output 5 in inactive. The transfer into AR occurs with a clock pulse transition only when output 5 or output 6 of the decoder are active.
- The inputs for arithmetic logic shift unit come from the outputs of the decoders associated with the symbols AND, ADD, and DRTAC.

MICROPROGRAM SEQUENCER

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



- There are two multiplexers in the circuit. The first multiplexer selects an address from one of four sources and routes it into a control address register CAR. The second multiplexer tests the value of a selected status bit and the result of the test is applied to an input logic circuit.
- The output from CAR is incremented and applied to one of the multiplexer inputs and to the subroutine register SBR. The other three inputs to multiplexer number 1 come from the address field of the present microinstruction, from the output of SBR, and from an external source that maps the instruction.
- The CD (condition) field of the microinstruction selecting one of the status bits in the second multiplexer. If the bit selected is equal to 1, the T (test) variable is equal to 1; otherwise, it is equal to 0.

- The T value together with the two bits from the BR (branch) field go to an input logic circuit. The input logic in a particular sequencer will determine the type of operations that are available in the unit. Typical sequencer operations are: increment, branch or jump, call and return from subroutine, load an external address, push or pop the stack, and other address sequencing operations.
- The input logic circuit has three inputs, I0, I1, and T, and three outputs, S0, S1 and L. variables S0 and S1 select one of the source addresses for CAR. Variable L enables the load input in SBR.
- The binary values of the two selection variables determine the path in the multiplexer. For example, with S1 S0 = 10, multiplexer input number 2 is selected and establishes a transfer path from SBR to CAR.
- The truth table can be used to obtain the simplified Boolean functions for the input logic:

$$S_1 = I_1$$
$$S_0 = I_1 I_0 + I_1'T$$
$$L = I_1' I_0 T$$

I_1I_0T	Meaning	Source of Address	S_1S_0	L
000	In-Line	CAR+1	00	0
001	JMP	CS(AD)	01	0
010	In-Line	CAR+1	00	0
011	CALL	CS(AD) and SBR <- CAR+1	01	1
10x	RET	SBR	10	0
11x	MAP	DR(11-14)	11	0

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 below.

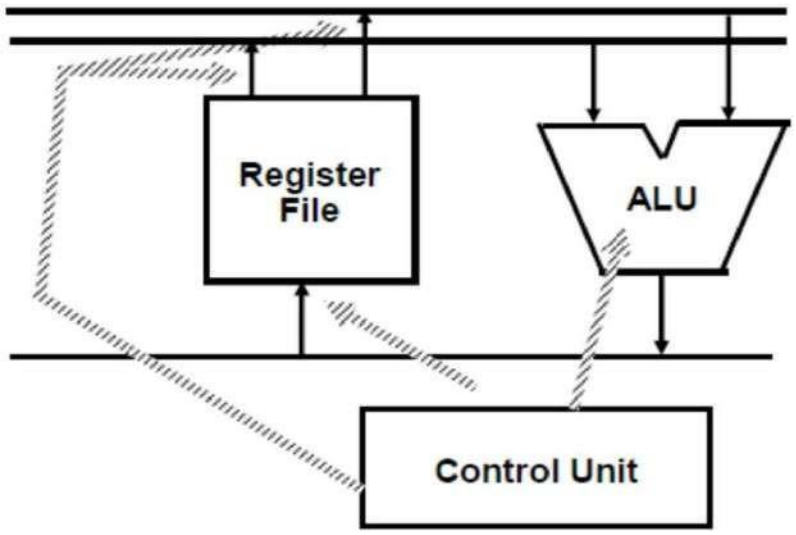


Fig: Major components of CPU.

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

- ✓ The CPU performs a variety of functions dictated by the type of instructions that are incorporated in the computer. Computer architecture is sometimes defined as the computer structure and behavior as seen by the programmer that uses machine language instructions.
- ✓ One boundary where the computer designer ad the computer programmer sees the same machine is the part of the CPU associated with the instruction set.
- ✓ From the designer's point of view, the computer instruction set provides the specifications for the design of the CPU. The design of a CPU is a task that in large part involves choosing the hardware for implementing the machine instructions.
- ✓ The user who programs the computer in machine or assembly language must be aware of the register set, the memory structure, instruction performs.

GENERAL REGISTER ORGANIZATION

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

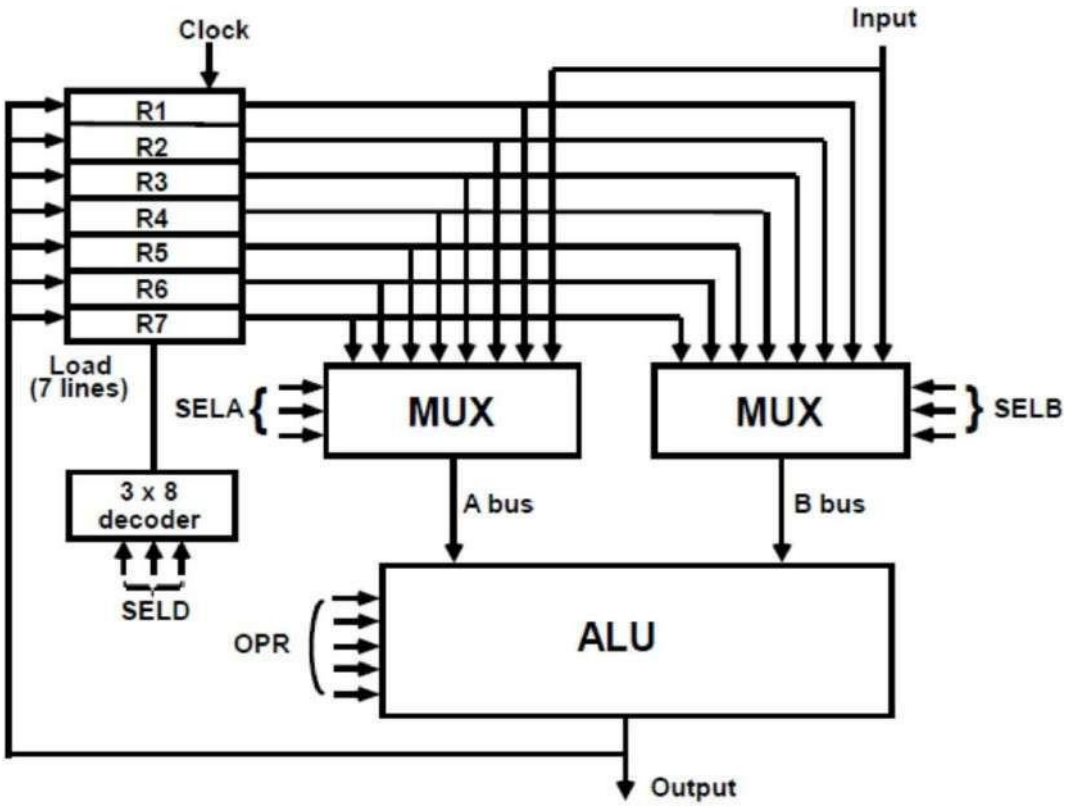


Fig: A bus organization for seven CPU registers

EXAMPLE:

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

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

Table: 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

Table: 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

CONTROL WORD

- ✓ The combined value of a binary selection inputs specifies the control word.
- ✓ It consists of four fields SELA, SELB, and SELD contains three bit each and SELOPR field contains four bits thus the total bits in the control word are 13-bits.

Fig: Format of Control Word

- ✓ The three bit of SELA select a source registers of the A input of the ALU.
- ✓ The three bits of SELB select a source registers of the B input of the ALU.
- ✓ The three bits of SELED select a destination register using the decoder.
- ✓ The four bits of SELOPR select the operation to be performed by ALU.

STACK ORGANIZATION

✓ A stack or last-in first-out (LIFO) is useful feature that is included in the CPU of most computers.

STACK

- ➢ A stack is a storage device that stores information in such a manner that the item stored last is the first item retrieved.

✓ The operation of a stack can be compared to a stack of trays. The last tray placed on top of the stack is the first to be taken off.

✓ In the computer stack is a memory unit with an address register that can count the address only.

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

✓ The two operations that are performed on stack are the insertion and deletion.

✓ The operation of insertion is called PUSH.

✓ The operation of deletion is called POP.

✓ These operations are simulated by incrementing and decrementing the stack pointer register (SP).

REGISTER STACK

✓ A stack can be placed in a portion of a large memory or it can be organized as a collection of a finite number of memory words or registers.

✓ The below figure shows the organization of a 64-word register stack.

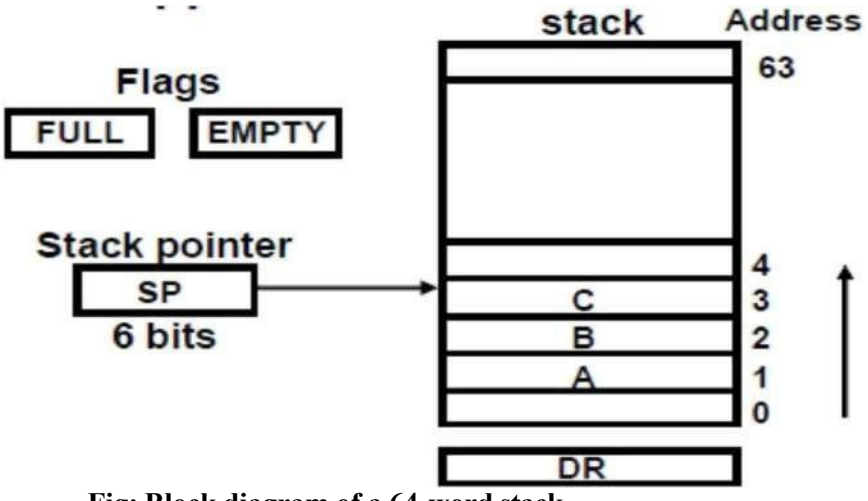


Fig: Block diagram of a 64-word stack

✓ The stack pointer register SP contains a binary number whose value is equal to the address of the word is currently on top of the stack. Three items are placed in the stack: A, B, C, in that order.

✓ In above figure is on top of the stack so that the content of SP is 3.

- ✓ For removing the top item, the stack is popped by reading the memory word at address 3 and decrementing the content of stack SP.
- ✓ Now the top of the stack is B, so that the content of SP is 2.
- ✓ Similarly for inserting the new item, the stack is pushed by incrementing SP and writing a word in the next higher location in the stack.
- ✓ In a 64-word stack, the stack pointer contains 6 bits because $2^6 = 64$.
- ✓ Since SP has only six bits, it cannot exceed a number greater than 63 (111111 in binary).
- ✓ When 63 is incremented by 1, the result is 0 since 111111 + 1. = 1000000 in binary, but SP can accommodate only the six least significant bits.
- ✓ Then the one-bit register FULL is set to 1, when the stack is full.
- ✓ Similarly when 000000 is decremented by 1, the result is 111111, and then the one-bit register EMTY is set 1 when the stack is empty of items.
- ✓ DR is the data register that holds the binary data to be written into or read out of the stack.

PUSH

- ✓ Initially, SP is cleared to 0, EMTY is set to 1, and FULL is cleared to 0, so that SP points to the word at address 0 and the stack is marked empty and not full.
- ✓ If the stack is not full (if FULL = 0), a new item is inserted with a push operation.
- ✓ The push operation is implemented with the following sequence of microoperations:

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

- ✓ The stack pointer is incremented so that it points to the address of next-higher word.
- ✓ A memory write operation inserts the word from DR the top of the stack.
- ✓ The first item stored in the stack is at address 1.
- ✓ The last item is stored at address 0.
- ✓ If SP reaches 0, the stack is full of items, so FULL is to 1.
- ✓ This condition is reached if the top item prior to the last push way location 63 and, after incrementing SP, the last item is stored in location 0.
- ✓ Once an item is stored in location 0, there are no more empty registers in the stack, so the EMTY is cleared to 0.

POP

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

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

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

MEMORY STACK

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

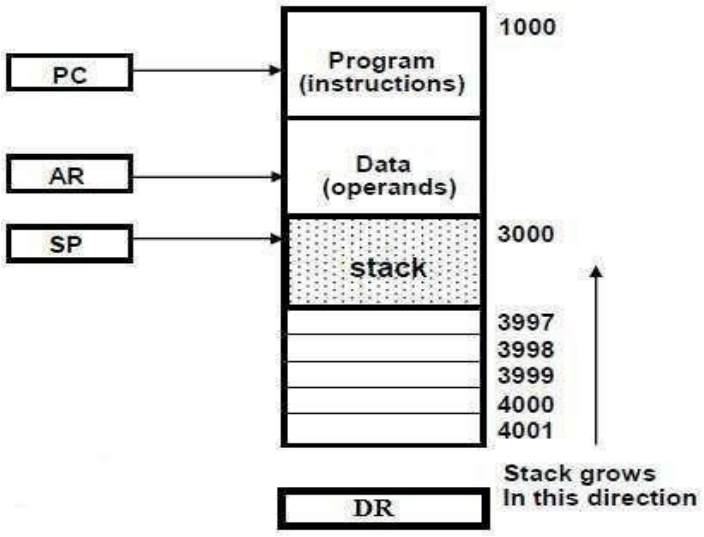


Fig: Computer memory with program, data, and stack segments.

- ✓ The program counter PC points at the address of the next instruction in program.
- ✓ The address register AR points at an array of data.
- ✓ The stack pointer SP points at the top of the stack.
- ✓ The three registers are connected to a common address bus, and either one can provide an address for memory.
  - ➢ PC is used during the fetch phase to read an instruction.
  - ➢ AR is used during the exec phase to read an operand.
  - ➢ SP is used to push or pop items into or from stack.
- ✓ As above fig shown the initial value of SP is 4001 and the stack grows with decreasing addresses.
- ✓ Thus the first item stored in the stack is at address 4000, the second item is stored at address 3999, and the last address that can be used for the stack is 3000.
- ✓ No provisions are available for stack limit checks.
- ✓ The items in the stack communicate with a data register DR. A new item is inserted with the push operation as follows:
  - ➢ SP SP-1
  - ➢ M [SP] DR
- ✓ The stack pointer is decremented so that it points at the address of the next word.
- ✓ A memory write operation inserts the word from DR into the top of stack. A new item is deleted with a pop operation as follows:
  - ➢ DR M [SP]
  - ➢ SP SP+1
- ✓ The top item is read from the stack into DR. The stack pointer is then decremented to point at the next item in the stack.
- ✓ Most computers do not provide hardware to check for stack overflow (full stack) or underflow (empty stack).
- ✓ The stack limits can be checked by using processor registers:
  - ➢ One to hold the upper limit (3000 in this case)
  - ➢ Other to hold the lower limit (4001 in this case).
- ✓ After a push operation, SP compared with the upper-limit register and after a pop operation, SP is a compared with the lower-limit register.
- ✓ The two microoperations needed for either the push or pop are
- ➢ An access to memory through SP
- ➢ Updating SP.

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

REVERSE POLISH NOTATION

- ✓ A stack organization is very effective for evaluating arithmetic expressions.
- ✓ The common arithmetic expressions are written in infix notation, with each operator written between the operands.
- ✓ Consider the simple arithmetic expression.

  A*B+C*D

- ✓ For evaluating the above expression it is necessary to compute the product A*B, store this product result while computing C*D, and then sum the two products.
- ✓ For doing this type of infix notation, it is necessary to scan back and forth along the expression to determine the next operation to be performed.
- ✓ The Polish mathematician Lukasiewicz showed that arithmetic expression can be represented in prefix notation.
- ✓ This representation, often referred to as Polish notation, places the operator before the operands. So it is also called as prefix notation.
- ✓ The Postfix notation, referred to as reverse polish notation (RPN), places the operator after the operands.
- ✓ The following examples demonstrate the three representations
  - ➢ Eg:
    - ➢ A+B----- > Infix notation

      +AB-------> Prefix or Polish notation
    - ➢ AB+--------> Post or reverse Polish notation
- ✓ The reverse Polish notation is in a form suitable for stack manipulation. The expression
  - ➢ A*B+C*D
- ✓ Is written in reverse polish notation as
  - ➢ AB* CD* +
- ✓ And it is evaluated as follows
  - ➢ Scan the expression from left to right.
  - ➢ When operator is reached, perform the operation with the two operands found on the left side of the operator.
  - ➢ Remove the two operands and the operator and replace them by the number obtained from the result of the operation.

- Continue to scan the expression and repeat the procedure for every operation encountered until there are no more operators.

✓ For the expression above it find the operator * after A and B. So it perform the operation A*B and replace A, B and * with the result.

✓ The next operator is a * and it previous two operands are C and D, so it perform the operation C*D and places the result in places C, D and *.

✓ The next operator is + and the two operands to be added are the two products, so we add the two quantities to obtain the result.

✓ The conversion from infix notation to reverse Polish notation must take into consideration the operational hierarchy adopted for infix notation.

✓ This hierarchy dictates that we first perform all arithmetic inside inner parentheses, then inside outer parentheses, and do multiplication and division operations before addition and subtraction operations.

EVALUATION OF ARITHMETIC EXPRESSIONS

✓ Reverse Polish notation, combined with a stack arrangement of registers, is the most efficient way known for evaluating arithmetic expressions.

✓ This procedure is employed in some electronic calculators and also in some computer.

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

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

✓ In reverse polish notation, it is expressed as

- 34 * 56* +

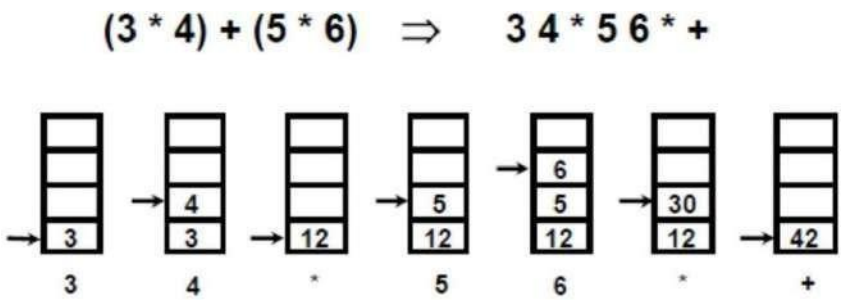


Fig: Stack operations to evaluate (3 * 4 + 5 * 6)

✓ Each box represents one stack operation and the arrow always points to the top of the stack.

✓ Scanning the expression from left to right, we encounter two operands.

✓ First the number 3 is pushed into the stack, then the number 4.

✓ The next symbol is the multiplication operator *.

- ✓ This causes a multiplication of the two top most items the stack.
- ✓ The stack is then popped and the product is placed on top of the stack, replacing the two original operands.
- ✓ Next we encounter the two operands 5 and 6, so they are pushed into the stack.
- ✓ The stack operation results from the next * replaces these two numbers by their product.
- ✓ The last operation causes an arithmetic addition of the two topmost numbers in the stack to produce the final result of 42.

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 ← A [C] + 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.

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:

- ➢ 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.
- ➢ 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:

Effective address =address part of instruction + 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 below

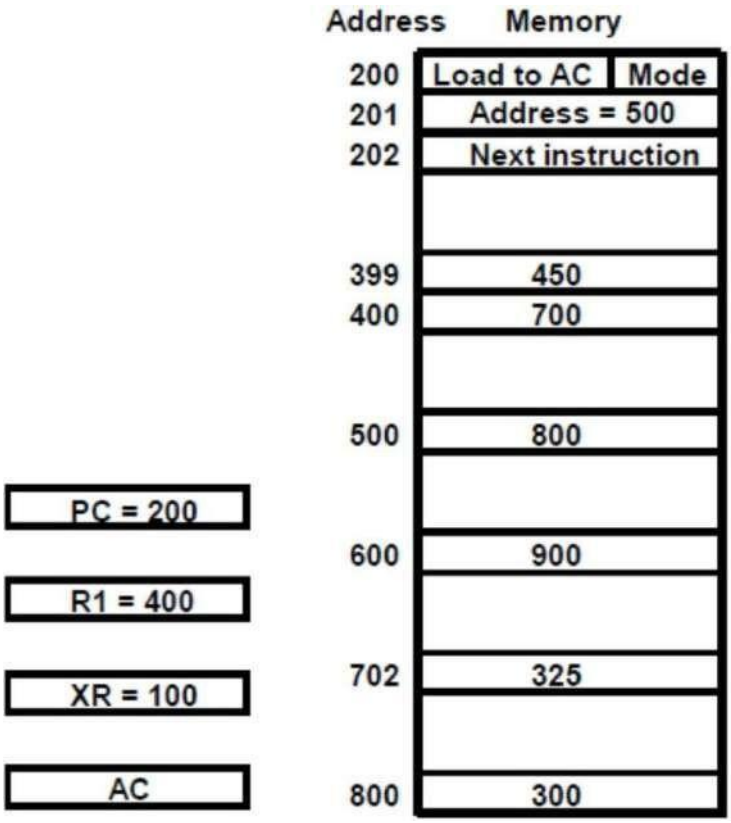


Fig: Numerical example for addressing mode.

✓ 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 below lists the values of the effective address and the operand loaded into AC for the nine addressing modes.

Table: Tabular List of Numerical Example

Addressing Mode	Effective Address			Content of AC
Direct address	500	/* AC ← (500)	*/	800
Immediate operand	-	/* AC ← 500	*/	500
Indirect address	800	/* AC ← ((500))	*/	300
Relative address	702	/* AC ← (PC+500)	*/	325
Indexed address	600	/* AC ← (RX+500)	*/	900
Register	-	/* AC ← R1	*/	400
Register indirect	400	/* AC ← (R1)	*/	700
Autoincrement	400	/* AC ← (R1)+	*/	700
Autodecrement	399	/* AC ← -(R)	*/	450

DATA TRANSFER AND MANIPULATION

✓ Most computer instructions can be classified into three categories:

- ➢ Data transfer instructions
- ➢ Data manipulation instructions
- ➢ 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 below gives a list of eight data transfer instructions used in many computers.

Table: 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 below shows the recommended assembly language convention and actual transfer accomplished in each case

Table: Data Transfer Instructions with Different Addressing Modes

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

- ➢ 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:
  - ➢ Arithmetic instructions

- Logical and bit manipulation instructions
- Shift instructions

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 below.

Table: 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.

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 below.

Table: 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.

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 below lists four types of shift instructions.

Table: 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 thru carry	RORC
Rotate left thru 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.

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 below.

Table: Typical Program Control Instructions

Name	Mnemonic
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RTN
Compare(by –)	CMP
Test(by AND)	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 circuits in the CPU have status register for storing the status bit conditions.
- ✓ Status bits are also called condition-code bits or flag bits.
- ✓ Figure below shows block diagram of an 8-bit ALU with a 4-bit status register.

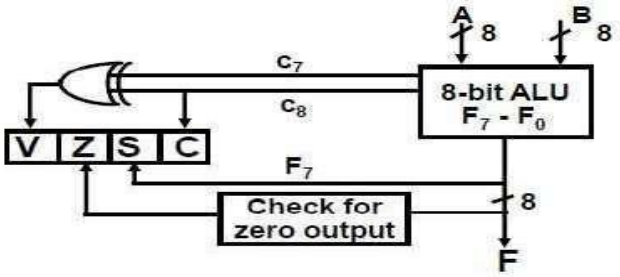


Fig: Status Flag Circuit

✓ 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 C8 is 1. It is cleared to 0 if the carry is 0.
- ➢ S (sign) is set to 1 if the highest-order bit F7 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.

CONDITIONAL BRANCH INSTRUCTIONS

✓ Table below gives a list of the most common branch instructions. Each mnemonic is constructed with the letter B (for branch) and an abbreviation of the condition name.

Mnemonic	Branch condition	Tested condition
BZ	Branch if zero	Z = 1
BNZ	Branch if not zero	Z = 0
BC	Branch if carry	C = 1
BNC	Branch if no carry	C = 0
BP	Branch if plus	S = 0
BM	Branch if minus	S = 1
BV	Branch if overflow	V = 1
BNV	Branch if no overflow	V = 0
	Unsigned compare conditions (A - B)	
BHI	Branch if higher	A > B
BHE	Branch if higher or equal	A ≥ B
BLO	Branch if lower	A < B
BLOE	Branch if lower or equal	A ≤ B
BE	Branch if equal	A = B
BNE	Branch if not equal	A ≠ B
	Signed compare conditions (A - B)	
BGT	Branch if greater than	A > B
BGE	Branch if greater or equal	A ≥ B
BLT	Branch if less than	A < B
BLE	Branch if less or equal	A ≤ B
BE	Branch if equal	A = B
BNE	Branch if not equal	A ≠ B

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
  - ➢ 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
  - ➢ 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 ← SP – 1	Decrement stack pointer
M [SP] ← PC	Push content of PC onto the stack
PC ← effective address	Transfer control to the subroutine

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

PC ← M [SP]	Pop stack and transfer to PC
SP ← 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 form 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.

UNIT-III

Data Types:

- ✓ Data and instructions cannot be entered and processed directly into the computers using human language
- ✓ Any type of data must first be converted into machine-readable form i.e. binary form.
- ✓ 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.
- ✓ Possible data types in registers are:
    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.
- ✓ Computers not only process numbers, letters and special symbols but also complex types of data such as sounds and pictures. However , these complex types of data take a lot of memory and processor time when coded in binary form
- ✓ This limitation necessitates the need to develop better ways of handling long streams of binary digits.
- ✓ Higher number systems are used in computing to reduce these streams of binary digits into manageable form to improve the processing speed and optimize memory usage.
- ✓ As far as computers are concerned, number systems can be classified into following categories
    1) Decimal number system
    2) Binary number system
    3) Octal number system
    4) Hexadecimal number system.
- ✓ 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.
- ✓ The radix for decimal number system is 10. The 10 symbols are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9.
- ✓ The string of digits 724.5 represents the quantity

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

- ✓ The radix for binary number system is 2. The two digit symbols used are 0 and 1.

- ✓ The string of digits 101101 represents the quantity

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

- ✓ For example, to show the equality between decimal and binary forty-five we will write

$$(101101)_2 = (45)_{10}.$$

- ✓ Besides the decimal and binary number systems, the octal (radix 8) and hexadecimal (radix 16) are important in digital computer work.
- ✓ The radix for octal number system is 8. The eight symbols of the octal system are 0, 1, 2, 3, 4, 5, 6, and 7.
- ✓ The radix for hexadecimal number system is 16.The 16 symbols of the hexadecimal system are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, 0, E, and F. The symbols A, B, C, D, E, and F correspond to the decimal numbers 10, 11, 12, 13, 14, 15, respectively.
- ✓ A number in radix r can be converted to the familiar decimal system by forming the sum of the weighted digits.
- ✓ For example, octal 736.4 is converted to decimal as follows:

$$(736.4)_8 = 7 \times 8^2 + 3 \times 8^1 + 6 \times 8^0 + 4 \times 8^{-1}$$

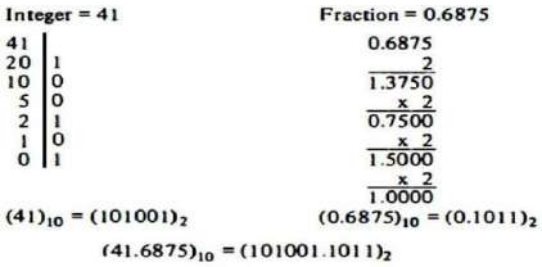
$$= 7 \times 64 + 3 \times 8 + 6 \times 1 + 4/8 = (478.5)_{10}$$

- ✓ 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 from decimal to radix r system is carried out by separating the number into its integer and fraction parts and converting each part separately.
- ✓ Divide the integer successively by r and accumulate the remainders.
- ✓ Multiply the fraction successively by r until the fraction becomes zero.

Example: Conversion of decimal 41.6875 into binary.



- ✓ The conversion from and to **binary, octal, and hexadecimal** representation plays an important part in digital computers.
- ✓ Each octal digit corresponds to three binary digits (Since $2^3 = 8$).
- ✓ Each hexadecimal digit corresponds to four binary digits (Since $2^4 = 16$).

for example, a 16-bit register.

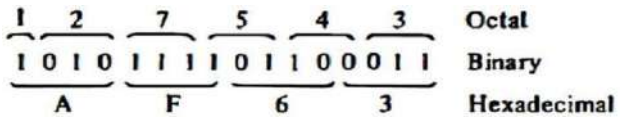


Figure: Binary, Octal and hexadecimal conversion

Table: Binary-Coded Octal and Hexadecimal Numbers

Decimal equivalent	Octal Number	Binary-Coded Octal	Hexadecimal Number	Binary-Coded Hexadecimal
0	0	000	0	0000
1	1	001	1	0001
2	2	010	2	0010
3	3	011	3	0011
4	4	100	4	0100
5	5	101	5	0101
6	6	110	6	0110
7	7	111	7	0111
8	10	001 000	8	1000
9	11	001 001	9	1001
10	12	001 010	A	1010
11	13	001 011	B	1011
12	14	001 100	C	1100
13	15	001 101	D	1101
14	16	001 110	E	1110
15	17	001 111	F	1111
20	24	010 100	14	0001 0100
50	62	110 010	32	0011 0010
99	143	001 100 011	63	0110 0011
248	370	011 111 000	F8	1111 1000

- ✓ 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.
- ✓ A binary code is a group of n bits that assume up to 2^n distinct combinations.
- ✓ 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.
- ✓ The bit assignment most commonly used for the decimal digits is the straight binary assignment listed in the first 10 entries.
- ✓ This particular code is called **binary-coded decimal** and is commonly referred as BCD.
- ✓ BCD is different from converting a decimal number to binary.
- ✓ For example, the decimal number 99 is represented by the string of bits 1100011.
- ✓ When represented in BCD, it becomes 1001 1001. below table.

Table: Binary-Coded Decimal Number

Decimal Number	Binary-Coded Decimal Number	Decimal Number	Binary-Coded Decimal Number
0	0000	10	0001 0000
1	0001	11	0001 0001
2	0010	12	0001 0010
3	0011	13	0001 0011
4	0100	14	0001 0100
5	0101	15	0001 0101
6	0110	20	0010 0000
7	0111	50	0101 0000
8	1000	99	1001 1001
9	1001	248	0010 0100 1000

- ✓ The standard alphanumeric code is the ASCII (American Standard Code for Information Interchange), which uses seven bits to code 128 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.

Table: 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 0100
O	100 1111	(	010 1000
P	101 0000	)	010 1001
Q	101 0001	*	010 1010
R	101 0010	+	010 1011
S	101 0011	,	010 1100
T	101 0100	-	010 1101
U	101 0101	.	010 1110
V	101 0110	/	010 1111
W	101 0111	=	011 1101
X	101 1000		
Y	101 1001		
Z	101 1010		

Complements:

- ✓ Complements are used in digital computers for simplifying subtraction and logical manipulation.
- ✓ Two types of complements for each base r system: r's complement and (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, the 9's complement of N is $(10^n - 1) - N$.
- ✓ The 9's complement of 546700 is 999999 – 546700 = 453299.
- ✓ The 9's complement of 453299 is 999999 – 453299 = 546700.
- ✓ For binary, the 1's complement of N is $(2^n - 1) - N$.
- ✓ The 1's complement of 1011001 is 1111111 – 1011001 = 0100110.
- ✓ The 1's complement is the true complement of the number - just toggle all bits.
- ✓ The r's complement of an n-digit number N in base r is defined as $r^n - N$.
- ✓ This is the same as adding 1 to the (r – 1)'s complement.
- ✓ The 10's complement of 546700 is 546700 + 1 = 546701.
- ✓ The 10's complement of 453299 is 453299 + 1 =453300.
- ✓ The 2's complement of 1011001 is 0100110 + 1 =0100111.
- ✓ Subtraction of unsigned two n-digit numbers: M - N

1) Add the minuend M to the r's complement of the subtrahend N. This result is

$$M + (r^n - N) = M - N + r^n.$$

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.

Example: The subtraction 72532 - 13250 = 59282. The l0's complement of 13250 is 86750.

M	= 72532
l0's complement of N	= +86750
Sum	= 159282
Discard ends carry 10^5	= -100000
Answer	= 59282

Example: With M < N. The subtraction 13250 – 72532 produces negative 59282. Using the procedure with complements, we have

M	= 13250

10's complement of N = +27468

Sum = 40718

There is no end carry.

Answer = -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 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.

Example: X = 1010100 and Y = 1000011, we perform the subtraction X – Y and Y - X using 2's complements:

X = 1010100

2' s complement of Y = +0111101

Sum = 10010001

Discard ends carry 2^7 = - 10000000

Answer: X - Y = 0010001

Y = 1000011

2's complement of X = +0101 100

Sum = 1101111

No end carry.

Answer = - 0010001 (2's complement of 1101111).

Fixed-Point Representation:

- ✓ Positive integers, including zero, can be represented as unsigned numbers.
- ✓ To represent negative integers, we need a notation for negative values. In ordinary arithmetic, a negative number is indicated by a - sign and a positive number by a + sign.
- ✓ Because of hardware limitations, computers must represent everything with 1's and 0's, including the sign of a number.
- ✓ To represent the sign MSB position of the number. The convention is to make the sign bit equal to 0 for positive and to 1 for negative.
- ✓ Two ways to designate binary point position in a register.
  1) Fixed point position
  2) Floating-point position

- ✓ The fixed-point position usually uses one of the following positions.

  (1) A binary point in the extreme left of the register to make it a fraction.

  (2) A binary point in the extreme right of the register to make it an integer.

  (3) In both cases, a binary point is not actually present.

- ✓ The floating-point representations use a second register to designate the position of the binary point in the first register.
- ✓ When an integer binary is positive, the MSB, or the sign is represented by 0 and the remaining bits represent the magnitude.
- ✓ When the number is negative, the MSB, 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

Example: consider the signed number 14 stored in an 8-bit register. + 14

- The only way to represent it is 00001110.

Example: consider the signed number 14 stored in an 8-bit register. - 14

There are three different ways to represent - 14 with eight bits.

1) In signed-magnitude representation: 1 0001110
2) In signed-1's complement representation: 1 1110001
3) In signed-2's complement representation: 1 1110010

- ✓ Typically use signed 2's complement representation for negative number.
- ✓ Addition of two signed-magnitude numbers follow the normal rules.
  - If same signs, add the two magnitudes and use the common sign.
  - Differing signs, subtract the smaller from the larger and use the sign of the larger magnitude.
  - Must compare the signs and magnitudes and then either add or subtract.
- ✓ Addition of two signed 2's complement numbers does not require a comparison or subtraction – only addition and complementation.
  - Add the two numbers, including their sign bits.
  - Discard any carry out of the sign bit position.
  - All negative numbers must be in the 2's complement form.
  - If the sum obtained is negative, then it is in 2's complement form.

+6	00000110	-6	11111010
+13	00001101	+13	00001101
+19	00010011	+7	00000111

+6	00000110	-6	11111010
-13	11110011	-13	11110011
-7	11111001	-19	11101101

- ✓ Subtraction of two signed 2's complement numbers is as follows.
  - Take the 2's complement form of the subtrahend (including sign bit).
  - Add it to the minuend (including the sign bit).
  - A carry out of the sign bit position is discarded.
- ✓ An overflow occurs when two numbers of n digits each are added, and the sum occupies n + 1 digits.
- ✓ Overflows are problems since the width of a register is finite.
- ✓ Therefore, a flag is set if this occurs and can be checked by the user.
- ✓ Detection of an overflow depends on if the numbers are signed or unsigned.
- ✓ For unsigned numbers, an overflow is detected from the end carry out of the MSB.
- ✓ For addition of signed numbers, an overflow cannot occur if one is positive, and one is negative – both must have the same sign.
- ✓ An overflow can be detected if the carry into the sign bit position and the carry out of the sign bit position are not equal.

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

- ✓ 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.
- ✓ This takes much more space than the equivalent binary representation and the circuits required to perform decimal arithmetic are more complex.
- ✓ Representation of signed decimal numbers in BCD is like the representation of signed numbers in binary.
- ✓ Either signed magnitude or signed complement systems.
- ✓ The sign of a number is represented with four bits.
  - 0000 for +
  - 1001 for –
- ✓ To obtain the 10's complement of a BCD number, first take the 9's complement and then add one to the least significant digit.

Example: (+375) + (-240) = +135

0 375	$(0000\ 0011\ 0111\ 1010)_{BCD}$
+9 760	$(1001\ 0111\ 0110\ 0000)_{BCD}$
0 135	$(0000\ 0001\ 0011\ 0101)_{BCD}$

Floating-Point Representation

✓ The floating-point representation of a number has two parts.

✓ The first part represents a signed, fixed-point number – the mantissa.

✓ The second part designates the position of the binary point – the exponent.

✓ The mantissa may be a fraction or an integer.

Example: the decimal number +6132.789 is

- Fraction: +0.6123789
- Exponent: +04
- Equivalent to $+0.6132789 \times 10^{+4}$

✓ A floating-point number is always interpreted to represent $m \times r^e$.

Example: the binary number +1001.11 (with 8-bit fraction and 6-bit exponent)

- Fraction: 01001110
- Exponent: 000100
- Equivalent to $+(.1001110)2 \times 2^{+4}$

✓ A floating-point number is said to be normalized if the most significant digit of the mantissa is nonzero.

✓ The decimal number 350 is normalized, 00350 is not.

✓ The 8-bit number 00011010 is not normalized.

✓ Normalize it by fraction = 11010000 and exponent = -3.

✓ Normalized numbers provide the maximum possible precision for the floating-point number.

Other Binary Codes

✓ Digital systems can process data in discrete form only.

✓ Continuous, or analog, information is converted into digital form by means of an analog-to-digital converter.

✓ The reflected binary or Gray code is sometimes used for the converted digital data.

✓ The Gray code changes by only one bit as it sequences from one number to the next

✓ Gray code counters are sometimes used to provide the timing sequences that control the operations in a digital system.

Table for 4-Bit Gray Code

Binary code	Decimal Equivalent	Binary code	Decimal Equivalent
0000	0	1100	8
0001	1	1101	9
0011	2	1111	10
0010	3	1110	11
0110	4	1010	12
0111	5	1011	13
0101	6	1001	14
0100	7	1000	15

- ✓ Binary codes for decimal digits require a minimum of four bits.
- ✓ Other codes besides BCD exist to represent decimal digits.

Table for Four Different Binary Codes for Decimal Digits

Decimal Digit	BCDigit 8421	2421	Excess – 3	Excess – 3 gray
0	0000	0000	0011	0010
1	0001	0001	0100	0110
2	0010	0010	0101	0111
3	0011	0011	0110	0101
4	0100	0100	0111	0100
5	0101	1011	1000	1100
6	0110	1100	1001	1101
7	0111	1101	1010	1111
8	1000	1110	1011	1110
9	1001	1111	1100	1010
Unused Bit combination	1010	0101	0000	0000
	1011	0110	0001	0001
	1100	0111	0010	0011
	1101	1000	1101	1000
	1110	1001	1110	1001
	1111	1010	1111	1011

- ✓ The 2421 code and the excess-3 code are both self-complementing.
- ✓ The 9's complement of each digit is obtained by complementing each bit in the code.
- ✓ The 2421 code is a weighted code.
- ✓ The bits are multiplied by indicated weights and the sum gives the decimal digit.
- ✓ The excess-3 code is obtained from the corresponding BCD code added to 3.

COMPUTER ARITHMETIC:

INTRODUCTION

- ✓ The four basic arithmetic operations in computers are: Addition, Subtraction, Multiplications, and Division.
- ✓ Arithmetic processor is part of processor unit and it executes arithmetic operations.
- ✓ Arithmetic instructions specify data type binary or decimal, fixed point or floating point
- ✓ Fixed point numbers represents integers while floating point numbers represent fractional numbers.
- ✓ Negative numbers may be in signed magnitude or signed complement representation.
  - ➢ Fixed point binary signed magnitude
  - ➢ Fixed point binary 2's complement
  - ➢ Floating point binary
  - ➢ Floating point BCD

ADDITION AND SUBTRACTION:

- **ADDITION AND SUBTRACTION WITH SIGNED MAGNITUDE DATA**

- ✓ Is familiar as used in every day arithmetic calculations
- ✓ We designate magnitudes of two numbers as A and B. when those sign numbers are added we have eight different conditions depending on the sign bits and operations performed. The following table shows those conditions and other columns shows operations performed.

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)

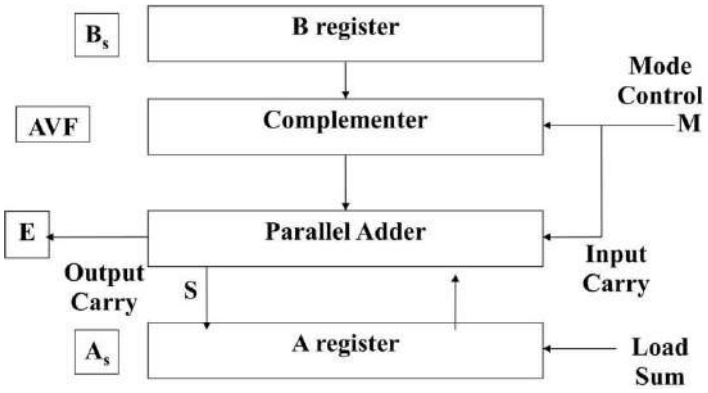
Addition (Subtraction) Algorithm will be as

- ✓ When A and B have identical (different) signs, add the two magnitudes and attach the sign of A to result.

- ✓ When signs of A and B are different (same), subtract smaller number from larger number. Choose sign of result as sign of A if A>B, or complement of sign of A if A<B.
- ✓ If two magnitudes are equal, subtract B from A and make sign of result positive.

HARDWARE IMPLEMENTATION

- ✓ The two numbers are stored in registers A and B and their signs in flip-flops A_s and B_s. Result is transferred to A register with its sign
- ✓ First, a parallel – adder is needed to perform the mirooperation A + B. Second, comparator circuit is needed to establish if A>B, A=B, A<B. Third, two parallel-subtractor needed to perform the microoperations A – B and B – A. The sign relationship can be determined from an exclusive OR gate with A_s and B_s as inputs.
- ✓ The figure shows the hardware for signed magnitude addition-subtraction. Consists of registers A and B, sign bits A_s and B_s



- ✓ 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 control M.
- ✓ The complementer consists of exclusive-OR gates and the parallel adder consists of full-adder circuits
- ✓ The M signal is also applied to the input carry of the adder. 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 l's complement of B is applied to the adder, the input carry is 1, and output

S = A+B + 1. This is equal to A plus the 2's complement of B, which is equivalent to the subtraction A - B.

HARDWARE ALGORITHM:

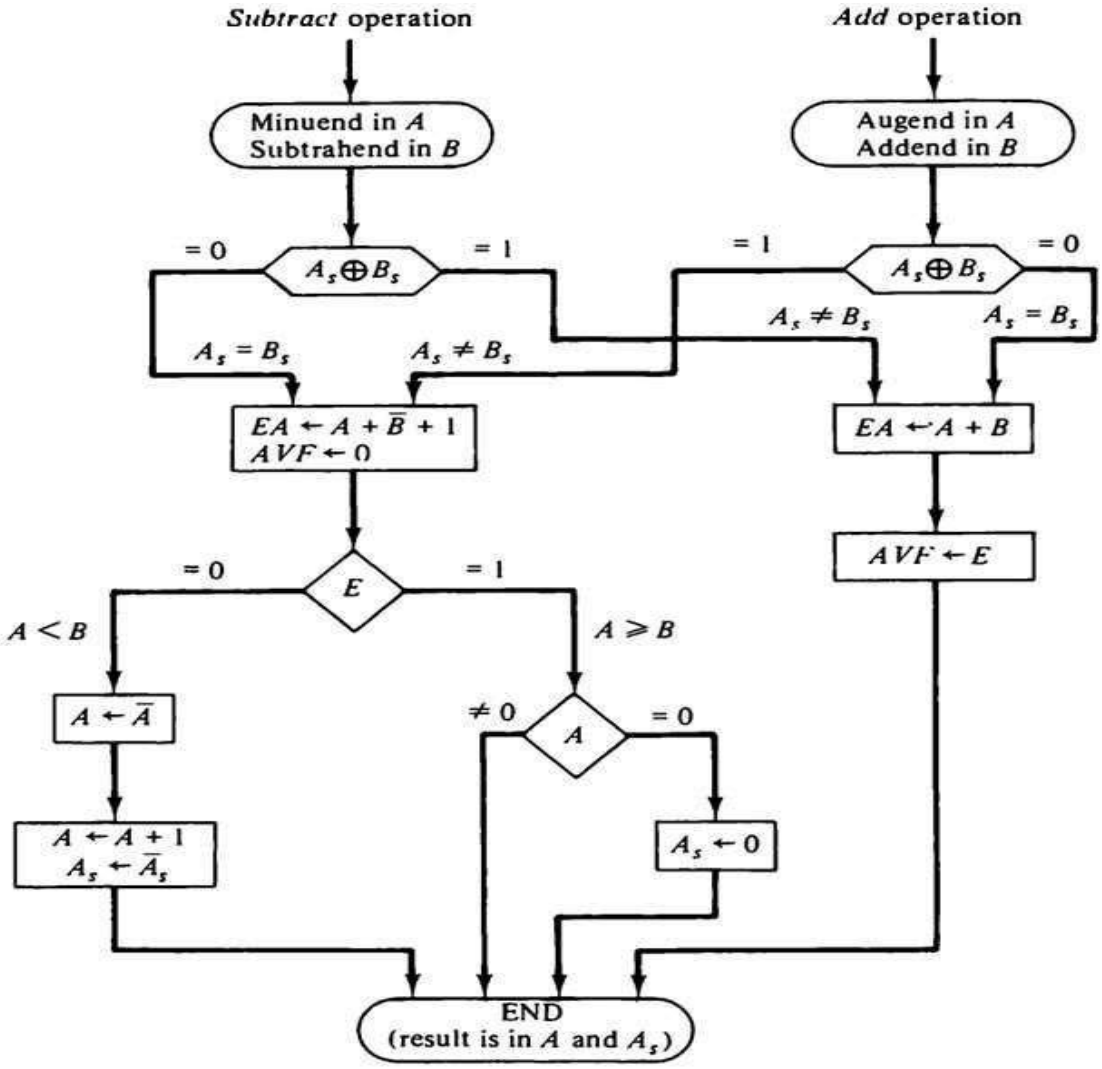


Figure 10-2 Flowchart for add and subtract operations.

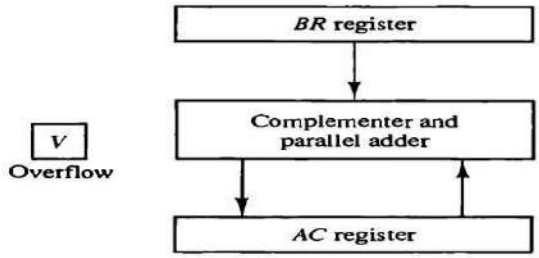
- ✓ The two signs A_s and B_s are compared by XOR gate. If result is 0 then As = Bs and if result is 1 the As ≠ Bs.
- ✓ For add operations if have same sign bits the magnitude must be added. For subtract operations different sign bits means magnitudes be added as well.
- ✓ E bit is carry bit after addition and moves to AVE overflow bit only at this state.
- ✓ If sign bits are different in add operations or the same in subtract operations the two magnitudes will be subtracted A – B. No overflow can occur here.

- ✓ After subtract if E=1 this means A>=B and if E=0 then A<B. Then here it is necessary to get 2's complement of A (by invert A then add 1) and sign of A is inverted only in this case.
- ✓ Final result will be in A and its sign in A_s.

ADDITION AND SUBTRACTION WITH SIGNED 2'S COMPLEMENT DATA

- ✓ The left most bit in binary number is the sign bit. If 0 the number is positive and if 1 then number is negative. If sign bit is 1 the entire number is represented in 2's complement.
- ✓ The addition of two numbers represented in 2's complement is carried out by normal binary addition with carry discarded.
- ✓ The subtraction is carried out by taking 2's complement (B) of subtrahend and adding it to minuend (A).
- ✓ When two numbers of n digits each are added and the sum occupies n + 1 digits, we say that an overflow occurred Overflow can be detected by inspecting last 2 carries out of addition by XOR them. If different then overflow is detected.

Hardware for signed-2's complement addition and subtraction.



- ✓ The algorithm for adding and subtracting two binary numbers in signed-2' s complement representation is shown in the flowchart
- ✓ For addition simply implement add then see overflow. For subtract add 2's complement of B to A and watch overflow since the A and –B could be of same sign.

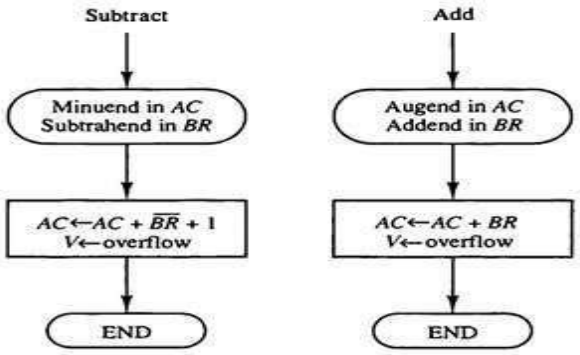


Figure 10-4 Algorithm for adding and subtracting numbers in signed-2's complement representation.

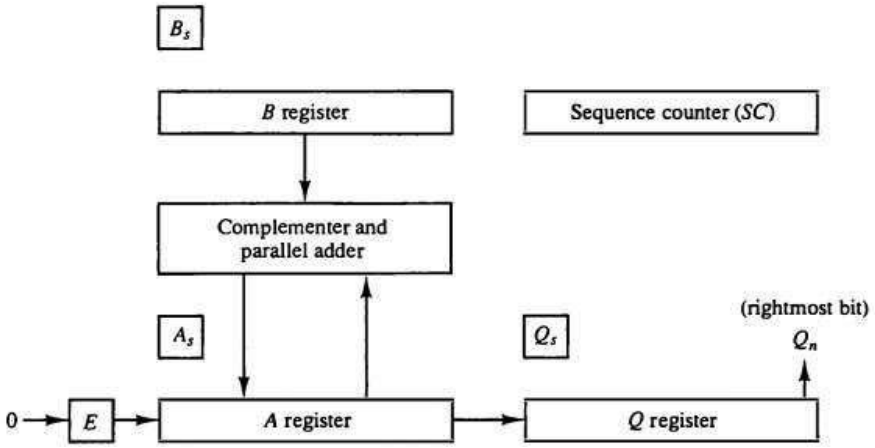
- ✓ Comparing this algorithm with its signed-magnitude counterpart, we note that it is much simpler to add and subtract numbers if negative numbers are maintained in signed-2' s complement representation. For this reason most computers adopt this representation over the more familiar signed-magnitude.

MULTIPLICATION ALGORITHMS

- ✓ The multiplication of two numbers in signed magnitude representation is carried out by successive shift and adds.
- ✓ Look at successive bits of multiplier (least significant bit first), if bit=1 multiplicand is copied else if bit=0, zero is copied down shifted one bit to left from previous copies. Finally all numbers are added and their sum forms the product.

HARDWARE IMPLEMENTATION FOR SIGNED MAGNITUDE DATA

Figure 10-5 Hardware for multiply operation.



- ✓ It is necessary to supply adder for summation two binary numbers and successively accumulate the partial products in a register
- ✓ Instead of shifting multiplicand to the left the partial product is shifted to the right.
- ✓ When corresponding bit of multiplier is 0 no need to add zeros to partial product, just do nothing.

- ✓ Hardware for multiplication consist of complementer and parallel adder, Registers A and B with their sign bits A_s and B_s, multiplier Q register, and its sign in Q_s. SC is set to number of bits in multiplier and when reaches zero it means partial addition has finished and product is ready in A.
- ✓ Multiplicands in B register while multipliers in Q register initially. The sum of A and B is the partial product in EA. A and Q registers are shifted together giving rise to new bit from Q in Q_n bit which tells us if that bit is 0 or 1.

ALGORITHM

- ✓ Signs of Q_s and B_s are compared then A_s and Q_s are set as sign of the product. A and E bit are cleared with SC set to number of bits in Q
- ✓ Low order bit in Q is tested. If Qn=1 then B is added to A. If Qn=0 then nothing is done.
- ✓ Registers EAQ are shifted to the right.
- ✓ SC is decremented. Process stops when SC=0.

✓ Note that product is occupying A and Q registers and after each shift it replaces A and Q registers.

Figure 10-6 Flowchart for multiply operation.

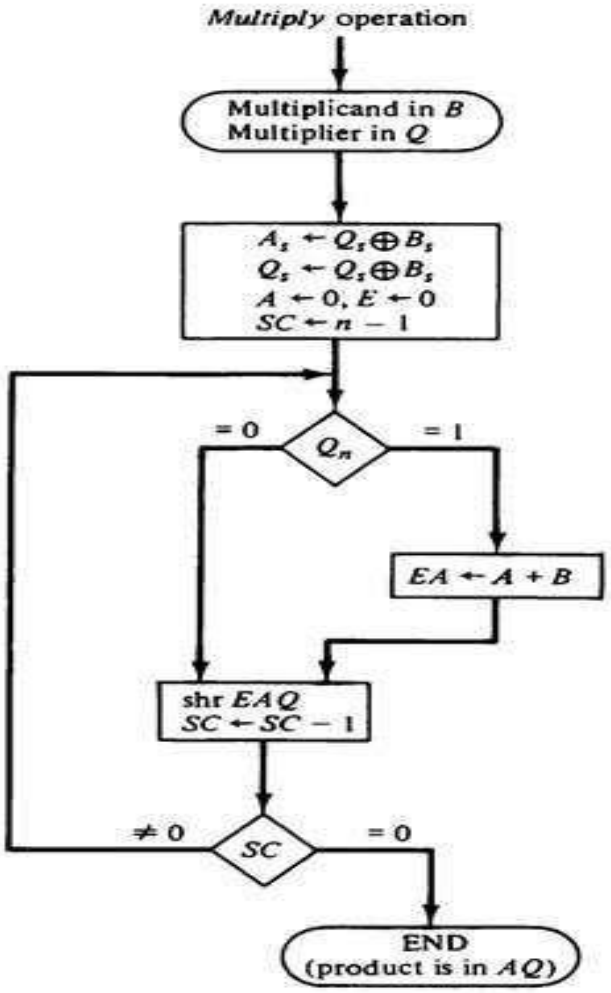


TABLE 10-2 Numerical Example for Binary Multiplier

Multiplicand B = 10111	E	A	Q	SC
Multiplier in Q	0	00000	10011	101
Q_n = 1; add B		10111		
First partial product	0	10111		
Shift right EAQ	0	01011	11001	100
Q_n = 1; add B		10111		
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		10111		
Fifth partial product	0	11011		
Shift right EAQ	0	01101	10101	000
Final product in AQ = 0110110101				

BOOTH MULTIPLICATION ALGORITHM

✓ Booth algorithm requires examination of the multiplier bits and shifting of the partial product. Prior to the shifting, the multiplicand may be added to the partial product, subtracted from the partial product, or left unchanged according to the following rules:

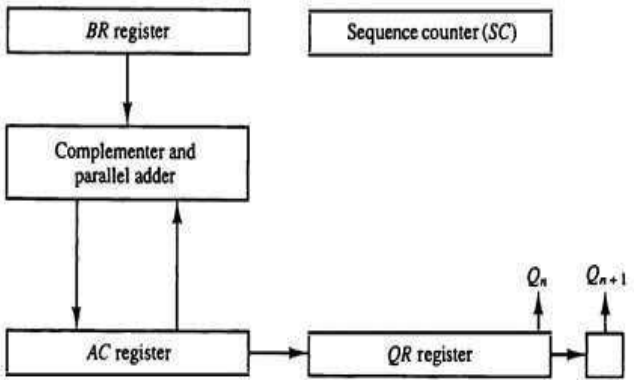
1. The multiplicand is subtracted from the partial product upon encountering the first least significant 1 in a string of 1's in the multiplier.
2. The multiplicand is added to the partial product upon encountering the first 0 (provided that there was a previous 1) in a string of 0's in the multiplier.
3. The partial product does not change when the multiplier bit is identical to the previous multiplier bit.

✓ The algorithm works for positive or negative multipliers in 2's complement representation.

✓ The hardware implementation of Booth algorithm requires the register configuration shown in Fig.

✓ Q_n designates the least significant bit of the multiplier in register QR. An extra flip-flop Q_{n+1} is appended to QR to facilitate a double bit inspection of the multiplier.

Figure 10-7 Hardware for Booth algorithm.



✓ The flowchart for Booth algorithm is shown in Fig.

✓ 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 inspected. If the two bits are equal to 10, it means that the first 1 in a string of 1's has been encountered. This requires a subtraction of the multiplicand from the partial product in AC. If the two bits are equal to 01, it means that the first 0 in a string of 0's has been encountered. This requires the addition of the multiplicand to 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.

✓ 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.

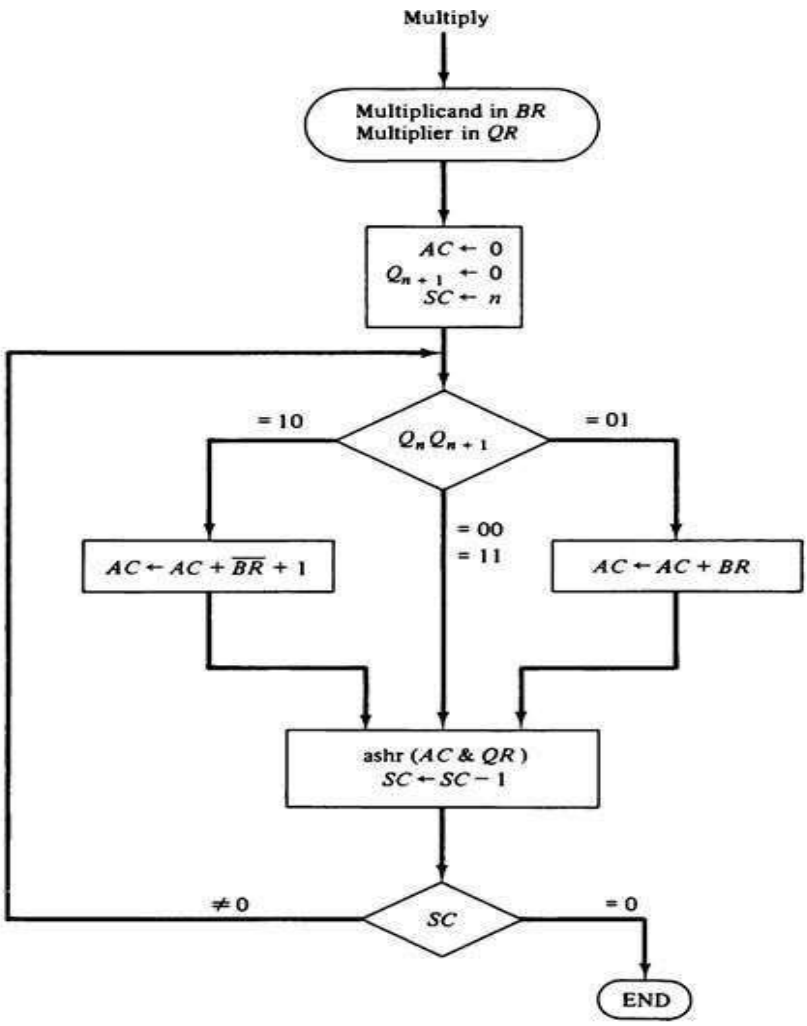


Figure 10-8 Booth algorithm for multiplication of signed-2's complement numbers.

A numerical example of Booth algorithm is shown in Table for n = 5.

It shows the step-by-step multiplication of

(- 9) x (- 13) = + 117.

Note that the multiplier in QR is negative and that the multiplicand in BR is also negative. The 10-bit product appears in AC and QR and is positive.

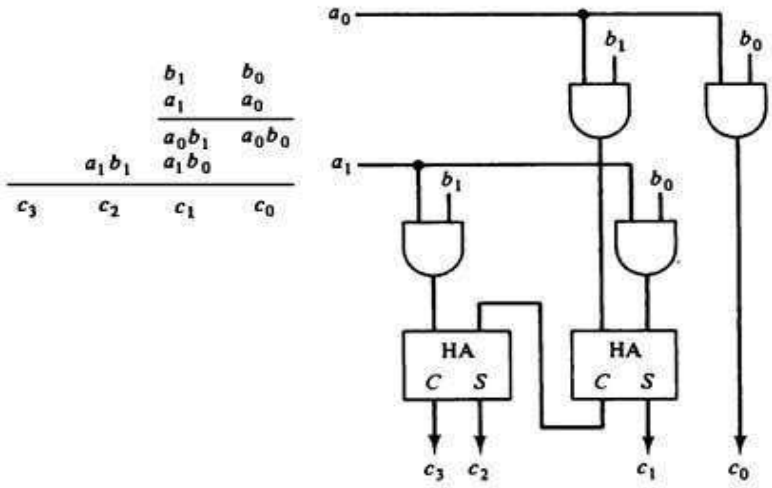
TABLE 10-3 Example of Multiplication with Booth Algorithm

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

ARRAY MULTIPLIER

- ✓ Checking the bits of the multiplier one at a time and forming partial products is a sequential operation that requires a sequence of add and shift microoperations.
- ✓ The multiplication of two binary numbers can be done with one microoperation by means of a combinational circuit that forms the product bits all at once. 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.
- ✓ However, an array multiplier requires a large number of gates, and for this reason it was not economical until the development of integrated circuits.
- ✓ Implementation of an array multiplier with a combinational circuit, for the multiplication of two 2-bit numbers as shown in Fig.

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



- ✓ The multiplicand bits are b1 and b0, the multiplier bits are a1 and a0, and the product is c3 c2 c1 c0
- ✓ The first partial product is formed by multiplying a0 by b1 b0•
- ✓ The multiplication of two bits such as a0 and b0 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 a1 by b1 b0 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.

- ✓ A combinational circuit binary multiplier with more bits can be constructed in a similar fashion. A bit of the multiplier is ANDed with each bit of the multiplicand in as many levels as there are bits in the multiplier. The binary output in each level of AND gates is added in parallel with the partial product of the previous level to form a new partial product. The last level produces the product. For j multiplier bits and k multiplicand bits we need j x k AND gates and (j - 1) k-bit adders to produce a product of j + k bits.
- ✓ For example, 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 is shown in Fig.

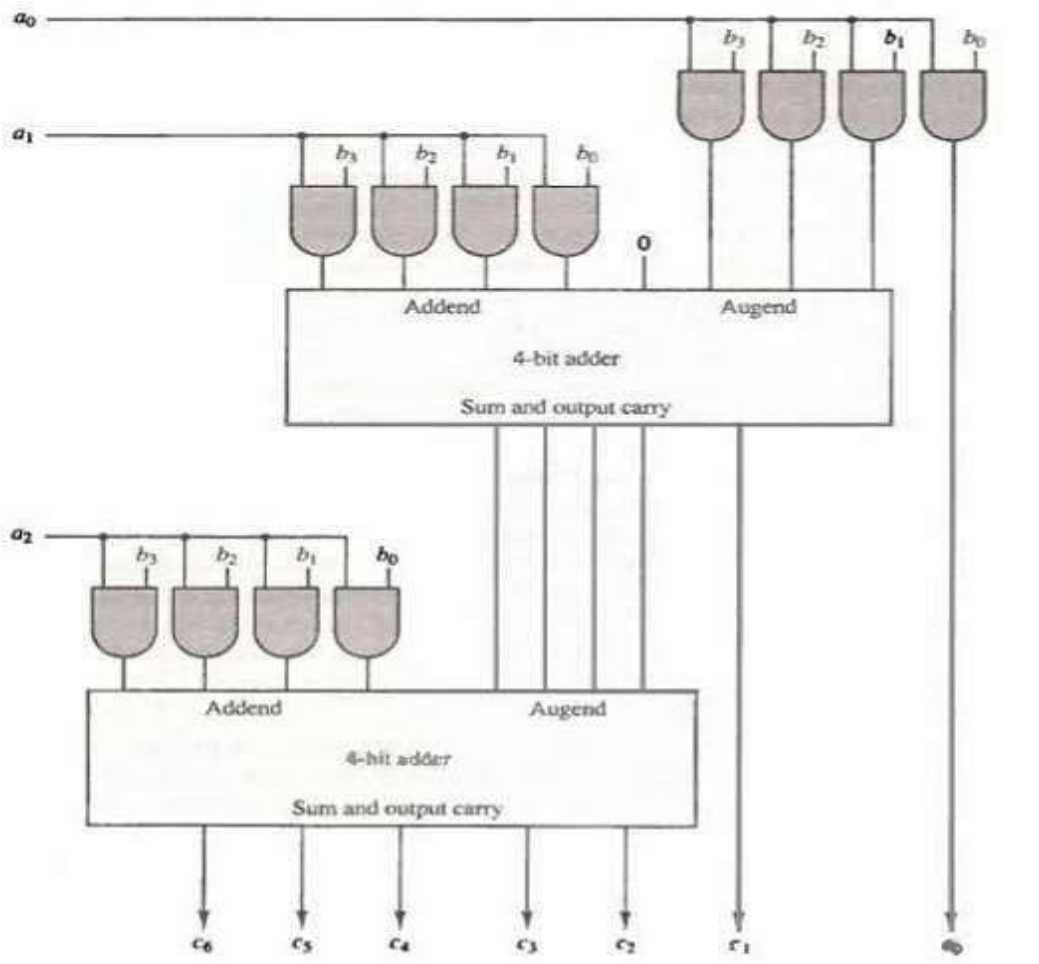


Figure 10.10 4-bit by 3-bit array multiplier.

DIVISION ALGORITHMS

Division of two fixed-point binary numbers in signed-magnitude representation is done with paper and pencil by a process of successive compare, shift, and subtract operations.

Figure 10-11 Example of binary division.

Divisor:	11010	Quotient = Q
B = 10001	)0111000000	Dividend = A
	01110	5 bits of $A < B$, quotient has 5 bits
	011100	6 bits of $A \geq B$
	−10001	Shift right B and subtract; enter 1 in Q
	−010110	7 bits of remainder $\geq B$
	−−10001	Shift right B and subtract; enter 1 in Q
	−−001010	Remainder $< B$; enter 0 in Q; shift right B
	−−−010100	Remainder $\geq B$
	−−−−10001	Shift right B and subtract; enter 1 in Q
	−−−−000110	Remainder $< B$; enter 0 in Q
	−−−−−00110	Final remainder

HARDWARE IMPLEMENTATION FOR SIGNED-MAGNITUDE DATA

- ✓ When the division is implemented in a digital computer, it is convenient to change the process slightly. Instead of shifting the divisor to the right, the dividend, or partial remainder, is shifted to the left, thus leaving the two numbers in the required relative position.
- ✓ Subtraction may be achieved by adding A to the 2's complement of B. The information about the relative magnitudes is then available from the end-carry.
- ✓ The hardware for implementing the division operation is identical to that required for multiplication. Register EAQ is now shifted to the left with 0 inserted into Q, and the previous value of E lost.
- ✓ The numerical example: 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 >= B.
- ✓ A quotient bit 1 is inserted into Q, 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, 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 bits are also shifted and after five shifts, the quotient is in Q and the final remainder is in A.
- ✓ The sign of the quotient is determined from the signs of the dividend and the divisor. If the two signs are alike, the sign o f the quotient is plus. If they are unlike, the sign is minus.
- ✓ The sign of the remainder is the same as the sign of the dividend.

Divisor $B = 10001$, $\bar{B} + 1 = 01111$

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

DIVIDE OVERFLOW

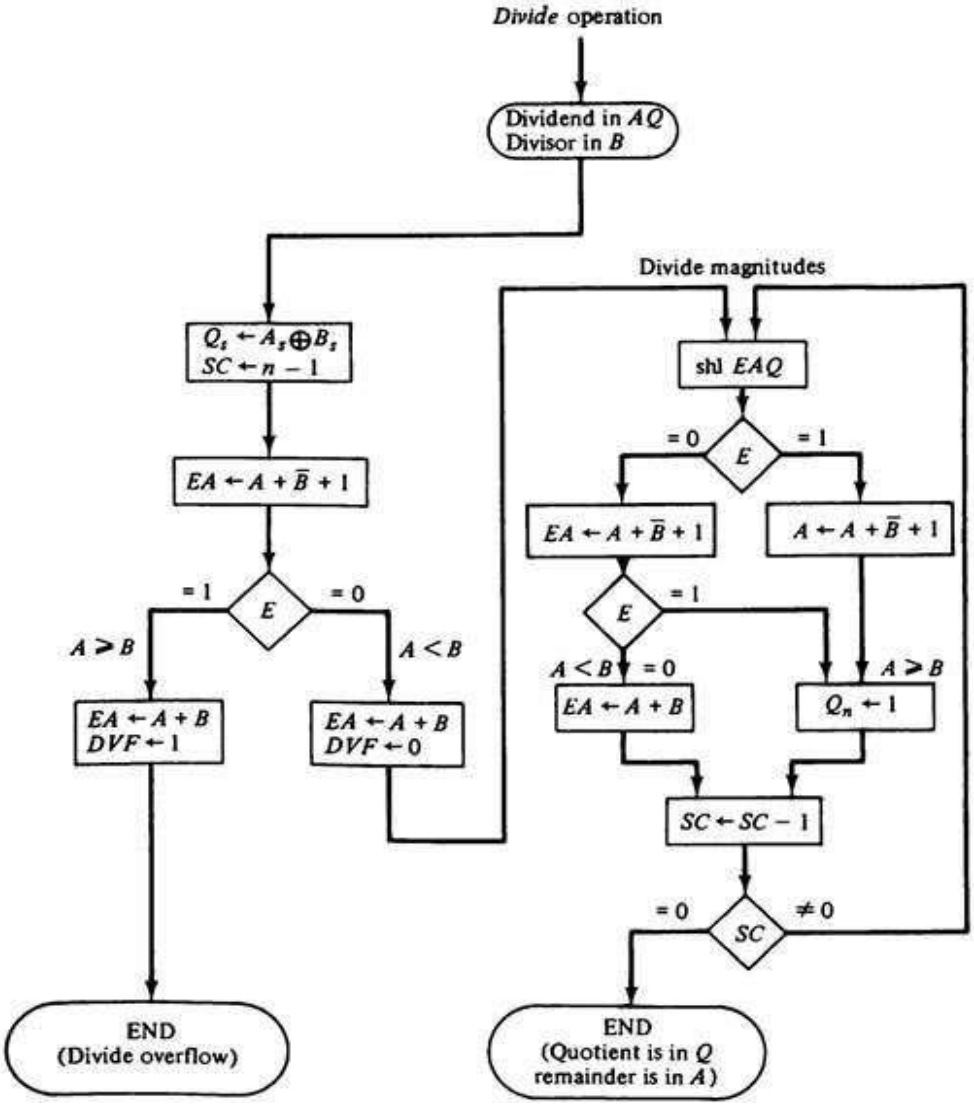
- ✓ The division operation may result in a quotient with an overflow. This is critical when the operation is implemented with hardware. This is because the length of registers is finite and will not hold a number that exceeds the standard length.
- ✓ When the dividend is twice as long as the divisor, the condition for overflow can be stated as follows: *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. The divide-overflow condition takes care of this condition as well. This occurs because any dividend will be greater than or equal to a divisor which is equal to zero. Overflow condition is usually detected when a special flip-flop (divide-overflow flip-flop and label it DVF) is set.
- ✓ The occurrence of a divide overflow can be handled in a variety of ways. In some computers it is the responsibility of the programmers to check if DVF is set after each divide instruction. They then can branch to a subroutine that takes a corrective measure such as rescaling the data to avoid overflow.

- ✓ In some older computers, the occurrence of a divide overflow stopped the computer and this condition was referred to as a divide stop. Stopping the operation of the computer is not recommended because it is time consuming.
- ✓ The procedure in most computers is to provide an interrupt request when DVF is set. The interrupt causes the computer to suspend the current program and branch to a service routine to take a corrective measure. The most common corrective measure is to remove the program and type an error message explaining the reason why the program could not be completed. It is then the responsibility of the user who wrote the program to rescale the data or take any other corrective measure. The best way to avoid a divide overflow is to use floating point data.

HARDWARE ALGORITHM

The hardware divide algorithm is shown in the following flowchart.

Figure 10-13 Flowchart for divide operation.



- ✓ 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 >=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.
- ✓ The division o f the magnitudes starts by shifting the dividend in A Q to the left with the high-order bit shifted into E. If the bit shifted into E is 1, we know that EA > B because EA consists of a 1 followed by n - 1 bits while B consists of only n -1 bits. In this case, B must be subtracted from EA and 1 inserted into Q, for the quotient bit.
- ✓ Since register A is missing the high-order bit of the dividend (which is in E), its value is EA - 2^{n-1} • Adding to this value the 2' s complement of B results in

$$(EA - 2^{n-1}) + (2^{n-1} - B) = EA - B$$

The carry from this addition is not transferred to E if we want E to remain a l.

- ✓ If the shift-left operation inserts a 0 into E, the divisor is subtracted by adding its 2's complement value and the carry is transferred into E. If E = 1, it signifies that A >=B; therefore, Q_n is set to 1. If E = 0, it signifies that A < B and the original number is restored by adding B to A.
- ✓ In the latter case we leave a 0 in Q, (0 was inserted during the shift).This process is repeated again with register A holding the partial remainder. After n - 1 times, the quotient magnitude is formed in register Q and the remainder is found in register A. The quotient sign is in Q_s and the sign of the remainder in A_s is the same as the original sign of the dividend.

OTHER ALGORITHMS

Two other methods are available for dividing numbers, the comparison method and the non restoring method.

- ✓ In the *comparison method* A and B are compared prior to the subtraction operation. Then if A>=B, B is subtracted from A. If A < B nothing is done. The partial remainder is shifted left and the numbers are compared again. The comparison can be determined prior to the subtraction by inspecting the end-carry out of the parallel-adder prior to its transfer to register E .
- ✓ In the *nonrestoring method*, B is not added if the difference is negative but instead, the negative difference is shifted left and then B is added. To see why this is possible consider the case when A < B. We note that the operations performed are A - B + B; that is, B is subtracted and then

added to restore A. The next time around the loop, this number is shifted left (or multiplied by 2) and B subtracted again. This gives 2(A - B + B) - B = 2A - B. This result is obtained in the nonrestoring method by leaving A - B as is.

- ✓ The next time around the loop, the number is shifted left and B added to give 2(A - B) + B = 2A - B, which is the same as before.
- ✓ Thus, in the nonrestoring method, B is subtracted if the previous value of Q_n was a 1, but B is added if the previous value of Q_n was a 0 and no restoring of the partial remainder is required. This process saves the step of adding the divisor if A is less than B, but it requires special control logic to remember the previous result. The first time the dividend is shifted, B must be subtracted. Also, if the last bit of the quotient is 0, the partial remainder must be restored to obtain the correct final remainder.

FLOATING-POINT ARITHMETIC OPERATIONS

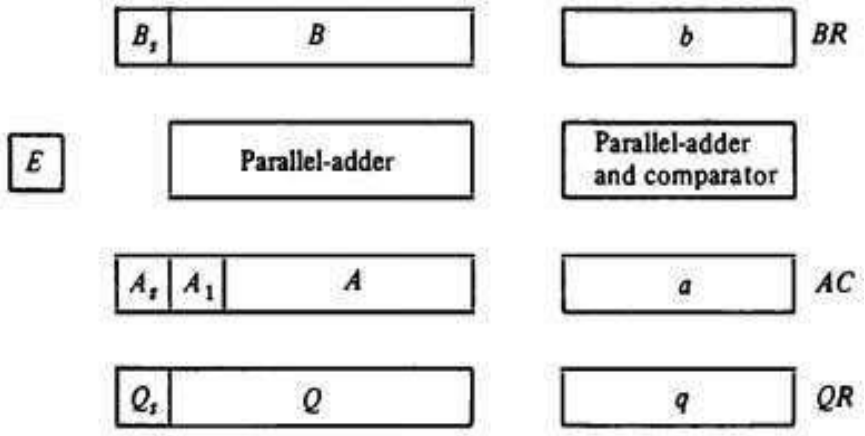
- ✓ A floating point number in computer registers consists of two parts: a mantissa m and an exponent e. The two parts represent a number obtained from multiplying m times a radix r raised to the value of e; thus $\mathbf{m * r^e}$
- ✓ The mantissa may be a fraction or an integer. The location of the radix point and the value of the radix r are assumed and are not included in the registers.
- ✓ A floating-point number is normalized if the most significant digit of the mantissa is nonzero.
- ✓ Arithmetic operations with floating-point numbers are more complicated than with fixed-point numbers and their execution takes longer and requires more complex hardware.
- ✓ 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 its exponent is adjusted until it is equal to the other exponent.
- ✓ When the mantissas are stored in registers, shifting to the left causes a loss of most significant digits. Shifting to the right 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.
- ✓ 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.
- ✓ 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.
- ✓ 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
- ✓ The operations performed with the mantissas are the same as in fixed point numbers, so the two can share the same registers and circuits.
- ✓ The operations performed with the exponents are compare and increment (for aligning the mantissas), add and subtract (for multiplication and division), and decrement (to normalize the result). The exponent may be represented in any one of the three representations: signed-magnitude, signed-2's complement, or signed-1's complement.
- ✓ A fourth representation employed in many computers is known as a biased exponent. In this representation, the sign bit is removed from being a separate entity. The bias is a positive number

that is added to each exponent as the floating-point number is formed, so that internally all exponents are positive.

REGISTER CONFIGURATION

✓ The register organization for floating-point operations is shown in Fig.

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



✓ There are three registers, BR, AC, and QR. Each register is subdivided into two parts. The mantissa part has the same uppercase letter symbols as in fixed-point representation. The exponent part uses the corresponding lowercase letter symbol

✓ 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 A1.

✓ 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. A separate parallel-adder is used for the exponents. Since the exponents are biased, they do not have a distinct sign bit but are represented as a biased positive quantity.

✓ The number in the mantissa will be taken as a fraction, so the binary point is assumed to reside to the left of the magnitude part.

✓ The numbers in the registers are assumed to be initially normalized. After each arithmetic operation, the result will be normalized. Thus all floating-point operands coming from and going to the memory unit are always normalized

ADDITION AND SUBTRACTION

- ✓ During addition or 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.
- ✓ A floating-point number that is zero cannot be normalized. If this number is used during the computation, the result may also be zero. Instead of checking for zeros during the normalization process we check for zeros at the beginning and terminate the process if necessary.
- ✓ The alignment of the mantissas must be carried out prior to their operation. After the mantissas are added or subtracted, the result may be un-normalized. The normalization procedure ensures that the result is normalized prior to its transfer to memory.
- ✓ The flowchart for adding or subtracting two floating-point binary numbers is shown in Fig.
- ✓ 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 also complement its sign if the numbers are to be subtracted.
- ✓ If neither number is equal to zero, we proceed to align the mantissas. The magnitude comparator attached to exponents a and b provides three outputs that indicate their relative magnitude. 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. The magnitude part is added or subtracted depending on the operation and the 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. No underflow may occur in this case because the original mantissa that was not shifted during the alignment was already in a normalized position.
- ✓ 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 in 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.

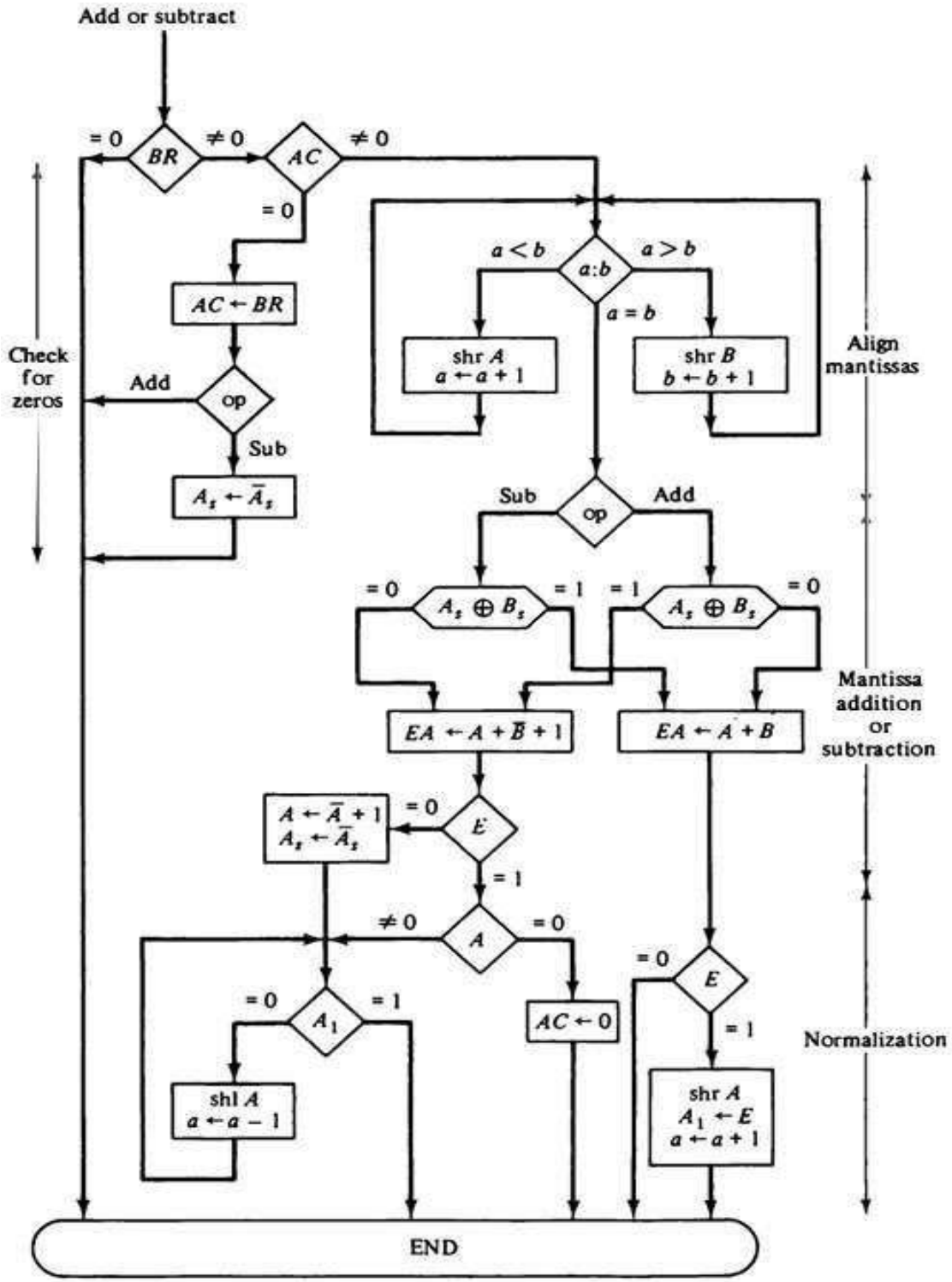
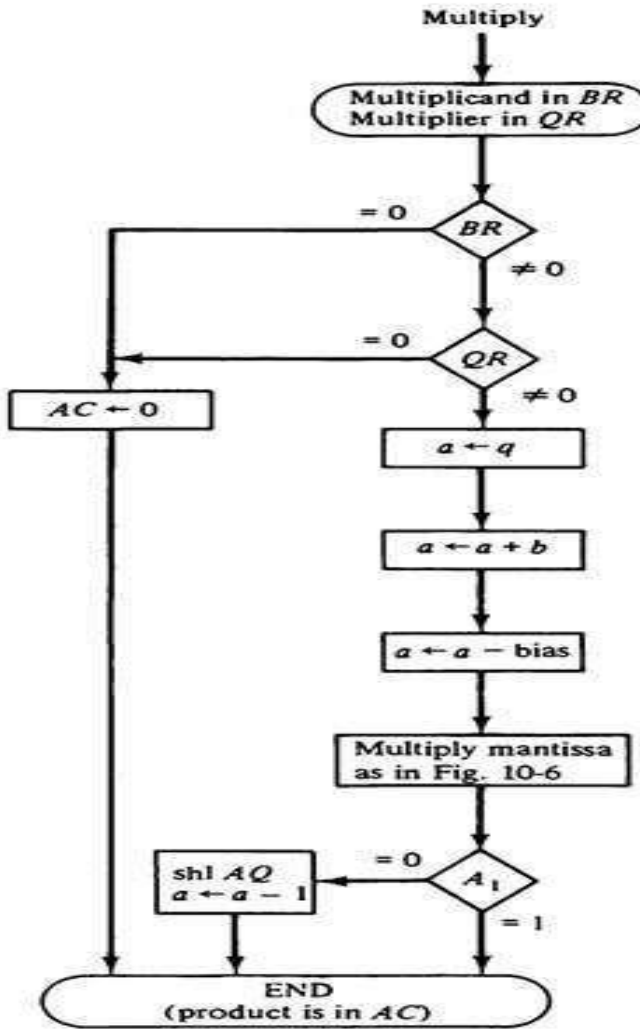


Figure 10-15 Addition and subtraction of floating-point numbers.

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 of the mantissas is performed in the same way as in fixed-point to provide a double-precision product. The double-precision answer is used in fixed-point numbers to increase the accuracy of the product.
- ✓ In floating-point, the range of a single-precision mantissa combined with the exponent is usually accurate enough so that only single precision numbers are maintained. Thus the half most significant bits of the mantissa product and the exponent will be taken together to form a single precision floating-point product.
- ✓ The multiplication algorithm can be subdivided into four parts:
    1. Check for zeros.
    2. Add the exponents.
    3. Multiply the mantissas.
    4. Normalize the product.
- ✓ The flowchart for floating-point multiplication is shown in Fig.
- ✓ The two operands are checked to determine if they contain a zero. If either operand is equal to zero, the product in the AC is set to zero and the operation is terminated. If neither of the operands is equal to zero, the process continues with the exponent addition.
- ✓ 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. The multiplication of the mantissas is done as in the fixed-point case with the product residing in A and Q.
- ✓ Overflow cannot occur during multiplication, so there is no need to check for it. 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. The multiplier and multiplicand were originally normalized and contained fractions. The smallest normalized operand is 0.1, so the smallest possible product is 0.01. Therefore, only one leading zero may occur. Although the low-order half of the mantissa is in Q, we do not use it for the floating-point product. Only the value in the AC is taken as the product.

Figure 10-16 Multiplication of floating-point numbers.



DIVISION

- ✓ Floating-point division requires that the exponents be subtracted and the mantissas divided. The mantissa division is done as in fixed-point except that the dividend has a single-precision mantissa that is placed in the AC. Remember that the mantissa dividend is a fraction and not an integer. For integer representation, a single-precision dividend must be placed in register Q and register A must be cleared. The zeros in A are to the left of the binary point and have no significance.
- ✓ In fraction representation, a single-precision dividend is placed in register A and register Q is cleared. The zeros in Q are to the right of the binary point and have no significance.
- ✓ If the dividend is greater than or equal to the divisor, the dividend fraction is shifted to the right and its exponent incremented by 1. For normalized operands this is a sufficient operation to ensure that no mantissa divide dividend alignment overflow will occur. The operation above is referred to as a dividend alignment.

✓ The division of two normalized floating-point numbers will always result in a normalized quotient provided that a dividend alignment is carried out before the division. Therefore, unlike the other operations, the quotient obtained after the division does not require a normalization.

The division algorithm can be 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.

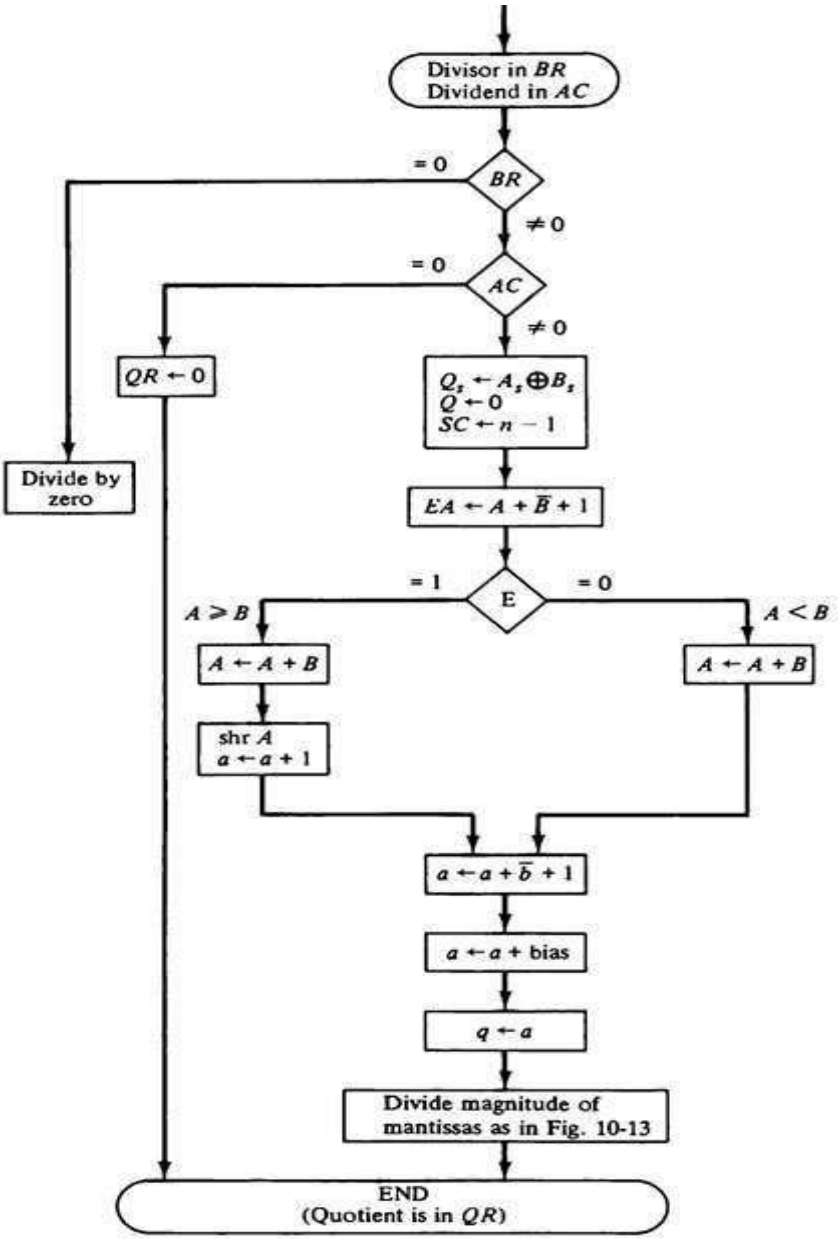


Figure 10-17 Division of floating-point numbers.

- ✓ 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. An alternative procedure would be to set the quotient in QR to the most positive number possible (if the dividend is positive) or to the most negative possible (if the dividend is negative).
- ✓ If the dividend in AC is zero, the quotient in QR is made zero and the operation terminates. If the operands are not zero, we proceed to determine the sign of the quotient and store it in Q_s.
- ✓ 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.
- ✓ The dividend alignment is similar to the divide-overflow check in the fixed-point operation. The proper alignment requires that the fraction dividend be smaller than the divisor. The two fractions are compared by a subtraction test.
- ✓ The carry in E determines their relative magnitude. The dividend fraction is restored to its original value by adding the divisor.
- ✓ If A >=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.
- ✓ Next, 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. 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 to 1.

DECIMAL ARITHMETIC UNIT

- ✓ The user of a computer prepares data with decimal numbers and receives results in decimal form.
- ✓ 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.
- ✓ Many computers have hardware for arithmetic calculations with both binary and 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.

BCD ADDER

- ✓ 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 and are labeled by symbols K, Z8, Z4,, Z2, and Z1. K is the carry and the subscripts under the letter Z represents the weights 8, 4, 2, and 1 that can be assigned to the four bits in the BCD code.

TABLE 10-4 Derivation of BCD Adder

Binary Sum					BCD Sum					
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	Decimal
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

- ✓ When the binary sum is equal to or less than 1001, the corresponding BCD number is identical 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.

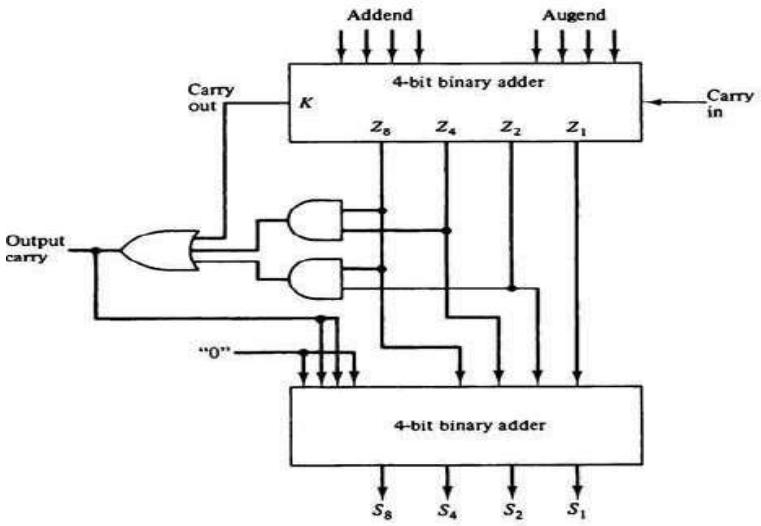
- ✓ One method of adding decimal numbers in BCD would be to employ one 4-bit binary adder and perform the arithmetic operation one digit at a time.
- ✓ If the result is equal or greater than 1010, it is corrected by adding 0110 to the binary sum.
- ✓ The logic circuit that detects the necessary correction is needed when the binary sum has an output carry K = 1. The other six combinations from 1010 to 1111 that need a correction have a 1 in position Z8. To distinguish them from binary 1000 and 1001 which also have a 1 in position Z8 we specify further that either Z4 or Z2 must have a 1. The condition for a correction and an output-carry can be expressed by the Boolean function

$$C = K + Z8Z4 + Z8Z2$$

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.

Figure 10-18 Block diagram of BCD adder.



- ✓ 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.

BCD SUBTRACTION

- ✓ Subtraction is performed by taking the 9's or 10's complement of the subtrahend and adding it to the minuend.
- ✓ 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.
- ✓ 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.
- ✓ The Boolean functions for the 9' s complementer circuit are

X1= B1M'+B1'M

X2=B2

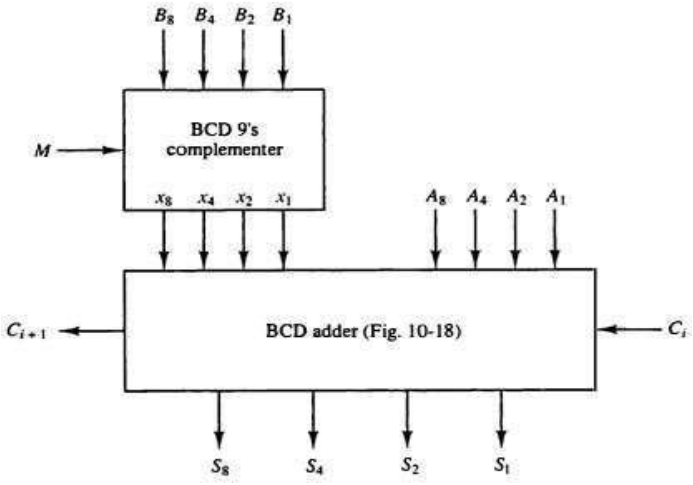
X4 = B4 M ' + (B4' B2 + B4 B2') M

X8 = B8 M ' + B8' B4' B2' M

From these equations we say 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 o r subtract two BCD digits is shown in Fig.

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



- ✓ 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.

11.6 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 11.4 and are labeled by symbols K, Z_8, Z_4, Z_2, and Z_1. 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 and therefore no conversion is needed. When the binary sum is greater than 1001, we obtain a nonvalid BCD representation. The addition of binary 6 (0110) to the binary stun 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 stun. 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 stun has an output carry $K = 1$. The other six combinations from 1010 to 1111 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_8, we specify further that either Z_4 or Z_2 must have a 1. The condition for a correction and an output-carry can be expressed by the Boolean function

TABLE 11.4 Derivation of BCD Adder

	Binary sum				BCD sum					
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	Decimal
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

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. 11.18. 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

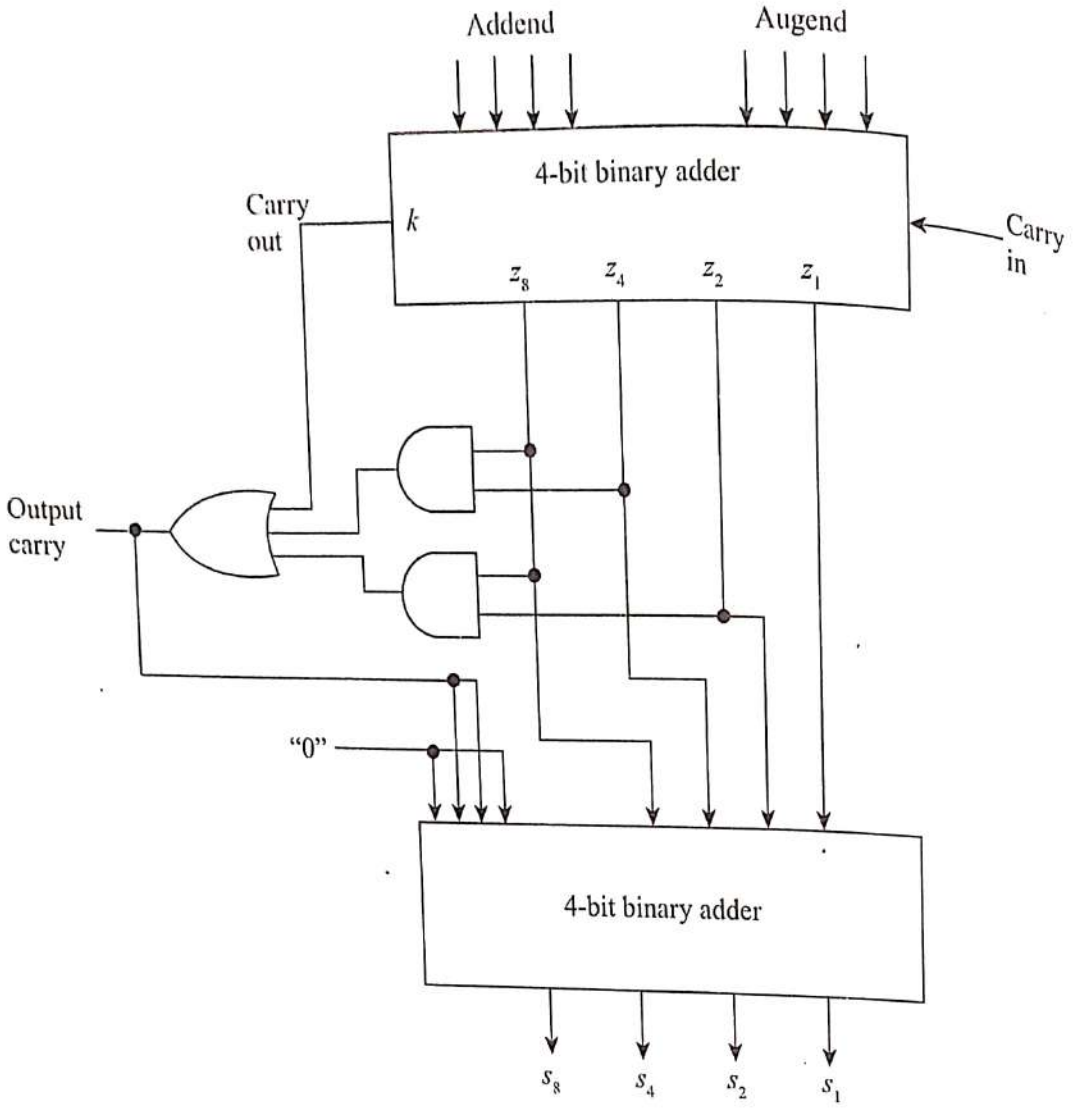


FIGURE 11.18 Block diagram of BCD adder.

adder circuit for the correction does not need all four full-adders, and this circuit can be optimized.

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 exdusive-OR of B_2 and B_4 is 1; and x_8 is 1 when $B_8B_4B_2 = 000$. The Boolean functions for the 9's complementer circuit are

$$x_1 = B_1M + B_1'M$$

$$x_2 = B_2$$

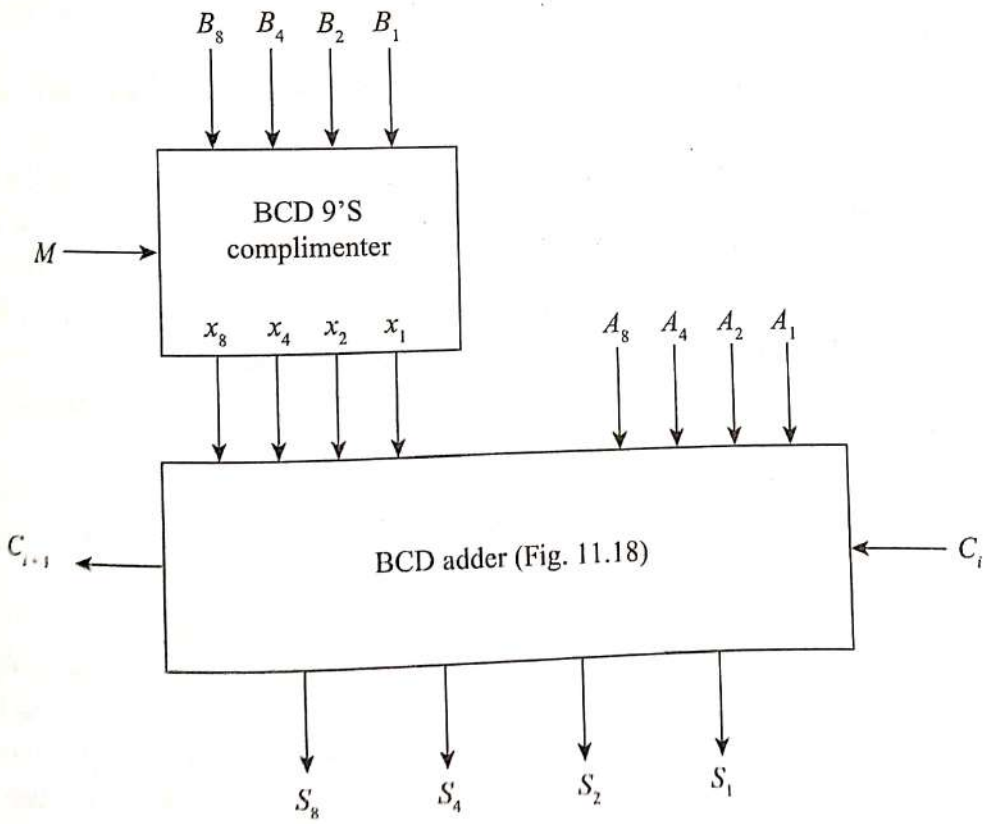


FIGURE 11.19 One stage of a decimal arithmetic unit.

$$x_4 = B_4 M' + \left(B_4' B_2 + B_4' B_2'\right) M$$

$$x_8 = B_8 M' + B_8' B_4 B_2' M$$

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. 11.19. 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_{i+1} from one stage must be connected to the input carry C_i 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 stun 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.

11.7 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 11.5, 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 $A \leftarrow A + \overline{B} + 1$ 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 10 states from 0000 to 1001 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.

TABLE 11.5 Decimal Arithmetic Microoperation Symbols

Symbolic designation	Description
$A \leftarrow A + B$	Add decimal numbers and transfer sum into A
$\overline{B}$	9's complement of B
$A \leftarrow A + \overline{B} + 1$	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

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

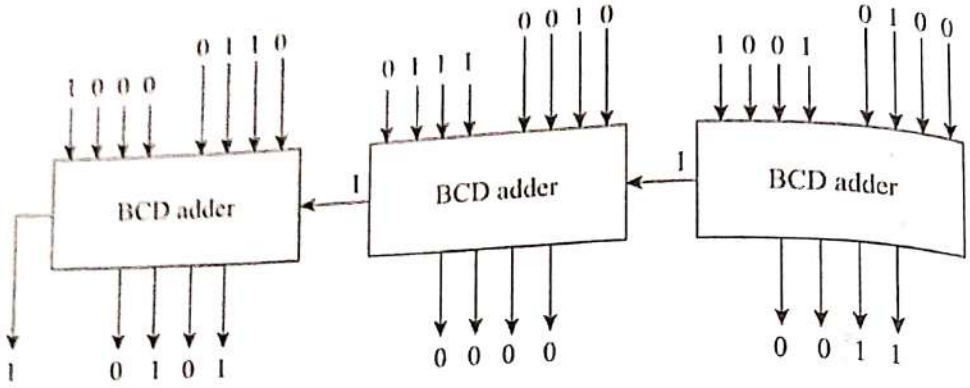
The microoperation *dshr A* shifts the decimal number one digit to the right to give 0786. This shift is over the four bits and changes the content of the register into

0000 0111 1000 0110

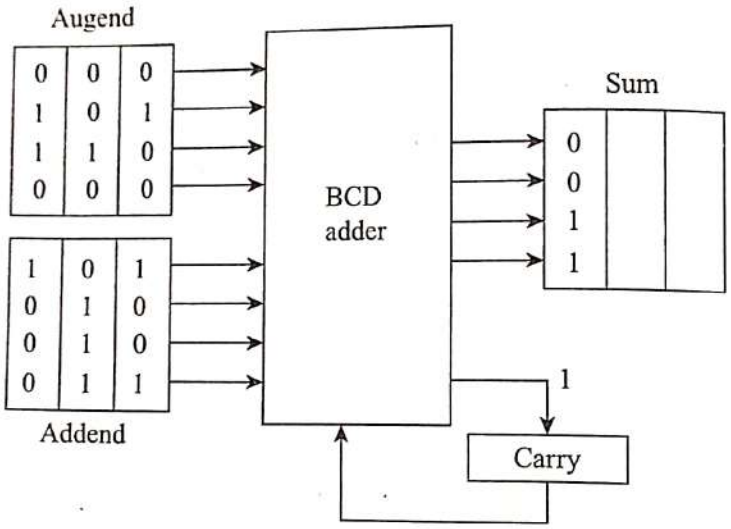
Addition and Subtraction

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. 11.19.

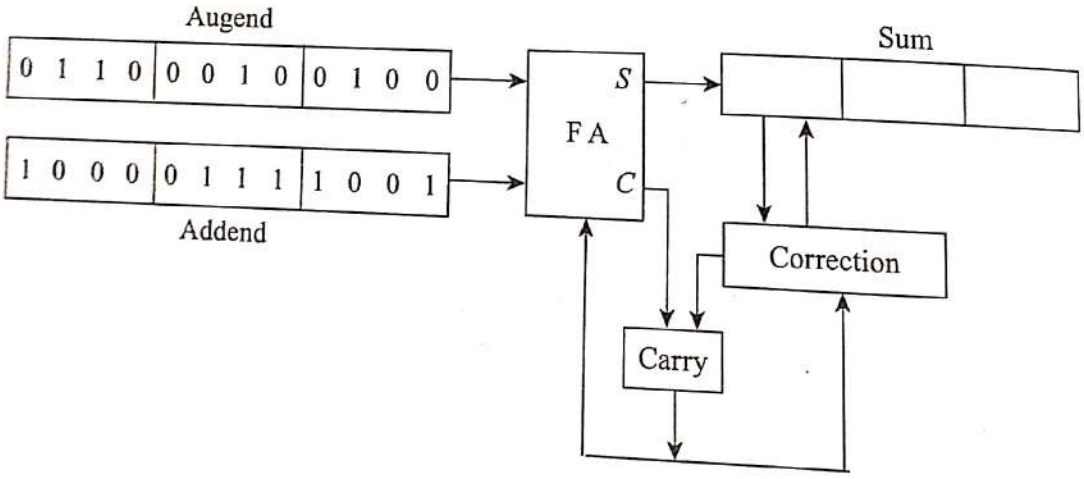
Decimal data can be added in three different ways, as shown in Fig. 11.20. The parallel method uses a decimal arithmetic unit composed of as many BCD adders as there are digits in the number. The sum is formed in parallel and requires only one microoperation. In the digit-serial bit-parallel method, 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.



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



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



(c) All serial decimal addition

FIGURE 11.20 Three ways of adding decimal numbers.

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

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. 11.21. 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_s, B_s, and Q_s.

Registers A and B have four more bits designated by A_e and B_e that provide an extension of one more digit to the registers. The BCD arithmetic unit adds the five digits in parallel and places the sum in the five-digit A register. The end-carry goes to flip-flop E. The purpose of digit A_e is to accommodate an overflow while adding the multiplicand to the partial product during multiplication. The purpose of digit B_e is to form the 9's complement of the divisor when subtracted from the partial remainder during the division operation. The least significant digit in register Q is denoted by Q_L. This digit can be incremented or decremented.

A decimal operand coming from memory consists of 17 bits. One bit (the sign) is transferred to B_s and the magnitude of the operand is

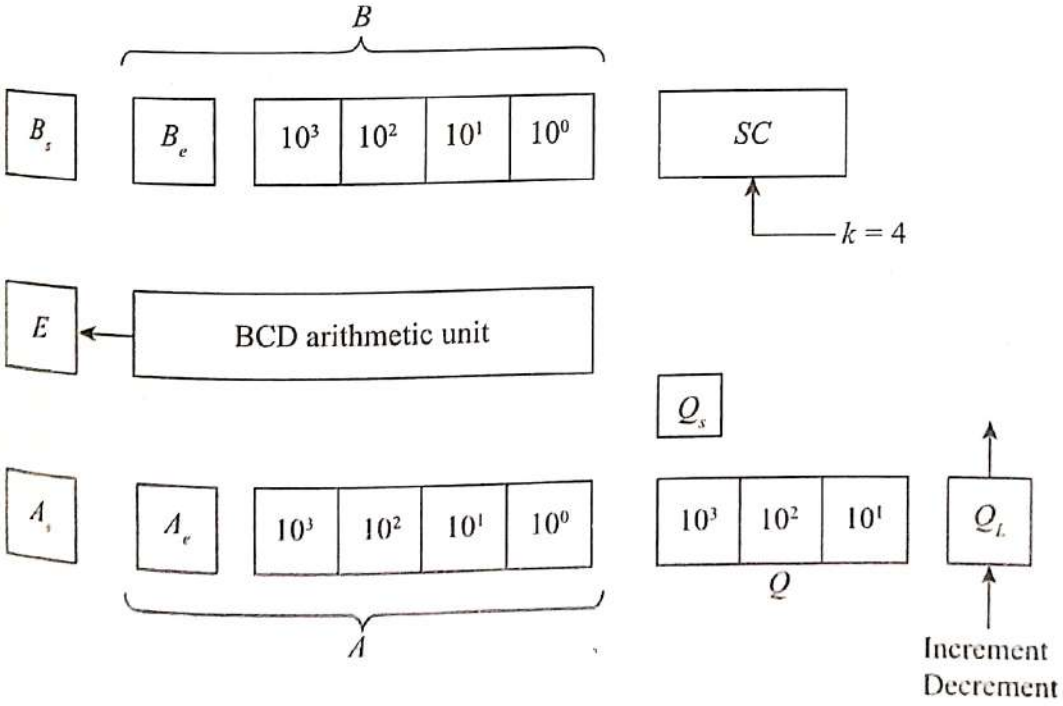


FIGURE 11.21 Registers for decimal arithmetic multiplication and division.

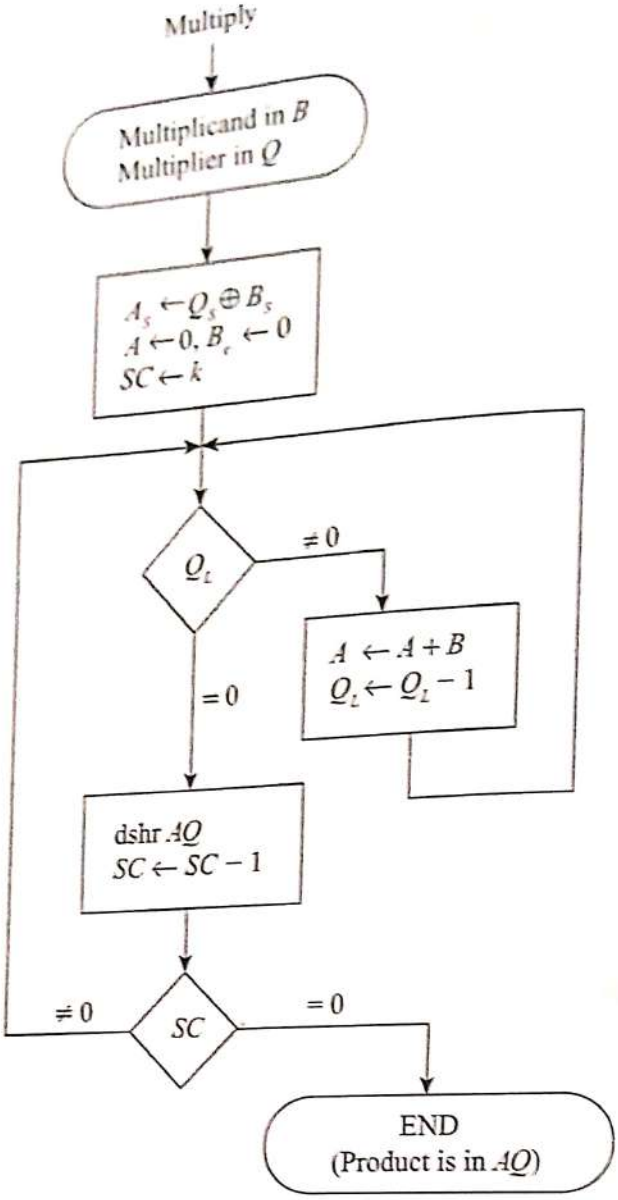


FIGURE 11.22 Flowchart for decimal multiplication.

placed in the lower 16 bits of B. Both B_e and A_e are cleared initially. The result of the operation is also 17 bits long and does not use the A_e part of the A register.

The decimal multiplication algorithm is shown in Fig. 11.22. Initially, the entire A register and B_e 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_L is checked. If it is not equal to 0, the multiplicand in B is added to the partial product in A once and Q_L is decremented. Q_L 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_e 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_e and transfers the next multiplier quotient into Q_L. The process is then repeated k times to form a double-length product in AQ.

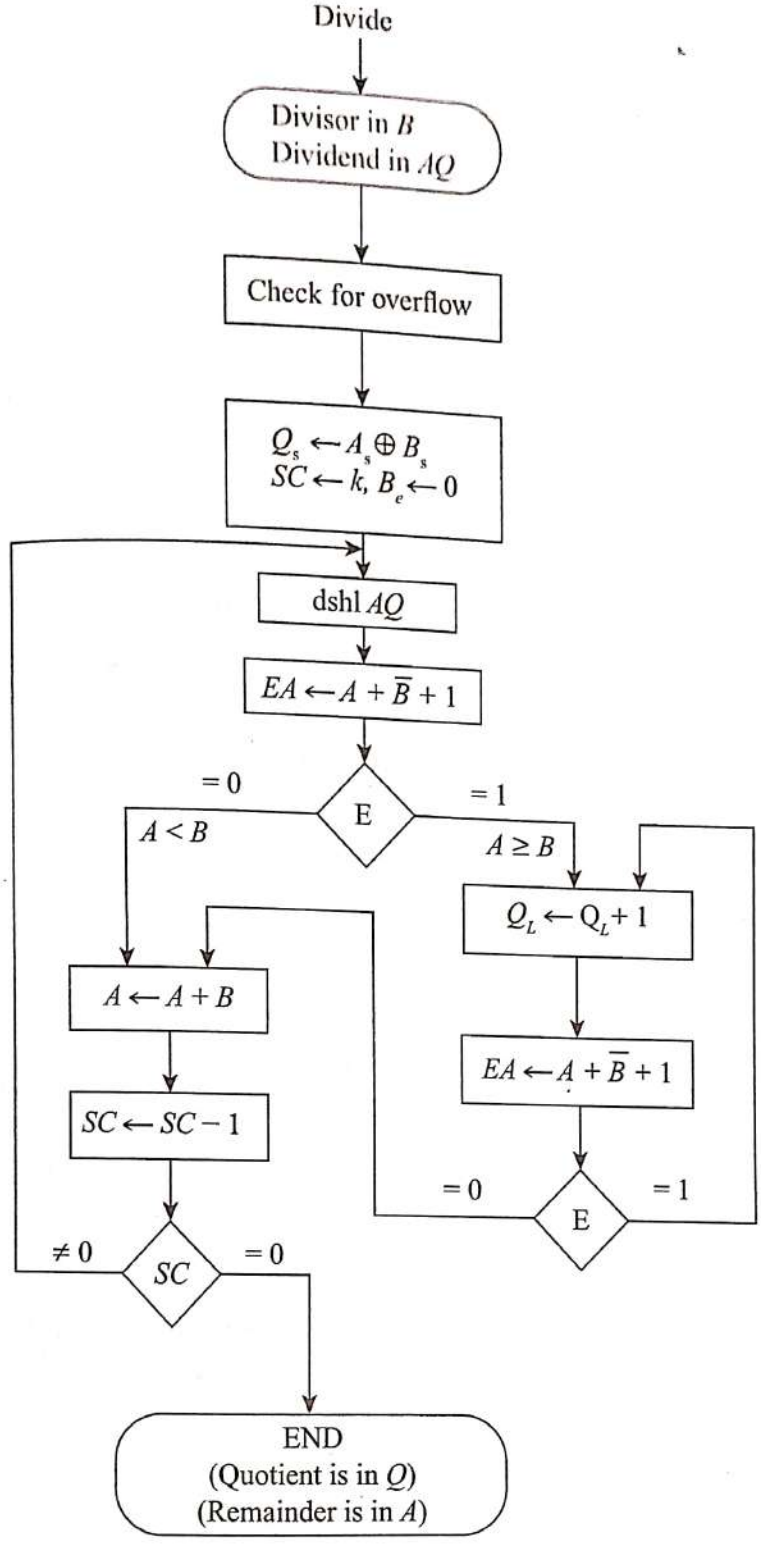


FIGURE 11.23 Flowchart for decimal division.

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. 11.23. 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_e. The divisor is then subtracted by adding its 10's complement value. Since B_e 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 Q_L stays at 0 (inserted there during the shift). If $E = 1$, it signifies that $A \geq B$. The quotient digit in Q_L 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.

UNIT- IV

INPUT-OUTPUT INTERFACE

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

  1. Peripherals are electromechanical and electromagnetic devices and their manner of operation is different from the operation of the CPU and memory, which are electronic devices. Therefore, a conversion of signal values may be required.

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

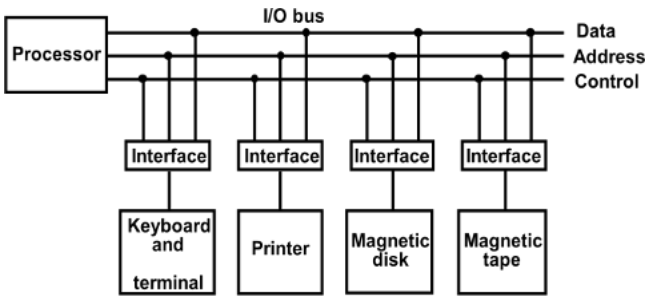
  3. Data codes and formats in peripherals differ from the word format in the CPU and memory.

  4. The operating modes of peripherals are different from each other and each must be controlled so as not to disturb the operation of other peripherals connected to the CPU.

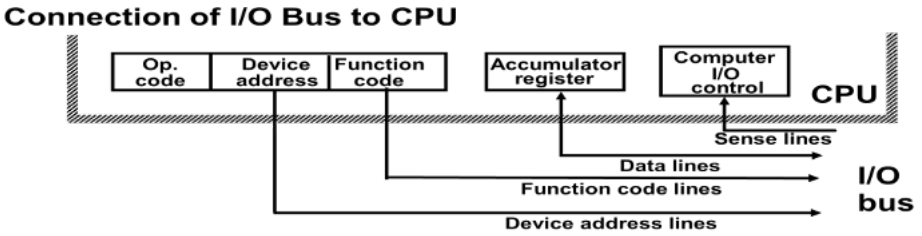
- ✓ To resolve these differences, computer systems include special hardware components between the CPU and peripherals to supervise and synchronize all input and output transfers. These components are called interface units because they interface between the processor bus and the peripheral device. In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

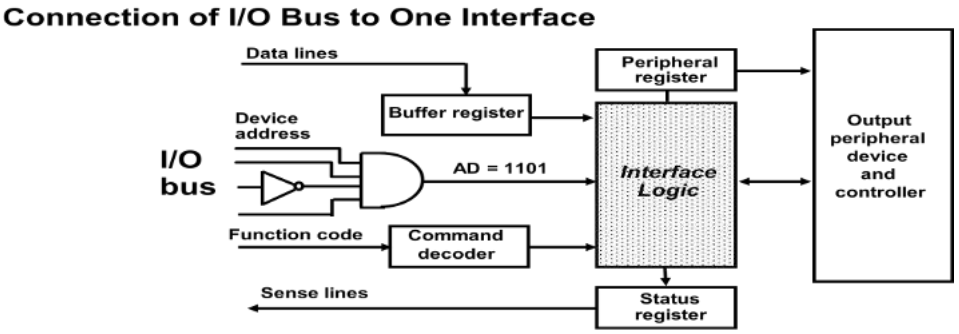
I/O BUS AND INTERFACE MODULES

- ✓ A typical communication link between the processor and several peripherals is shown in Fig.
- ✓ The I/O bus consists of data lines, address lines, and control lines.
- ✓ Each peripheral device has associated with it an interface unit. Each interface decodes the address and control received from the I/O bus, interprets them for the peripheral, and provides signals for the peripheral controller. It also synchronizes the data flow and supervises the transfer between peripheral and processor. Each peripheral has its own controller that operates the particular electromechanical device.



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





- ✓ At the same time the processor provides a function code in the control lines.
- ✓ There are four types of commands that an interface may receive. They are classified as control, status, data output, and data input.
- ✓ A control command is issued to activate the peripheral and to inform it what to do.
- ✓ A status command is used to test various status conditions in the interface and the peripheral.
- ✓ A data output command causes the interface to respond by transferring data from the bus into one of its registers.
- ✓ 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

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

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

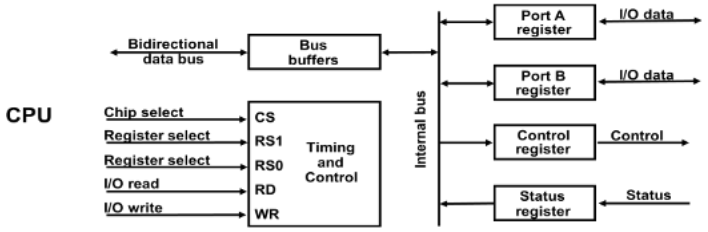
  In the first method, the computer has independent sets of data, address, and control buses, one for accessing memory and the other for I/O. This is done in computers that provide a separate I/O processor (IOP) in addition to the central processing unit (CPU). The memory communicates with both the CPU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

ISOLATED VERSUS MEMORY-MAPPED I/O

- ✓ Many computers use one common bus to transfer information between memory or I/O and the CPU. The distinction between a memory transfer and I/O transfer is made through separate read and write lines. The CPU specifies whether the address on the address lines is for a memory word or for an interface register by enabling one of two possible read or write lines. The I/O read and I/O write control lines are enabled during an I/O transfer. The memory read and memory write control lines are enabled during a memory transfer.
- ✓ In the **isolated I/O** configuration, the CPU has distinct input and output instructions, and each of these instructions is associated with the address of an interface register. When the CPU fetches and decodes the operation code of an input or output instruction, it places the address associated with the instruction into the common address lines. At the same time, it enables the I/O read (for input) or I/O write (for output) control line. This informs the external components that are attached to the common bus that the address in the address lines is for an interface register and not for a memory word. On the other hand, when the CPU is fetching an instruction or an operand from memory, it places the memory address on the address lines and enables the memory read or memory write control line. This informs the external components that the address is for a memory word and not for an I/O interface.
- ✓ The other alternative is to use the same address space for both memory and I/O. This is the case in computers that employ only one set of read and write signals and do not distinguish between memory and I/O addresses. This configuration is referred to as **memory mapped I/O**. The computer treats an interface register as being part of the memory system.
- ✓ In a memory-mapped I/O organization there is no specific input or output instructions. The CPU can manipulate I/O data residing in interface registers with the same instructions that are used to manipulate memory words. Each interface is organized as a set of registers that respond to read and write requests in the normal address space. Typically, a segment of the total address space is reserved for interface registers, but in general, they can be located at any address as long as there is not also a memory word that responds to the same address.
- ✓ Computers with memory-mapped I/O can use memory-type instructions to access I/O data. It allows the computer to use the same instructions for either input-output transfers or for memory transfers.
- ✓ The advantage is that the load and store instructions used for reading and writing from memory can be used to input and output data from I/O registers.
- ✓ In a typical computer, there are more memory-reference instructions than I/O instructions. With memory mapped I/O all instructions that refer to memory are also available for I/O. .

EXAMPLE OF I/O INTERFACE

- ✓ An example of an I/O interface unit is shown in block diagram



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

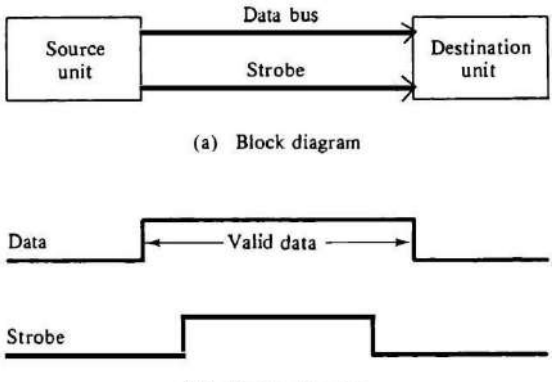
- ✓ It consists of two data registers called ports, a control register, a status register, bus buffers, and timing and control circuits. The interface communicates with the CPU through the data bus.
- ✓ The chip select and register select inputs determine the address assigned to the interface. The I/O read and write are two control lines that specify an input or output, respectively.
- ✓ The four registers communicate directly with the I/O device attached to the interface. 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 command is passed to the I/O device by sending a word to the appropriate interface register. In a system like this, the function code in the I/O bus is not needed because control is sent to the control register, status information is received from the status register, and data are transferred to and from ports A and B registers. Thus the transfer of data, control, and status information is always via the common data bus.
- ✓ The distinction between data, control, or status information is determined from the particular register with which the CPU communicates.
- ✓ The control register receives control information from the CPU. By loading appropriate bits into the control register, the interface and the I/O device attached to it can be placed in a variety of operating modes.
- ✓ The interface registers communicate with the CPU through the bidirectional data bus.
- ✓ The address bus selects the interface unit through the chip select and the two register select inputs. A circuit must be provided externally (usually, a decoder) to detect the address assigned to the interface registers. This circuit enables the chip select (CS) input when the interface is selected by the address bus. The two register select inputs RS1 and RS0 are usually connected to the two least significant lines of the lines address bus. These two inputs select one of the four registers in the interface as specified in the table accompanying the diagram.
- ✓ The content of the selected register is transfer into the CPU via the data bus when the I/O read signal is enabled. The CPU transfers binary information into the selected register via the data bus when the I/O write input is enabled.

ASYNCHRONOUS DATA TRANSFER

- ✓ The internal operations in a digital system are synchronized by means of clock pulses supplied by a common pulse generator.
- ✓ If the registers in the interface share a common clock with the CPU registers, the transfer between the two units is said to be synchronous.
- ✓ In most cases, the internal timing in each unit is independent from the other in that each uses its own private clock for internal registers. In that case, the two units are said to be asynchronous to each other.
- ✓ Asynchronous data transfer between two independent units requires that control signals be transmitted between the communicating units to indicate the time at which data is being transmitted.
- ✓ One way of achieving this is by means of a **strobe pulse** supplied by one of the units to indicate to the other unit when the transfer has to occur. Another method commonly used is to accompany each data item being transferred with a control signal that indicates the presence of data in the bus. The unit receiving the data item responds with another control signal to acknowledge receipt of the data. This type of agreement between two independent units is referred to as **handshaking.**

STROBE CONTROL

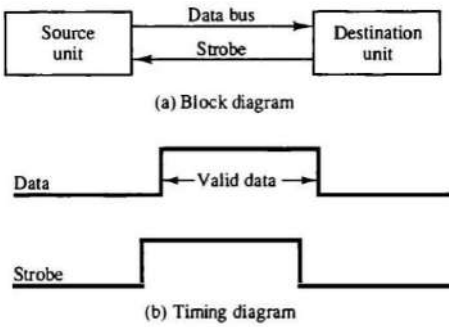
- ✓ The strobe control method of asynchronous data transfer employs a single control line to time each transfer. The strobe may be activated by either the source or the destination unit.
- ✓ The data bus carries the binary information from source unit to the destination unit. Typically, the bus has multiple lines to transfer an entire byte or word. The strobe is a single line that informs the destination unit when a valid data word is available in the bus.
- ✓ As shown in the timing diagram the source unit first places the data on the data bus. After a brief delay to ensure that the data settle to a steady value, the source activates the strobe pulse.



Source-initiated strobe for data transfer.

- ✓ The information on the data bus and the strobe signal remain in the active state for a sufficient time period to allow the destination unit to receive the data. Often, the destination unit uses the falling edge of the strobe pulse to transfer the contents of the data bus into one of its internal registers.

- ✓ The 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. The falling edge of the strobe pulse can be used again to trigger a destination register. The destination unit then disables the strobe



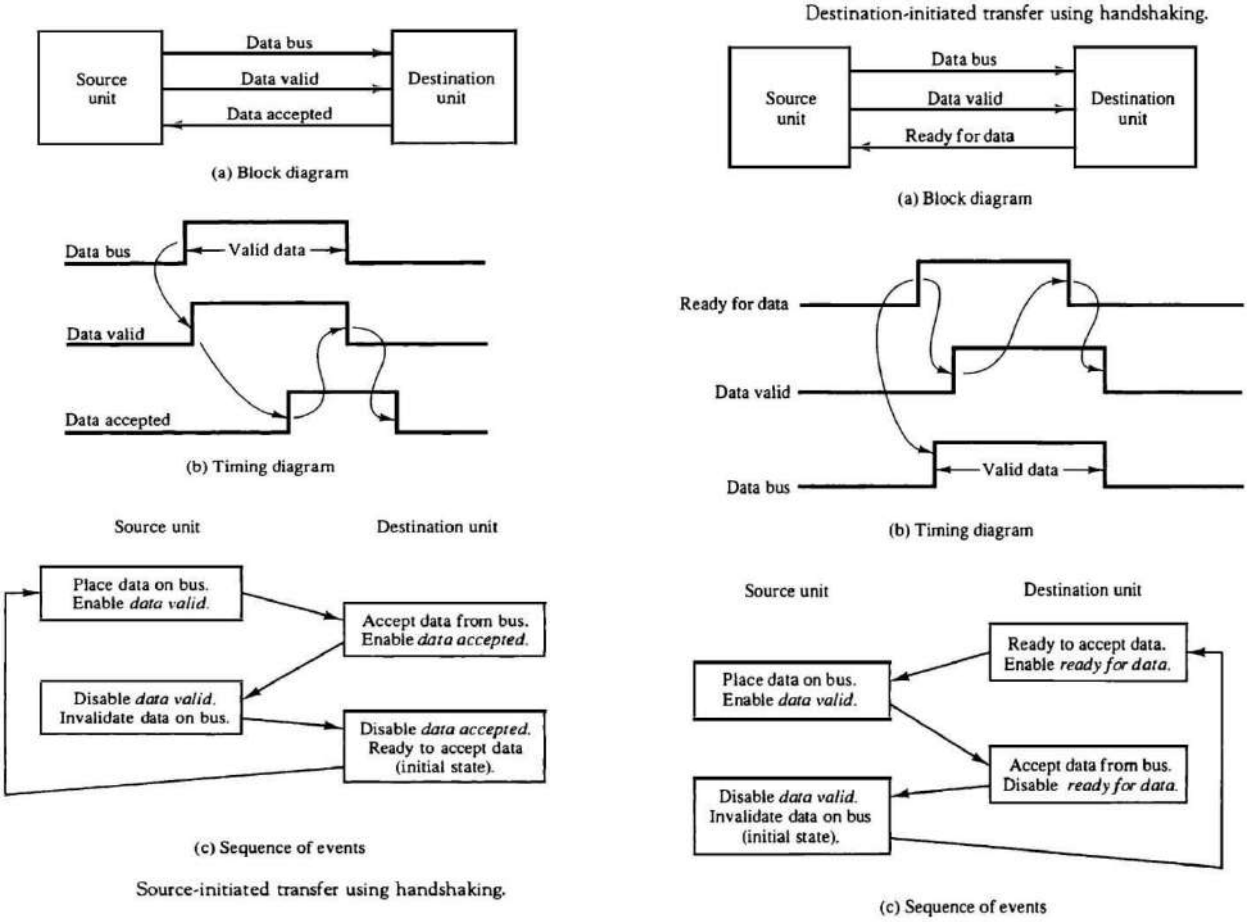
Destination-initiated strobe for data transfer.

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

HANDSHAKING

- ✓ The disadvantage of the strobe method is that the source unit that initiates the transfer has no way of knowing whether the destination unit has actually received the data item that was placed in the bus. Similarly, a destination unit that initiates the transfer has no way of knowing whether the source unit has actually placed the data on the bus. The handshake method solves this problem by introducing a second control signal that provides a reply to the unit that initiates the transfer.
- ✓ The basic principle of the handshaking method of data transfer is as follows. One control line is in the same direction as the data flow in the bus from the source to the destination. It is used by the source unit to inform the destination unit whether there are valued data in the bus.
- ✓ The other control line is in the other direction from the destination to the source. It is used by the destination unit to inform the source whether it can accept data. The sequence of control during the transfer depends on the unit that initiates the transfer.
- ✓ The two handshaking lines are **data valid,** which is generated by the source unit, and **data accepted,** generated by the destination unit.
- ✓ The timing diagram shows the exchange of signals between the two units. In the destination-initiated transfer the source does not place data on the bus until after it receives the ready for data signal from the destination unit.
- ✓ The handshaking scheme provides a high degree of flexibility and reality because the successful completion of a data transfer relies on active participation by both units. If one unit is faulty, the data transfer will not be completed. Such an error can be detected by means of a timeout mechanism, which produces an alarm if the data transfer is not completed within a predetermined time. The timeout is implemented by means of an internal clock that starts

counting time when the unit enables one of its handshaking control signals. If the return handshake signal does not respond within a given time period, the unit assumes that an error has occurred

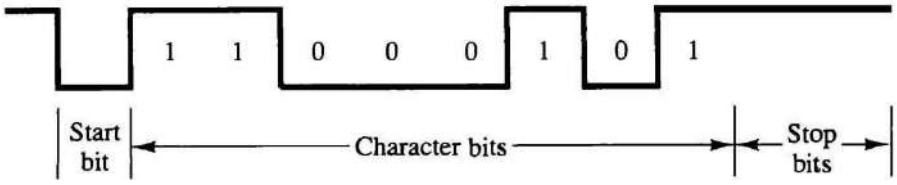


ASYNCHRONOUS SERIAL TRANSFER

- ✓ The transfer of data between two units may be done in parallel or serial. In parallel data transmission, each bit of the message has its own path and the total message is transmitted at the same time.
- ✓ In serial data transmission, each bit in the message is sent in sequence one at a time.
- ✓ Parallel transmission is faster but requires many wires. It is used for short distances and where speed is important. Serial transmission is slower but is less expensive since it requires only one pair of conductors.
- ✓ Serial transmission can be synchronous or asynchronous. In synchronous transmission, the two units share a common clock frequency and bits are transmitted continuously at the rate dictated by the clock pulses. In long-distant serial transmission, each unit is driven by a separate clock of the same frequency. Synchronization signals are transmitted periodically between the two units to keep their clocks in step with each other.

- ✓ In asynchronous transmission, binary information is sent only when it is available and the line remains idle when there is no information to be transmitted.
- ✓ Serial asynchronous data transmission technique used in many interactive terminals employs special bits that are inserted at both ends of the character code. With this technique, each character consists of three parts: a start bit, the character bits, and stop bits.
- ✓ The convention is that the transmitter rests at the 1-state when no characters are transmitted. The first bit, called the start bit, is always a 0 and is used to indicate the beginning of a character. The last bit called the stop bit is always a 1.
- ✓ An example of this format is shown in Fig.

Asynchronous serial transmission.



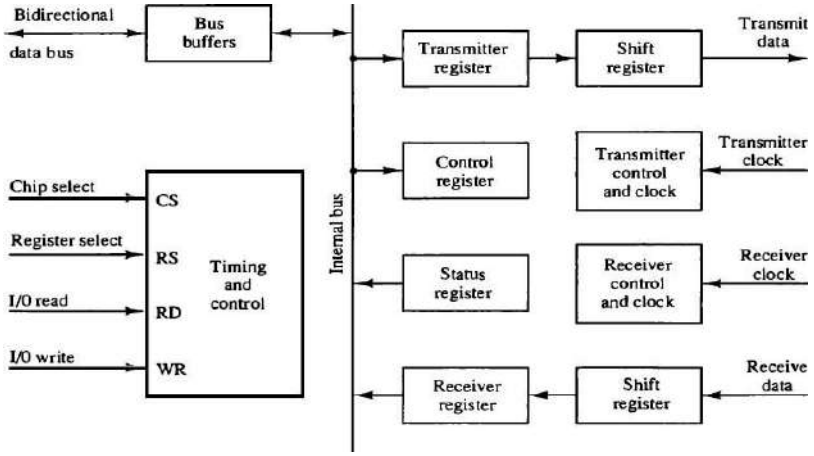
- ✓ A transmitted character can be detected by the receiver from knowledge of the transmission rules:

  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 gives from 1 to 0. A clock in the receiver examines the line at proper bit times. The receiver knows the transfer rate of the bits and the number of character bits to accept. After the character bits are transmitted, one or two stop bits are sent. The stop bits are always in the 1-state and frame the end of the character to signify the idle or wait state.
- ✓ At the end of the character the line is held at the 1-state for a period of at least one or two bit times so that both the transmitter and receiver can resynchronize. The length of time that the line stays in this state depends on the amount of time required for the equipment to resynchronize.
- ✓ Some older electromechanical terminals use two stop bits, but newer terminals use one stop bit.
- ✓ The line remains in the 1-state until another character is transmitted. The stop time ensures that a new character will not follow for one or two bit times.

Asynchronous Communication Interface

- ✓ The block diagram of an asynchronous communication interface is shown in Fig.



CS	RS	Operation	Register selected
0	×	×	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

Block diagram of a typical asynchronous communication interface.

- ✓ It functions as both a transmitter and a receiver. 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 the 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 bits in the status register are used for input and output flags and for recording certain errors that may occur during the transmission. 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 and 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.
- ✓ The operation of the asynchronous communication interface is initialized by the CPU by sending a byte to the control register. The initialization procedure places the interface in a specific mode of operation as it defines certain parameters such as the baud rate to use, how many bits are in each character, whether to generate and check parity, and how many stop bits

are appended to each character. 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 operation of the transmitter portion of the interface is as follows. 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 character is transferred in parallel from the transmitter register to the shift register and the appropriate number of stop bits are appended into the shift register. The transmitter register is then marked empty. The character can now be transmitted one bit at a time by shifting the data in the shift register at the specified baud rate. The CPU can transfer another character to the transmitter register after checking the flag in the status register. The interface is said to be double buffered because a new character can be loaded as soon as the previous one starts transmission.
- ✓ The operation of the receiver portion of the interface is similar. The receive data input is in the 1-state when the line is idle. The receiver control monitors the receive-data line for a 0 signal to detect the occurrence of a start bit. Once a start bit has been detected, the incoming bits of the character are shifted into the shift register at the prescribed baud rate. After receiving the data bits, the interface checks for the parity and stop bits. The character without the start and stop bits is then transferred in parallel from the shift register to the receiver register. 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, it reads the data from the receiver register. The interface checks for any possible errors during transmission and sets appropriate bits in the status register. The CPU can read the status register at any time to check if any errors have occurred. Three possible errors that the interface checks during transmission are parity error, framing error, and overrun error. Parity error occurs if the number of l's in the received data is not the correct parity. A framing error occurs if the right number of stop bits is not detected at the end of the received character. 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. Overrun error results in a loss of characters in the received data stream.

First-In, First-Out Buffer

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. When placed between two units, the FIFO can accept data from the source unit at one rate of transfer and deliver the data to the destination unit at another rate. If the source unit is slower than the destination unit, the buffer can be filled with data at a slow rate and later emptied at the higher rate. If the source is faster than the destination, the FIFO is useful for those cases where the source data arrive in bursts that fill out the buffer but the time between bursts is long enough for the destination unit to empty some or all the information from the buffer. Thus a FIFO buffer can be useful in some applications when data are transferred asynchronously. It piles up data as they come in and gives them away in the same order when the data are needed.

The logic diagram of a typical 4×4 FIFO buffer is shown in Fig. 11-9. It consists of four 4-bit registers RI, $I = 1, 2, 3, 4$, and a control register with

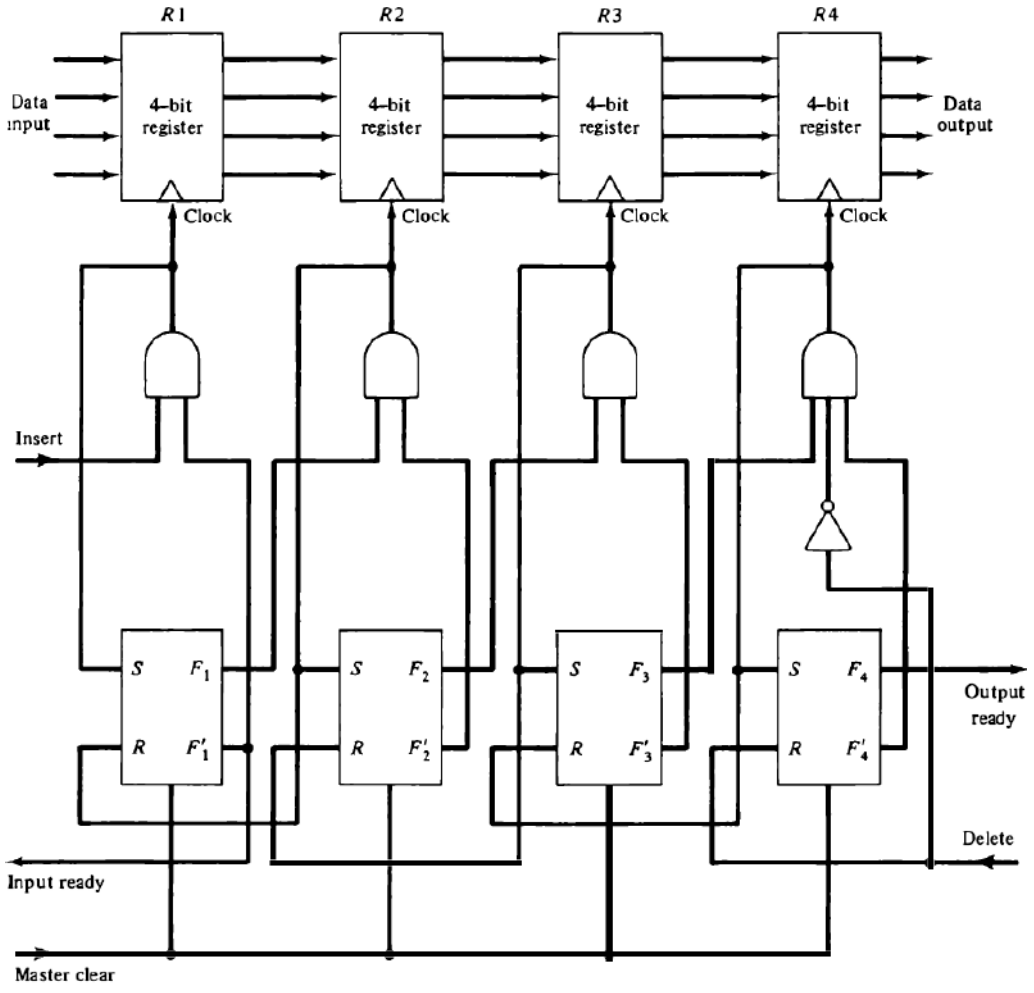


Figure 11-9 Circuit diagram of 4×4 FIFO buffer.

flip-flops F_i, $i = 1, 2, 3, 4$, 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 is set to 1 indicates that a 4-bit data word is stored in the corresponding register RI. A 0 in F_i indicates that the corresponding register does not contain valid data. The control register directs the movement of data through the registers. Whenever the F_i bit of the control register is set ($F_i = 1$) and the F_{i+1} bit is reset ($F'_{i+1} = 1$), a clock is generated causing register $R(I + 1)$ to accept the data from register RI. The same clock transition sets F_{i+1} to 1 and resets F_i to 0. This causes the control flag to move one position to the right together with the data. Data in the registers move down the FIFO toward the output as long as there are empty locations ahead of it. This ripple-through operation stops when the data reach a register RI with the next flip-flop F_{i+1} being set to 1, or at the last register $R4$. An overall master clear is used to initialize all control register flip-flops to 0.

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 $R1$ is empty. Data are loaded from the input lines by enabling the clock in $R1$ 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 $R2$ is empty. The data in $R1$ are transferred into $R2$ and F_1 is cleared. This enables the *input ready* line, indicating that the inputs are now available for another data word. 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 data falling through the registers stack up at the output end. 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 $R4$. The output data from $R4$ 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 output are no longer valid. Only after the *delete* signal goes back to 0 can the data from $R3$ move into $R4$. If the FIFO is empty, there will be no data in $R3$ and F_4 will remain in the reset state. Note that the two control lines *output ready* and *delete* constitute a source-initiated pair of handshake lines.

MODES OF TRANSFER

- ✓ Binary information received from an external device is usually stored in memory for later processing. Information transferred from the central computer into an external device originates in the memory unit. The CPU merely executes the I/O instructions and may accept the data temporarily, but the ultimate source or destination is the memory unit.
- ✓ Data transfer between the central computer and I/O devices may be handled in a variety of modes. Some modes use the CPU as an intermediate path; other transfer the data directly to and from the memory unit.
- ✓ Data transfer to and from peripherals may be handled in one of three possible modes:
    1. Programmed I/O
    2. Interrupt-initiated I/O
    3. Direct memory access (DMA)
- ✓ Programmed I/O operations are the result of I/O instructions written in the computer program. Each data item transfer is initiated by an instruction in the program. Usually, the transfer is to and from a CPU register and peripheral. Other instructions are needed to transfer the data to and from CPU and memory. Transferring data under program control requires constant monitoring of the peripheral by the CPU. Once a data transfer is initiated, the CPU is required to monitor the interface to see when a transfer can again be made. It is up to the programmed instructions executed in the CPU to keep close tabs on everything that is taking place in the interface unit and the I/O device.
- ✓ In the programmed I/O method, the CPU stays in a program loop until the I/O unit indicates that it is ready for data transfer. This is a time-consuming process since it keeps the processor busy needlessly. It can be avoided by using an interrupt facility and special commands to inform the interface to issue an interrupt request signal when the data are available from the device. In the meantime the CU can proceed to execute another program. The interface meanwhile keeps monitoring the device. When the interface determines that the device is ready for data transfer, it generates an interrupt request to the computer. Upon detecting the external interrupt signal, the CPU momentarily stops the task it is processing, branches to a service program to process the I/O transfer, and then returns to the task it was originally performing Transfer of data under programmed I/O is between CPU and peripheral.
- ✓ In direct memory access (DMA), the interface transfers data into and out of the memory unit through the memory bus. The CPU initiates the transfer by supplying the interface with the starting address and the number of words needed to be transferred and then proceeds to execute other tasks. When the transfer is made, the DMA requests memory cycles through the memory bus. When the request is granted by the memory controller, the DMA transfers the data directly

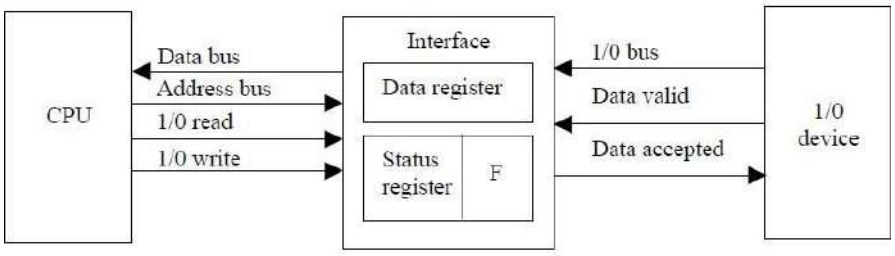
into memory. The CPU merely delays its memory access operation to allow the direct memory I/O transfer. Since peripheral speed is usually slower than processor speed, I/O-memory transfers are infrequent compared to processor access to memory.

- ✓ Many computers combine the interface logic with the requirements for direct memory access into one unit and call it an I/O processor (IOP). The IOP can handle many peripherals through a DMPA and interrupt facility. In such a system, the computer is divided into three separate modules: the memory unit, the CPU, and the IOP.

EXAMPLE OF PROGRAMMED I/O

- ✓ In the programmed I/O method, the I/O device does not have direct access to memory. A transfer from an I/O device to memory requires the execution of several instructions by the CPU, including an input instruction to transfer the data from the device to the CPU, and a store instruction to transfer the data from the CPU to memory. Other instructions may be needed to verify that the data are available from the device and to count the numbers of words transferred.
- ✓ An example of data transfer from an I/O device through an interface into the CPU is shown in Fig.

Data transfer form I/O device to CPU



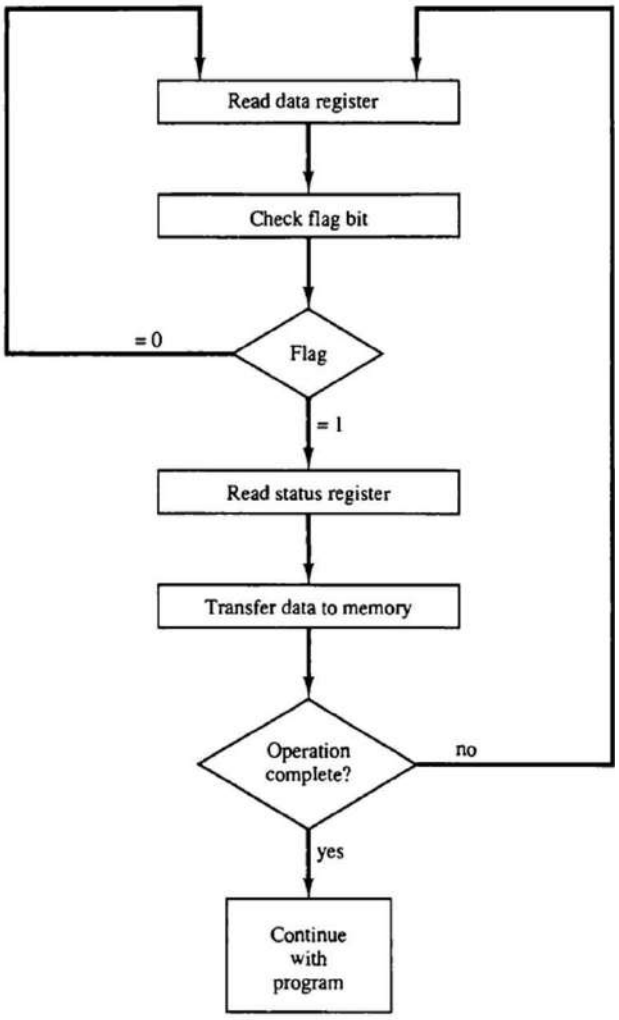
F = Flag bit

- ✓ The device transfers bytes of data one at a time as they are available. When a byte of data is available, the device places it in the I/O bus and enables its data valid line. The interface accepts the byte into its data register and enables the data accepted line. The interface sets a bit in the status register that we will refer to as an F or "flag" bit. The device can now disable the data valid line, but it will not transfer another byte until the data accepted line is disabled by the interface.
- ✓ A program is written for the computer to check the flag in the status register to determine if a byte has been placed in the data register by the I/O device. This is done by reading the status register into a CPU register and checking the value of the flag bit. If the flag is equal to 1, the CPU reads the data from the data register. The flag bit is then cleared to 0 by either the CPU or the interface, depending on how the interface circuits are designed. Once the flag is cleared, the interface disables the data accepted line and the device can then transfer the next data byte.

- ✓ A flowchart of the program that must be written for the CPU is shown in Fig.

It is assumed that the device is sending a sequence of bytes that must be stored in memory. The transfer of each byte requires three instructions:

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



Flowchart for CPU program to input data.

- ✓ Each byte is read into a CPU register and then transferred to memory with a store instruction. A common I/O programming task is to transfer a block of words form an I/O device and store them in a memory buffer.
- ✓ The programmed I/O method is particularly useful in small low-speed computers or in systems that are dedicated to monitor a device continuously. The difference in information transfer rate between the CPU and the I/O device makes this type of transfer inefficient.

INTERRUPT-INITIATED I/O

- ✓ An alternative to the CPU constantly monitoring the flag is to let the interface inform the computer when it is ready to transfer data. This mode of transfer uses the interrupt facility. While the CPU is running a program, it does not check the flag. However, when the flag is set, the computer is momentarily interrupted from proceeding with the current program and is informed of the fact that the flag has been set.
- ✓ The CPU deviates from what it is doing to take care of the input or output transfer. After the transfer is completed, the computer returns to the previous program to continue what it was doing before the interrupt.
- ✓ The CPU responds to the interrupt signal by storing the return address from the program counter into a memory stack and then control branches to a service routine that processes the

required I/O transfer. The way that the processor chooses the branch address of the service routine varies from tone unit to another. In principle, there are two methods for accomplishing this. One is called vectored interrupt and the other, no vectored interrupt. In a non vectored interrupt, the branch address is assigned to a fixed location in memory. In a vectored interrupt, the source that interrupts supplies the branch information to the computer. This information is called the interrupt vector. In some computers the interrupt vector is the first address of the I/O service routine. In other computers the interrupt vector is an address that points to a location in memory where the beginning address of the I/O service routine is stored.

SOFTWARE CONSIDERATIONS

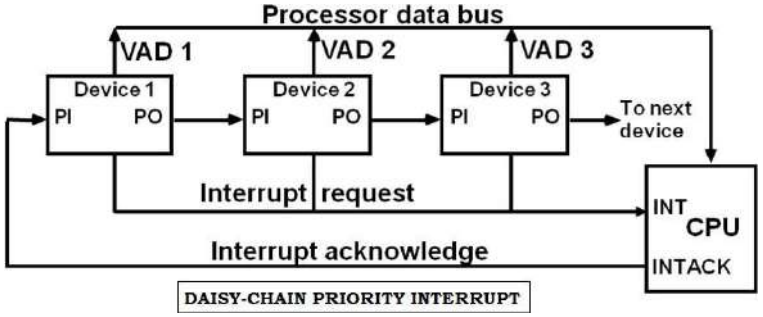
- ✓ The previous discussion was concerned with the basic hardware needed to interface I/O devices to a computer system. A computer must also have software routines for controlling peripherals and for transfer of data between the processor and peripherals. I/O routines must issue control commands to activate the peripheral and to check the device status to determine when it is ready for data transfer. Once ready, information is transferred item by item until all the data are transferred. In some cases, a control command is then given to execute a device function such as stop tape or print characters. Error checking and other useful steps often accompany the transfers.
- ✓ In interrupt-controlled transfers, the I/O software must issue commands to the peripheral to interrupt when ready and to service the interrupt when it occurs. In DMA transfer, the I/O software must initiate the DMA channel to start its operation.
- ✓ Software control of input-output equipment is a complex undertaking. For this reason I/O routines for standard peripherals are provided by the manufacturer as part of the computer system. They are usually included within the operating system. Most operating systems are supplied with a variety of I/O programs to support the particular line of peripherals offered for the computer. I/O routines are usually available as operating system procedures and the user refers to the established routines to specify the type of transfer required without going into detailed machine language programs.

PRIORITY INTERRUPT

- ✓ Data transfer between the CPU and an I/O device is initiated by the CPU. The CPU cannot start the transfer unless the device is ready to communicate with the CPU. The readiness of the device can be determined from an interrupt signal.
- ✓ Numbers of I/O devices are attached to the computer; several sources will request service simultaneously. The first task of the interrupt system is to identify the source of the interrupt and decide which device to service first
- ✓ A priority interrupts is a system to determine which interrupt is to be served first when two or more requests are made simultaneously. Also determines which interrupts are permitted to interrupt the computer while another is being serviced. Higher priority interrupts can make requests while servicing a lower priority interrupt
- ✓ Establishing the priority of simultaneous interrupts can be done by software or hardware.
- ✓ Priority Interrupt by Software(Polling)
  - Priority is established by the order of polling the devices(interrupt sources)
  - Flexible since it is established by software
  - Low cost since it needs a very little hardware
  - Very slow
- ✓ Priority Interrupt by Hardware
  - Require a priority interrupt manager which accepts all the interrupt requests to determine the highest priority request
  - Fast since identification of the highest priority interrupt request is identified by the hardware. Each interrupt source has its own interrupt vector to access directly to its own service routine
- ✓ The hardware priority function can be established by either a serial or a parallel connection of interrupt lines. The serial connection is also known as the daisy chaining method.

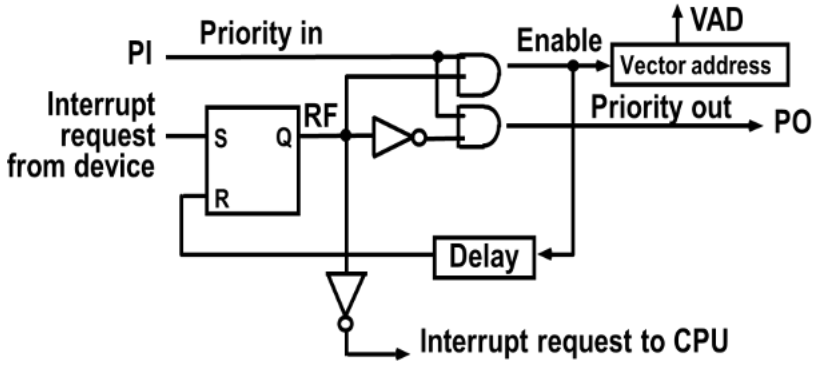
DAISY-CHAINING PRIORITY

- ✓ 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.



DAISY-CHAIN PRIORITY INTERRUPT

- ✓ 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 out) output only if device 1 is not requesting an interrupt.
- ✓ If device 1 has a pending interrupt, it blocks the acknowledge signal from the next device by placing a 0 in the PO output. It then proceeds to insert its own interrupt vector address (VAD) into the data bus for the CPU to use during the interrupt cycle.
- ✓ 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 following figure shows the internal logic that must be included with in each device when connected in the daisy-chaining scheme.



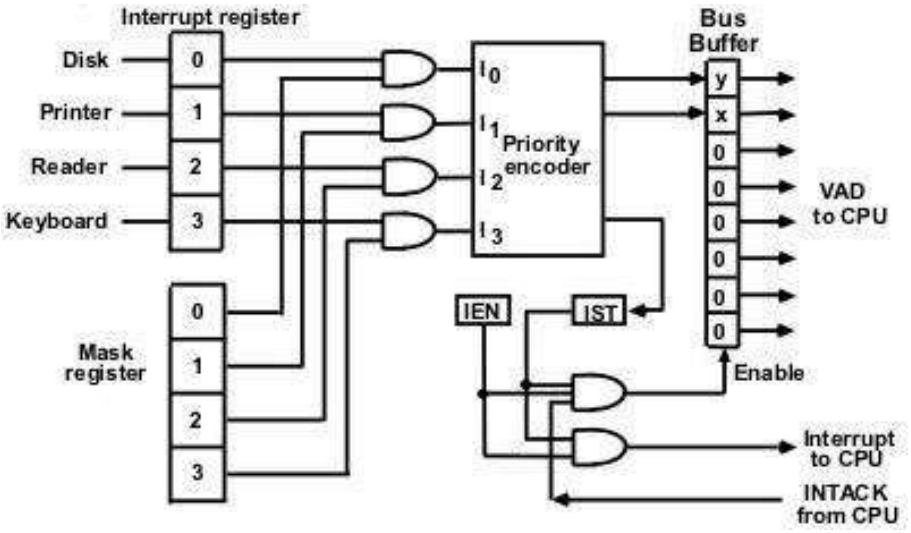
PI	RF	PO	Enable
0	0	0	0
0	1	0	0
1	0	1	0
1	1	1	1

Fig: One stage of the daisy- chain priority arrangement

- ✓ The device sets its RF flip-flop when it wants to interrupt the CPU. The output of the RF flip-flop goes through an open-collector inverter, a circuit that provides the wired logic for the common interrupt line.
- ✓ If PI = 0, both PO and the enable line to VAD are equal to 0, irrespective of the value of RF.
- ✓ If PI = 1 and RF = 0, then PO = 1 and the vector address is disabled. This condition passes the acknowledge signal to the next device through PO.
- ✓ The device is active when PI = 1 and RF = 1. This condition places a 0 in PO and enables the vector address for the data bus. It is assumed that each device has its own distinct vector address.
- ✓ The RF flip-flop is reset after a sufficient delay to ensure that the CPU has received the vector address.

PARALLEL PRIORITY INTERRUPT

- ✓ The parallel priority interrupt method uses a register whose bits are set separately by the interrupt signal from each device.
- ✓ Priority is established according to the position of the bits in the register. In addition to the interrupt register the circuit may include a mask register whose purpose is to control the status of each interrupt request.
- ✓ The mask register can be programmed to disable lower-priority interrupts while a higher-priority device is being serviced. It can also provide a facility that allows a high-priority device to interrupt the CPU while a lower-priority device is being serviced.
- ✓ The priority logic for a system of four interrupt sources is shown in Fig.



- ✓ It consists of an interrupt register whose individual bits are set by external conditions and cleared by program instructions.
- ✓ The mask register has the same number of bits as the interrupt register. By means of program instructions, it is possible to set or reset any bit in the mask register.
- ✓ Each interrupt bit and its corresponding mask bit are applied to an AND gate to produce the four inputs to a priority encoder. In this way an interrupt is recognized only if its corresponding mask bit is set to 1 by the program.
- ✓ The priority encoder generates two bits of the vector address, which is transferred to the CPU.
- ✓ Another output from the encoder sets an interrupt status flip-flop IST when an interrupt that is not masked occurs.
- ✓ The interrupt enable flip-flop IEN can be set or cleared by the program to provide an overall control over the interrupt system.
- ✓ The outputs of IST ANDed with IEN provide a common interrupt signal for the CPU.
- ✓ The interrupt acknowledge INTACK signal from the CPU enables the bus buffers in the output register and a vector address VAD is placed into the data bus.

Priority Encoder

- ✓ The priority encoder is a circuit that implements the priority function. The logic of the priority encoder is such that if two or more inputs arrive at the same time, the input having the highest priority will take precedence.

Priority Encoder Truth table

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

Interrupt Cycle

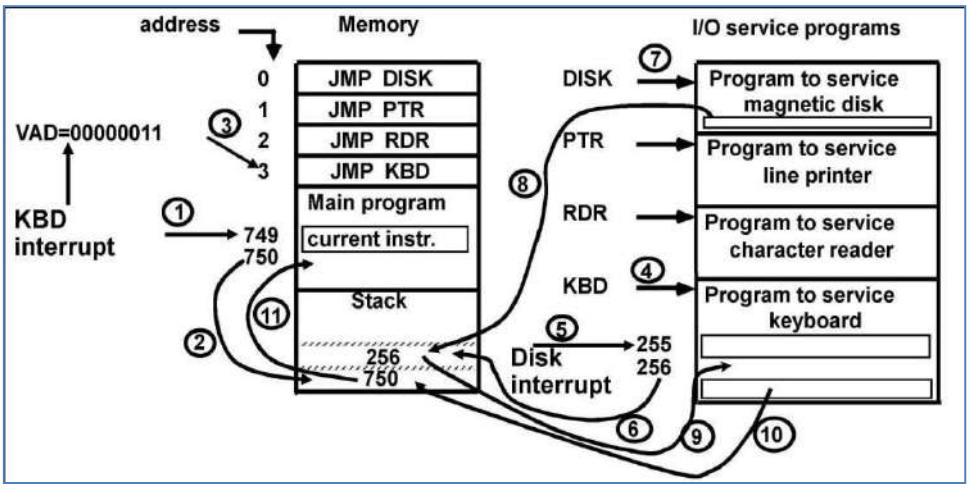
- ✓ The interrupt enable flip-flop IEN can be set or cleared by program instructions.
- ✓ When IEN is cleared, the interrupt request coming from IST is neglected by the CPU.
- ✓ The program-controlled IEN bit allows the programmer to choose whether to use the interrupt facility. If an instruction to clear IEN has been inserted in the program, it means that the user does not want his program to be interrupted. An instruction to set IEN indicates that the interrupt facility will be used while the current program is running.
- ✓ Most computers include internal hardware that clears IEN to 0 every time an interrupt is acknowledged by the processor
- ✓ At the end of each instruction cycle the CPU checks IEN and the interrupt signal from IST. If either is equal to 0, control continues with the next instruction.
- ✓ If both IEN and IST are equal to 1, the CPU goes to an interrupt cycle.
- ✓ During the interrupt cycle the CPU performs the following sequence of microoperations:

SP ← SP-1	Decrement stack pointer
M[SP] ←PC	Push PC into stack
INTACK ←1	Enable interrupt acknowledge
PC ← VAD	Transfer vector address to PC
lEN ←0	Disable further interrupts

Go to fetch next instruction

Software Routines

- ✓ A priority interrupt system is a combination of hardware and software techniques
- ✓ The following figure shows the programs that must reside in memory for handling the interrupt system.



Initial and Final Operations

- ✓ Each interrupt service routine must have an initial and final set of operations for controlling the registers in the hardware interrupt system
- ✓ **Initial Sequence**

  [1] Clear lower level Mask reg. bits

  [2] IST ←0

  [3] Save contents of CPU registers

  [4] IEN←1

  [5] Go to Interrupt Service Routine
- ✓ **Final Sequence**

  [1] IEN ←0

  [2] Restore CPU registers

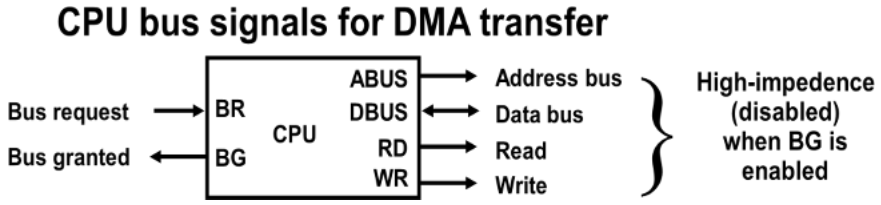
  [3] Clear the bit in the Interrupt Reg

  [4] Set lower level Mask reg. bits

  [5] Restore return address into PC, and IEN ←1
- ✓ The initial and final operations are referred to as **overhead operations** or **housekeeping chores**. They are not part of the service program proper but are essential for processing interrupts.
- ✓ All overhead operations can be implemented by software. This is done by inserting the proper instructions at the beginning and at the end of each service routine. Some of the overhead operations can be done automatically by the hardware

DIRECT MEMORY ACCESS (DMA):

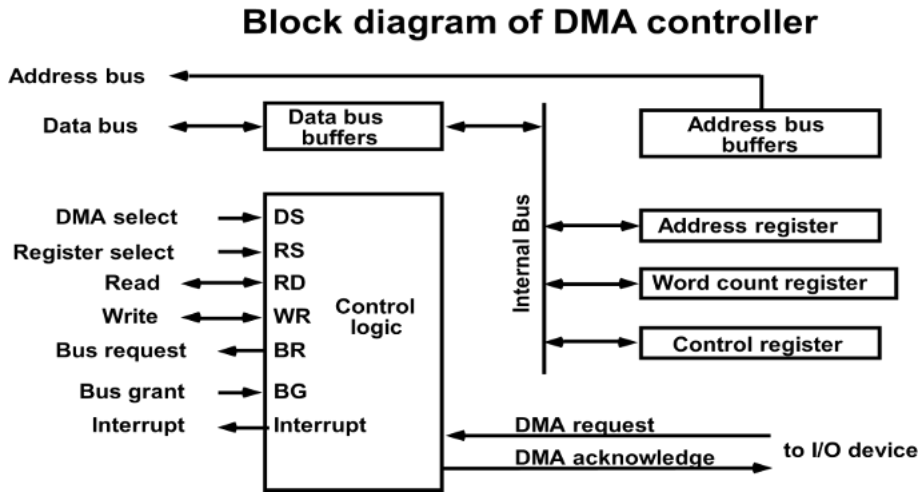
- ✓ The transfer of data between a fast storage device such as magnetic disk and memory is often limited by the speed of the CPU. Removing the CPU from the path and letting the peripheral device manage the memory buses directly would improve the speed of transfer. This transfer technique is called **direct memory access (DMA)**.
- ✓ During DMA transfer, the CPU is idle and has no control of the memory buses.
- ✓ A DMA controller takes over the buses to manage the transfer directly between the I/O device and memory.
- ✓ The CPU may be placed in an idle state in a variety of ways. One common method extensively used in microprocessors is to disable the buses through special control signals.



- ✓ The bus request (BR) input is used by the DMA controller to request the CPU to cease control of the buses. When this input is active, the CPU terminates the execution of the current instruction and places the address bus, the data bus, and the read and write lines into a high-impedance state behaves like an open circuit, which means that the output is disconnected and does not have a logic significance.
- ✓ The CPU activates the Bus grant (BG) output to inform the external DMA that the buses are in the high-impedance state. The DMA that originated the bus request can now take control of the buses to conduct memory transfers without processor intervention. When the DMA terminates the transfer, it disables the bus request line. The CPU disables the bus grant, takes control of the buses, and returns to its normal operation.
- ✓ When the DMA takes control of the bus system, it communicates directly with the memory. The transfer can be made in several ways. In DMA **burst transfer**, a block sequence consisting of a number of memory words is transferred in a continuous burst while the DMA controller is master of the memory buses. This mode of transfer is needed for fast devices such as magnetic disks, where data transmission cannot be stopped or slowed down until an entire block is transferred.
- ✓ An alternative technique called **cycle stealing** allows the DMA controller to transfer one data word at a time after which it must return control of the buses to the CPU. The CPU merely delays its operation for one memory cycle to allow the direct memory I/O transfer to "steal" one memory cycle.

DMA CONTROLLER

- ✓ The following figure shows the block diagram of a typical DMA controller



- ✓ The unit communicates with the CPU via the data bus and control lines. The registers in the DMA are selected by the CPU through the address bus by enabling the DS (DMA select) and RS (register select) inputs. The RD (read) and WR (write) inputs are bidirectional.
- ✓ When the BG (bus grant) input is 0, the CPU can communicate with the DMA registers through the data bus to read from or write to the DMA registers.
- ✓ When BG = 1, the CPU has relinquished(ceased) the buses and the DMA can communicate directly with the memory by specifying an address in the address bus and activating the RD or WR control.
- ✓ The DMA communicates with the external peripheral through the request and acknowledge lines by using a prescribed handshaking procedure.
- ✓ The DMA controller has three registers: an address register, a word count register, and a control register. The address register contains an address to specify the desired location in memory. The address bits go through bus buffers into the address bus. The address register is incremented after each word that is transferred to memory.
- ✓ The word count register is incremented after each word that is transferred to memory. The word count register holds the number of words to be transferred. This register is decremented by one after each word transfer and internally tested for zero.
- ✓ The control register specifies the mode of transfer. All registers in the DMA appear to the CPU as I/O interface registers. Thus the CPU can read from or write into the DMA registers under program control via the data bus.
- ✓ The DMA is first initialized by the CPU. After that, the DMA starts and continues to transfer data between memory and peripheral unit until an entire block is transferred. The initialization process is essentially a program consisting of I/O instructions that include the address for

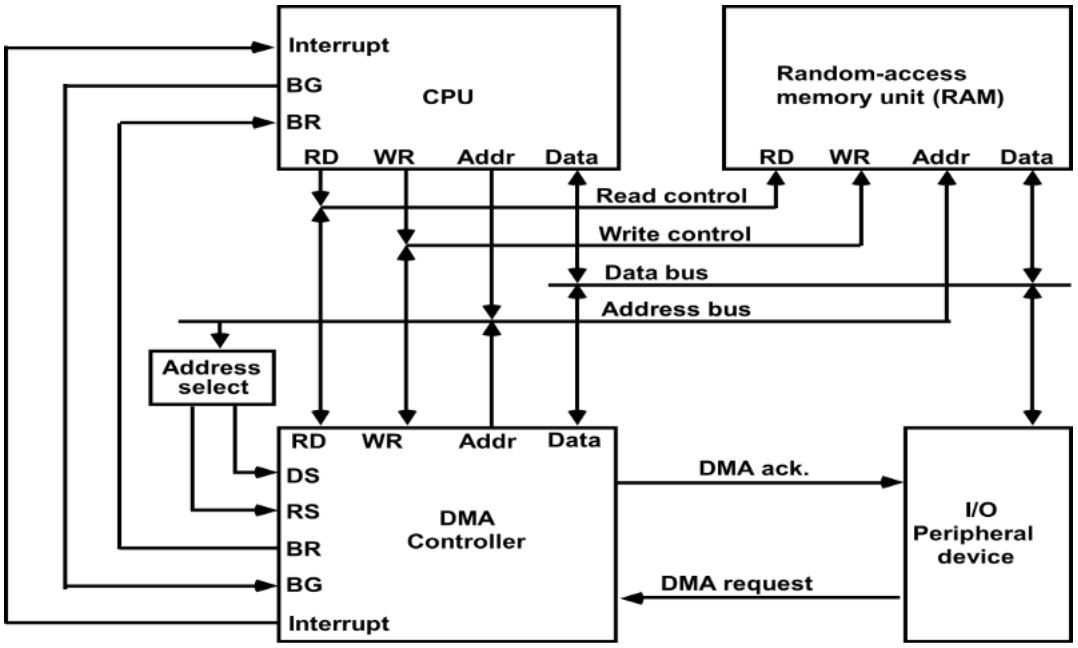
selecting particular DMA registers. The CPU initializes the DMA by sending the following information through the data bus:

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

- ✓ The starting address is stored in the address register. The word count is stored in the word count register, and the control information in the control register.
- ✓ Once the DMA is initialized, the CPU stops communicating with the DMA unless it receives an interrupt signal or if it wants to check how many words have been transferred.

DMA Transfer

- ✓ The position of the DMA controller among the other components in a computer system is illustrated in following fig.



- ✓ The CPU communicates with the DMA through the address and data buses as with any interface unit. The DMA has its own address, which activates the DS and RS lines.
- ✓ The CPU initializes the DMA through the data bus. Once the DMA receives the start control command, it can start the transfer between the peripheral device and the memory.
- ✓ When the peripheral device sends a DMA request, the DMA controller activates the BR line, informing the CPU to relinquish the buses. The CPU responds with its BG line, informing the DMA that its buses are disabled.
- ✓ The DMA then puts the current value of its address register into the address bus, initiates the RD or WR signal, and sends a DMA acknowledge to the peripheral device.

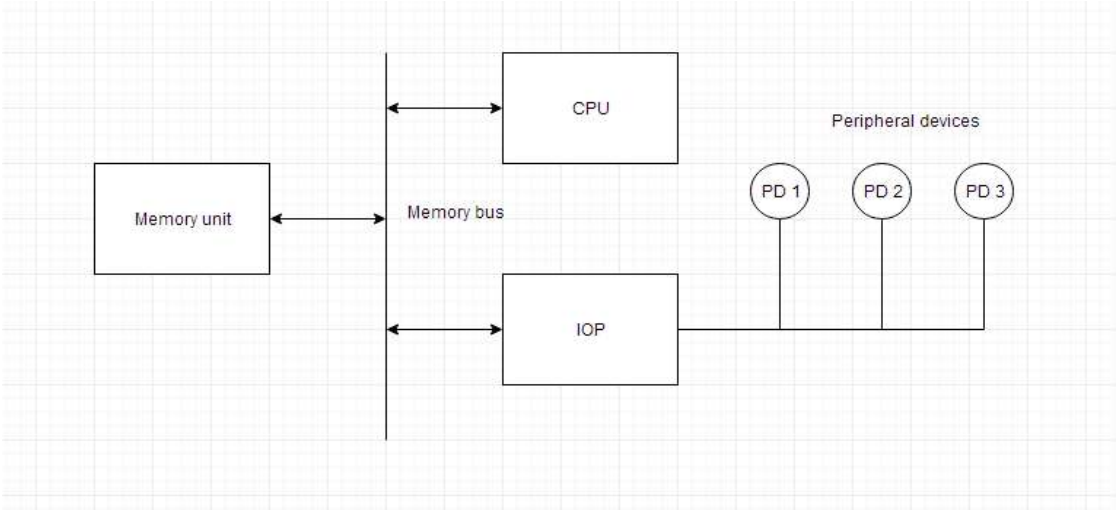
- ✓ Note that the RD and WR lines in the DMA controller are bidirectional. The direction of transfer depends on the status of the BG line. When BG = 0, the RD and WR are input lines allowing the CPU to communicate with the internal DMA registers. When BG = 1, the RD and WR and output lines from the DMA controller to the random-access memory to specify the read or write operation for the data.
- ✓ When the peripheral device receives a DMA acknowledge, it puts a word in the data bus (for write) or receives a word from the data bus (for read). Thus the DMA controls the read or write operations and supplies the address for the memory.
- ✓ The peripheral unit can then communicate with memory through the data bus for direct transfer between the two units while the CPU is momentarily disabled.
- ✓ For each word that is transferred, the DMA increments its address register and decrements its word count register. If the word count does not reach zero, the DMA checks the request line coming from the peripheral.
- ✓ For a high-speed device, the line will be active as soon as the previous transfer is completed. A second transfer is then initiated, and the process continues until the entire block is transferred.
- ✓ If the peripheral speed is slower, the DMA request line may come somewhat later. In this case the DMA disables the bus request line so that the CPU can continue to execute its program. When the peripheral requests a transfer, the DMA requests the buses again.
- ✓ If the word count register reaches zero, the DMA stops any further transfer and removes its bus request. It also informs the CPU of the termination by means of an interrupt.
- ✓ When the CPU responds to the interrupt, it reads the content of the word count register. The zero value of this register indicates that all the words were transferred successfully. The CPU can read this register at any time to check the number of words already transferred.
- ✓ A DMA controller may have more than one channel. In this case, each channel has a request and acknowledges pair of control signals which are connected to separate peripheral devices. Each channel also has its own address register and word count register within the DMA controller.
- ✓ A priority among the channels may be established so that channels with high priority are serviced before channels with lower priority.
- ✓ DMA transfer is very useful in many applications.
- ✓ It is used for fast transfer of information between magnetic disks and memory.
- ✓ It is also useful for updating the display in an interactive terminal.

INPUT-OUTPUT PROCESSOR (IOP)

- ✓ An input-output processor (IOP) is a processor with direct memory access capability. In this, the computer system is divided into a memory unit and number of processors.
- ✓ Each IOP controls and manage the input-output tasks. The IOP is similar to CPU except that it handles only the details of I/O processing. The IOP can fetch and execute its own instructions.
- ✓ These IOP instructions are designed to manage I/O transfers only.

BLOCK DIAGRAM OF I/O PROCESSOR

- ✓ Below is a block diagram of a computer along with various I/O Processors. The memory unit occupies the central position and can communicate with each processor.
- ✓ The CPU processes the data required for solving the computational tasks. The IOP provides a path for transfer of data between peripherals and memory. The CPU assigns the task of initiating the I/O program.
- ✓ The IOP operates independent from CPU and transfer data between peripherals and memory.



- ✓ The communication between the IOP and the devices is similar to the program control method of transfer. And the communication with the memory is similar to the direct memory access method.
- ✓ In large scale computers, each processor is independent of other processors and any processor can initiate the operation.
- ✓ The CPU can act as master and the IOP act as slave processor. The CPU assigns the task of initiating operations but it is the IOP, who executes the instructions, and not the CPU. CPU instructions provide operations to start an I/O transfer. The IOP asks for CPU through interrupt.
- ✓ Instructions that are read from memory by an IOP are also called commands to distinguish them from instructions that are read by CPU. Commands are prepared by programmers and are stored in memory. Command words make the program for IOP. CPU informs the IOP where to find the commands in memory.

- ✓ CPU logic is usually faster than main memory access time, with the result that processing speed is limited primarily by the speed of main memory.
- ✓ A technique used to compensate for the mismatch in operating speeds is to employ in extremely fast, small cache between the CPU and main memory whose access time is close to processor logic clock cycle time.
- ✓ The reason for having two or three levels of memory hierarchy is economics.
- ✓ As the storage capacity of the memory increases, the cost per bit for storing binary information decreases and the access time of the memory becomes longer.
- ✓ The overall goal of using a memory hierarchy is to obtain the highest-possible average access speed while minimizing the total cost of the entire memory system.
- ✓ Auxiliary and cache memories are used for different purposes. The cache holds those parts of the program and data that are most heavily used, while the auxiliary memory holds those parts that are not presently used by the CPU. Moreover, the CPU has direct access to both cache and main memory but not to auxiliary memory. The transfer from auxiliary to main memory is usually done by means of direct memory access of large blocks of data. The typical access time ratio between cache and main memory is about 1 to 7. For example, a typical cache memory may have an access time of 100ns, while main memory access time may be 700ns. Auxiliary memory average access time is usually 1000 times that of main memory. Block size in auxiliary memory typically ranges from256 to 2048 words, while cache block size is typically from 1 to 16 words.
- ✓ Many operating systems are designed to enable the CPU to process a number of independent programs concurrently. This concept, called **multiprogramming**, refers to the existence of two or more programs in different parts of the memory hierarchy at the same time.
- ✓ In a 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 main 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.
- ✓ It is the task of the operating system to maintain in main memory a portion of this information that is currently active.
- ✓ The part of the computer system that supervises the flow of information between auxiliary memory and main memory is called the **memory management system.**

Memory Organization

MAIN MEMORY

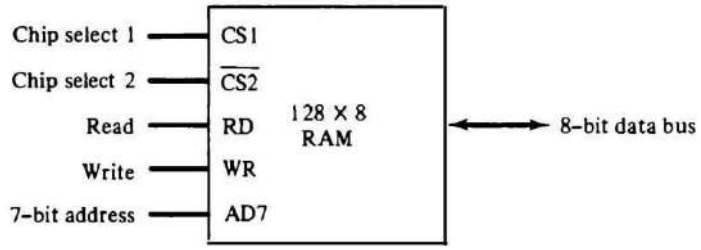
- ✓ The main memory is the central storage unit in a computer system. It is a relatively large and fast memory used to store programs and data during the computer operation.
- ✓ The principal technology used for the main memory is based on semiconductor integrated circuits.
- ✓ Integrated circuit RAM chips are available in two possible operating modes, **static** and **dynamic.** The static RAM consists essentially of internal flip-flops that store the binary information. The stored information remains valid as long as power is applied to unit. The dynamic RAM stores the binary information in the form of electric charges that are applied to capacitors. The capacitors are provided inside the chip by MOS transistors. The stored charge on the capacitors tends to discharge with time and the capacitors must be periodically recharged by refreshing the dynamic memory.
- ✓ The dynamic RAM offers reduced power consumption and larger storage capacity in a single memory chip.
- ✓ The static RAM is easier to use and has shorted read and write cycles.
- ✓ Most of the main memory in a general-purpose computer is made up of RAM integrated circuit chips, but a portion of the memory may be constructed with ROM chips.
- ✓ RAM refers to a random-access memory, but it is used to designate a read/write memory to distinguish it from a read-only memory, although ROM is also random access.
- ✓ RAM is used for storing the bulk of the programs and data that are subject to change. ROM is used for storing programs that are permanently resident in the computer
- ✓ The ROM portion of main memory is needed for storing an initial program called a **bootstrap loader**. The bootstrap loader is a program whose function is to start the computer software operating when power is turned on.
- ✓ Since RAM is volatile, its contents are destroyed when power is turned off. The contents of ROM remain unchanged after power is turned off and on again.
- ✓ The startup of a computer consists of turning the power on and starting the execution of an initial program. Thus when power is turned on, the hardware of the computer sets the program counter to the first address of the bootstrap loader. The bootstrap program loads a portion of the operating system from disk to main memory and control is then transferred to the operating system, which prepares the computer for general use.
- ✓ RAM and ROM chips are available in a variety of sizes. If the memory needed for the computer is larger than the capacity of one chip, it is necessary to combine a number of chips to form the

required memory size. Ex: 1024 × 8 memory can be constructed with 128 × 8 RAM chips and 512 × 8 ROM chips.

RAM AND ROM CHIPS

- ✓ A RAM chip is better suited for communication with the CPU if it has one or more control inputs that select the chip only when needed. Another common feature is a bidirectional data bus that allows the transfer of data either from memory to CPU during a read operation or from CPU to memory during a write operation.
- ✓ A bidirectional bus can be constructed with three-state buffers.
- ✓ The block diagram of a RAM chip is shown in Fig.

Typical RAM chip.



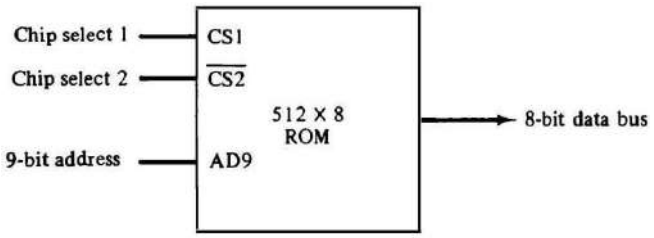
(a) Block diagram

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 eight 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 chips select (CS) control inputs are for enabling the chip only when it is selected by the microprocessor. The availability of more than one control input to select the chip facilitates the decoding of the address lines when multiple chips are used in the microcomputer.
- ✓ The read and write inputs are sometimes combined into one line labeled R/W. When the chip is selected, the two binary states in this line specify the two operations or read or write.
- ✓ The unit is in operation only when CS1 = 1 and $\overline{CS2}$ = 0.
- ✓ If the chip select inputs are not enabled, or if they are enabled but the read but the read or write inputs are not enabled, the memory is inhibited and its data bus is in a high-impedance state.

- ✓ When CS1 = 1 and $\overline{CS2}$ = 0, the memory can be placed in a write or read mode. When the WR input is enabled, the memory stores a byte from the data bus into a location specified by the address input lines.
- ✓ When the RD input is enabled, the content of the selected byte is placed into the data bus. The RD and WR signals control the memory operation as well as the bus buffers associated with the bidirectional data bus.
- ✓ A ROM chip is organized externally in a similar manner. ROM can only read, the data bus can only be in an output mode. The block diagram of a ROM chip is shown in Fig.



Typical ROM chip.

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

MEMORY ADDRESS MAP

- ✓ The designer of a computer system must calculate the amount of memory required for the particular application and assign it to either RAM or ROM. The interconnection between memory and processor is then established form knowledge of the size of memory needed and the type of RAM and ROM chips available.
- ✓ A memory address map, is a pictorial representation of assigned address space for each chip in the system.
- ✓ To demonstrate with a particular example, assume that a computer system needs 512 bytes of RAM and 512 bytes of ROM.
- ✓ The memory address map for this configuration is shown in Table.

Memory Address Map for Microprocomputer

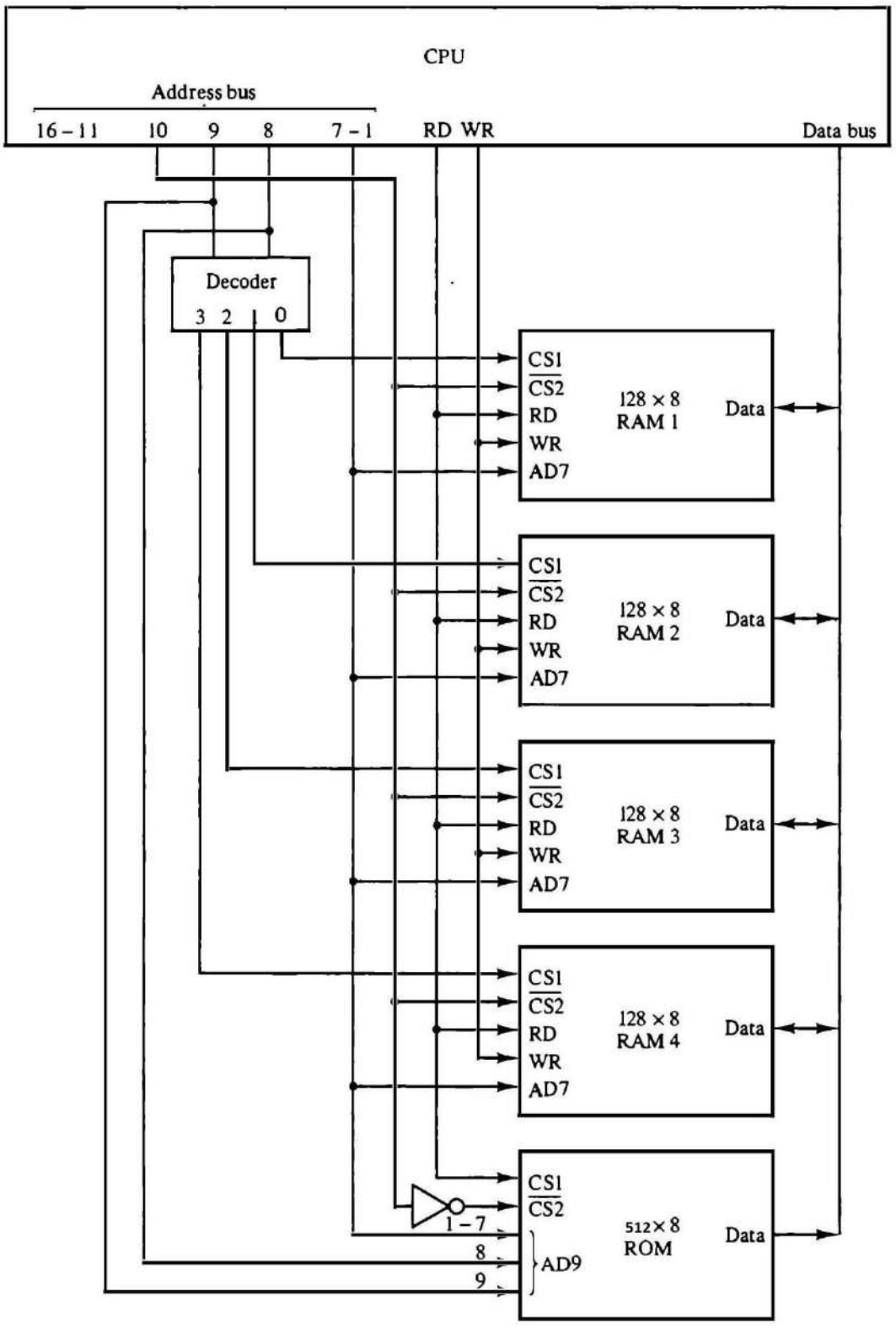
Component	Hexadecimal address	Address bus 10	9	8	7	6	5	4	3	2	1
RAM 1	0000–007F	0	0	0	x	x	x	x	x	x	x
RAM 2	0080–00FF	0	0	1	x	x	x	x	x	x	x
RAM 3	0100–017F	0	1	0	x	x	x	x	x	x	x
RAM 4	0180–01FF	0	1	1	x	x	x	x	x	x	x
ROM	0200–03FF	1	x	x	x	x	x	x	x	x	x

- ✓ The small x's under the address bus lines designate those lines that must be connected to the address inputs in each chip.
- ✓ The RAM chips have 128 bytes and need seven address lines. The ROM chip has 512 bytes and needs 9 address lines.
- ✓ It is now necessary to distinguish between four RAM chips by assigning to each a different address. For this particular example we choose bus lines 8 and 9 to represent four distinct binary combinations.
- ✓ The distinction between a RAM and ROM address is done with another bus line. Here we choose line 10 for this purpose. When line 10 is 0, the CPU selects a RAM, and when this line is equal to 1, it selects the ROM.
- ✓ The first hexadecimal digit represents lines 13 to 16 and is always 0. The next hexadecimal digit represents lines 9 to 12, but lines 11 and 12 are always 0. The range of hexadecimal addresses for each component is determined from the x's associated with it. These x's represent a binary number that can range from an all-0's to an all-1's value.

MEMORY CONNECTION TO CPU

- ✓ RAM and ROM chips are connected to a CPU through the data and address buses.
- ✓ The low-order lines in the address bus select the byte within the chips and other lines in the address bus select a particular chip through its chip select inputs.
- ✓ The connection of memory chips to the CPU is shown in Fig. This configuration gives a memory capacity of 512 bytes of RAM and 512 bytes of ROM.
- ✓ Each RAM receives the seven low-order bits of the address bus to select one of 128 possible bytes. The particular RAM chip selected is determined from lines 8 and 9 in the address bus. This is done through a 2 × 4 decoder whose outputs go to the CS1 input in each RAM chip. Thus, when address lines 8 and 9 are equal to 00, the first RAM chip is selected. When 01, the second RAM chip is selected, and so on.
- ✓ The RD and WR outputs from the microprocessor are applied to the inputs of each RAM chip.

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



Memory connection to the CPU.

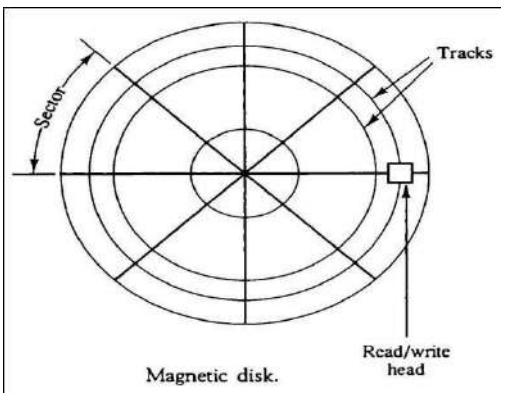
Memory Organization

AUXILIARY MEMORY:

- ✓ The most common auxiliary memory devices used in computer systems are magnetic disks and magnetic tapes. Other components used, but not as frequently, are magnetic drums, magnetic bubble memory, and optical disks.
- ✓ The important characteristics of any device are its access mode, access time, transfer rate, capacity, and cost.
- ✓ The average time required to reach a storage location in memory and obtain its contents is called the access time. The access time consists of a seek time required to position the read-write head to a location and a transfer time required to transfer data to or from the device.
- ✓ Auxiliary storage is organized in records or blocks. A record is a specified number of characters or words. Reading or writing is always done on entire records. The transfer rate is the number of characters or words that the device can transfer per second, after it has been positioned at the beginning of the record.
- ✓ Magnetic drums and disks are quite similar in operation. Both consist of high-speed rotating surfaces coated with a magnetic recording medium. The rotating surface of the drum is a cylinder and that of the disk, a round flat plate. Bits are recorded as magnetic spots on the surface as it passes a stationary mechanism called a write head. Stored bits are detected by a change in magnetic field produced by a recorded spot on the surface as it passes through a read head.

MAGNETIC DISKS

- ✓ A magnetic disk is a circular plate constructed of metal or plastic coated with magnetized material. Often both sides of the disk are used and several disks may be stacked on one spindle with read/write heads available on each surface.
- ✓ All disks rotate together at high speed and are not stopped or started from access purposes.
- ✓ Bits are stored in the magnetized surface in spots along concentric circles called tracks. The tracks are commonly divided into sections called sectors. In most systems, the minimum quantity of information which can be transferred is a sector.
- ✓ Some units use a single read/write head from each disk surface. The track address bits are used by a mechanical assembly to move the head into the specified track position before reading or writing.



Magnetic disk.

- ✓ In other disk systems, separate read/write heads are provided for each track in each surface. The address can then select a particular track electronically through a decoder circuit. This type of unit is more expensive and is found only in very large computer systems.
- ✓ A disk system is addressed by address bits that specify the disk number, the disk surface, the sector number and the track within the sector.
- ✓ After the read/write heads are positioned in the specified track, the system has to wait until the rotating disk reaches the specified sector under the read/write head.
- ✓ Information transfer is very fast once the beginning of a sector has been reached.
- ✓ Disks may have multiple heads and simultaneous transfer of bits from several tracks at the same time.
- ✓ A track in a given sector near the circumference is longer than a track near the center of the disk. If bits are recorded with equal density, some tracks will contain more recorded bits than others. To make all the records in a sector of equal length, some disks use a variable recording density with higher density on tracks near the center than on tracks near the circumference. This equalizes the number of bits on all tracks of a given sector.
- ✓ Disks that are permanently attached to the unit assembly and cannot be removed by the occasional user are called hard disks. A disk drive with removable disks is called a floppy disk.
- ✓ The disks used with a floppy disk drive are small removable disks made of plastic coated with magnetic recording material. There are two sizes commonly used, with diameters of 5.25 and 3.5 inches. The 3.5-inch disks are smaller and can store more data than can the 5.25-inch disks.

MAGNETIC TAPE

- ✓ The Magnetic tape itself is a strip of plastic coated with a magnetic recording medium. Bits are recorded as magnetic spots on the tape along several tracks. Usually, seven or nine bits are recorded simultaneously to form a character together with a parity bit.
- ✓ Read/write heads are mounted one in each track so that data can be recorded and read as a sequence of characters.
- ✓ Magnetic tape units can be stopped, started to move forward or in reverse, or can be rewound.
- ✓ Gaps of unrecorded tape are inserted between records where the tape can be stopped. The tape starts moving while in a gap and attains its constant speed by the time it reaches the next record.
- ✓ Each record on tape has an identification bit pattern at the beginning and end. By reading the bit pattern at the beginning, the tape control identifies the record number. By reading the bit pattern at the end of the record, the control recognizes the beginning of a gap. A tape unit is addressed by specifying the record number of characters in the record. Records may be of fixed or variable length.

Memory Organization

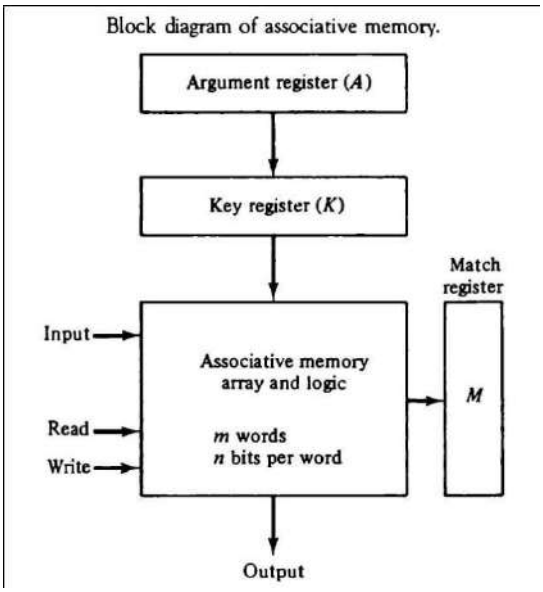
ASSOCIATIVE MEMORY

- ✓ Many data-processing applications require the search of items in a table stored in memory. An assembler program searches the symbol address table in order to extract the symbol's binary equivalent.
- ✓ The number of accesses to memory depends on the location of the item and the efficiency of the search algorithm. Many search algorithms have been developed to minimize the number of accesses while searching for an item in a random or sequential access memory.
- ✓ The time required to find an item stored in memory can be reduced considerably if stored data can be identified for access by the content of the data itself rather than by an address.
- ✓ A memory unit accessed by content is called an ***associative memory or content addressable memory (CAM).***
- ✓ When a word is to be read from an associative memory, the content of the word, or part of the word, is specified. The memory locates all words which match the specified content and marks them for reading.
- ✓ An associative memory is more expensive than a random access memory because each cell must have storage capability as well as logic circuits for matching its content with an external argument. For this reason, associative memories are used in applications where the search time is very critical and must be very short.

HARDWARE ORGANIZATION

- ✓ The block diagram of an associative memory is shown in Fig.
- ✓ It consists of a memory array and logic for *m* words with *n* bits per word. The argument register A and key register K each have n bits, one for each bit of a word. The match register M has m bits, one for each memory word.
- ✓ Each word in memory is compared in parallel with the content of the argument register. The words that match the bits of the argument register set a corresponding bit in the match register.

Block diagram of associative memory.



- ✓ After the matching process, those bits in the match register that have been set indicate the fact that their corresponding words have been matched.
- ✓ Reading is accomplished by a sequential access to memory for those words whose corresponding bits in the match register have been set.

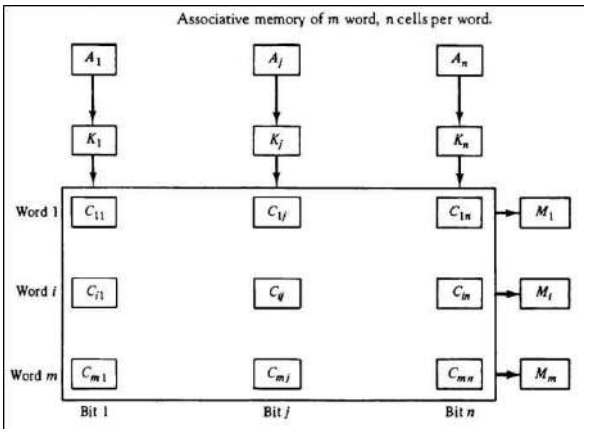
- ✓ The key register provides a mask for choosing a particular field or key in the argument word. The entire argument is compared with each memory word if the key register contains all 1's. Otherwise, only those bits in the argument that have 1's in their corresponding position of the key register are compared.
- ✓ To illustrate with a numerical example, suppose that the argument register A and the key register K have the bit configuration shown below. Only the three leftmost bits of A are compared with memory words because K has 1's in these positions.

  A 101 111100

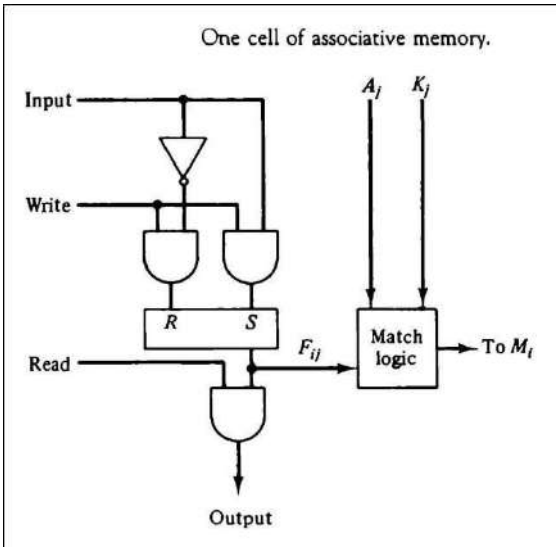
  K 111 000000

  Word 1 100 111100 no match

  Word 2 101 000001 match
- ✓ The relation between the memory array and external registers in an associative memory is shown in Fig.
- ✓ 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 Cij is the cell for bit j in word i.
- ✓ A bit Aj in the argument register is compared with all the bits in column j of the array provided that Kj = 1. This is done for all columns j = 1, 2,…,n.



- ✓ If a match occurs between all the unmasked bits of the argument and the bits in word i, the corresponding bit Mi in the match register is set to 1. If one or more unmasked bits of the argument and the word do not match, Mi is cleared to 0
- ✓ The internal organization of a typical cell Cij is shown in Fig.
- ✓ It consists of a flipflop storage element Fij and the circuits for reading, writing, and matching the cell.
- ✓ The input bit is transferred into the storage cell during a write operation. The bit stored is read out during a read operation.
- ✓ The match logic compares the content of the storage cell with the corresponding unmasked bit of the argument and provides an output for the decision logic that sets the bit in Mi.



Memory Organization

MATCH LOGIC

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

  xj= Aj Fij + A'jF'ij

  where xj = 1 if the pair of bits in position j are equal; otherwise, xj = 0.
- ✓ For a word i to be equal to the argument in A we must have all xj variables equal to 1.
- ✓ This is the condition for setting the corresponding match bit Mi to 1. The Boolean function for this condition is

  Mi= x1 x2 x3 … xn
- ✓ Include the key bit Kj in the comparison logic. The requirement is that if Kj = 0,the corresponding bits of Aj and Fij need no comparison. Only when Kj = 1 must they be compared. This requirement is achieved by ORing each term with K'j , thus:

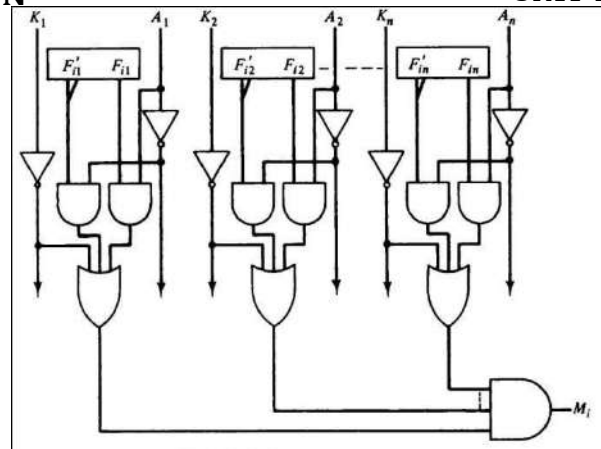
  xj + K'j = xj if Kj=1

  1 if Kj=0
- ✓ When Kj = 1, we have Kj' = 0 and xj + 0 = xj. When Kj = 0, then Kj' = 1 xj + 1 = 1. A term (xj +Kj') will be in the 1 state if its pair of bitsis not compared. This is necessary because each term is ANDed with all other terms so that an output of 1 will have no effect. The comparison of the bits has an effect only when Kj = 1.
- ✓ The match logic for word i in an associative memory can now be expressed by the following Boolean function:

  Mi = (x1 + K '1) (x2 + K '2) (x3 + K '3) …. (xn + K 'n)
- ✓ Each term in the expression will be equal to 1 if its corresponding Kj = 0. If Kj = 1, the term will be either 0 or 1 depending on the value of xj. A match will occur and Mi will be equal to 1 if all terms are equal to 1.
- ✓ If we substitute the original definition of xj. the Boolean function above can be expressed as follows:

  $$\mathbf{Mi} = \prod_{j=1}^{n} (\mathbf{Aj\ Fij} + \mathbf{A'jF'ij} + \mathbf{K'j})$$
- ✓ The circuit for matching one word is shown in Fig. Each cell requires two AND gates and one OR gate. The inverters for Aj and Kj are needed once for each column and are used for all bits in the column. The output of all OR gates in the cells of the same word go to the input of a common AND gate to generate the match signal for Mi. Mi will be logic 1 if a match occurs and 0 if no match occurs. Note that if the key register contains all 0's, output Mi will be a 1

irrespective of the value of A or the word. This occurrence must be avoided during normal operation.



Match logic for one word of associative memory.

READ OPERATION

- ✓ If more than one word in memory matches the unmasked argument field, all the matched words will have 1's in the corresponding bit position of the 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 Mi bit is a 1.
- ✓ In most applications, the associative memory stores a table with no two identical items under a given key. In this case, only one word may match the unmasked argument field. By connecting output Mi directly to the read line in the same word position (instead of the M register), the content of the matched word will be presented automatically at the output lines and no specialread command signal is needed.

WRITE OPERATION

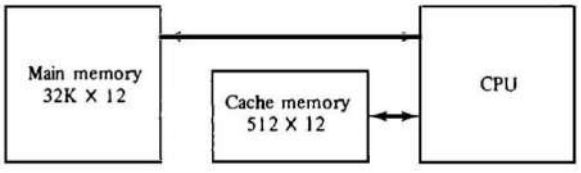
- ✓ An associative memory must have a write capability for storing the information to be searched.
- ✓ Writing in an associative memory can take different forms, depending on the application. If the entire memory is loaded with new information at once prior to a search operation then the writing can be done by addressing each location in sequence. This will make the device a random-access memory for writing and a content addressable memory for reading. The advantage here is that the address for input can be decoded as in a random-access memory. Thus instead of having m address lines, one for each word in memory, the number of address lines can be reduced by the decoder to d lines, where $m = 2^d$.
- ✓ If unwanted words have to be deleted and new words inserted one at a time, there is a need for a special register to distinguish between active and inactive words. This register, sometimes called a *tag register*.
- ✓ For every active word stored in memory, the corresponding bit in the tag register is set to 1. A word is deleted from memory by clearing its tag bit to 0.
- ✓ Words are stored in memory by scanning the tag register until the first 0 bit is encountered. This gives the first available inactive word and a position for writing a new word. After the new word is stored in memory it is made active by setting its tag bit to 1. An unwanted word when deleted from memory can be cleared to all 0's if this value is used to specify an empty location.

CACHE MEMORY

- ✓ Locality of Reference: The references to memory at any given time interval tends to be confined within a localized area.
- ✓ When a program loop is executed, the CPU repeatedly refers to the set of instructions in memory that constitute the loop.
- ✓ Every time a given subroutine is called, its set of instructions is fetched from memory. Thus loops and subroutines tend to localize the references to memory for fetching instructions.
- ✓ Iterative procedures refer to common memory locations and array of numbers are confined within a local portion of memory
- ✓ If the active portions of the program and data are placed in a fast small memory, the average memory access time can be reduced, thus reducing the total execution time of the program. Such a fast small memory is referred to as a cache memory. The cache is the fastest component in the memory hierarchy and approaches the speed of CPU components.
- ✓ When the CPU needs to access memory, the cache is examined. If the word is found in the cache, it is read from the fast memory. If the word addressed by the CPU is not found in the cache, the main memory is accessed to read the word. A block of words containing the one just accessed is then transferred from main memory to cache memory
- ✓ The performance of cache memory is frequently measured in terms of a quantity called **hit ratio**. When the CPU refers to memory and finds the word in cache, it is said to produce a hit. If the word is not found in cache, it is in main memory and it counts as a miss. *The ratio of the number of hits divided by the total CPU references to memory (hits plus misses) is the hit ratio.*
- ✓ The average memory access time of a computer system can be improved considerably by use of a cache.
- ✓ The transformation of data from main memory to cache memory is referred to as a mapping process. Three types of mapping procedures are :

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

- ✓ Consider the following memory organization:



Example of cache memory.

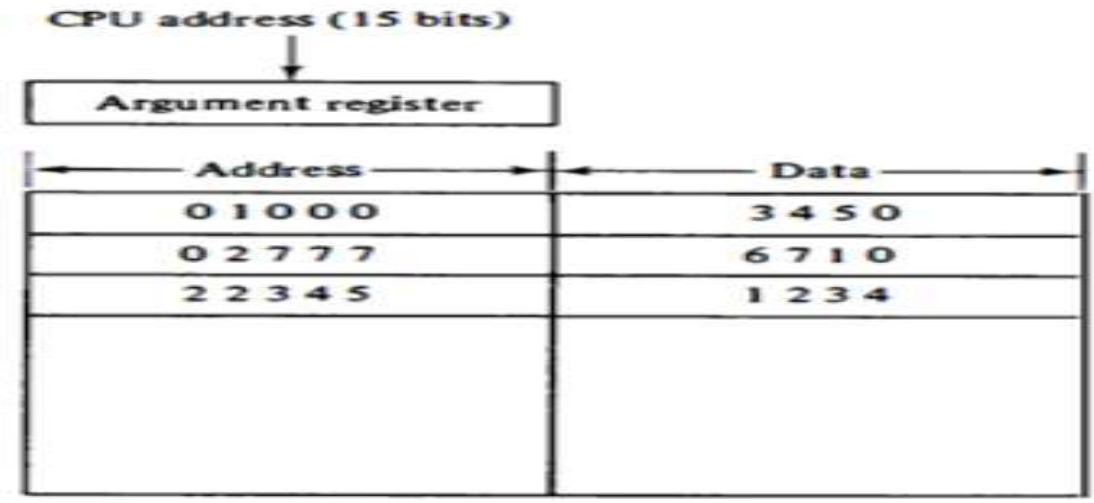
Memory Organization

Cache Memory If the active portions of the program and data are placed in a fast small memory, the average memory access time can be reduced, thus reducing the total execution time of the program. Such a fast small memory is referred to as a cache memory. It is placed between the CPU and main memory. The cache memory access time is less than the access time of main memory by a factor of 5 to 10. The cache is the fastest component in the memory hierarchy and approaches the speed of CPU components. The fundamental idea of cache organization is that by keeping the most frequently accessed instructions and data in the fast cache memory, the average memory access time will approach the access time of the cache.

Although the cache is only a small fraction of the size of main memory, a large fraction of memory requests will be found in the fast cache memory because of the locality of reference property of programs. The basic operation of the cache is as follows. When the CPU needs to access memory, the cache is examined. If the word is found in the cache, it is read from the fast memory. If the word addressed by the CPU is not found in the cache, the main memory is accessed to read the word. A block of words containing the one just accessed is then transferred from main memory to cache memory. The block size may vary from one word (the one just accessed) to about 16 words adjacent to the one just accessed. In this manner, some data are transferred to cache so that future references to memory find the required words in the fast cache memory. The performance of cache memory is frequently measured in terms of a quantity called hit ratio. When the CPU refers to memory and finds the word in cache, it is said to produce a hit. If the word is not found in cache, it is in main memory and it counts as a miss. The ratio of the number of hits divided by the total CPU references to memory (hits plus misses) is the hit ratio. The hit ratio is best measured experimentally by running representative programs in the computer and measuring the number of hits and misses during a given interval of time. Hit ratios of 0.9 and higher have been reported.

This high ratio verifies the validity of the locality of reference property. The basic characteristic of cache memory is its fast access time. Therefore, very little or no time must be wasted when searching for words in the cache. The transformation of data from main memory to cache memory is referred to as a mapping process. **Three types of mapping procedures are of practical interest when considering the organization of cache memory: 1. Associative mapping 2. Direct mapping 3. Set-associative mapping Associative Mapping.**

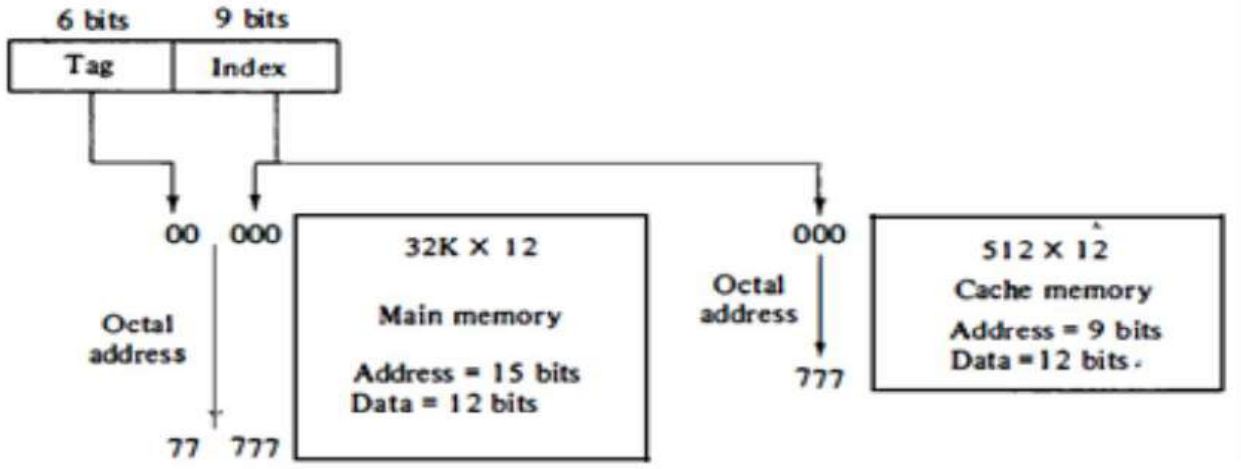
The fastest and most flexible cache organization uses an associative memory. This organization is illustrated in Figure below. The **associative memory** stores both the address and content (data) of the memory word. This permits any location in cache to store any word from main memory. The diagram shows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



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

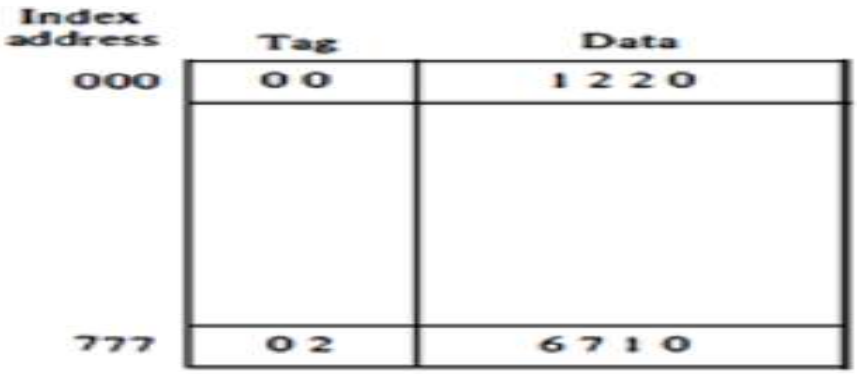
Direct Mapping Associative memories are expensive compared to random-access memories because of the added logic associated with each cell. The possibility of using a random-access memory for the cache is investigated in Figure below. The CPU address of 15 bits is divided into two fields. The nine least significant bits constitute the index field and the remaining six bits from the tag field. The figure shows that main memory needs an address that includes both the

tag and the index bits. The number of bits in the index field is equal to the number of address bits required to access the cache memory.



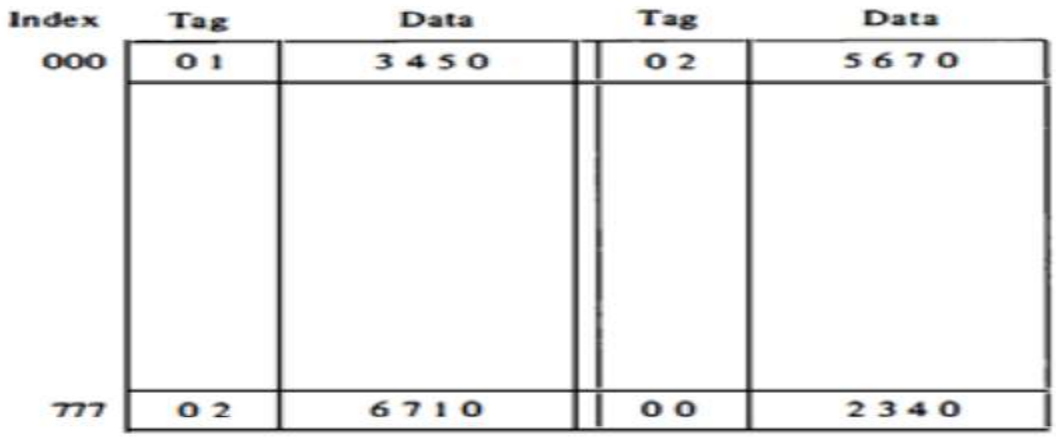
In the general case, there are 2' words in cache memory and 2" words in main memory. The n-bit memory address is divided into two fields: k bits for the index field and n - k bits for the tag field. The direct mapping cache organization uses the n-bit address to access the main memory and the k-bit index to access the cache. The internal organization of the words in the cache memory is as shown in Figure below. Each word in cache consists of the data word and its associated tag. When a new word is first brought into the cache, the tag bits are stored alongside the data bits. When the CPU generates a memory request, the index field is used for the address to access the cache. The tag field of the CPU address is compared with the tag in the word read from the cache. If the two tags match, there is a hit and the desired data word is in cache.

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



The word at address zero is presently stored in the cache (index = 000, tag = 00, data = 1220). Suppose that the CPU now wants to access the word at address 02000. The index address is 000, so it is 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 02 and data of 5670.

Set-Associative Mapping It was mentioned previously that the disadvantage of direct mapping is that two words with the same index in their address but with different tag values cannot reside in cache memory at the same time. A third type of cache organization, called set-associative mapping, is an improvement over the direct mapping organization in that each word of cache can store two or more words of memory under the same index address. Each data word is stored together with its tag and the number of tag-data items in one word of cache is said to form a set. An example of a set-associative cache organization for a set size of two is shown in Figure below. Each index address refers to two data words and their associated tags.



Index	Tag	Data	Tag	Data
000	01	3450	02	5670
777	02	6710	00	2340

The octal numbers listed in Figure above are with reference to the main memory contents already illustrated. The words stored at addresses 01000 and 02000 of main memory are stored in cache memory at index address 000. Similarly, the words at addresses 02777 and 00777 are stored in cache at index address 777. When the CPU generates a memory request, the index value of the address is used to access the cache. The tag field of the CPU address is then compared with both tags in the cache to determine if a 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. However, an increase in the set size increases the number of bits in words of cache and requires more complex comparison logic.

UNIT- V
PIPELINE AND MULTIPROCESSORS

RISC Processor

RISC stands for **Reduced Instruction Set Computer Processor**, microprocessor architecture with a simple collection and highly customized set of instructions. It is built to minimize the instruction execution time by optimizing and limiting the number of instructions. It means each instruction cycle requires only one clock cycle, and each cycle contains three parameters: fetch, decode and execute.

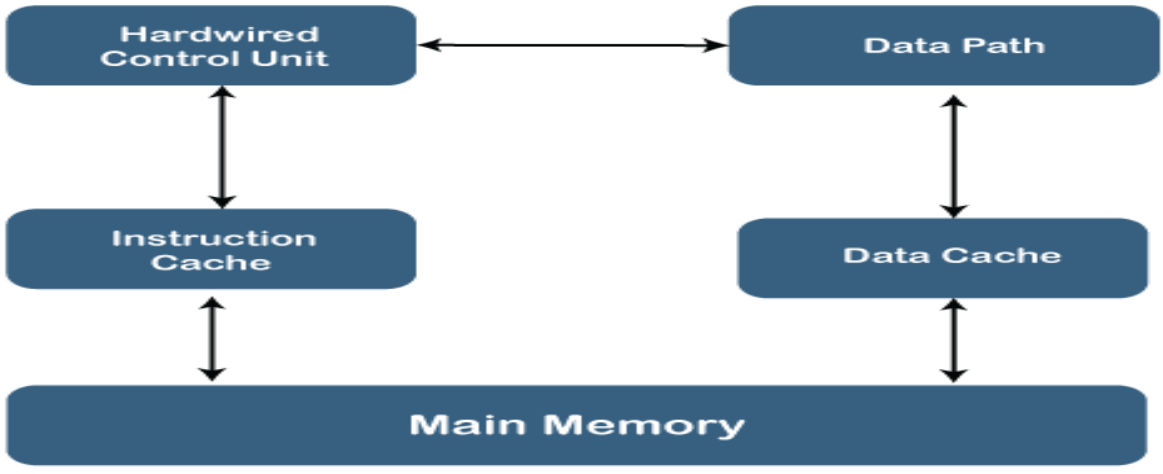
The RISC processor is also used to perform various complex instructions by combining them into simpler ones. RISC chips require several transistors, making it cheaper to design and reduce the execution time for instruction.

Advantages of RISC Processor

1. The RISC processor's performance is better due to the simple and limited number of the instruction set.
2. RISC processor is simpler than a CISC processor because of its simple and quick design, and it can complete its work in one clock cycle.

RISC Architecture

It is a highly customized set of instructions used in portable devices due to system reliability such as Apple iPod, mobiles/smart phones, Nintendo DS,



RISC Architecture

Characteristics of RISC Processor

Some important features of RISC processors are:

1. **One cycle execution time:** For executing each instruction in a computer, the RISC processors require one CPI (Clock per cycle). And each CPI includes the fetch, decode and execute method applied in computer instruction.
2. **Pipelining technique:** The pipelining technique is used in the RISC processors to execute multiple parts or stages of instructions to perform more efficiently.
3. **A large number of registers:** RISC processors are optimized with multiple registers that can be used to store instruction and quickly respond to the computer and minimize interaction with computer memory.
4. It supports a simple addressing mode and fixed length of instruction for executing the pipeline.
5. It uses LOAD and STORE instruction to access the memory location.
6. Simple and limited instruction reduces the execution time of a process in a RISC.

CISC Processor

The CISC Stands for **Complex Instruction Set Computer**, developed by the Intel. It has a large collection of complex instructions that range from simple to very complex and specialized in the assembly language level, which takes a long time to execute the instructions. So, CISC approaches reducing the number of instruction on each program and ignoring the number of cycles per instruction. It emphasizes to build complex instructions directly in the hardware because the hardware is always faster than software. However, CISC chips are relatively slower as compared to RISC chips but use little instruction than RISC. Examples of CISC processors are VAX, AMD, Intel x86 and the System/360.

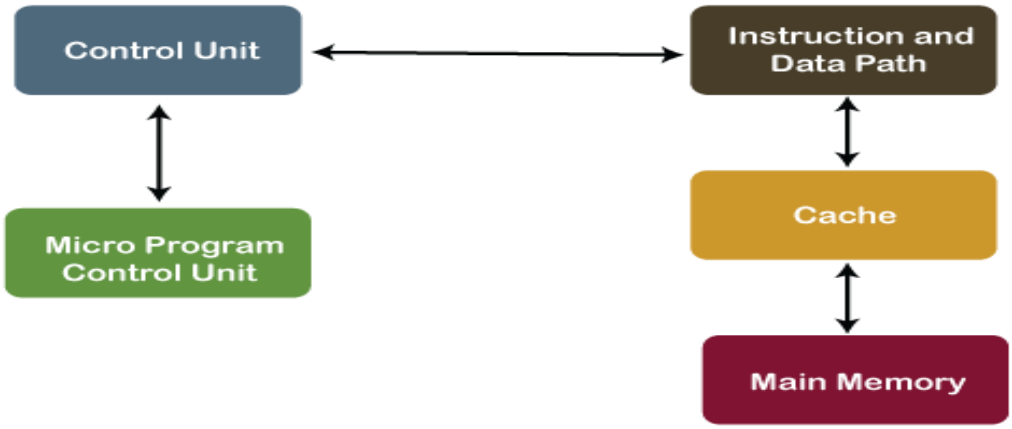
Characteristics of CISC Processor

Following are the main characteristics of the RISC processor:

1. The length of the code is shorts, so it requires very little RAM.
2. CISC or complex instructions may take longer than a single clock cycle to execute the code.
3. Less instruction is needed to write an application.
4. It provides easier programming in assembly language.
5. Support for complex data structure and easy compilation of high-level languages.
6. It is composed of fewer registers and more addressing nodes, typically 5 to 20.
7. Instructions can be larger than a single word.
8. It emphasizes the building of instruction on hardware because it is faster to create than the software.

CISC Processors Architecture

The CISC architecture helps reduce program code by embedding multiple operations on each program instruction, which makes the CISC processor more complex. The CISC architecture-based computer is designed to decrease memory costs because large programs or instruction required large memory space to store the data, thus increasing the memory requirement, and a large collection of memory increases the memory cost, which makes them more expensive.



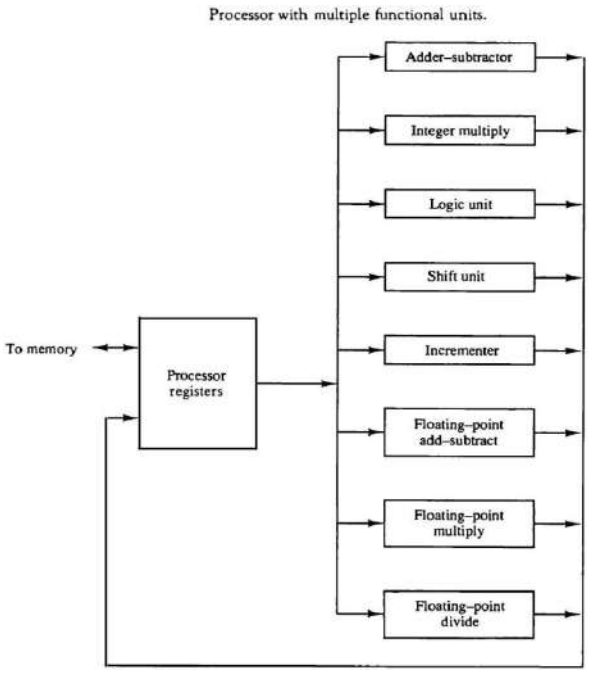
CISC Architecture

COMPARISON OF RISC AND CISC IN COMPUTER ARCHITECTURE?

S.NO.	RISC	CISC
1	RISC is a Reduced Instruction Set Computer	CISC is a Complex Instruction Set Computer
2	There are few addressing modes	There are many addressing modes.
3	There are few instructions.	There are many instructions
4	It can include simple instructions and takes one cycle.	It can include complex instructions and takes multiple cycles
5	Hardware executes the instructions.	Micro-program executes the instructions
6	There are Fixed format instructions.	There are Variable format instructions.
7	It can be easier to decode as instructions have a fixed format.	It can be complex to decode as instructions have variable format
8	RISC is highly pipelined.	CISC is not pipelined or less pipelined.
9	The RISC consumes low power.	The CISC consumes high power.
10	It uses more number of registers	It uses less number of registers

PARALLEL PROCESSING

- ✓ A parallel processing system is able to perform concurrent data processing to achieve faster execution time
- ✓ The system may have two or more ALUs and be able to execute two or more instructions at the same time
- ✓ Also, the system may have two or more processors operating concurrently
- ✓ Goal is to increase the *throughput* – the amount of processing that can be accomplished during a given interval of time
- ✓ Parallel processing increases the amount of hardware required
- ✓ Example: the ALU can be separated into three units and the operands diverted to each unit under the supervision of a control unit
- ✓ All units are independent of each other
- ✓ A multifunctional organization is usually associated with a complex control unit to coordinate all the activities among the various components
- ✓ Parallel processing can be classified from:
  - ➢ The internal organization of the processors
  - ➢ The interconnection structure between processors
  - ➢ The flow of information through the system
  - ➢ The number of instructions and data items that are manipulated simultaneously
- ✓ The sequence of instructions read from memory is the *instruction stream*



- ✓ The operations performed on the data in the processor is the *data stream*
- ✓ Parallel processing may occur in the instruction stream, the data stream, or both
- ✓ Flynn's Computer classification:
  - ➢ 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 – Instructions are executed sequentially. Parallel processing may be achieved by means of multiple functional units or by pipeline processing
- ✓ SIMD – Includes multiple processing units with a single control unit. All processors receive the same instruction, but operate on different data.
- ✓ MIMD – A computer system capable of processing several programs at the same time.

PIPELINING

- ✓ Pipelining is a technique of decomposing a sequential process into sub operations, with each sub process being executed in a special dedicated segment that operates concurrently with all other segments
- ✓ 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
- ✓ Each segment consists of an input register followed by an combinational circuit
- ✓ A clock is applied to all registers after enough time has elapsed to perform all segment activity
- ✓ The information flows through the pipeline one step at a time
- ✓ Example: Ai * Bi + Ci for i = 1, 2, 3, …, 7
- ✓ The sub operations performed in each segment are:

R1 ← Ai , R2 ← Bi
R3 ← R1 * R2, R4 ← Ci
R5 ← R3 + R4

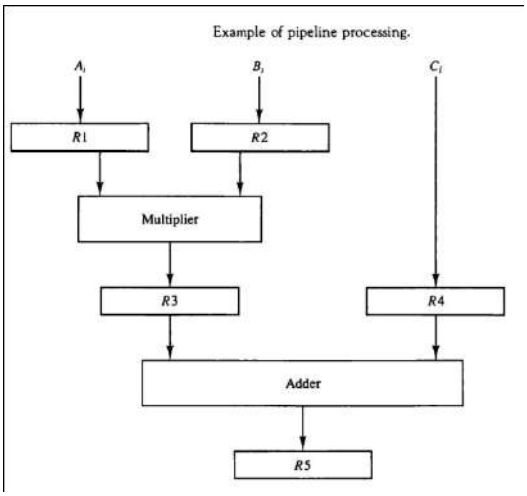
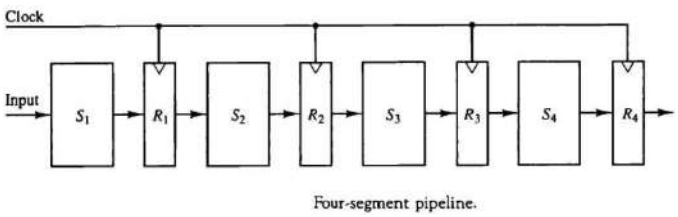


TABLE 9-1 Content of Registers in Pipeline Example

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	—	—	—	—	$A_7 * B_7 + C_7$

- ✓ Any operation that can be decomposed into a sequence of sub-operations of about the samecomplexity can be implemented by a pipeline processor
- ✓ The technique is efficient for those applications that need to repeat the same task many time with different sets of data
- ✓ The general structure of a four-segment pipeline is as shown in fig;
- ✓ A task is the total operation performed going through all segments of a pipeline



Four-segment pipeline.

- ✓ The behavior of a pipeline can be illustrated with a *space-time* diagram

- ✓ This shows the segment utilization as a function of time
- ✓ Once the pipeline is full, it takes only one clock period to obtain an output

  Consider a *k*-segment pipeline with a clock cycle time t_p to execute *n* tasks

Space-time diagram for pipeline.

Segment:	1	2	3	4	5	6	7	8	9	→ Clock cycles
1	T_1	T_2	T_3	T_4	T_5	T_6				
2		T_1	T_2	T_3	T_4	T_5	T_6			
3			T_1	T_2	T_3	T_4	T_5	T_6		
4				T_1	T_2	T_3	T_4	T_5	T_6	

- ✓ The first task T_1 requires time kt_p to complete
- ✓ The remaining $n - 1$ tasks finish at the rate of one task per clock cycle and will be completed after time $(n - 1)t_p$
- ✓ The total time to complete the *n* tasks is $[k + n - 1]t_p$
- ✓ The above example requires [4 + 6 – 1] clock cycles to finish
- ✓ Consider a non-pipeline unit that performs the same operation and takes t_n time to complete each task
- ✓ The total time to complete *n* tasks would be nt_n
- ✓ The *speedup* of a pipeline processing over an equivalent non-pipeline processing is defined by the ratio

$$S = \frac{nt_n}{(k + n - 1)t_p}$$

- ✓ As the number of tasks increase, the speedup becomes

$$S = \frac{t_n}{t_p}$$

- ✓ If we assume that the time to process a task is the same in both circuits, $t_n = k\,t_p$

$$S = \frac{kt_p}{t_p} = k$$

- ✓ Therefore, the theoretical maximum speedup that a pipeline can provide is *k*
- ✓ Example:

  Cycle time = t_p = 20 ns # of segments = k = 4 # of tasks = n = 100

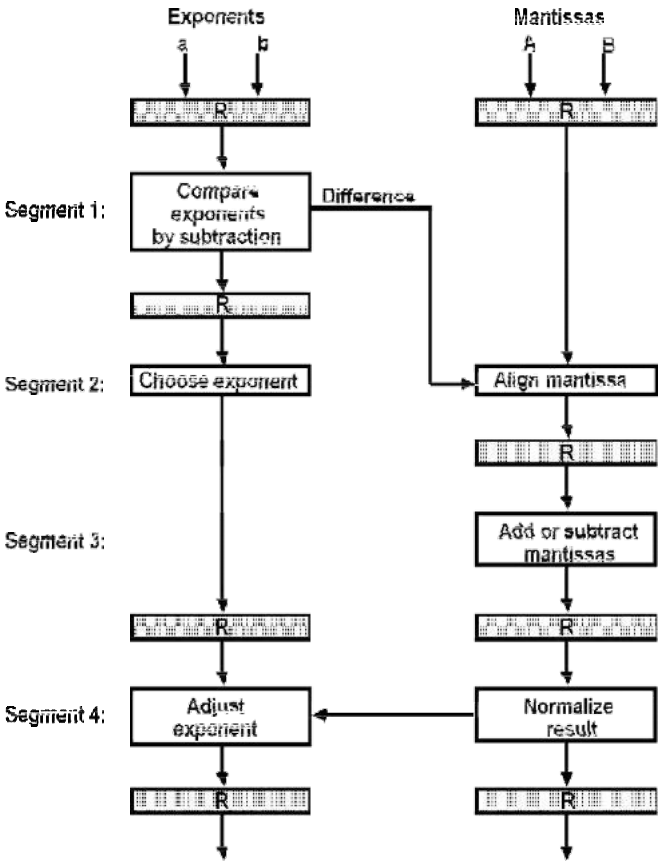
- ✓ The pipeline system will take $(k + n - 1)t_p$ = (4 + 100 –1)20ns = 2060 ns
- ✓ Assuming that $t_n = kt_p$ = 4 * 20 = 80 ns,
- ✓ A non-pipeline system requires nkt_p = 100 * 80 = 8000 ns
- ✓ The speedup ratio = 8000/2060 = 3.88
- ✓ The pipeline cannot operate at its maximum theoretical rate
- ✓ One reason is that the clock cycle must be chosen to equal the time delay of the segment with the maximum propagation time
- ✓ Pipeline organization is applicable for arithmetic operations and fetching instructions

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
- ✓ Example for floating-point addition and subtraction
- ✓ Inputs are two normalized floating-point binary numbers

  $X = A \times 2^a$

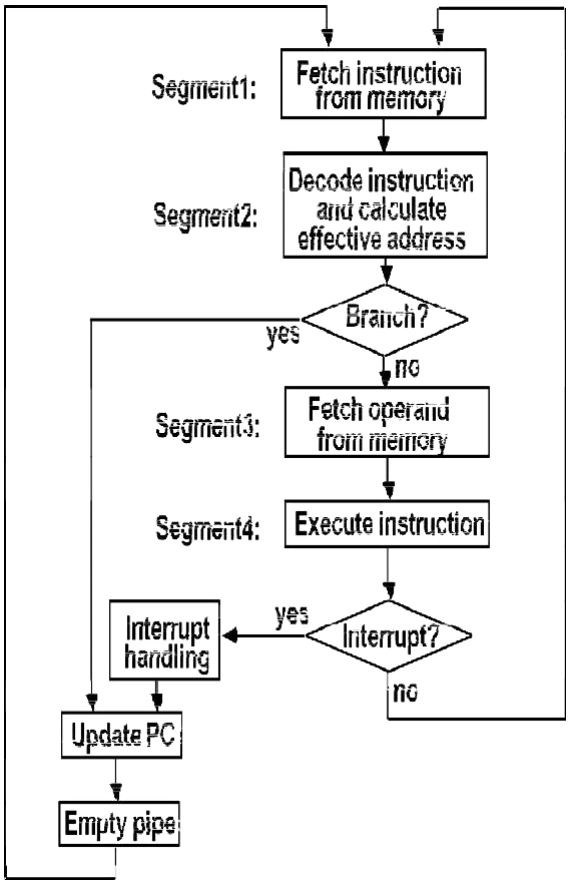
  $Y = B \times 2^b$
- ✓ A and B are two fractions that represent the mantissas
- ✓ a and b are the exponents
- ✓ Four segments are used to perform the following:
  - Compare the exponents
  - Align the mantissas
  - Add or subtract the mantissas
  - Normalize the result
- ✓ $X = 0.9504 \times 10^3$ and $Y = 0.8200 \times 10^2$
- ✓ 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
- ✓ Segment 2 shifts the mantissa of Y to the right to obtain $Y = 0.0820 \times 10^3$
- ✓ The mantissas are now aligned
- ✓ Segment 3 produces the sum $Z = 1.0324 \times 10^3$



- ✓ Segment 4 normalizes the result by shifting the mantissa once to the right and incrementing the exponent by one to obtain $Z = 0.10324 \times 10^4$
- ✓ The comparator, shifter, adder-subtractor, incrementer, and decrementer in the floating-point pipeline are implemented with combinational circuits.
- ✓ Suppose that the time delays of the four segments are t1 = 60 ns, t2 = 70 ns, t3 = 100 ns, t4 = 80 ns, and the interface registers have a delay of tr = 10 ns. The clock cycle is chosen to be tp = t3 + tr = 110 ns. An equivalent non-pipeline floating point adder-subtractor will have a delay time tn = t1 + t2 + t3 + t4 + tr = 320ns. In this case the pipelined adder has a speedup of 32011 10 = 2. 9 over the non-pipelined adder.

INSTRUCTION PIPELINE

- ✓ 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
- ✓ If a branch out of sequence occurs, 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 forming a two segment pipeline
- ✓ A FIFO buffer can be used for the fetch segment
- ✓ Thus, an instruction stream can be placed in a queue, waiting for decoding and processing by the execution segment
- ✓ This reduces the average access time to memory for reading instructions
- ✓ Whenever there is space in the buffer, the control unit initiates the next instruction fetch phase
- ✓ The following steps are needed to process each instruction:
    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
- ✓ The pipeline may not perform at its maximum rate due to:
- ➢ Different segments taking different times to operate
- ➢ Some segment being skipped for certain operations
- ➢ Memory access conflicts



Example: Four-segment instruction pipeline

- ✓ Assume that the decoding can be combined with calculating the EA in one segment
- ✓ Assume that most of the instructions store the result in a register so that the execution and storing of the result can be combined in one segment
- ✓ 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

- ✓ Up to four sub operations in the instruction cycle can overlap and up to four different instructions can be in progress of being processed at the same time
- ✓ The following figure shows the operation of the instruction pipeline. The four segments are represented in the diagram with an abbreviated symbol.
- ➢ FI: Fetch an instruction from memory
- ➢ DA:Decode the instruction and calculate the effective address of the operand
- ➢ FO: Fetch the operand
- ➢ EX: Execute the operation

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

- ✓ It is assumed that the processor has separate instruction and data memories
- ✓ 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.
- ✓ Reasons for the pipeline to deviate from its normal operation are:
- ✓ **Resource conflicts** caused by access to memory by two segments at the same time. Most of these instructions can be resolved by using separate instruction and data memories.
- ✓ **Data dependency** conflicts arise when an instruction depends on the result of a previous instruction, but his result is not yet available
- ✓ **Branch difficulties** arise from program control instructions that may change the value of PC

Methods to handle data dependency:

- ✓ **Hardware interlocks** are circuits that detect instructions whose source operands are destinations of prior instructions. Detection causes the hardware to insert the required delays without altering the program sequence.
- ✓ **Operand forwarding** uses special hardware to detect a conflict and then avoid it by routing the data through special paths between pipeline segments. This requires additional hardware paths through multiplexers as well as the circuit to detect the conflict.
- ✓ **Delayed load** is a procedure that gives the responsibility for solving data conflicts to the compiler. The compiler 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.

Methods to handle branch instructions:

- ✓ **Prefetching the target instruction** in addition to the next instruction allows either instruction to be available.
- ✓ A **branch target buffer (BTB)** is an associative memory included in the fetch segment of the branch instruction that stores the target instruction for a previously executed branch. It also stores the next few instructions after the branch target instruction. This way, the branch instructions that have occurred previously are readily available in the pipeline without interruption.
- ✓ The **loop buffer** is a variation of the BTB. It is a small very high speed register file maintained by the instruction fetch segment of the pipeline. Stores all branches within a loop segment.
- ✓ **Branch prediction** uses some additional logic to guess the outcome of a conditional branch instruction before it is executed. The pipeline then begins pre-fetching instructions from the predicted path.
- ✓ **Delayed branch** is used in most RISC processors so that the compiler rearranges the instructions to delay the branch.

RISC Pipeline

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 aninstruction pipeline using a small number of sub-operations, with each being executed in oneclock cycle. Because of the fixed-length instruction format, the decoding of the operation canoccur at the same time as the register selection. All data manipulation instructions haveregister-to-register operations. Since all operands are in registers, there is no needfor 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 destinationregister.

- 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.
    - Cache memory: operate at the same speed as the CPU clock
- One of the major advantages of RISC is its ability to execute instructions at the rate of one per clock cycle.
    - In effect, it is to start each instruction with each clock cycle and to pipeline the processor to achieve the goal of single-cycle instruction execution.
    - RISC can achieve pipeline segments, requiring just one clock cycle.
- *Compiler* supported 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.

Example: Three-Segment Instruction Pipeline

- A typical set of instructions for a RISC processor are discussed earlier.
- Thee are three types of instructions:
    - The data manipulation instructions: operate on data in processor registers
    - The data transfer instructions:
    - The program control instructions:

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).
- 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 sub-operations and implemented inthree segments:
    - I: Instruction fetch
        - Fetches the instruction from program memory
    - A: ALU operation
        - The instruction is decoded and an ALU operation is performed.
        - It performs an operation for a data manipulation instruction.
        - It evaluates the effective address for a load or store instruction.
        - It calculates the branch address for a program control instruction.
    - E: Execute instruction
        - 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.
        - It transfers the branch address to the program counter.

Delayed Load

- Consider the operation of the following four instructions:
    1. LOAD: R1 ← M[address 1]
    2. LOAD: R2 ← M[address 2]
    3. ADD: R3 ← R1 +R2
    4. STORE: M[address 3] ← R3

  There will be a *data conflict* in instruction 3 because the operand in R2 is not yetavailable 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 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 toR2 instruction. The data is loaded into R2 in clock cycle 4. The add instruction uses the valueof R2 in step 5. Thus the no-op instruction isused 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 doesnot have to check if the content of the register being accessed is currently valid or not.

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

9 (a) Pipeline timing with data conflict

	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

9 (b) Pipeline timing with delayed load

Delayed Branch

- 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 is designed to analyze the instructions *before and after the branch* and *rearrange the program sequence* by inserting useful instructions in the delay steps.
- 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

- 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 program in Fig. 9-10(b) is rearranged by placing the add and subtract instructions *after the branch instruction*.
- PC is updated to the value of X in clock cycle 5.

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

10 (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

10 (b) Rearranging the instructions

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. Inother 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 105and 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 cycle6 is from address 350, which is the instruction at the branch address.

Vector Processing

- In many science and engineering applications, the problems can be formulated in terms of vectors and matrices that lend themselves to vector processing.
- Computers with vector processing capabilities are in demand in specialized applications. e.g.

- Long-range weather forecasting
- Petroleum explorations
- Seismic data analysis
- Medical diagnosis
- Artificial intelligence and expert systems
- Image processing
- Mapping the human genome

- 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.
- 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=[v1,v2,…,Vn].
- To examine the difference between a conventional scalar processor and a vector processor, consider the following Fortran DO loop:

```
   DO 20 I = 1, 100
20      C(I) = B(I) + A(I)
```

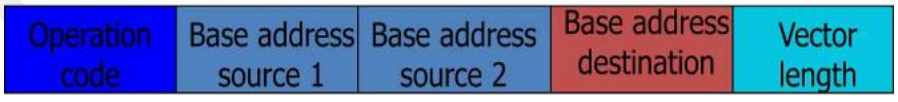
- This is implemented in machine language by the following sequence of operations.

```
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
```

- 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.

```
C(1:100) = A(1:100) + B(1:100)
```

- A possible instruction format for a vector instruction is shown in Fig. 9-11.
  - 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.
  - The base address and length in the vector instruction specify a group of CPU registers.

Matrix Multiplication

- The multiplication of two n x n matrices consists of n^2 inner products or n^3 multiply-add operations.
- Consider, for example, the multiplication of two 3 x 3 matrices A and B.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

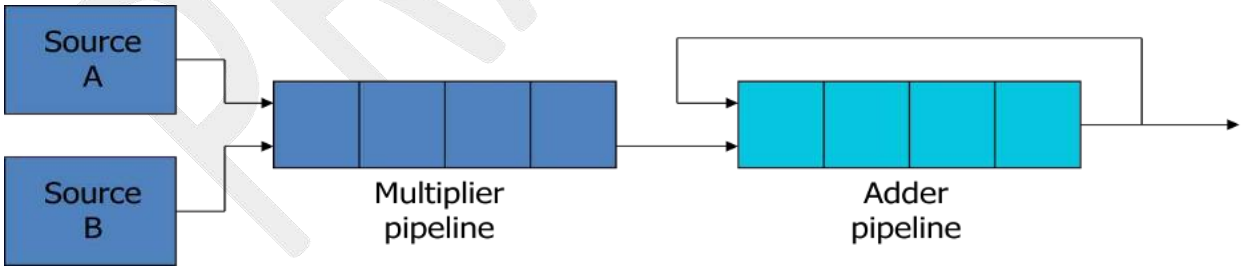
The product matrix C is a 3×3 matrix whose elements are related to the elements of A and B by the inner product:

$$c_{ij} = \sum_{k=1}^{3} a_{ik} \times b_{kj}$$

For example, the number in the first row and first column of matrix C is calculated by letting $i = 1, j = 1$, to obtain

  - $c_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31}$
  - This requires three multiplication and (after initializing c_{11} to 0) three additions.
- In general, the inner product consists of the sum of *k* product terms of the form $C = A_1B_1+A_2B_2+A_3B_3+\ldots+A_kB_k$.
  - 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.

$$\begin{aligned} C = {} & A_1B_1 + A_5B_5 + A_9B_9 + A_{13}B_{13} + \cdots \\ & + A_2B_2 + A_6B_6 + A_{10}B_{10} + A_{14}B_{14} + \cdots \\ & + A_3B_3 + A_7B_7 + A_{11}B_{11} + A_{15}B_{15} + \cdots \\ & + A_4B_4 + A_8B_8 + A_{12}B_{12} + A_{16}B_{16} + \cdots \end{aligned}$$

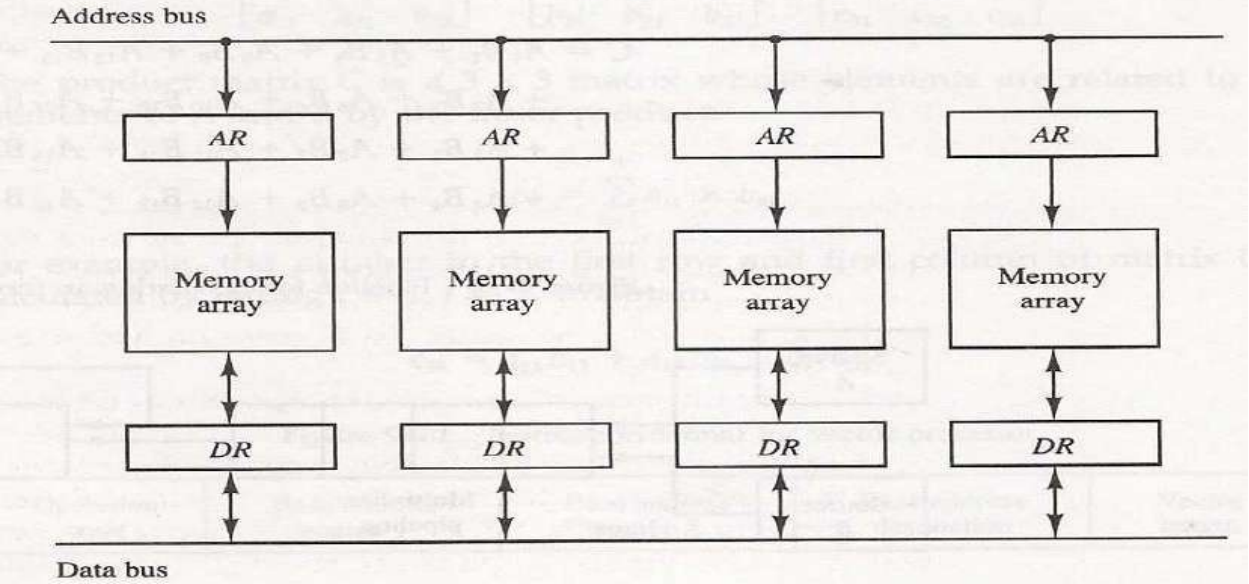


Memory Interleaving

- *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.
  - 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.
- Fig. 9-13 shows a memory unit with four 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.
- By staggering the memory access, the effective memory cycle time can be *reduced by a factor close to the number of modules*.

Supercomputers

- A commercial computer with vector instructions and pipelined floating-point arithmetic operations is referred to as a *supercomputer*.
    - To speed up the operation, the components are *packed tightly* together to minimize the distance that the electronic signals have to travel.
- 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.
- It is specifically optimized for the type of numerical calculations involving vectors and matrices of floating-point numbers.
- They are limited in their use to a number of scientific applications, such *as numerical weather forecasting, seismic wave analysis, and space research*.
- 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*.
- A typical supercomputer has a basic cycle time of 4 to 20 ns.
- The examples of supercomputer:

- Cray-1: it uses vector processing with 12 distinct functional units in parallel; a large number of registers (over 150); multiprocessor configuration (Cray X-MP and Cray Y-MP)
- Fujitsu VP-200: 83 vector instructions and 195 scalar instructions; 300 megaflops

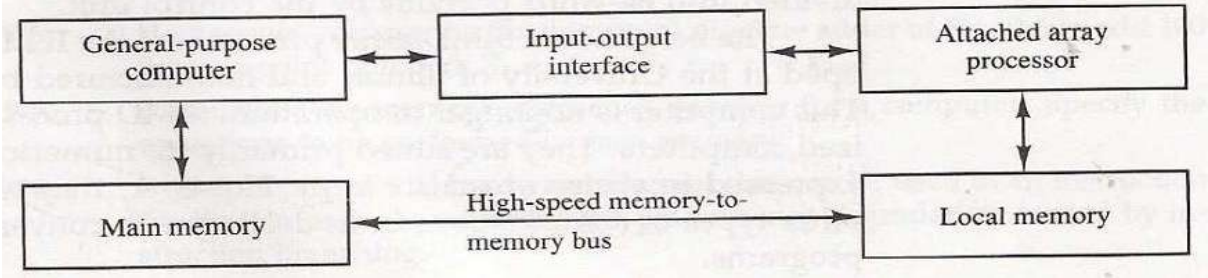
Array Processors : Introduction

- 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.
  - Attached array processor:
    - Is an auxiliary processor.
    - It is intended to improve the performance of the host computer in specific numerical computation tasks.
  - SIMD array processor:
    - Has a single-instruction multiple-data organization.
    - It manipulates vector instructions by means of multiple functional units responding to a common instruction.

Attached Array Processor

- Its purpose is to enhance the performance of the computer by providing vector processing for complex scientific applications.
  - Parallel processing with multiple functional units
- Fig. 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.

Figure 9-14 Attached array processor with host computer.



- 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 100megaflops.
- The objective of the attached array processor is to provide *vector manipulation capabilities* to a conventional computer at a fraction of the cost of supercomputer.

SIMD Array Processor

- An SIMD array processor is a computer with multiple processing units operating in parallel.
- 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 PE includes an ALU, a floating-point arithmetic unit, and working registers.
    - Vector instructions are broadcast to all PEs simultaneously.
- 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.

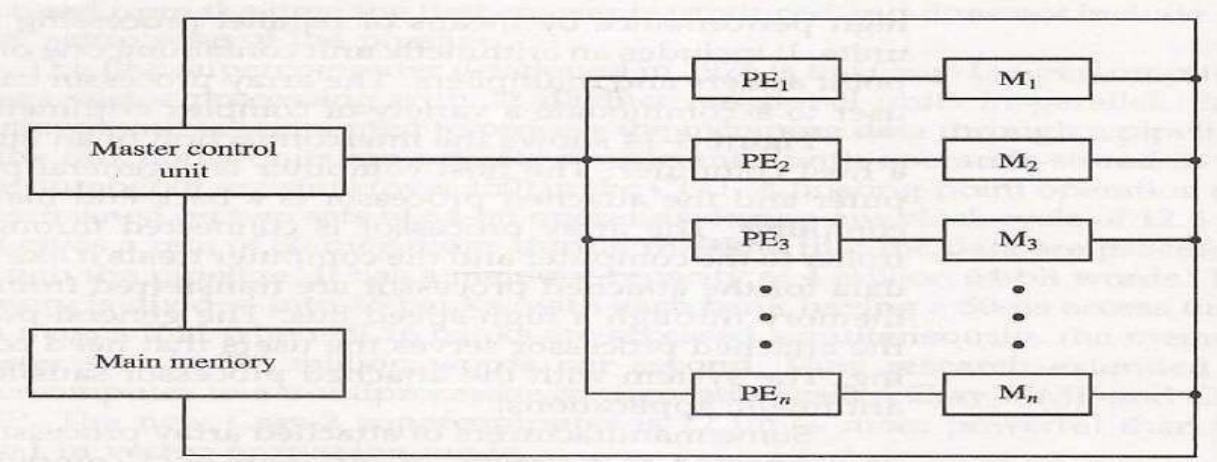


Figure 9-15 SIMD array processor organization.

- For example, the ILLIAC IV computer developed at the University of Illinois and manufactured by the Burroughs Corp.
    - Are highly specialized computers.
    - They are suited primarily for numerical problems that can be expressed in vector or matrix form.

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 (lOP).
- ✓ As it is most commonly defined, a multiprocessor system implies the existence of multiple CPUs. 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. The network consists of several autonomous computers that may or may not communicate with each other. A multiprocessor system is controlled by one operating system that provides interaction between processors and all the components of the system cooperate in the solution of a problem.
- ✓ Although some large-scale computers include two or more CPUs in their overall system. 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
- ✓ 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.
- ✓ The benefit derived from a multiprocessor organization is an improved system performance. The system derives its high performance 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.
- ✓ 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**. Most commercial tightly coupled multiprocessors provide a cache memory with each CPU.
- ✓ 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.
- ✓ 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.

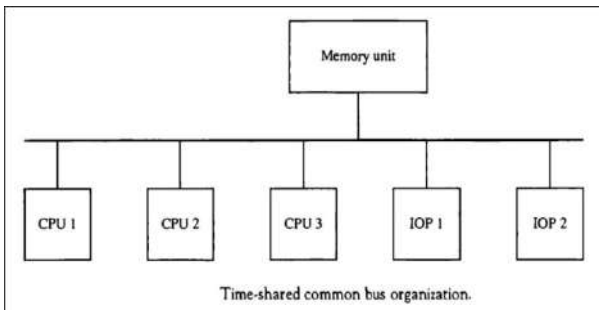
INTERCONNECTION STRUCTURES

- ✓ The components that form a multiprocessor system are CPUs, lOPs 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:

1. Time-shared common bus
2. Multiport memory
3. Crossbar switch
4. Multistage switching network
5. Hypercube system

Time-Shared Common Bus

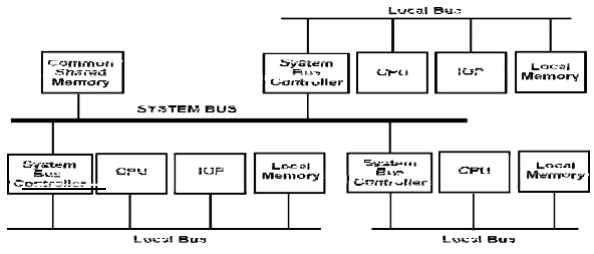
- ✓ A common-bus multiprocessor system consists of a number of processors connected through a common path to a memory unit. A time-shared common bus for five processors is shown in Fig.



Time-shared common bus organization.

- ✓ Only one processor can communicate with the memory or another processor at any given time.
- ✓ Transfer operations are conducted by the processor that is in control of the bus at the time.
- ✓ 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 transfer 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.
- ✓ 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.
- ✓ A more economical implementation of a dual bus structure is depicted in Fig.
- ✓ Each local bus may be connected to a CPU, an lOP, or any combination of processors.

SYSTEM BUS STRUCTURE FOR MULTIPROCESSORS

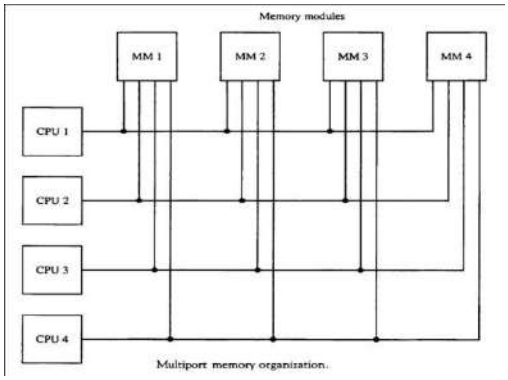


- ✓ A system bus controller links each local bus to a common system bus.

- ✓ The IO devices connected to the local lOP, as well as the local memory, are available to the local processor.
- ✓ If an lOP is connected directly to the system bus, the IO 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 IO devices.
- ✓ Part of the local memory may be designed as a cache memory attached to the CPU

Multiport Memory

- ✓ A multiport memory system employs separate buses between each memory module and each CPU. This is shown in Fig. 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.

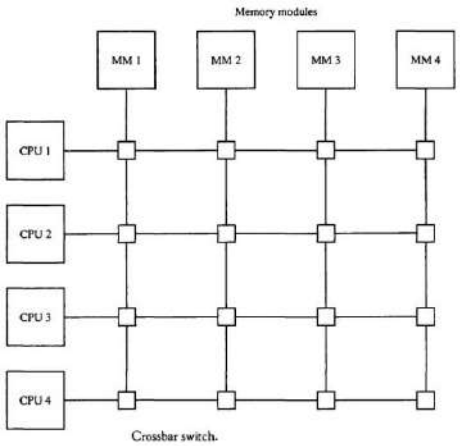


Multiport memory organization.

- ✓ 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.
- ✓ The advantage of the multi port 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.

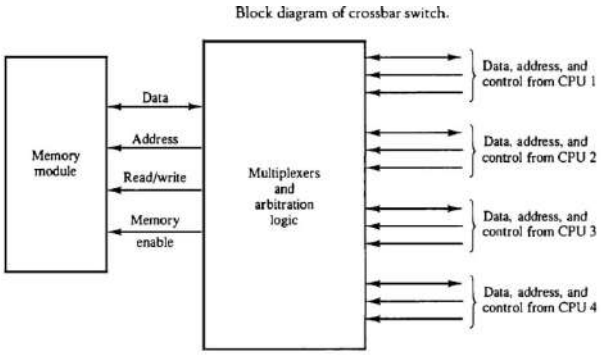
Crossbar Switch

- ✓ The crossbar switch organization consists of a number of cross points that are placed at intersections between processor buses and memory module paths.
- ✓ The small square in each cross point 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.



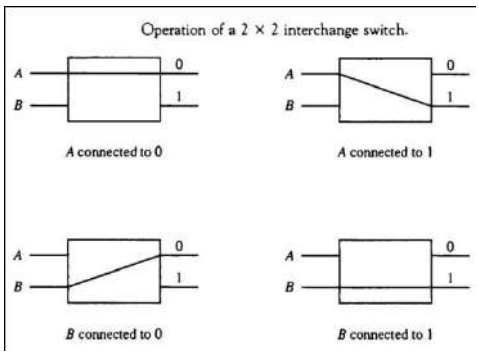
Crossbar switch.

- ✓ 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.
- ✓ The functional design of a crossbar switch connected to one memory module is shown in figure.
- ✓ The circuit consists of multiplexers that select the data address, and control from one CPU for communication with the memory module.
- ✓ Priority levels are established by the arbitration logic to select one CPU when two or more CPUs attempt to access the same memory.
- ✓ A crossbar switch organization supports simultaneous transfers from memory modules because there is a separate path associated with each module.

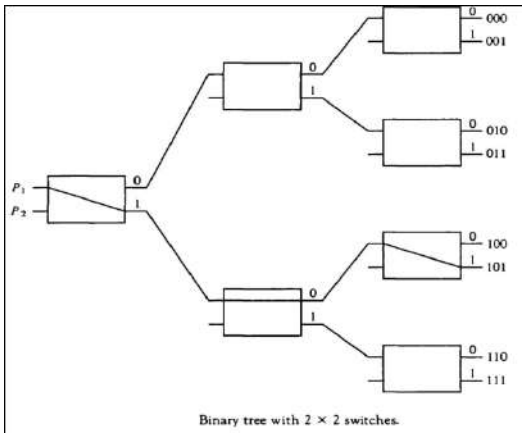


Multistage Switching Network

- ✓ The basic component of a multistage network is a two-input, two-out interchange switch.
- ✓ 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.

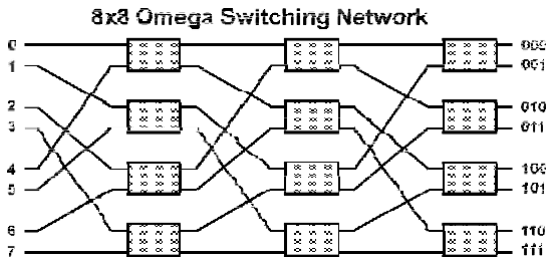


- ✓ Using the 2 x 2 switch as a building block, it is possible to build multistage network to control the communication between a number of sources and destinations.
- ✓ Consider the binary tree shown Fig. The two processors P1 and P2 are connected through switches to eight memory modules marked in binary from 000 through 111.
- ✓ The path from source to a destination is determined from the binary bits of the destination number. 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.



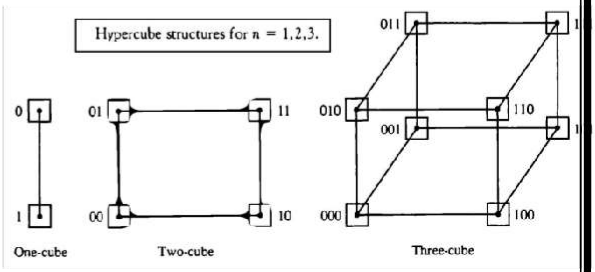
- ✓ Many different topologies have been proposed for multistage switching networks to control processor-memory communication in a tightly coupled multiprocessor system or to control the communication between the processing elements in a loosely coupled system.

- ✓ One such topology is the omega switching network shown in Fig.
- ✓ 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.



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.
- ✓ Each processor has direct communication paths to n other neighbor processors. These paths correspond to the edges of the cube.
- ✓ Fig 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.
- ✓ 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).
- ✓ 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 XOR 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.

INTERPROCESSOR ARBITRATION

- ✓ 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 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.
- ✓ 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.

System Bus

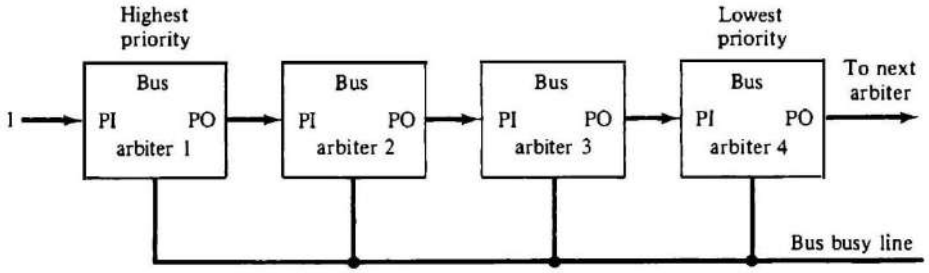
- ✓ A typical system bus consists of approximately 100 signal lines. These lines are divided into three functional groups: data, address, and control. In addition, there are power distribution lines that supply power to the components.
- ✓ For example, the IEEE standard 796 multibus system has 16 data lines, 24 address lines, 26 control lines, and 20 power lines, for a total of 86 lines.
- ✓ 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.
- ✓ In an **asynchronous bus**, each data item being transferred is accompanied by handshaking control signals to indicate when the data are transferred from the source and received by the destination
- ✓ The following table lists the 86 lines that are available in the IEEE standard 796 multibus.

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)	

Serial Arbitration Procedure

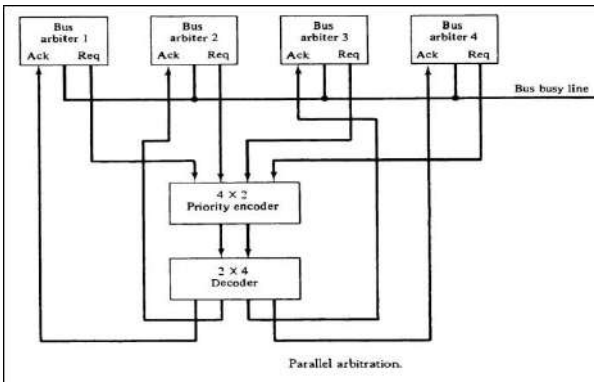
- ✓ Arbitration procedures service all processor requests on the basis of established priorities. A hardware bus priority resolving technique can be established by means of a serial or parallel connection of the units requesting control of the system bus.
- ✓ The serial priority resolving technique is obtained from a daisy-chain connection of bus arbitration circuits similar to the priority interrupt logic.
- ✓ 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.



Serial (daisy-chain) arbitration.

- ✓ 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.
- ✓ 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



Parallel arbitration.

- ✓ The parallel bus arbitration technique uses an external priority encoder and a decoder as shown in Fig. 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 bus. The processor takes control of the bus if its acknowledge input line is enabled.

Dynamic Arbitration Algorithms

- ✓ A dynamic priority algorithm gives the system the capability for changing the priority of the devices while the system is in operation.
- ✓ 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.
- ✓ 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.
- ✓ 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.
- ✓ 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.
- ✓ 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.
- ✓ 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.
- ✓ 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.

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 lOP 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 lOPs associated with two different CPUs. This type of link allows each CPU to treat the other as an IO device so that messages can be transferred through the IO 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 IO 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.
- ✓ 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.
- ✓ 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 ←M[SEM] Test semaphore

  M[SEM]←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.

cache Coherence

The primary advantage of cache is its ability to reduce the average access time in uni-processors. 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 Multiprocessors without cache coherence enforcement mechanisms.

Sometime during the operation an element X from main memory is loaded into the three processors, P1, P2, and P3• 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. . A store to X (of the value of 120) into the cache of processor P1 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

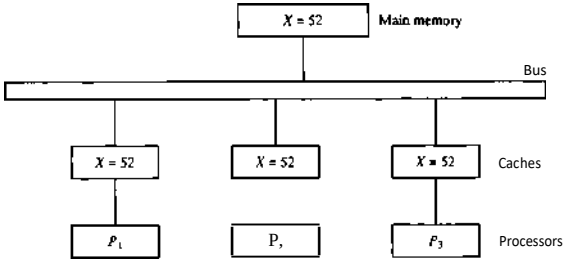
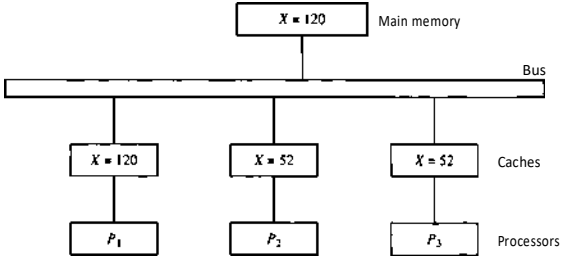
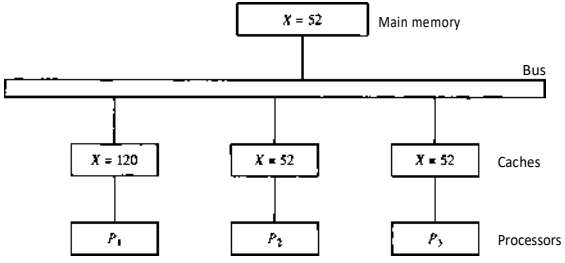


Figure. Cache configuration after a load on X.

Fig. Cache configuration after a store to X by processor P1•



(a) With write-through cache policy



(b) With write-back cache policy