



**Malla Reddy Engineering College**  
(UGC Autonomous Institution, Approved by AICTE, & Affiliated to JNTUH).  
Accredited by NAAC with 'A++' Grade (Cycle-III), Maisammaguda (H),  
Medchal-Malkajgiri District, Secunderabad, Telangana-500100, [www.mrec.ac.in](http://www.mrec.ac.in)



**Department of Information Technology**

**II B.TECH I SEM (A.Y.2024-25)**

**Lecture Notes**

**On**

**C0509-COMPUTER ORGANIZATION & ARCHITECTURE**

<b>2022-23 Onwards (MR-22)</b>	<b>MALLA REDDY ENGINEERING COLLEGE (Autonomous)</b>	<b>B.Tech. III Semester</b>		
<b>Code: C0509</b>	<b>Computer Organization and Architecture (Common for CSE, CSE(DS), CSE (AI and ML), CSE(Cyber Security), CSE(IOT), AIML and IT)</b>	<b>L</b>	<b>T</b>	<b>P</b>
<b>Credits: 3</b>		<b>3</b>	<b>-</b>	<b>-</b>

**Prerequisites:** A Course on “DLD”.

### Objectives

1. The purpose of the course is to introduce principles of Digital fundamentals computer organization and the basic architectural concepts.
2. It begins with basic organization, design, and programming of a simple digital computer and introduces simple register transfer language to specify various computer operations.
3. Topics include computer arithmetic, instruction set design, microprogrammed control unit, pipelining and vector processing, memory organization and I/O systems, and multiprocessors.

### MODULE – I

**[10 Periods]**

**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.

### MODULE – II

**[10 Periods]**

**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.

### MODULE – III

**[9 Periods]**

**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.

### MODULE – IV

**[10 Periods]**

**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.

### MODULE – V

**[9 Periods]**

**Reduced Instruction Set Computer:** CISC Characteristics, RISC Characteristics. Pipeline and Vector Processing: Parallel Processing, Pipelining, Arithmetic Pipeline, Instruction Pipeline, RISC Pipeline, Vector Processing, Array Processor. Multi Processors: Characteristics of Multiprocessors, Interconnection Structures, Interprocessor arbitration, Interprocessor communication and synchronization, Cache Coherence.

**Textbook:**

1. Computer System Architecture, M. Moris Mano, 3rd Edition, Pearson/PHI.

**References:**

1. Computer Organization, Car Hamacher, ZvonksVranesic, SafeaZaky, 5th Edition, McGraw Hill.
2. Computer Organization and Architecture, William Stallings 6th Edition, Pearson/PHI.
3. Structured Computer Organization, Andrew S. Tanenbaum, 4th Edition, PHI/Pearson.

**E-Resources:**

1. <https://books.google.co.in/books?isbn=8131700704>
2. [http://ndl.iitkgp.ac.in/document/yVCWqd6u7wgye1qwH9xY7Eh9eBOsT1ELoYpKlg\\_xngrkluevXOJLs1TbxS8q2icgUs3hL4\\_KAi5So5FgXcVg](http://ndl.iitkgp.ac.in/document/yVCWqd6u7wgye1qwH9xY7Eh9eBOsT1ELoYpKlg_xngrkluevXOJLs1TbxS8q2icgUs3hL4_KAi5So5FgXcVg)
3. [http://ndl.iitkgp.ac.in/document/yVCWqd6u7wgye1qwH9xY7xAYUzYSIXI4zudlsolr-e7wQNrNXLxbgGFxbkoyx1iN3YbHuFrzI2jc\\_70rWMEwQ](http://ndl.iitkgp.ac.in/document/yVCWqd6u7wgye1qwH9xY7xAYUzYSIXI4zudlsolr-e7wQNrNXLxbgGFxbkoyx1iN3YbHuFrzI2jc_70rWMEwQ)
4. <http://nptel.ac.in/courses/106106092/>

**Outcomes:**

1. Understand the basics of instructions sets and their impact on processor design.
2. Demonstrate an understanding of the design of the functional units of a digital computer system.
3. Evaluate cost performance and design trade-offs in designing and constructing a computer processor including memory.
4. Design a pipeline for consistent execution of instructions with minimum hazards.
5. Recognize and manipulate representations of numbers stored in digital computers.

CO- PO, PSO Mapping (3/2/1 indicates strength of correlation) 3-Strong, 2-Medium, 1-Weak															
COs	Programme Outcomes (POs)												PSOs		
	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3
CO1	3	2	2									2			
CO2		2	3									1			
CO3	2	2	3	2	2							2			
CO4	3														
CO5	3														

# COMPUTER ORGANIZATION AND ARCHITECTURE

## MODULE-I

### DIGITAL COMPUTER

A digital system that performs various computational tasks.

**Digital:** The word digital implies that the information in the computer is represented by variables that take a limited number of discrete values. These discrete values are processed internally by components that can maintain a limited number of discrete states. (0,1,...9 provide 10 discrete values)

First electronic digital computer developed in 1940s which was used primarily for numerical computations (using discrete elements i.e., digits). From this application the term digital computer has emerged.

Digital computers function more reliably if only two states are used because of physical restriction of components and because of human logic tends to binary (True/False or YES/NO statements)

Digital components that are constrained to take discrete values are further constrained to take only two values- binary 0/1.

Information in digital computers is represented by a group of bits (binary digit is called a bit).

By using various coding techniques, groups of bits can be made to represent not only binary numbers but also other discrete symbols-decimal digits, letters of alphabets.

By use of binary arrangements and by using various coding techniques, the groups of bits are used to develop computer sets of instructions for performing various types of computations.

### BLOCK DIAGRAM OF DIGITAL COMPUTER

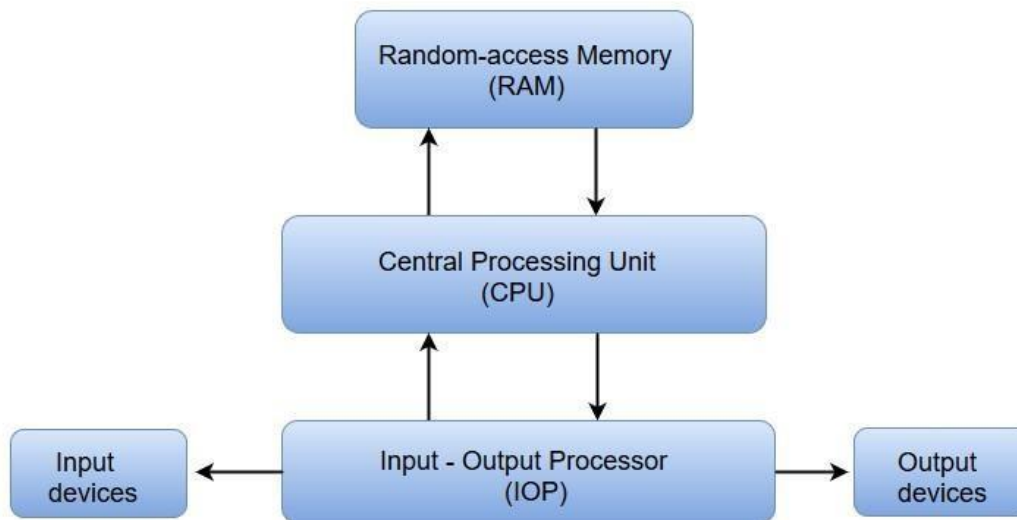
A computer system is subdivided into two functional entities: hardware, software.

**Hardware**-it consists of all the electronic components and electric mechanical devices that comprise the physical entity of the device.

**Software**-it consists of the instructions and data that the computer manipulates to perform various data processing tasks.



### Block diagram of a digital computer:



**Central Processing Unit (CPU):** Contains an arithmetic and logic unit for manipulating data, a number of registers for storing data, and a control circuit for fetching and executing instructions.

**Memory Unit (MU):** Contains storage for instructions and data. ROM(Read-Only Memory) is a part of memory unit. MU is also called as Primary memory/Internal memory/Principal memory.

**Random Access Memory (RAM):** used for real-time processing of the data. It allows your computer to switch between programs and have large files ready to view.

**Input-Output devices:** They are used for generating inputs from the user and displaying the final results to the user.

Eg: keyboard, mouse, terminals, magnetic disk drives, and other communication devices.

## COMPUTER ORGANIZATION

It deals with the internal view of the computer and the roles that the internal components play during the execution of a program. It includes the organization of major parts of a computer such as the processor, memory and peripheral devices.

**Processor organization:** It deals with main components of a processor, how these are interconnected and how these operate execution of an instruction.

**Memory organization:** The memory organization of a memory unit deals with how its different components are structured and interconnected.

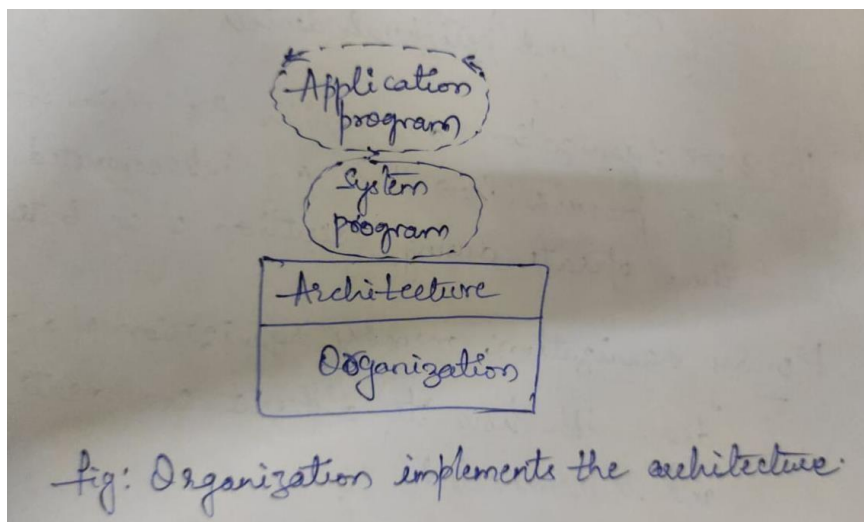
## COMPUTER ARCHITECTURE

It deals with the external view of a computer, that is, it is concerned with the structure and behavior of the computer viewed by a user such as assembly language programmer or machine language programmer.

It can be defined as an interface between hardware and software.

A programmer needs to be aware of:

- ✓ Specific instructions supported by the processor (called as processor instruction set),
- ✓ Instruction formats,
- ✓ Specific registers and their roles,
- ✓ The techniques for accessing the data stored in the memory,
- ✓ The way to perform input-output operations.



System program is directly interacting with the computer hardware. They are written for specific computer architecture.

Eg: Operating Systems, devices, drivers, compilers, etc.

Application programs invoke the services offered by the system programs. They are independent of the architecture and are converted to machine-dependent programs through a system such as a compiler.

## LOGIC GATES USED IN DIGITAL COMPUTER

Binary information is represented in digital computers by physical quantities called signals. Electrical signals such as voltages exist through out the computers in either one of the two recognizable state. The two states represent a binary variable that can be equal to 1 or 0.

Eg: A digital computer utilize a signal of 3volts to represent binary 1 and 0.5volts to represent binary 0. The input terminal of digital circuits will accept binary signals of only 3 and 0.5 volts to represent binary input and output corresponding to 1 or 0 resp.

At the core level, computer communicates in the form of 0 or 1, which is nothing but low and high voltage signals.

**The manipulation of binary information is done by logic circuits called gates. Gates are blocks of hardware that produce signals of binary 1 or 0 when input logic requirements are satisfied.**


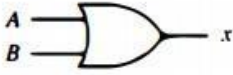

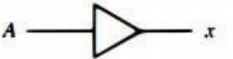
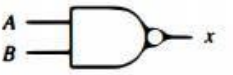
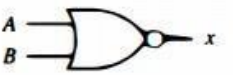

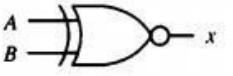
### **LOGIC GATES:**

Binary logic deals with binary variables and with operations that assume a large meaning. It is used to describe in algebraic, or tabular form, the manipulation done by logic circuits called Gates.

Gates are blocks of hardware that produce graphic symbol and it's operation can be described by means of an algebraic expression. The input-output relationship of the binary variables for each gate can be represented by a Truth-Table.

List of Logic Gates:

1. AND
2. OR
3. NOT
4. NAND
5. NOR
6. XOR
7. XNOR

Name	Graphic symbol	Algebraic function	Truth table															
AND		$x = A \cdot B$ or $x = AB$	<table border="1"> <thead> <tr> <th>A</th> <th>B</th> <th>x</th> </tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>1</td></tr> </tbody> </table>	A	B	x	0	0	0	0	1	0	1	0	0	1	1	1
A	B	x																
0	0	0																
0	1	0																
1	0	0																
1	1	1																
OR		$x = A + B$	<table border="1"> <thead> <tr> <th>A</th> <th>B</th> <th>x</th> </tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>1</td></tr> </tbody> </table>	A	B	x	0	0	0	0	1	1	1	0	1	1	1	1
A	B	x																
0	0	0																
0	1	1																
1	0	1																
1	1	1																
Inverter		$x = A'$	<table border="1"> <thead> <tr> <th>A</th> <th>x</th> </tr> </thead> <tbody> <tr><td>0</td><td>1</td></tr> <tr><td>1</td><td>0</td></tr> </tbody> </table>	A	x	0	1	1	0									
A	x																	
0	1																	
1	0																	
Buffer		$x = A$	<table border="1"> <thead> <tr> <th>A</th> <th>x</th> </tr> </thead> <tbody> <tr><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td></tr> </tbody> </table>	A	x	0	0	1	1									
A	x																	
0	0																	
1	1																	
NAND		$x = (AB)'$	<table border="1"> <thead> <tr> <th>A</th> <th>B</th> <th>x</th> </tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>0</td></tr> </tbody> </table>	A	B	x	0	0	1	0	1	1	1	0	1	1	1	0
A	B	x																
0	0	1																
0	1	1																
1	0	1																
1	1	0																
NOR		$x = (A + B)'$	<table border="1"> <thead> <tr> <th>A</th> <th>B</th> <th>x</th> </tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>0</td></tr> </tbody> </table>	A	B	x	0	0	1	0	1	0	1	0	0	1	1	0
A	B	x																
0	0	1																
0	1	0																
1	0	0																
1	1	0																
Exclusive-OR (XOR)		$x = A \oplus B$ or $x = A'B + AB'$	<table border="1"> <thead> <tr> <th>A</th> <th>B</th> <th>x</th> </tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>0</td></tr> </tbody> </table>	A	B	x	0	0	0	0	1	1	1	0	1	1	1	0
A	B	x																
0	0	0																
0	1	1																
1	0	1																
1	1	0																
Exclusive-NOR or equivalence		$x = (A \oplus B)'$ or $x = A'B' + AB$	<table border="1"> <thead> <tr> <th>A</th> <th>B</th> <th>x</th> </tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>1</td></tr> </tbody> </table>	A	B	x	0	0	1	0	1	0	1	0	0	1	1	1
A	B	x																
0	0	1																
0	1	0																
1	0	0																
1	1	1																

## 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.

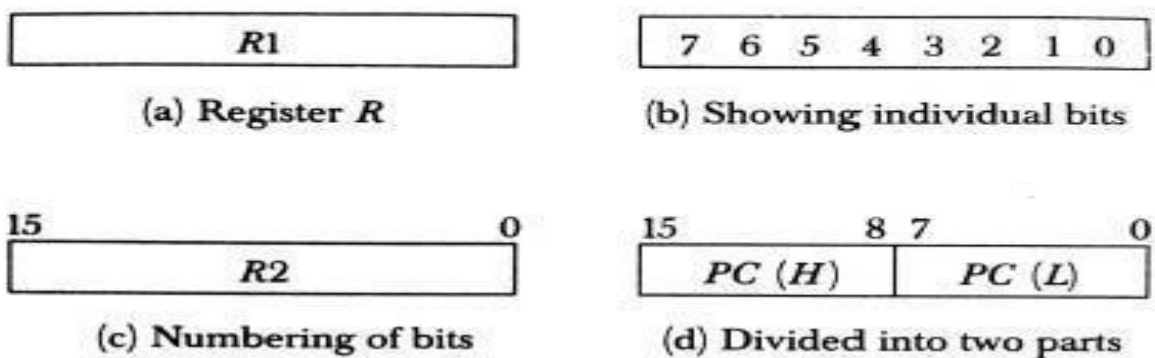
## REGISTER TRANSFER LANGUAGE

- 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 **RI** (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** Block diagram of register.



- Figure 4-1 shows the representation of registers in block diagram form.
- 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  $R2 \leftarrow R1$  denotes a transfer of the content of register  $R1$  into register  $R2$ .
- It designates a replacement of the content of  $R2$  by the content of  $R1$ .
- By definition, the content of the source register  $R1$  does not change after the transfer.
- If we want the transfer to occur only under a predetermined control condition then it can be shown by an if-then statement.

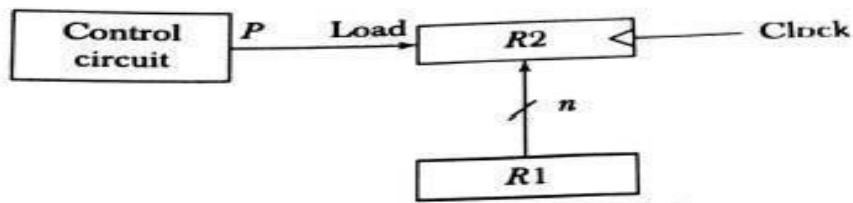
$\text{if } (P=1) \text{ then } R2 \leftarrow R1$

- $P$  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

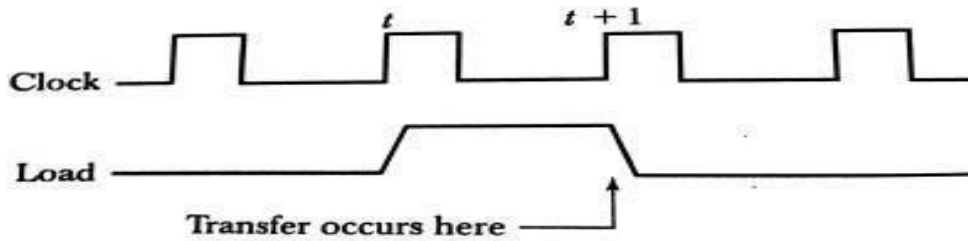
$P: R2 \leftarrow R1$

- 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)

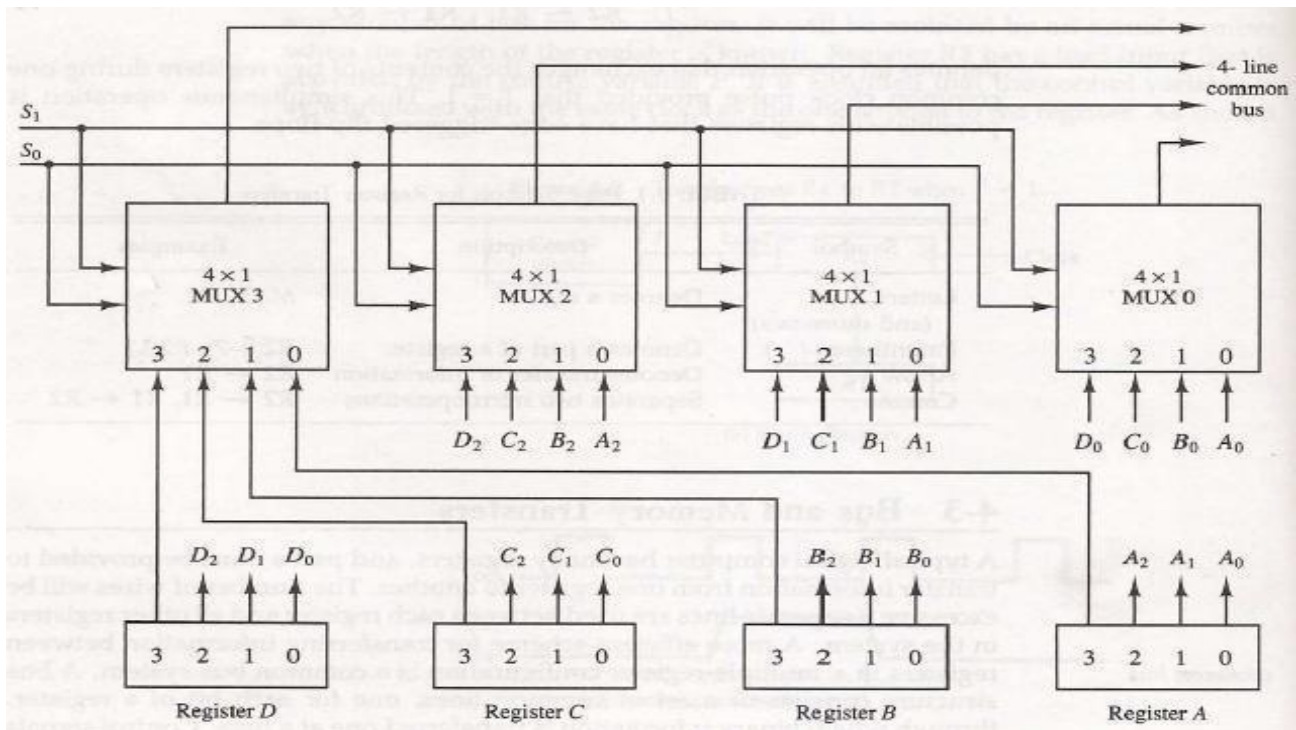
denotes an operation that exchanges the contents of two registers during one common clock pulse provided that T=1.

### Bus and Memory Transfers:

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

## COMMON BUS SYSTEM IS WITH MULTIPLEXERS:

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



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

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

In general a bus system has

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

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

$$\text{BUS} \leftarrow C, R1 \leftarrow \text{BUS}$$

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.

$$R1 \leftarrow C$$

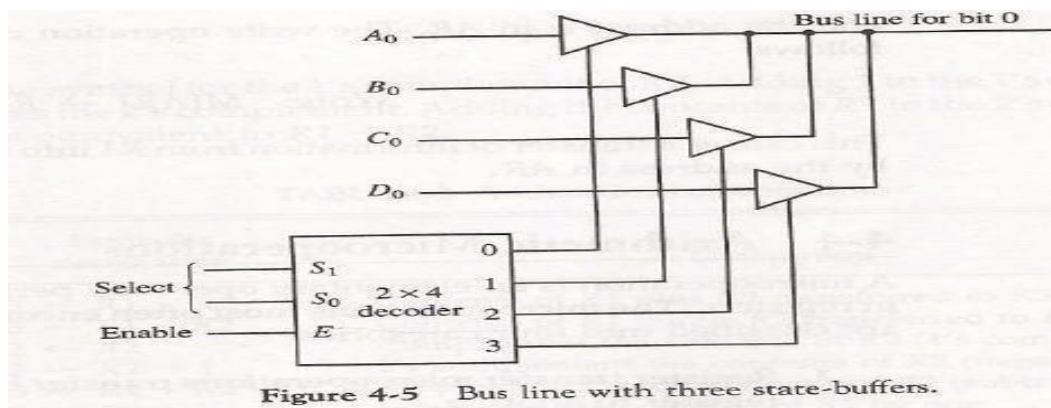
### THREE-STATE BUS BUFFERS:

- 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.

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



- The graphic symbol of a three-state buffer gate is shown in Fig. 4-4.
- 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:  $DR \leftarrow M [AR]$

- 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:  $M [AR] \leftarrow R1$

## 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 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:

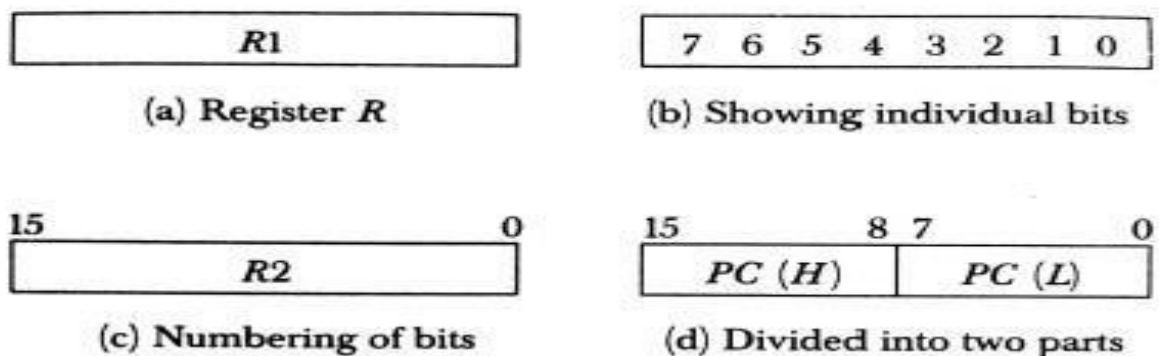
- The symbolic notation used to describe the micro-operation transfer among registers is called RTL (Register Transfer Language).
- The use of *symbols* instead of a *narrative explanation* provides an organized and concise manner for listing the micro-operation sequences in registers and the control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



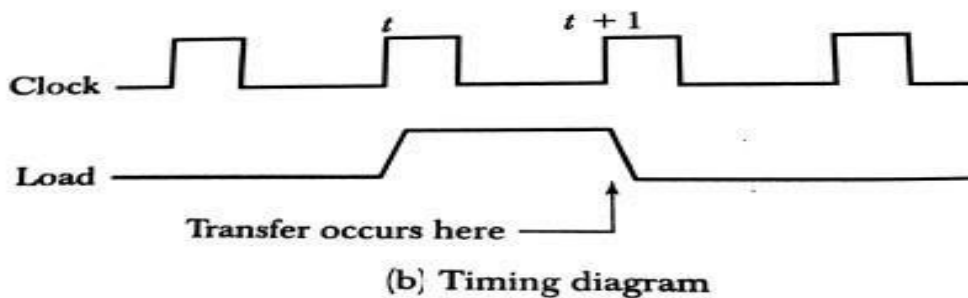
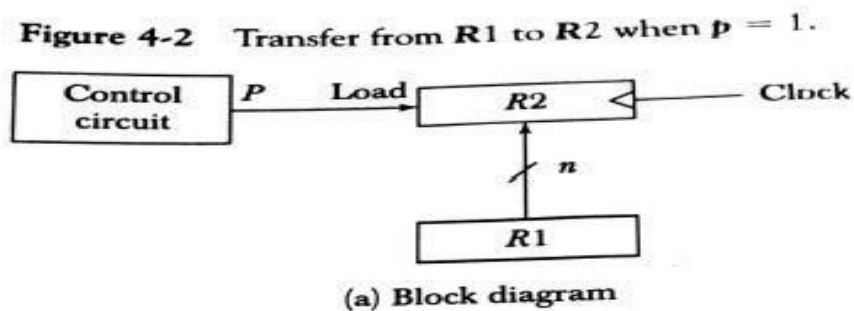
- The most common way to represent a register is by a rectangular box with the name of the register inside, as in Fig. 4-1(a).
- The individual bits can be distinguished as in (b).
- The numbering of bits in a 16-bit register can be marked on top of the box as shown in (c).
- 16-bit register is partitioned 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
 
$$P: R2 \leftarrow R1$$
- Control condition is terminated by a colon implies transfer operation be executed by the hardware only if  $P=1$ .
- Every statement written in a register transfer notation implies a hardware construction for implementing the transfer.
- Figure 4-2 shows the block diagram that depicts the transfer from R1 to R2.



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



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

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

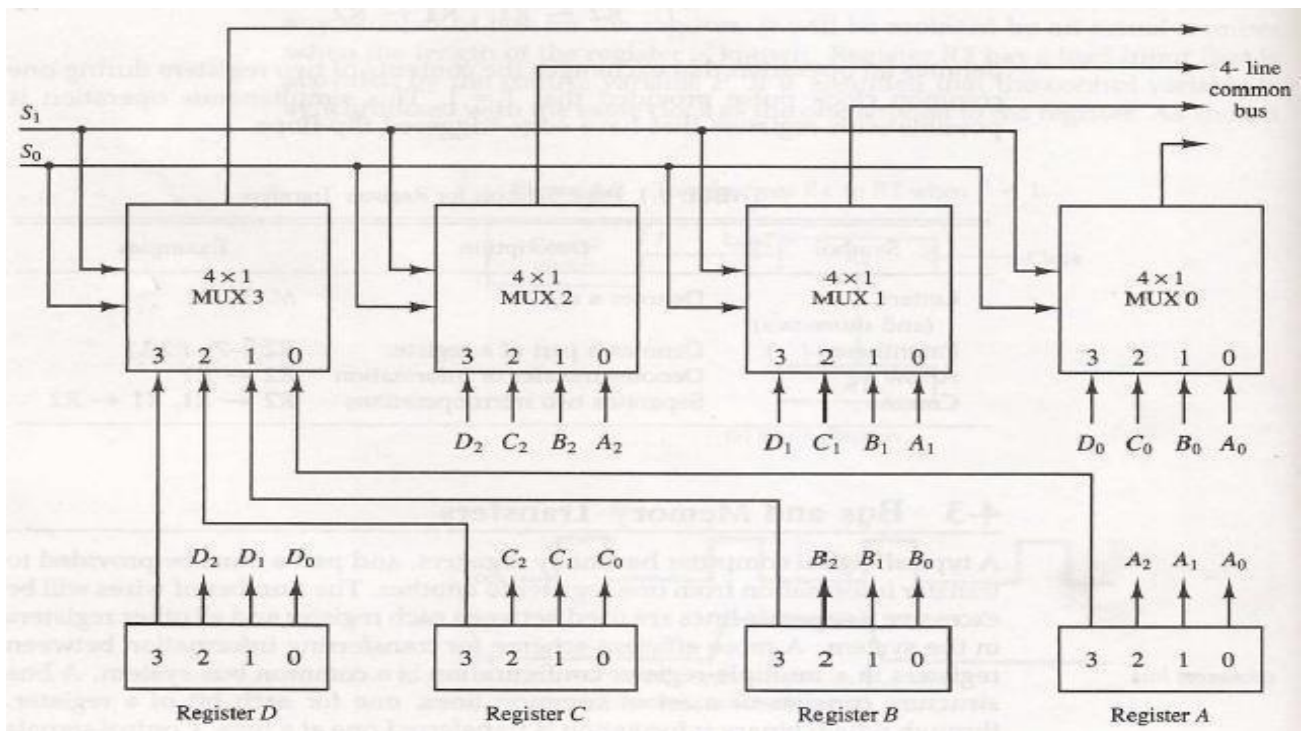
- A comma is used to separate two or more operations that are executed at the same time.
- The statement  
**T : R2 ← R1, R1 ← R2** (exchange operation)  
denotes an operation that exchanges the contents of two registers during one common clock pulse provided that T=1.

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

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

Common bus system is with multiplexers:

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



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

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

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

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

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

$$\text{BUS} \leftarrow C, R1 \leftarrow \text{BUS}$$

➤ 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.

$$R1 \leftarrow C$$

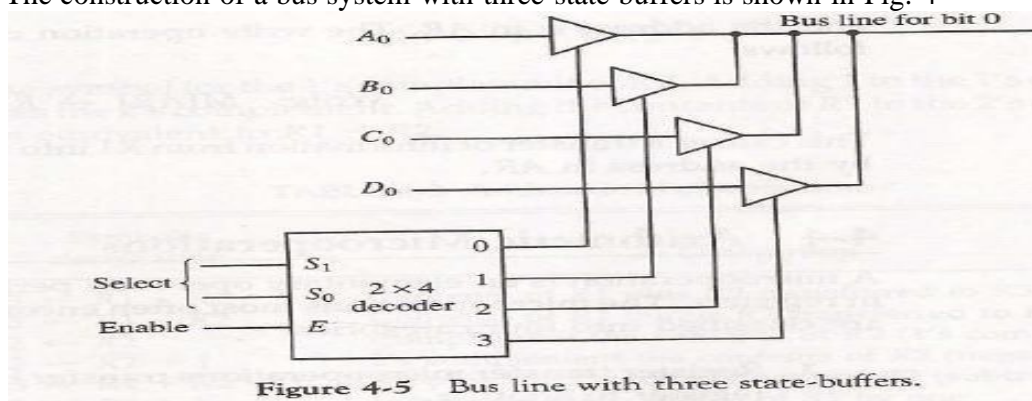
### Three-State Bus Buffers:

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

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



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



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

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

### Memory Transfer:

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

Read: DR ← M [AR]

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

Write: M [AR] ← R1

### Types of Micro-operations:

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

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

### Arithmetic Micro-operations:

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

$$R3 \leftarrow R1 + R2$$

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

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

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

### Binary Adder:

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

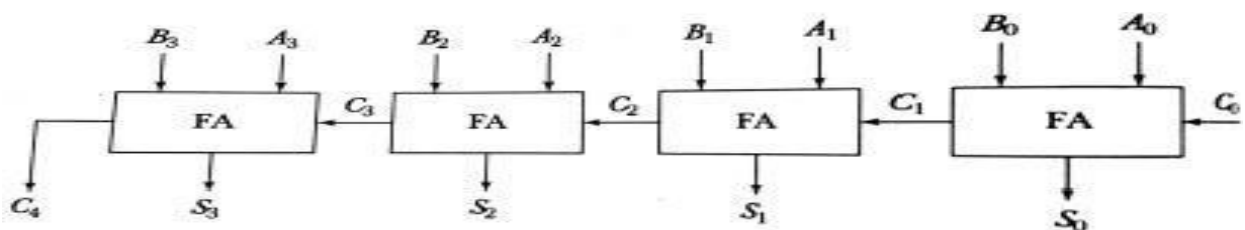
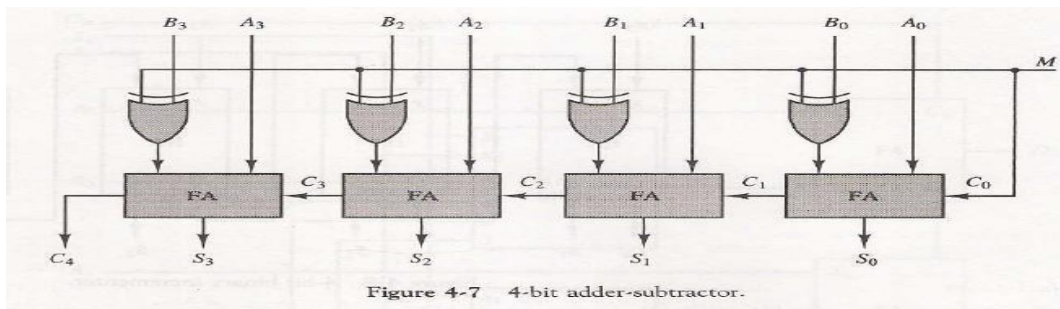


Figure 4-6 4-bit binary adder.

- The augends bits of A and the addend bits of B are designated by subscript numbers from right to left, with subscript 0 denoting the low-order bit.
- The carries are connected in a chain through the full-adders. The input carry to the binary adder is C<sub>0</sub> and the output carry is C<sub>4</sub>. The S outputs of the full-adders generate the required sum bits.
- An n-bit binary adder requires n full-adders.

## Binary Adder – Subtractor:

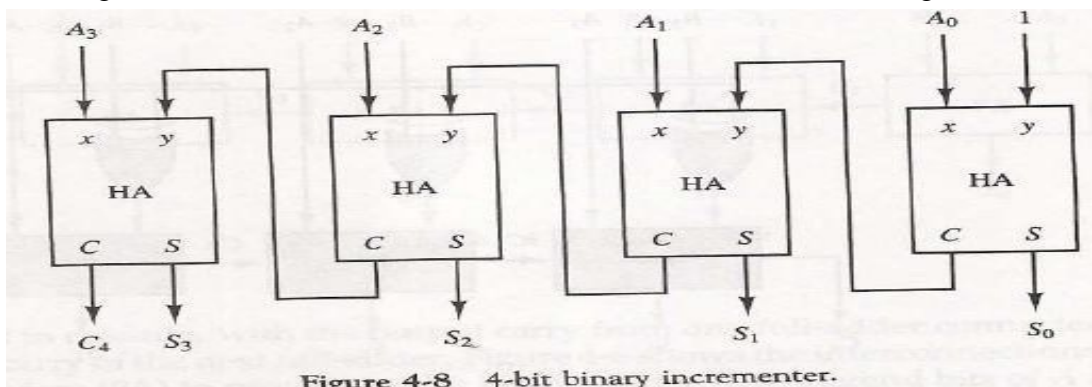
- The addition and subtraction operations can be combined into one common circuit by including an exclusive-OR gate with each full-adder.
- A 4-bit adder-subtractor circuit is shown in Fig. 4-7.



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

## Binary Incrementer:

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

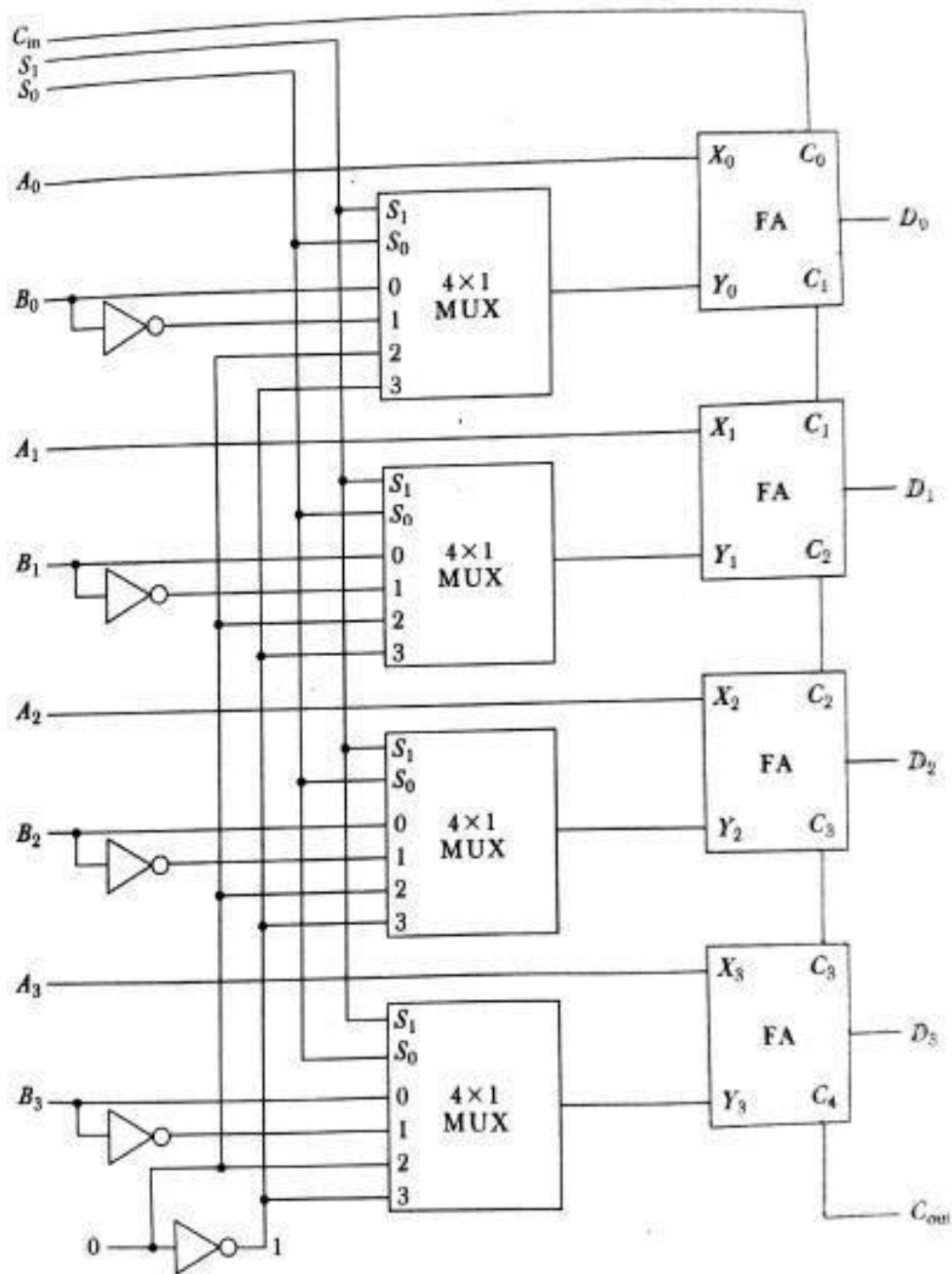


- One of the inputs to the least significant half-adder (HA) is connected to logic-1 and the other input is connected to the least significant bit of the number to be incremented.
- The output carry from one half-adder is connected to one of the inputs of the next-higher-order half-adder.
- The circuit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

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

### Arithmetic Circuit:

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



- There are two 4-bit inputs  $A$  and  $B$  and a 4-bit output  $D$ .
- The four inputs from  $A$  go directly to the  $X$  inputs of the binary adder.
- Each of the four inputs from  $B$  are connected to the data inputs of the multiplexers.
- The multiplexers data inputs also receive the complement of  $B$ .
- The other two data inputs are connected to logic-0 and logic-1.
- The four multiplexers are controlled by two selection inputs  $S_1$  and  $S_0$ . The input carry  $C_{in}$ , goes to the carry input of the FA in the least significant position. The other carries are connected from one stage to the next.
- By controlling the value of  $Y$  with the two selection inputs  $S_1$  and  $S_0$  and making  $C_{in}$  equal to 0 or 1, it is possible to generate the eight arithmetic microoperations listed in Table 4.4.

**TABLE 4.4 Arithmetic Circuit Function Table**

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

**Addition:**

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

**Subtraction:**

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

**Increment:**

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



### Decrement:

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

### Logic Micro-operations:

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

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

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

### List of Logic Microoperations:

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

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

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.

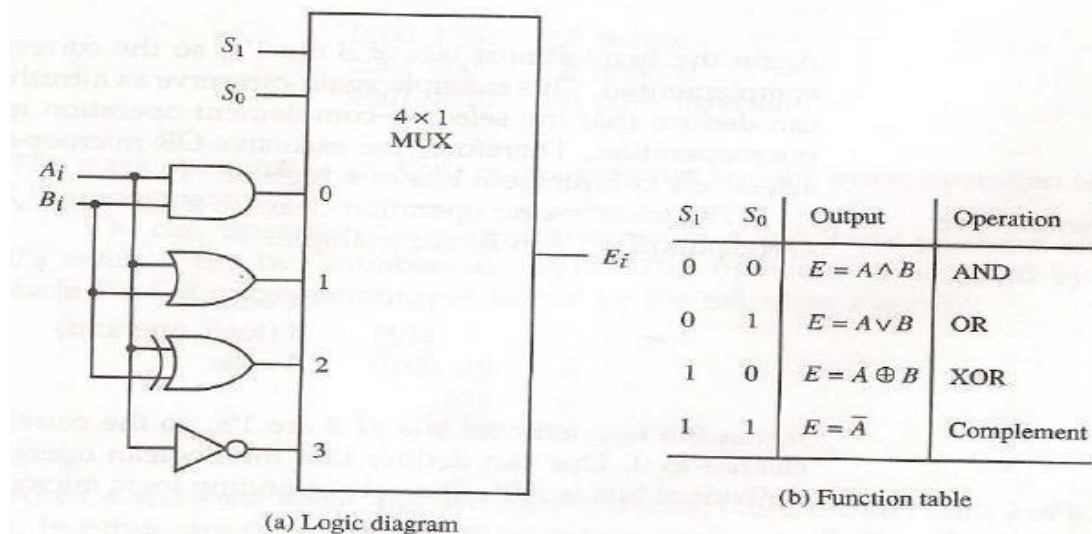
TABLE 4-6 Sixteen Logic Microoperations

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

Hardware Implementation:

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

Figure 4-10 One stage of logic circuit.



## Some Applications:

- Logic micro-operations are very useful for manipulating individual bits or a portion of a word stored in a register.
- They can be used to change bit values, delete a group of bits or insert new bits values into a register.
- The following example shows how the bits of one register (designated by A) are manipulated by logic microoperations as a function of the bits of another register (designated by B).

- Selective set

- ✓ The *selective-set* operation sets to 1 the bits in register A where there are corresponding 1's in register B. It does not affect bit positions that have 0's in B. The following numerical example clarifies this operation:

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

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

- Selective complement

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

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

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

- Selective clear

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

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

- ✓ The corresponding logic microoperation is

$$A \leftarrow A \wedge \bar{B}$$

- Mask

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

0110	1010	A before
<u>0000</u>	<u>1111</u>	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)
<u>0000 1010</u>	A after masking
and then insert the new value:	
0000 1010	A before
1001 0000	B (insert)
<u>1001 1010</u>	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
<u>1010</u>	B
0000	$A \leftarrow A \oplus B$

**Shift Microoperations:**

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

TABLE 4-7 Shift Microoperations

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

➤ **Logical Shift:**

- A *logical* shift is one that transfers 0 through the serial input.
- The symbols *shl* and *shr* for logical shift-left and shift-right microoperations.
- The microoperations that specify a 1-bit shift to the left of the content of register R and a

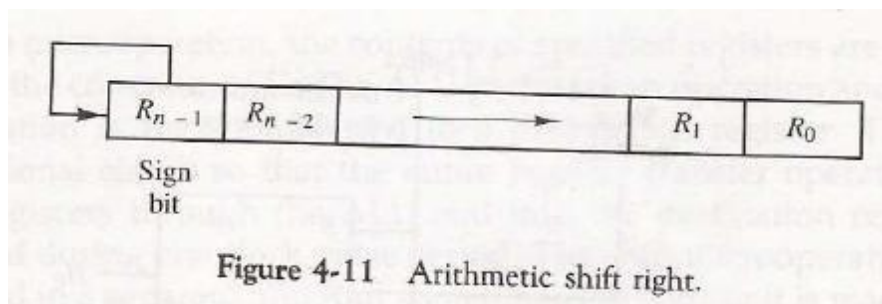
- 1-bit shift to the right of the content of register R shown in table 4.7.
- The bit transferred to the end position through the serial input is assumed to be 0 during a logical shift.

➤ **Circular Shift:**

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

➤ **Arithmetic Shift:**

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



**Hardware Implementation:**

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



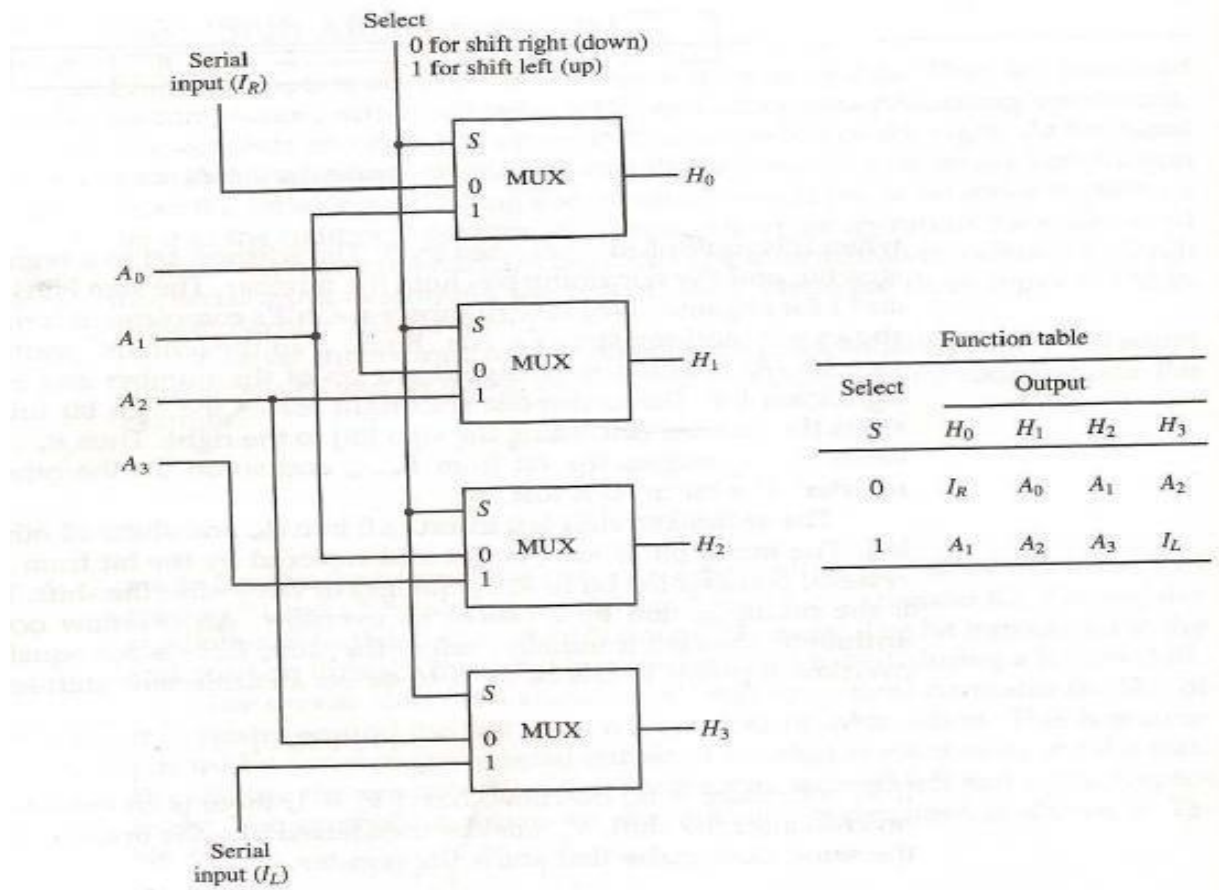
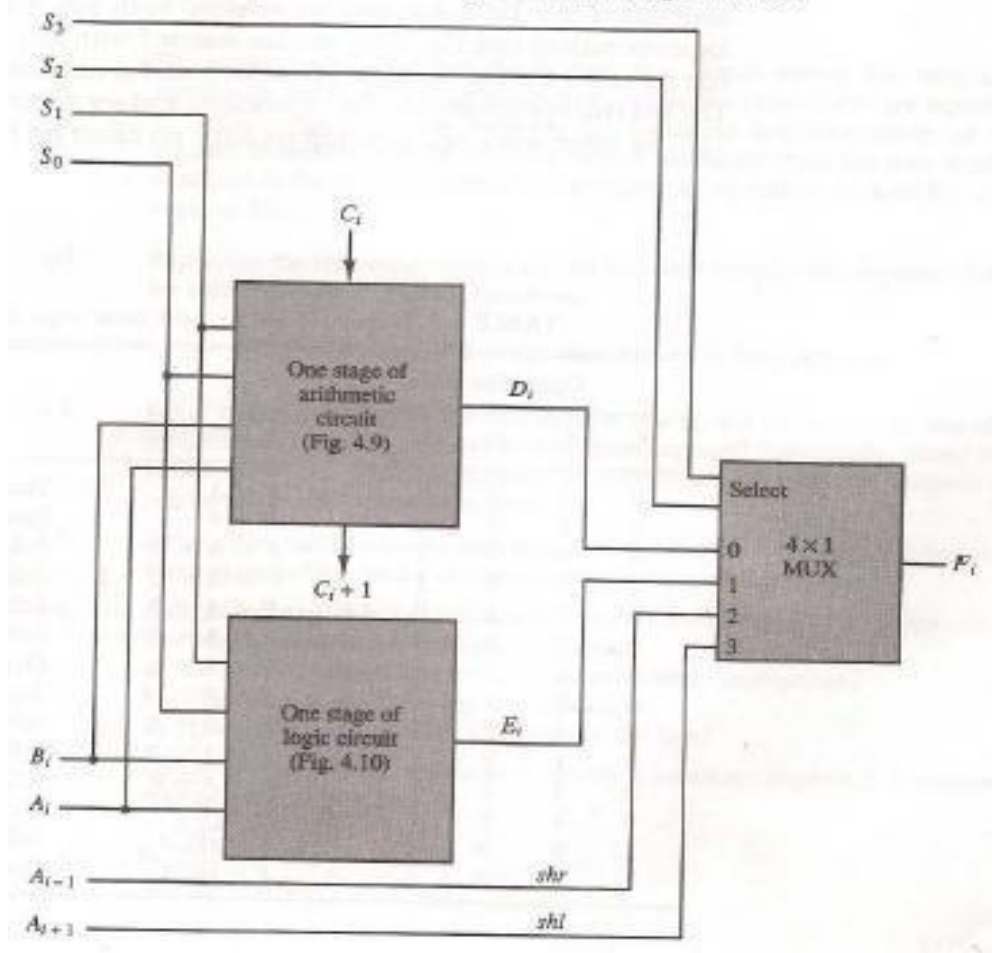


Figure 4-12 4-bit combinational circuit shifter.

### Arithmetic Logic Shift Unit:

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

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



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

TABLE 4-8 Function Table for Arithmetic Logic Shift Unit

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

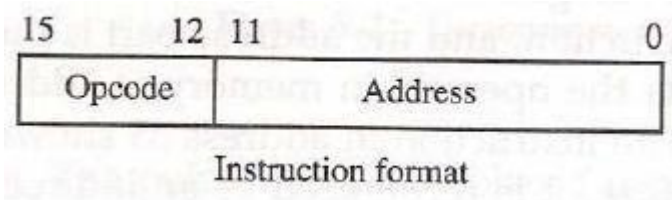
# BASIC COMPUTER ORGANIZATION AND DESIGN

## CONTENTS:

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

## 1. Instruction Codes:

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



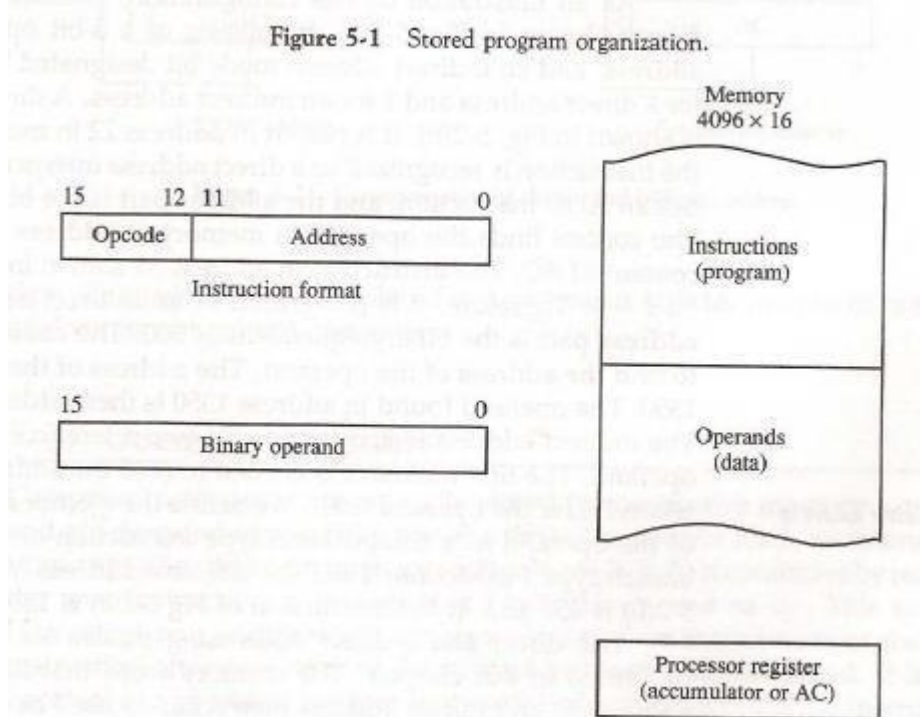
- Operation Code (opcode):
  - ✓ group of bits that define the operation
  - ✓ Eg: add, subtract, multiply, shift, complement.
  - ✓ No. of bits required for opcode depends on no. of operations available in computer.
  - ✓ n bit opcode  $\geq 2^n$  (or less) operations
- Address (operand):
  - ✓ specifies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



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

## Stored Program Organization:

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



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

## Addressing of Operand:

- Sometimes convenient to use the address bits of an instruction code not as an address but as the actual operand.
- When the second part of an instruction code specifies an operand, the instruction is said to have an **immediate operand**.
- When the second part specifies the address of an operand, the instruction is said to have a **direct address**.
- When second part of the instruction designate an address of a memory word in which the address of the operand is found such instruction have **indirect address**.
- One bit of the instruction code can be used to distinguish between a direct and an indirect address.
- The instruction code format shown in Fig. 5-2(a). It consists of a 3-bit operation code, a 12-bit address, and an indirect address mode bit designated by I. The mode bit is 0 for a direct address and 1 for an indirect address.

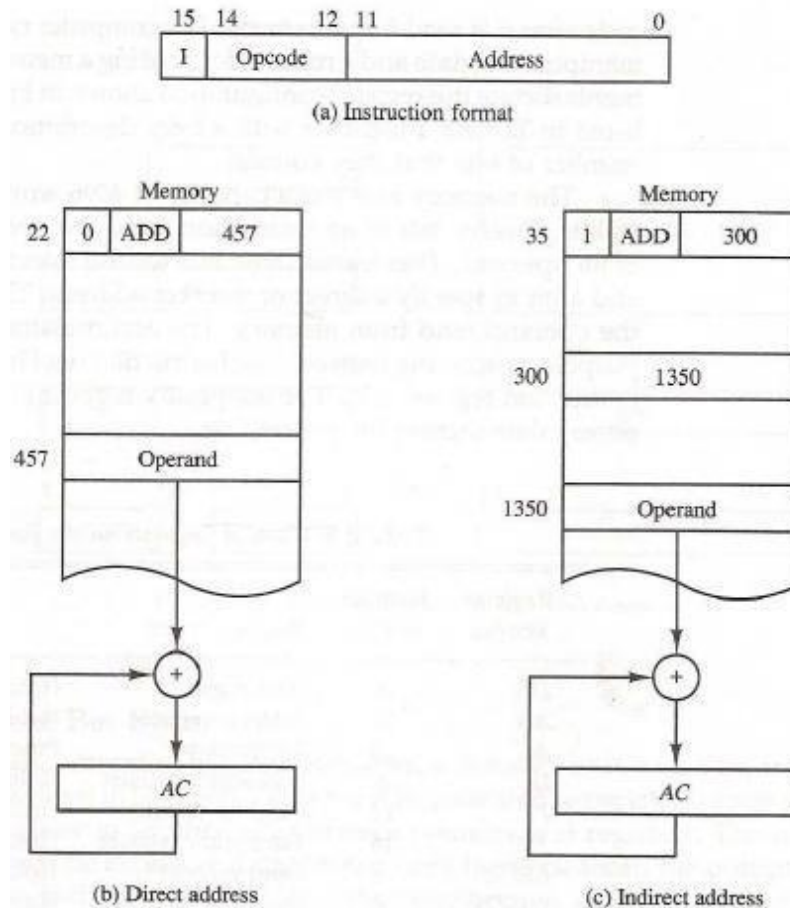


Figure 5-2 Demonstration of direct and indirect address.

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

## 2. Computer Registers:

- What is the need for computer registers?
  - ✓ The need of the registers in computer for
    - Instruction sequencing needs a counter to calculate the address of the next instruction after execution of the current instruction is completed (**PC**).
    - Necessary to provide a register in the control unit for storing the instruction code after it is read from memory (**IR**).
    - Needs processor registers for manipulating data (AC and TR) and a register for holding a memory address (**AR**).
- The above requirements dictate the register configuration shown in Fig. 5-3.

- The registers are also listed in Table 5.1 together with a brief description of their function and the number of bits that they contain.

**TABLE 5-1** List of Registers for the Basic Computer

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

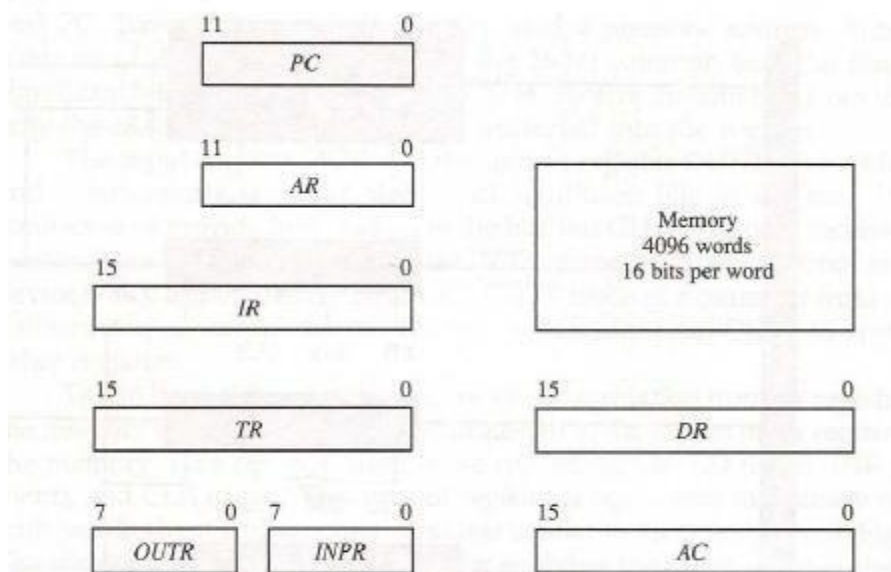


Figure 5-3 Basic computer registers and memory.

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

### Common Bus System:

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

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

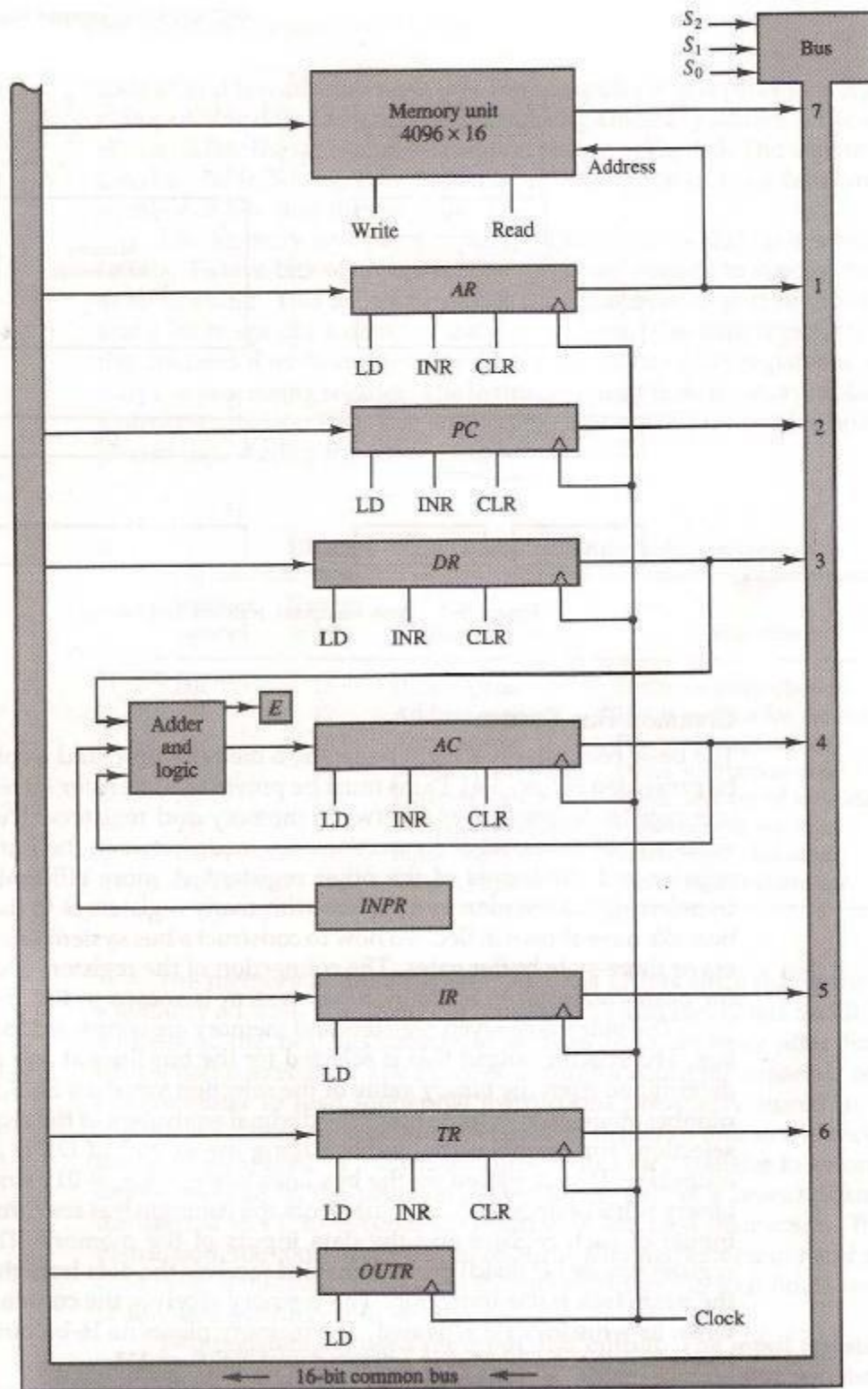


Figure 5-4 Basic computer registers connected to a common bus.

- The lines from the common bus are connected to the inputs of each register and the data inputs of the memory.

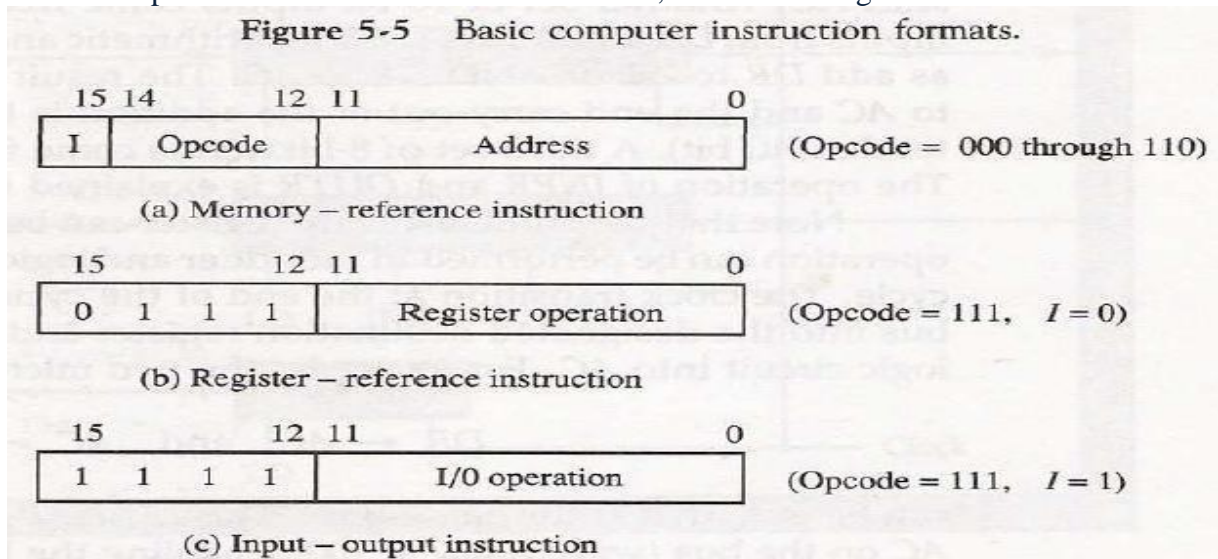


- The particular register whose LD (load) input is enabled receives the data from the bus during the next clock pulse transition.
- The memory receives the contents of the bus when its write input is activated.
- The memory places its 16-bit output onto the bus when the read input is activated and  $S_2S_1S_0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

### 3. Computer Instructions:

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

Figure 5-5 Basic computer instruction formats.



- The operation code (opcode) part of the instruction contains three bits and the meaning of the remaining 13 bits depends on the operation code encountered.
- A memory-reference instruction uses 12 bits to specify an address and one bit to specify the addressing mode *I*.
- *I* is equal to 0 for direct address and to 1 for indirect address.
- The register-reference instructions are recognized by the operation code 1.11 with a 0 in the leftmost bit (bit 15) of the instruction.
- A register-reference instruction specifies an operation on the AC register. So an operand from memory is not needed. Therefore, the other 12 bits are used to specify the operation to be executed.
- An input—output instruction does not need a reference to memory and is recognized by the operation code 111 with a 1 in the leftmost bit of the instruction.
- The remaining 12 bits are used to specify the type of input—output operation.
- The instructions for the computer are listed in Table 5-2.

**TABLE 5-2 Basic Computer Instructions**

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

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

### **Instruction Set Completeness:**

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

### **4. Timing and Control:**

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

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



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

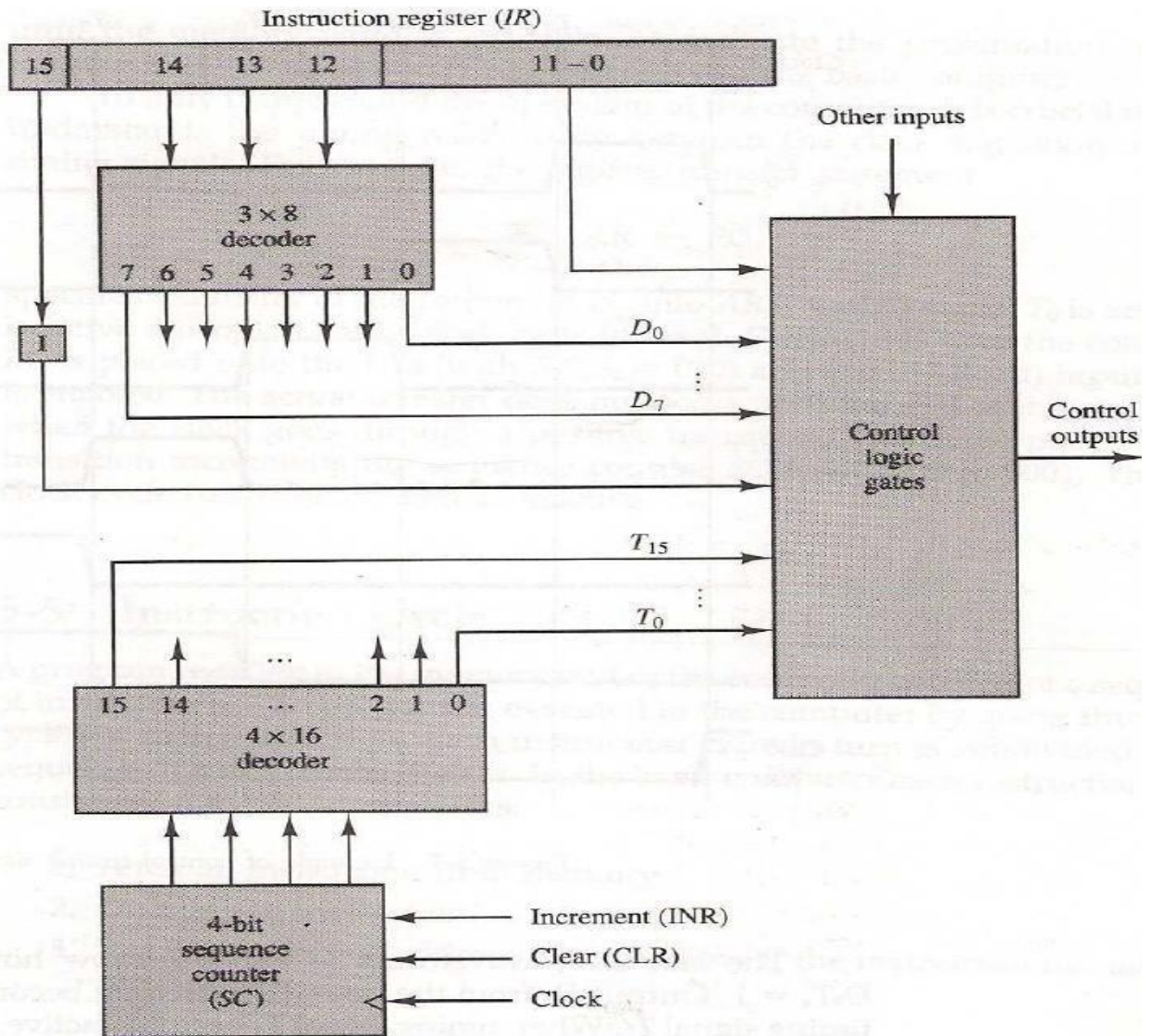


Figure 5-6 Control unit of basic computer.

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

$$D_3T_4: SC \leftarrow 0$$

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



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

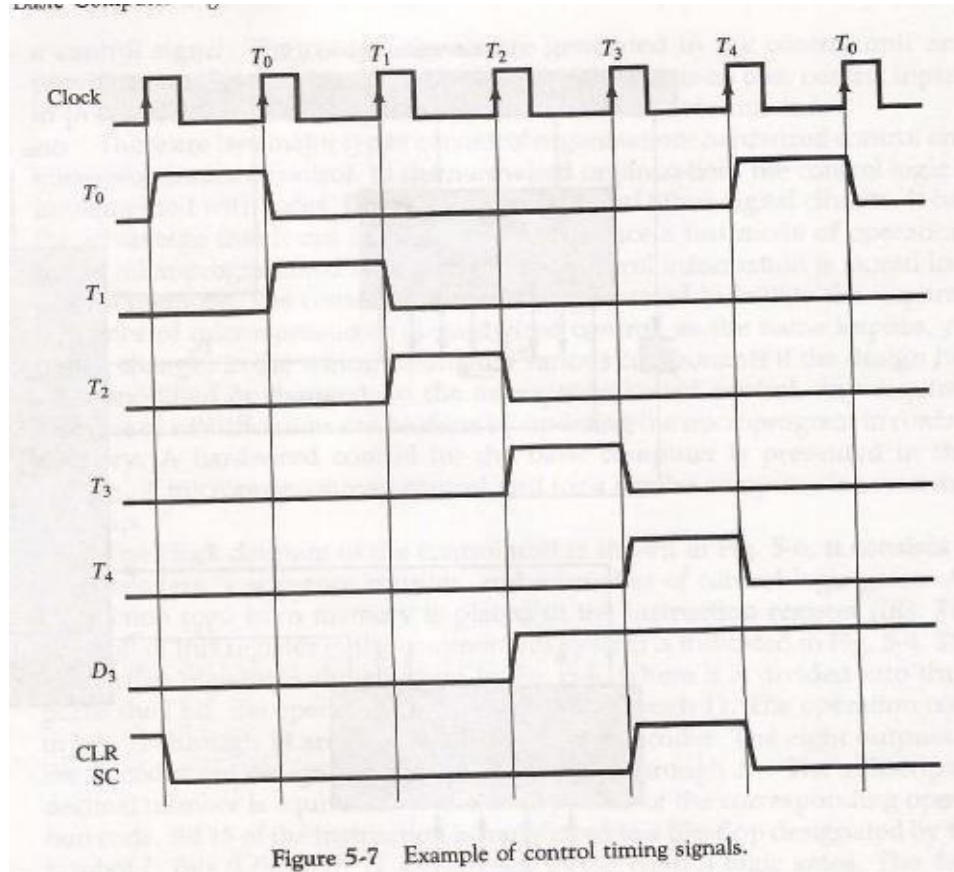


Figure 5-7 Example of control timing signals.

## **5. Instruction Cycle:**

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

## **Fetch and Decode:**

- Initially, the program counter PC is loaded with the address of the first instruction in the program.

- The sequence counter  $SC$  is cleared to 0, providing a decoded timing signal  $T_0$ .
- The microoperations for the fetch and decode phases can be specified by the following register transfer statements.

$T_0: AR \leftarrow PC$   
 $T_1: IR \leftarrow M[AR], PC \leftarrow PC + 1$   
 $T_2: D_0, \dots, D_7 \leftarrow \text{Decode } IR(12-14), AR \leftarrow IR(0-11), I \leftarrow IR(15)$

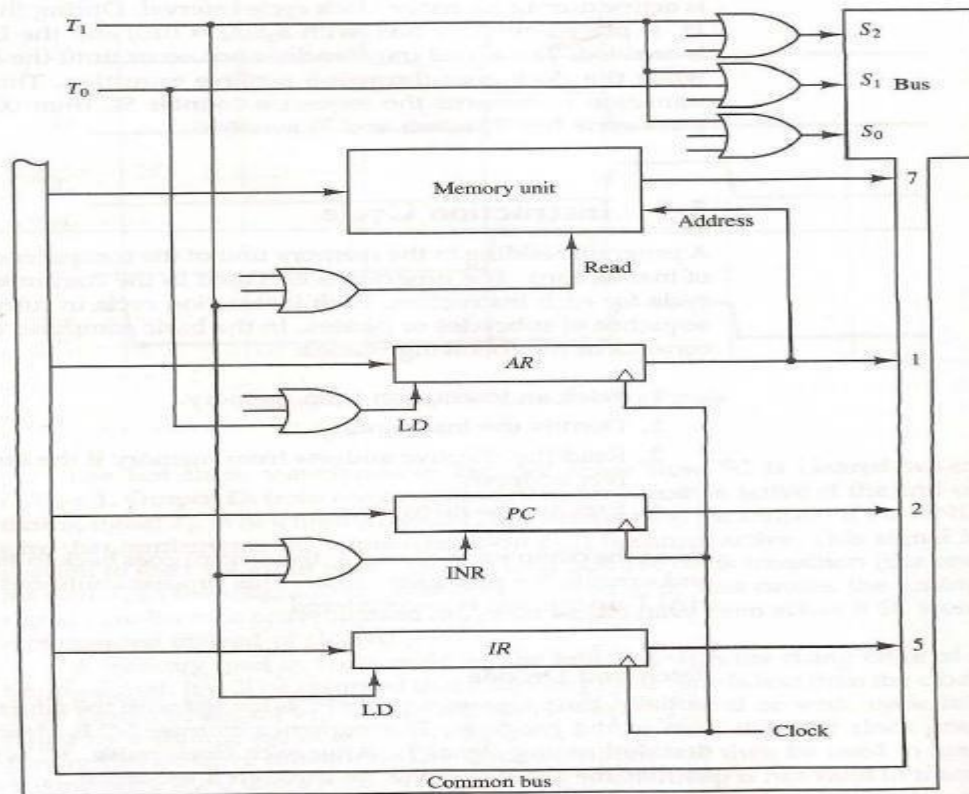


Figure 5-8 Register transfers for the fetch phase.

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

### Determine the Type of Instruction:

- The timing signal that is active after the decoding is  $T_3$ .
- During time  $T_3$ , the control unit determine the type of instruction that was read from the memory.
- The flowchart of fig.5-9 shows the initial configurations for the instruction cycle and also how the control determines the instruction cycle type after the decoding.
- Decoder output  $D_7$  is equal to 1 if the operation code is equal to binary 111.
- If  $D_7=1$ , the instruction must be a register-reference or input-output type.
- If  $D_7 = 0$ , the operation code must be one of the other seven values 000 through 110, specifying a memory-reference instruction.

- Control then inspects the value of the first bit of the instruction, which is now available in flip-flop I.
- If  $D_7 = 0$  and  $I = 1$ , indicates a memory-reference instruction with an indirect address. So it is then necessary to read the effective address from memory.
- If  $D_7 = 0$  and  $I = 0$ , indicates a memory-reference instruction with a direct address.
- If  $D_7 = 1$  and  $I = 0$ , indicates a register-reference instruction.
- If  $D_7 = 0$  and  $I = 1$ , indicates an input-output instruction.
- The three instruction types are subdivided into four separate paths.
- The selected operation is activated with the clock transition associated with timing signal  $T_3$ .
- This can be symbolized as follows:

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

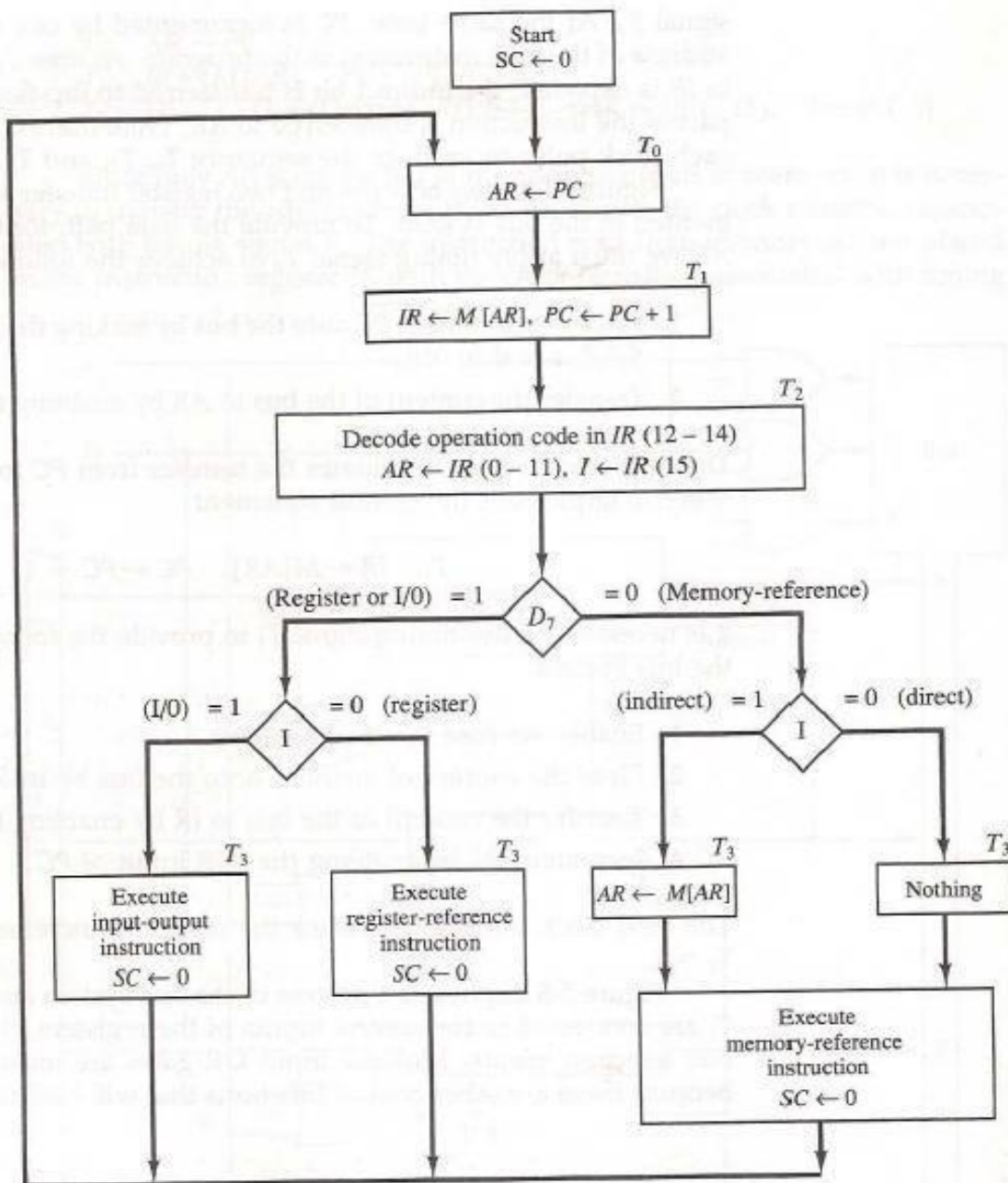


Figure 5-9 Flowchart for instruction cycle (initial configuration).

## Register-Reference Instructions:

- Register-reference instructions are recognized by the control when  $D_7 = 1$  and  $I=0$ .
- These instructions use bits 0 through 11 of the instruction code to specify one of 12 instructions.
- These 12 bits are available in IR (0-11).
- The control functions and microoperations for the register-reference instructions are listed in Table 5-3.
- These instructions are executed with the clock transition associated with timing variable  $T_3$ .
- Control function needs the Boolean relation  $D_7I'T_3$ , which we designate for convenience by the symbol  $r$ .
- By assigning the symbol  $B_i$  to bit  $i$  of  $IR$ , all control functions can be simply denoted by  $rB_i$ .

TABLE 5-3 Execution of Register-Reference Instructions

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

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

## 6. Memory-Reference Instructions:

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



- The symbolic description of each instruction is specified in the table in terms of register transfer notation.

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

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

### **AND to AC:**

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

$$D_0T_4: DR \leftarrow M[AR]$$

$$D_0T_5: AC \leftarrow AC \wedge DR, SC \leftarrow 0$$

### **ADD to AC:**

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

$$D_1T_4: DR \leftarrow M[AR]$$

$$D_1T_5: AC \leftarrow AC + DR, E \leftarrow C_{out}, SC \leftarrow 0$$

### **LDA:** Load to AC

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

$$D_2T_4: DR \leftarrow M[AR]$$

$$D_2T_5: AC \leftarrow DR, SC \leftarrow 0$$

### **STA:** Store AC

- This instruction stores the content of AC into the memory word specified by the effective address.

- Since the output of AC is applied to the bus and the data input of memory is connected to the bus, we can execute this instruction with one microoperation.

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

### **BUN:** Branch Unconditionally

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

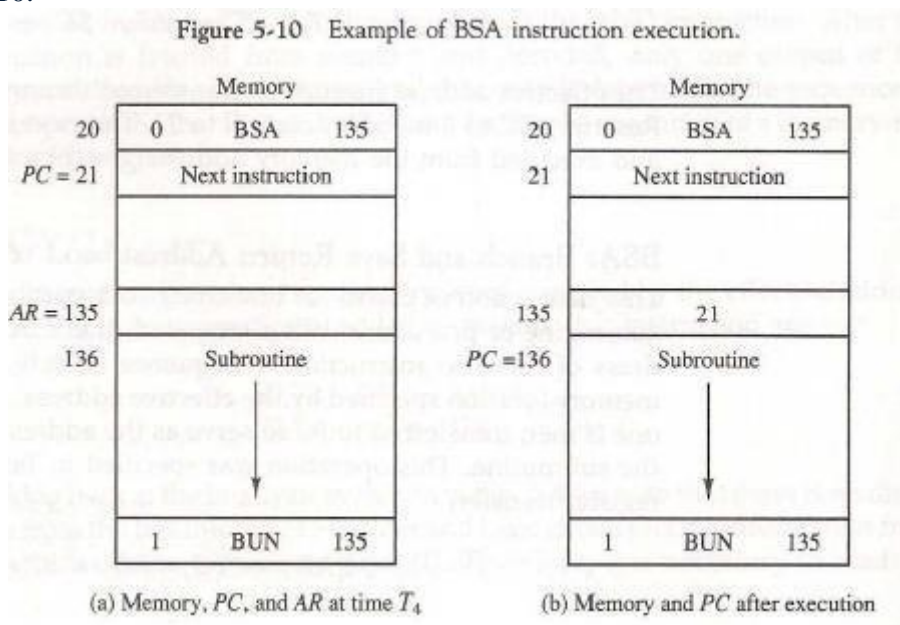
$$D_4T_4: PC \leftarrow AR, SC \leftarrow 0$$

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

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

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

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



- The BSA instruction is assumed to be in memory at address 20.
- The I bit is 0 and the address part of the instruction has the binary equivalent of 135.
- After the fetch and decode phases, PC contains 21, which is the address of the next instruction in the program (referred to as the return address). AR holds the effective address 135.
- This is shown in part (a) of the figure.
- The BSA instruction performs the following numerical operation:
 
$$M[135] \leftarrow 21, PC \leftarrow 135 + 1 = 136$$
- The result of this operation is shown in part (b) of the figure.
- The return address 21 is stored in memory location 135 and control continues with the subroutine program starting from address 136.
- The return to the original program (at address 21) is accomplished by means of an indirect BUN instruction placed at the end of the subroutine.

- When this instruction is executed, control goes to the indirect phase to read the effective address at location 135, where it finds the previously saved address 21.
- When the BUN instruction is executed, the effective address 21 is transferred to PC.
- The next instruction cycle finds *PC* with the value 21, so control continues to execute the instruction at the return address.
- The BSA instruction must be executed with a sequence of two microoperations:

$$D_5T_4: M[AR] \leftarrow PC, AR \leftarrow AR + 1$$

$$D_5T_5: PC \leftarrow AR, SC \leftarrow 0$$

### ISZ: Increment and Skip if Zero

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

$$D_6T_4: DR \leftarrow M[AR]$$

$$D_6T_5: DR \leftarrow DR + 1$$

$$D_6T_6: M[AR] \leftarrow DR, \text{ if } (DR = 0) \text{ then } (PC \leftarrow PC + 1), SC \leftarrow 0$$

### Control Flowchart:

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

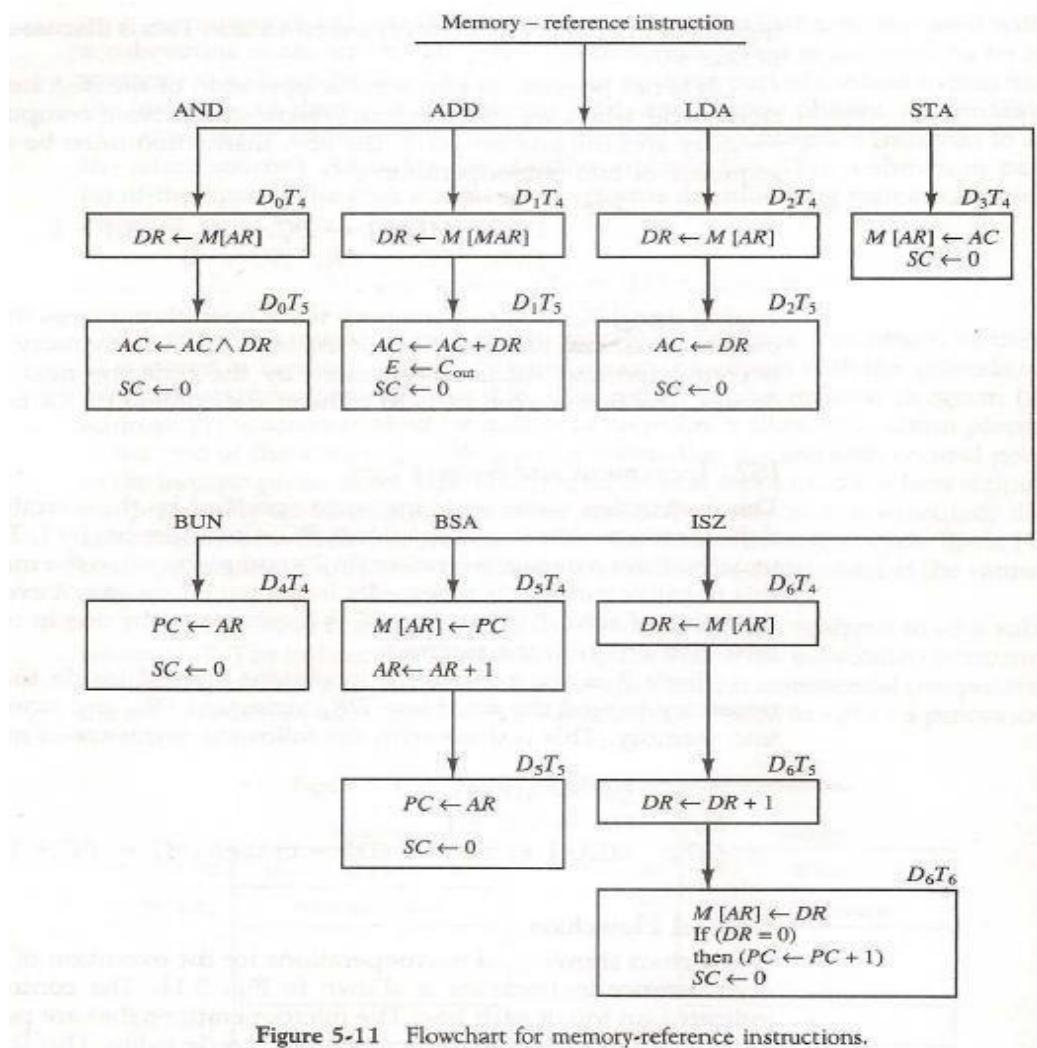


Figure 5-11 Flowchart for memory-reference instructions.

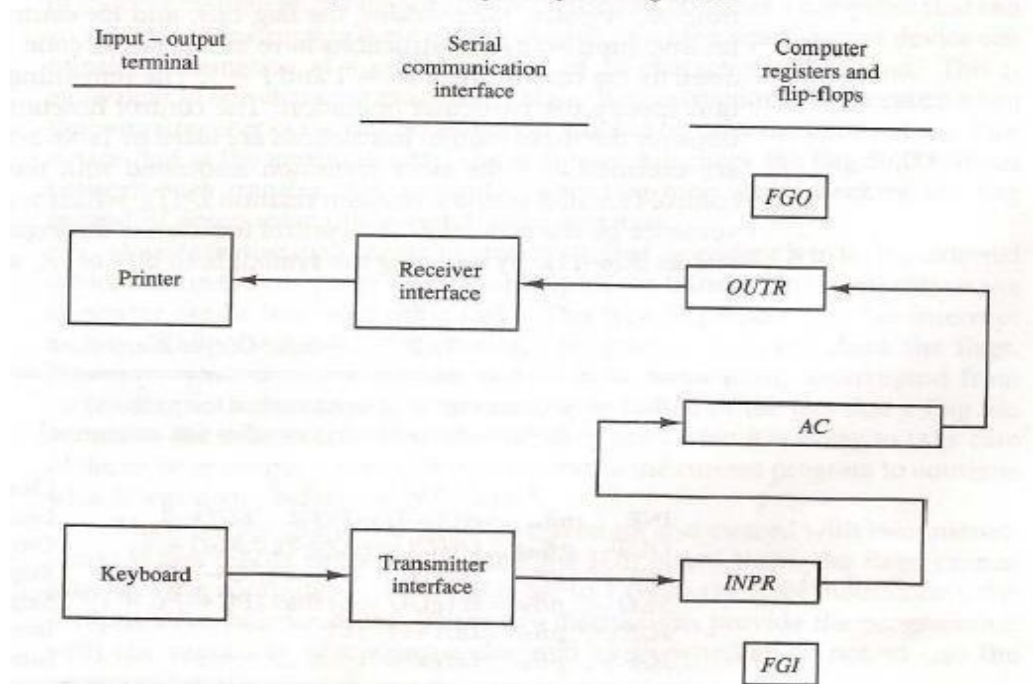
## 7. Input-Output and Interrupt:

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

### **Input-Output Configuration:**

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

Figure 5-12 Input-output configuration.



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

### **Input-Output Instructions:**

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



- The control functions and microoperations for the input-output instructions are listed in Table 5-5.

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

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

### Program Interrupt:

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

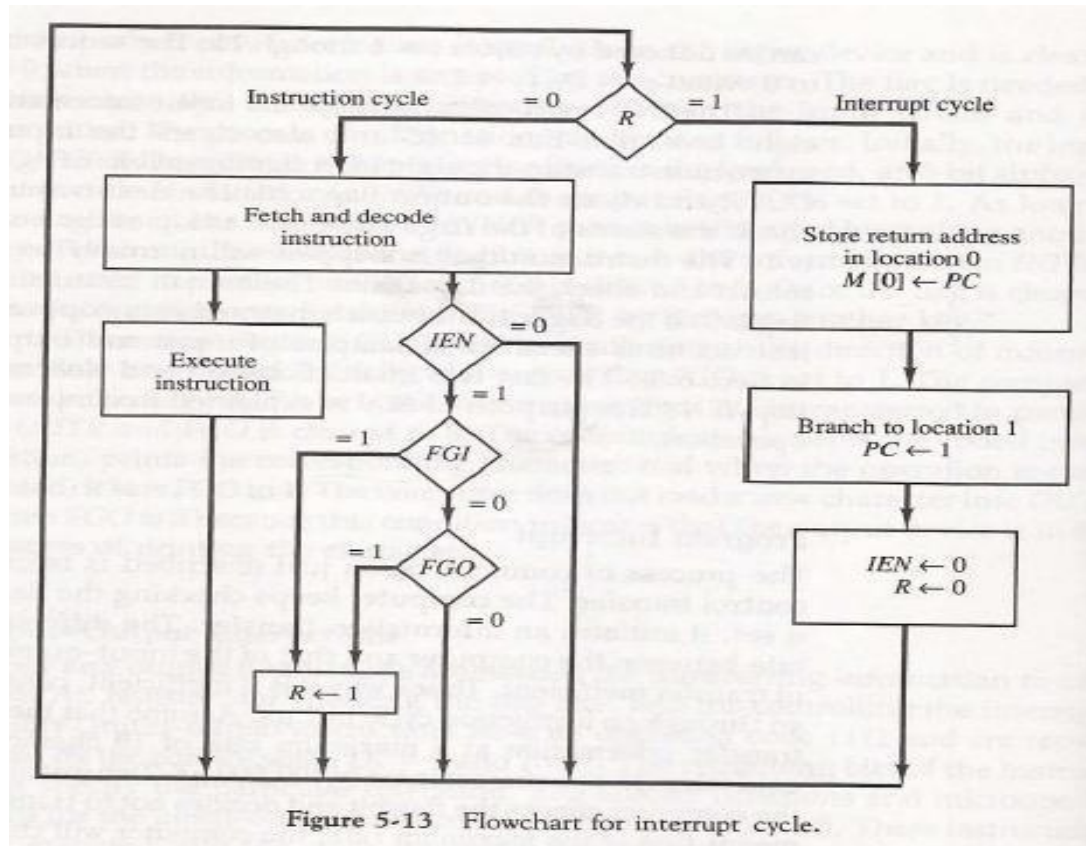
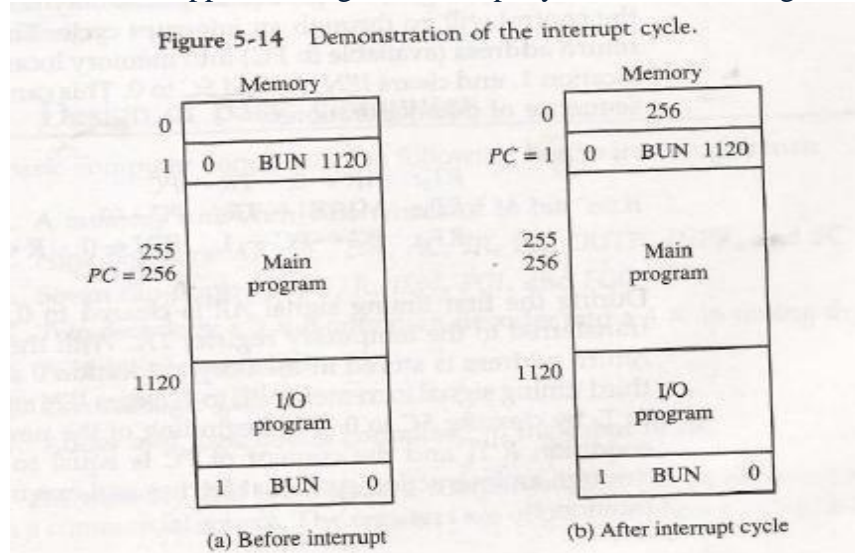


Figure 5-13 Flowchart for interrupt cycle.

### Interrupt cycle:

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



- When an interrupt occurs and R is set to 1 while the control is executing the instruction at address 255.
- At this time, the return address 256 is in PC.
- The programmer has previously placed an input—output service program in memory starting from address 1120 and a BUN 1120 instruction at address 1. This is shown in Fig. 5.14(a).
- When control reaches timing signal T<sub>0</sub> and finds that R = 1, it proceeds with the interrupt cycle.
- The content of PC (256) is stored in memory location 0, PC is set to 1, and R is cleared to 0.
- The branch instruction at address 1 causes the program to transfer to the input—output service program at address 1120.

- This program checks the flags, determines which flag is set, and then transfers the required input or output information.
- Once this is done, the instruction ION is executed to set IEN to 1 (to enable further interrupts), and the program returns to the location where it was interrupted.
- This is shown in Fig. 5-14(b).

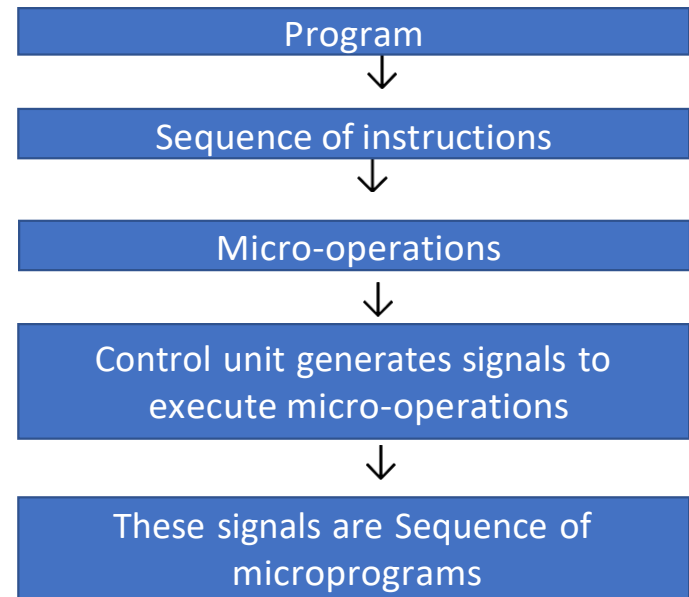
## UNIT-2

# Microprogrammed Control

- Control memory
- Address Sequencing
- Microprogram Example
- Design of Control Unit

- What is a microprogram?
- A sequence of micro instructions is a microprogram.

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



- **Hardwired Control:**

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

- instructions **Microprogrammed Control:**

- Control information is stored in control memory.
- Control memory is programmed to initiate the required sequence of micro-operations. The
- key characteristics are
  - Speed of operation is low when compared with hardwired
  - Less complex
  - Less expensive
  - Flexibility to add new instructions

## Control Memory

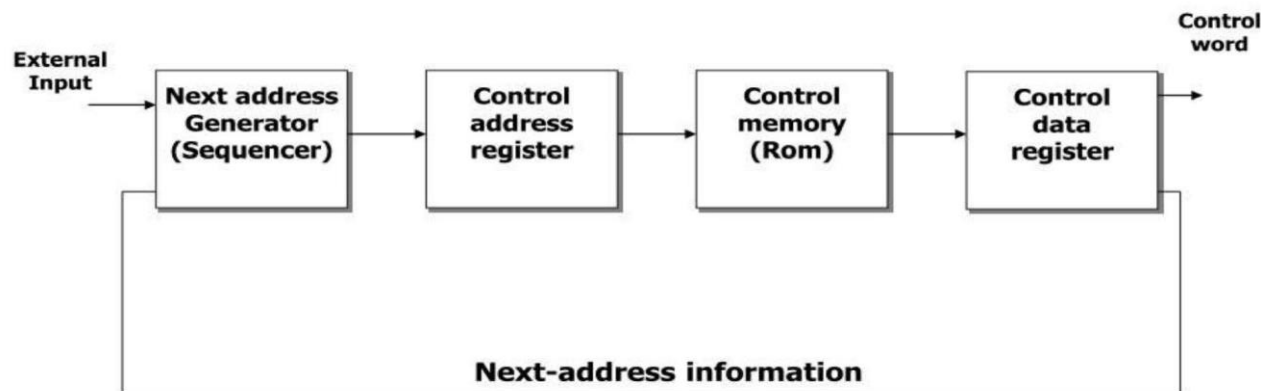
- The control function that specifies a microoperation is called as control variable.
- When control variable is in one binary state, the corresponding microoperation is executed. For
- the other binary state the state of registers does not change.
- The active state of a control variable may be either 1 state or the 0 state, depending on the application.
- Example;
- For bus-organized systems the control signals that specify microoperations are groups of bits that select the paths in multiplexers, decoders, and arithmetic logic units.

### Control Word:

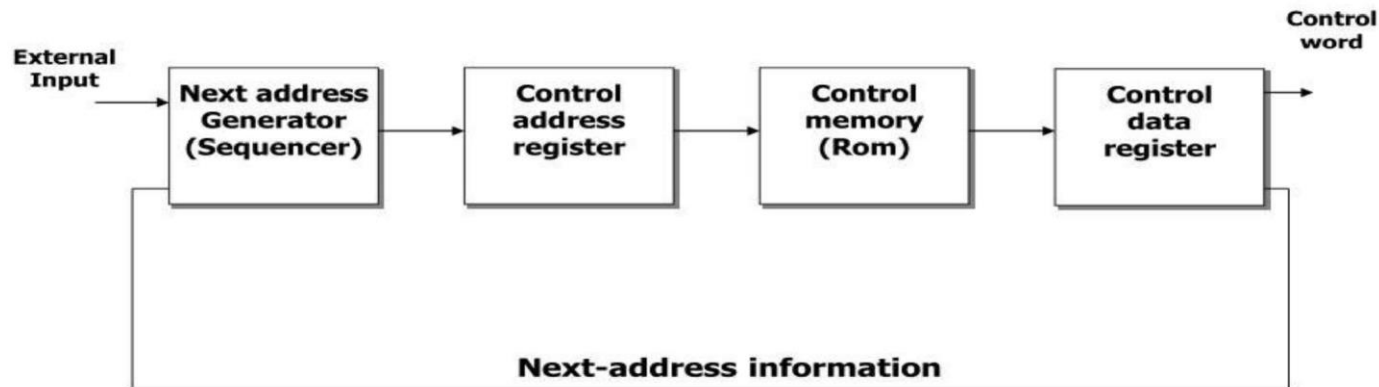
- The control variables at any given time can be represented by a string of 1's and 0's called a control word.
- All control words can be programmed to perform various operations on the components of the system.

- **Microprogram control unit:**

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



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





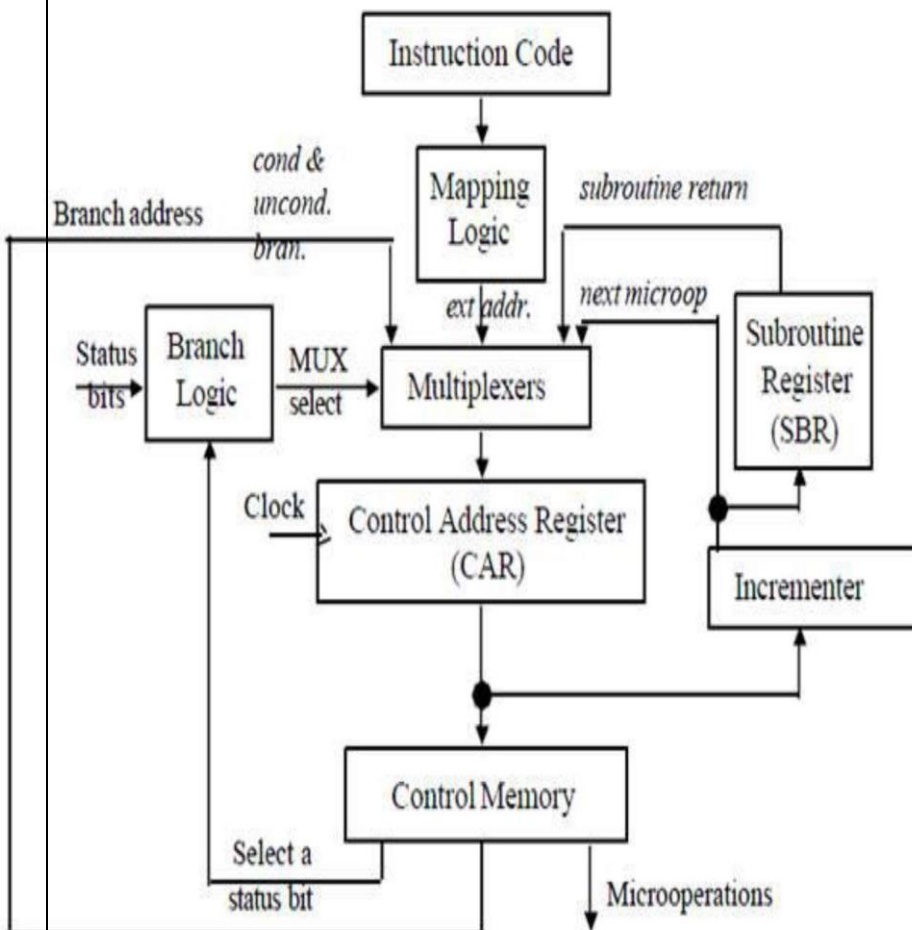
- A computer with a microprogrammed control unit will have two separate memories:
  - main memory (RAM)
  - control memory (ROM)
- The microprogram consists of microinstructions that specify various internal control signals for execution of register microoperations
- These microinstructions generate 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

## Addressing sequence

- Microinstructions are stored in control memory in groups, with each group specifying a routine.
- Each computer instruction has its own microprogram routine to generate the microoperations.
- The hardware that controls the address sequencing of the control memory must be capable of sequencing the microinstructions within a routine and be able to branch from one routine to another
- Steps the control must undergo during the execution of a single computer instruction are as follows Initial
- address is loaded into the control address register(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.
- At the end of the fetch routine instruction is placed in the instruction register- IR
- The control memory then goes through the routine to determine the effective address of the operand with the help of mode bits and branch micro instructions
- At the end of this routine Address register AR holds operand address
- The next step is to generate the microoperations that execute the instruction fetched from memory by considering the opcode and applying a mapping process.
- **The transformation of the instruction code bits to an address in control memory where the routine of instruction located is referred to as mapping process.**
- After execution, control must return to the fetch routine by executing an unconditional branch

- In brief the address sequencing capabilities required in a 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.

In the figure four different paths form which the control address register (CAR) receives the address.

✓ The incrementer increments the content of the control register address register by one, to select the next microinstruction in sequence.

✓ Branching is achieved by specifying the branch address in one of the fields of the microinstruction.

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

✓ An external address is transferred into control memory via a mapping logic circuit.

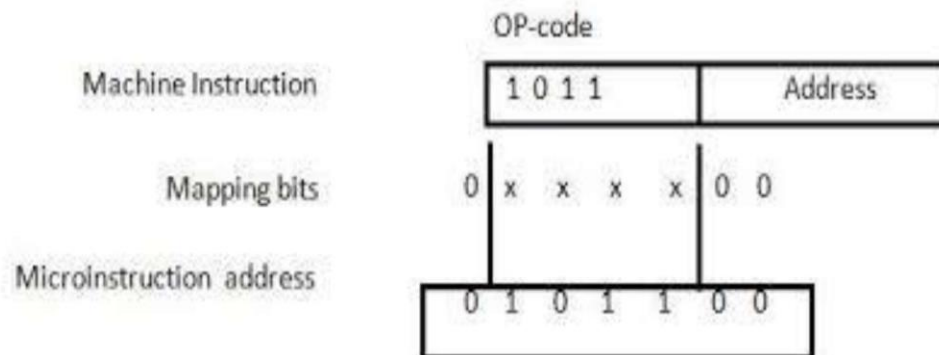
✓ The return address for a subroutine is stored in a special register, that value is used when the micropogram wishes to return from the subroutine.

## Conditional branching

- Conditional branching is obtained by using part of the microinstruction to select a specific status bit in order to determine its condition.
- The status conditions are special bits in the system that provide parameter information such as the carry-out of an adder, the sign bit of a number, the mode bits of an instruction, and i/o status conditions.
- The status bits, together with the field in the microinstruction that specifies a branch address, control the branch logic.
- The branch logic tests the condition, if met then branches, otherwise, increments the CAR.
- Conditional branching can be implemented with a multiplexer. If there are 8 status bit conditions, then 3 bits in the microinstruction are used to specify any one of the condition and they provide the selection variables for the multiplexer.
- If the selected status bit is in 1 state, the output of multiplexer is 1, otherwise it is 0.
- A 1 output in the multiplexer generates a control signal to transfer the branch address from the microinstruction into the control address register.
- A 0 output in the multiplexer causes the address register to be incremented. For
- unconditional branching, fix the value of one status bit to be 1.
- Reference to this bit causes the branch address to be loaded into the control address register unconditionally.

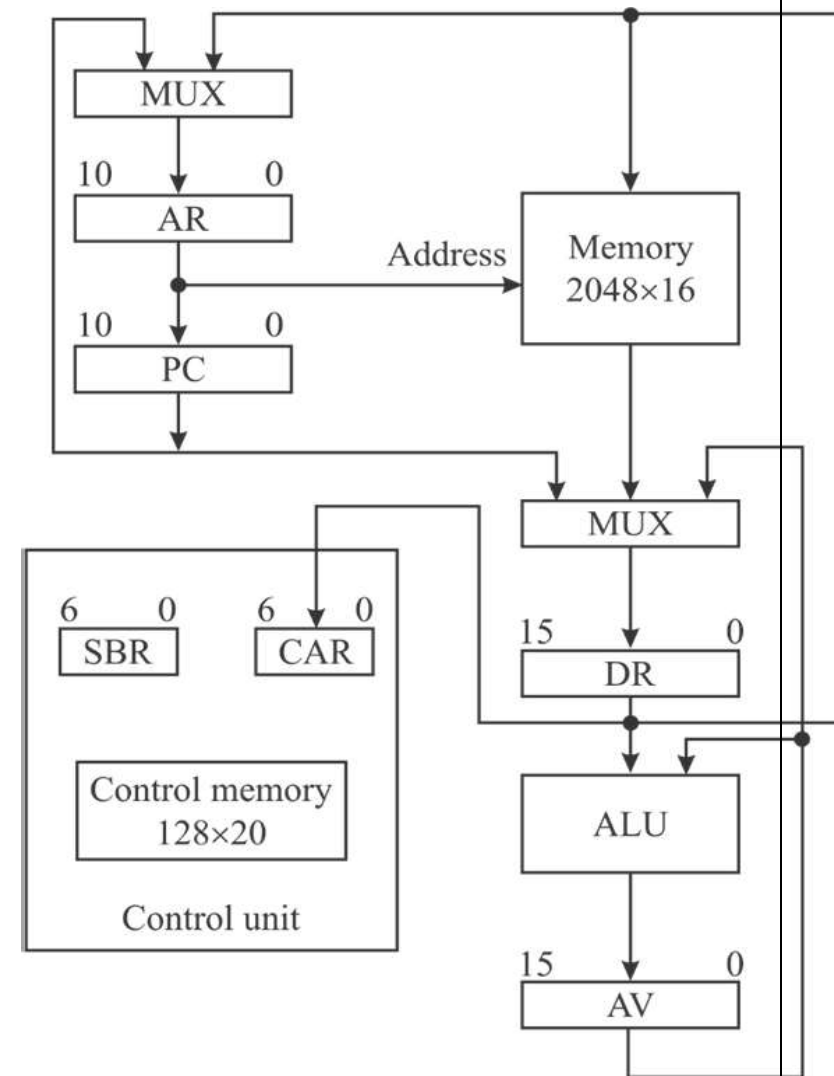
## Mapping of instructions

- A special type of branch exists when a microinstruction specifies a branch to the first word in control memory where a microprogram routine is located.
- The status bits for this type of branch are the bits in the opcode.
- Assume an opcode of four bits and a control memory of 128 locations. The mapping process converts the 4-bit opcode to a 7-bit address for control memory shown in below figure.
- Mapping consists of placing a 0 in the most significant bit of the address, transferring the four operation code bits, and clearing the two least significant bits of the control address register.
- This provides for each computer instruction a microprogram routine with a capacity of four microinstructions.
- The mapping function is implemented by integrated circuit called programmable logic device



## Microprogram example

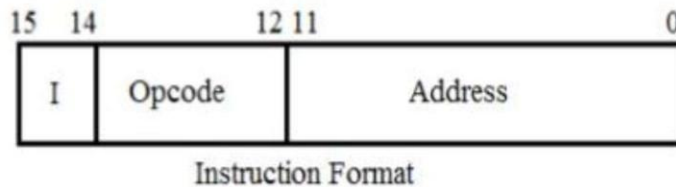
- The process of code generation for the control memory is called microprogramming.
- The block diagram of the computer configuration is shown in the figure. **Two memory units:**
  1. Main memory – stores instructions and data
  2. Control memory – stores microprogram
- **Four processor registers**
  1. Program counter – PC
  2. Address register – AR
  3. Data register – DR
  4. Accumulator register - AC
- **Two control unit registers**
  1. Control address register – CAR
  2. Subroutine register – SBR
- Transfer of information among registers in the processor is through MUXs rather than a bus.
- DR receives information from AC, PC or memory.
- AR can receive information from PC or DR



# Computer instruction format

## Instruction format

- An **Instruction format** is a binary format which specifies a computer instruction
- It specifies the **address** of the operand, the **Opcode**, the **addressing mode** of the instruction



- Three fields for an instruction:
  1. 1-bit field for direct/indirect addressing
  2. 4-bit opcode
  3. 11-bit address field

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

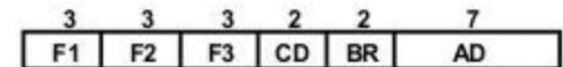


## Micro instruction format

The microinstruction format is composed of 20 bits divided into four parts

- 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] because control memory has 128 words
- Each of the three microoperation fields can specify one of seven possibilities.
- No more than three microoperations can be chosen for a microinstruction.
- If fewer than three are needed, the code 000 = NOP.
- The three bits in each field are encoded to specify seven distinct microoperations listed in below table.
- The condition field (CD) is two bits to specify four status bit conditions .
- The branch field (BR) consists of two bits and is used with the address field to choose the address of the next microinstruction.

### Microinstruction Format



F1, F2, F3: Microoperation fields  
CD: Condition for branching  
BR: Branch field  
AD: Address field

## Symbols and Binary Code for Microinstruction Fields

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

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

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

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

## Symbolic microinstructions

- Different symbols can be used to construct the micro instructions in symbolic form.
- Each line of an assembly language microprogram defines a symbolic microinstruction and is divided into five parts

➤ Lable

➤ Microoperations

➤ CD

➤

BR ➤

AD

Label	Microops	CD	BR	AD
	ORG 0			
ADD:	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ADD	U	JMP	FETCH

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 field. If NOP, then translated to 9 zeros
3. The condition field specifies one of the four conditions U,I,S,Z.
4. The branch field has one of the four branch symbols JMP,CALL,RET,MAP
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 origin i;e the first address of a microprogram routine.
- Eg; ORG 64- places first microinstruction at control memory 1000000 which is equivalent to decimal number 64.

## 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.
- the fetch routine starts at address 64.
- The fetch routine requires the following three microinstructions at locations 64-66.
- The microinstructions needed for fetch routine are:
- The address of instruction is transferred from PC to AR and the instruction is read from memory into DR and PC is incremented.
- The address part is transferred to AR and the control is transferred to one of 16 routines by mapping the operation code part of the instruction from DR into CAR.
- Using assembly language conventions like above we can write symbolic micro programs as shown in the table.

### Microinstructions for fetch routine:

$AR \leftarrow PC$ $DR \leftarrow M[AR], PC \leftarrow PC + 1$ $AR \leftarrow DR(0-10), CAR(2-5) \leftarrow DR(11-14), CAR(0,1,6) \leftarrow 0$
---

### Symbolic microprogram for fetch routine:

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

### Binary micropogram for fetch routine:

Binary address	F1	F2	F3	CD	BR	AD
100000	110	000	000	00	00	1000001
100001	000	100	101	00	00	1000010
100010	101	000	000	00	11	0000000

# Symbolic Microprogram

- Control memory: 128 20-bit words
- First 64 words: Routines for 16 machine instructions
- Last 64 words: Used for other purpose (e.g., fetch routine and other subroutines)
- Mapping: OP-code XXXX into 0XXXX00, first address for 16 routines are 0(0 0000 00), 4(0 0001 00), 8, 12, 16, 20, ..., 60

## Partial Symbolic Microprogram

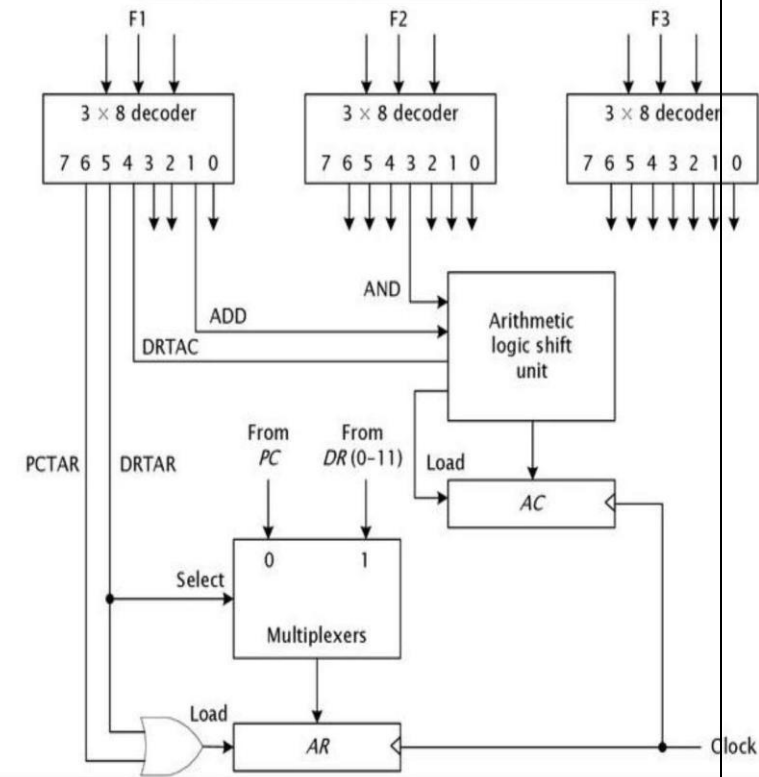
Label	Microops	CD	BR	AD
ADD:	ORG 0			
	NOP		I	CALL INDRCT
	READ		U	JMP NEXT
	ADD		U	JMP FETCH
BRANCH:	ORG 4			
	NOP		S	JMP OVER
	NOP		U	JMP FETCH
OVER:	NOP		I	CALL INDRCT
	ARTPC		U	JMP FETCH
STORE:	ORG 8			
	NOP		I	CALL INDRCT
	ACTDR		U	JMP NEXT
	WRITE		U	JMP FETCH
EXCHANGE:	ORG 12			
	NOP		I	CALL INDRCT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

## Design of control unit

- The control memory out of each subfield must be decoded to provide the distinct microoperations.
- The outputs of the decoders are connected to the appropriate inputs in the processor unit.
- The figure shows the three decoders and some of the connections that must be made from their outputs.
- The three fields of the microinstruction in the output of control memory are decoded with a 3x8 decoder to provide eight outputs.
- Each of the output must be connected to proper circuit to initiate the corresponding microoperation.
- When  $F1 = 101$  (binary 5), the next pulse transition transfers the content of DR (0-10) to AR.
- Similarly, when  $F1 = 110$  (binary 6) there is a transfer from PC to AR (symbolized by PCTAR).
- As shown in Fig, outputs 5 and 6 of decoder F1 are connected to the load input of AR so that when either one of these outputs is active, information from the multiplexers is transferred to AR.
- The multiplexers select the information from DR when output 5 is active and from PC when output 6 is inactive.
- The transfer into AR occurs with a clock transition only when output 5 or output 6 of the decoder is active.
- For the arithmetic logic shift unit the control signals are instead of coming from the logical gates, now these inputs will now come from the outputs of AND, ADD and DRTAC respectively.

## Decoding of $F$ fields

Fig. 7-7 Decoding of microoperation fields

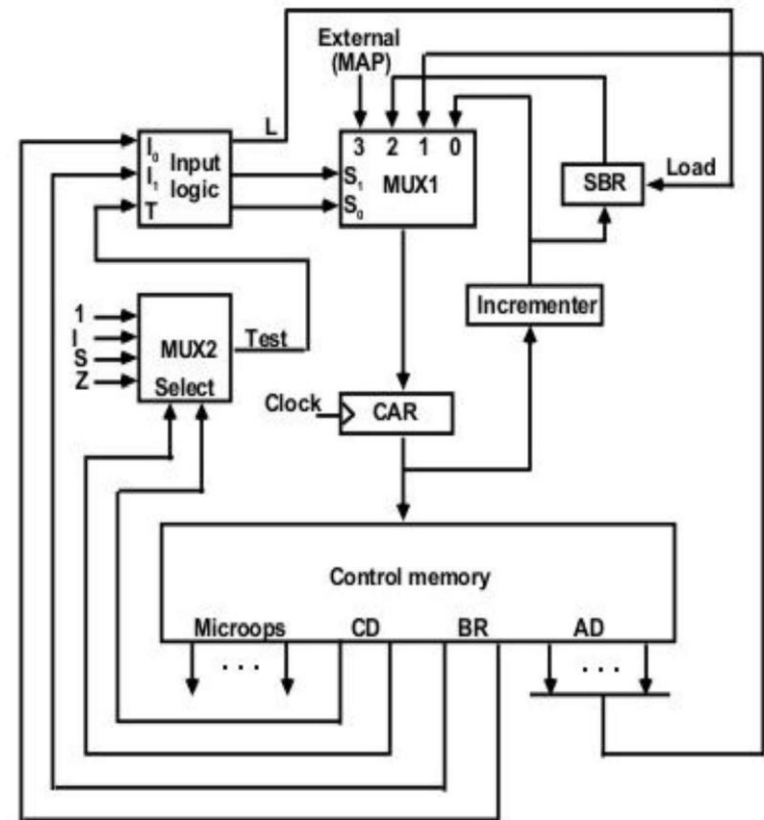




## Microprogram sequencer

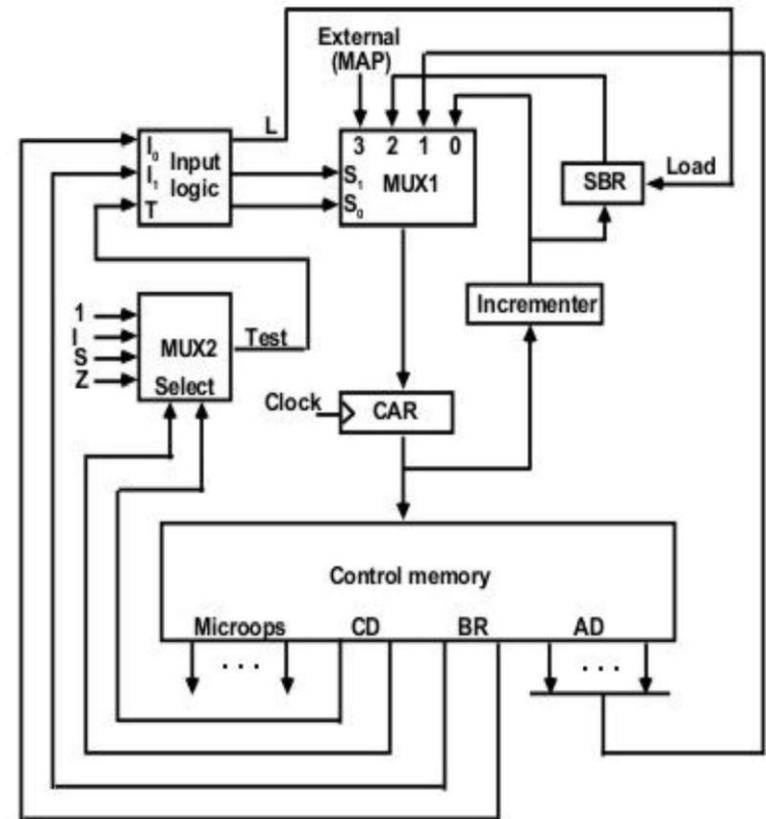
- The basic components of a microprogrammed control unit are the control memory and the circuits that select the next address.
- The address selection is called a microprogram sequencer.
- The purpose of a microprogram sequencer is to present an address to the control memory so that a microinstruction may be read and executed.
- The next-address logic of the sequencer determines the specific address to be loaded into the control address register.
- The block diagram of the microprogram sequencer is shown in the figure.
- The control memory is included in the diagram to show the interaction between the sequencer and the memory attached to it.
- There are two multiplexers in the circuit.
  1. The first multiplexer selects an address from one of four sources and routes it into control address register CAR.
  2. The second multiplexer tests the value of a selected status bit and the result of the test is applied to an input logic circuit.
- The output from CAR provides the address for the control memory.

## Microprogram Sequencer

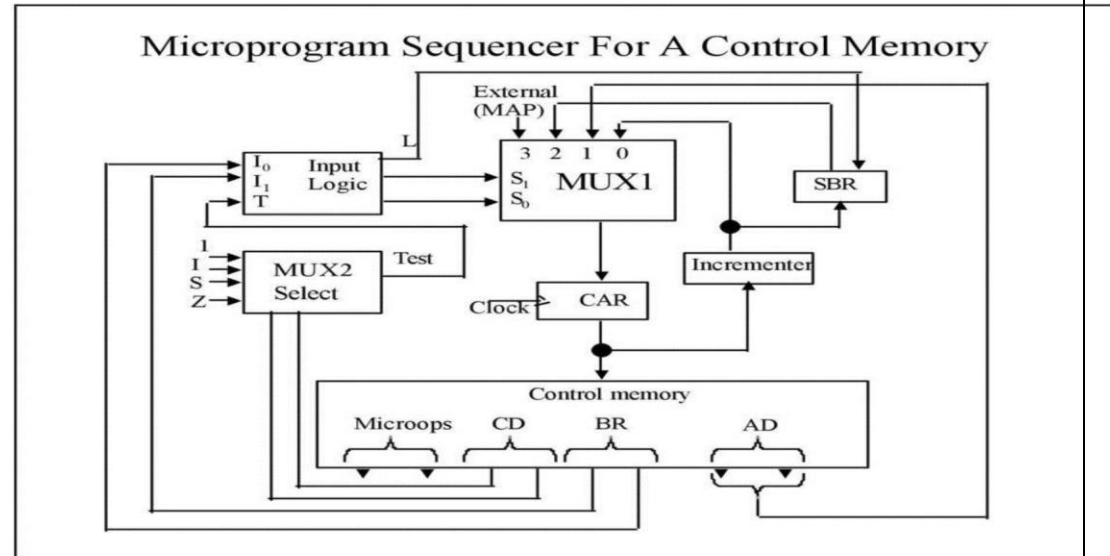


# Microprogram Sequencer

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



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



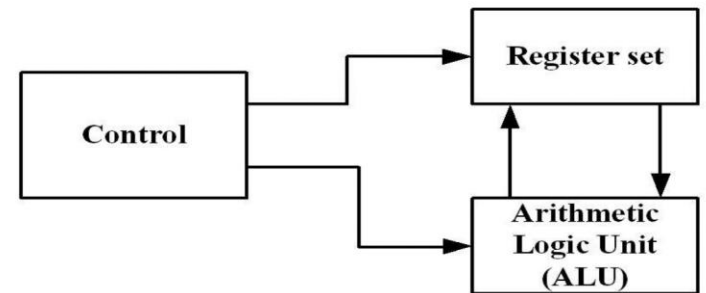
### Input Logic Truth Table For A Microprogrammed Sequencer

BR Field		Input			MUX1		Load SBR	
		I <sub>1</sub>	I <sub>0</sub>	T	S <sub>1</sub>	S <sub>0</sub>	L	
0	0	0	0	0	0	0	0	Next address
0	0	0	0	1	0	1	0	Specified addr.
0	1	0	1	0	0	0	0	
0	1	0	1	1	0	1	1	
1	0	1	0	x	1	0	0	Subroutine ret.
1	1	1	1	x	1	1	0	Ext. addr.

# Central Processing Unit

- The main part of the computer that performs the bulk of data-processing operations is called the central processing unit and is referred to as the CPU.
- The CPU is made up of three major parts, as shown in Fig
  1. The register set stores intermediate data used during the execution of the instructions.
  2. The arithmetic logic unit (ALU) performs the required microoperations for executing the instructions.
  3. The control unit supervises the transfer of information among the registers and instructs the ALU as to which operation to perform.

**Central Processing Unit**

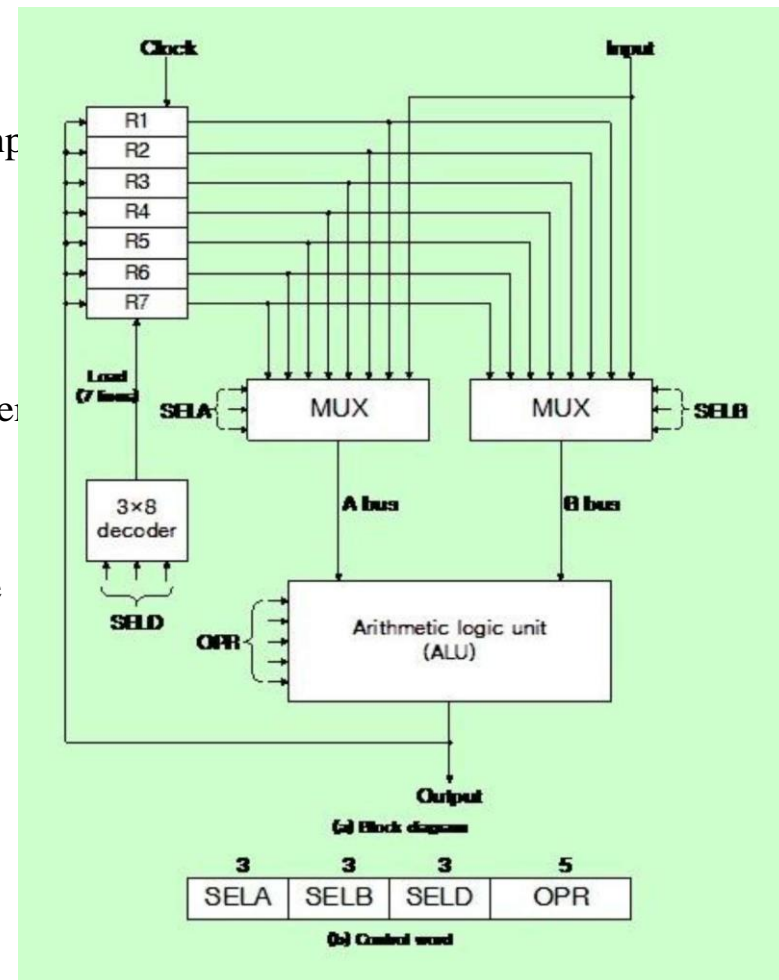


**Major components of CPU**

## General register organization

- Memory locations are needed for storing pointers, counters, return address, temporary results etc.
- Referring to these memory locations is very time consuming because memory access is the most time consuming operation in a computer.
- Therefore it is convenient to store these intermediate values in processor registers.
- When there are many registers in the system they are connected through a common bus system.
- The registers communicate with each other for data transfer as well as for performing some micro operations.
- Hence it is necessary to provide a common unit that performs arithmetic, logic and shift operations in the processor.

- A bus organization for 7 CPU registers is shown in the figure.
- The outputs of each register is connected to the two multiplexers(MUX) to form the two buses A and B.
- The selection lines in each multiplexer select one register or their data for the particular bus.
- The A and B buses form the inputs to a common arithmetic logic unit (ALU).
- The operation selected in the ALU determines the arithmetic or logic micro operation that is to be performed.
- The result of micro operation goes into the inputs of all the registers.
- The register that receives the information is selected by a decoder.
- The decoder activates one of the register load inputs, thus providing transfer path between the data in the output bus and the inputs of the selected destination register.
- The control unit that operates the CPU bus system directs the information flow through the registers and ALU by selecting the various components in the system.

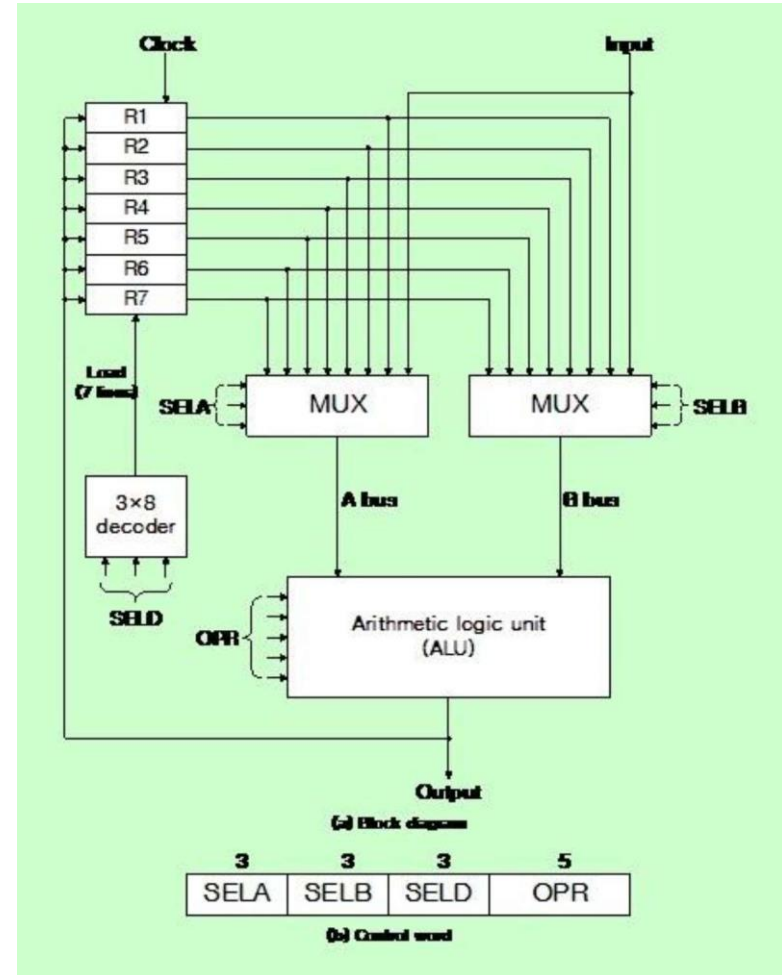


- Eg; to perform the following operation.

$$R_1 \leftarrow R_2 + R_3$$

The control must provide binary selection variables to the following selector inputs.

1. MUX A selector (SELA): to place the contents of  $R_2$  into bus A.
2. MUX B selector (SELB) : to place the content of  $R_3$  into bus B.
3. ALU operation selector (OPR): to provide the arithmetic addition  $A+B$ .
4. Decoder destination selector(SELD): to transfer the content of the output bus into  $R_1$ .





- The four control selection variables are generated in the control unit and must be available at the beginning of a clock cycle.
- The data from the two source registers propagate through the gates in the multiplexers and the ALU, to the output bus, and onto the input of the destination register, all during the clock cycle interval.
- Then, when the next clock transition occurs, the binary information from the output bus is transferred into  $R_1$ .
- To achieve a fast response time, the ALU is constructed with high-speed circuits.
- The buses are implemented with multiplexers or three-state gates

## Control word

- Control word is defined as a word whose individual bits represent various control signals.
- There are 14 selection inputs in the unit, and their combined value specifies a control word. The
- 14 bit control word is defined in the following fig, it consists of 4 fields.



- Three fields contain 3 bits each and last field contains 5 bits.
- The three bits of SELA select a source register for the A input of the ALU.
- The three bits of SELB select a source register for the B input of the ALU.
- The three bits of SELD select a destination register using the decoder and its seven load outputs. The
- five bits of OPR select one of the operations in the ALU.
- The 14 bit control word when applied to the selection inputs specify a particular microoperation.

- The encoding of the register selections is specified in table
- The **3-bit binary code** listed in the first column of the table specifies the binary code for each of the three fields.

The **register** selected by fields SELA, SELB, and SELD is the one whose decimal number is equivalent to the binary number in the code. When SELA or SELB is 000, the corresponding multiplexer selects the external input data.

When **SELD = 000**, no destination register is selected but the contents of the output bus are available in the external output.

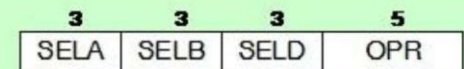
The ALU provides arithmetic and logic operations.

The CPU must also provide shift operations. The shifter may be placed in the input of the ALU to provide a preshift capability, or at the output of the ALU to provide postshifting capability.

In some cases, the shift operations are included with the ALU.

**TABLE 1** Encoding of Register Selection Fields

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



(b) Control word

- The encoding of the ALU operations for the CPU is shown in the following table.
- The OPR field has 5 bits and each operation is designated with a symbolic name.
- **Examples of microoperations**
- **A control word of 14 bits** is needed to specify a microoperation in the CPU. The control word for a given microoperation can be derived from the selection variables.

For example, the subtract microoperation given by the statement

$$R1 \leftarrow R2 - R3$$

- specifies R2 for the A input of the ALU, R3 for the B input of the ALU, R1 for the destination register, and an ALU operation to subtract A - B.

Thus the control word is specified by the four fields and the corresponding binary value for each field is obtained from the encoding listed in Tables 1 and 2.

The binary control word for the subtract microoperation is

**010 011 001 00101** and is obtained as follows:

**TABLE 2** Encoding of ALU Operations

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

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

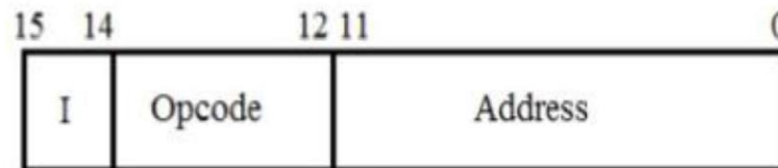
- **The control word** for this microoperation and a few others are listed in Table 3.
- **The increment and transfer microoperations** do not use the B input of the ALU.
- **For these cases**, the B field is marked with a dash. We assign 000 to any unused field when formulating the binary control word, although any other binary number may be used.
- **To place the content of a register** into the output terminals we place the content of the register into the A input of the ALU, but none of the registers are selected to accept the data.
- **The ALU operation TSFA** places the data from the register, through the ALU, into the output terminals.
- **The direct transfer** from input to output is accomplished with a control word of all 0's (making the B field 000).
- **A register** can be cleared to 0 with an exclusive-OR operation. This is because  $x \oplus x = 0$ .
- **It is apparent** from these examples that many other microoperations can be generated in the CPU.
- **The most efficient way** to generate control words with a large number of bits is to store them in a memory unit.
- **A memory unit** that stores control words is referred to as a control memory.
- **By reading consecutive control words from memory**, it is possible to initiate the desired sequence of microoperations for the CPU.
- **This type of control** is referred to as microprogrammed control.

TABLE 3 Examples of Microoperations for the CPU

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

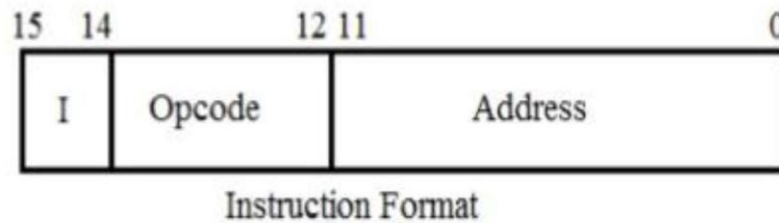
## Instruction formats

- An instruction is a group of bits that instructs the computer to do some operation. These bits are arranged in some instruction code format.
- Control unit in the CPU will interpret each instruction code and provide the necessary control functions needed to process the instruction.
- A computer will usually have a variety of instruction code formats.
- The format of an instruction is represented in a rectangular box symbolizing the bits of the instruction as they appear in memory words or in a control register.
- The bits of the instruction are divided into groups called fields.
- The most common fields found in instruction formats are:
  1. An operation code field that specifies the operation to be performed
  2. An address field that designates a memory address or a processor register.
  3. A mode field that specifies the way the operand or the effective address is determined.



Instruction Format

- Computers may have instructions of several different lengths containing varying number of addresses.
- The number of address fields in the instruct format of a computer depends on the internal organization of its registers.
- Most computers have one of three types of CPU organizations:
  1. Single accumulator organization.
  2. General register organization.
  3. Stack organization.





- **Single Accumulator Organization:**

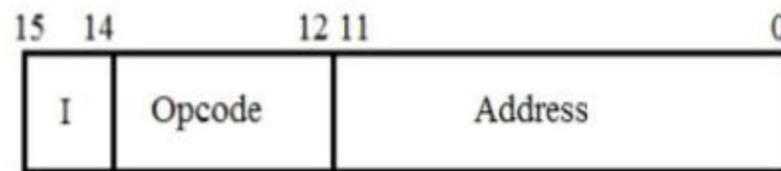
- In an accumulator type organization all the operations are performed with an implied accumulator register.
- The instruction format in this type of computer uses one address field.
- For example, the instruction that specifies an arithmetic addition defined by an assembly language instruction as

**ADD X**          here X is the address of the operand.

- The ADD instruction in this case results in the operation

**$AC \leftarrow AC + M[X]$ .**

- AC is the accumulator register and M[X] symbolizes the memory word located at address X



Instruction Format

## General register organization:

- The instruction format in this type of computer needs three register address fields.
- Eg 1; the instruction for an arithmetic addition may be written in an assembly language as

**ADD R1, R2, R3**

This denote the operation  **$R1 \leftarrow R2 + R3$** .

- The number of address fields in the instruction can be reduced from three to two if the destination register is the same as one of the source registers.
- Eg2; Thus the instruction **ADD R1, R2** would denote the operation

**$R1 \leftarrow R1 + R2$** .

Only register addresses for R1 and R2 need be specified in this instruction.

- General register-type computers employ two or three address fields in their instruction format.
- Each address field may specify a processor register or a memory word.
- An instruction symbolized by **ADD R1, X** would specify the operation  **$R1 \leftarrow R1 +$**
- **$M[X]$** . It has two address fields, one for register R1 and the other for the memory address X.

## Stack organization:

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

- To illustrate The influence of the number of addresses on computer programs, we will evaluate the arithmetic statement

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

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

### Three Address Instructions:

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

- Three - Address Instruction

ADD R1, A, B

$R1 \leftarrow M[A] + M[B]$

ADD R2, C, D

$R2 \leftarrow M[C] + M[D]$

MUL X, R1, R2

$M[X] \leftarrow R1 * R2.$

- 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 to specify either a processor register or memory word. The program to evaluate  $X = (A+B) * (C+D)$  is as follows

- Two - Address Instruction

MOV R1, A	$R1 \leftarrow M[A]$
ADD R1, B	$R1 \leftarrow R1 + M[B]$
MOV R2, C	$R2 \leftarrow M[C]$
ADD R2, D	$R2 \leftarrow R2 + M[D]$
MUL R1, R2	$R1 \leftarrow R1 * R2$
MOV X, R1	$M[X] \leftarrow R1$

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

- **One Address Instructions:**

- One-address instructions use an implied accumulator (AC) register for all data manipulation.
- AC contains the result of all operations.
- The program to evaluate  $X=(A+B) * (C+D)$  is

- **One-Address**

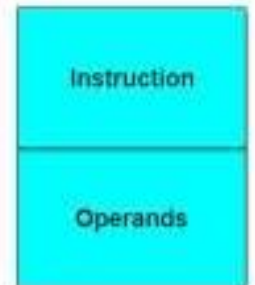
1. LOAD A ;  $AC \leftarrow M[A]$
2. ADD B ;  $AC \leftarrow AC + M[B]$
3. STORE T ;  $M[T] \leftarrow AC$
4. LOAD C ;  $AC \leftarrow M[C]$
5. ADD D ;  $AC \leftarrow AC + M[D]$
6. MUL T ;  $AC \leftarrow AC * M[T]$
7. STORE X ;  $M[X] \leftarrow AC$



Instruction Format



Memory



- 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).

- Evaluate  $X = (A + B) * (C + D)$
- PUSH      A      TOS  $\leftarrow$  A
- PUSH      B      TOS  $\leftarrow$  B
- ADD                    TOS  $\leftarrow$  (A+B)
- PUSH      C      TOS  $\leftarrow$  C
- PUSH      D      TOS  $\leftarrow$  D
- ADD                    TOS  $\leftarrow$  (C+D)
- MUL                    TOS  $\leftarrow$  (C+D)\*(A+B)
- POP        X      M[X]  $\leftarrow$  TOS

Push A
Push B
ADD
Push C
Push D
ADD
Mult
Store

## Addressing modes

- Operands are chosen during program execution depending on the addressing mode of the instruction. Computers
- use addressing mode techniques to
  1. To provide facilities such as pointers to memory, counters for loop control, indexing of data, and program relocation.
  2. To reduce the number of bits in the addressing field of the instruction
- **Types of addressing modes**
  - Implied Mode
  - Immediate Mode
  - Register Mode
  - Register Indirect Mode
  - Autoincrement or Autodecrement Mode
  - Direct Address Mode
  - Indirect Address Mode
  - Relative Address Mode
  - Indexed Addressing Mode
  - Base Register Addressing Mode
- 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 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.

- **Register Mode:**

- When the address specifies a processor register, the instruction is said to be in the register mode. In
- this mode the operands are in registers that reside within the CPU.
- The particular register is selected from a register field in the instruction.

- **Register Indirect Mode:**

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

- **Auto-increment or Auto-Decrement Mode:**

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

- The basic two mode of addressing used in CPU are direct and indirect address mode.

- **Direct Address Mode:**

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

- **Indirect Address Mode:**

- In this mode the address field of the instruction gives the address where the effective address is stored in memory.
- Control fetches the instruction from memory and uses its address part to access memory again to read the effective address.
- A few addressing modes require that the address field of the instruction be added to the content of a specific register in the CPU.
- The effective address in these modes is obtained from the following computation:
- $\text{Effective address} = \text{address part of instruction} + \text{content of CPU register}$
- The CPU register used in the computation may be the program counter, an index register, or a base register.
- We have a different addressing mode which is used for a different application.

### Relative Address Mode:

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

### Indexed Addressing Mode:

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

### Base Register Addressing Mode:

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

## Example

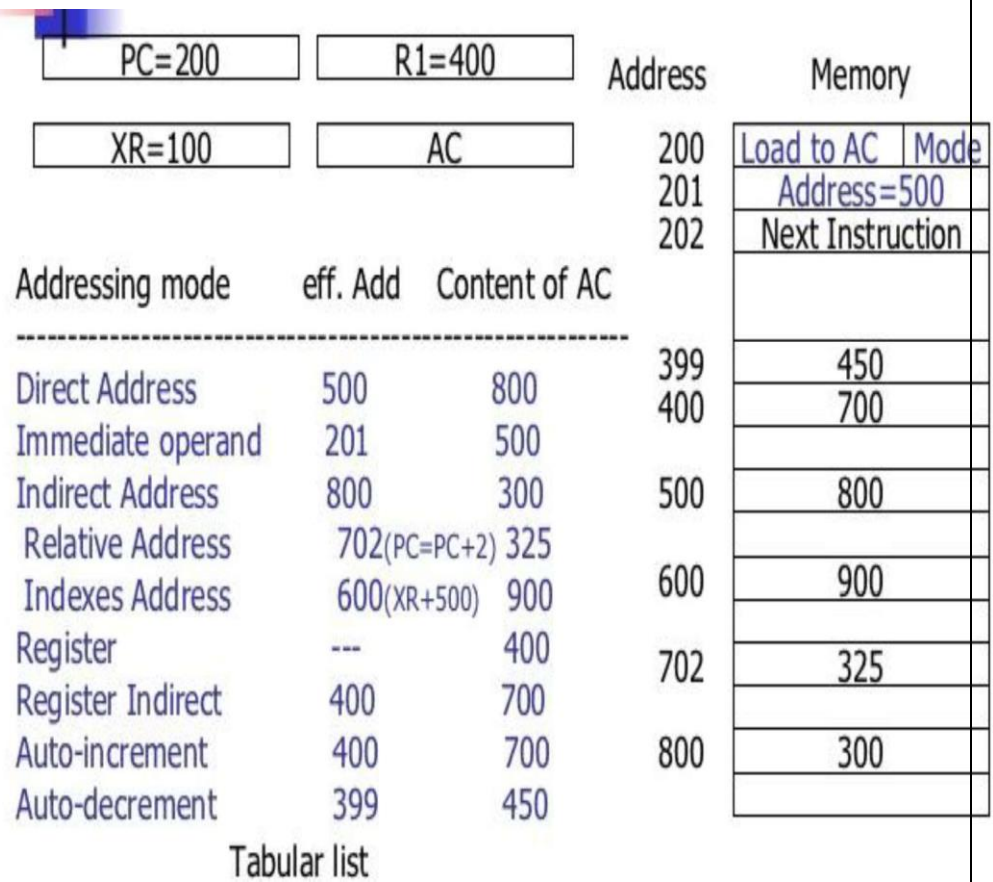
- To show the differences between the various modes, we will show the effect of the addressing modes on the instruction defined in Fig
- The two-word instruction at address 200 and 201 is a "load to AC" instruction with an address field equal to 500.
- The first word of the instruction specifies the 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 800.
- In the immediate mode the second word of the instruction is taken as the operand rather than an address, so 500 is loaded into AC
- 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.

PC=200	R1=400	Address	Memory
XR=100	AC	200	Load to AC   Mode
		201	Address=500
		202	Next Instruction
Addressing mode	eff. Add	Content of AC	
-----	-----	-----	
Direct Address	500	800	399
Immediate operand	201	500	400
Indirect Address	800	300	500
Relative Address	702(PC=PC+2)	325	
Indexes Address	600(XR+500)	900	600
Register	---	400	702
Register Indirect	400	700	
Auto-increment	400	700	800
Auto-decrement	399	450	

Tabular list



- 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.



## Data Transfer and Manipulation

- Most computer instructions can be classified into three categories:
  1. Data transfer instructions
  2. Data manipulation instructions
  3. Program control instructions
- **Data Transfer Instructions:**
  - Data transfer instructions move data from one place in the computer to another without changing the data content.
  - The most common transfers are between memory and processor registers, between processor registers and input or output, and between the processor registers themselves.
  - Table gives a list of eight data transfer instructions used in many computers.

### Typical Data Transfer Instructions

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

➤ **Data Manipulation Instructions:**

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

**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 means software subroutines.
- A list of typical arithmetic instructions is given in Table 8-7.

**Typical Arithmetic Instructions**

<b>Name</b>	<b>Mnemonic</b>
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with carry	ADDC
Subtract with borrow	SUBB
Subtract reverse	SUBR
Negate	NEG

- **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.

<b>Name</b>	<b>Mnemonic</b>
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

- **Shift Instruction**

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

**TABLE 8-9** Typical Shift Instructions

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

## Program control

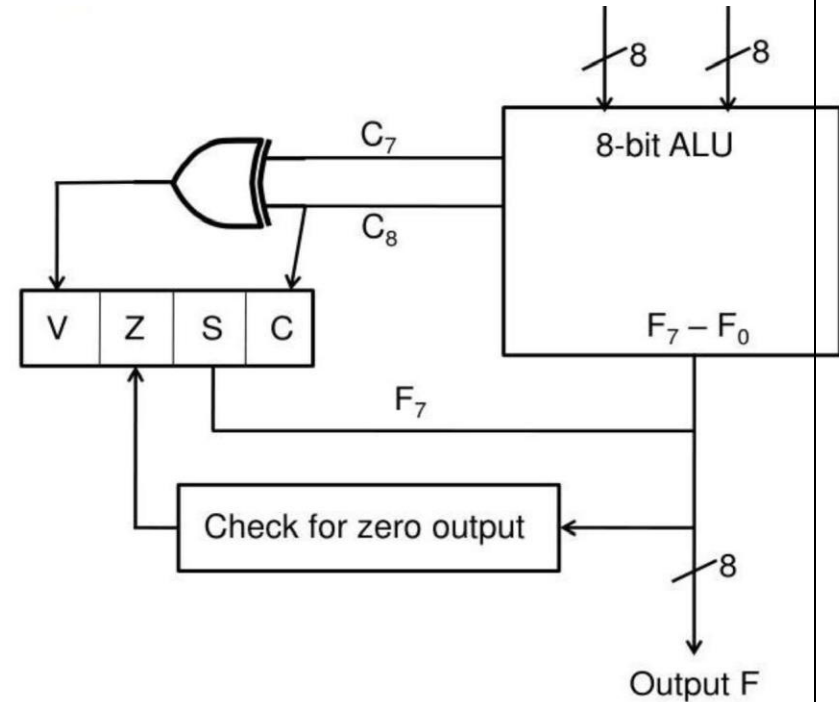
- Program control instructions specify conditions for altering the content of the program counter.
- The change in value of the program counter as a result of the execution of a program control instruction causes a break in the sequence of instruction execution.
- This instruction provides control over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

**Typical Program Control Instructions**

Name	Mnemonic
Branch	BR
Jump	JMP
Skip next instruction	SKP
Call procedure	CALL
Return from procedure	RET
Compare (by subtraction)	CMP
Test (by ANDing)	TEST

- **Status Bit Conditions:**

- The ALU circuit in the CPU have status register for storing the status bit conditions.
- Status bits are also called condition-code bits or flag bits.
- Following Figure shows block diagram of an 8-bit ALU with a 4-bit status register
- The four status bits are symbolized by C, S, Z, and V. The bits are set or cleared as a result of an operation performed in the ALU.
- Bit C (carry) is set to 1 if the end carry  $C_8$  is 1. It is cleared to 0 if the carry is 0.
- S (sign) is set to 1 if the highest-order bit  $F_7$  is 1. It is set to 0 if the bit is 0.
- Bit Z (zero) is set to 1 if the output of the ALU contains all 0's. It is clear to 0 otherwise. In other words,  $Z = 1$  if the output is zero and  $Z = 0$  if the output is not zero.
- Bit V (overflow) is set to 1 if the exclusive-OR of the last two carries equal to 1, and cleared to 0 otherwise.
- The above status bits are used in conditional jump and branch instructions.





# Module 3

## Data Representation

### Section 3.1 – Data Types

- Registers contain either data or control information
- Control information is a bit or group of bits used to specify the sequence of command signals needed for data manipulation
- Data are numbers and other binary-coded information that are operated on
- Possible data types in registers:
  - Numbers used in computations
  - Letters of the alphabet used in data processing
  - Other discrete symbols used for specific purposes
- All types of data, except binary numbers, are represented in binary-coded form
- 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 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 string of digits 101101 in the binary number system represents the quantity

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

- $(101101)_2 = (45)_{10}$
- We will also use the octal (radix 8) and hexadecimal (radix 16) number systems

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

$$(F3)_{16} = F \times 16^1 + 3 \times 16^0 = (243)_{10}$$

- 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

**Figure 3-1** Conversion of decimal 41.6875 into binary.

Integer = 41	Fraction = 0.6875
$\begin{array}{r l} 41 & \\ 20 & 1 \\ 10 & 0 \\ 5 & 0 \\ 2 & 1 \\ 1 & 0 \\ 0 & 1 \end{array}$	$\begin{array}{r} 0.6875 \\ \underline{\phantom{0.}2} \\ 1.3750 \\ \underline{\phantom{0.}x 2} \\ 0.7500 \\ \underline{\phantom{0.}x 2} \\ 1.5000 \\ \underline{\phantom{0.}x 2} \\ 1.0000 \end{array}$
$(41)_{10} = (101001)_2$	$(0.6875)_{10} = (0.1011)_2$
$(41.6875)_{10} = (101001.1011)_2$	

- Each octal digit corresponds to three binary digits
- Each hexadecimal digit corresponds to four binary digits
- Rather than specifying numbers in binary form, refer to them in octal or hexadecimal and reduce the number of digits by 1/3 or 1/4, respectively

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

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

**TABLE 3-1 Binary-Coded Octal Numbers**

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

TABLE 3-2 Binary-Coded Hexadecimal Numbers

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

- A binary code is a group of  $n$  bits that assume up to  $2^n$  distinct combinations
- A four bit code is necessary to represent the ten decimal digits – 6 are unused
- The most popular decimal code is called *binary-coded decimal* (BCD)
- BCD is different from converting a decimal number to binary
- For example 99, when converted to binary, is 1100011
- 99 when represented in BCD is 1001 1001

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

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

- The standard alphanumeric binary code is ASCII
- This uses seven bits to code 128 characters
- Binary codes are required since registers can hold binary information only

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

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

### Section 3.2 – 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 2389 is  $7610 + 1 = 7611$
- The 2's complement of 101100 is  $010011 + 1 = 010100$
  
- Subtraction of unsigned  $n$ -digit numbers:  $M - N$ 
  - Add  $M$  to the  $r$ 's complement of  $N$  – this results in  

$$M + (r^n - N) = M - N + r^n$$
  - If  $M \geq N$ , the sum will produce an end carry  $r^n$  which is discarded
  - 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:  $72532 - 13250 = 59282$ . The 10's complement of 13250 is 86750.

M	= 72352
10's comp. of N	= <u>+86750</u>
Sum	= 159282
Discard end carry	= <u>-100000</u>
Answer	= 59282

Example for  $M < N$ :  $13250 - 72532 = -59282$

M	= 13250
10's comp. of N	= <u>+27468</u>
Sum	= 40718
No end carry	
Answer	= -59282 (10's comp. of 40718)

Example for  $X = 1010100$  and  $Y = 1000011$

X	= 1010100
2's comp. of Y	= <u>+0111101</u>
Sum	= 10010001
Discard end carry	= <u>-10000000</u>
Answer $X - Y$	= 0010001

Y	= 1000011
2's comp. of X	= <u>+0101100</u>
Sum	= 1101111



No end carry

Answer

= -0010001 (2's comp. of 1101111)

### Section 3.3 – Fixed-Point Representation

- Positive integers and zero can be represented by unsigned numbers
- Negative numbers must be represented by signed numbers since + and – signs are not available, only 1's and 0's are
- Signed numbers have msb as 0 for positive and 1 for negative – msb is the sign bit
- Two ways to designate binary point position in a register
  - Fixed point position
  - Floating-point representation
- Fixed point position usually uses one of the two following positions
  - A binary point in the extreme left of the register to make it a fraction
  - A binary point in the extreme right of the register to make it an integer
  - In both cases, a binary point is not actually present
- The floating-point representations uses a second register to designate the position of the binary point in the first register
  
- When an integer is positive, the msb, or sign bit, is 0 and the remaining bits represent the magnitude
- When an integer is negative, the msb, or sign bit, is 1, but the rest of the number can be represented in one of three ways
  - Signed-magnitude representation
  - Signed-1's complement representation
  - Signed-2's complement representation
  
- Consider an 8-bit register and the number +14
  - The only way to represent it is 00001110
- Consider an 8-bit register and the number –14
  - Signed magnitude:           1 0001110
  - Signed 1's complement:    1 1110001
  - Signed 2's complement:     1 1110010
- Typically use signed 2's complement
  
- 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

$\begin{array}{r} +6 \quad 00000110 \\ +13 \quad 00001101 \\ \hline +19 \quad 00010011 \end{array}$	$\begin{array}{r} -6 \quad 11111010 \\ +13 \quad 00001101 \\ \hline +7 \quad 00000111 \end{array}$
$\begin{array}{r} +6 \quad 00000110 \\ -13 \quad 11110011 \\ \hline -7 \quad 11111001 \end{array}$	$\begin{array}{r} -6 \quad 11111010 \\ -13 \quad 11110011 \\ \hline -19 \quad 11101101 \end{array}$

- 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 have to 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

$\begin{array}{r} +70 \quad 0 \ 1000110 \\ +80 \quad 0 \ 1010000 \\ \hline +150 \quad 1 \ 0010110 \end{array}$	$\begin{array}{r} -70 \quad 1 \ 0111010 \\ -80 \quad 1 \ 0110000 \\ \hline -150 \quad 0 \ 1101010 \end{array}$
--	--

- 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 similar to 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) <sub>BCD</sub>
+9 760	(1001 0111 0110 0000) <sub>BCD</sub>
0 135	(0000 0001 0011 0101) <sub>BCD</sub>

### Section 3.4 – 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 x 10<sup>+4</sup>
- 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

### Section 3.5 – 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 3-5 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 3-6** Four Different Binary Codes for the Decimal Digit

Decimal digit	BCD		Excess-3	Excess-3 gray
	8421	2421		
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 combinations	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

### Section 3.6 – Error Detection Codes

- Transmitted binary information is subject to noise that could change bits 1 to 0 and vice versa
- An *error detection code* is a binary code that detects digital errors during transmission
- The detected errors cannot be corrected, but can prompt the data to be retransmitted
- The most common error detection code used is the *parity bit*

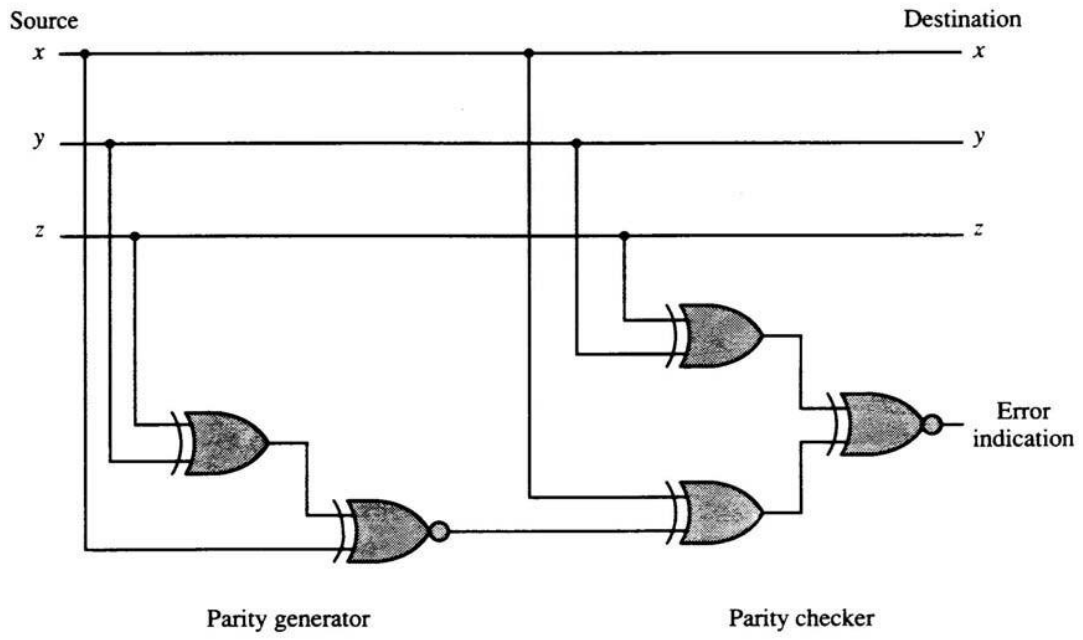
- A parity bit is an extra bit included with a binary message to make the total number of 1's either odd or even

**TABLE 3-7 Parity Bit Generation**

Message xyz	P(odd)	P(even)
000	1	0
001	0	1
010	0	1
011	1	0
100	0	1
101	1	0
110	1	0
111	0	1

- The P(odd) bit is chosen to make the sum of 1's in all four bits odd
- The even-parity scheme has the disadvantage of having a bit combination of all 0's
- Procedure during transmission:
  - At the sending end, the message is applied to a *parity generator*
  - The message, including the parity bit, is transmitted
  - At the receiving end, all the incoming bits are applied to a *parity checker*
  - Any odd number of errors are detected
- Parity generators and checkers are constructed with XOR gates (odd function)
- An odd function generates 1 iff an odd number of input variables are 1

Figure 3-3 Error detection with odd parity bit.





## **Computer Arithmetic: Introduction:**

Data is manipulated by using the arithmetic instructions in digital computers. Data is manipulated to produce results necessary to give solution for the computation problems. The Addition, subtraction, multiplication and division are the four basic arithmetic operations. If we want then we can derive other operations by using these four operations.

To execute arithmetic operations there is a separate section called arithmetic processing unit in central processing unit. The arithmetic instructions are performed generally on binary or decimal data. Fixed-point numbers are used to represent integers or fractions. We can have signed or unsigned negative numbers. Fixed-point addition is the simplest arithmetic operation.

If we want to solve a problem then we use a sequence of well-defined steps. These steps are collectively called algorithm. To solve various problems we give algorithms.

In order to solve the computational problems, arithmetic instructions are used in digital computers that manipulate data. These instructions perform arithmetic calculations.

And these instructions perform a great activity in processing data in a digital computer. As we already stated that with the four basic arithmetic operations addition, subtraction, multiplication and division, it is possible to derive other arithmetic operations and solve scientific problems by means of numerical analysis methods.

A processor has an arithmetic processor(as a sub part of it) that

executes arithmetic operations. The data type, assumed to reside in processor, registers during the execution of an arithmetic instruction. Negative numbers may be in a signed magnitude or signed complement representation. There are three ways of representing negative fixed point - binary numbers signed magnitude, signed 1's complement or signed 2's complement. Most computers use the signed magnitude representation for the mantissa.

### **Addition and Subtraction :**

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

We designate the magnitude of the two numbers by A and B.

Where the signed numbers are added or subtracted, we find that there are eight different conditions to consider, depending on the sign of the numbers and the operation performed. These conditions are listed in the first column of Table 4.1. The other columns in the table show the actual operation to be performed with the magnitude of the numbers. The last column is needed to present a negative zero. In other words, when two equal numbers are subtracted, the result should be +0 not -0.

The algorithms for addition and subtraction are derived from the table and can be stated as follows (the words parentheses should be used for the subtraction algorithm)

# Addition and Subtraction of Signed-Magnitude Numbers

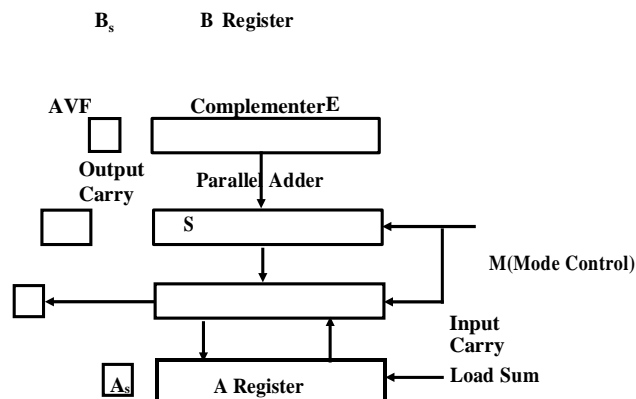
## SIGNED MAGNITUDE ADDITION AND

## SUBTRACTION

**Addition:**  $A + B$  ; A: Augend; B: Addend  
**Subtraction:**  $A - B$  ; A: Minuend; B: Subtrahend

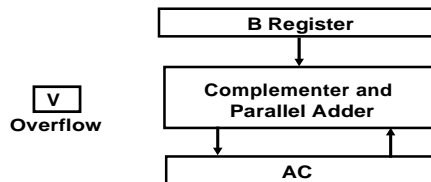
Operation	Add Magnitude	Subtract Magnitude		
		When A > B	When A < B	When A = B
$(+A) + (+B)$	$+(A + B)$			
$(+A) + (-B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(-A) + (+B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$
$(-A) + (-B)$	$-(A + B)$			
$(+A) - (+B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(+A) - (-B)$	$+(A +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

### Hardware Implementation

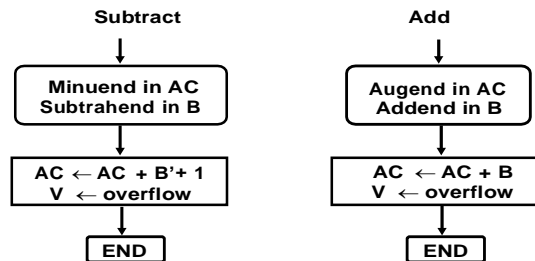


## SIGNED 2'S COMPLEMENT ADDITION AND SUBTRACTION

### Hardware



### Algorithm



## Algorithm:

- The flowchart is shown in Figure 7.1. The two signs A, and B, are compared by an exclusive-OR gate.

If the output of the gate is 0 the signs are identical; If it is 1, the signs are different.

- For an add operation, identical signs dictate that the magnitudes be added. For a subtract operation, different signs dictate that the magnitudes be added.
- The magnitudes are added with a microoperation  $E \leftarrow A + B$ , where E is a register that combines E and A. The carry in E after the addition constitutes an overflow if it is equal to 1. The value of E is transferred into the add-overflow flip-flop AVF.
- The two magnitudes are subtracted if the signs are different for an add operation or identical for a subtract operation. The magnitudes are subtracted by adding A to the 2's complemented B. No overflow can occur if the numbers are subtracted so AVF is cleared to 0.
- 1 in E indicates that  $A \geq B$  and the number in A is the correct result. If this number is zero, the sign A must be made positive to avoid a negative zero.
- 0 in E indicates that  $A < B$ . For this case it is necessary to take the 2's complement of the value in A. The operation can be done with one microoperation  $A \leftarrow A' + 1$ .
- However, we assume that the A register has circuits for microoperations complement and increment, so the 2's complement is obtained from these two microoperations.
- In other paths of the flowchart, the sign of the result is the same as the sign of A, so no change in A is required. However, when  $A < B$ , the sign of the result is the complement of the original sign of A. It is then necessary to complement A, to obtain the correct sign.
- The final result is found in register A and its sign in  $A_s$ . The value in AVF provides an overflow indication. The final value of E is immaterial.
- Figure 7.2 shows a block diagram of the hardware for implementing the addition and subtraction operations.

It consists of registers A and B and sign flip-flops  $A_s$  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 two numbers.
- The add-overflow flip-flop AVF holds the overflow bit when A and B are added.
- The A register provides other microoperations that may be needed when we specify the sequence of steps in the algorithm.

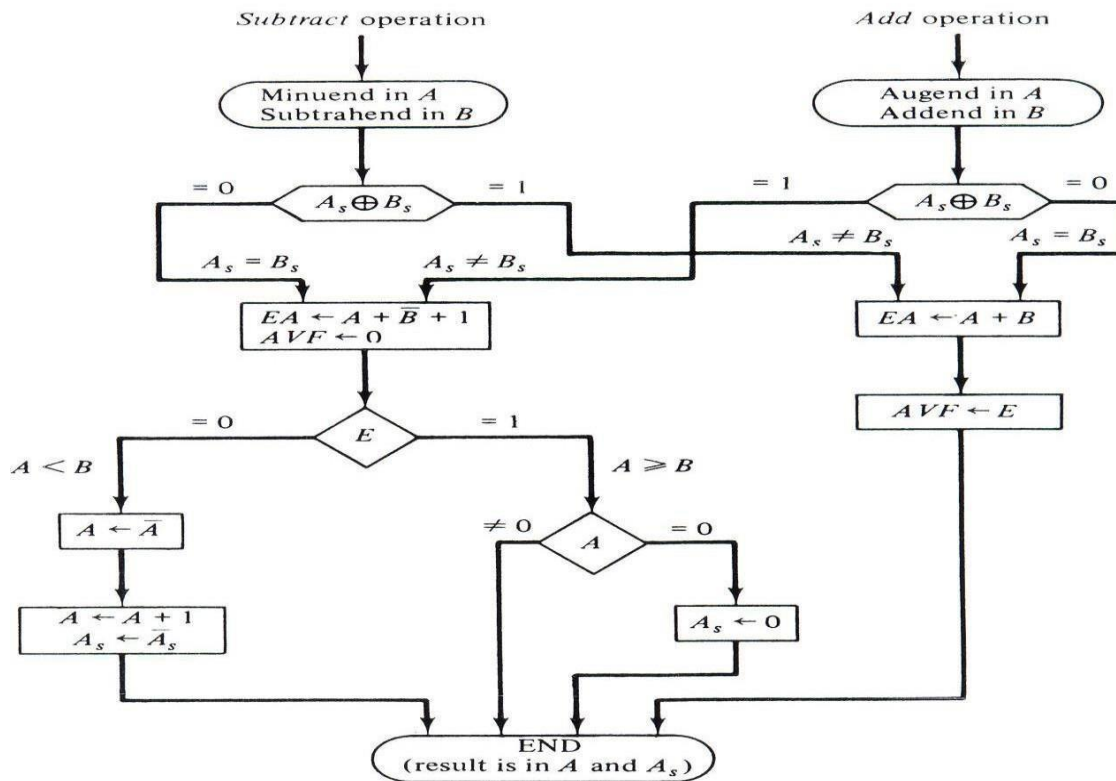
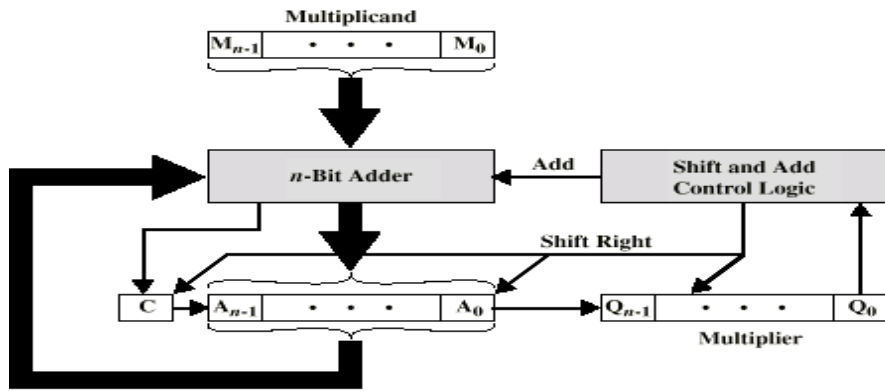


Figure 10-2 Flowchart for add and subtract operations.

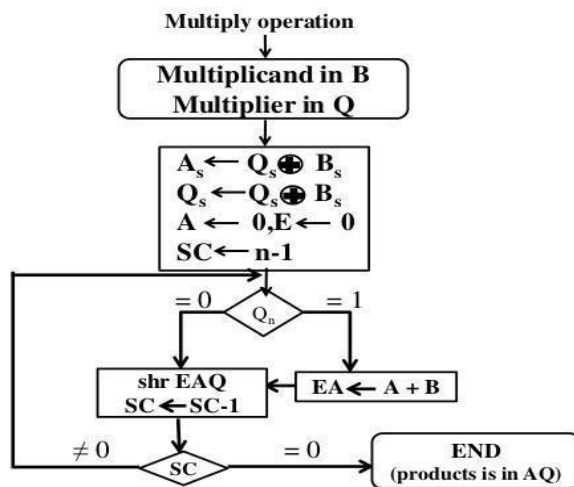
### Multiplication Algorithm:

In the beginning, the multiplicand is in B and the multiplier in Q. Their corresponding signs are in B<sub>s</sub> and Q<sub>s</sub> respectively. We compare the signs of both A and Q and set to corresponding sign of the product since a double-length product will be stored in registers A and Q. Registers A and E are cleared and the sequence counter SC is set to the number of bits of the multiplier. Since an operand must be stored with its sign, one bit of the word will be occupied by the sign and the magnitude will consist of n-1 bits.

Now, the low order bit of the multiplier in Q<sub>n</sub> is tested. If it is 1, the multiplicand (B) is added to present partial product (A), 0 otherwise. Register EAQ is then shifted once to the right to form the new partial product. The sequence counter is decremented by 1 and its new value checked. If it is not equal to zero, the process is repeated and a new partial product is formed. When SC = 0 we stop the process.



C	A	Q	M		
0	0000	1101	1011	Initial Values	
0	1011	1101	1011	Add	} First Cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	} Second Cycle
0	1101	1111	1011	Add	
0	0110	1111	1011	Shift	} Third Cycle
1	0001	1111	1011	Add	
0	1000	1111	1011	Shift	} Fourth Cycle



**Figure: Flowchart for multiply operation.**

**Booth's algorithm :**

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

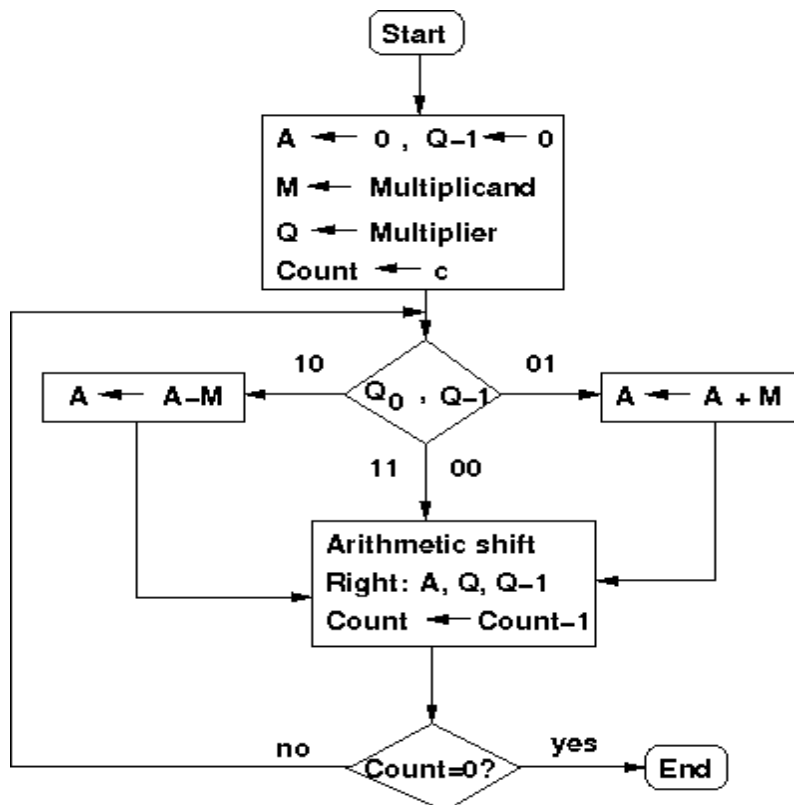
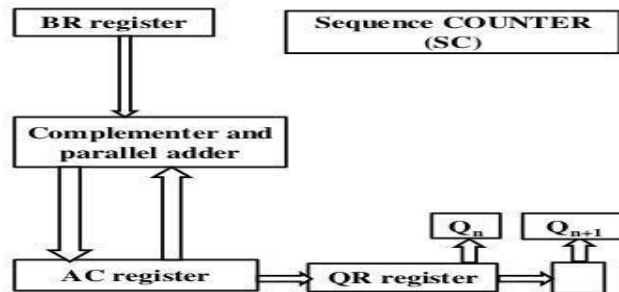
It operates on the fact that strings of 0's in the multiplier require no addition but just

shifting, and a string of 1's in the multiplier from bit weight  $2^k$  to weight  $2^m$  can be treated as  $2^{k+1} - 2^m$ .

- For example, the binary number 001110 (+14) has a string 1's from  $2^3$  to  $2^1$  ( $k=3, m=1$ ). The number can be represented as  $2^{k+1} - 2^m = 2^4 - 2^1 = 16 - 2 = 14$ . Therefore, the multiplication  $M \times 14$ , where  $M$  is the multiplicand and 14 the multiplier, can be done as  $M \times 2^4 - M \times 2^1$ .
- Thus the product can be obtained by shifting the binary multiplicand  $M$  four times to the left and subtracting  $M$  shifted left once.

## Hardware for Booth Algorithm

- Sign bits are not separated from the rest of the registers
- rename registers A, B, and Q as AC, BR and QR respectively
- $Q_n$  designates the least significant bit of the multiplier in register QR
- Flip-flop  $Q_{n+1}$  is appended to QR to facilitate a double bit inspection of the multiplier



- As in all multiplication schemes, booth algorithm requires examination of the multiplier bits and shifting of partial product.
- Prior to the shifting, the multiplicand may be added to the partial product, subtracted from the partial, 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 in a string of 0's in the multiplier.
  3. The partial product does not change when multiplier bit is identical to the previous multiplier bit.
- ▢ The algorithm works for positive or negative multipliers in 2's complement representation.
  - ▢ This is because a negative multiplier ends with a string of 1's and the last operation will be a subtraction of the appropriate weight.
  - ▢ 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.

### Division Algorithms

Division of two fixed-point binary numbers in signed magnitude representation is performed with paper and pencil by a process of successive compare, shift and subtract operations. Binary division is much simpler than decimal division because here the quotient digits are either 0 or 1 and there is no need to estimate how many times the dividend or partial remainder fits into the divisor. The division process is described in Figure

$$\begin{array}{r}
 \text{Quotient} \\
 \text{Dividend} \\
 \text{Divisor } 1101 \overline{) 100010010} \\
 \underline{-1101} \\
 10000 \\
 \underline{-1101} \\
 1110 \\
 \underline{-1101} \\
 1 \quad \text{Remainder}
 \end{array}$$

The divisor is compared with the five most significant bits of the dividend. Since the 5-bit number is smaller than B, we again repeat the same process. Now the 6-bit number is greater than B, so we place a 1 for the quotient bit in the sixth position above the dividend. Now we shift the divisor once to the right and subtract it from the dividend. The difference is known as a partial remainder because the division could have stopped here to obtain a quotient of 1 and



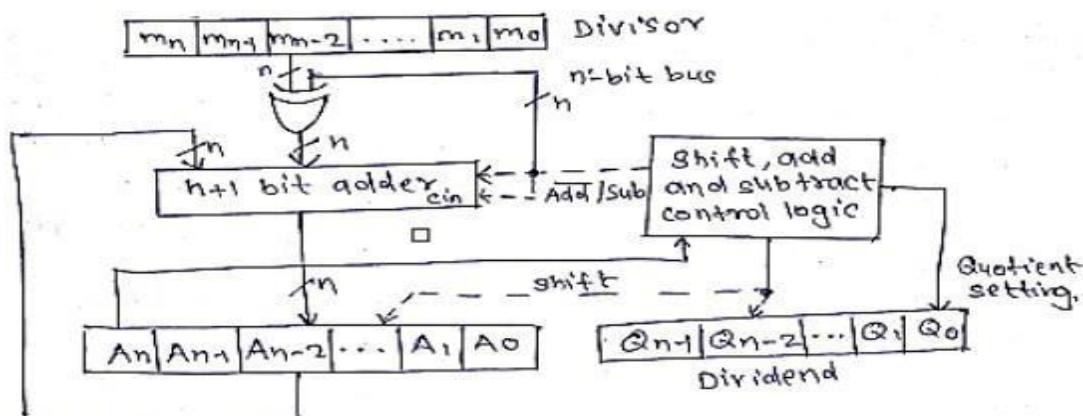
a remainder equal to the partial remainder. Comparing a partial remainder with the divisor continues the process. If the partial remainder is greater than or equal to the divisor, the quotient bit is equal to

1. The divisor is then shifted right and subtracted from the partial remainder. If the partial remainder is smaller than the divisor, the quotient bit is 0 and no subtraction is needed. The divisor is shifted once to the right in any case. Obviously the result gives both a quotient and a remainder.

### Hardware Implementation for Signed-Magnitude Data

In hardware implementation for signed-magnitude data in a digital computer, it is convenient to change the process slightly. Instead of shifting the divisor to the right, two dividends, or partial remainders, are shifted to the left, thus leaving the two numbers in the required relative position. Subtraction is achieved by adding A to the 2's complement of B. End carry gives the information about the relative magnitudes.

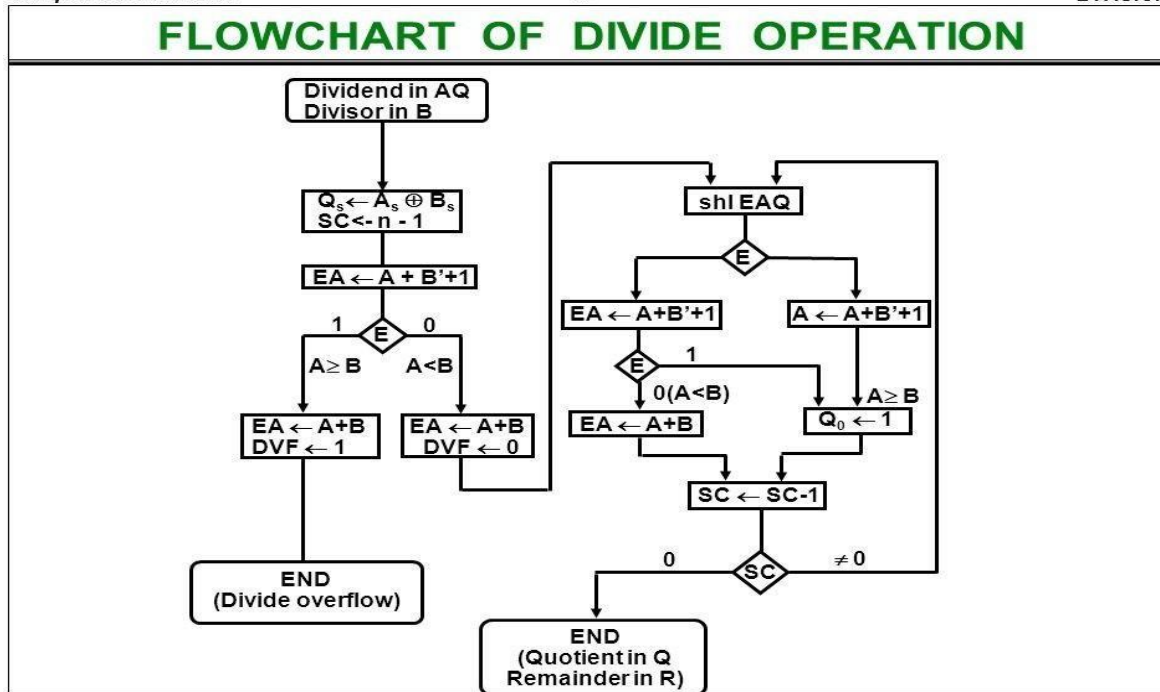
The hardware required is identical to that of multiplication. Register EAQ is now shifted to the left with 0 inserted into  $Q_n$  and the previous value of E is lost. The example is given in Figure 4.10 to clear the proposed division process. The divisor is stored in the B register and the double-length 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. E



### Hardware Implementation for Signed-Magnitude Data

**Algorithm:**

**FLOWCHART OF DIVIDE OPERATION**



Example of Binary Division with Digital Hardware

Divisor B = 10001

	E	A	Q	SC
		$\bar{B} + 1 = 01111$		
Dividend:		01110	00000	5
shl EAQ	0	11100	00000	
add $\bar{B} + 1$		01111		
E = 1	1	01011		
Set $Q_4 = 1$	1	01011	00001	4
shl EAQ	0	10110	00010	
Add $\bar{B} + 1$		01111		
E = 1	1	00101		
Set $Q_3 = 1$	1	00101	00011	3
shl EAQ	0	01010	00110	
Add $\bar{B} + 1$		01111		
E = 0; leave $Q_2 = 0$	0	11001	00110	
Add B		10001		2
Restore remainder	1	01010		
shl EAQ	0	10100	01100	
Add $\bar{B} + 1$		01111		
E = 1	1	00011		
Set $Q_1 = 1$	1	00011	01101	1
Shl EAQ	0	00110	11010	
Add $\bar{B} + 1$		01111		
E = 0; leave $Q_0 = 0$	0	10101	11010	
Add B		10001		
Restore remainder	1	00110	11010	0
Neglect E				
Remainder R:		00110		
Quotient in Q:			11010	

Floating-point Arithmetic operations :

In many high-level programming languages we have a facility for specifying floating-point numbers. The most common way is by a real declaration statement. High level programming languages must have a provision for handling floating-point arithmetic operations. The operations are generally built in the internal hardware. If no hardware is available, the compiler must be designed with a package of floating-point software subroutine. Although the hardware method is more expensive, it is much more efficient than the software method. Therefore, floating-point hardware is included in most computers and is omitted only in very small ones.

#### Basic Considerations :

There are two part of a floating-point number in a computer - a mantissa  $m$  and an exponent  $e$ . The two parts represent a number generated from multiplying  $m$  times a radix  $r$  raised to the value of  $e$ . Thus

$$m \times r^e$$

The mantissa may be a fraction or an integer. The position of the radix point and the value of the radix  $r$  are not included in the registers. For example, assume a fraction representation and a radix

10. The decimal number 537.25 is represented in a register with  $m = 53725$  and  $e = 3$  and is interpreted to represent the floating-point number

$$.53725 \times 10^3$$

A floating-point number is said to be normalized if the most significant digit of the mantissa is nonzero. So the mantissa contains the maximum possible number of significant digits. We cannot normalize a zero because it does not have a nonzero digit. It is represented in floating-point by all 0's in the mantissa and exponent.

Floating-point representation increases the range of numbers for a given register. Consider a computer with 48-bit words. Since one bit must be reserved for the sign, the range of fixed-point integer numbers will be  $+(2^{47} - 1)$ , which is approximately  $+10^{14}$ . The 48 bits can be used to represent a floating-point number with 36 bits for the mantissa and 12 bits for the exponent. Assuming fraction representation for the mantissa and taking the two sign bits into consideration, the range of numbers that can be represented is

$$+(1 - 2^{-35}) \times 2^{2047}$$

This number is derived from a fraction that contains 35 1's, an exponent of 11 bits (excluding its sign), and because  $2^{11} - 1 = 2047$ . The largest number that can be accommodated is approximately  $10^{615}$ . The mantissa that can be accommodated is 35 bits (excluding the sign) and if considered as an integer it can store a number as large as  $(2^{35} - 1)$ . This is approximately equal to  $10^{10}$ , which is equivalent to a decimal number of 10 digits.

Computers with shorter word lengths use two or more words to represent a floating-point

number. An 8-bit microcomputer uses four words to represent one floating-point number. One word of 8 bits are reserved for the exponent and the 24 bits of the other three words are used in the mantissa.

Arithmetic operations with floating-point numbers are more complicated than with fixed-point numbers. Their execution also takes longer time 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. We do this alignment by shifting one mantissa while its exponent is adjusted until it becomes equal to the other exponent. Consider the sum of the following floating-point numbers:

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

Floating-point multiplication and division need not do an alignment of the mantissas. Multiplying the two mantissas and adding the exponents can form the product. Dividing the mantissas and subtracting the exponents perform division.

The operations done 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 compared and incremented (for aligning the mantissas), added and subtracted (for multiplication) and division), and decremented (to normalize the result). We can represent the exponent in any one of the three representations - signed-magnitude, signed 2's complement or signed 1's complement.

Biased exponents have the advantage that they contain only positive numbers. Now it becomes simpler to compare their relative magnitude without bothering about their signs. Another advantage is that the smallest possible biased exponent contains all zeros. The floating-point representation of zero is then a zero mantissa and the smallest possible exponent.

### Register Configuration

The register configuration for floating-point operations is shown in figure 4.13. As a rule, the same registers and adder used for fixed-point arithmetic are used for processing the mantissas. The difference lies in the way the exponents are handled.

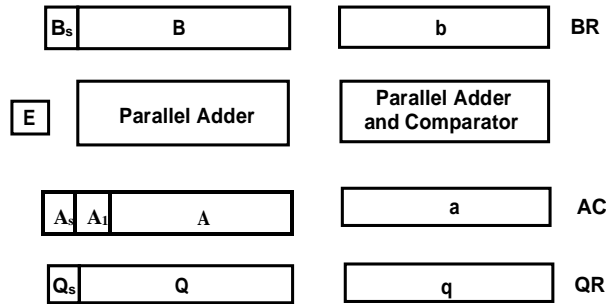
The register organization for floating-point operations is shown in Fig. 4.13. Three registers are there, BR, AC, and QR. Each register is subdivided into two parts. The mantissa part has the same uppercase letter symbols as in fixed-point representation. The exponent part may use corresponding lower-case letter symbol.

## FLOATING POINT ARITHMETIC OPERATIONS

$$F = m \times r^e$$

where m: Mantissa  
r: Radix  
e: Exponent

### Registers for Floating Point Arithmetic



*Computer Organization*

*Prof. H. Yoon*

Figure 4.13: Registers for Floating Point arithmetic operations

Assuming 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 diagram shows the most significant bit of  $A$ , labeled by  $A_1$ . The bit in his position must be a 1 to normalize the number. Note that the symbol AC represents the entire register, that is, the concatenation of  $A_s$ ,  $A$  and  $a$ .

In the similar way, 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 loads the sum into  $A$  and the carry into  $E$ . A separate parallel adder can be used for the exponents. The exponents do not have a distinct sign bit because they are biased but are represented as a biased positive quantity. It is assumed that the floating-point number are so large that the chance of an exponent overflow is very remote and so the exponent overflow will be neglected. The exponents are also connected to a magnitude comparator that provides three binary outputs to indicate their relative magnitude.

The number in the mantissa will be taken as a fraction, so they binary point is assumed to reside to the left of the magnitude part. Integer representation for floating point causes certain scaling problems during multiplication and division. To avoid these problems, we adopt a fraction representation.

The numbers in the registers should initially be normalized. After each arithmetic operation, the result will be normalized. Thus all floating-point operands are always normalized.

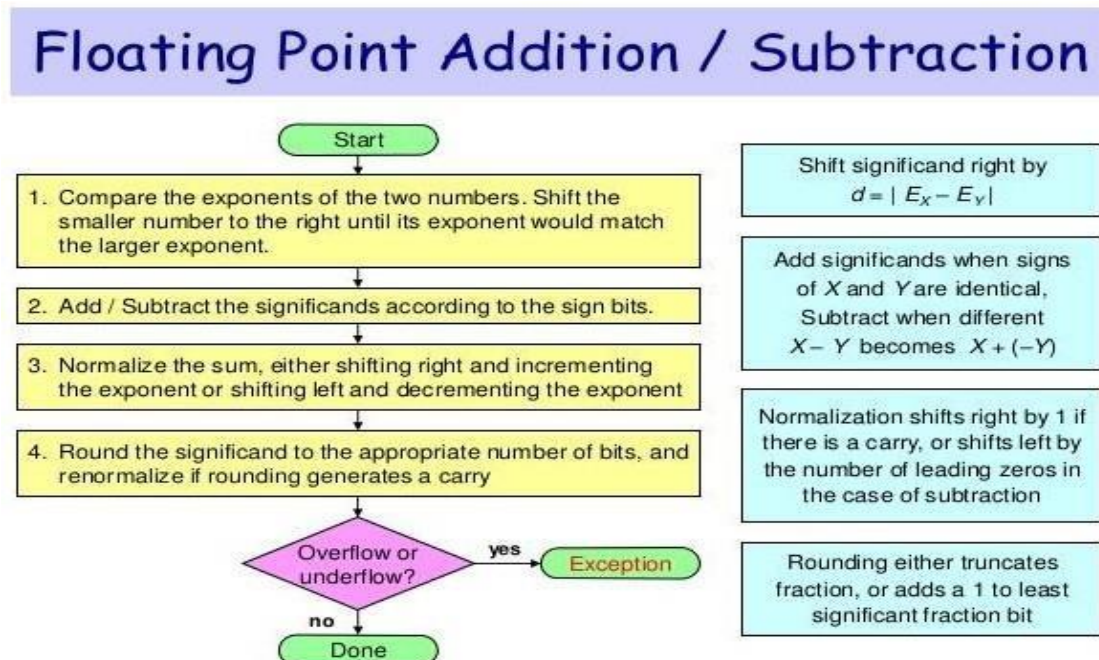
## Addition and Subtraction of Floating Point Numbers

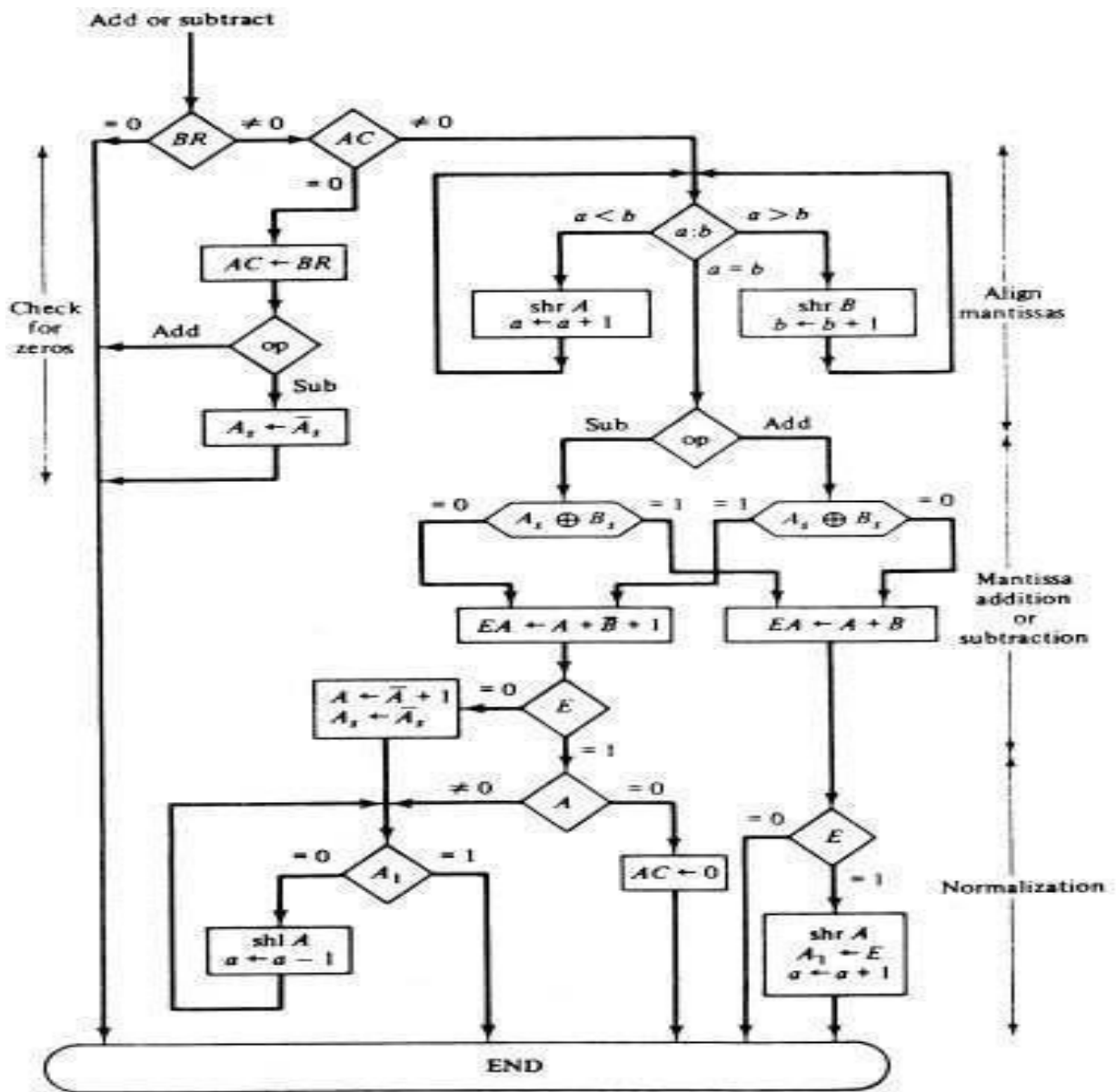
During addition or subtraction, the two floating-point operands are kept in AC and BR. The sum or difference is formed in the AC. The algorithm can be divided into four consecutive parts:

1. Check for zeros.
2. Align the mantissas.
3. Add or subtract the mantissas
4. Normalize the result

A floating-point number cannot be normalized, if it is 0. If this number is used for 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 before it is transferred to memory.

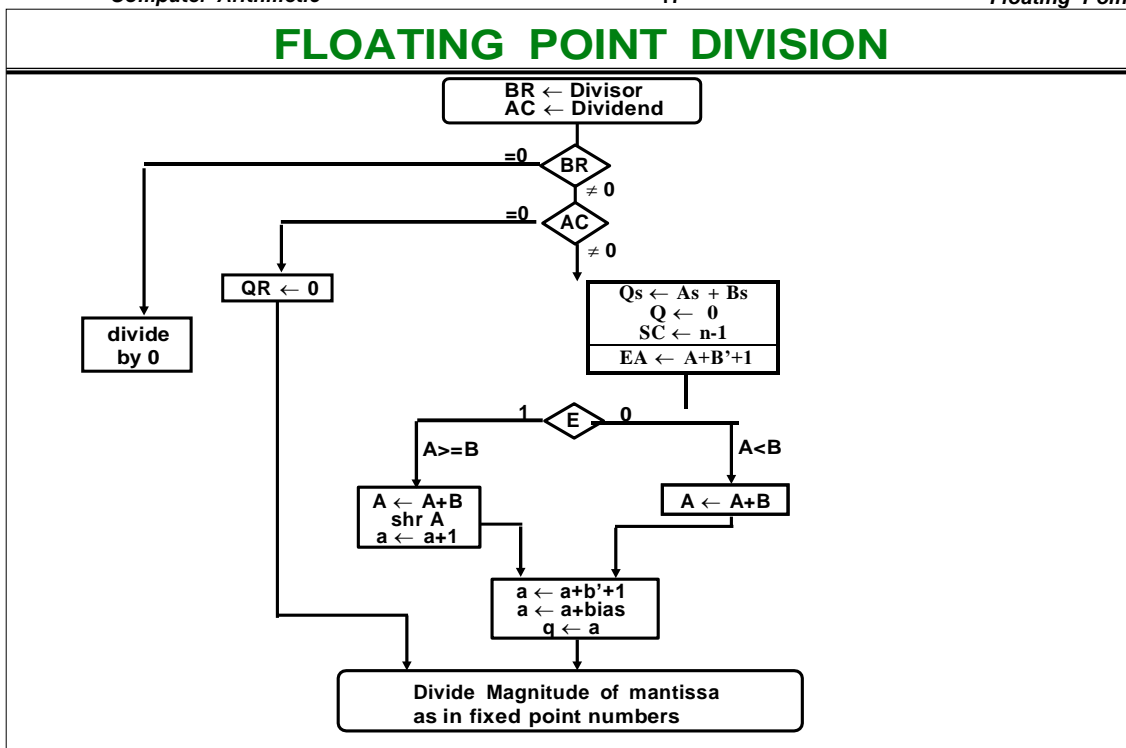
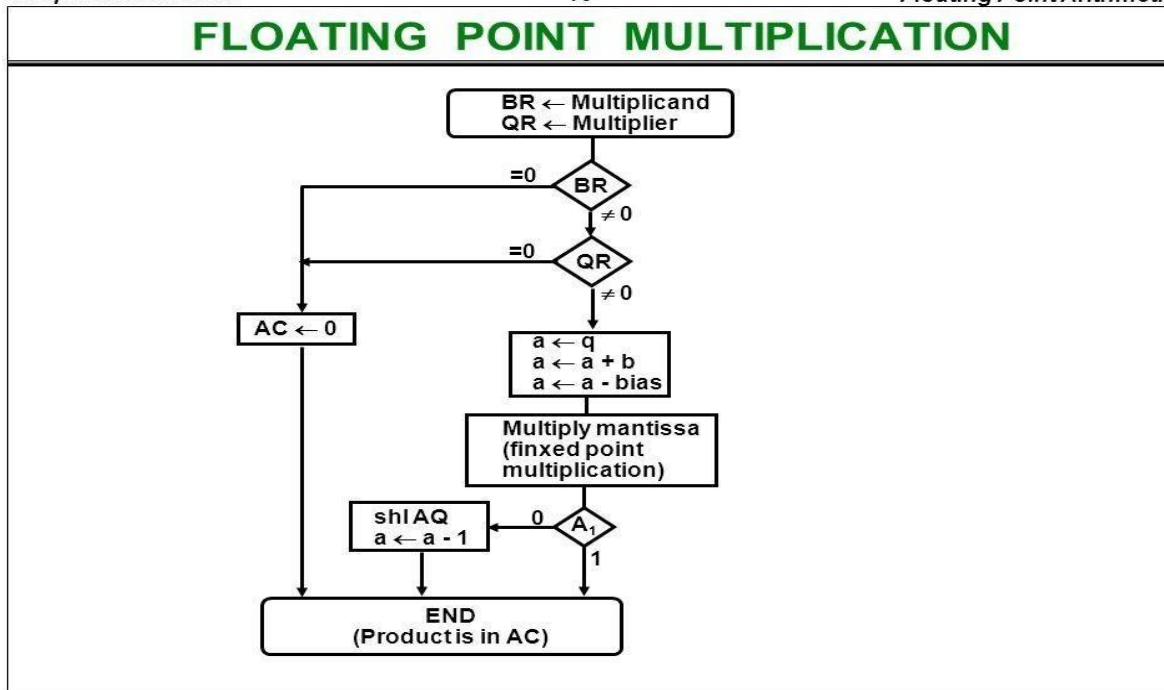
If the magnitudes were subtracted, there may be zero or may have an underflow in the result. If the mantissa is equal to zero the entire floating-point number in the AC is cleared to zero. Otherwise, the mantissa must have at least one bit that is equal to 1. The mantissa has an underflow if the most 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  $A1 = 1$ . When  $A1 = 1$ , the mantissa is normalized and the operation is completed.





Algorithm for Floating Point Addition and Subtraction

# Multiplication:





**UNIT-IV****MEMORY AND INPUT/OUTPUT ORGANIZATION****Memory Organization:**

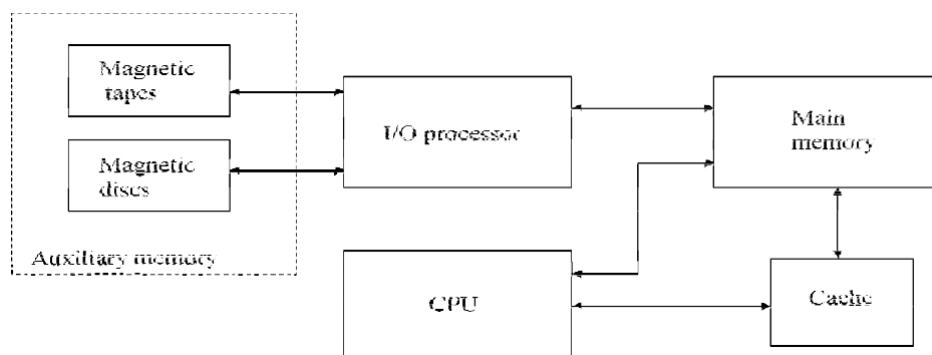
- ✚ Memory Hierarchy**
- ✚ Main Memory**
- ✚ Auxiliary Memory**
- ✚ Associative Memory**
- ✚ Cache Memory**
- ✚ Virtual Memory.**

**Input/output Organization:**

- ✚ Input-Output Interface**
- ✚ Asynchronous Data Transfer**
- ✚ Modes of Transfer**
- ✚ Priority Interrupt**
- ✚ Direct Memory Access (DMA).**

## MEMORY HIERARCHY

- ✓ The memory unit is an essential component in any digital computer since it is needed for storing programs and data. A very small computer with a limited application may be able to fulfill its intended task without the need of additional storage capacity.
- ✓ Most general-purpose computers would run more efficiently if they were equipped with additional storage beyond the capacity of the main memory.
- ✓ It is more economical to use low-cost storage devices to serve as a backup for storing the information that is not currently used by the CPU.
- ✓ The memory unit that communicates directly with the CPU is called the **main memory**. Devices that provide backup storage are called **auxiliary memory**. The most common auxiliary memory devices used in computer systems are magnetic disks and tapes. They are used for storing system programs, large data files, and other backup information. Only programs and data currently needed by the processor reside in main memory. All other information is stored in auxiliary memory and transferred to main memory when needed.
- ✓ The memory hierarchy system consists of all storage devices employed in a computer system from the slow but high-capacity auxiliary memory to a relatively faster main memory, to an even smaller and faster cache memory accessible to the high-speed processing logic.



Memory hierarchy in computer system

- ✓ The main memory occupies a central position by being able to communicate directly with the CPU and with auxiliary memory devices through an I/O processor.
- ✓ When programs not residing in main memory are needed by the CPU, they are brought in from auxiliary memory. Programs not currently needed in main memory are transferred into auxiliary memory to provide space for currently used programs and data.
- ✓ A special very-high speed memory called a **cache** is sometimes used to increase the speed of processing by making current programs and data available to the CPU at a rapid rate. The cache memory is employed in computer systems to compensate for the speed differential between main memory access time and processor logic.

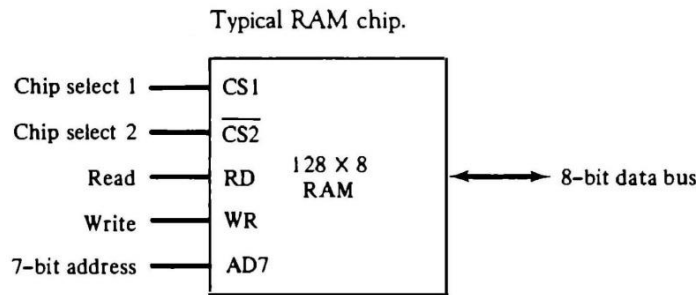
- ✓ CPU logic is usually faster than main memory access time, with the result that processing speed is limited primarily by the speed of main memory.
- ✓ A technique used to compensate for the mismatch in operating speeds is to employ in extremely fast, small cache between the CPU and main memory whose access time is close to processor logic clock cycle time.
- ✓ The 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 from 256 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**.

## 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 shorter 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:  $1024 \times 8$  memory can be constructed with  $128 \times 8$  RAM chips and  $512 \times 8$  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.



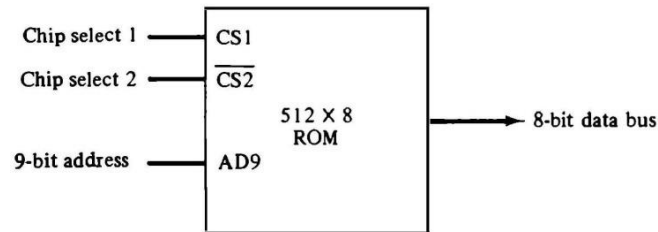
(a) Block diagram

CS1	$\overline{\text{CS2}}$	RD	WR	Memory function	State of data bus
0	0	x	x	Inhibit	High-impedance
0	1	x	x	Inhibit	High-impedance
1	0	0	0	Inhibit	High-impedance
1	0	0	1	Write	Input data to RAM
1	0	1	x	Read	Output data from RAM
1	1	x	x	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  $\text{CS1} = 1$  and  $\overline{\text{CS2}} = 0$ .
- ✓ 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  $\text{CS1} = 1$  and  $\overline{\text{CS2}} = 0$ , 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 from 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.

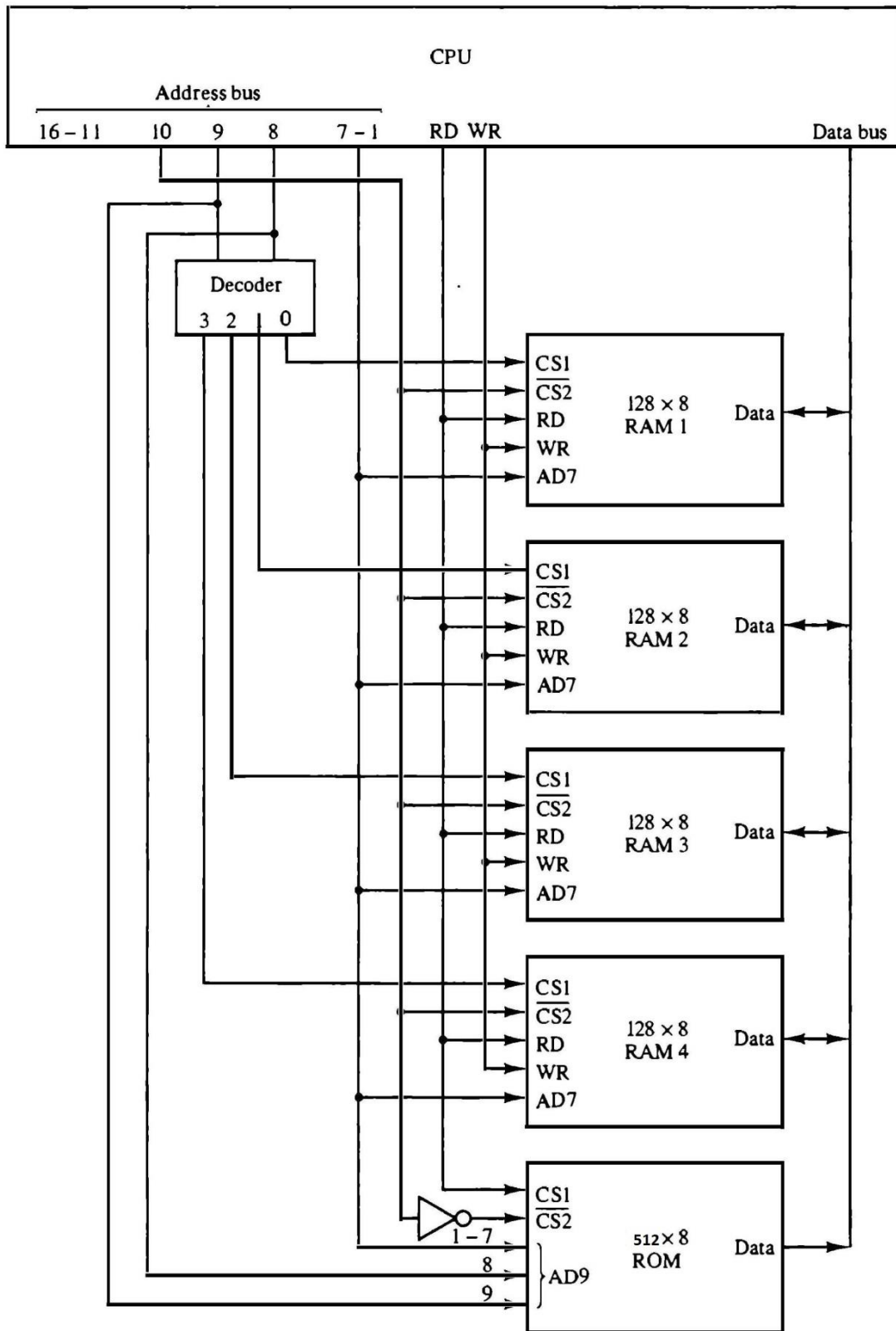
Memory Address Map for Microcomputer

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  $2 \times 4$  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.

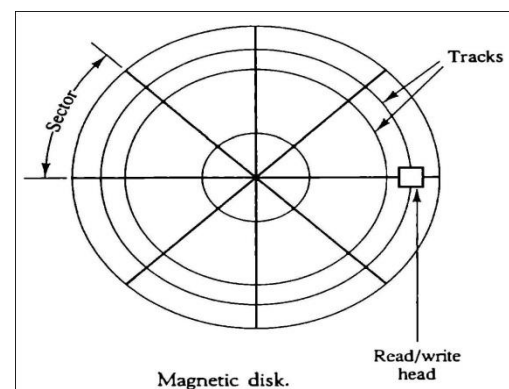


## AUXILIARY MEMORY:

- ✓ 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.



- ✓ 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**

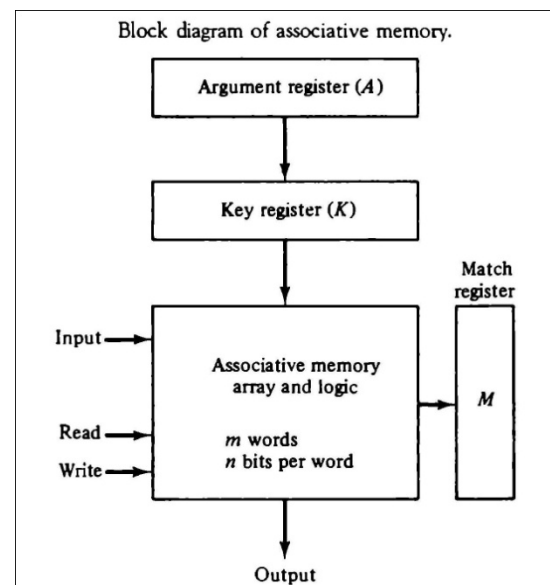
- ✓ The Magnetic tape itself is a strip of plastic coated with a magnetic recording medium. Bits are recorded as magnetic spots on the tape along several tracks. Usually, seven or nine bits are recorded simultaneously to form a character together with a parity bit.
- ✓ Read/write heads are mounted one 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.

## 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.
- ✓ 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.

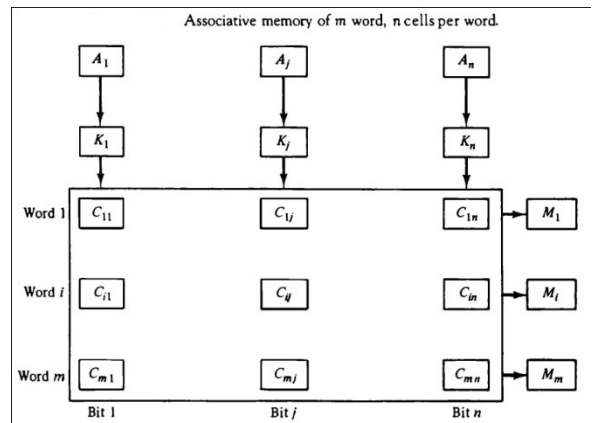
A 101 111100

K 111 000000

Word 1 100 111100 no match

Word 2 101 000001 match

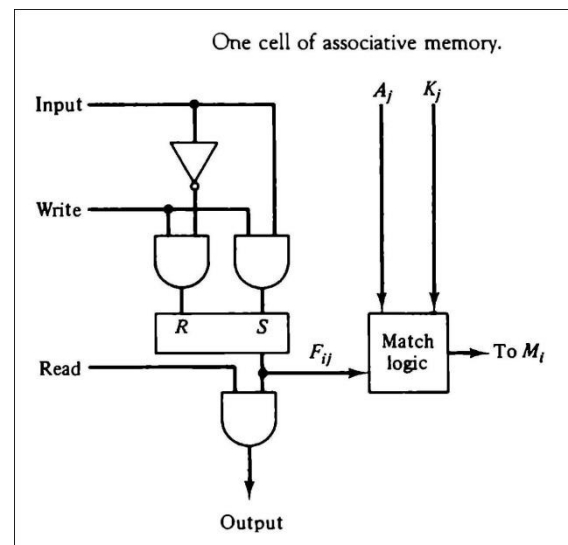
- ✓ The relation between the memory array and external registers in an associative memory is shown in Fig.



- ✓ 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  $C_{ij}$  is the cell for bit j in word i.
- ✓ A bit  $A_j$  in the argument register is compared with all the bits in column j of the array provided that  $K_j = 1$ . This is done for all columns  $j = 1, 2, \dots, n$ .
- ✓ If a match occurs between all the unmasked bits of the argument and the bits in word i, the corresponding bit  $M_i$  in the match register is set to 1. If one or more unmasked bits of the argument and the word do not match,  $M_i$  is cleared to 0

- ✓ The internal organization of a typical cell  $C_{ij}$  is shown in Fig.

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



**MATCH LOGIC**

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

$$x_j = A_j F_{ij} + A'_j F'_{ij}$$

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

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

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

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

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

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

$$M_i = (x_1 + K'_1) (x_2 + K'_2) (x_3 + K'_3) \dots (x_n + K'_n)$$

- ✓ Each term in the expression will be equal to 1 if its corresponding  $K_j = 0$ . If  $K_j = 1$ , the term will be either 0 or 1 depending on the value of  $x_j$ . A match will occur and  $M_i$  will be equal to 1 if all terms are equal to 1.
- ✓ If we substitute the original definition of  $x_j$ , the Boolean function above can be expressed as follows:

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

- ✓ The circuit for matching one word is shown in Fig. Each cell requires two AND gates and one OR gate. The inverters for  $A_j$  and  $K_j$  are needed once for each column and are used for all bits in the column. The output of all OR gates in the cells of the same word go to the input of a common AND gate to generate the match signal for  $M_i$ .  $M_i$  will be logic 1 if a match occurs and 0 if no match occurs. Note that if the key register contains all 0's, output  $M_i$  will be a 1

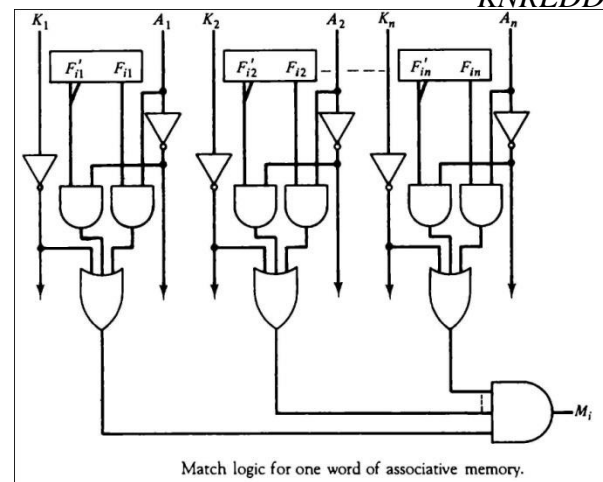
irrespective of the value of A or the word. This occurrence must be avoided during normal operation.

### 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  $M_i$  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  $M_i$  directly to the read line in the same word position (instead of the M register), the content of the matched word will be presented automatically at the output lines and no special read command signal is needed.

### WRITE OPERATION

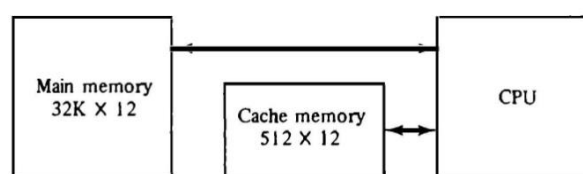
- ✓ An associative memory must have a write capability for storing the information to be searched.
- ✓ Writing in an associative memory can take different forms, depending on the application. If the entire memory is loaded with new information at once prior to a search operation then the writing can 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.



Match logic for one word of associative memory.

## CACHE MEMORY

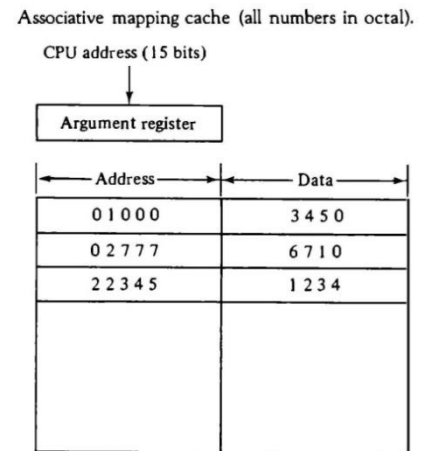
- Locality of Reference: The references to memory at any given time interval tends to be confined within a localized area.
- When a program loop is executed, the CPU repeatedly refers to the set of instructions in memory that constitute the loop.
- Every time a given subroutine is called, its set of 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.

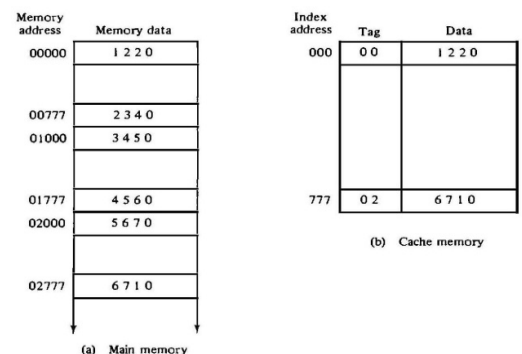
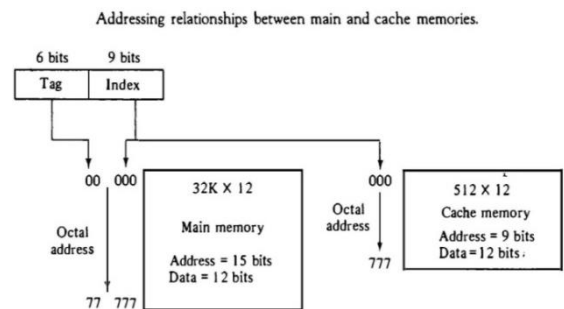
**ASSOCIATIVE MAPPING**

- The faster and most flexible cache organization use an associative memory. 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.
- 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.
- Direct mapping uses RAM instead of CAM.
- The n-bit memory address is divided into two fields: k bits for the index field and n-k bits for the tag field. The direct mapping cache organization uses the n-bit address to access the main memory and the k-bit index to access the cache.
- The internal organization of the words in the cache memory is as shown in Fig
- 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.

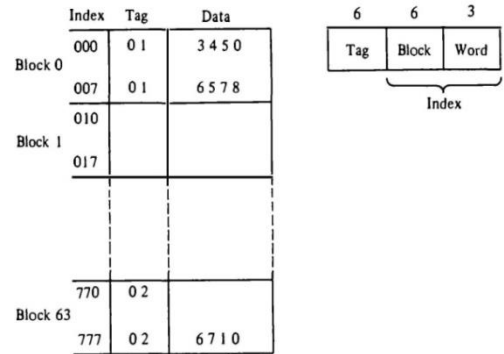


Direct mapping cache organization.



- The tag field of the CPU address is compared with the tag in the word read from the cache. If the two tags match, there is a hit and the desired data word is in cache. If 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.
- Suppose that the CPU now wants to access the word at address 02000. The index address is 000, so it is sued to access the cache. The two tags are then compared. The cache tag is 00 but the address tag is 02, which does not produce a match. Therefore, the main memory is accessed and the data word 5670 is transferred to the CPU. The cache word at index address 000 is then replaced with a tag of 02 and data of 5670.
- The direct-mapping uses a block size of one word. The same organization but using a block size of 8 words is shown in Fig.

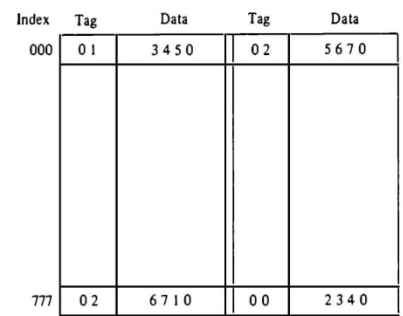
- The index field is now divided into two parts: the block field and the word field. The tag field stored within the cache is common to all eight words of the same block.
- Every time a miss occurs, an entire block of eight words must be transferred from main memory to cache memory. Although this takes extra time, the hit ratio will most likely improve with a larger block size because of the sequential nature of computer programs.



Direct mapping cache with block size of 8 words.

**SET-ASSOCIATIVE MAPPING**

- 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.
- Each index address refers to two data words and their associated tags. Each tag requires six bits and each data word has 12 bits, so the word length is  $2(6 + 12) = 36$  bits. An index address of nine bits can accommodate 512 words. Thus the size of cache memory is  $512 \times 36$ . It can accommodate 1024



Two-way set-associative mapping cache.

- 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 hit ratio will improve as the set size increases because more words with the same index but different tags can reside in cache.
- When a miss occurs in a set-associative cache and the set is full, it is necessary to replace one of the tag-data items with a new value. The most common replacement algorithms used are: random replacement, first-in first out (FIFO), and least recently used (LRU).

#### WRITING INTO CACHE

- An important aspect of cache organization is concerned with memory write requests. If the operation is a write, there are two ways that the system can proceed.
- The simplest and most commonly used procedure is to up data main memory with every memory write operation, with cache memory being updated in parallel if it contains the word at the specified address. This is called the ***write-through method***. This method has the advantage that main memory always contains the same data as the cache,. This characteristic is important in systems with direct memory access transfers.
- The second procedure is called the ***write-back method***. In this method only the cache location is updated during a write operation. The location is then marked by a flag so that later when the words are removed from the cache it is copied into main memory. The reason for the write-back method is that during the time a word resides in the cache, it may be updated several times; however, as long as the word remains in the cache, it does not matter whether the copy in main memory is out of date, since requests from the word are filled from the cache. It is only when the word is displaced from the cache that an accurate copy need be rewritten into main memory.

#### CACHE INITIALIZATION

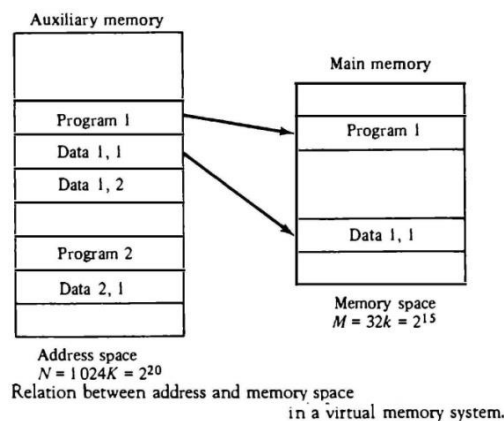
- The cache is initialized when power is applied to the computer or when the main memory is loaded with a complete set of programs from auxiliary memory. After initialization the cache is considered to be empty, built in effect it contains some non-valid data. It is customary to include with each word in cache a valid bit to indicate whether or not the word contains valid data.
- The cache is initialized by clearing all the valid bits to 0. The valid bit of a particular cache word is set to 1 the first time this word is loaded from main memory and stays set unless the cache has to be initialized again. The introduction of the valid bit means that a word in cache is initialization condition has the effect of forcing misses from the cache until it fills with valid data.

## VIRTUAL MEMORY

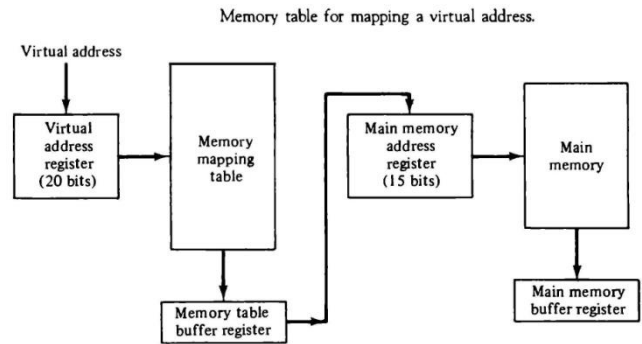
- In a memory hierarchy system, programs and data are brought into main memory as they are needed by the CPU.
- Virtual memory is a concept used in some large computer systems that permit the user to construct programs as though a large memory space were available, equal to the totality of auxiliary memory.
- A virtual memory system provides a mechanism for translating program-generated addresses into correct main memory locations. This is done dynamically, while programs are being executed in the CPU. The translation or mapping is handled automatically by the hardware by means of a mapping table.

## ADDRESS SPACE AND MEMORY SPACE

- An address used by a programmer will be called a virtual address, and the set of such addresses the address space.
- An address in main memory is called a location or physical address. The set of such locations is called the memory space.
- In most computers the address and memory spaces are identical. The address space is allowed to be larger than the memory space in computers with virtual memory.
- As an illustration, consider a computer with a main-memory capacity of 32K words ( $K = 1024$ ). Fifteen bits are needed to specify a physical address in memory since  $32K = 2^{15}$ . Suppose that the computer has available auxiliary memory for storing  $220 = 1024K$  words. Thus auxiliary memory has a capacity for storing information equivalent to the capacity of 32 main memories.
- Denoting the address space by  $N$  and the memory space by  $M$ , we then have for this example  $N = 1024K$  and  $M = 32K$ .
- In a multiprogram computer system, programs and data are transferred to and from auxiliary memory and main memory based on demands imposed by the CPU. Suppose that program 1 is currently being executed in the CPU. Program 1 and a portion of its associated data is moved from auxiliary memory into main memory as shown in Fig.



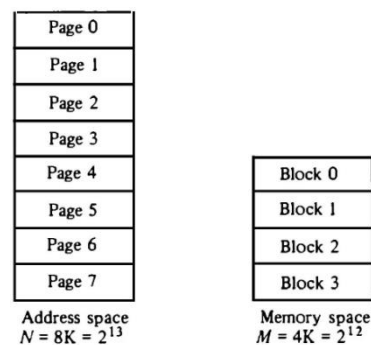
- Portions of programs and data need not be in contiguous locations in memory since information is being moved in and out, and empty spaces may be available in scattered locations in memory.
- The address field of an instruction code will consist of 20 bits but physical memory addresses must be specified with only 15 bits.
- A table is then needed, to map a virtual address of 20 bits to a physical address of 15 bits. The mapping is a dynamic operation, which means that every address is translated immediately as a word is referenced by CPU.
- The mapping table may be stored in a separate memory or in main memory. In the first case, an additional memory unit is required as well as one extra memory access time. In the second case, the table takes space from main memory and two accesses to memory are required with the program running at half speed. A third alternative is to use an associative memory.



**ADDRESS MAPPING USING PAGES**

- The table implementation of the address mapping is simplified if the information in the address space and the memory space are each divided into groups of fixed size. The physical memory is broken down into groups of equal size called **blocks**, which may range from 64 to 4096 words each. The term **page** refers to groups of address space of the same size.
- A page refers to the organization of address space, while a block refers to the organization of memory space. The programs are also considered to be split into pages. Portions of programs are moved from auxiliary memory to main memory in records equal to the size of a page. The term “page frame” is sometimes used to denote a block.

Consider a computer with an address space of 8K and a memory space of 4K. If we split each into groups of 1K words we obtain eight pages and four blocks as shown in Fig.

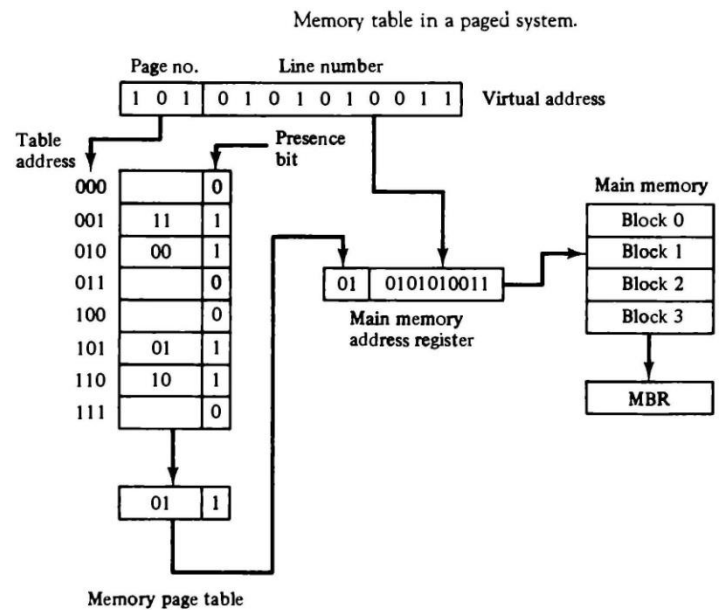


Address space and memory space split into groups of 1K words.

- At any given time, up to four pages of address space may reside in main memory in any one of the four blocks.
- The mapping from address space to memory space is facilitated if each virtual address is considered to be represented by two numbers: a page number address and a line within the page.
- In a computer with  $2^p$  words per page, p bits are used to specify a line address and the remaining high-order bits of the virtual address specify the page number.

- Note that the line address in address space and memory space is the same; the only mapping required is from a page number to a block number
- The organization of the memory mapping table in a paged system is shown in Fig.

- The memory-page table consists of eight words, one for each page. The address in the page table denotes the page number and the content of the word gives the block number where that page is stored in main memory. The table shows that pages 1, 2, 5 and 6 are now available in main memory in blocks 3, 0, 1, and 2, respectively. A presence bit in each location indicates whether the page has been transferred from auxiliary memory into main memory. A 0 in the presence

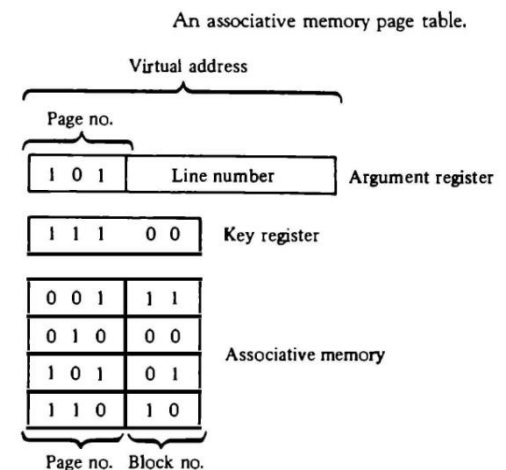


bit indicates that this page is not available in main memory. The CPU references a word in memory with a virtual address of 13 bits. The three high-order bits of the virtual address specify a page number and also an address for the memory-page table. The content of the word in the memory page table at the page number address is read out into the memory table buffer register. If the presence bit is a 1, the block number thus read is transferred to the two high-order bits of the main memory address register. The line number from the virtual address is transferred into the 10 low order bits of the memory address register. A read signal to main memory transfers the content of the word to the main memory buffer register ready to be used by the CPU. If the presence bit in the word read from the page table is 0, it signifies that the content of the word referenced by the virtual address does not reside in main memory. A call to the operating system is then generated to fetch the required page from auxiliary memory and place it into main memory before resuming computation.

#### ASSOCIATIVE MEMORY PAGE TABLE

- A random-access memory page table is inefficient with respect to storage utilization.
- In general, system with  $n$  pages and  $m$  blocks would require a memory page table of  $n$  locations of which up to  $m$  blocks will be marked with block numbers and all others will be empty.
- Consider an address space of 1024K words and memory space of 32K words. If each page or block contains 1K words, the number of pages is 1024 and the number of blocks 32. The capacity of the memory-page table must be 1024 words and only 32 locations may have a presence bit equal to 1. At any given time, at least 992 locations will be empty and not in use.

- A more efficient way to organize the page table would be to construct it with a number of words equal to the number of blocks in main memory.
- This method can be implemented by means of an associative memory with each word in memory containing a page number together with its corresponding block number. The page field in each word is compared with the page number in the virtual address. If a match occurs, the word is read from memory and its corresponding block number is extracted.
- Consider the case of eight pages and four blocks.
- Each entry in the associative memory array consists of two fields. The first three bits specify a field for storing the page number. The last two bits constitute a field for storing the block number. The virtual address is placed in the argument register. The page number bits in the argument are compared with all page numbers in the page field of the associative memory. If the page number is found, the 5-bit word is read out from memory. The corresponding block number, being in the same word, is transferred to the main memory address register. If no match occurs, a call to the operating system is generated to bring the required page from auxiliary memory.



## PAGE REPLACEMENT

- A virtual memory system is a combination of hardware and software techniques. The memory management software system handles all the software operations for the efficient utilization of memory space. It must decide (1) which page in main memory ought to be removed to make room for a new page, (2) when a new page is to be transferred from auxiliary memory to main memory, and (3) where the page is to be placed in main memory.
- The hardware mapping mechanism and the memory management software together constitute the architecture of a virtual memory.
- When a program starts execution, one or more pages are transferred into main memory and the page table is set to indicate their position. The program is executed from main memory until it attempts to reference a page that is still in auxiliary memory. This condition is called **page fault**. When page fault occurs, the execution of the present program is suspended until the required page is brought into main memory. Since loading a page from auxiliary memory to main memory is basically an I/O operation, the operating system assigns this task to the I/O processor.

- In the meantime, controls transferred to the next program in memory that is waiting to be processed in the CPU. Later, when the memory block has been assigned and the transfer completed, the original program can resume its operation.
- When a page fault occurs in a virtual memory system, it signifies that the page referenced by the CPU is not in main memory. A new page is then transferred from auxiliary memory to main memory. If main memory is full, it would be necessary to remove a page from a memory block to make room for the new page. The policy for choosing pages to remove is determined from the replacement algorithm that is used.
- Two of the most common replacement algorithms used are the first-in first-out (FIFO) and the least recently used (LRU). The FIFO algorithm selects for replacement the page that has been in memory the longest time. Each time a page is loaded into memory, its identification number is pushed into a FIFO stack. FIFO will be full whenever memory has no more empty blocks. When a new page must be loaded, the page least recently brought in is removed. The page to be removed is easily determined because its identification number is at the top of the FIFO stack. The FIFO replacement policy has the advantage of being easy to implement. It has the disadvantages that under certain circumstances pages are removed and loaded from memory too frequently.
- The LRU policy is more difficult to implement but has been more attractive on the assumption that the least recently used page is a better candidate for removal than the least recently loaded pages in FIFO. The LRU algorithm can be implemented by associating a counter with every page that is in main memory. When a page is referenced, its associated counter is set to zero. At fixed intervals of time, the counters associated with all pages presently in memory are incremented by 1. The least recently used page is the page with the highest count. The counters are often called aging registers, as their count indicates their age, that is, how long ago their associated pages have been referenced.

## PERIPHERAL DEVICES

- The input-output subsystem of a computer, referred to as I/O, provides an efficient mode of communication between the central system and the outside environment
- Input or output devices attached to the computer are also called peripherals.

- Input Devices

- Keyboard

- Optical input devices

- Card Reader
    - Bar code reader
    - Digitizer
    - Optical Mark Reader

- Screen Input Devices

- Touch Screen
    - Light Pen
    - Mouse

- Analog Input Devices

- Output Devices

- CRT

- Printer (Impact, Ink Jet, Laser, Dot Matrix)

- Plotter

- Speakers

- Input and output devices that communicate with people and the computer are usually involved in the transfer of alphanumeric information to and from the device and the computer is **ASCII** (American Standard Code for Information Interchange).
- ASCII is a 7 bit code, but most computers manipulate an 8-bit quantity as a single unit called a byte. Therefore, ASCII characters most often are stored one per byte.



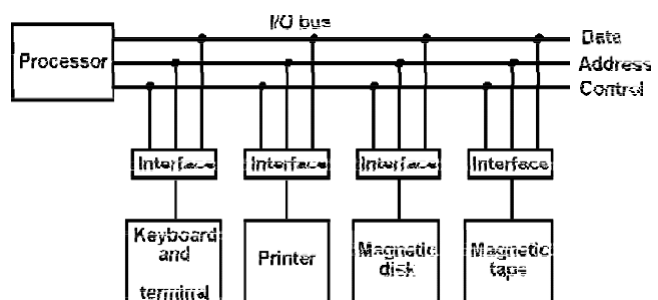
## INPUT-OUTPUT INTERFACE

- 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.

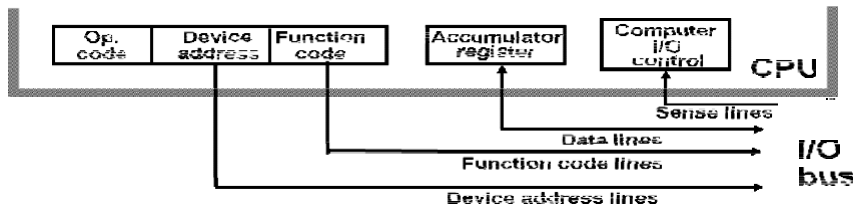
- Each peripheral device has associated with it an interface unit. Each interface decodes the address and control received from the I/O bus, interprets them for the peripheral, and provides signals for the peripheral controller. It also synchronizes the data



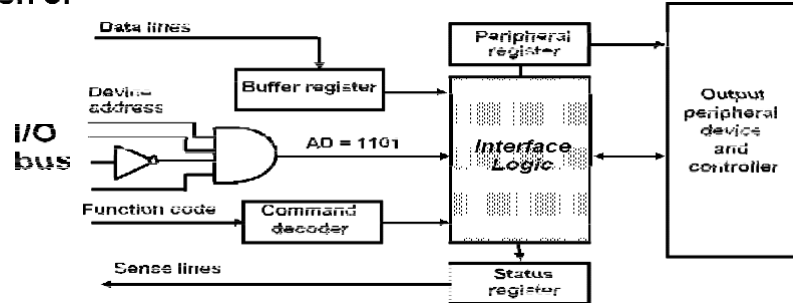
flow and supervises the transfer between peripheral and processor. Each peripheral has its own controller that operates the particular electromechanical device.

- 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.

### Connection of I/O Bus to CPU



### Connection of I/O Bus to One 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.

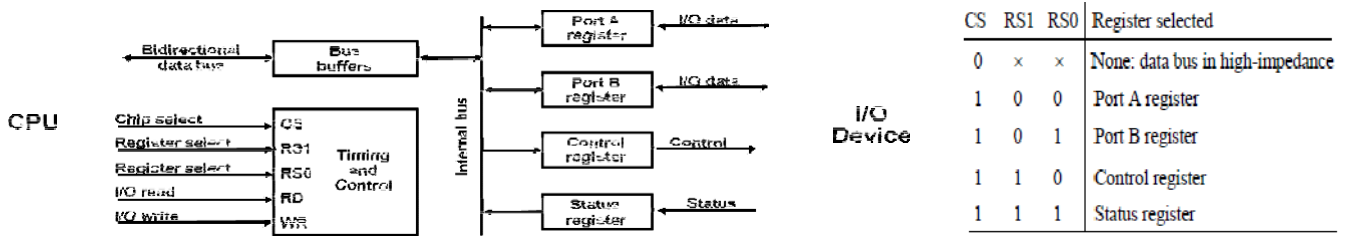
In the first method, the computer has independent sets of data, address, and control buses, one for accessing memory and the other for I/O. This is done in computers that provide a separate I/O processor (IOP) in addition to the central processing unit (CPU). The memory communicates with both the CPU 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



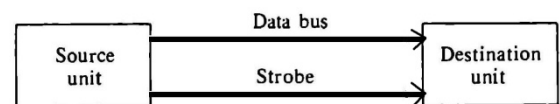
- It consists of two data registers called ports, a control register, a status register, bus buffers, and timing and control circuits. The interface communicates with the CPU through the data bus.
- The chip select and register select inputs determine the address assigned to the interface. The I/O read 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

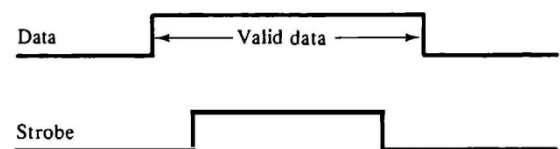
- The internal operations in a digital system are synchronized by means of clock pulses supplied by a common pulse generator.
- If the registers in the interface share a common clock with the CPU registers, the transfer between the two units is said to be synchronous.
- In most cases, the internal timing 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.
- 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.



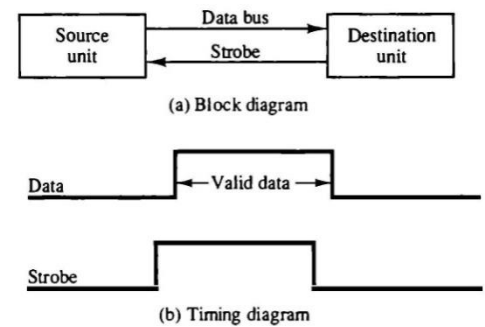
(a) Block diagram



(b) Timing diagram

Source-initiated strobe for data transfer.

- 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



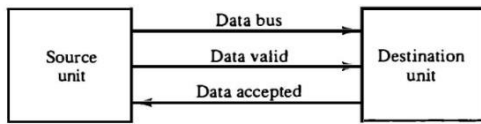
Destination-initiated strobe for data transfer.

- 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

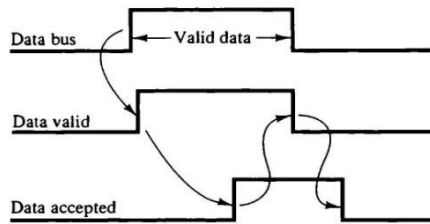
### HANDSHAKING

- 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

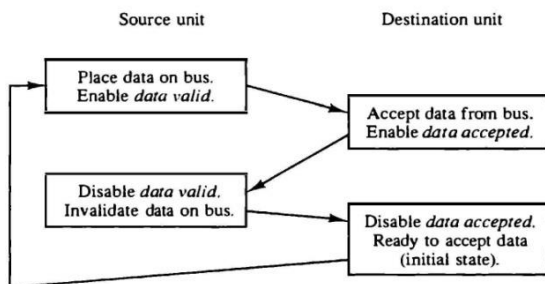
counting time when the unit enables one of its handshaking control signals. If the return handshake signal does not respond within a given time period, the unit assumes that an error has occurred



(a) Block diagram



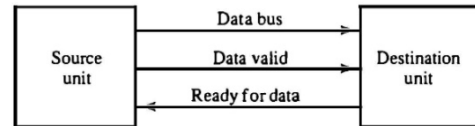
(b) Timing diagram



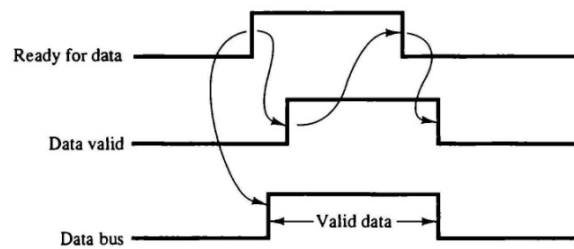
(c) Sequence of events

Source-initiated transfer using handshaking.

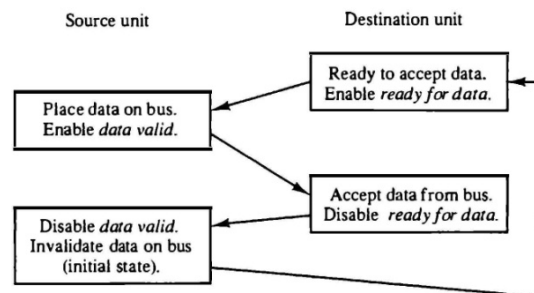
Destination-initiated transfer using handshaking.



(a) Block diagram



(b) Timing diagram

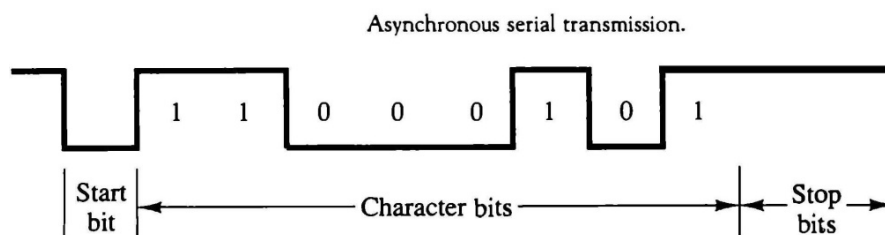


(c) Sequence of events

## 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.

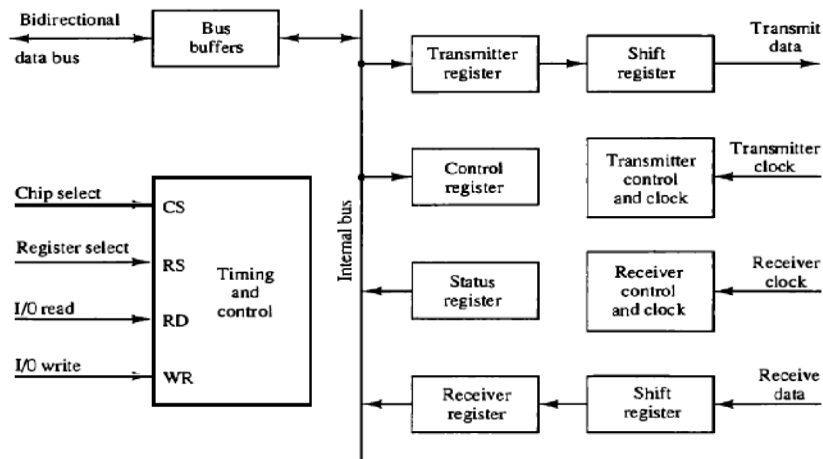


- A transmitted character can be detected by the receiver from knowledge of the transmission rules:
  1. When a character is not being sent, the line is kept in the 1-state.
  2. The initiation of a character transmission is detected from the start bit, which is always 0.
  3. The character bits always 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	x	x	None: data bus in high-impedance
1	0	WR	Transmitter register
1	1	WR	Control register
1	0	RD	Receiver register
1	1	RD	Status register

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 1'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  $4 \times 4$  FIFO buffer is shown in Fig. 11-9. It consists of four 4-bit registers  $R_i$ ,  $i = 1, 2, 3, 4$ , and a control register with

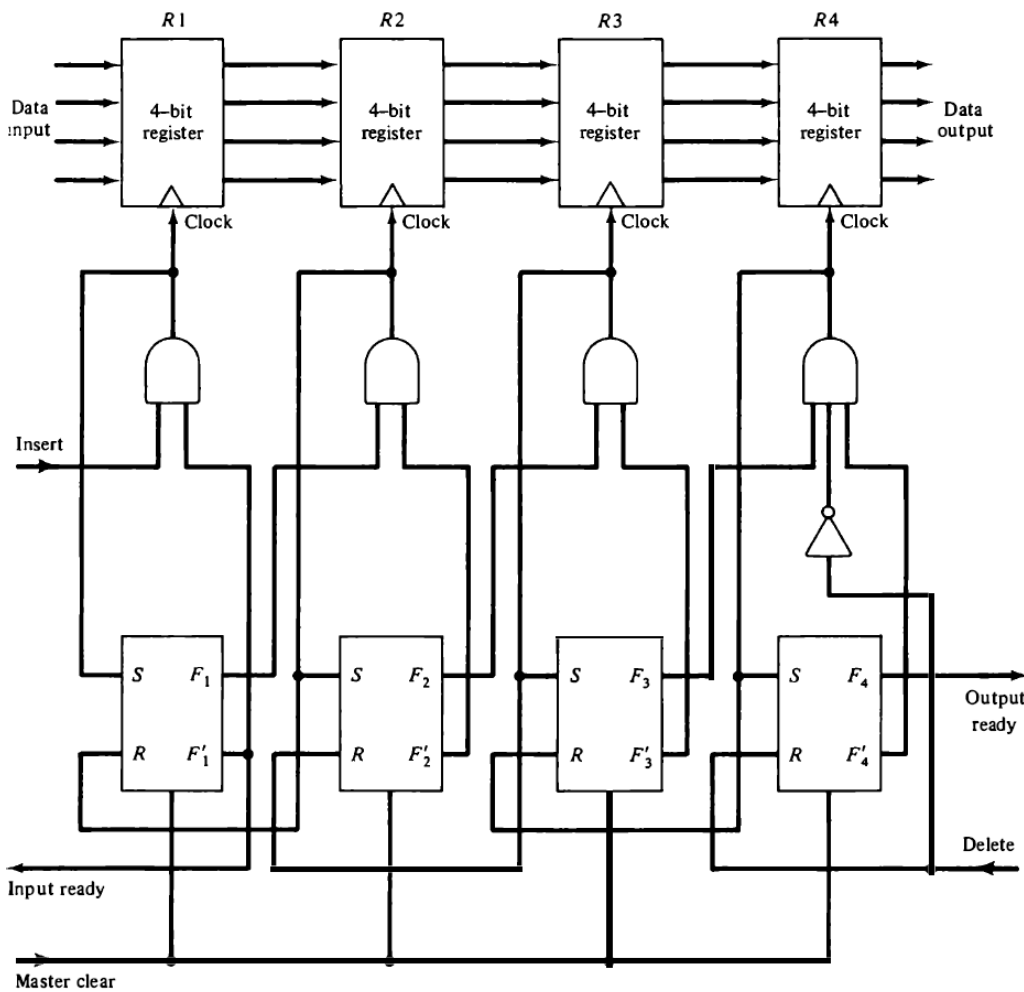


Figure 11-9 Circuit diagram of  $4 \times 4$  FIFO buffer.

**MODES OF TRANSFER**

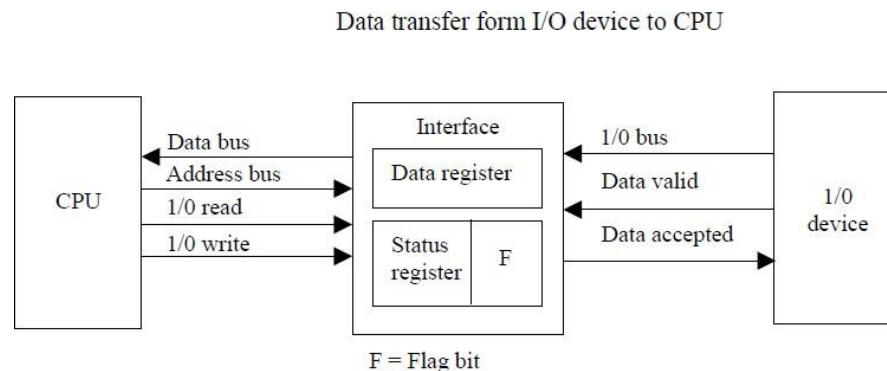
- Binary information received from an external device is usually stored in memory for later processing. Information transferred from the central computer into an external device originates in the memory unit. The CPU merely executes the I/O instructions and may accept the data temporarily, but the ultimate 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.



- 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:

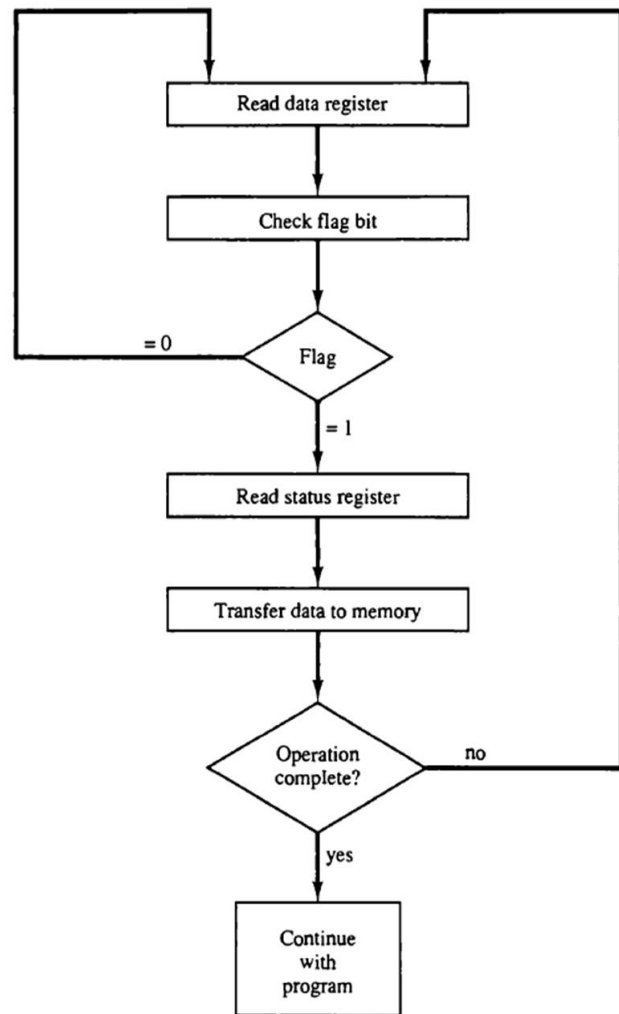
1. Read the status register.
2. Check the status of the flag bit and branch to step 1 if not set or to step 3 if set.
3. Read the data register.

- 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 from 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



Flowchart for CPU program to input data.

required I/O transfer. The way that the processor chooses the branch address of the service routine varies from one unit to another. In principle, there are two methods for accomplishing this. One is called vectored interrupt and the other, non 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**

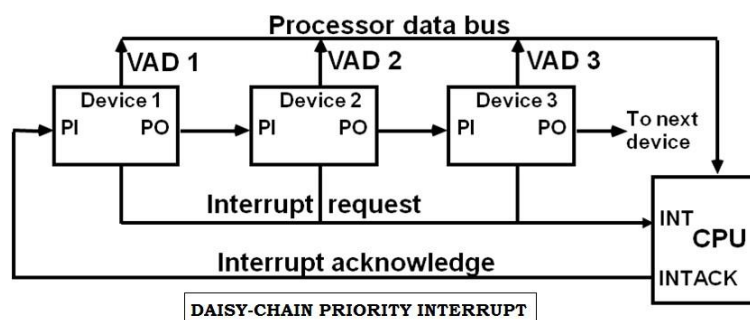
- The previous discussion was concerned with the basic hardware needed to interface I/O devices to a computer system. A computer must also have software routines for controlling peripherals and for transfer of data between the processor and peripherals. I/O routines must issue control commands to activate 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.





- 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.

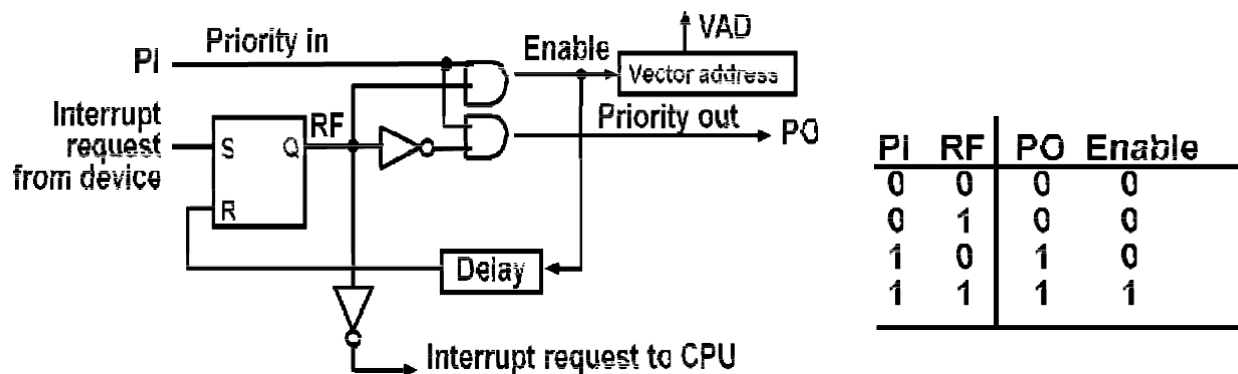
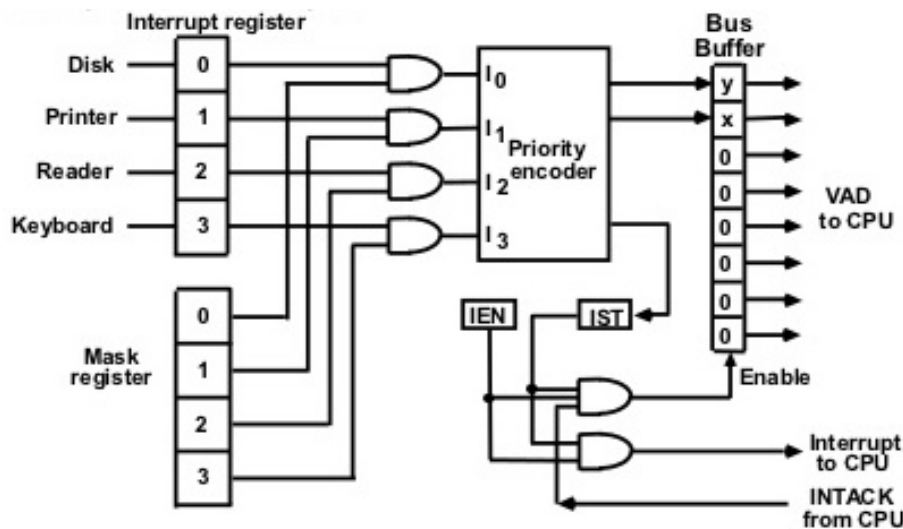


Fig: One stage of the daisy- chain priority arrangement

- 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	$x = I_0' I_1'$ $y = I_0' I_1 + I_0' I_2'$ $(IST) = I_0 + I_1 + I_2 + I_3$
0	1	d	d	0	1	1	
0	0	1	d	1	0	1	
0	0	0	1	1	1	1	
0	0	0	0	d	d	0	

### Interrupt Cycle

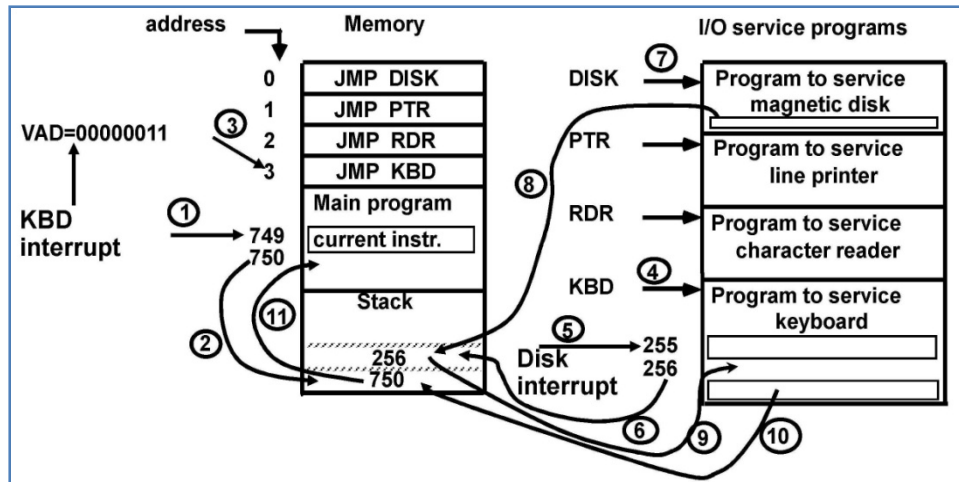
- The interrupt enable flip-flop IEN can be set or cleared by program instructions.
- When IEN is cleared, the interrupt request coming from IST is neglected by the CPU.
- The program-controlled IEN bit allows the programmer to choose whether to use the interrupt facility. If an instruction to clear 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
IEN ← 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.



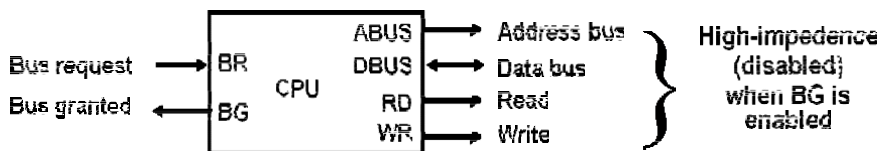
### Initial and Final Operations

- 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]  $IST \leftarrow 0$
  - [3] Save contents of CPU registers
  - [4]  $IEN \leftarrow 1$
  - [5] Go to Interrupt Service Routine
- **Final Sequence**
  - [1]  $IEN \leftarrow 0$
  - [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  $IEN \leftarrow 1$
- 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.

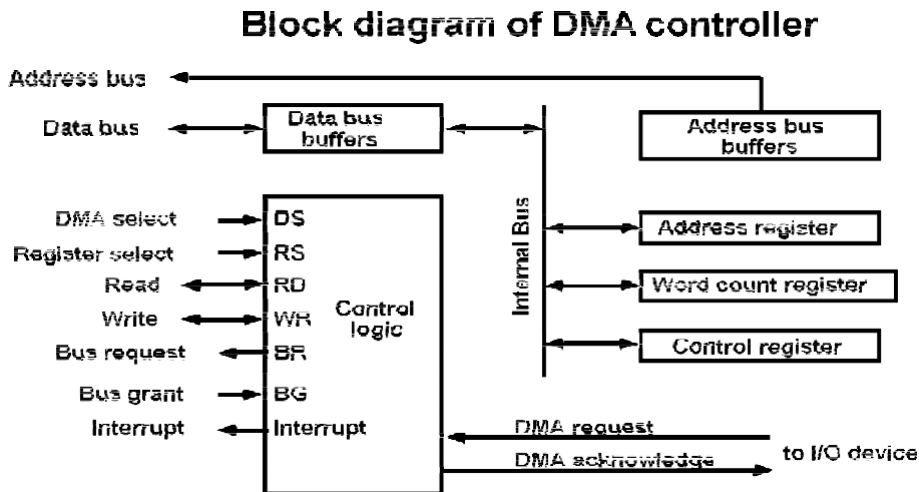
### CPU bus signals for DMA transfer



- 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



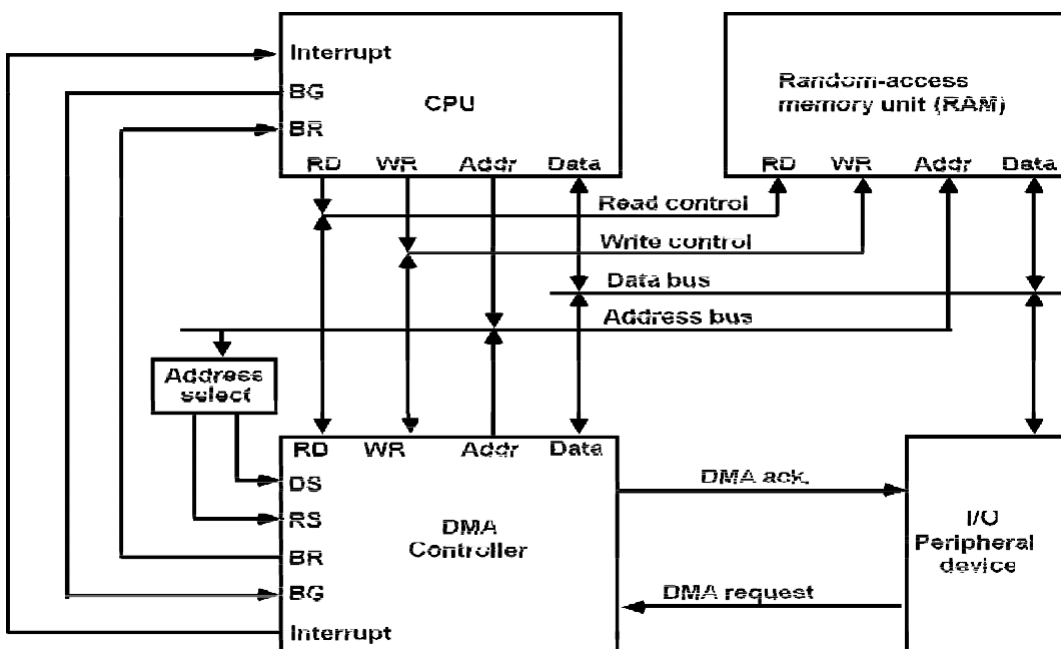
- The unit communicates with the CPU via the data bus and control lines. The registers in the DMA are selected by the CPU through the address bus by enabling the DS (DMA select) and RS (register select) inputs. The RD (read) and WR (write) inputs are bidirectional.
- When the BG (bus grant) input is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



## **UNIT-V**

### **PIPELINE AND MULTIPROCESSORS**

#### **Pipeline**

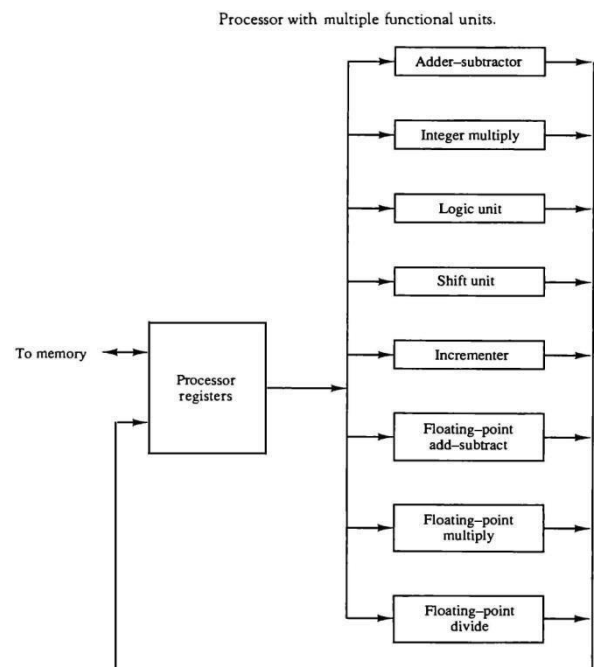
- + Parallel Processing**
- + Pipelining**
- + Arithmetic Pipeline**
- + Instruction Pipeline.**

#### **Multiprocessors**

- + Characteristics of Multiprocessors**
- + Interconnection Structures**
- + Inter Processor Arbitration**
- + Inter Processor Communication and Synchronization**

## 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.



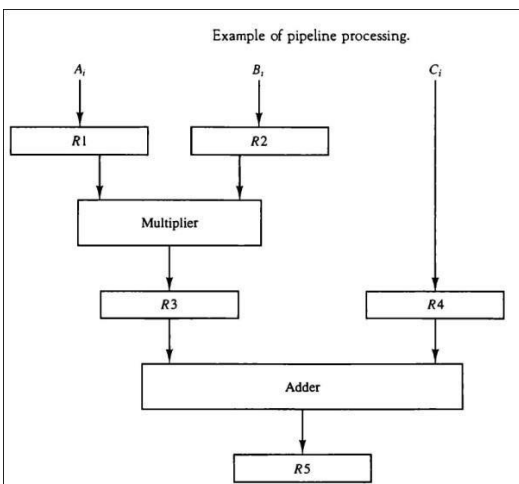
# PIPELINING

- ✓ 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:  $A_i * B_i + C_i$  for  $i = 1, 2, 3, \dots, 7$
- ✓ The sub operations performed in each segment are:

$$R1 \leftarrow A_i, R2 \leftarrow B_i$$

$$R3 \leftarrow R1 * R2, R4 \leftarrow C_i$$

$$R5 \leftarrow R3 + R4$$

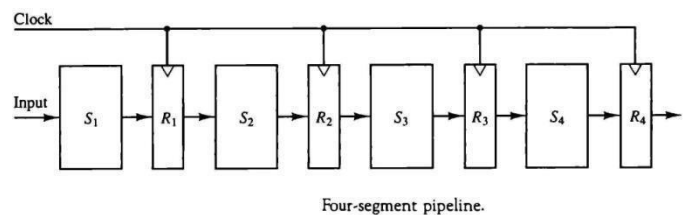


**TABLE 9-1** Content of Registers in Pipeline Example

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 suboperations of about the same complexity can be implemented by a pipeline processor
- ✓ The technique is efficient for those applications that need to repeat the same task many 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



- ✓ 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.

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

→ Clock cycles

✓ 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:

$$\text{Cycle time} = t_p = 20 \text{ ns} \quad \# \text{ of segments} = k = 4 \quad \# \text{ of tasks} = n = 100$$

✓ The pipeline system will take  $(k + n - 1)t_p = (4 + 100 - 1)20\text{ns} = 2060 \text{ ns}$

✓ Assuming that  $t_n = kt_p = 4 * 20 = 80 \text{ ns}$ ,

✓ A non-pipeline system requires  $nt_p = 100 * 80 = 8000 \text{ ns}$

✓ 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

$$X = A \times 2^a$$

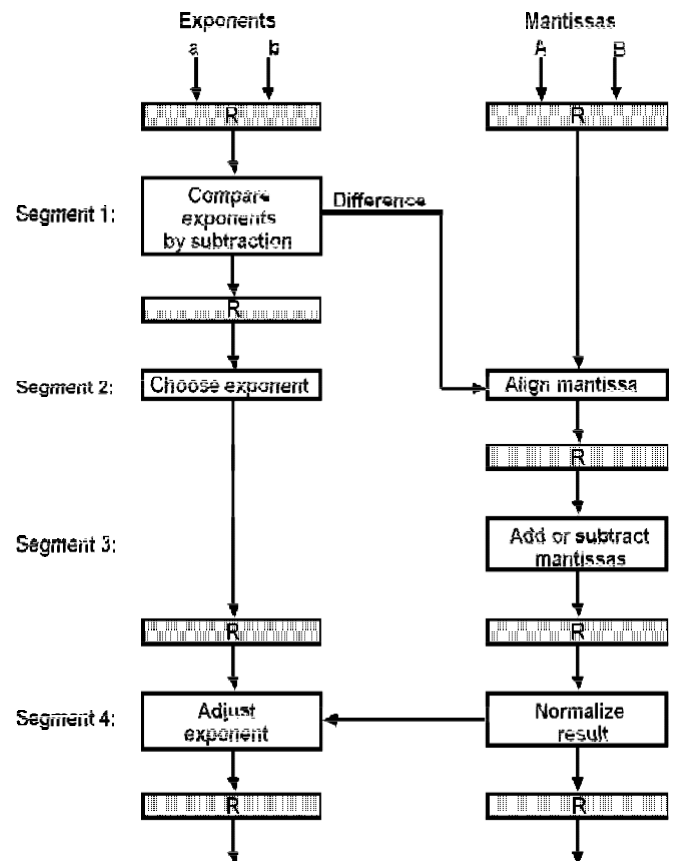
$$Y = B \times 2^b$$

- ✓ 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

$$X = 0.9504 \times 10^3 \text{ and } Y = 0.8200 \times 10^2$$

- ✓ 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 decremter in the floating-point pipeline are implemented with combinational circuits.
- ✓ Suppose that the time delays of the four segments are  $t_1 = 60 \text{ ns}$ ,  $t_2 = 70 \text{ ns}$ ,  $t_3 = 100 \text{ ns}$ ,  $t_4 = 80 \text{ ns}$ , and the interface registers have a delay of  $t_r = 10 \text{ ns}$ . The clock cycle is chosen to be  $t_p = t_3 + t_r = 110 \text{ ns}$ . An equivalent non-pipeline floating point adder-subtractor will have a delay time  $t_n = t_1 + t_2 + t_3 + t_4 + t_r = 320 \text{ ns}$ . In this case the pipelined adder has a speedup of  $320 / 110 = 2.9$  over the non-pipelined adder.



## INSTRUCTION PIPELINE

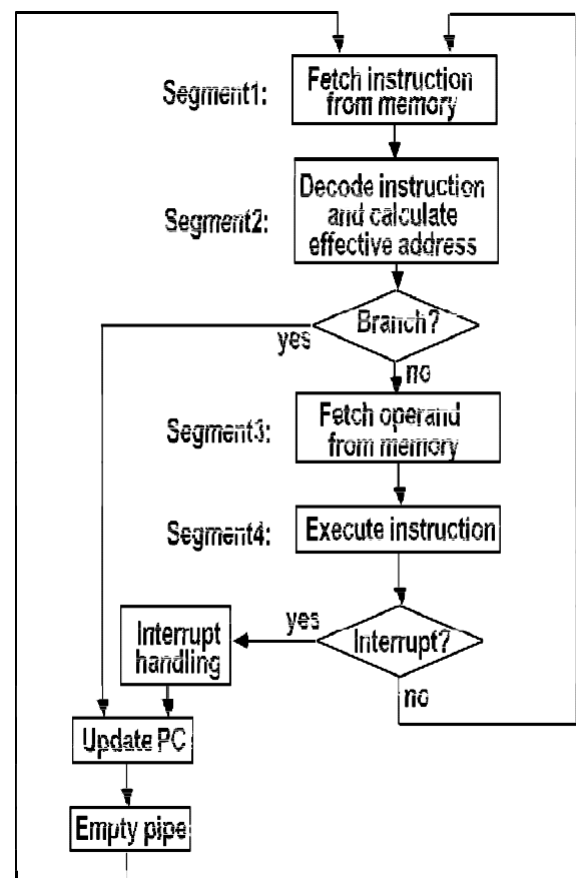
- ✓ An instruction pipeline reads consecutive instructions from memory while previous instructions are being executed in other segments
- ✓ This causes the instruction fetch and execute phases to overlap and perform simultaneous operations
- ✓ If a branch out of sequence occurs, the pipeline must be 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)			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 prefetching instructions from the predicted path.
- ✓ **Delayed branch** is used in most RISC processors so that the compiler rearranges the instructions to delay the branch.



## CHARACTERISTICS OF MULTIPROCESSORS

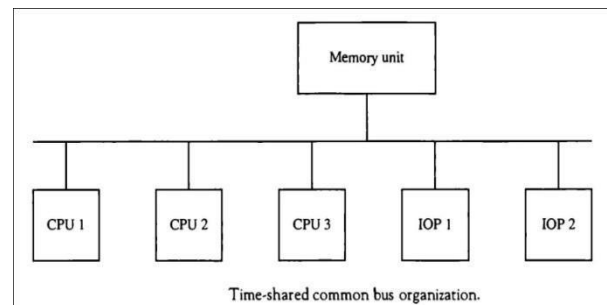
- ✓ A multiprocessor system is an interconnection of two or more CPUs with memory and input-output equipment. The term "processor" in multiprocessor can mean either a central processing unit (CPU) or an input-output processor (IOP).
- ✓ As it is most commonly defined, a multiprocessor system implies 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, IOPs connected to input-output devices, and a memory unit that may be partitioned into a number of separate modules.
- ✓ The interconnection between the components can have different physical configurations, depending on the number of transfer paths that are available between the processors and memory in a shared memory system or among the processing elements in a loosely coupled system.
- ✓ There are several physical forms available for establishing an interconnection network. Some of these schemes are:
  1. Time-shared common bus
  2. Multiport memory
  3. Crossbar switch
  4. Multistage switching network
  5. Hypercube system

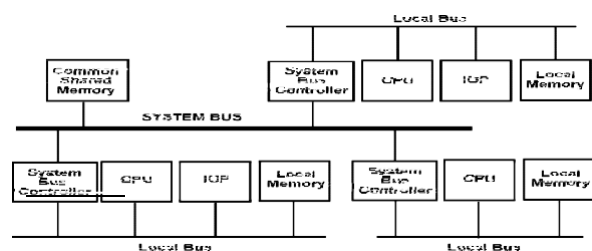
### Time-Shared Common Bus

- ✓ 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.



- ✓ 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.

### SYSTEM BUS STRUCTURE FOR MULTIPROCESSORS

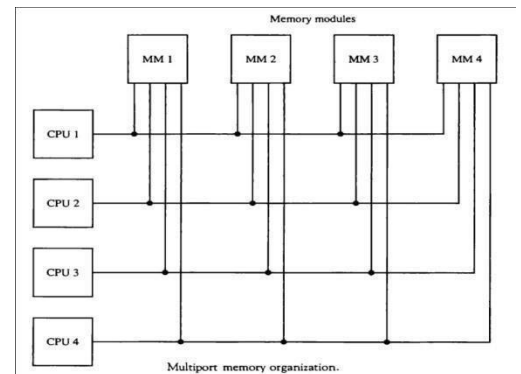


- ✓ A system bus controller links each local bus to a common system bus.

- ✓ The IO devices connected to the local IOP, as well as the local memory, are available to the local processor.
- ✓ If an IOP 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

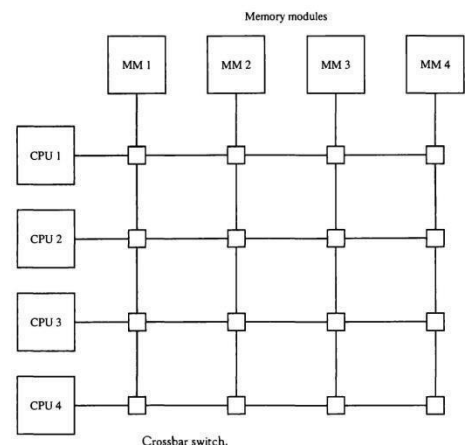
### **Multiport Memory**

- ✓ 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.
- ✓ 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.

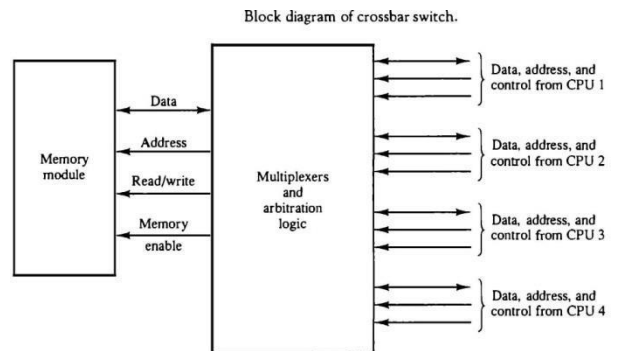


### **Crossbar Switch**

- ✓ 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.

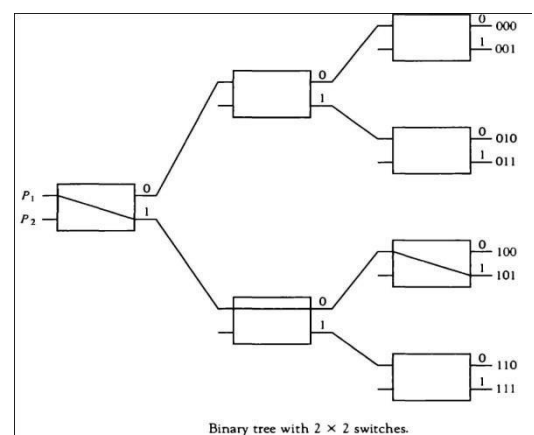
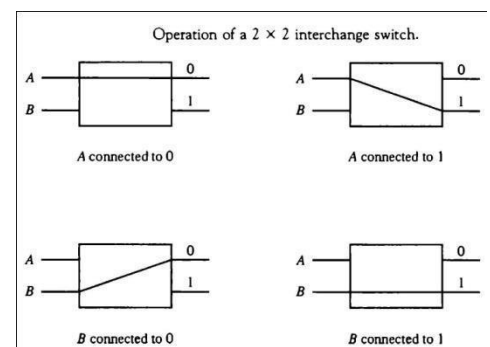


- ✓ It examines the address that is placed in the bus to determine whether its particular module is being addressed.
- ✓ It also resolves multiple requests for access to the same memory module on a predetermined priority basis.
- ✓ The functional design of a crossbar switch connected to one memory module 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.

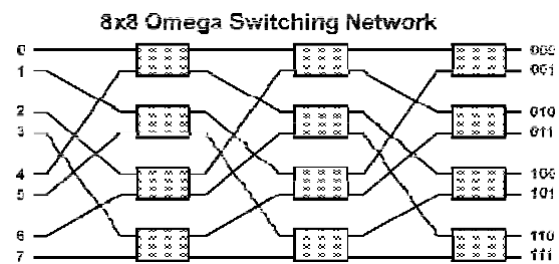


### Multistage Switching Network

- ✓ 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.

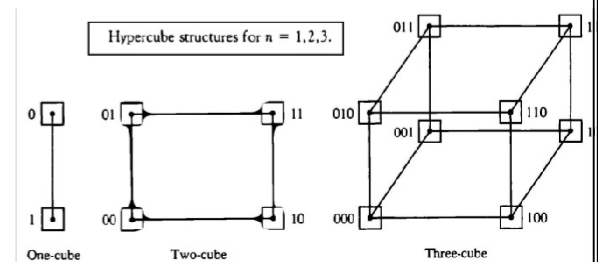


- ✓ One such topology is the omega switching network shown in Fig.
- ✓ In this configuration, there is exactly one path from each source to any particular destination.
- ✓ Some request patterns, however, cannot be connected simultaneously. For example, any two sources cannot be connected simultaneously 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.

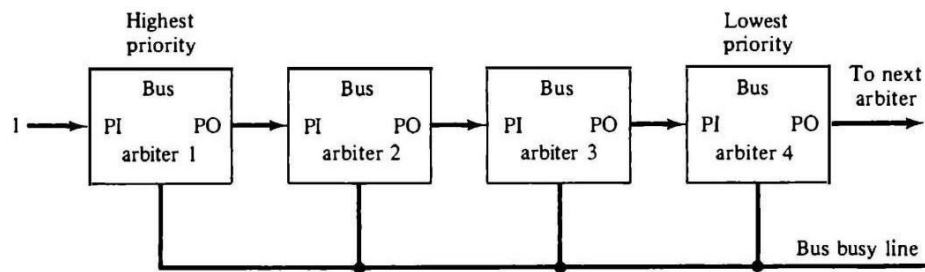
### System Bus

- ✓ 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	
<b>Data and address</b>	
Data lines (16 lines)	DATA0–DATA15
Address lines (24 lines)	ADRS0–ADRS23
<b>Data transfer</b>	
Memory read	MRDC
Memory write	MWTC
IO read	IORC
IO write	IOWC
Transfer acknowledge	TACK
<b>Interrupt control</b>	
Interrupt request (8 lines)	INT0–INT7
Interrupt acknowledge	INTA
<b>Miscellaneous control</b>	
Master clock	CCLK
System initialization	INIT
Byte high enable	BHEN
Memory inhibit (2 lines)	INH1–INH2
Bus lock	LOCK
<b>Bus arbitration</b>	
Bus request	BREQ
Common bus request	CBRQ
Bus busy	BUSY
Bus clock	BCLK
Bus priority in	BPRN
Bus priority out	BPRO
<b>Power and ground (20 lines)</b>	

## 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.

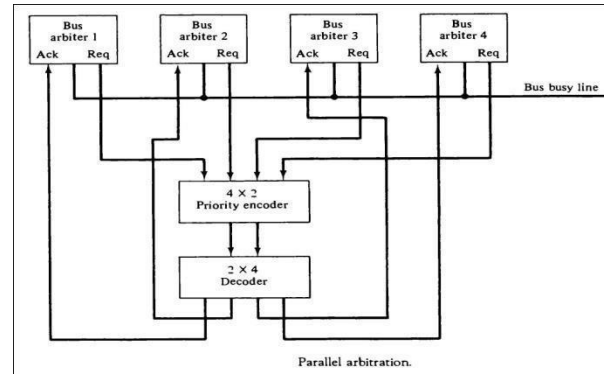


Serial (daisy-chain) arbitration.

- ✓ 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

- ✓ 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 IOP may be available to all CPUs. This provides a facility for sharing of system programs stored in the disk.
- ✓ A communication path between two CPUs can be established through a link between two IOPs associated with two different CPUs. This type of link allows each CPU to treat the other as an 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:

$R \leftarrow M[SEM]$                       Test semaphore  
 $M[SEM] \leftarrow 1$                       Set semaphore

- ✓ The semaphore is tested by transferring its value to a processor register R and then it is set to 1. The value in R determines what to do next.
- ✓ If the processor finds that  $R = 1$ , it knows that the semaphore was originally set. (The fact that it is set again does not change the semaphore value.) That means that another processor is executing a critical section, so the processor that checked the semaphore does not access the shared memory.
- ✓ If  $R = 0$ , it means that the common memory (or the shared resource that the semaphore represents) is available. The semaphore is set to 1 to prevent other processors from accessing memory. The processor can now execute the critical section.
- ✓ The last instruction in the program must clear location SEM to zero to release the shared resource to other processors. Note that the lock signal must be active during the execution of the test-and-set instruction. It does not have to be active once the semaphore is set.
- ✓ Thus the lock mechanism prevents other processors from accessing memory while the semaphore is being set. The semaphore itself, when set, prevents other processors from accessing shared memory while one processor is executing a critical section.