COMPUTER ORGANIZATION AND ARCHITECTURE

III Semester (IARE - R16)

Dr. Y. MOHANA ROOPA , Professor
Dr. P. L. SRINIVASA MURTHY , Professor
Mr. N.V.KRISHNA RAO , Associate Professor
Ms. A.SWAPNA, Assistant Professor

COMPUTER SCIENCE AND ENGINEERING
INSTITUTE OF AERONAUTICAL ENGINEERING
(Autonomous)
DUNDIGAL, HYDERABAD-500 043

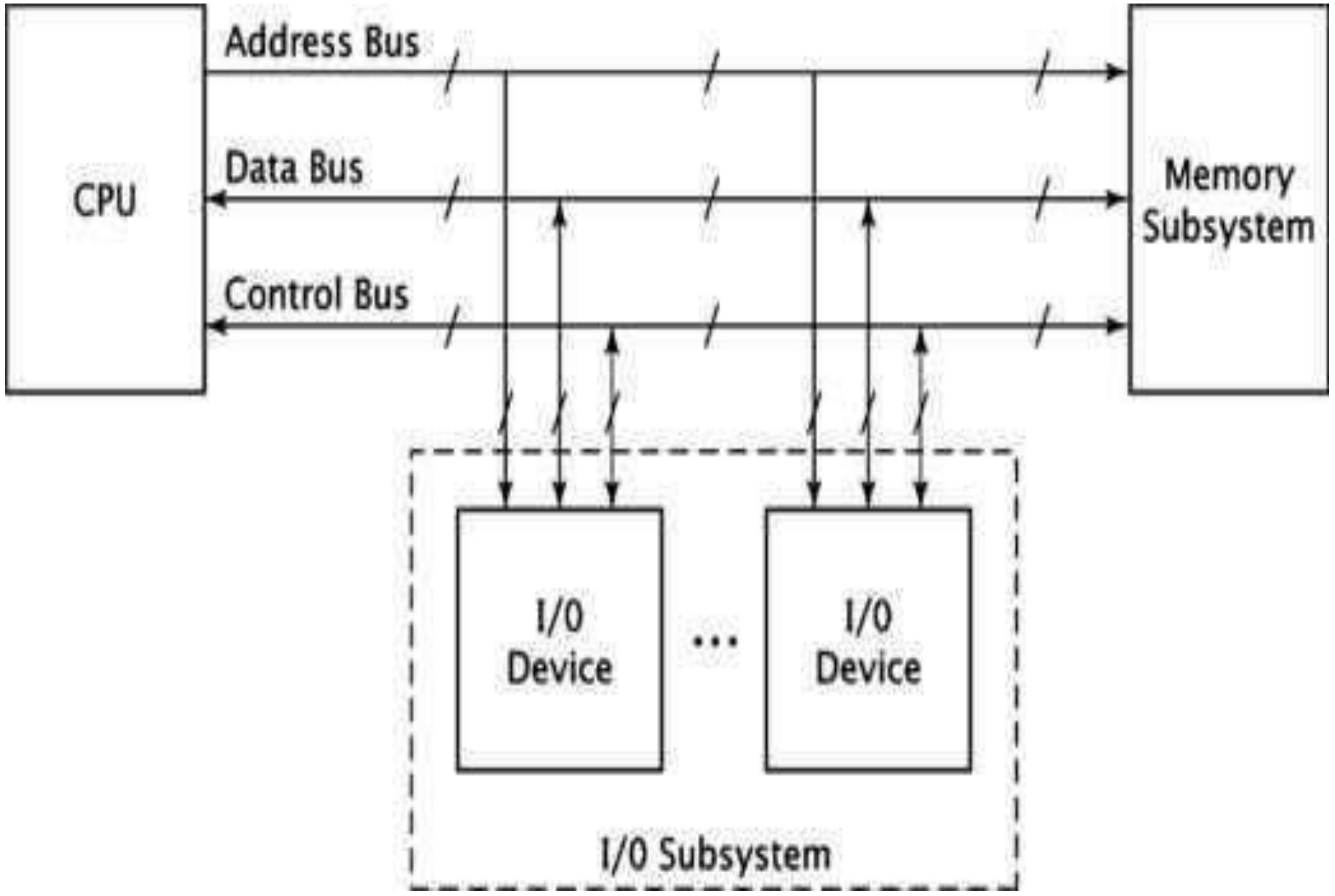
Unit-1

Introduction to Computer Organization and Architecture

Basic Computer Organization

- The basic computer organization has three main components:
- CPU
- Memory subsystem
- I/O subsystem

Generic computer Organization



System bus

The system bus has three buses,

- Address bus
- Data bus
- Control bus

Address bus

The uppermost bus is the **address bus**. When the CPU reads data or instructions from or writes data to memory, it must specify the address of the memory location it wishes to access

Data bus

Data is transferred via the **data bus**. When CPU fetches data from memory it first outputs the memory address on to its address bus. Then memory outputs the data onto the data bus. Memory then reads and stores the data at the proper locations.

Control bus

Control bus carries the control signal. Control signal is the collection of individual control signals. These signals indicate whether data is to be read into or written out of the CPU.

Instruction cycle

The instruction cycle is the procedure a microprocessor goes through to process an instruction.

It has three phases:

- Fetch
- Decode
- Execute

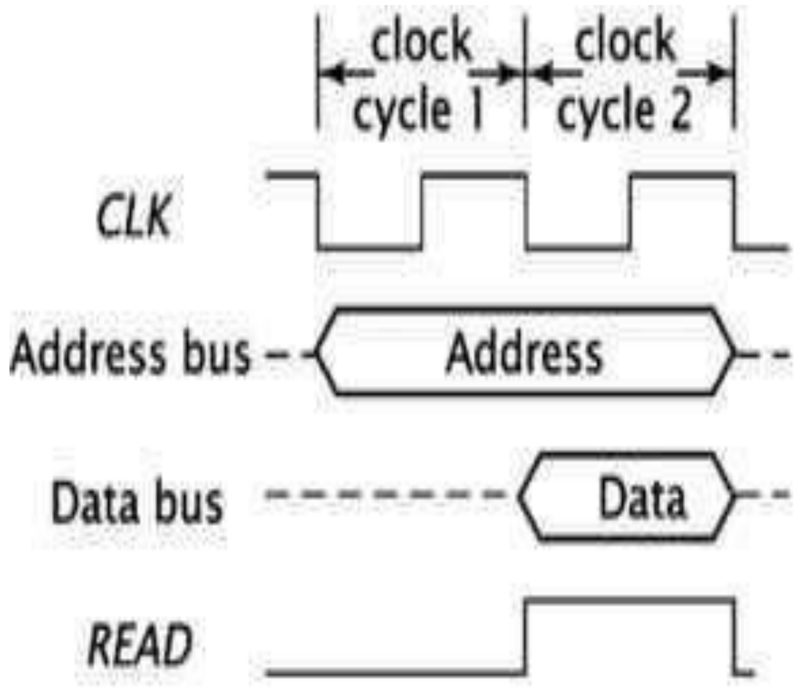
Instruction cycle

- First the processor **fetches** or reads the instruction from memory.
- Then it **decodes** the instruction determining which instruction it has fetched.
- Finally, it performs the operations necessary to **execute** the instruction.
- It performs some operation internally, and supplies the address, data & control signals needed by memory & I/O devices to execute the instruction.

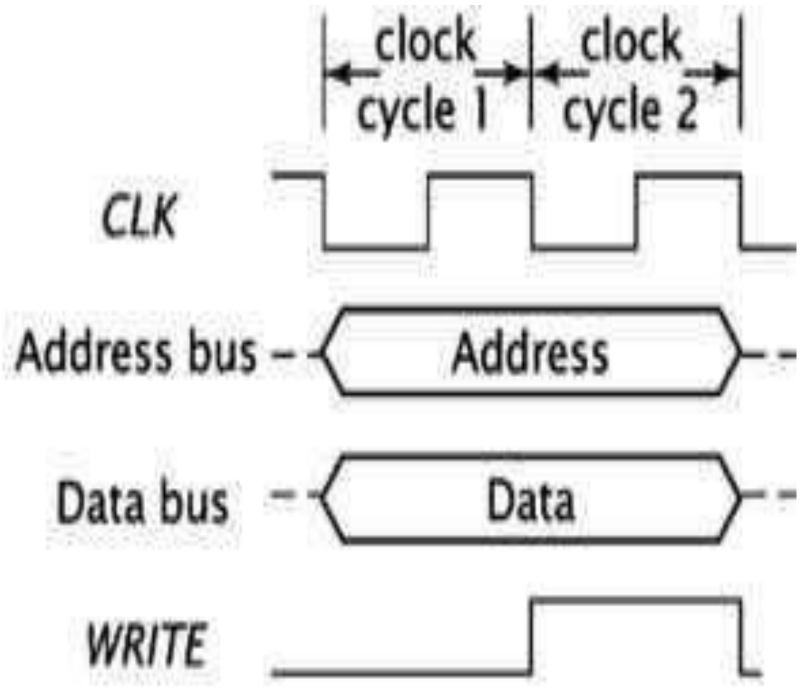
Control signals

- The **READ** signal is a signal on the control bus which the microprocessor asserts when it is ready to read data from memory or I/O device.
- When **READ** signal is asserted the memory subsystem places the instruction code be fetched on to the computer system's data bus. The microprocessor then inputs the data from the bus and stores its internal register.
- **READ** signal causes the memory to read the data, the **WRITE** operation causes the memory to store the data

Timing diagrams



(a)



(b)

Memory read operation

•In fig (a) the microprocessor places the address on to the bus at the beginning of a clock cycle, a 0/1 sequence of clock. One clock cycle later, to allow for memory to decode the address and access its data, the microprocessor asserts the READ control signal.

•This causes the memory to place its data onto the system data bus. During this clock cycle, the microprocessor reads the data off the system bus and stores it in one of the registers.

•At the end of the clock cycle it removes the address from the address bus and deasserts the READ signal. Memory then removes the data from the data from the data bus completing the memory read operation

Memory write operation

•In fig(b) the processor places the address and data onto the system bus during the first clock pulse.

•The microprocessor then asserts the WRITE control signal at
the end of the second clock cycle.

•At the end of the second clock cycle the processor completes the memory write operation by removing the address and data from the system bus and deasserting the WRITE signal.

CPU Organization

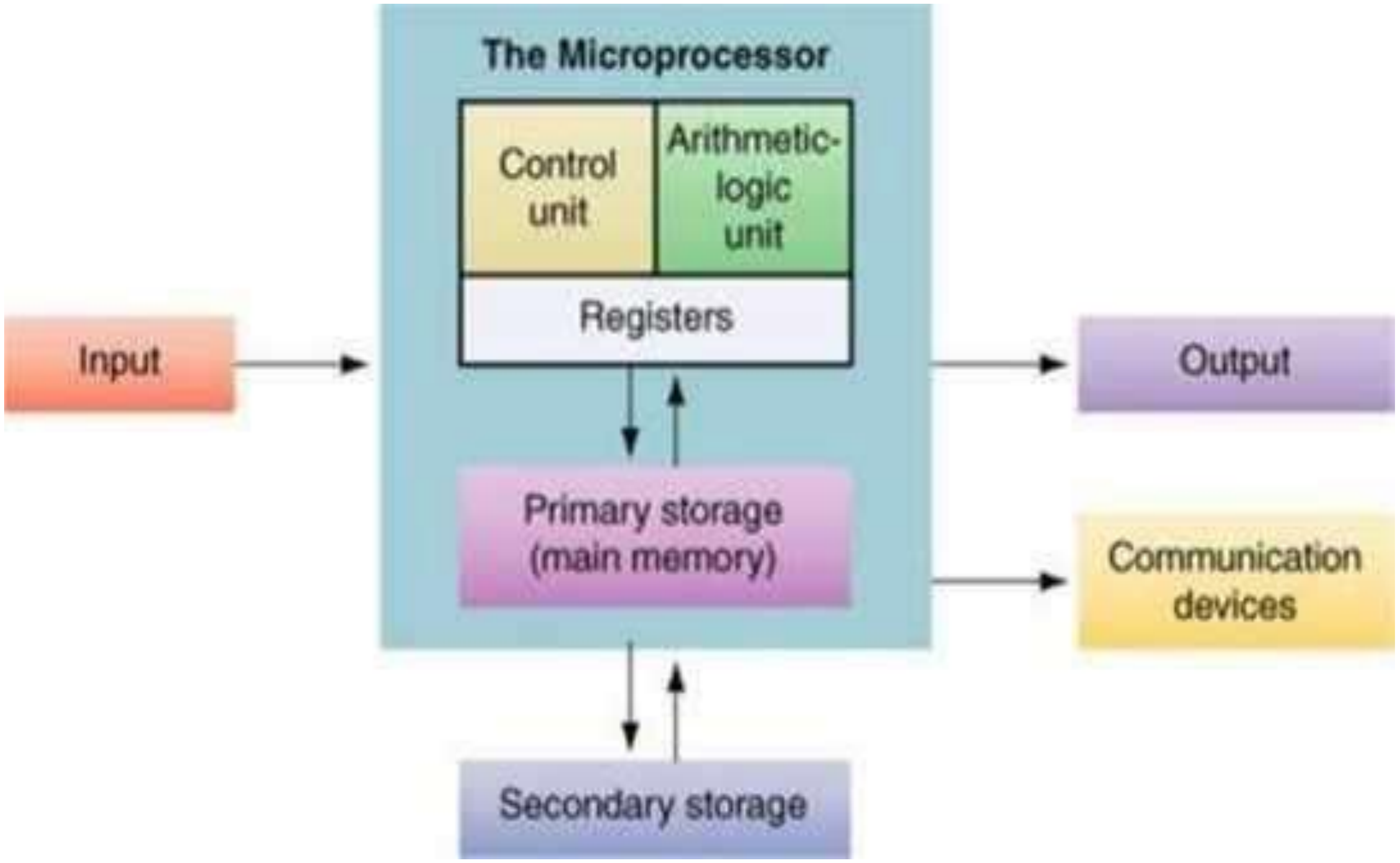
Central processing unit (CPU) is the electronic circuitry within a computer that carries out the instructions of a computer program by performing the basic arithmetic, logical, control and input/output (I/O) operations specified by the instructions.

CPU Organization

- In the computer all the all the major components are connected with the help of the **system bus**.

- **Data bus** is used to shuffle data between the various components in a computer system.
- When the software wants to access some particular memory location or I/O device it places the corresponding address on the **address bus**.
- The **control bus** is an eclectic collection of signals that control how the processor communicates with the rest of the system. The **read** and **write** control lines control the direction of data on the data bus.

CPU Organization



CPU Organization

•The register section, as its name implies, includes a set of registers
and a bus or other communication mechanism.

•The register in a processor’s instruction set architecture are found in the section of the CPU.

•The system address and data buses interact with this section of CPU. The register section also contains other registers that are not directly accessible by the programmer.

CPU Organization

- The fetch portion of the instruction cycle, the processor first outputs the address of the instruction onto the address bus. The processor has a register called the **program counter**.

- At the end of the instruction fetch, the CPU reads the
instruction code from the system data bus.

- It stores this value in an internal register, usually called the
instruction register”.

CPU Organization

- The **arithmetic / logic unit (or) ALU** performs most arithmetic
and logic operations such as adding and ANDing values.

- CPU controls the computer, the **control unit** controls the CPU. The control unit receives some data values from the register unit, which it used to generate the control signals.

- The **control unit** also generates the signals for the system
control bus such as READ, WRITE, IO/ signals

Memory Subsystem Organization

- Memory is the group of circuits used to store data.
- Memory components have some number of memory locations, each word of which stores a binary value of some fixed length.
- The number of locations and the size of each location vary from memory chip to memory chip, but they are fixed within individual chip.

Memory chips Internal organization

- Memory is usually organized in the form of arrays, in which each cell is capable of storing one bit information.

- Each row of cell constitutes a memory word, and all cells of a row are connected to a common column called word line, which is driven by the address decoder on the chip

Types of Memory

- There are two types of memory chips,

1. Read Only Memory (ROM)
2. Random Access Memory (RAM)

ROM Chips

- Masked ROM(or) simply ROM
- PROM(Programmed Read Only Memory)
- EPROM(Electrically Programmed Read Only Memory)
- EEPROM(Electrically Erasable PROM)
- Flash Memory

ROM Chips

Masked ROM

- A masked ROM or simply ROM is programmed with data as chip is fabricated.
- The mask is used to create the chip and chip is designed with the required data hardwired in it.

- **PROM**
  - Some ROM designs allow the data to be loaded by the user, thus providing programmable ROM (PROM).
  - Programmability is achieved by inserting a fuse at point P in the above fig. Before it is programmed, the ŵeŵoƌy ĐoŶtaiŶs all Ϭ's.
  - The useƌ iŶseƌt ϭ's at the ƌeƋuiƌed loĐatioŶs ďy ďuƌŶiŶg

out the fuse at these locations using high current pulse.

  - The fuses in PROM cannot restore once they are

ďlowŶ, PROM's ĐaŶ oŶly ďe pƌogƌaŵŵed oŶĐe.

ROM Chips

- **EPROM**
  - EPROM is the another ROM chip allows the stored data to be erased and new data to be loaded. Such an erasable reprogrammable ROM is usually called an EPROM.
  - Programming in EPROM is done by charging of capacitors. The charged and uncharged capacitors cause each word of memory to store the correct value.
  - The chip is erased by being placed under UV light, which causes the capacitor to leak their charge.

ROM Chips

- **EEPROM**

- A significant disadvantage of the EPROM is the chip is physically removed from the circuit for reprogramming and that entire contents are erased by the UV light.
- Another version of EPROM is EEPROM that can be both programmed and erased electrically, such chips called EEPROM, do not have to remove for erasure.
- The only disadvantage of EEPROM is that different voltages

are need for erasing, writing, reading and stored data

ROM Chips

Flash Memory

A special type of EEPROM is called a flash memory is electrically erase data in blocks rather than individual locations.

It is well suited for the applications that writes blocks of data and can be used as a solid state hard disk. It is also used for data storage in digital computers.

Memory Subsystem Organization

- **RAM Chips:**
  - RAM stands for Random access memory. This often referred to as read/write memory. Unlike the ROM it initially contains no data.
  - The data pins are bidirectional unlike in ROM.
  - A ROM chip loses its data once power is removed so it is a volatile memory.
  - RAM chips are differentiated based on the data they maintain.

- Dynamic RAM (DRAM)
- Static RAM (SRAM)

Memory chips Internal organization

- **Dynamic RAM**

- DRAM chips are like leaky capacitors. Initially data is stored in the DRAM chip, charging its memory cells to their maximum values.
- The charging slowly leaks out and would eventually go too low

to represent valid data.

- Before this a refresher circuit reads the content of the DRAM

and rewrites data to its original locations.

- DRAM is used to construct the RAM in personal computers.

Memory chips Internal organization

- **Static RAM**

- Static RAM are more likely the register .Once the data is written to SRAM, its contents stay valid it does not have to be refreshed.
- Static RAM is faster than DRAM but it is also much more expensive. Cache memory in the personal computer is constructed from SRAM.

Memory chips Internal organization

- **Dynamic RAM**

- DRAM chips are like leaky capacitors. Initially data is stored in the DRAM chip, charging its memory cells to their maximum values.
- The charging slowly leaks out and would eventually go too low

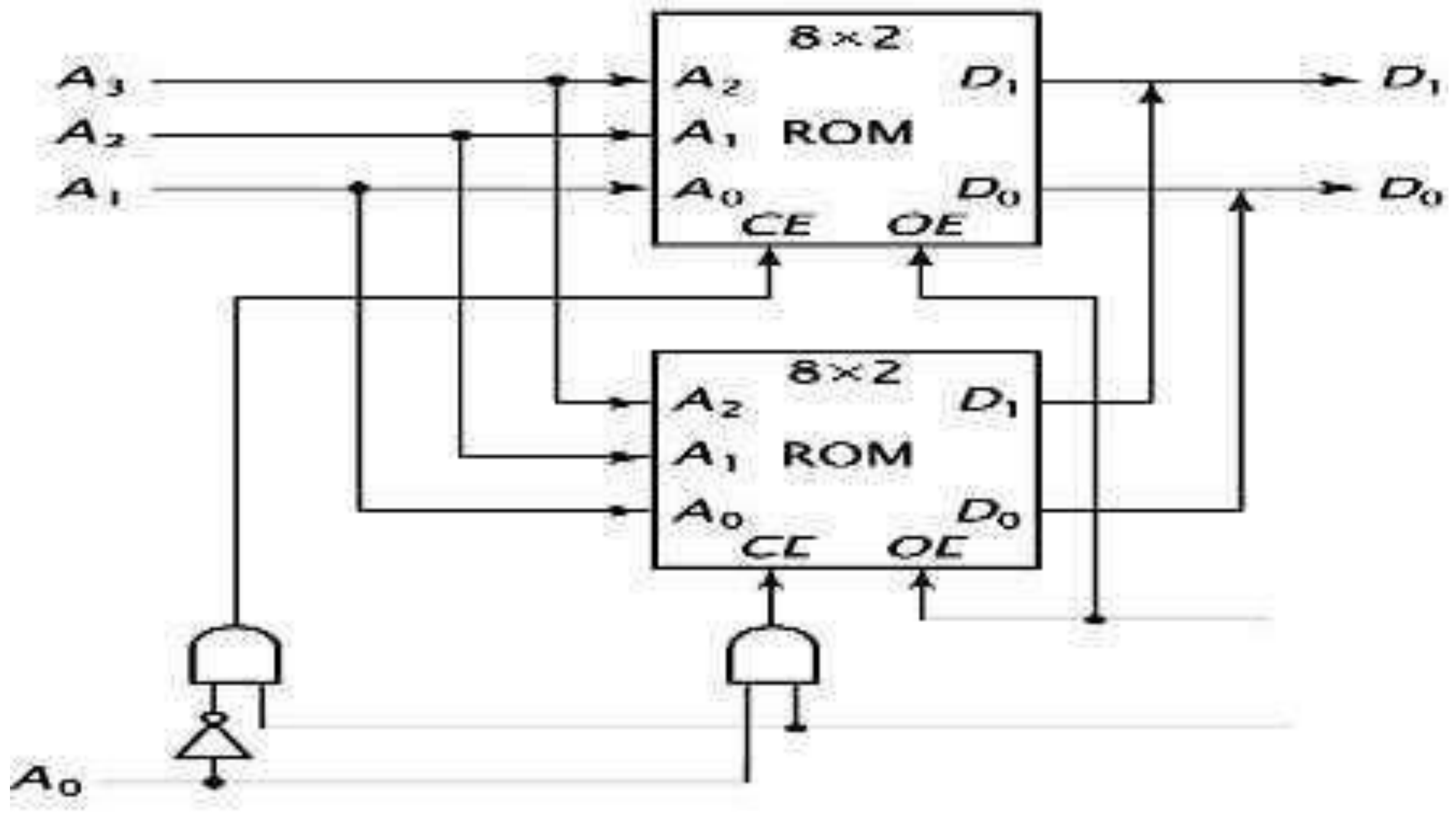
to represent valid data.

- Before this a refresher circuit reads the content of the DRAM

and rewrites data to its original locations.

- DRAM is used to construct the RAM in personal computers.

Memory subsystem configuration



(b)

Multi byte organization

- There are two commonly used organizations for multi byte data.
  - Big endian
  - Little endian

Multi byte organization

- In BIG-ENDIAN systems the most significant byte of a multi-byte data item always has the lowest address, while the least significant byte has the highest address.

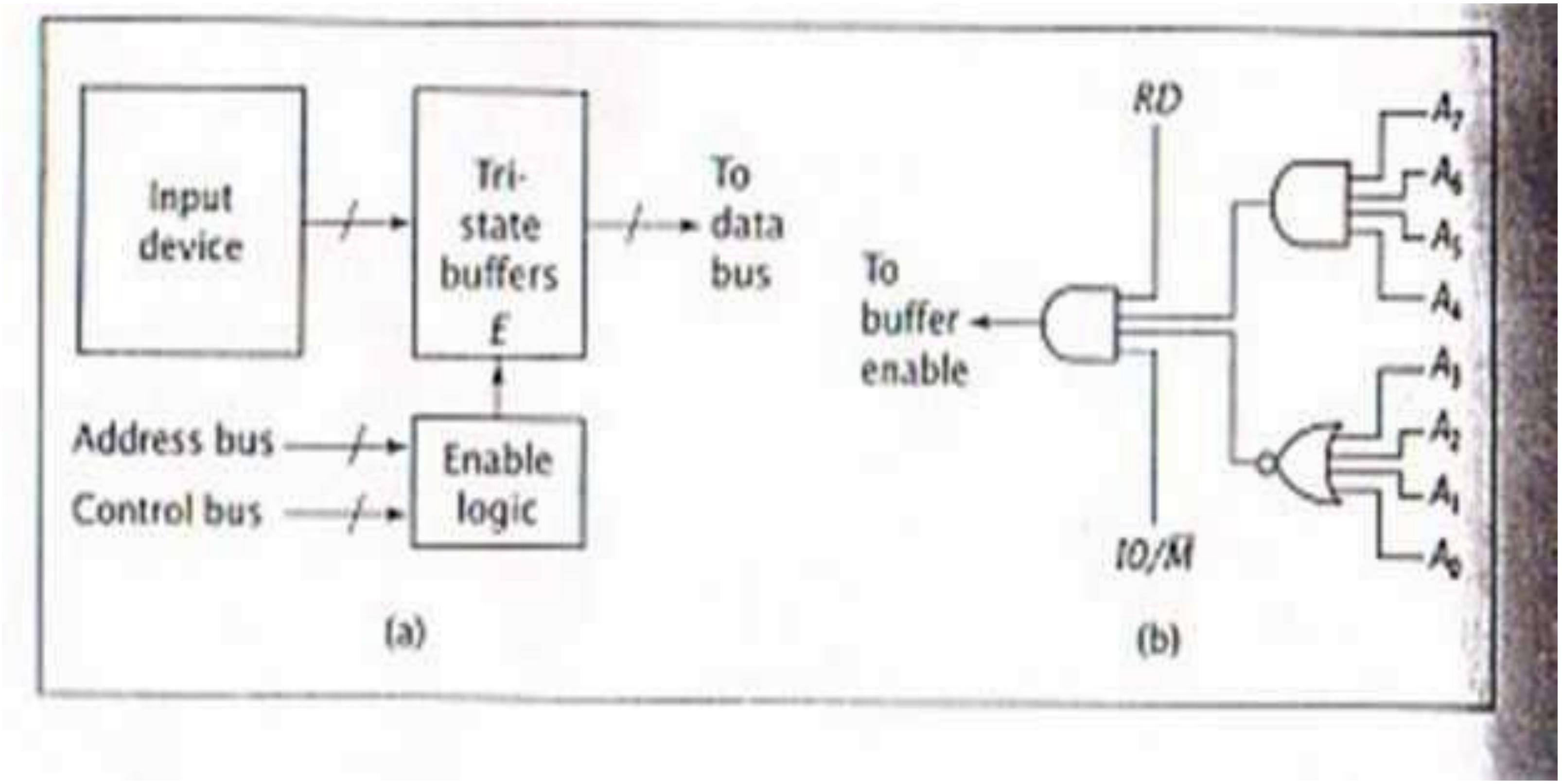
- In LITTLE-ENDIAN systems, the least significant byte of a multi-byte data item always has the lowest address, while the most significant byte has the highest address.

I/O Subsystem Organization

The I/O subsystem is treated as an independent unit in the computer The CPU initiates I/O commands generically

- Read, write, scan, etc.
- This simplifies the CPU

Input Device



Input Device

- The data from the input device goes to the tri-state buffers. When the value in the address and control buses are correct, the buffers are enabled and data passes on the data bus.

- The CPU can then read this data. If the conditions are not right the logic block does not enable the buffers and do not place on the bus.

- The enable logic contains 8-bit address and also generates

two control signals RD and I/O.

Output Device

- The design of the interface circuitry for an output device such as a computer monitor is somewhat different than for the input device.
- Tri-state buffers are replaced by a register.
- The tri-state buffers are used in input device interfaces to

make sure that one device writes data to the bus at any time.

- Since the output devices read from the bus, rather that writes

data to it, they don’t need the buffers.

- The data can be made available to all output devices but the devices only contains the correct address will read it in

Multi byte organization

- There are two commonly used organizations for multi byte data.
  - Big endian
  - Little endian

Multi byte organization

- In BIG-ENDIAN systems the most significant byte of a multi-byte data item always has the lowest address, while the least significant byte has the highest address.

- In LITTLE-ENDIAN systems, the least significant byte of a multi-byte data item always has the lowest address, while the most significant byte has the highest address.

Output Device

- The design of the interface circuitry for an output device such as a computer monitor is somewhat different than for the input device.
- Tri-state buffers are replaced by a register.
- The tri-state buffers are used in input device interfaces to

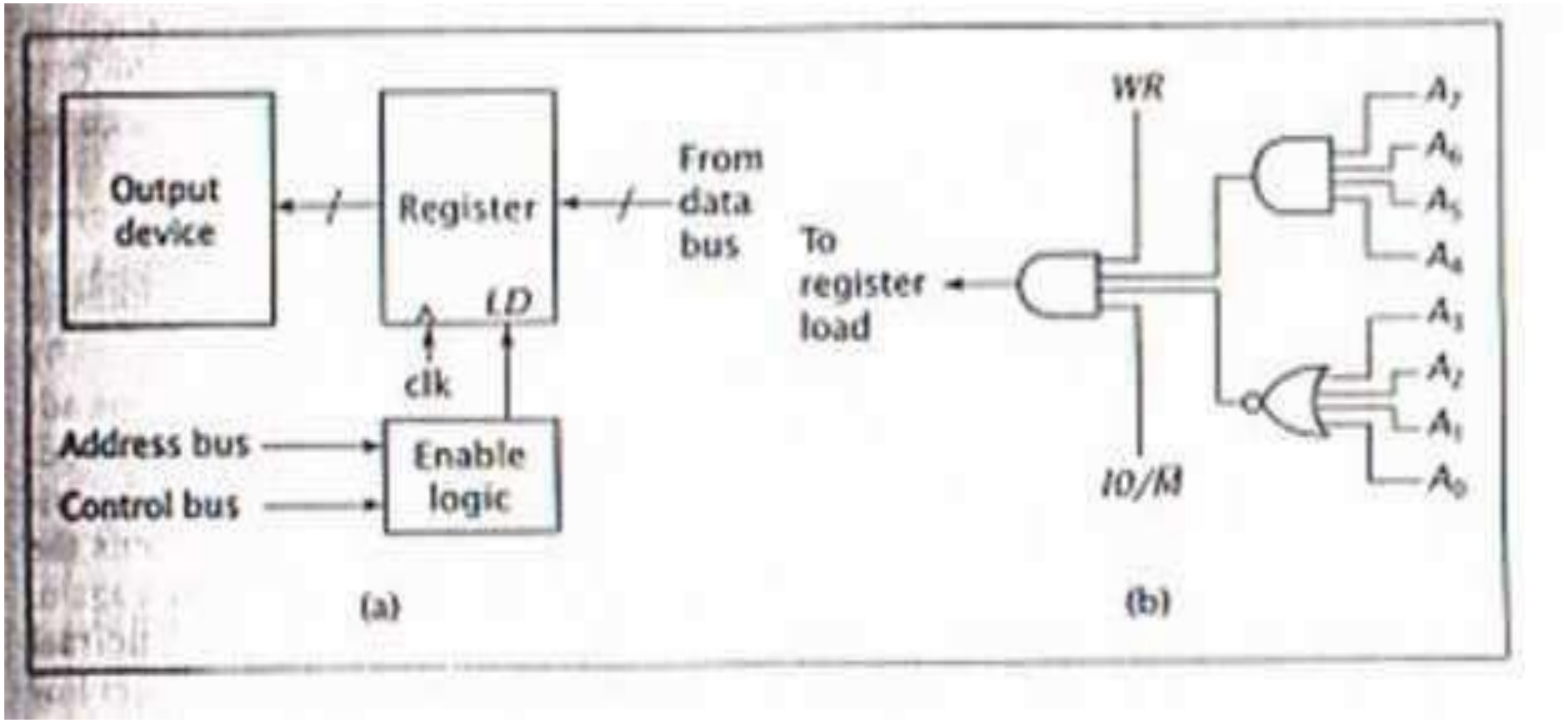
make sure that one device writes data to the bus at any time.

- Since the output devices read from the bus, rather that writes

data to it, they don't need the buffers.

- The data can be made available to all output devices but the devices only contains the correct address will read it in

Output Device



An output device: (a) with its interface and (b) the enable logic for the registers

Output Device

- Some devices are used for both input and output. Personal computer and hard disk devices are falls into this category. Such a devices requires a combined interface that is essential two interfaces.
- A bidirectional I/O device with its interface and enable load logic is shown in the Figure 1.7 below.

Output Device

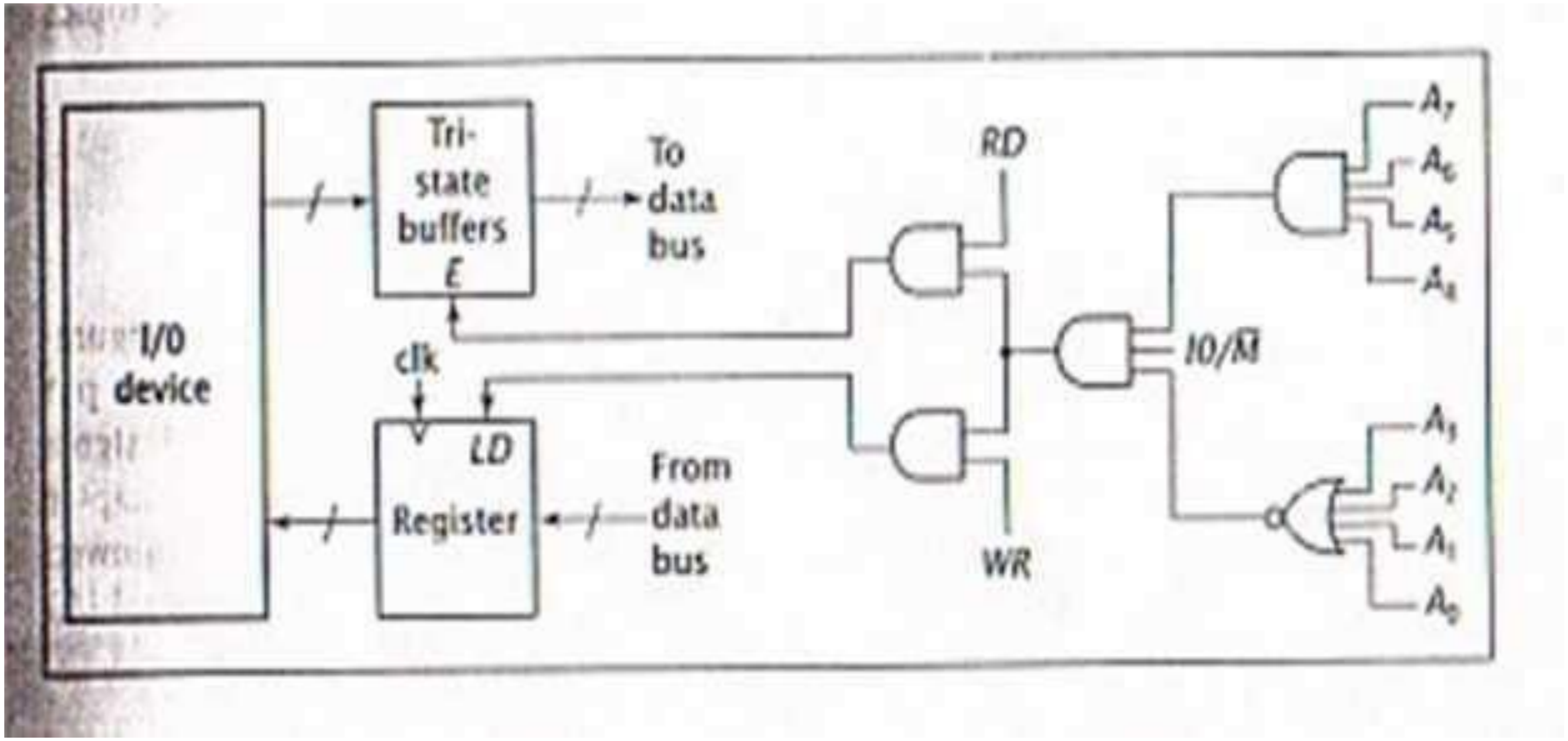


Figure 1.7: A bidirectional I/O device with its interface and enable/load logic

A Simple Computer- Levels of PL

- Computer programming languages are divided into 3 categories.

- High level language
- Assembly level language
- Machine level language

A Simple Computer- Levels of PL

- **High level languages** are platform independent that is these programs can run on computers with different microprocessor and operating systems without modifications. Languages such as C++, Java and FORTRAN are high level languages.

- **Assembly languages** are at much lower level of abstraction.

Each processor has its own assembly language

A Simple Computer- Levels of PL

- The lowest level of programming language is **machine level** languages. These languages contain the binary values that cause the microprocessor to perform certain operations. When microprocessor reads and executes an instruction it’s a machine language instruction.

A Simple Computer- Levels of PL

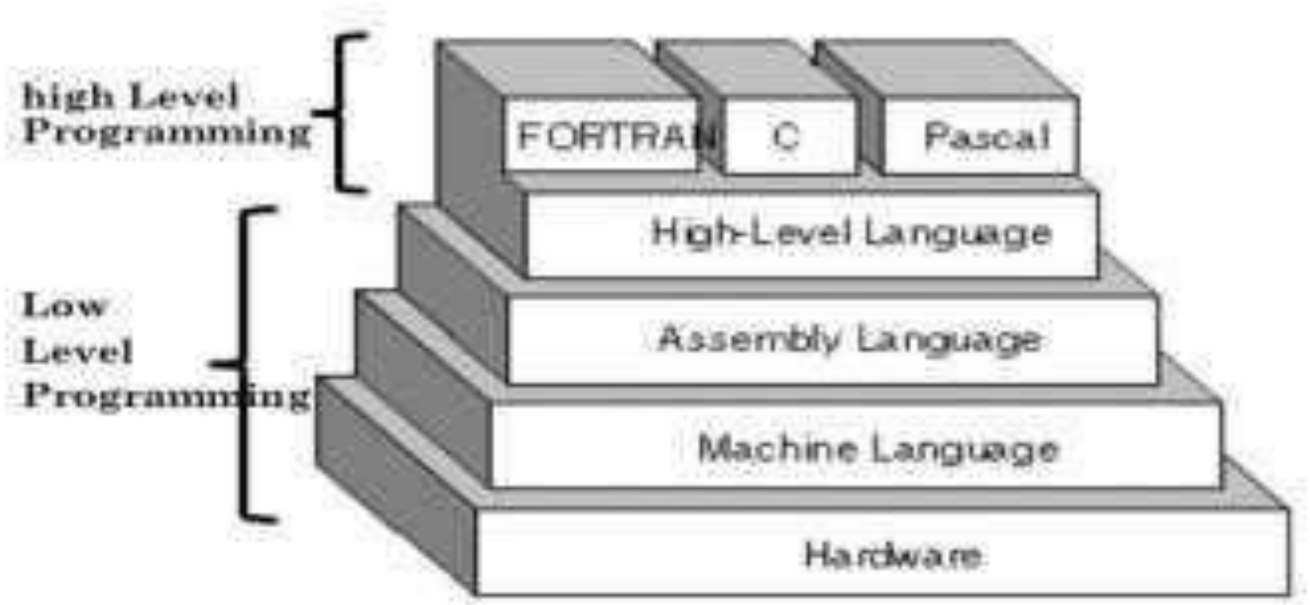


Figure 1.8: Levels of programming languages

A Simple Computer- Levels of PL

- High level language programs are compiled and assembly level language programs are assembled.

- A program written in the high level language is input to the compiler.

- compiler checks to make sure every statement in the program is valid. When the program has no syntax errors the compiler finishes the compiling the program that is source code and generates an object code file.

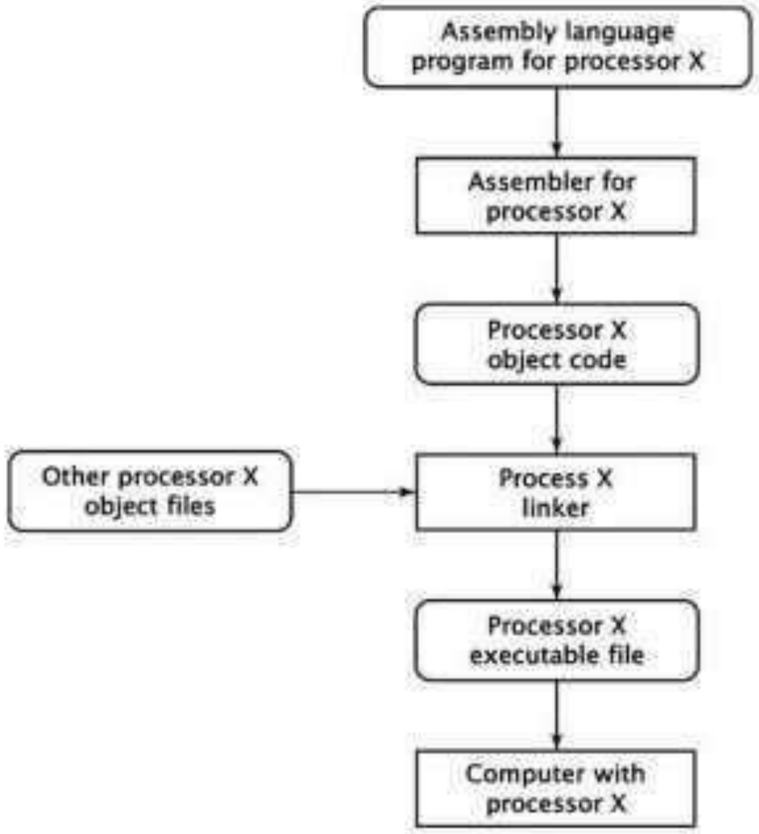
A Simple Computer- Levels of PL

- An **object code** is the machine language equivalent of source code.

- A **linker** combines the object code to any other object code. This combined code stores in the **executable file**.

A Simple Computer- Levels of PL

Assembling programs



Assembly language program for processor X
Assembler for processor X
Processor X object code
Other processor X object files
Process X linker
Processor X executable file
Computer with processor X

- Assembly language is specific to one microcontroller
- Converts the source code into object code
- The linker will combine the object code of your program with any other required object code to produce executable code
- Loader will load the executable code into memory, for execution

A Simple Computer- Levels of PL

- Programmers don’t written the programs in machine language rather programs written in assembly or high level are converted into machine level and then executed by the microprocessor.

•High level language programs are compiled and assembly level language programs are assembled.

•A program written in the high level language is input to the compiler. The compiler checks to make sure every statement in the program is valid. When the program has no syntax errors the compiler finishes the compiling the program that is source code and generates an object code file.

A Simple Computer- Levels of PL

- An **object code** is the machine language equivalent of source code.

- A **linker** combines the object code to any other object code. This combined code stores in the **executable file.**

A Simple Computer- Levels of PL

- **High level language** programs are compiled and assembly level language programs are assembled.

- A program written in the **high level language** is input to the compiler.

- compiler checks to make sure every statement in the program is valid. When the program has no syntax errors the compiler finishes the compiling the program that is source code and generates an object code file.

A Simple Computer- Levels of PL

- An **object code** is the machine language equivalent of source code.

- A **linker** combines the object code to any other object code. This combined code stores in the **executable file**.

Assembly Language Instructions

- **Instruction types**
- Assembly languages instructions are grouped together based on the operation they performed.
- Data transfer instructions
- Data operational instructions
- Program control instructions

Assembly Language Instructions

- **Data transfer instructions**
- **Load the data from memory into the microprocessor**: These instructions copy data from memory into a microprocessor register.
- **Store the data from the microprocessor into the memory**: This is similar to the load data expect data is copied in the opposite direction from a microprocessor register to memory.
- **Move data within the microprocessor**: These operations copies data from one microprocessor register to another.
- **Input the data to the microprocessor**: The microprocessor inputs the data from the input devices ex: keyboard in to one of its registers.

Assembly Language Instructions

- **Data operational instructions**
- **Data operational instructions** do modify their data values. They typically perform some operations using one or two data values (operands) and store result.

- **Arithmetic instructions** make up a large part of data operations instructions. Instructions that add, subtract, multiply, or divide values fall into this category. An instruction that increment or decrement also falls in to this category.

Assembly Language

- **Logical instructions** perform basic logical operations on data. They AND, OR, or XOR two data values or complement a single value.

- **Shift operations** as their name implies shift the bits of a data

values also comes under this category

Assembly Language

- **Program control instructions**
- Program control instructions are used to control the flow of a program. Assembly language instructions may include subroutines like in high level language program may have subroutines, procedures, and functions.
- A jump or branch instructions are generally used to go to

another part of the program or subroutine.

Instruction Set Architecture Design

- Designing of the instructi o n se t i s the most important in designing the microprocessor. A poor designed ISA even it is implemented well leads to bad micro processor.

- A well designed instruction set architecture on the other hand can result in a powerful processor that can meet variety of needs.

- In designing ISA the designer must evaluate the tradeoffs in performance and such constrains issues as size and cost when designing ISA specifications.

Instruction Set Architecture Design

- The issue of **completeness** of the ISA is one of the criteria in designing the processor that means the processor must have complete set of instructions to perform the task of the given application.

- Another criterion is instruction **orthogonality.** Instructions are orthogonal if they do not overlap, or perform the same function. A good instruction set minimizes the overlap between instructions.

Instruction Set Architecture Design

- Another area that the designer can optimize the ISA is the **register set**. Registers have a large effect on the performance of a CPU. The CPU Can store data in its internal registers instead of memory. The CPU can retrieve data from its registers much more likely than form the memory.

- Having too few registers causes a program to make more reference to the memory thus reducing performance

Simple Instruction Set Architecture

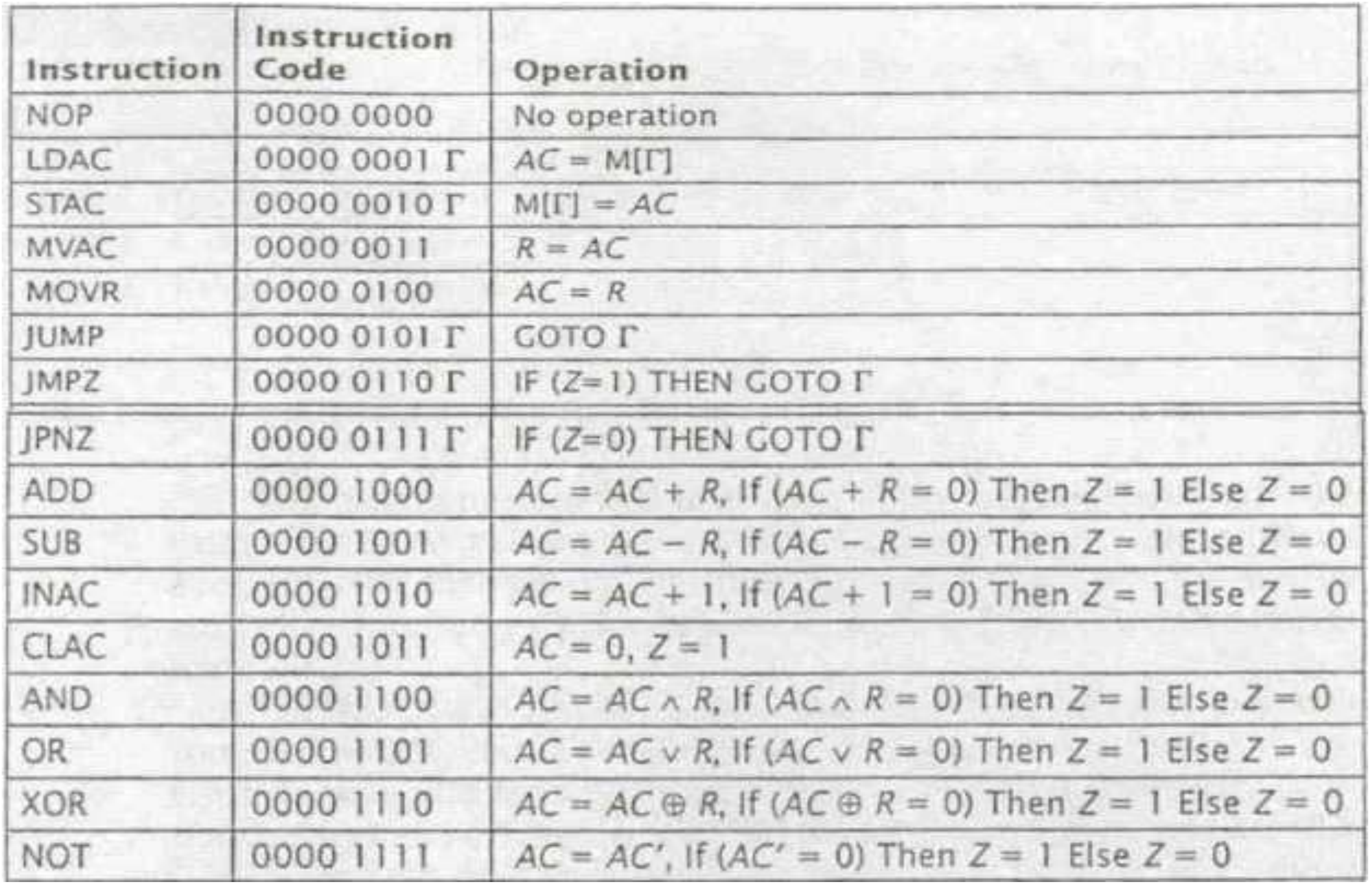
This processor inputting the data from and outputting the data to external devices such as microwave ovens keypad and display are treated as memory accesses. There are two types of input/output interactions that can design a CPU to perform.

- Isolated I/O
- Memory mapped I/O

Simple Instruction Set

- An **isolated I/O** input and output devices are treated as being separated from memory. Different instructions are used for memory and I/O.
- **Memory mapped I/O** treats input and output devices as memory locations the CPU access these I/O devices using the same instructions that it uses to access memory. For relatively simple CPU memory mapped I/O is used.
- There are three registers in ISA of this processor.
  - Accumulator (AC)
  - Register R
  - Zero flag (Z)

Simple Instruction Set



Instruction	Instruction Code	Operation
NOP	0000 0000	No operation
LDAC	0000 0001 Γ	$AC = M[\Gamma]$
STAC	0000 0010 Γ	$M[\Gamma] = AC$
MVAC	0000 0011	$R = AC$
MOVR	0000 0100	$AC = R$
JUMP	0000 0101 Γ	GOTO Γ
JMPZ	0000 0110 Γ	IF (Z=1) THEN GOTO Γ
JPNZ	0000 0111 Γ	IF (Z=0) THEN GOTO Γ
ADD	0000 1000	$AC = AC + R$, If $(AC + R = 0)$ Then $Z = 1$ Else $Z = 0$
SUB	0000 1001	$AC = AC - R$, If $(AC - R = 0)$ Then $Z = 1$ Else $Z = 0$
INAC	0000 1010	$AC = AC + 1$, If $(AC + 1 = 0)$ Then $Z = 1$ Else $Z = 0$
CLAC	0000 1011	$AC = 0$, $Z = 1$
AND	0000 1100	$AC = AC \wedge R$, If $(AC \wedge R = 0)$ Then $Z = 1$ Else $Z = 0$
OR	0000 1101	$AC = AC \vee R$, If $(AC \vee R = 0)$ Then $Z = 1$ Else $Z = 0$
XOR	0000 1110	$AC = AC \oplus R$, If $(AC \oplus R = 0)$ Then $Z = 1$ Else $Z = 0$
NOT	0000 1111	$AC = AC'$, If $(AC' = 0)$ Then $Z = 1$ Else $Z = 0$

Simple Instruction Set

- The final component is the instruction set architecture for this

relatively simple CPU is shown in the table above.

- The LDAC, STAC, JUMP, JMPZ AND JPNZ instructions all require

a 16-bit ŵeŵory address represeŶted by the syŵbol Γ .

- Since each byte of memory is 8-bit wide these instructions requires 3 bytes in memory. The first byte contains the opcode for the instruction and the remaining 2 bytes for the address.

Simple Instruction Set

- The instructions of this instruction set architecture can be grouped in to 3 categories
  - Data transfer instructions
  - Program control instructions
  - Data operational instructions

Relatively Simple Computer

- In this relatively simple computer Figure 1:11 shown below, we put all the hard ware components of the computer together in one system. This computer will have 8K ROM starting at address 0 fallowed by 8K RAM. It also has a memory mapped bidirectional I/O port at address 8000H.
-
- First let us look at the CPU since it uses 16 bit address labeled A15 through A0. System bus via pins through D7 to D0. The CPU also has the two control lines READ and WRITE.
-
-

Relatively Simple Computer

- Since it uses the memory mapped I/O it does not need a control signal such as .
- The relatively simple computer is shown in the figure below. It only contains the CPU details. Each part will be developed in the design.

Relatively Simple Computer

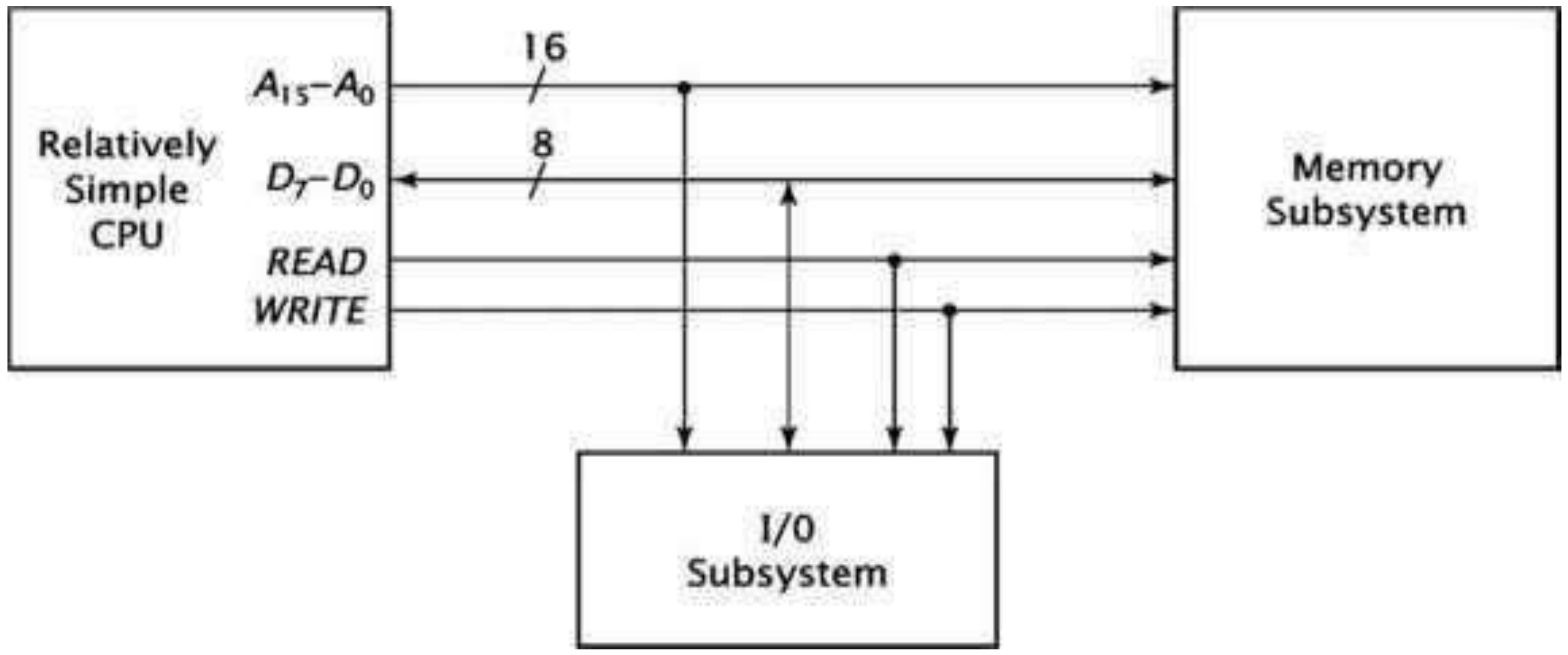


Figure 1:11: A relatively simple computer: CPU details only

Relatively Simple Computer

- To access a memory chip the processor must supply an address used by the chip. An 8K memory chip has 213 internal memory locations it has 13bit address input to select one of these locations.

- The address input of each chip receives CPU address bits A12 to A0 from system address bus. The remaining three bits A15, A14, and A13 will be used to select one of the memory chips.

Register Transfer Language

- A digital system is an interconnection of digital hardware module. Digital systems varies in size and complexity from a few integrated circuits to a complex of interconnected and interacting digital computers.

- Digital system design invariably uses a modular approach. The modules are constructed from such digital components as registers, decoders, arithmetic elements, and control logic.

- The various modules are interconnected with common data and control paths to form a digital computer system.

Register Transfer Language

- A micro operation 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 micro operations are shift, count, clear, and load

Register Transfer Language

The internal hardware organization of a digital computer is best defined by specifying:

•1. The set of registers it contains and their function.

•2. The sequence of micro operations performed on the binary information stored in the registers.

•3. The control that initiates the sequence of micro operations.

Register Transfer Language

- The symbolic notation used to describe the micro operation transfers among registers is called a **register transfer language**.
- The term "register transfer" implies the availability of hardware logic circuits that can perform a stated micro operation and transfer the result of the operation to the same or another register.
- The word "language" is borrowed from programmers, who apply this term to programming languages. A programming language is a procedure for writing symbols to specify a given computational process.

Register Transfer Language

- A register transfer language is a system for expressing in symbolic form the micro operation 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. It can also be used to facilitate the design process of digital systems.

- The register transfer language adopted here is believed to be as simple as possible, so it should not take very long to memorize.

Register Transfer Language

- Computer registers are designated by capital letters (sometimes followed by numerals) to denote the function of the register.

- For example, MAR,PC,IR The individual flip-flops

Register Transfer Language

- 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 R l. By definition, the content of the source register R1 does not change after the transfer.

Register Transfer Language

- The transfer to occur only under a predetermined control condition. This can be shown by means of an if-then statement.

If (P = 1) then (R2 <--R1)

Register Transfer Language

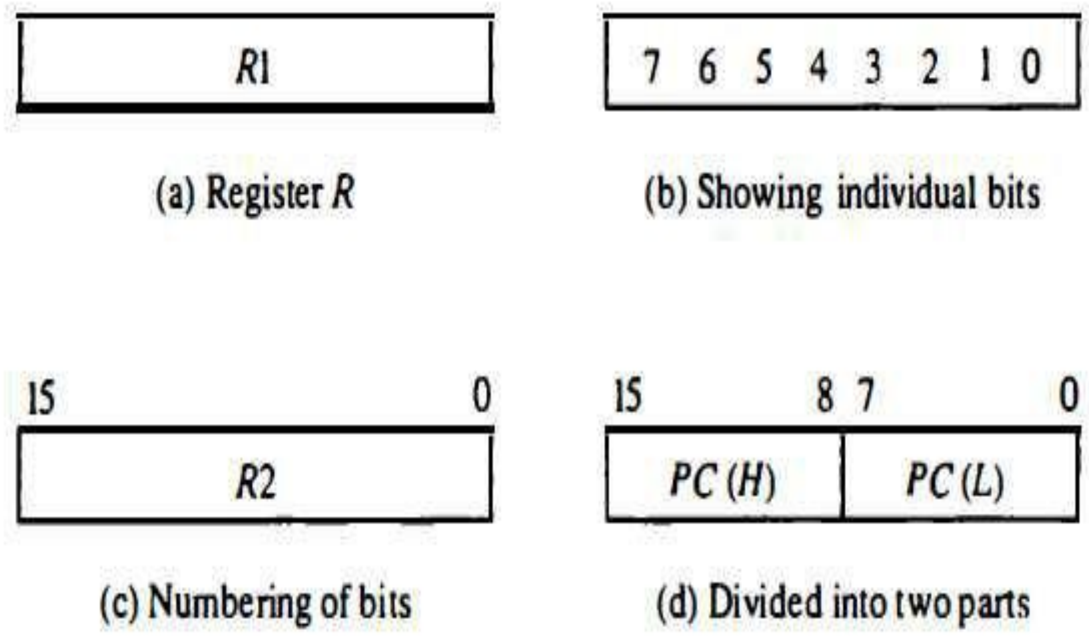


Figure 1 Block diagram of register.

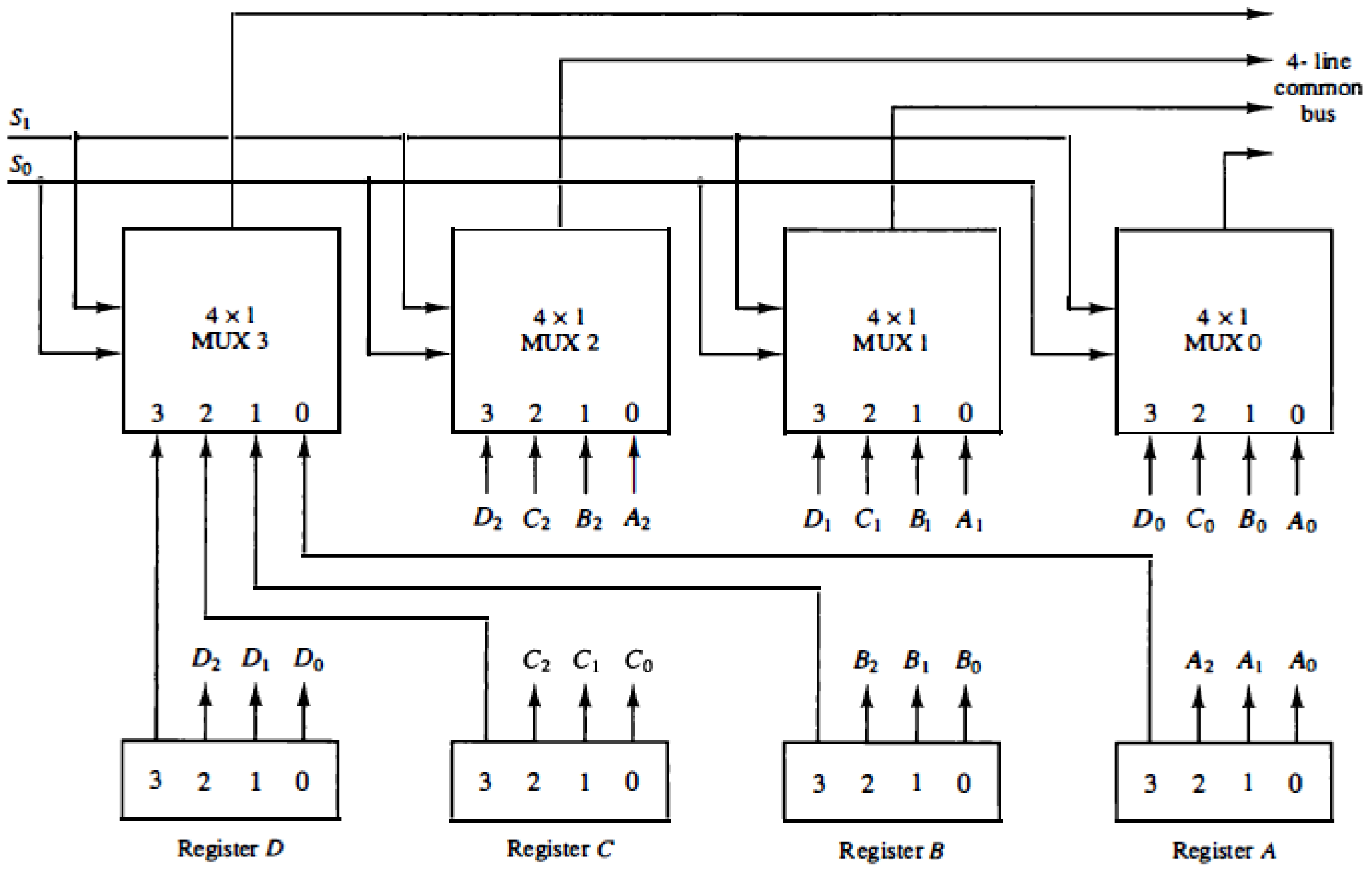
Basic Symbols For Register

Symbol	Description	Examples
Letters and numerals	Denotes a register	MAR,R2
Parenthesis( )	Denotes a part of resisters	R290-70,R2(L)
Arrow	Denotes transfer of information	R2 $\leftarrow$ R1
Comma ,	Separates two microoperations	R2 $\leftarrow$ R1, R1 $\leftarrow$ R2

Bus and Memory Transfers

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

Bus and Memory Transfers



Bus and Memory Transfers

- The two selection lines S1 and S0 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 S1S0 = 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 S1S0 = 01, and so on

Bus and Memory Transfers

Function table for Bus

S_1	S_0	Registers selected
0	0	A
0	1	B
1	0	C
1	1	D

Bus and Memory Transfers

- The symbolic statement for a bus transfer may mention the bus or its presence may be implied in the statement. When the bus is includes in the statement, the register transfer is symbolized as follows:

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

Bus and Memory Transfers

- The content of register C is placed on the bus, and the content of the bus is loaded into register R 1 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.

$$R1 \leftarrow C$$

Bus and Memory Transfers

- **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.

Bus and Memory Transfers

- The high-impedance state behaves like an open circuit, which means that the output is disconnected and does not have logic significance.
- Three-state gates may perform any conventional logic, such as AND or NAND.
- The graphic symbol of a three-state buffer gate is shown in Figure. It is distinguished from a normal buffer by having both a normal input and a control input. The control input determines the output state.

Bus and Memory Transfers

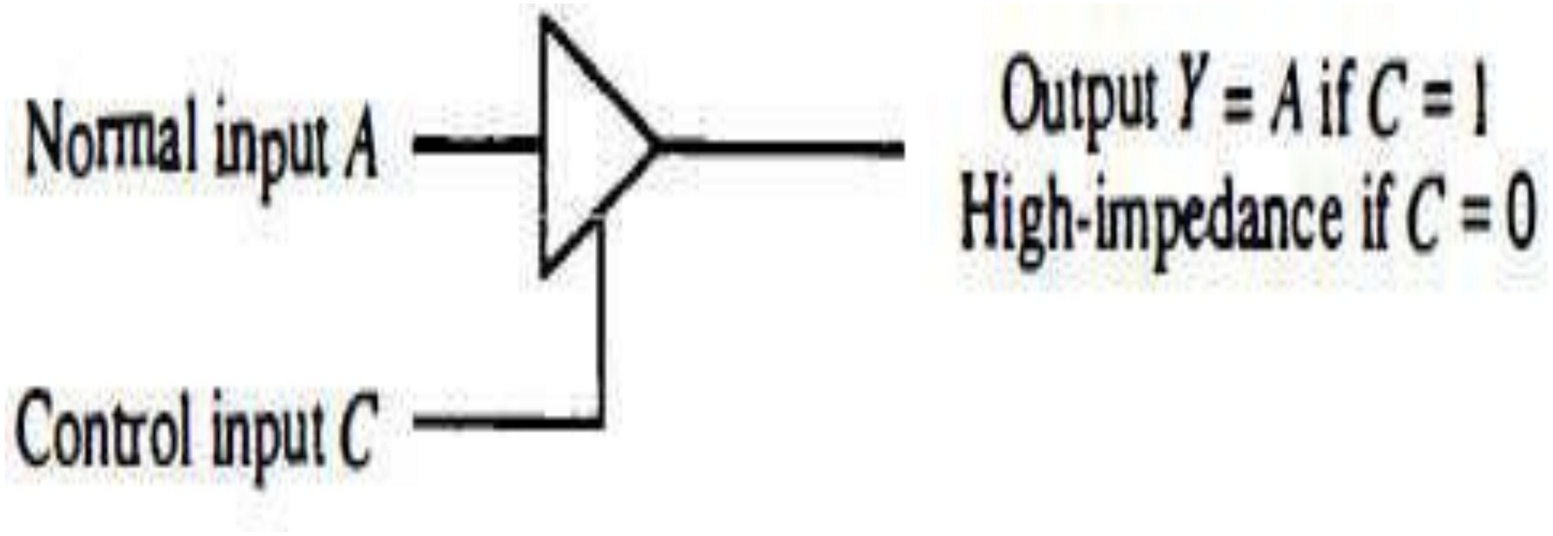


Figure 2.4: Graphic symbol for 3 state buffers

Bus and Memory Transfers

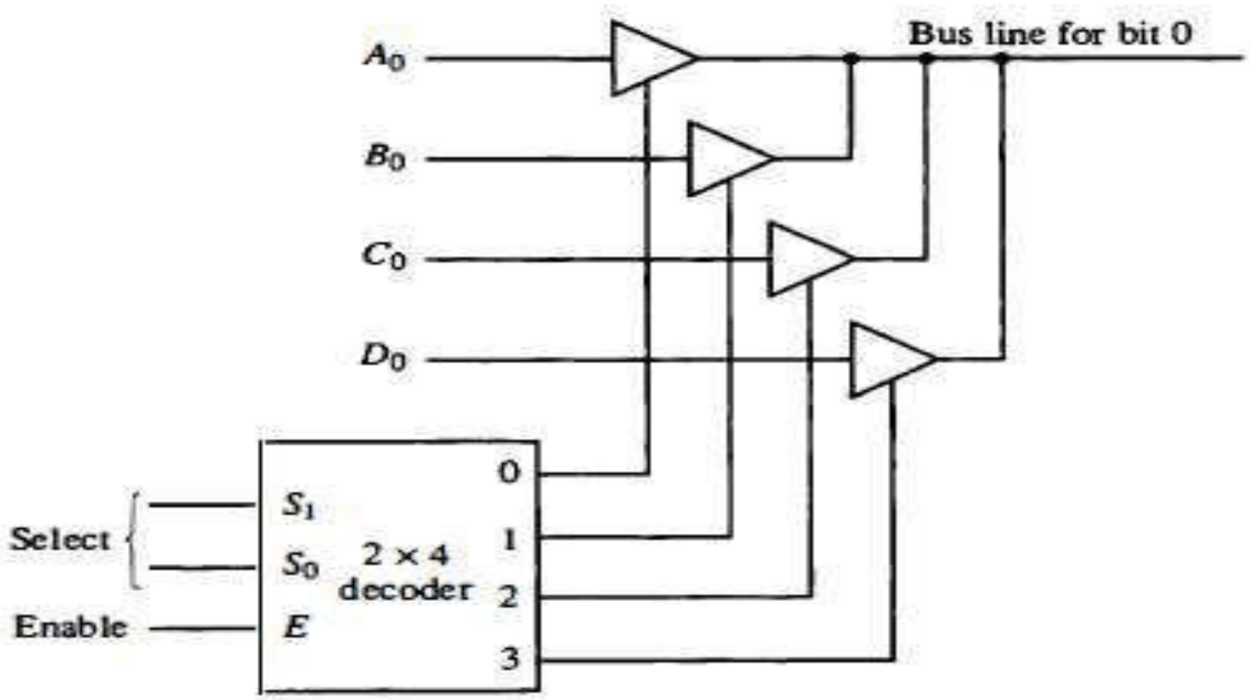


Figure 2.5: Bus line with 3 state buffers

- **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.

TABLE 3 Arithmetic Microoperations

Symbolic designation	Description
$R3 \leftarrow R1 + R2$	Contents of $R1$ plus $R2$ transferred to $R3$
$R3 \leftarrow R1 - R2$	Contents of $R1$ minus $R2$ transferred to $R3$
$R2 \leftarrow \overline{R2}$	Complement the contents of $R2$ (1's complement)
$R2 \leftarrow \overline{R2} + 1$	2's complement the contents of $R2$ (negate)
$R3 \leftarrow R1 + \overline{R2} + 1$	$R1$ plus the 2's complement of $R2$ (subtraction)
$R1 \leftarrow R1 + 1$	Increment the contents of $R1$ by one
$R1 \leftarrow R1 - 1$	Decrement the contents of $R1$ by one

Arithmetic micro operations

- **Binary Adder**
- To implement the add micro operation with hardware, we need the registers that hold the data and the digital component that performs the arithmetic addition.
- The digital circuit that forms the arithmetic sum of two bits

and a previous carry is called a full-adder.

- The digital circuit that generates the arithmetic sum of two

binary numbers of any lengths is called a binary adder.

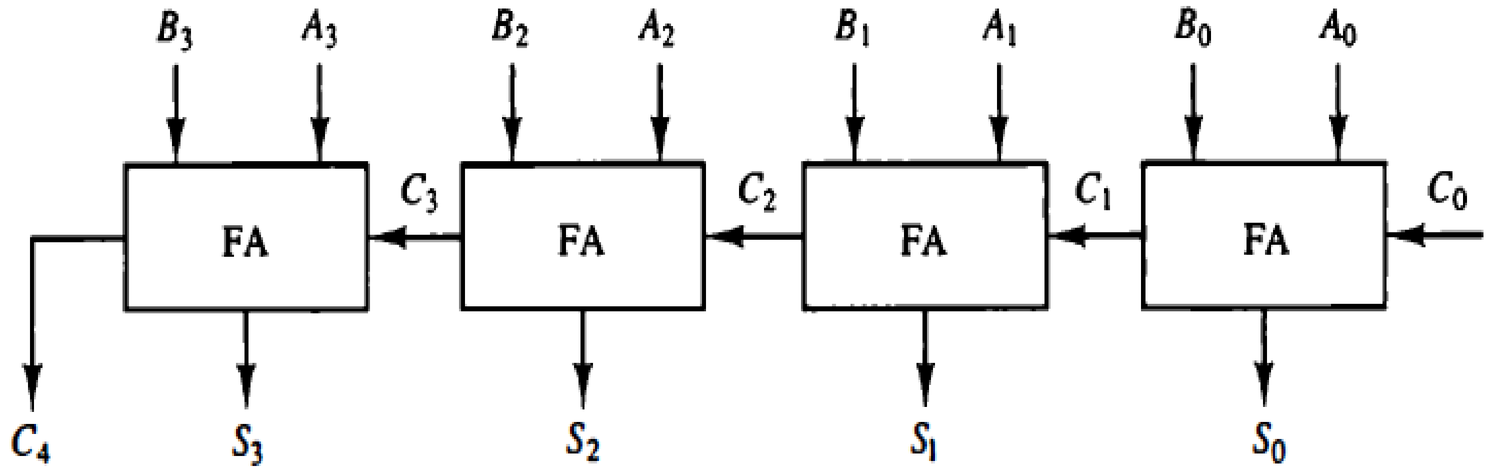


Figure 2.6: Binary Adder

- **Binary Adder - Subtractor**
- The subtraction of binary numbers can be done most conveniently by means of complements as discussed in Sec. 3-
  2. Remember that the subtraction A – B can be done by taking
  the 2's complement of B and adding it to A.

  •The 2's complement can be obtained by taking the 1' s complement and adding one to the least significant pair of bits. The 1's complement can be implemented with inverters and a one can be added to the sum through the input carry.

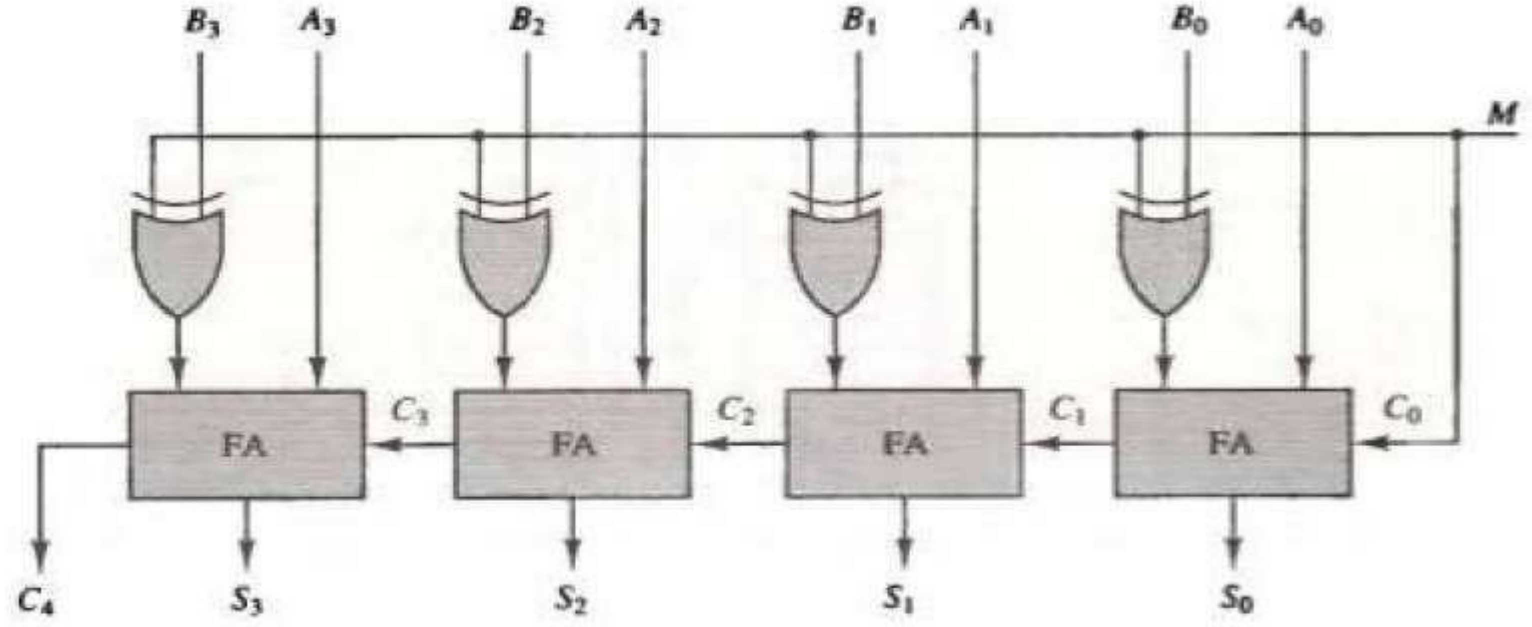


Figure 2.7: 4 - Bit Adder- Subtractor

- **Binary lncrementer**
- The increment micro operation 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 micro operation is easily implemented with a binary counter every time the count enable is active, the clock pulse transition increments the content of the register by one.

- There may be occasions when the increment micro operation must be done with a combinational circuit independent of a particular register.

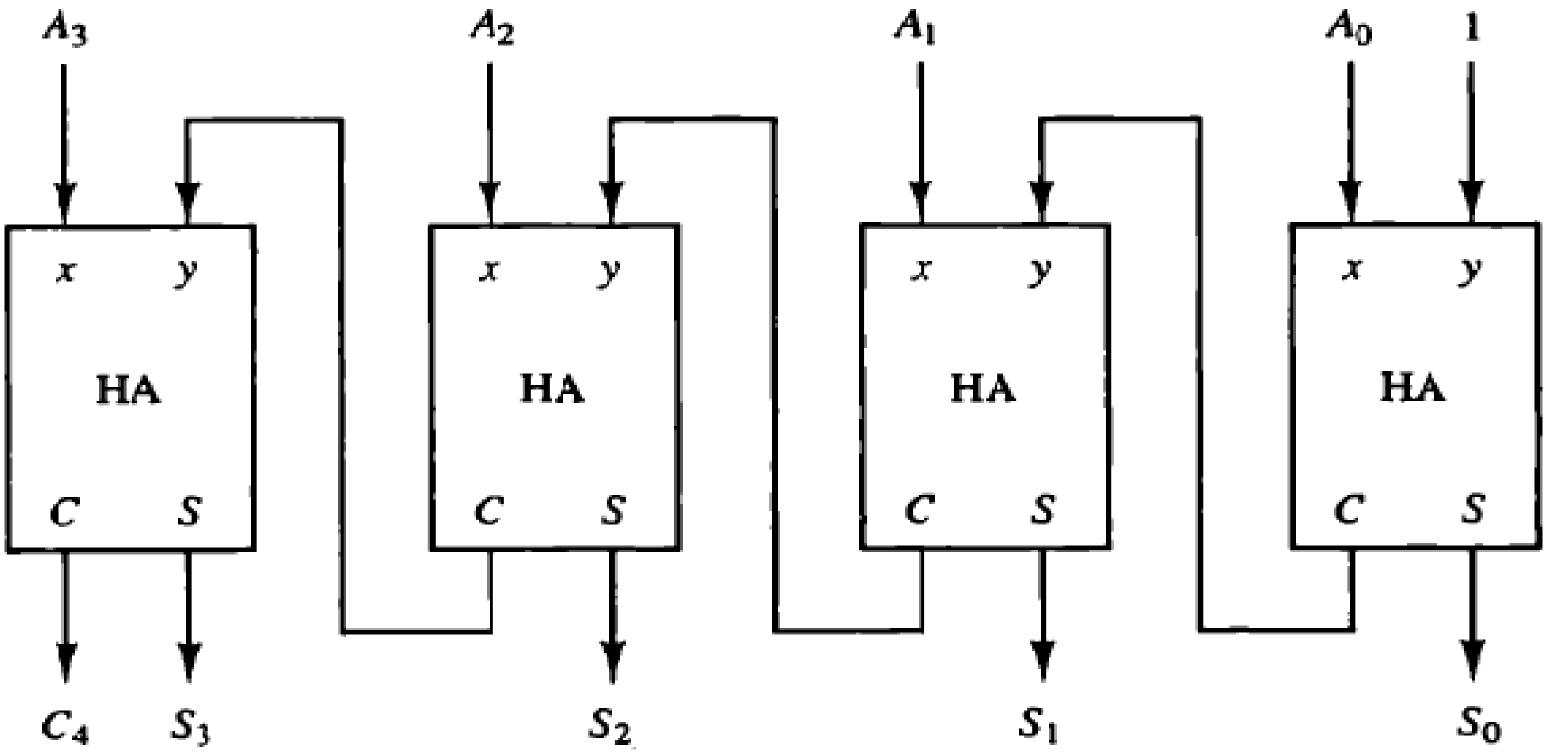


Figure 2.8: 4- bit Binary incrementer

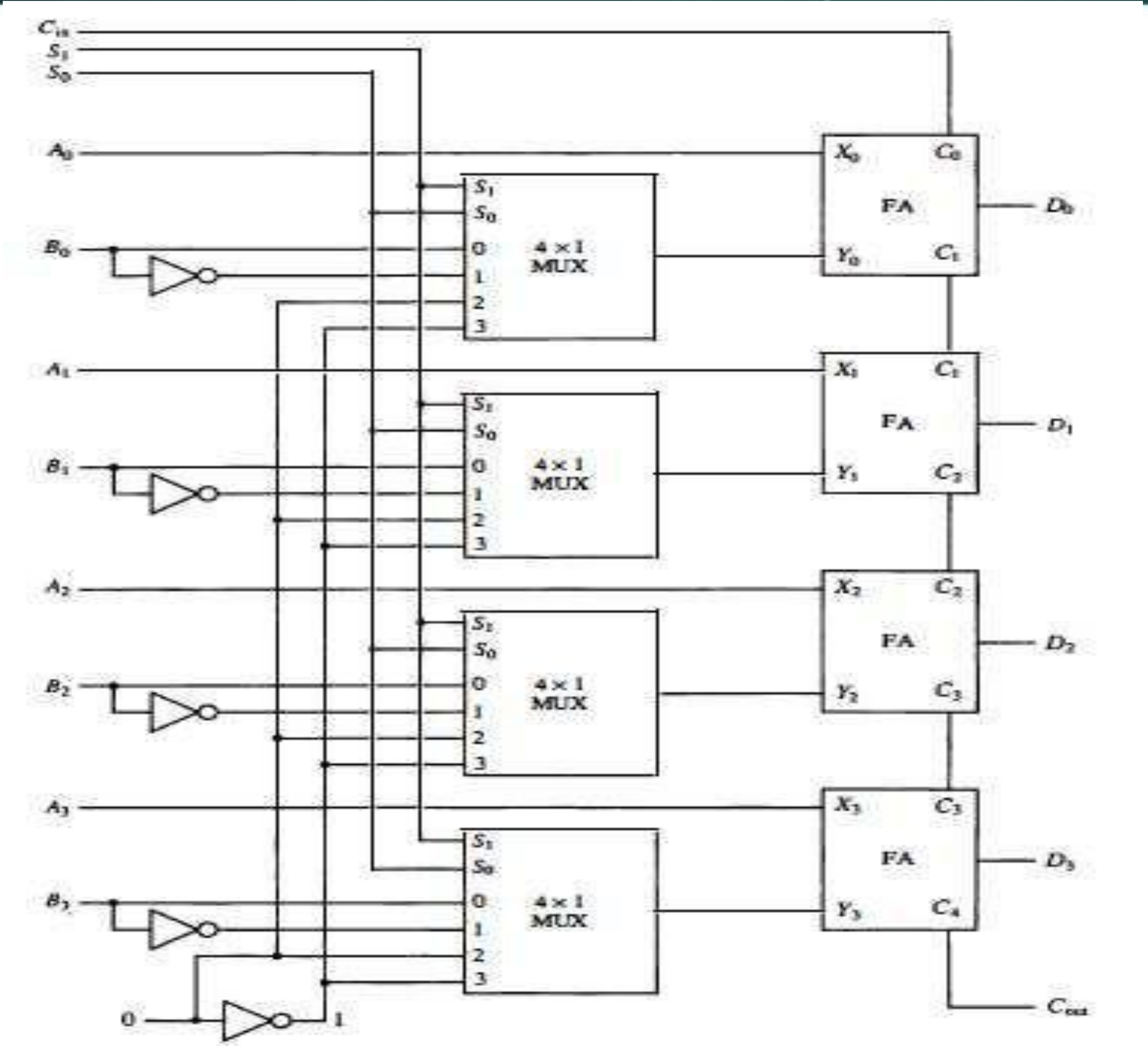
Arithmetic Circuit

- The arithmetic micro operations listed in Table 2.4 can be implemented in one circuit.
- Composite 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 Figure 2.9.
- The output of the binary adder is calculated from the following arithmetic sum:

$$D = A + Y + C_{in}$$

- Where A is the 4-bit binary number at the X inputs and Y is the 4-bit binary number at the Y inputs of the binary adder. C_{in} is the input carry, which can be equal to 0 or 1. Note that the symbol + in the equation above denotes an arithmetic plus.

- By controlling the value of Y with the two selection inputs S1 and S0 and making C_{in} equal to 0 or 1, it is possible to generate the eight arithmetic micro operations listed in Table 2.4



C_{in}
S_1
S_0
A_0
B_0
A_1
B_1
A_2
B_2
A_3
B_3
0
1
S_1
S_0
0
1
2
3
4 × 1 MUX
X_0
C_0
FA
Y_0
C_1
D_0
X_1
C_1
Y_1
C_2
D_1
X_2
C_2
Y_2
C_3
D_2
X_3
C_3
Y_3
C_4
D_3
C_{out}

TABLE 4 Arithmetic Circuit Function Table

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

Logic Microoperations

- Logic micro operations 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 micro operation with the contents of two registers R 1 and R2 is symbolized by the statement
- $P: R1 \leftarrow R1 \oplus R2$

- It specifies a logic micro operation to be executed on the individual bits of the registers provided that the control variable P = 1.
- As a numerical example, assume that each register has four bits. Let the content of R1 be 1010 and the content of R2 be 1 100. The exclusive-OR microoperation stated above symbolizes the following logic computation:
-
- 1010 Content of R 1
- 1 100 Content of R2
- 0110 Content of R 1 after P = 1

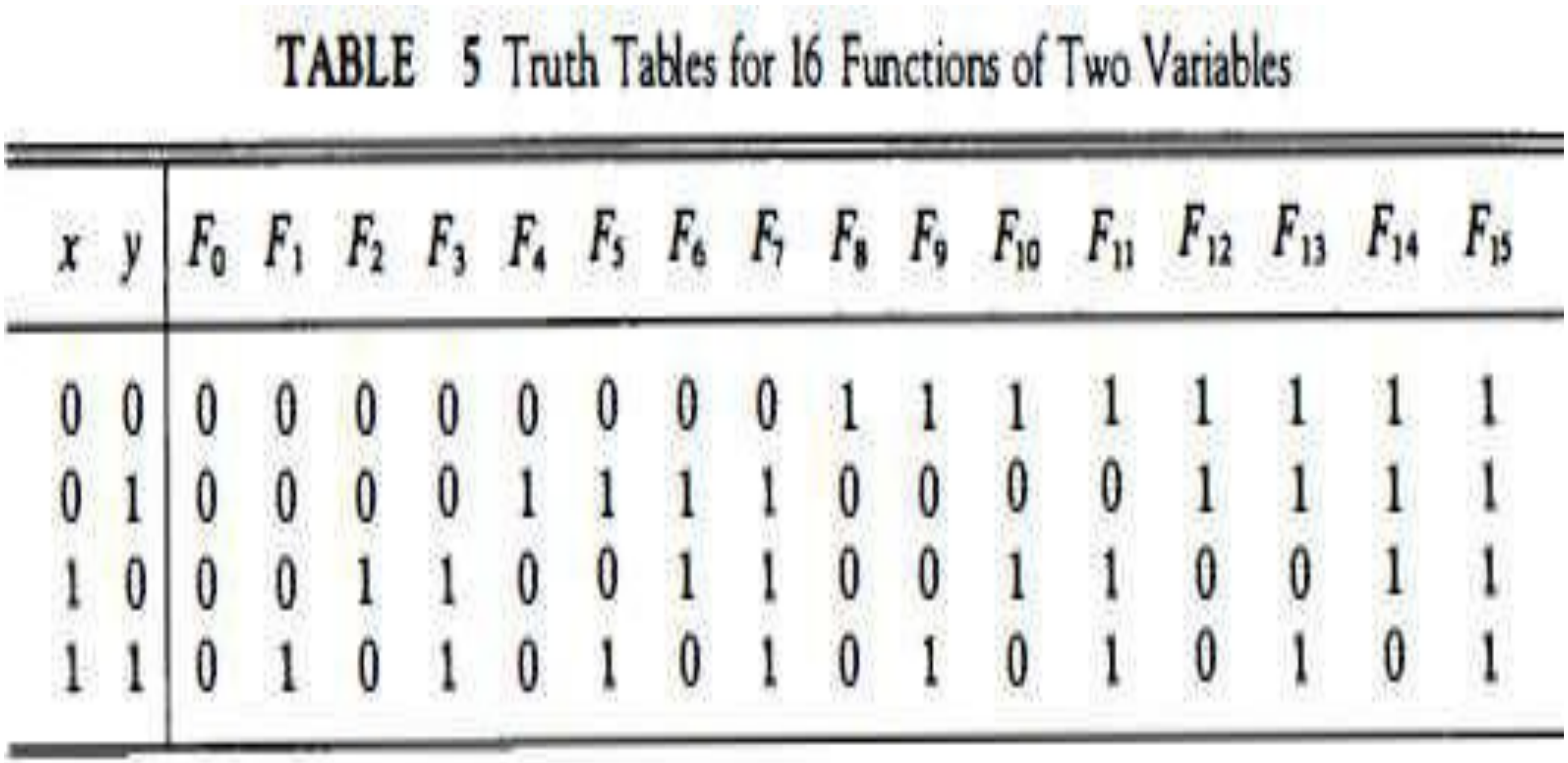


TABLE 5 Truth Tables for 16 Functions of Two Variables

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

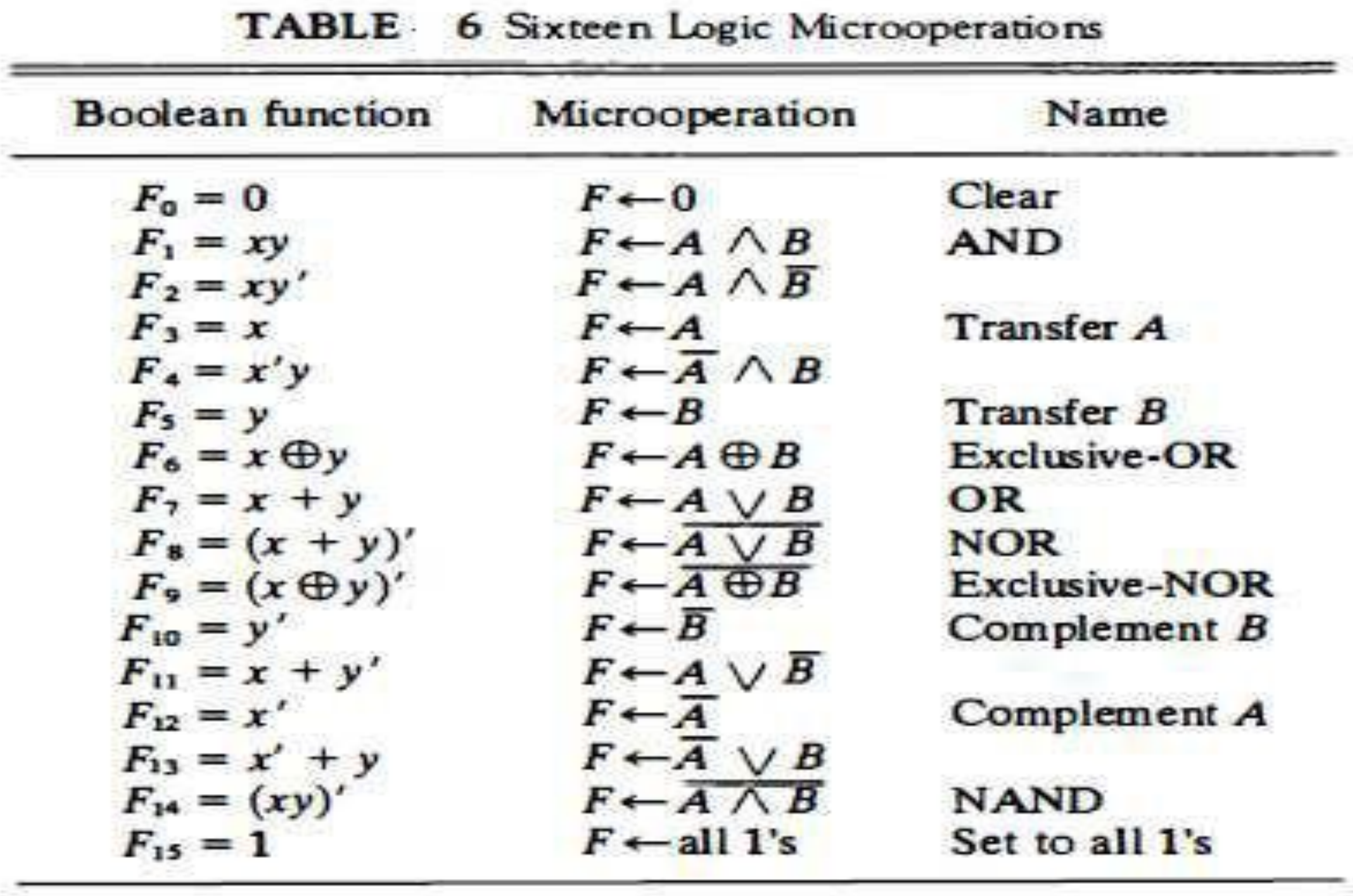
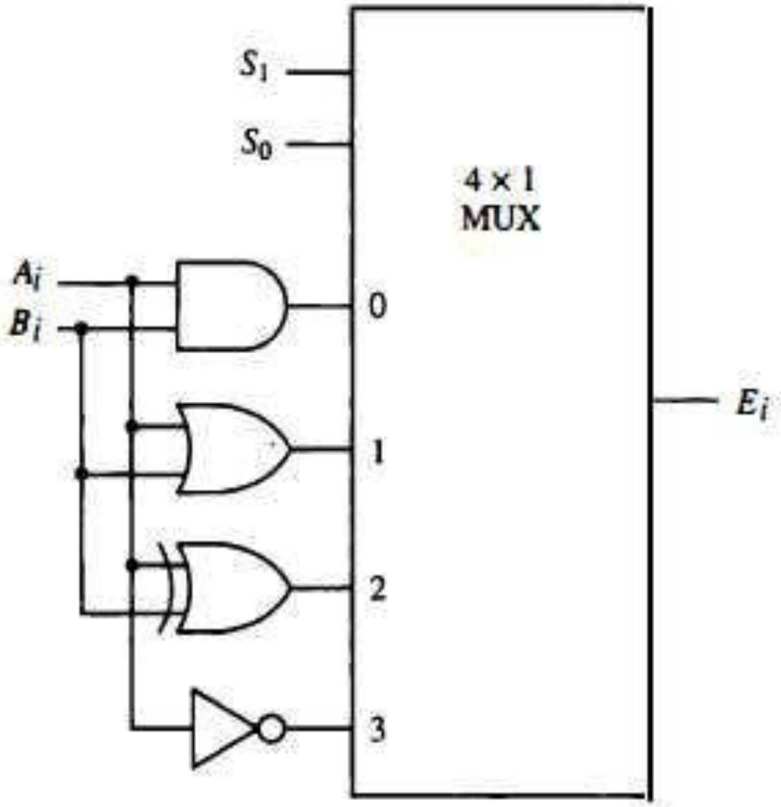


TABLE 6 Sixteen Logic Microoperations

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

Figure 10 One stage of logic circuit.



(a) Logic diagram

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

(b) Function table

Shift Microoperations

- Shift micro operations are used for serial transfer of data. They are also used in conjunction with arithmetic, logic, and other data-processing operations.
- The information transferred through the serial input determines the type of shift. There are three types of shifts: logical, circular, and arithmetic.
- A logical shift is one that transfers 0 through the serial input. We will adopt the symbols SHL and SHR for logical shift-left and shift-right micro operations. For example:
-
- $R1 \leftarrow \text{SHL } R1$
- $R2 \leftarrow \text{SHR } R2$

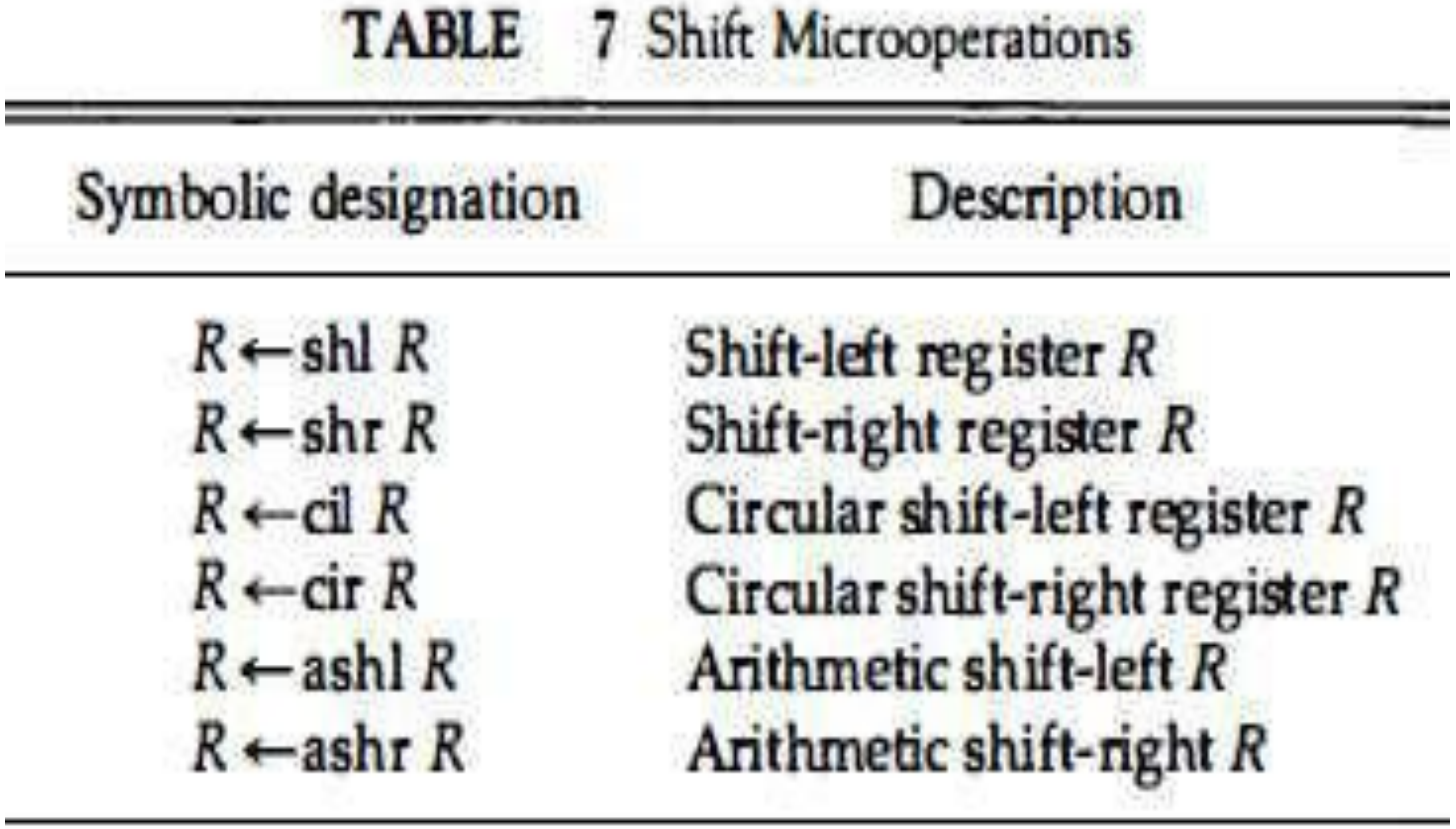


TABLE 7 Shift Microoperations

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

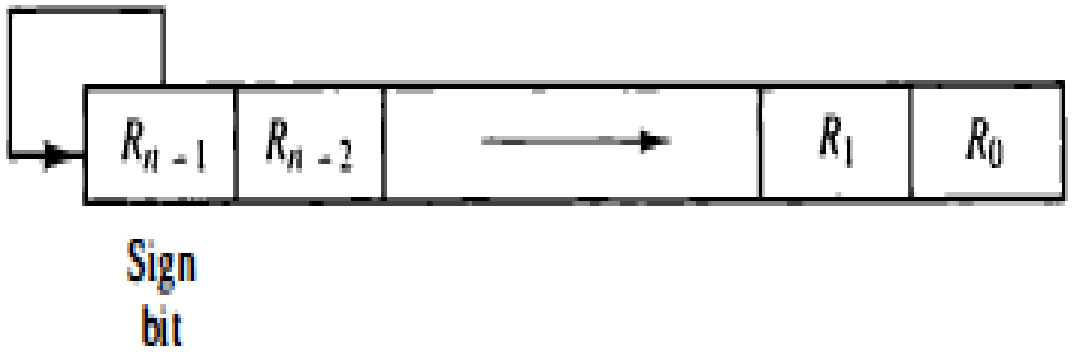


Figure 11 Arithmetic shift right.

- **Arithmetic Logic Shift Unit**
- 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 Figure 2.12.
- Figure 2.12 provides eight arithmetic operation, four logic

operations, and two shift operations.

- Table 2.8 lists the 14 operations of the ALU. The first eight are arithmetic operations and are selected with 5352 = 00. The next four are logic operations and are selected with 5352 = 01.

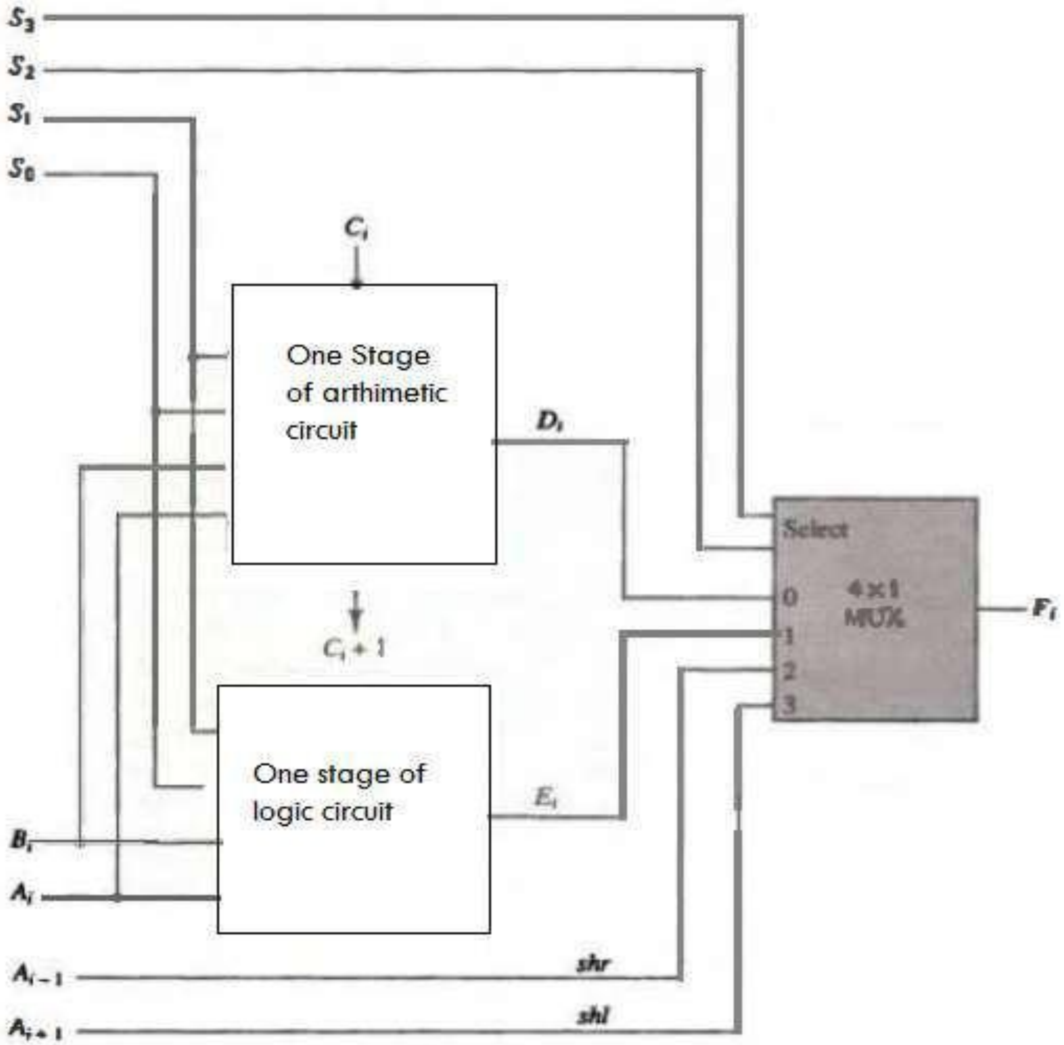

S_3
S_2
S_1
S_0
C_i
One Stage of arthimetic circuit
D_i
$C_i + 1$
One stage of logic circuit
E_i
B_i
A_i
A_{i-1}
shr
A_{i+1}
shl
Select
0
1
2
3
4×1 MUX
F_i

TABLE 8 Function Table for Arithmetic Logic Shift Unit

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

Control memory

- The control memory can be a read-only memory (ROM).The content of the words in ROM are fixed and cannot be altered by simple programming since no writing capability is available in the ROM.

- ROM words are made permanent during the hardware production of the unit. The use of a micro program involves placing all control variables in words of ROM for use by the control unit through successive read operations.

- The content of the word in ROM at a given address specifies a microinstruction. A more advanced development known as dynamic microprogramming

- The general configuration of a micro programmed control unit

is demonstrated in the block diagram of Figure 2.13.

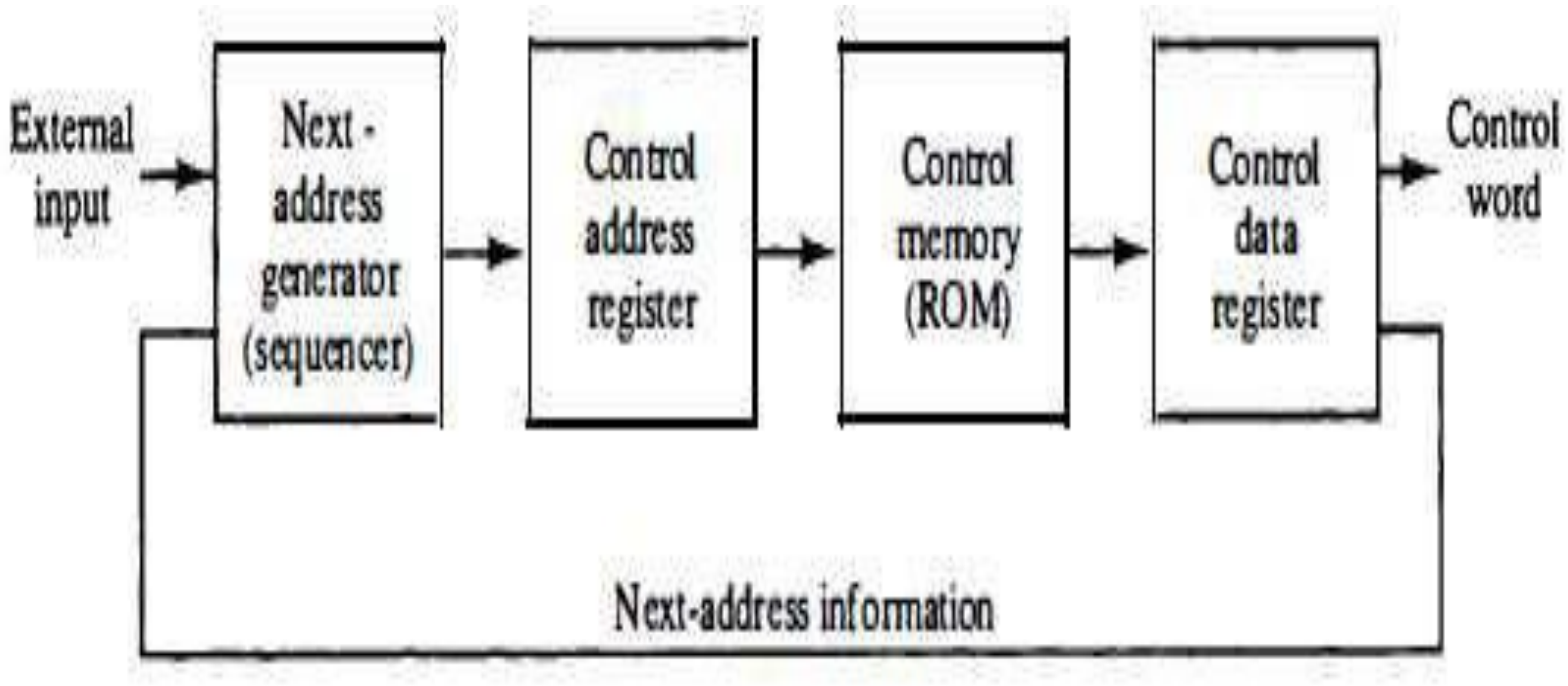
- The control memory is assumed to be a ROM, within which all

control information is permanently stored.

- The control memory address register specifies the address o f the microinstruction, and the control data register holds the microinstruction read from memory.

•The microinstruction contains a control word that specifies one or more micro operations for the data processor. Once these operations are executed, the control must determine the next address.

Micro programmed control



Address Sequencing

- Microinstructions are stored in control memory in groups, with each group specifying a routine.

- Each computer instruction has its own micro program routine in control memory to generate the micro operations that execute the instruction.

- 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.

- This address is usually the address of the first microinstruction

that activates the instruction fetch routine.

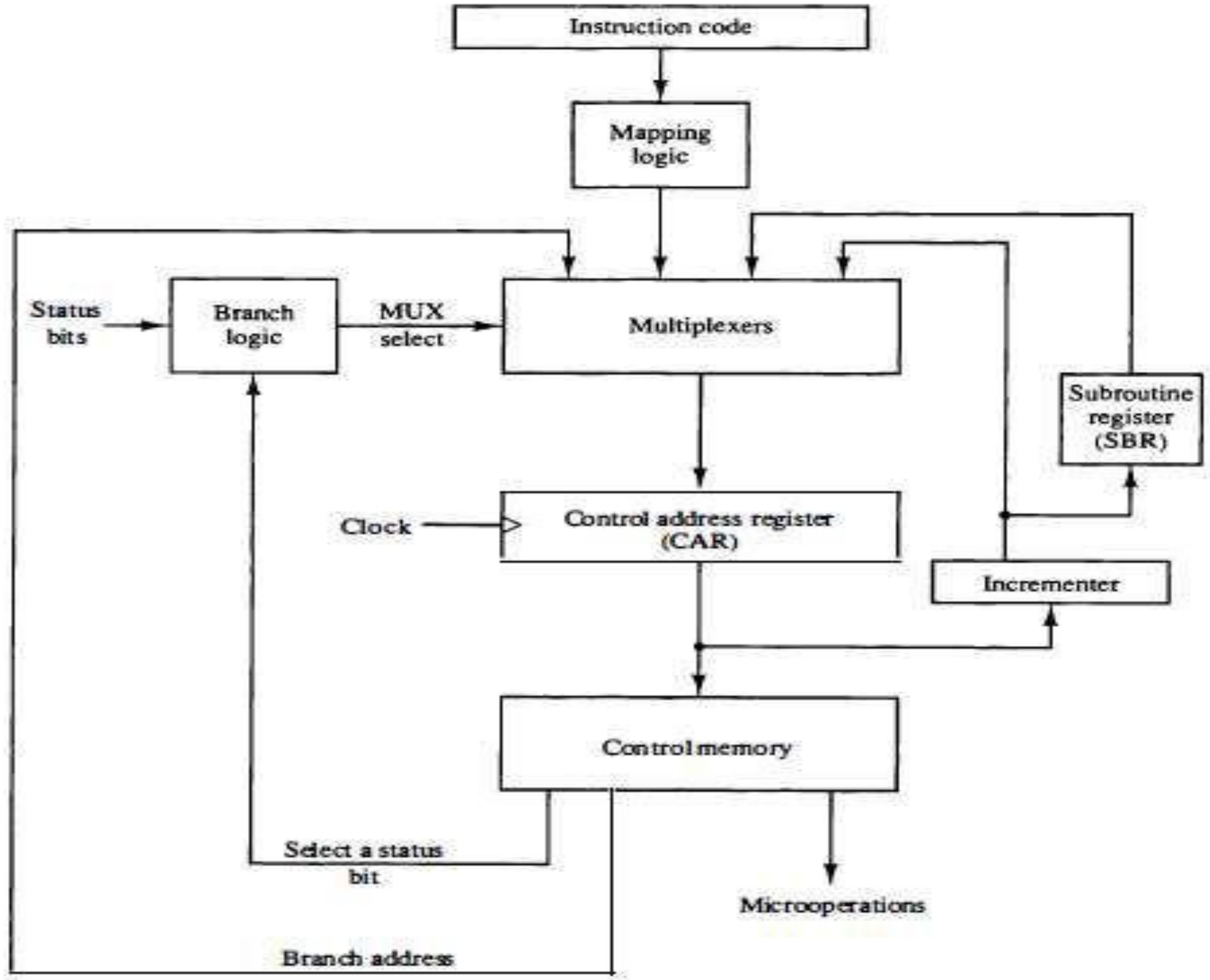
- The fetch routine may be sequenced by incrementing the control address register through the rest of its microinstructions.

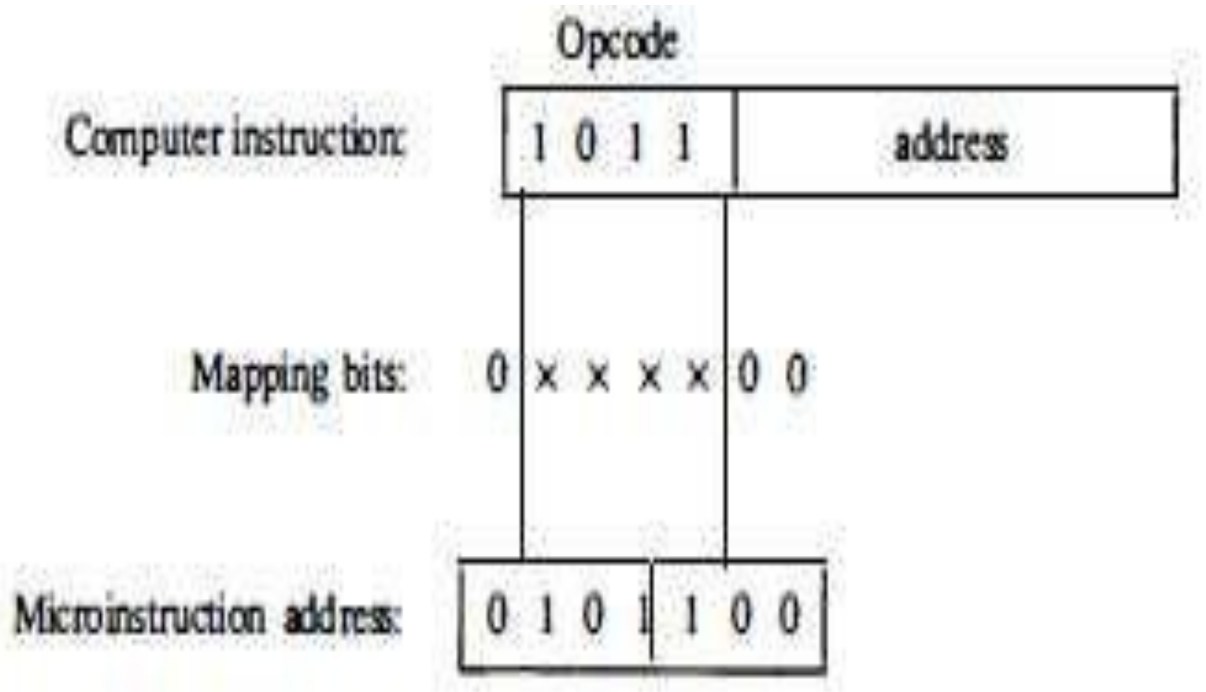
- At the end of the fetch routine, the instruction isin the

instruction register of the computer

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

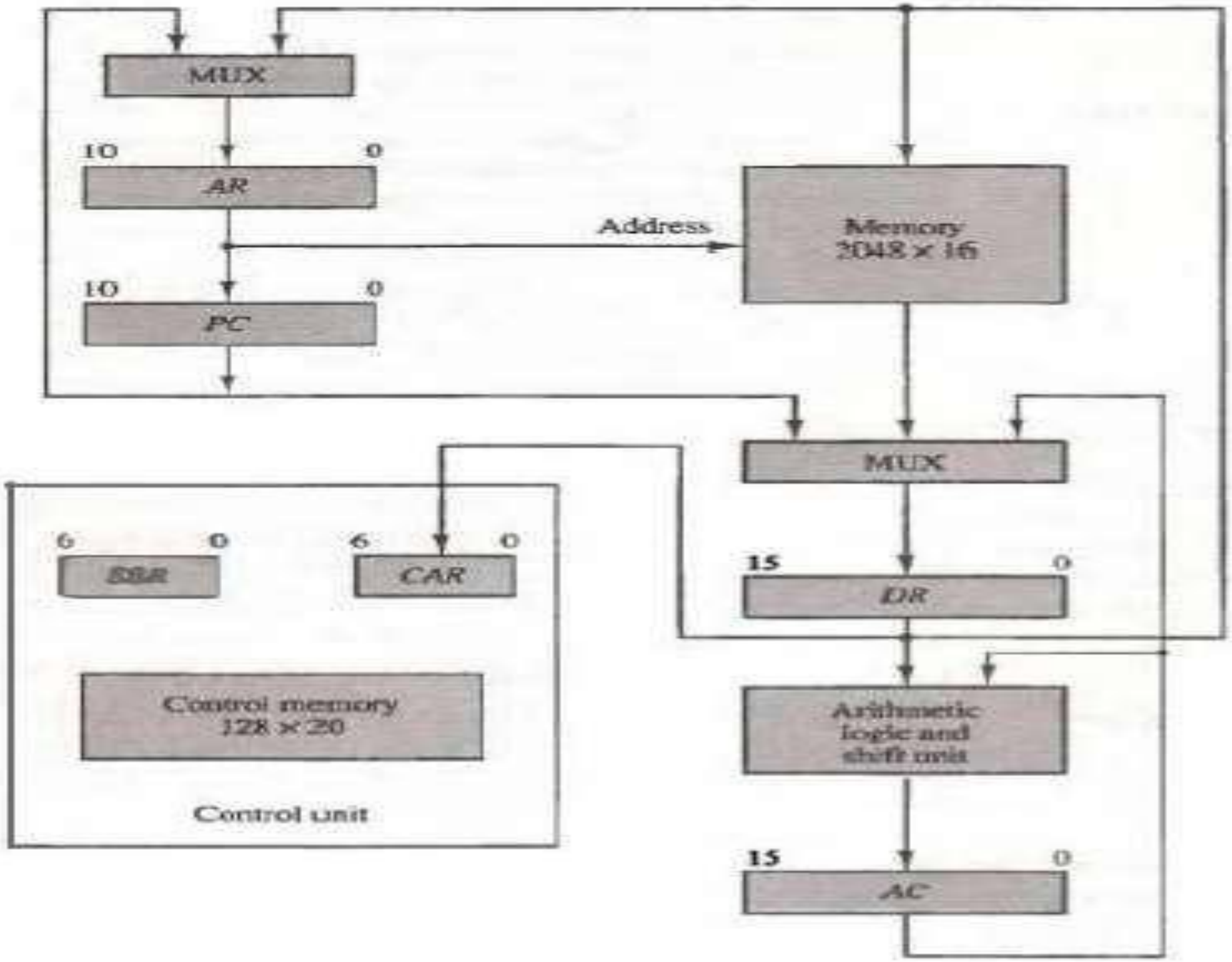

Instruction code
Mapping logic
Status bits
Branch logic
MUX select
Multiplexers
Subroutine register (SBR)
Clock
Control address register (CAR)
Incrementer
Control memory
Select a status bit
Microoperations
Branch address


Opcode
Computer instruction:
1 0 1 1
address
Mapping bits:
0 x x x x 0 0
Microinstruction address:
0 1 0 1 1 0 0

- **Subroutines**
- Subroutines are programs that are used by other routines to accomplish a particular task. A subroutine can be called from any point within the main body of the micro program.
- Frequently, many micro programs contain identical sections of code. Microinstructions can be saved by employing subroutines that use common sections of microcode.
- For example, the sequence of microoperations needed to generate the effective address of the operand for an instruction is common to all memory reference instructions.

- This sequence could be a subroutine that is called from within many other routines to execute the effective address computation.

- Micro programs that use subroutines must have a provision for storing the return address during a subroutine call and restoring the address during a subroutine return.

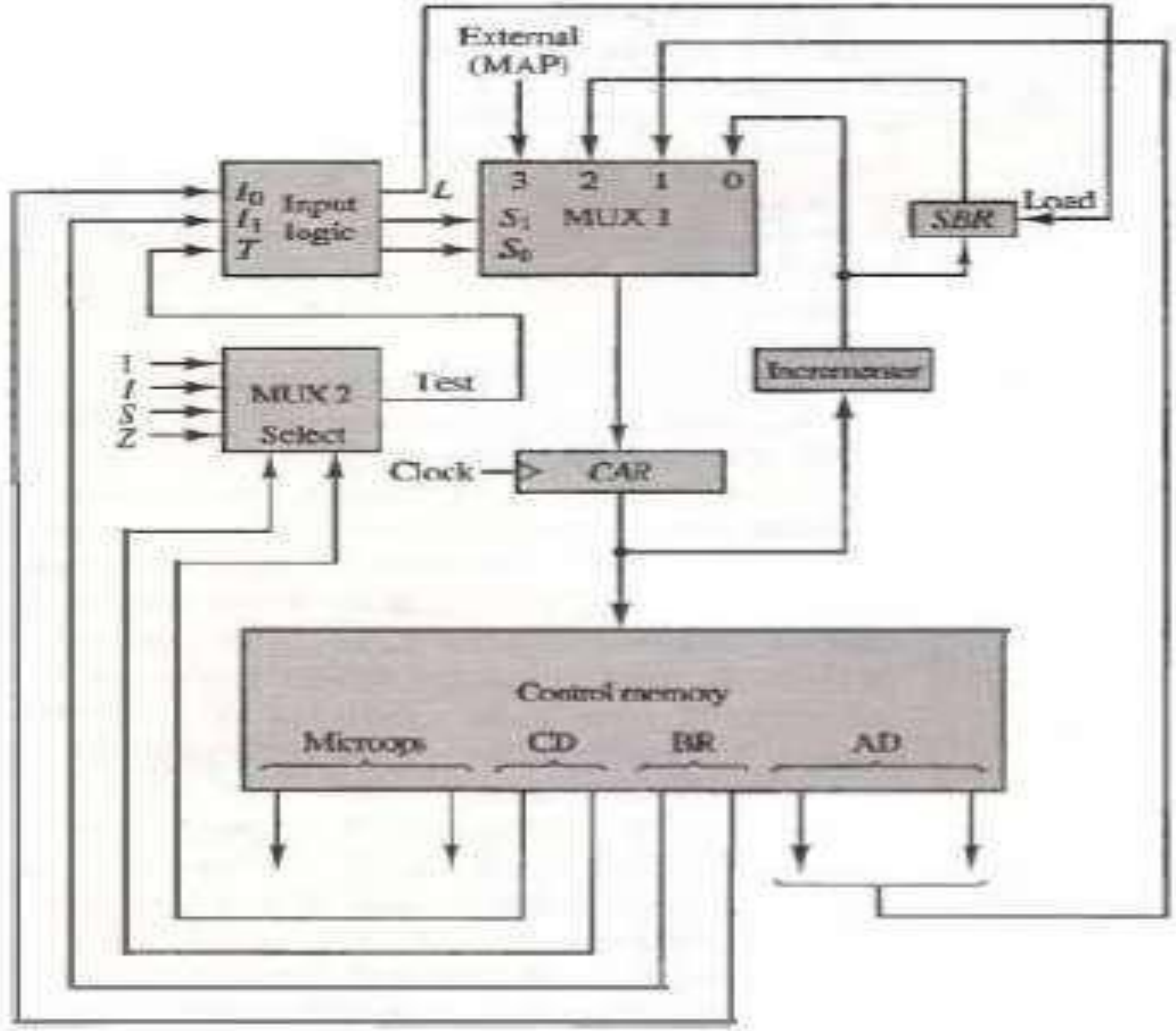

MUX
10
0
AR
Address
Memory
2048 × 16
10
0
PC
MUX
6
0
SBR
6
0
CAR
15
0
DR
Control memory
128 × 20
Arithmetic
logic and
shift unit
Control unit
15
0
AC

Micro program Example

- A micro program sequencer can be constructed with digital functions to suit a particular application.
- To guarantee a wide range of acceptability, an integrated circuit sequencer must provide an internal organization that can be adapted to a wide range of applications.
- The control memory is included in the diagram to show the interaction between the sequencer and the memory attached to it.

- The binary values of the two selection variables determine the path in the multiplexer.

- For example, with S1, So = 10, multiplexer input number 2 is selected and establishes a transfer path from SBR to CAR. Note that each of the four inputs as well as the output of MUX 1 contains a 7-bit address.


External (MAP)
3
2
1
0
I_0
I_1
T
Input logic
L
S_1
S_0
MUX 1
SBR
Load
Incrementer
1
I
S
Z
MUX 2
Select
Test
Clock
CAR
Control memory
Microops
CD
BR
AD

Design of Control Unit

- The bits of the microinstruction are usually divided into fields, with each field defining a distinct, separate function.

- The various fields encountered in instruction formats provide control bits to initiate microoperations in the system, special bits to specify the way that the next address is to be evaluated, and an address field for branching.

- The number of control bits that initiate microoperations can be reduced by grouping mutually exclusive variables into fields and encoding the k bits in each field to provide 2' microoperations.

- Figure 2.18 shows the three decoders and some of the connections that must be made from their outputs.

- Each of the three fields of the microinstruction presently available in the output of control memory are decoded with a 3 x 8 decoder to provide eight outputs.

- As shown in Figure 2.18 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.

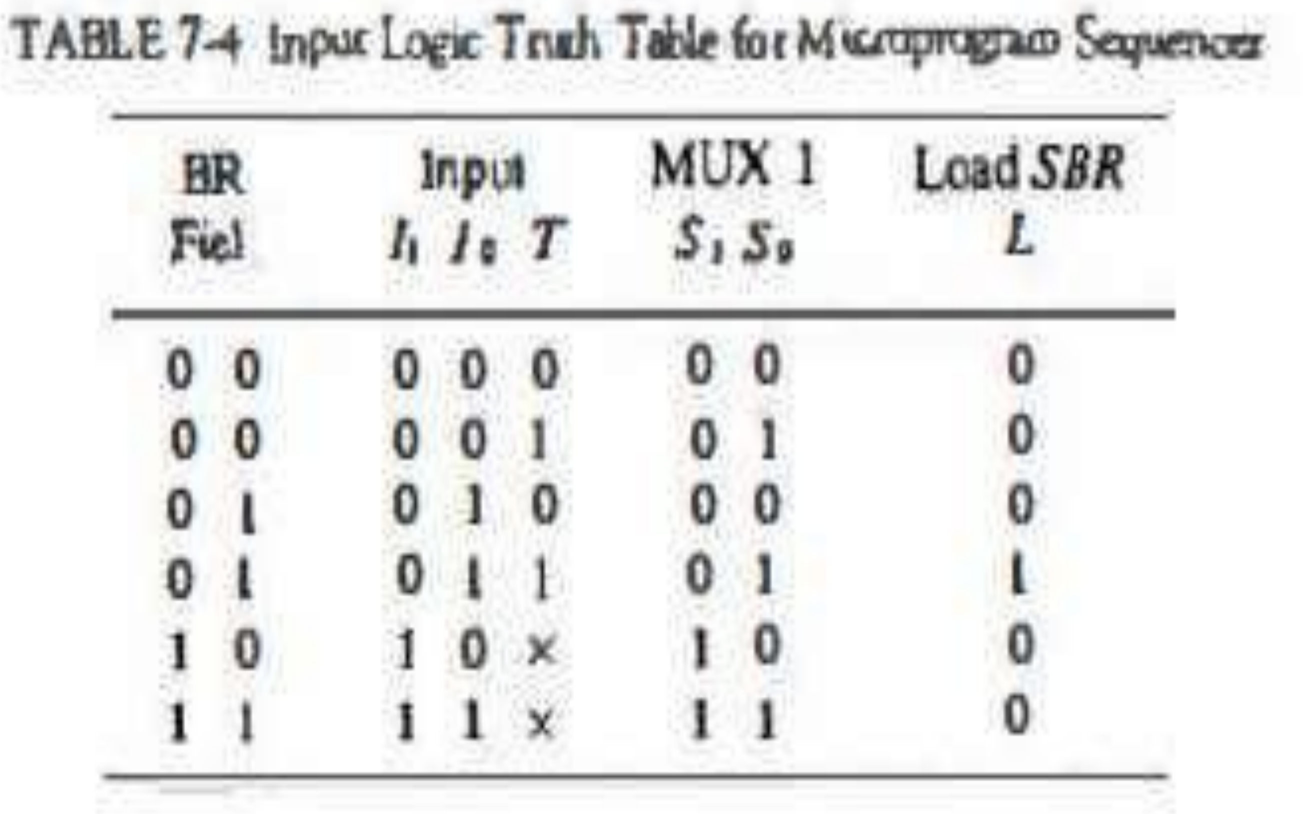
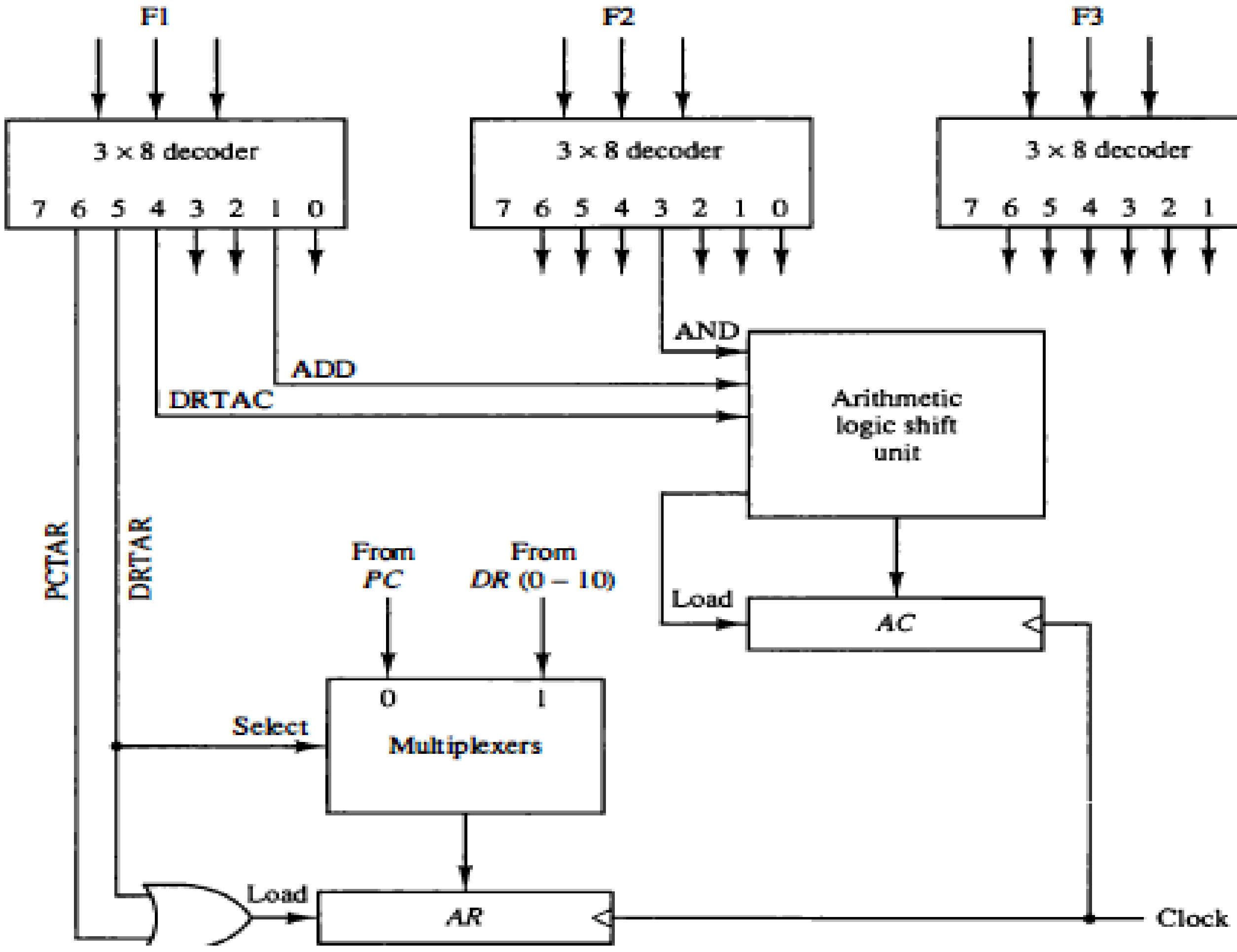


TABLE 7-4 Input Logic Truth Table for Microprogram Sequencer

BR Fiel	Input I_1 I_0 T	MUX 1 S_1 S_0	Load SBR L
0 0	0 0 0	0 0	0
0 0	0 0 1	0 1	0
0 1	0 1 0	0 0	0
0 1	0 1 1	0 1	1
1 0	1 0 ×	1 0	0
1 1	1 1 ×	1 1	0

- The multiplexers select the information from DR when output 5 is active and from PC when output 5 is inactive, as shown in Figure 2.18. The other outputs of the decoders that are associated with an AC operation must also be connected to the arithmetic logic shift unit in a similar fashion.

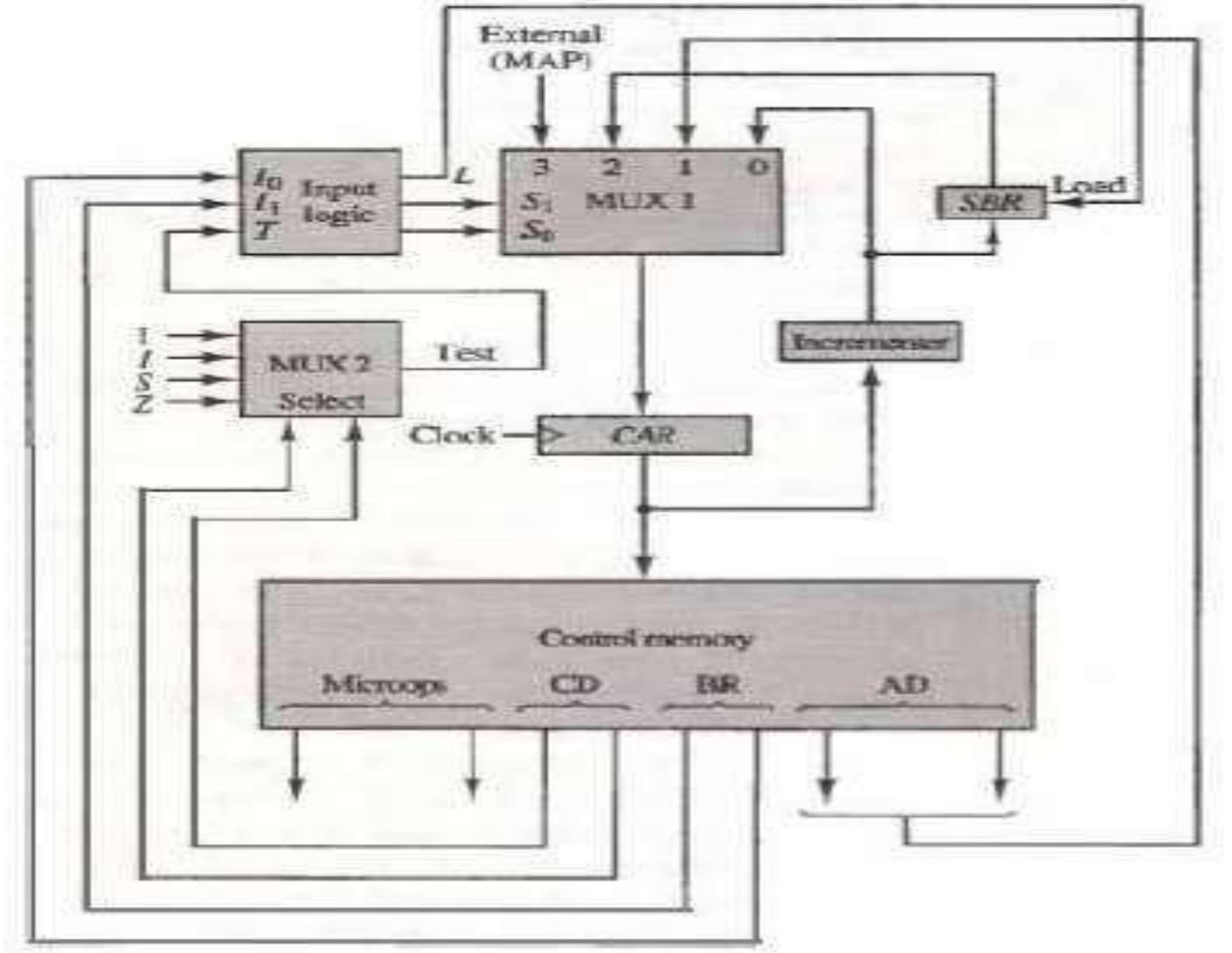

F1
F2
F3
3 × 8 decoder
7 6 5 4 3 2 1 0
3 × 8 decoder
7 6 5 4 3 2 1 0
3 × 8 decoder
7 6 5 4 3 2 1 0
AND
ADD
DRTAC
Arithmetic logic shift unit
PCTAR
DRTAR
From PC
From DR (0 – 10)
Load
AC
0
1
Select
Multiplexers
Load
AR
Clock

Design of Control Unit

- A microprogram sequencer can be constructed with digital functions to suit a particular application.
- However, just as there are large ROM units available in integrated circuit packages, so are general-purpose sequencers suited for the construction of microprogram control units.
- To guarantee a wide range of acceptability, an integrated circuit sequencer must provide an internal organization that can be adapted to a wide range of applications.
- The block diagram of the microprogram sequencer is shown in Fig. 19.

- The control memory is included in the diagram to show the interaction between the sequencer and the memory attached to it.

- There are two multiplexers in the circuit. The first multiplexer selects an address from one of four sources and routes it into a control address register CAR .


External
(MAP)
3
2
1
0
MUX 1
S_1
S_0
I_0
I_1
T
Input logic
L
SBR
Load
Incrementer
1
I
S
Z
MUX 2
Select
Test
Clock
CAR
Control memory
Microops
CD
BR
AD

- The input logic circuit in Fig. 19 has three inputs, l0, l1, and T, and three outputs, S0, S1, and L.Variables So and S, select one of the source addresses for CAR .
- Variable L enables the load input in SBR. The binary values of the two selection variables determine the path in the multiplexer.
- For example, with S1 So = 10, multiplexer input number 2 is selected and establishes a transfer path from SBR to CAR. Note that each of the four inputs as well as the output of MUX 1 contains a 7-bit address.

Instruction Cycle

- A microprogram sequencer can be constructed with digital functions to suit a particular application.
- However, just as there are large ROM units available in integrated circuit packages, so are general-purpose sequencers suited for the construction of microprogram control units.
- To guarantee a wide range of acceptability, an integrated circuit sequencer must provide an internal organization that can be adapted to a wide range of applications.
- The block diagram of the microprogram sequencer is shown in Fig. 19.

- An **instruction cycle** (sometimes called a **fetch–decode–execute cycle**) is the basic operational process of a computer.
- It is the process by which a computer retrieves a program instruction from its memory, determines what actions the instruction dictates, and carries out those actions.
- This cycle is repeated continuously by a computer's central processing unit (CPU), from boot-up to when the computer is shut down.

- In simpler CPUs the instruction cycle is executed sequentially, each instruction being processed before the next one is started.

- In most modern CPUs the instruction cycles are instead executed concurrently, and often in parallel, through an instruction pipeline: the next instruction starts being processed before the previous instruction has finished, which is possible because the cycle is broken up into separate steps.

- **Program counter (PC)**
- **Memory address registers (MAR)**
- **Memory data register (MDR)**
- **Instruction registers (IR)**
- **Control unit (CU)**

- Fetch the instruction
- Decode the instruction
- Read the effective address
- Execute the instruction

Data Representation

The data types found in the registers of digital computers may be classified as being one of the following categories:

- Numbers used in arithmetic computations,
- Letters of the alphabet used in data processing.
- Other discrete symbols used for specific purposes.

Number Systems

- A number system of base, or radix, r is a system that uses distinct symbols for r digits. Numbers are represented by a string of digit symbols.

- To determine the quantity that the number represents, it is necessary to multiply each digit by an integer power of r and then form the sum of all weighted digits.

- For example, the decimal number system in everyday use employs the radix 10 system.

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

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

Octal 736.4 is converted to decimal as follows:

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

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

- **Binary Numeral System - Base-2**
- Binary numbers uses only 0 and 1 digits.
- B denotes binary prefix.
- Examples:
- 101012 = 10101B = 1×24+0×23+1×22+0×21+1×20 = 16+4+1= 21
- 101112 = 10111B = 1×24+0×23+1×22+1×21+1×20 = 16+4+2+1= 23
- 1000112 = 100011B = 1×25+0×24+0×23+0×22+1×21+1×20 =32+2+1= 35

Data Representation

The data types found in the registers of digital computers may be classified as being one of the following categories:

- Numbers used in arithmetic computations,
- Letters of the alphabet used in data processing.
- Other discrete symbols used for specific purposes.

- **Octal Numeral System - Base-8**

- Octal numbers uses digits from 0...7.
- Examples:
- 278 = 2×81+7×80 = 16+7 = 23
- 308 = 3×81+0×80 = 24
- 43078 = 4×83+3×82+0×81+7×80= 2247

- **Decimal Numeral System - Base-10**

- Decimal numbers uses digits from 0...9.
- These are the regular numbers that we use.
- Example:
- 253810 = 2×103+5×102+3×101+8×100

- **Octal Numeral System - Base-8**

- Octal numbers uses digits from 0...7.

- Examples:
- 278 = 2×81+7×80 = 16+7 = 23
- 308 = 3×81+0×80 = 24
- 43078 = 4×83+3×82+0×81+7×80= 2247

- **Hexadecimal Numeral System - Base-16**

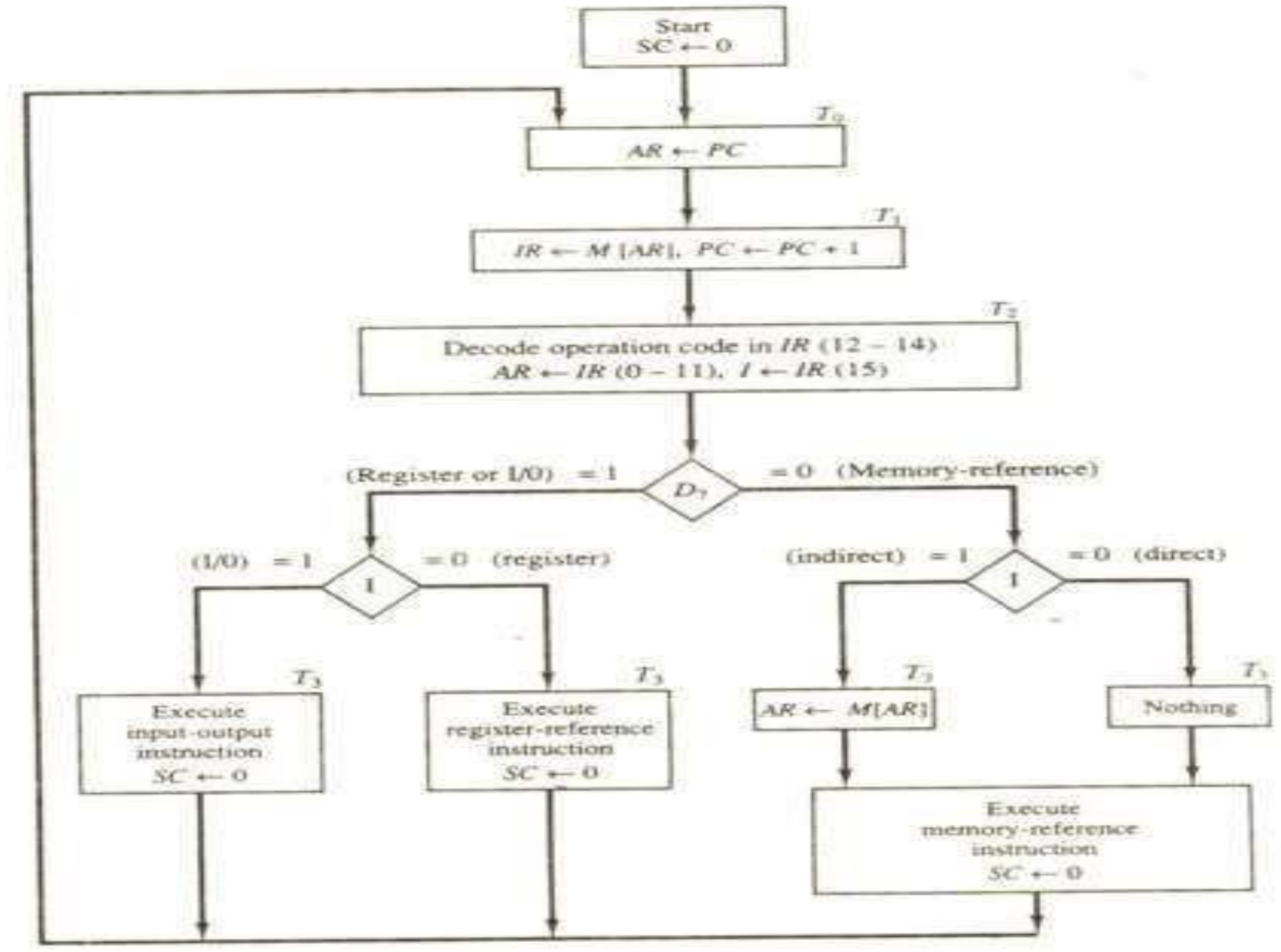
- Hex numbers uses digits from 0...9 and A...F.
- H denotes hex prefix.
- Examples:
- 2816 = 28H = 2×161+8×160 = 40
- 2F16 = 2FH = 2×161+15×160 = 47
- BC1216 = BC12H = 11×163+12×162+1×161+2×160= 48146

Memory Reference Instructions

DETERMINE THE TYPE OF INSTRUCTION:

•The timing signal that is active after the decoding is T3. During time T3, the control unit determines the type of instruction that was just read from memory.

•The following flowchart presents an initial configuration for the instruction cycle and shows how the control determines the instruction type after the decoding


Start
SC ← 0
T0
AR ← PC
T1
IR ← M[AR], PC ← PC + 1
T2
Decode operation code in IR (12 – 14)
AR ← IR (0 – 11), I ← IR (15)
(Register or I/O) = 1
D7
= 0 (Memory-reference)
(I/O) = 1
I
= 0 (register)
(indirect) = 1
I
= 0 (direct)
T3
Execute input-output instruction SC ← 0
T3
Execute register-reference instruction SC ← 0
T3
AR ← M[AR]
T3
Nothing
Execute memory-reference instruction SC ← 0

- **TYPES OF INSTRUCTIONS:**
- The basic computer has three 16-bi instruction code formats,

- 1. Memory Reference Instructions
- 2. Register Reference Instructions
- 3. Input/output Instructions

- **Memory reference instruction,**
- First 12 bits (0-11) specify an address. The address field is denoted by three x's (in hexadecimal notation) and is equivalent to 12-bit address. The last mode bit of the instruction represents by symbol I
- Next 3 bits specify operation code (opcode).
- Left most bit specify the addressing mode I
- I = 0 for direct address and I = 1 for indirect address. When I = 0, the last four bits of an instruction have a hexadecimal digit equivalent from 0 to 6 since the last bit is zero (0). When I = 1 the last four bits of an instruction have a hexadecimal digit equivalent from 8 to E since the last bit is one (1)

Table 3.1: Memory Reference Instructions

	Hexadecimal code		
Symbol	I = 0	I = 1	Description
AND	0xxx	8xxx	AND memory word to AC
ADD	1xxx	9xxx	ADD memory word to AC
LDA	2xxx	Axxx	LOAD Memory word to AC
STA	3xxx	Bxxx	Store content of AC in memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and save return address
ISZ	6xxx	Exxx	Increment and Skip if zero

- Register Reference Instructions

In Register Reference Instruction: First 12 bits (0-11) specify the register operation.

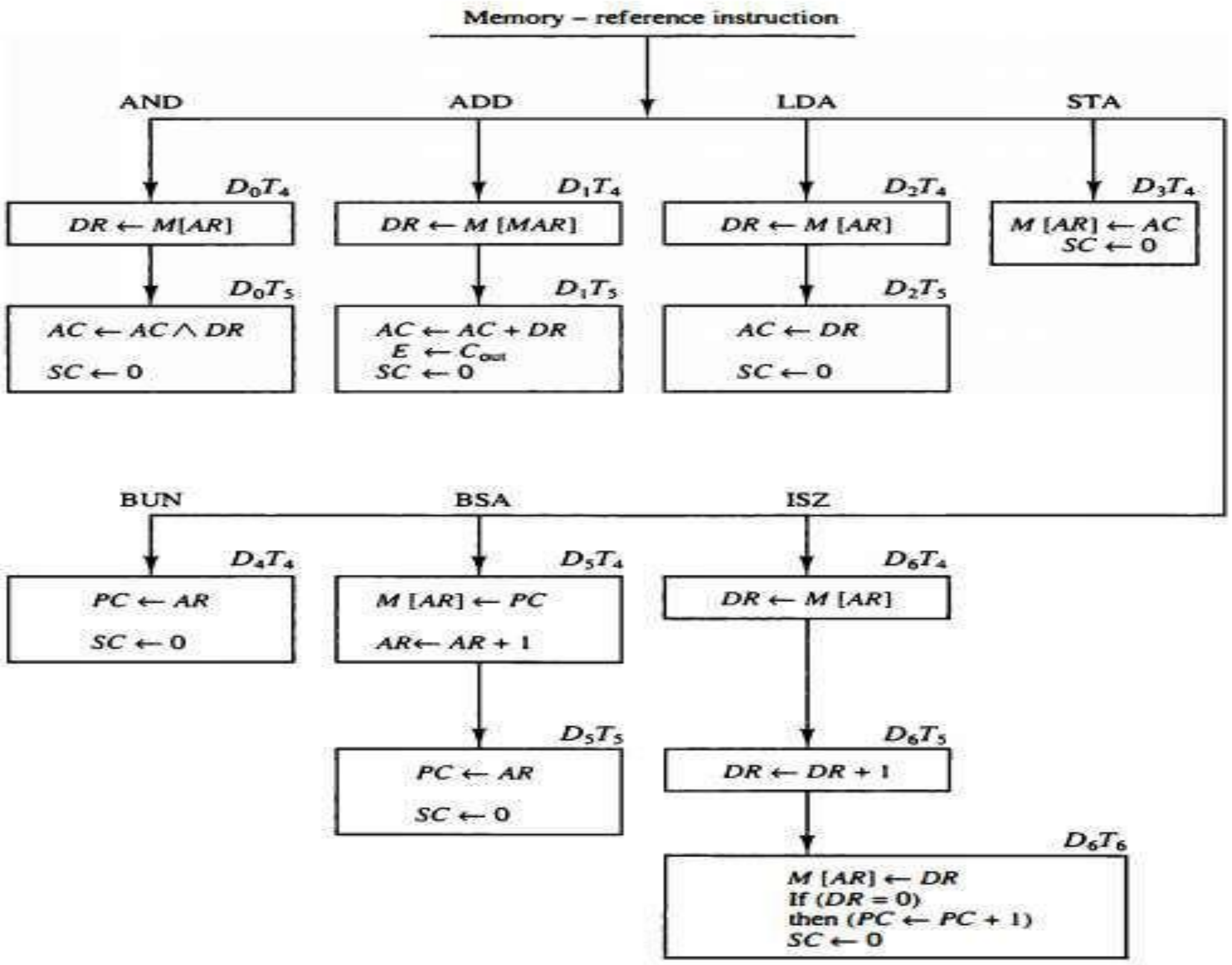
- The next three bits equals to 111 specify opcode.
- The last mode bit of the instruction is 0.
- Therefore, left most 4 bits are always 0111 which is equal to hexadecimal 7

- **I/O Reference Instructions**

- In I/O Reference Instruction: First 12 bits (0-11) specify the I/O operation.

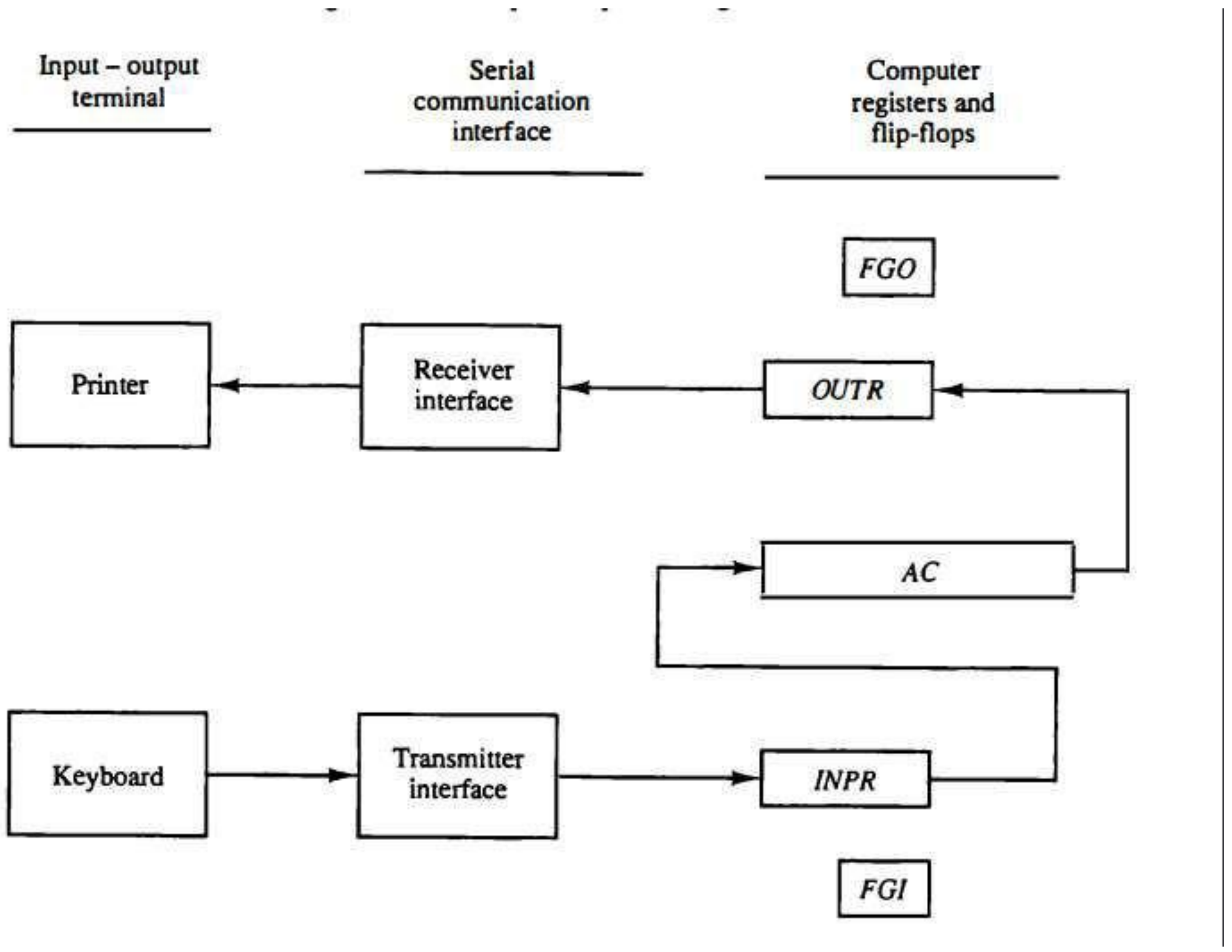
- The next three bits equals to 111 specify opcode.
- The last mode bit of the instruction is 1.
- Therefore, left most 4 bits are always 1111 which is equal to hexadecimal F.

Flowchart for memory-



Input-output

- **Input-output:**
- A computer can serve no useful purpose unless it communicates with the external environment.
- Instructions and data stored in memory must come from some input device.
- Commercial computers include many types of input and output devices.
- To demonstrate the most basic requirements for input and output communication.


Input – output terminal
Serial communication interface
Computer registers and flip-flops
FGO
Printer
Receiver interface
OUTR
AC
Keyboard
Transmitter interface
INPR
FGI

- **TYPES OF INSTRUCTIONS:**
- The basic computer has three 16-bi instruction code formats,

- 1. Memory Reference Instructions
- 2. Register Reference Instructions
- 3. Input/output Instructions

Input-Output Configuration

- The terminals send and receive serial information the 8-bits information is shifted from the keyboard into the input register (INPR).
- Information for the printer is stored in OUTR. These two registers communicate in the communication interface serially and with the
- Accumulator in parallel. The transmitter interface receives the information and transfer to INPR. The receiver interface receives from OUTR and transmits to the printer.
- The input register INPR consists of 8-bits and stores alphanumeric information. FGI is 1-bit controlled flip-flop or flag. A flag bit is set to 1.

- Input and Output interactions with electromechanical peripheral devices require huge processing times compared with CPU processing times. – I/O (milliseconds) versus CPU (nano/microseconds)

- Interrupts permit other CPU instructions to execute while waiting for I/O to complete.

- The I/O interface, instead of the CPU, monitors the I/O device.

Interrupt

Program Interrupt

•Programmed controlled procedure is to let the external device inform the computer when it is ready for the transfer.

•In the meantime the computer can be busy with other tasks. This type of transfer uses the interrupt facility.

•While the computer is running a program, it does not check the flags.

•However, when a flag is set, the computer is momentarily interrupted from proceeding with the current program and is informed of the fact that a flag has been set.

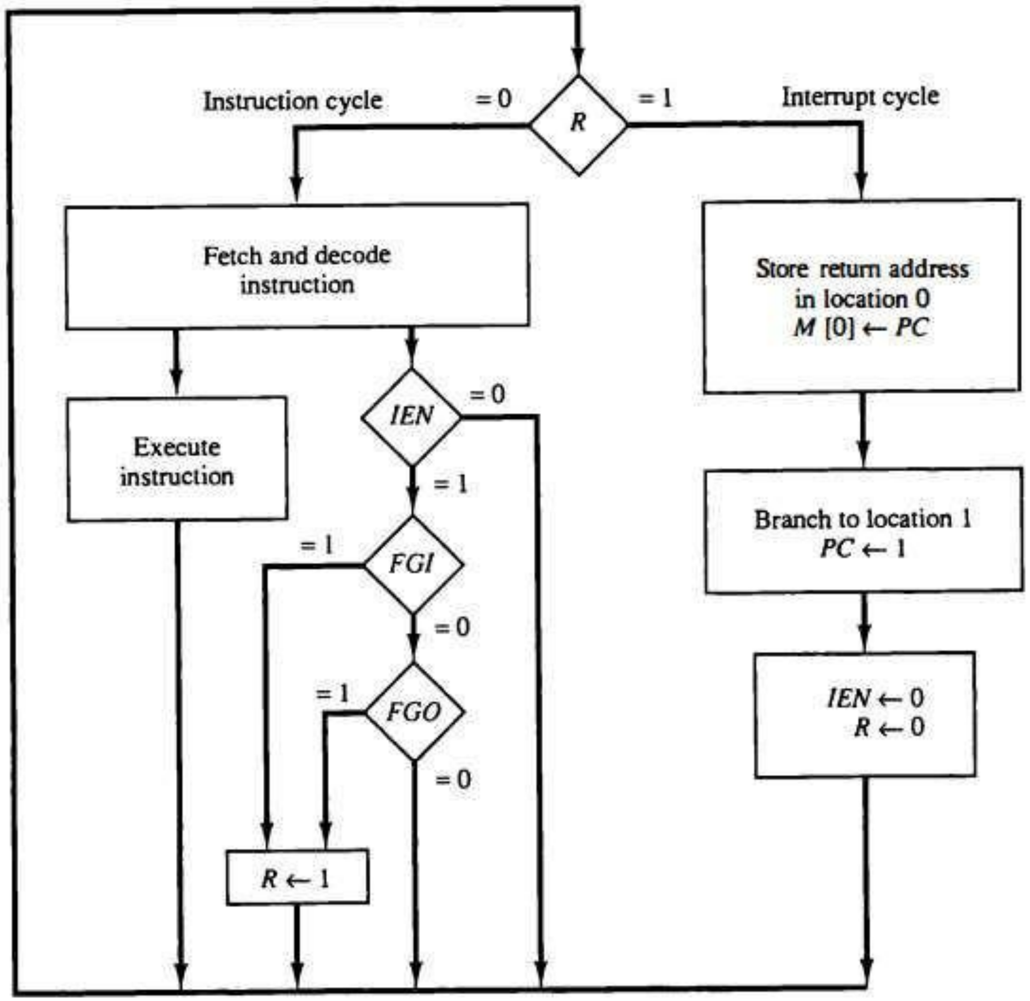
•The computer is wasting time while checking the flag instead of doing some other useful processing task.

- The interrupt enable flip-flop IEN can be set and cleared with two instructions.
- When IEN is cleared to 0 (with the IOF instruction), the flags cannot interrupt the computer.
- When IEN is set to 1 (with the ION instruction), the computer can be interrupted.
- These two instructions provide the programmer with the capability of making a decision as to whether or not to use the interrupt facility.

- The way that the interrupt is handled by the computer can be

explained by means of the flowchart of Figure 3.4.

- An interrupt flip-flop R is included in the computer. When R = 0, the computer goes through an instruction cycle.



Instruction cycle
= 0
R
= 1
Interrupt cycle
Fetch and decode instruction
Store return address in location 0 M [0] ← PC
Execute instruction
IEN
= 0
= 1
FGI
= 1
= 0
FGO
= 1
= 0
R ← 1
Branch to location 1 PC ← 1
IEN ← 0 R ← 0

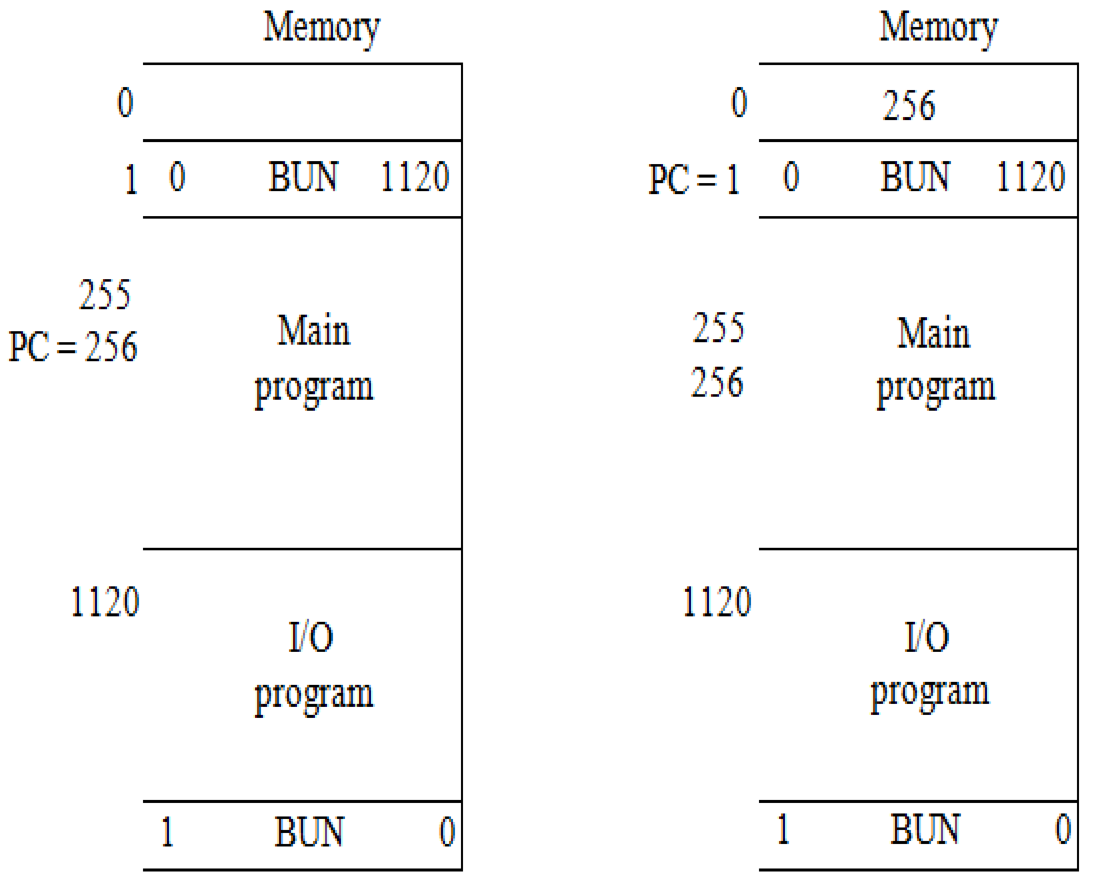
Interrupt

- **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 where it can be found later when the program returns to the instruction at which it was interrupted.

- This location may be a processor register, a memory stack, or a specific memory location



(a) Before interrupt (b) After interrupt cycle

Register transfer operation in interrupt cycle

The register transfer statements for the interrupt cycle.

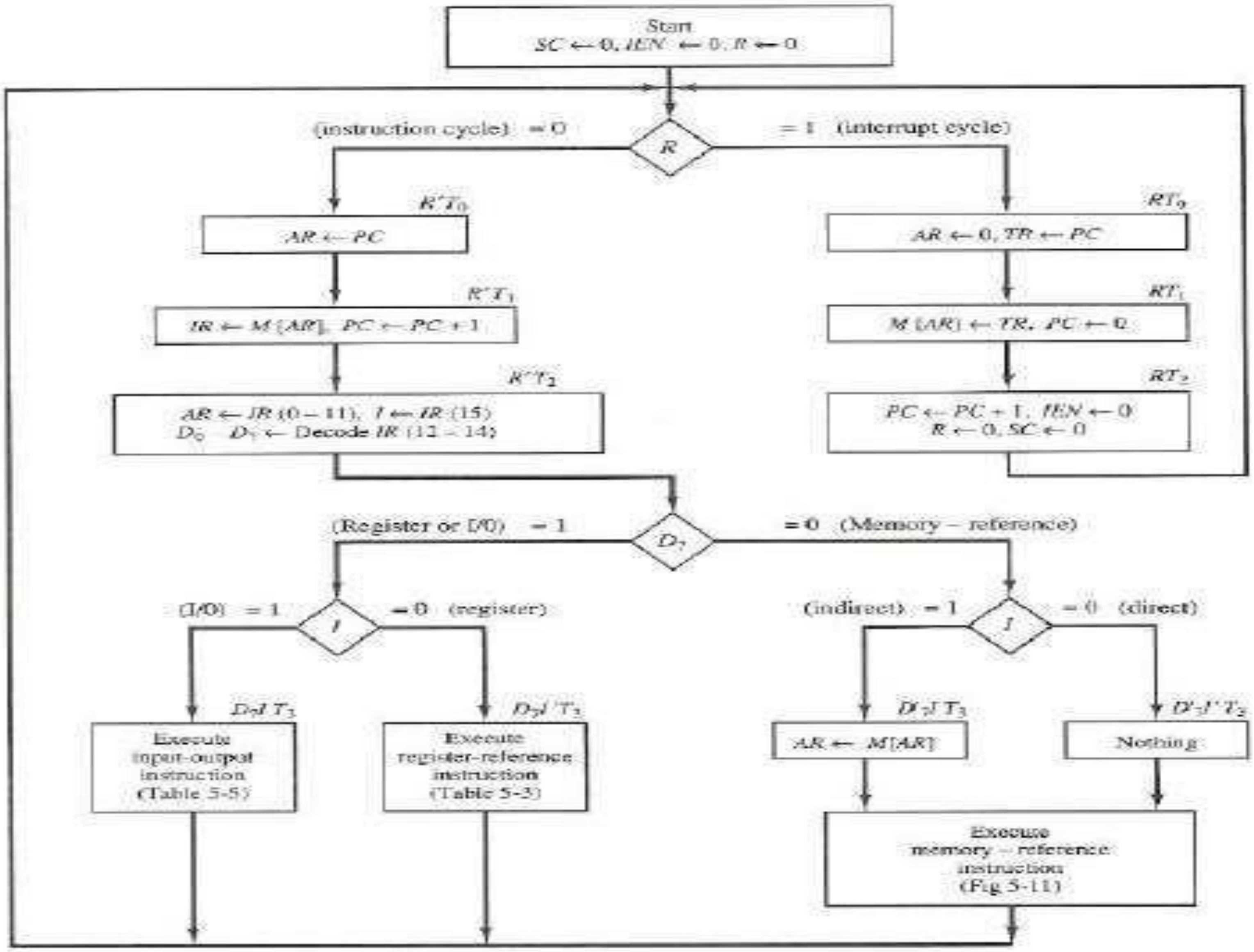
The interrupt cycle is initiated after the last execute phase if the interrupt flip-flop R is equal to 1.

This flip- flop is set to 1 if IEN = 1 and either FGI or FGO are equal
to 1.

This can happen with any clock transition except when timing signals T0, T1, or T2 are active. The condition for setting flip- flop R to 1 can be expressed with the following register transfer statement:

$T_0T_1T_2$ (IEN) (FGI + FGO): $R \rightarrow 1$

- Register Transfer Statements for Interrupt Cycle
- The fetch and decode phases of the instruction cycle must be modified:
- Replace Tб, Tб, T† with R'Tб, R’Tб, R’T†
- The interrupt cycle:
- RT0:*AR←б, TR←PC*
- *RTб: M [AR] ←TR, PC←б*
- *RT†:PC←PC+б, IEN←б, R←б, SC← б*



Start
$SC \leftarrow 0, IEN \leftarrow 0, R \leftarrow 0$
R
(instruction cycle) = 0
= 1 (interrupt cycle)
$R'T_0$
$AR \leftarrow PC$
$R'T_1$
$IR \leftarrow M[AR], PC \leftarrow PC + 1$
$R'T_2$
$AR \leftarrow IR(0-11), I \leftarrow IR(15)$
$D_0 \ldots D_7 \leftarrow$ Decode $IR(12-14)$
RT_0
$AR \leftarrow 0, TR \leftarrow PC$
RT_1
$M[AR] \leftarrow TR, PC \leftarrow 0$
RT_2
$PC \leftarrow PC + 1, IEN \leftarrow 0$
$R \leftarrow 0, SC \leftarrow 0$
D_7
(Register or I/O) = 1
= 0 (Memory – reference)
I
(I/O) = 1
= 0 (register)
I
(indirect) = 1
= 0 (direct)
D_7IT_3
Execute input-output instruction (Table 5-5)
$D_7I'T_3$
Execute register-reference instruction (Table 5-3)
D'_7IT_3
$AR \leftarrow M[AR]$
$D'_7I'T_3$
Nothing
Execute memory – reference instruction (Fig 5-11)

Addressing Modes

- Addressing mode is the way of addressing a memory location in instruction.

- Microcontroller needs data or operands on which the operation is to be performed.

- The method of specifying source of operand and output of result in an instruction is known as addressing mode.

- **Types of Addressing Modes:**

- Following are the types of Addressing Modes:

- Register Addressing Mode
- Direct Addressing Mode
- Register Indirect Addressing Mode
- Immediate Addressing Mode
- Index Addressing Mode

- **Register Addressing Mode:**
- In this addressing mode, the source of data or destination of result is Register. In this type of addressing mode the name of the register is given in the instruction where the data to be read or result is to be stored.

- Example: ADD A, R5   (The instruction will do the addition

of data in Accumulator with data in register R5)

- MOV DX, TAX_RATE ; Register in first operand
- MOV COUNT, CX ; Register in second operand
- MOV EAX, EBX ; Both the operands are in registers

Direct Addressing Mode:

In this type of Addressing Mode, the address of data to be read is directly given in the instruction. In case, for storing result the address given in instruction is used to store the result.

Example: MOV A, 46H (This instruction will move the contents of memory location 46H to Accumulator)

- For example,
- BYTE_VALUE DB 150 ; A byte value is defined
- WORD_VALUE DW 300 ; A word value is defined
- ADD BYTE_VALUE, 65 ; An immediate operand 65 is added

- **Mode Register Indirect Addressing:**
- In Register Indirect Addressing Mode, as its name suggests the data is read or stored in register indirectly. That is, we provide the register in the instruction, in which the address of the other register is stored or which points to other register where data is stored or to be stored.

- Example: MOV A, @R0 (This instruction will move the data to accumulator from the register whose address is stored in register R0).

- **Immediate Addressing Mode:** In Immediate Addressing Mode, the data immediately follows the instruction. This means that data to be used is already given in the instruction itself.

- Example: MOV A, #25H (This instruction will move the data 25H to Accumulator. The #sign show that preceding term is data, not the address.)

- **Index Addressing Mode:** Offset is added to the base index register to form the effective address if the memory location.
- This Addressing Mode is used for reading lookup tables in Program Memory. The Address of the exact location of the table is formed by adding the Accumulator Data to the base pointer.

- Example: MOVC, @A+DPTR (This instruction will move the data from the memory to Accumulator; the address is made by adding the contents of Accumulator and Data Pointer.

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** cause transfer of data from one location to another without changing the binary information content.

- **Data manipulation instructions** are those that perform arithmetic, logic, and shift operations.

- **Program control instructions** provide decision-making capabilities and change the path taken by the program when executed in the computer

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

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

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

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

Data Transfer And Manipulation

- **Logical and Bit Manipulation Instructions**
- Logical instructions perform binary operations on strings of bits stored in registers.

- They are useful for manipulating individual bits or a group of bits that represent binary-coded information.

- The logical instructions consider each bit of the operand separately and treat it as a Boolean variable.

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

- The **clear instruction** causes the specified operand to be replaced by D's. The complement instruction produces the 1's complement by inverting all the bits of the operand.

- The **AND, OR, and XOR** instructions produce the corresponding logical operations on individual bits of the operands.

- Although they perform **Boolean operations**, when used in computer instructions, the logical instructions should be considered as performing bit manipulation operations.

- **Shift Instructions**
- Instructions to shift the content of an operand are quite useful and are often provided in several variations. 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 either logical shifts, arithmetic shifts, or rotate-type operations. In either case the shift may be to the right or to the left.

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

- The rotate instructions produce a circular shift. Bits shifted out at one end of the word are not lost as in a logical shift but are circulated back into the other end.

- The rotate through carry instruction treats a carry bit as an extension of the register whose word is being rotated.

- Thus a rotate-left through carry instruction transfers the carry bit into the rightmost bit position of the register, transfers the leftmost bit position into the carry and at the same time shifts the entire register to the left

Program Control

•Instructions are always stored in successive memory locations. When processed in the CPU, the instructions are fetched from consecutive memory locations and executed.

•Each time an instruction is fetched from memory, the program counter is incremented so that it contains the address of the next instruction in sequence.

•After the execution of a data transfer or data manipulation instruction, control returns to the fetch cycle with the program counter containing the address of the instruction next in sequence.

- Some typical program control instructions are listed in Table 3.8.

  •The branch and jump instructions are used interchangeably to mean the same thing, but sometimes they are used to denote different addressing modes.

- The branch is usually a one-address instruction. It is written in assembly language as BR ADR, where ADR is a symbolic name for an address.

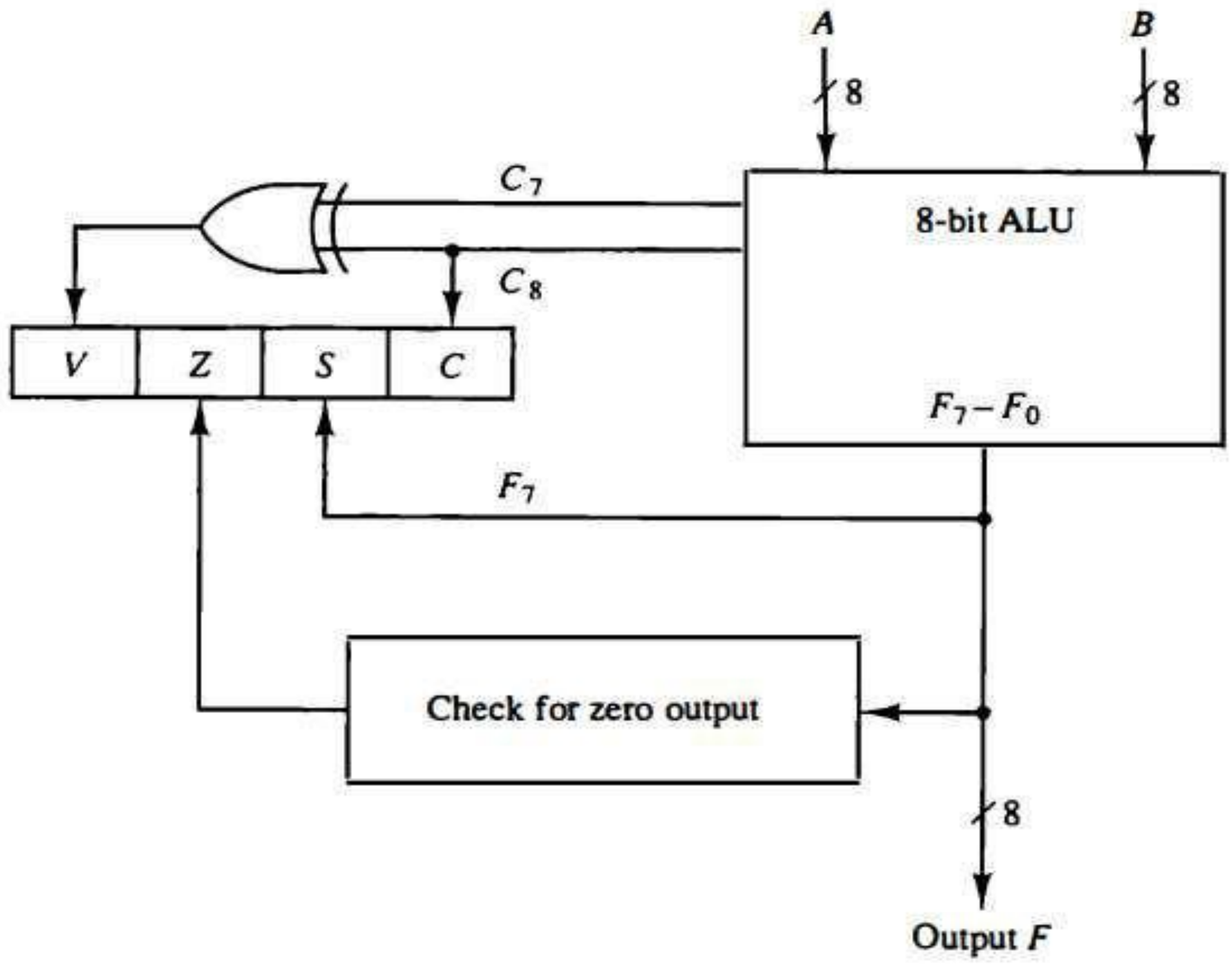
- When executed, the branch instruction causes a transfer of the value of ADR into the program counter.

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

•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.

•If the condition is not met, control proceeds with the next instruction in sequence where the programmer inserts an unconditional branch instruction.

•Thus a skip-branch pair of instructions causes a branch if the condition is not met, while a single conditional branch instruction causes a branch if the condition is met.


A
8
B
8
8-bit ALU
C_7
C_8
V
Z
S
C
$F_7 - F_0$
F_7
Check for zero output
8
Output F

- Bit C (carry) is set to 1 if the end carry C8 is 1. It is cleared to 0

if the carry is 0.

- Bit S (sign) is set to 1 if the highest-order bit F, is 1. It is set to 0

if the bit is 0.

- Bit Z (zero) is set to 1 ifthe output ofthe ALU contains all O's. !t is cleared to 0 otherwise. In other words, Z = 1 if the output is zero and Z = 0 if the output is not zero.

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

Program Control

- **Subroutine** Call and Return
- A subroutine is a self-contained sequence of instructions that performs a given computational task.

- During the execution of a program, a subroutine may be called to perform its function many times at various points in the main program.

- Each time a subroutine is called, a branch is executed to the beginning of the subroutine to start executing its set of instructions. After the subroutine has been executed, a branch is made back to the main program.

- A subroutine call is implemented with the following microoperations:

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

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

- **PC <- M[SP]** **Pop stack and transfer to PC**
- **SP <- SP + 1** **Increment stack pointer**

- These three procedural concepts are clarified further below.

- 1. The content of the program counter
- 2. The content of all processor registers
- 3. The content of certain status conditions

- **Types of Interrupts**

There are three major types of interrupts that cause a break in the normal execution of a program. They can be classified as:

- 1. External interrupts
- 2. Internal interrupts
- 3. Software interrupts

- Internal interrupts are synchronous with the program while

external interrupts are asynchronous.

- If the program is rerun, the internal interrupts will occur in the

same place each time.

- External interrupts depend on external conditions that are

independent of the program being executed at the time.

- External and internal interrupts are initiated from signals that occur in the hardware of the CPU.

- A software interrupt is initiated by executing an instruction.
- Software interrupt is a special call instruction that behaves like an interrupt rather than a subroutine call.
- It can be used by the programmer to initiate an interrupt procedure at any desired point in the program.

Program Control

- **Subroutine** Call and Return
- A subroutine is a self-contained sequence of instructions that performs a given computational task.

- During the execution of a program, a subroutine may be called to perform its function many times at various points in the main program.

- Each time a subroutine is called, a branch is executed to the beginning of the subroutine to start executing its set of instructions. After the subroutine has been executed, a branch is made back to the main program.

Addition & Subtraction

There are three ways of representing negative fixed-point binary numbers: signed-magnitude, signed-l's complement, or signed-2's complement.

- Most computers use the signed-2's complement representation when performing arithmetic operations with integers. For floating-point operations, most computers use the signed-magnitude representation for the mantissa.

- In this section we develop the addition and subtraction algorithms for data represented in signed-magnitude and again for data represented in signed-2's complement

- **Addition and Subtraction with Signed-Magnitude Data**

- The representation of numbers in signed-magnitude is familiar because it is used in everyday arithmetic calculations.

- The procedure for adding or subtracting two signed binary numbers with paper and pencil is simple and straightforward.

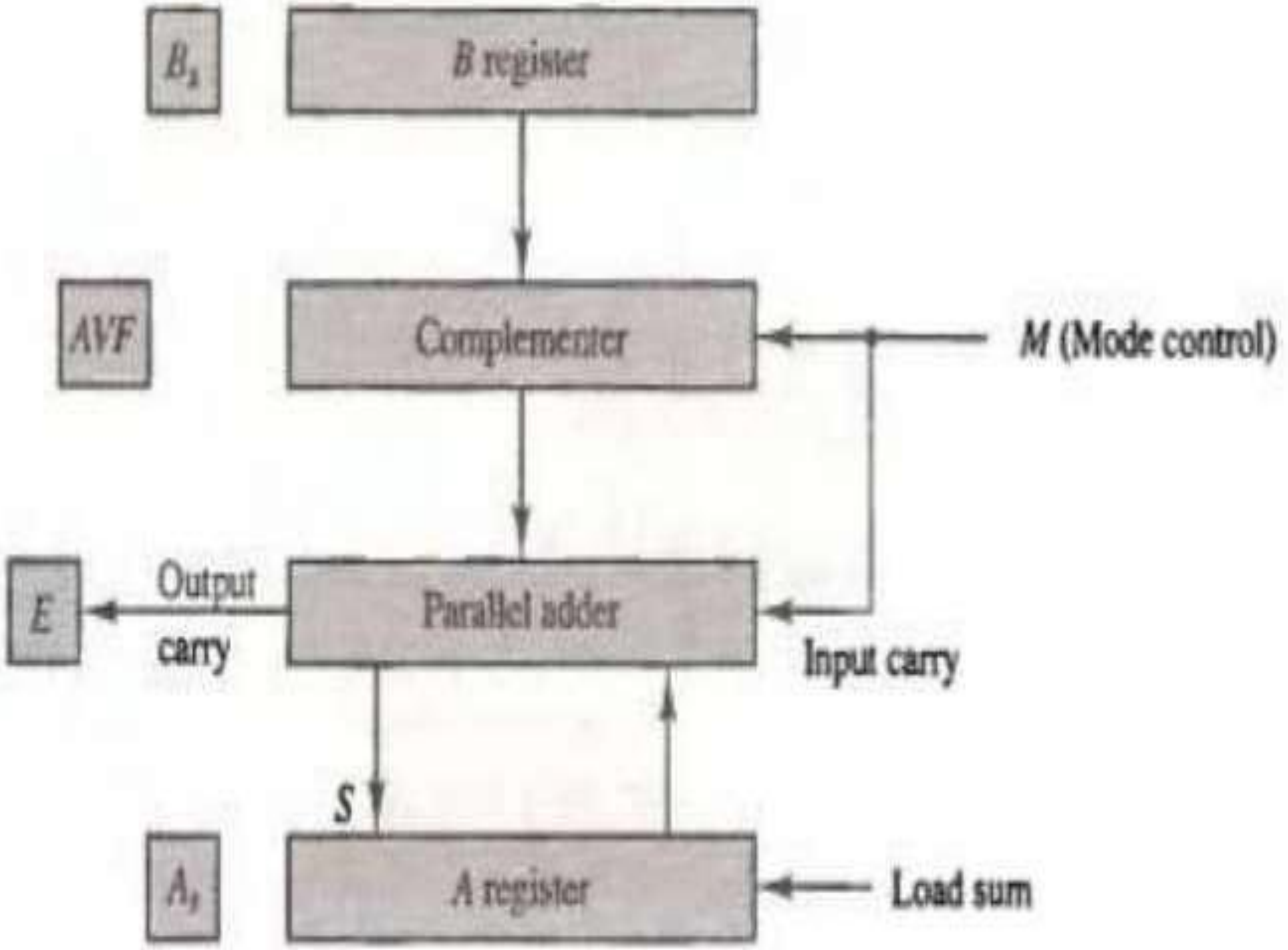
- A review of this procedure will be helpful for deriving the hardware algorithm.

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

- Addition (subtraction) algorithm: when the signs of A and B are identical (different), add the two magnitudes and attach the sign of A to the result.
- When the signs of A and B are different (identical), compare the magnitudes and subtract the smaller number from the larger.
- Choose the sign of the result to be the same as A if A > B or the complement of the sign of A if A < B. If the two magnitudes are equal, subtract B from A and make the sign of the result positive.

Addition & Subtraction

- **Hardware Implementation**
- To implement the two arithmetic operations with hardware, it is first necessary that the two numbers be stored in registers.
- Let A and B be two registers that hold the magnitudes of the numbers, and A, and B, be two flip-flops that hold the corresponding signs.
- The result of the operation may be transferred to a third register: however, a saving is achieved if the result is transferred into A and A. Thus A and A, together form an accumulator registers.


B_s
B register
AVF
Complementer
M (Mode control)
E
Output carry
Parallel adder
Input carry
S
A_s
A register
Load sum

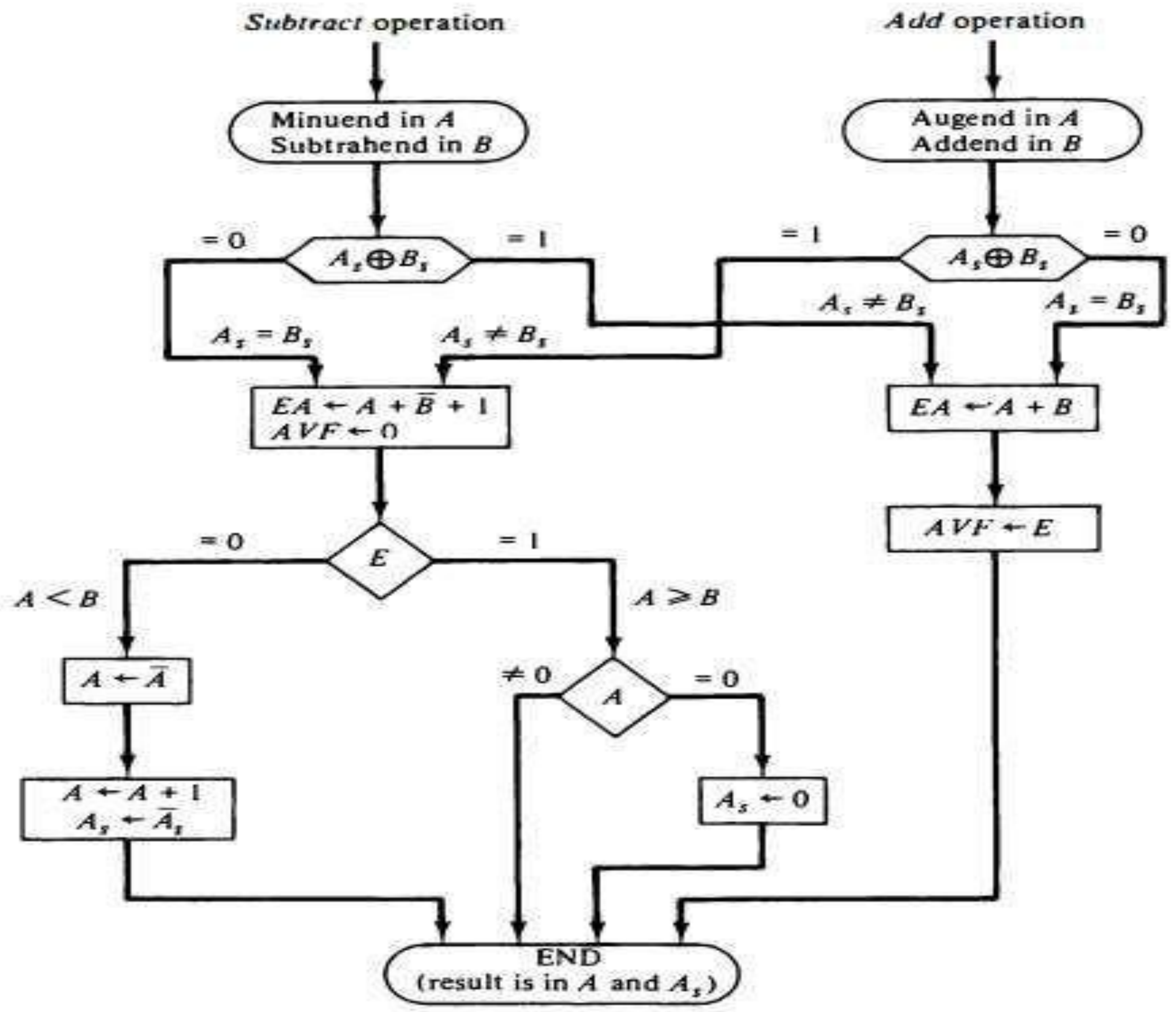
- The addition of A plus B is done through the parallel adder. The S (sum) output of the adder is applied to the input of the A register.

- The complementer provides an output of B or the complement of B depending on the state of the mode control M.

- **Hardware Algorithm**
- The flowchart for the hardware algorithm is presented in Figure 3.9. The two signs A, and B, are compared by an exclusive-OR gate.

- If the output of the gate is O, the signs are identical; if it is I, 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.



Subtract operation
Add operation
Minuend in A
Subtrahend in B
Augend in A
Addend in B
= 0
$A_s \oplus B_s$
= 1
= 1
$A_s \oplus B_s$
= 0
$A_s = B_s$
$A_s \neq B_s$
$A_s \neq B_s$
$A_s = B_s$
$EA \leftarrow A + \overline{B} + 1$
$AVF \leftarrow 0$
$EA \leftarrow A + B$
$AVF \leftarrow E$
= 0
E
= 1
$A < B$
$A \geq B$
$A \leftarrow \overline{A}$
$\neq 0$
A
= 0
$A \leftarrow A + 1$
$A_s \leftarrow \overline{A}_s$
$A_s \leftarrow 0$
END
(result is in A and A_s)

Floating Point Arithmetic

•In computing, floating-point arithmetic is arithmetic using formulaic representation of real numbers as an approximation so as to support a trade-off between range and precision.

•For this reason, floating-point computation is often found in systems which include very small and very large real numbers, which require fast processing times.

•A number is, in general, represented approximately to a fixed number of significant digits (the significant) and scaled using an exponent in some fixed base; the base for the scaling is normally two, ten, or sixteen.

- A number that can be represented exactly is of the following form:

$$\text{significand} \times \text{base}^{\text{exponent}},$$

where significand is an integer (i.e., in **Z**), base is an integer greater than or equal to two, and exponent is also an integer. For example:

$$1.2345 = \underbrace{12345}_{\text{significand}} \times \underbrace{10}_{\text{base}}{}^{\overbrace{-4}^{\text{exponent}}}.$$

- **Floating-point arithmetic operations**
- For ease of presentation and understanding, decimal radix with 7 digit precision will be used in the examples, as in the IEEE 754 decimal32 format.
- The fundamental principles are the same in any radix or precision, except that normalization is optional (it does not affect the numerical value of the result). Here, s denotes the significant and e denotes the exponent

Addition and subtraction

- A simple method to add floating-point numbers is to first represent them with the same exponent. In the example below, the second number is shifted right by three digits, and one then proceeds with the usual addition method:
  - $123456.7 = 1.234567 \times 10^5$
  - $101.7654 = 1.017654 \times 10^2 = 0.001017654 \times 10^5$
  - Hence:

$$
\begin{aligned}
123456.7 + 101.7654 &= (1.234567 \times 10^5) + (1.017654 \times 10^2) \\
&= (1.234567 \times 10^5) + (0.001017654 \times 10^5) \\
&= (1.234567 + 0.001017654) \times 10^5 \\
&= 1.235584654 \times 10^5
\end{aligned}
$$

- In detail:

- e=5; s=1.234567 (123456.7)
- + e=2; s=1.017654 (101.7654)

- e=5; s=1.234567
- + e=5; s=0.001017654 (after shifting)
- --------------------
- e=5; s=1.235584654 (true sum: 123558.4654)

- This is the true result, the exact sum of the operands. It will be rounded to seven digits and then normalized if necessary. The final result is
- e=5; s=1.235585 (final sum: 123558.5)

- Note that the lowest three digits of the second operand (654) are essentially lost.

- This is round-off error. In extreme cases, the sum of two non-zero numbers may be equal to one of them:

- e=5; s=1.234567
- + e=−ʈ; s=ϵ.ϴjcϱϰʈ

- e=5; s=1.234567
- + e=5; s=0.00000009876543 (after shifting)
- ----------------------
- e=5; s=1.23456709876543 (true sum)

- e=5; s=1.234567

Floating Point Arithmetic

- **Multiplication and division**
- To multiply, the significant are multiplied while the exponents are added, and the result is rounded and normalized.
- e=3; s=4.734612
- × e=5; s=5.417242
- ------------------------
- e=8; s=25.648538980104 (true product)
- e=8; s=25.64854 (after rounding)
- e=9; s=2.564854 (after normalization)

- **MULTIPLICATION**
- Example on decimal values given in scientific notation:
- 3.0 x 10 ** 1
- + 0.5 x 10 ** 2
- -----------------
- Algorithm: multiply mantissas Add exponents
-
- 3.0 x 10 ** 1
- + 0.5 x 10 ** 2
- -----------------
- 1.50 x 10 ** 3

- 0 10000100 0100
- X 1 00111100 1100
- ----------------
- Mantissa multiplication: 1.0100
- ;Don’t forget hidden bit x 1.1100
- ------
- 00000
- 00000
- 10100
- 10100
- 10100
- ---------
- 1000110000
- Becomes 10.00110000

- Add exponents: always add true exponents
- (Otherwise the bias gets added in twice)

biased:

- 10000100
- + 00111100
- ----------
- 10000100 01111111 (switch the order of the subtraction,
- - 01111111 - 00111100 so that we can get a negative value)
- ---------- ----------
- 00000101 01000011

- True exp is 5. True exp is -67
- Add true exponents 5 + (-67) is -62.
- Re-bias exponent: -62 + 127 is 65.
- Unsigned representation for 65 is 01000001. Put the result back together (and add sign bit).
- 1 01000001 10.00110000
- Normalize the result: (moving the radix point one place to the left increases the exponent by 1.)
- 1 01000001 10.00110000 Becomes 1 01000010 1.000110000

Decimal Arithmetic Unit

- The user of a computer prepares data with decimal numbers and receives results in decimal form.

- A CPU with an arithmetic logic unit can perform arithmetic microoperations with binary data.

- To perform arithmetic operations with decimal data, it is necessary to convert the input decimal numbers to binary, to perform all calculations with binary numbers, and to convert the results into decimal.

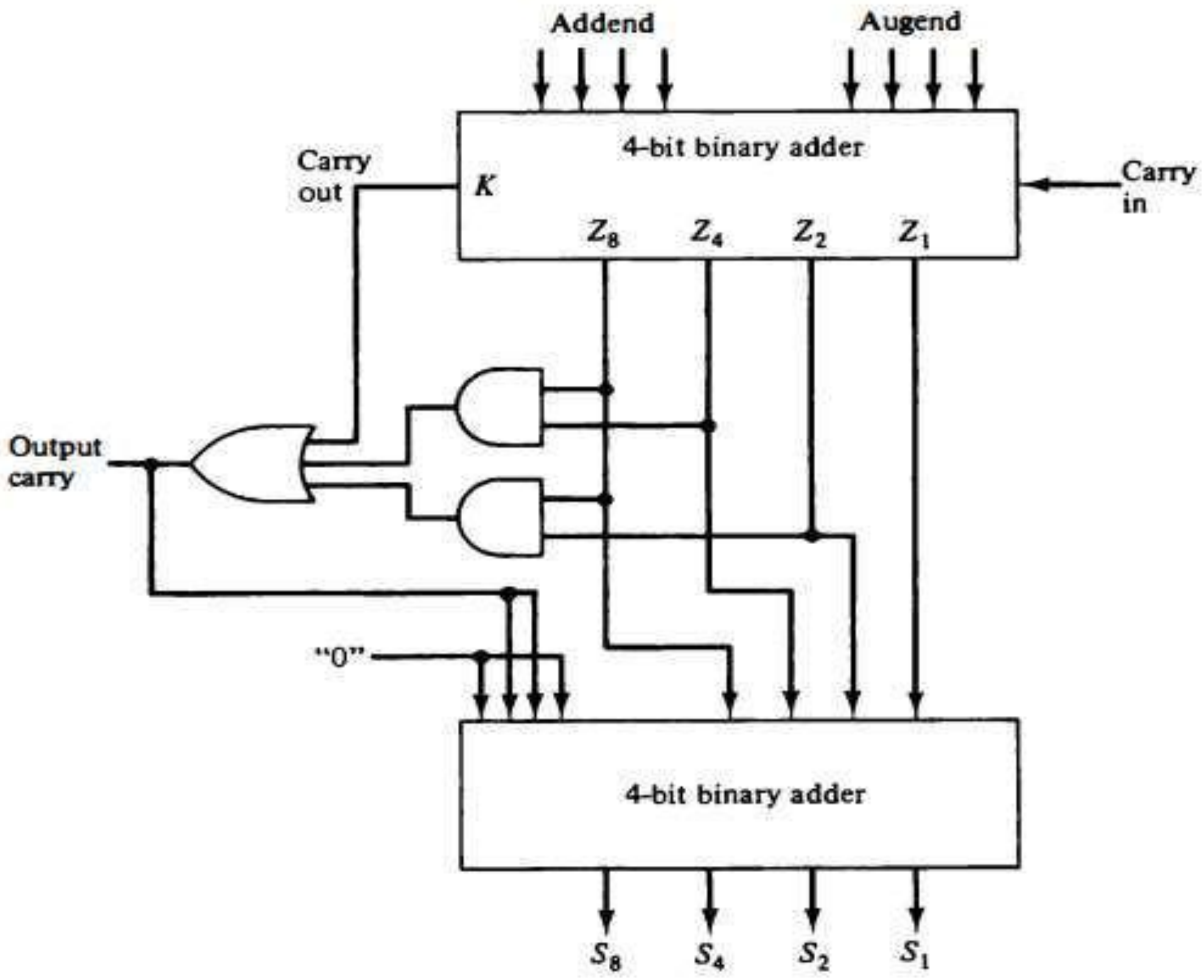
- **BCD Adder**

- Consider the arithmetic addition of two decimal digits in BCD, together with a possible carry from a previous stage.

- Since each input digit does not exceed 9, the output sum cannot be greater than 9 + 9 + 1 = 19, the 1 in the sum being an input-carry.

- Suppose that we apply two BCD digits to a 4-bit binary adder.

Binary Sum					BCD Sum					
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	Decimal
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19


Addend
Augend
4-bit binary adder
Carry out
K
Carry in
Z_8
Z_4
Z_2
Z_1
Output carry
"0"
4-bit binary adder
S_8
S_4
S_2
S_1

- A BCD adder is a circuit that adds two BCD digits in parallel and produces a sum digit also in BCD.

- A BCD adder must include the correction logic in its internal construction. To add 0110 to the binary sum, we use a second 4-bit binary adder as shown in Figure.

Memory Organization: Memory Hierarchy:

•The user of a computer prepares data with decimal numbers and receives results in decimal form.

•A CPU with an arithmetic logic unit can perform arithmetic microoperations with binary data.

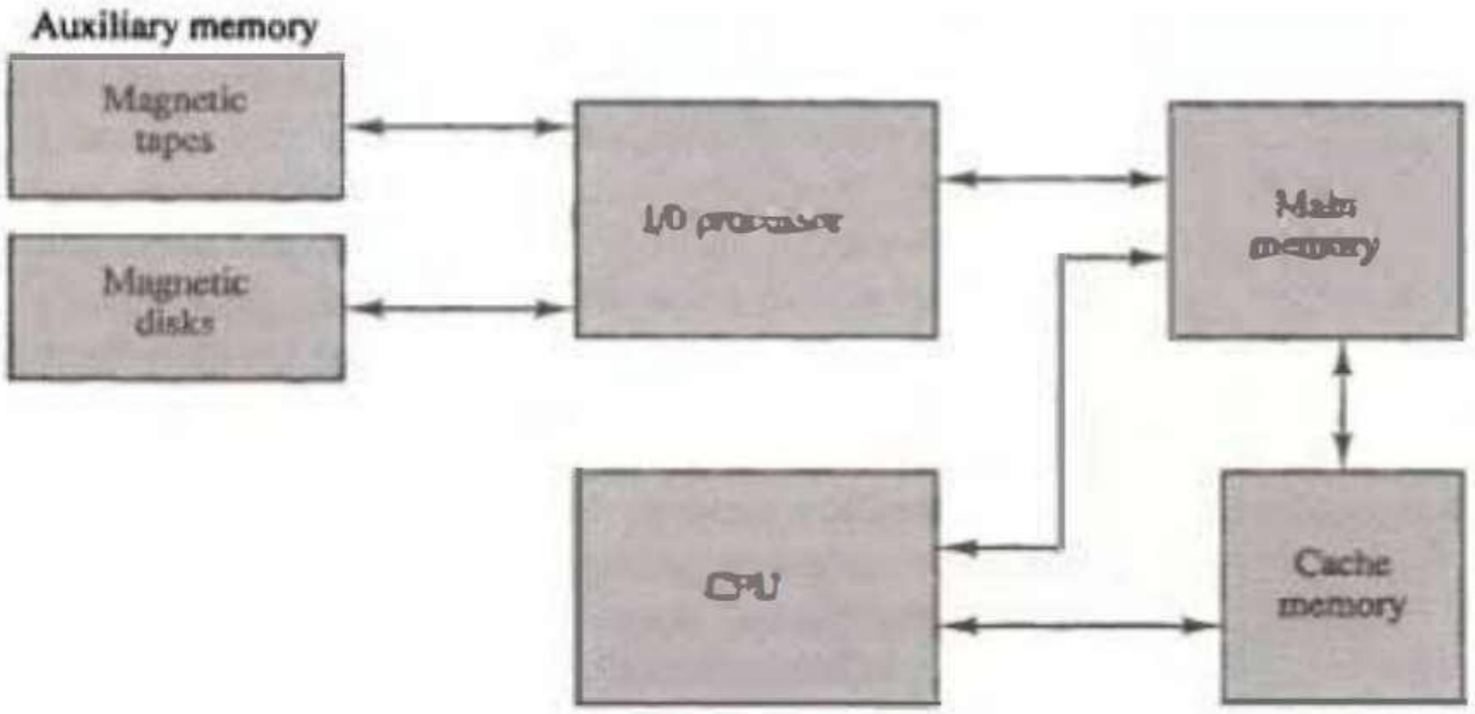
•To perform arithmetic operations with decimal data, it is necessary to convert the input decimal numbers to binary, to perform all calculations with binary numbers, and to convert the results into decimal.

- **MEMORY HIERARCHY**

- The memory unit is an essential component in any digital computer since it is needed for strong 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.

- The memory unit that communicates directly with the called 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. Another information is stored in auxiliary memory and transferred to main memory when needed.


Auxiliary memory
Magnetic tapes
Magnetic disks
I/O processor
Main memory
CPU
Cache memory

While the **I/O processor** manages data transfers between auxiliary memory and main memory, the cache organization is concerned with the transfer of information between main memory and CPU.

•Thus each is involved with a different level in the memory hierarchy system.

•The reason for having two or three levels of memory hierarchy is economics.

•As the storage capacity of the memory increases, the cost per bit for storing binary information decreases and the access time of the memory becomes longer.

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 the 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.

- Refreshing is done by cycling through the words every few milliseconds to restore the decaying charge.

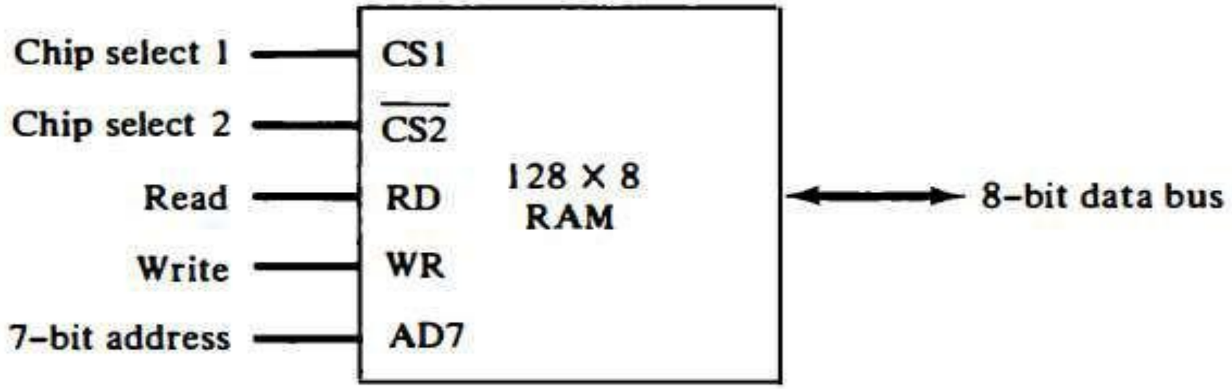
- The **dynamic RAM** offers reduced power consumption and larger storage capacity in a single memory chip.

- The **static RAM** is easier to use and has shorter read and write cycles.

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

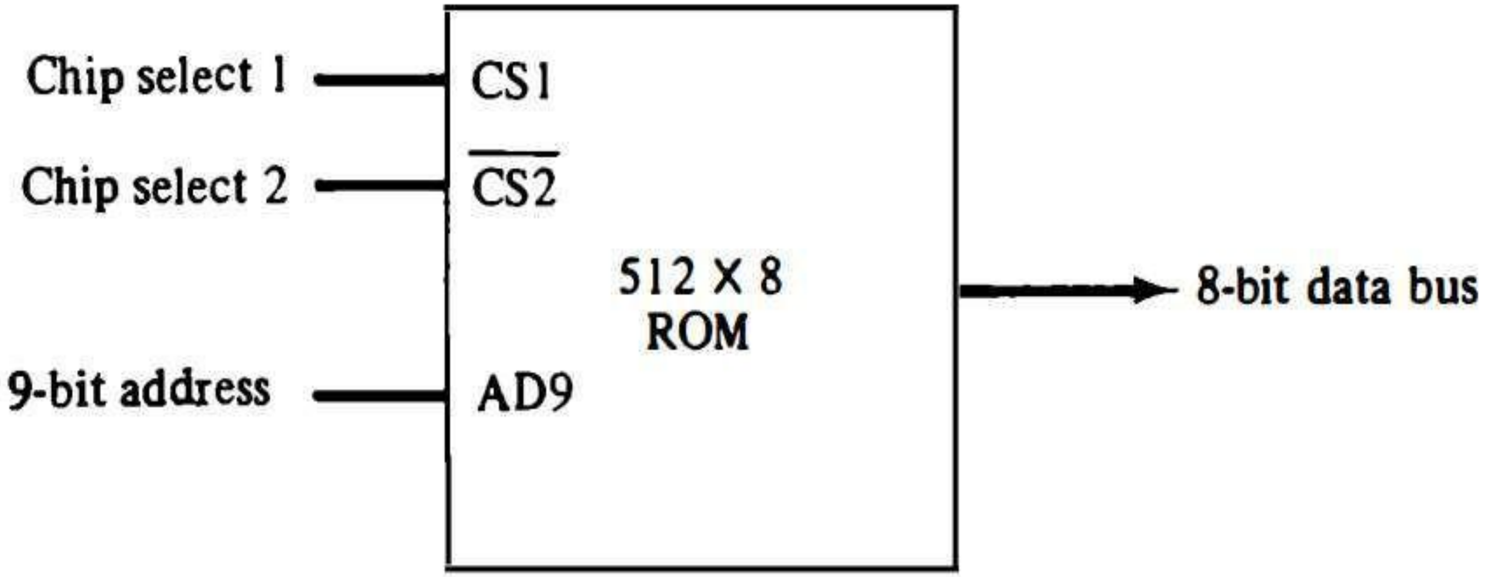


(a) Block diagram

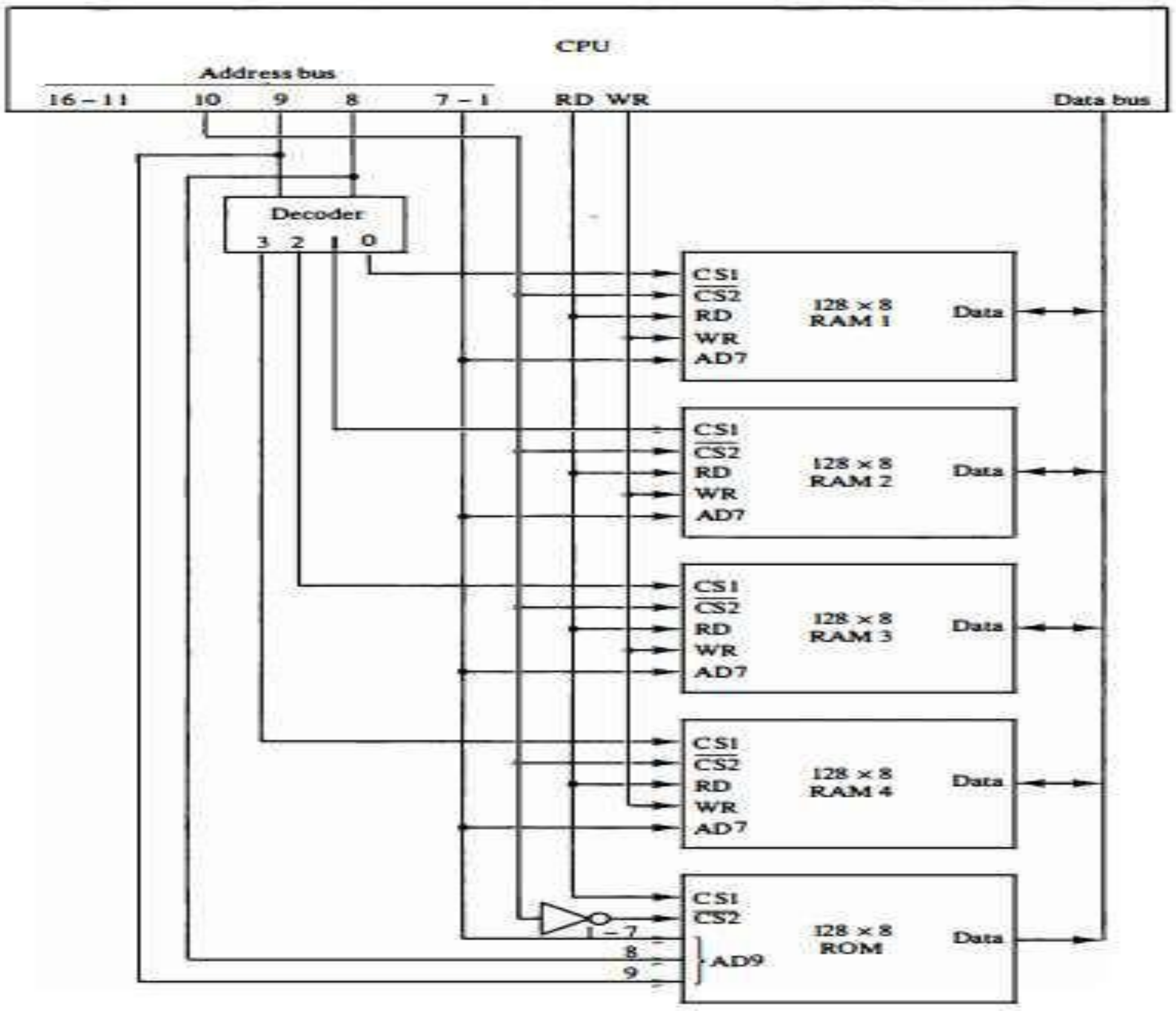
CS1	$\overline{CS2}$	RD	WR	Memory function	State of data bus
0	0	×	×	Inhibit	High-impedance
0	1	×	×	Inhibit	High-impedance
1	0	0	0	Inhibit	High-impedance
1	0	0	1	Write	Input data to RAM
1	0	1	×	Read	Output data from RAM
1	1	×	×	Inhibit	High-impedance

(b) Function table

Typical ROM chip

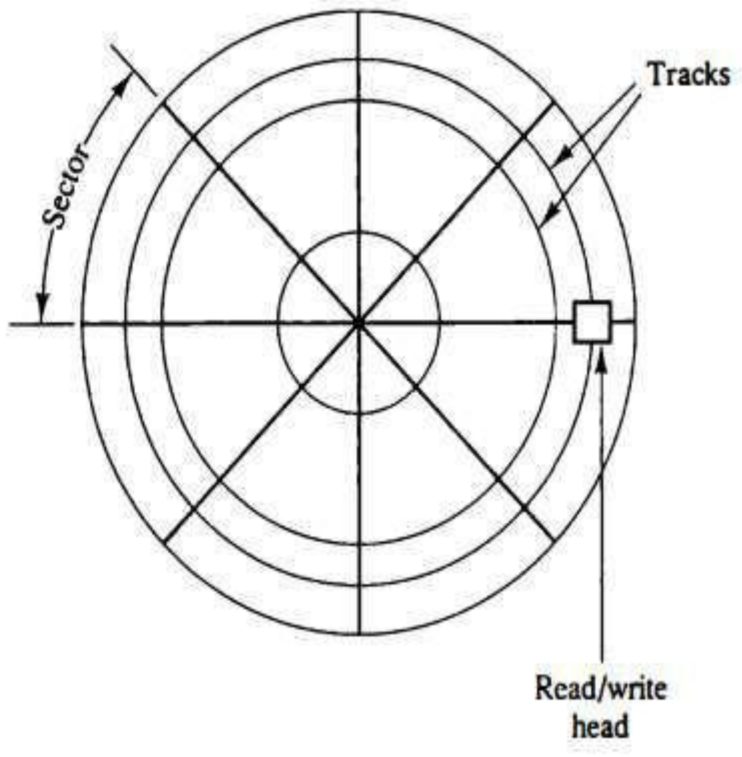


Memory connections of the CPU

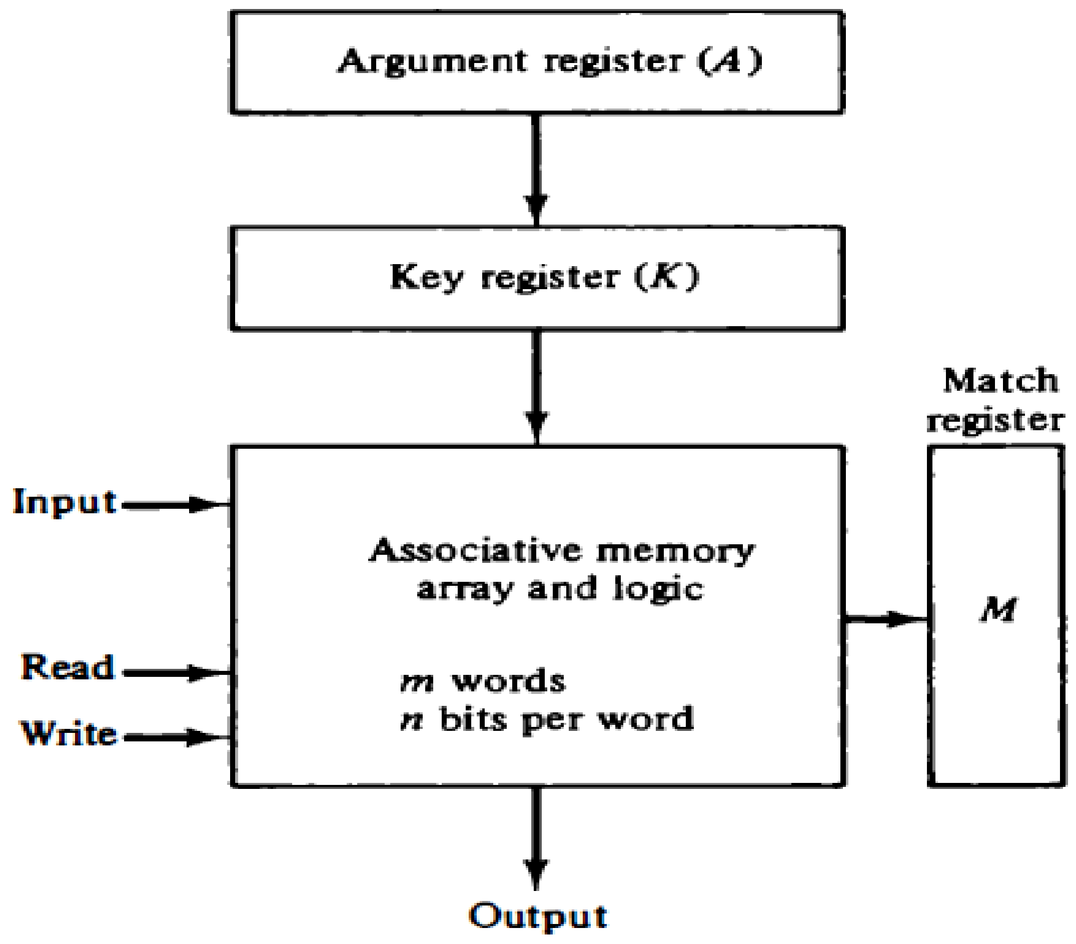


Auxiliary Memory

Magnetic Disks

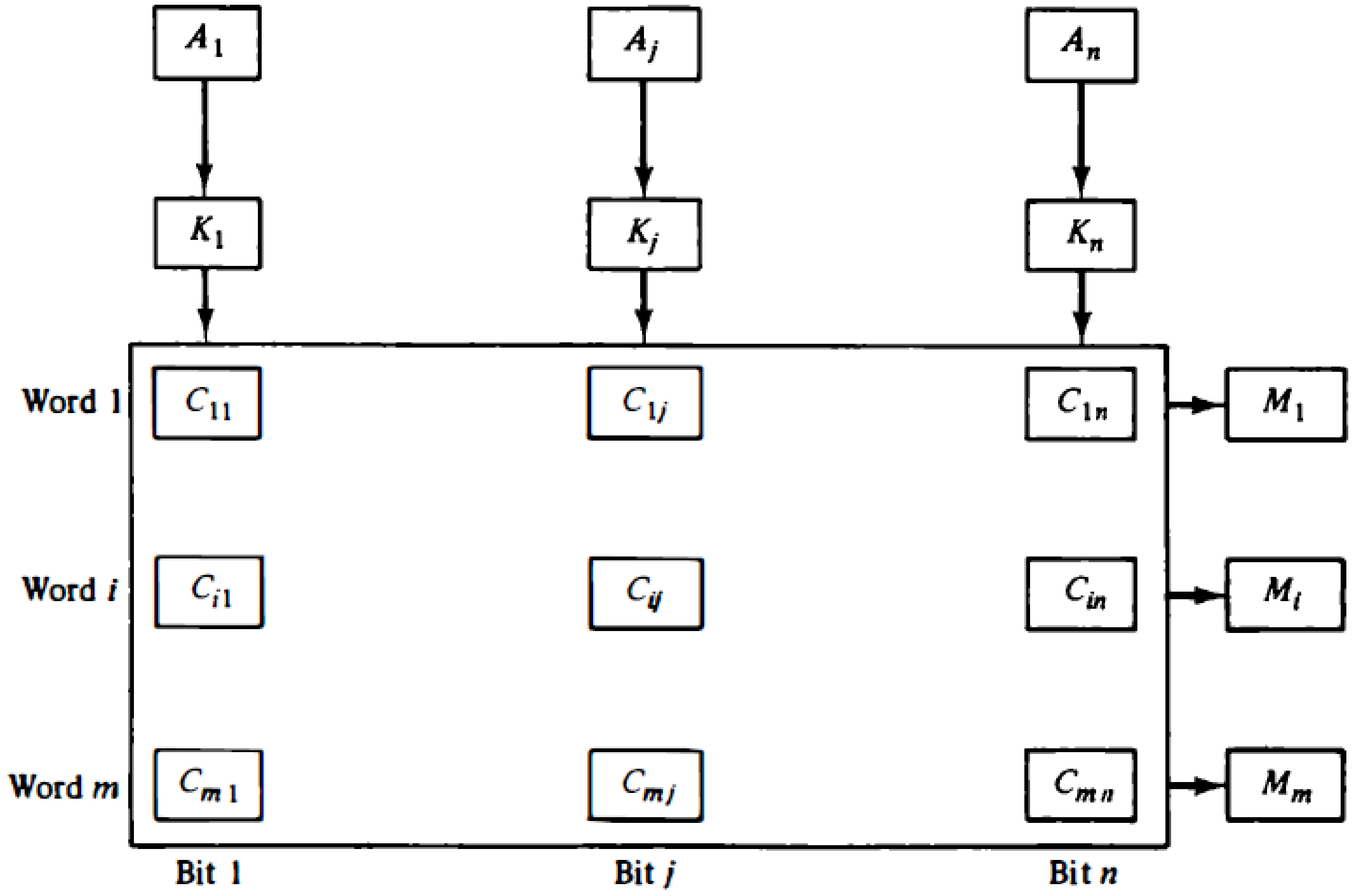


Associative Memory



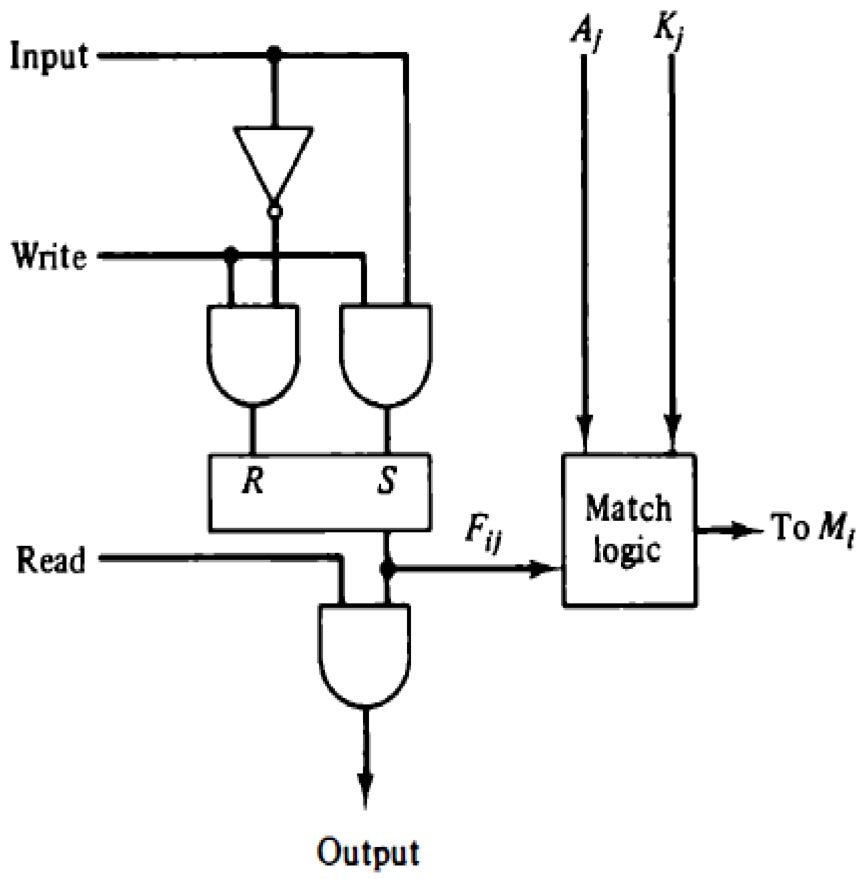
Example

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



A_1
A_j
A_n
K_1
K_j
K_n
Word 1
C_{11}
C_{1j}
C_{1n}
M_1
Word i
C_{i1}
C_{ij}
C_{in}
M_i
Word m
C_{m1}
C_{mj}
C_{mn}
M_m
Bit 1
Bit j
Bit n

One cell of associative memory



- The internal organization of a typical cell Cij is shown in
above figure.
- It consists of a flip-flop storage element Fij and the circuits for
reading, writing, and matching the cell.
- The input bit is transferred into the storage cell during a write
operation.
- The bit stored is read out during a read operation.
- The match logic compares the content of the storage cell with the corresponding unmasked bit of the argument and provides an output for the decision logic that sets the bit in Mi.

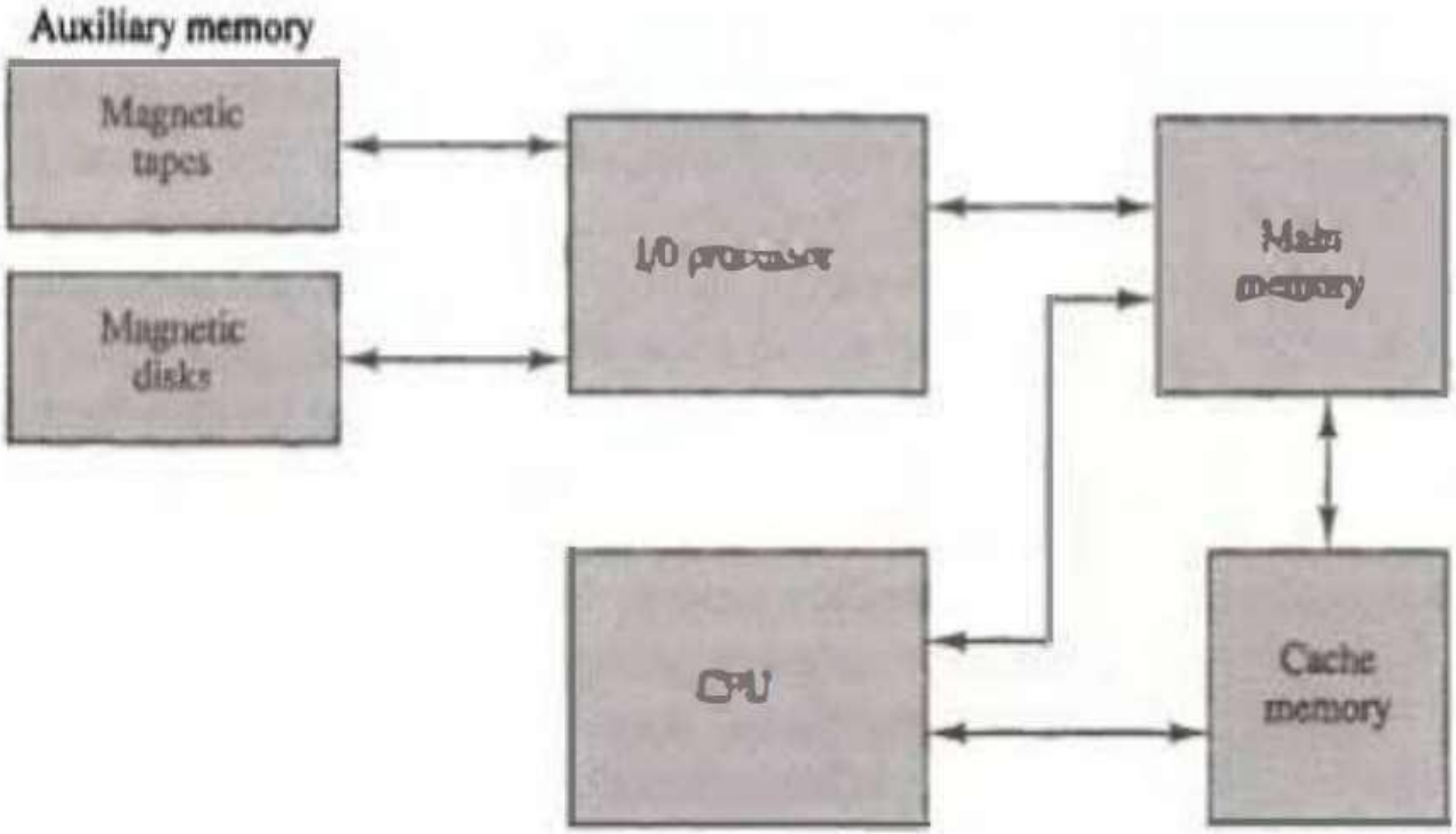
Cache Memory

- Analysis of a large number of typical programs has shown that the references to memory at any given interval of time tend to be confined within a few localized areas in memory.
- This phenomenon is known as the property of **localityof**

reference.

- 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**.

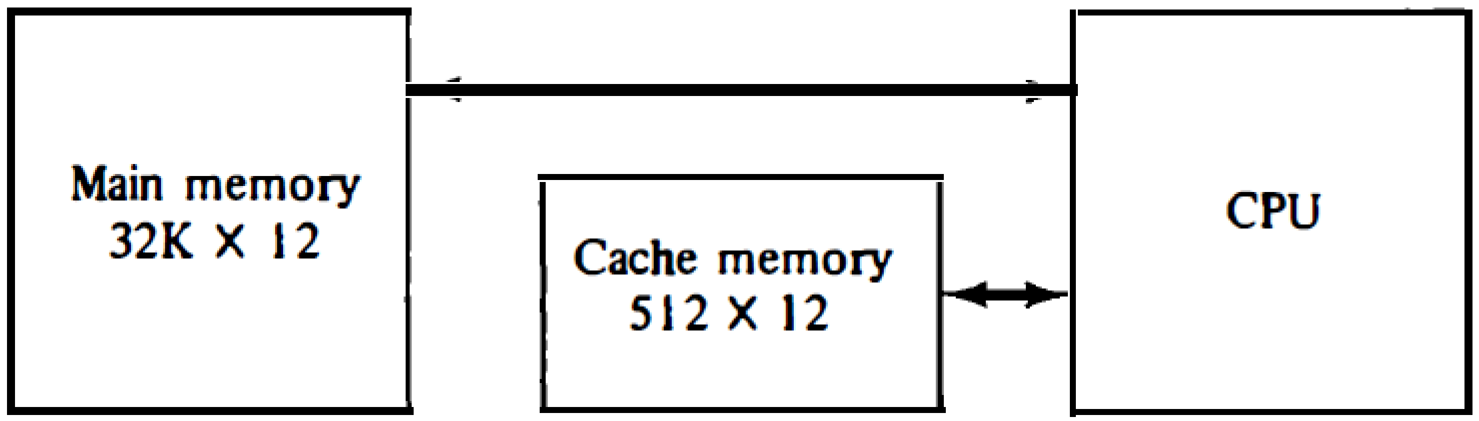
Memory Hierarchy in a computer system



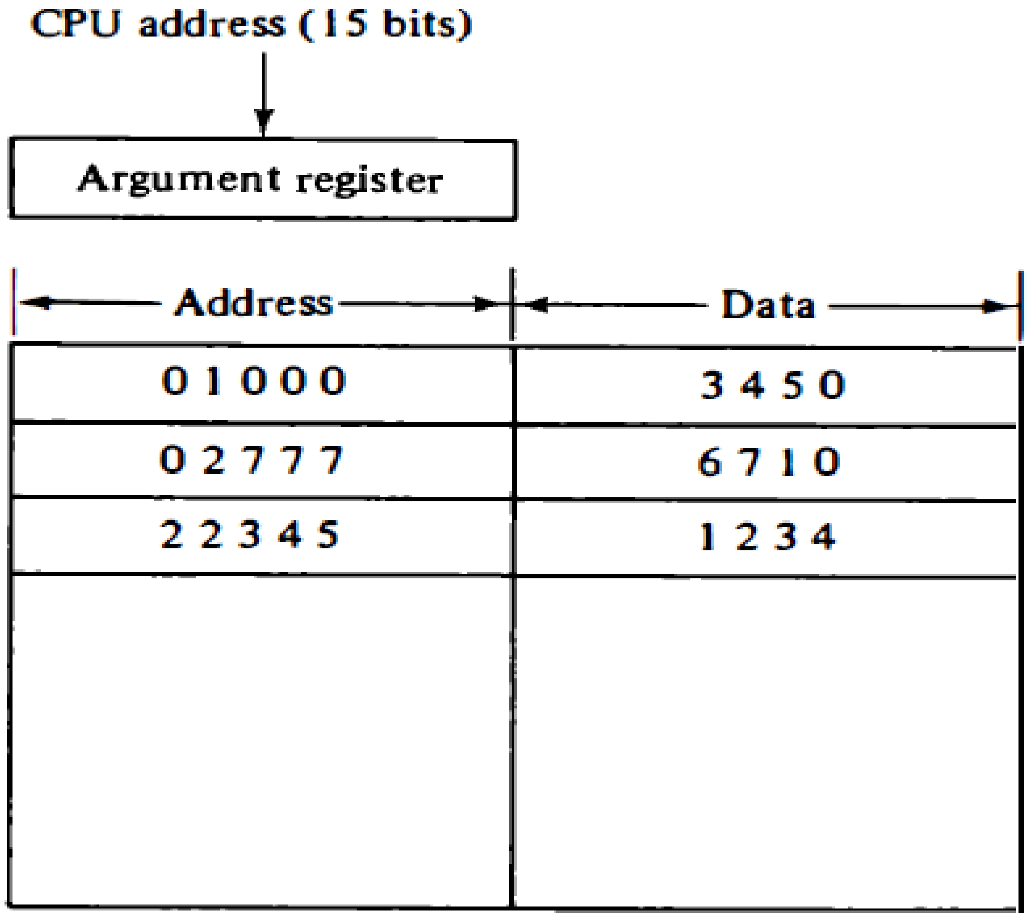
- Three types of mapping procedures are of practical interest when considering the organization of cache memory:

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

Example of Cache memory

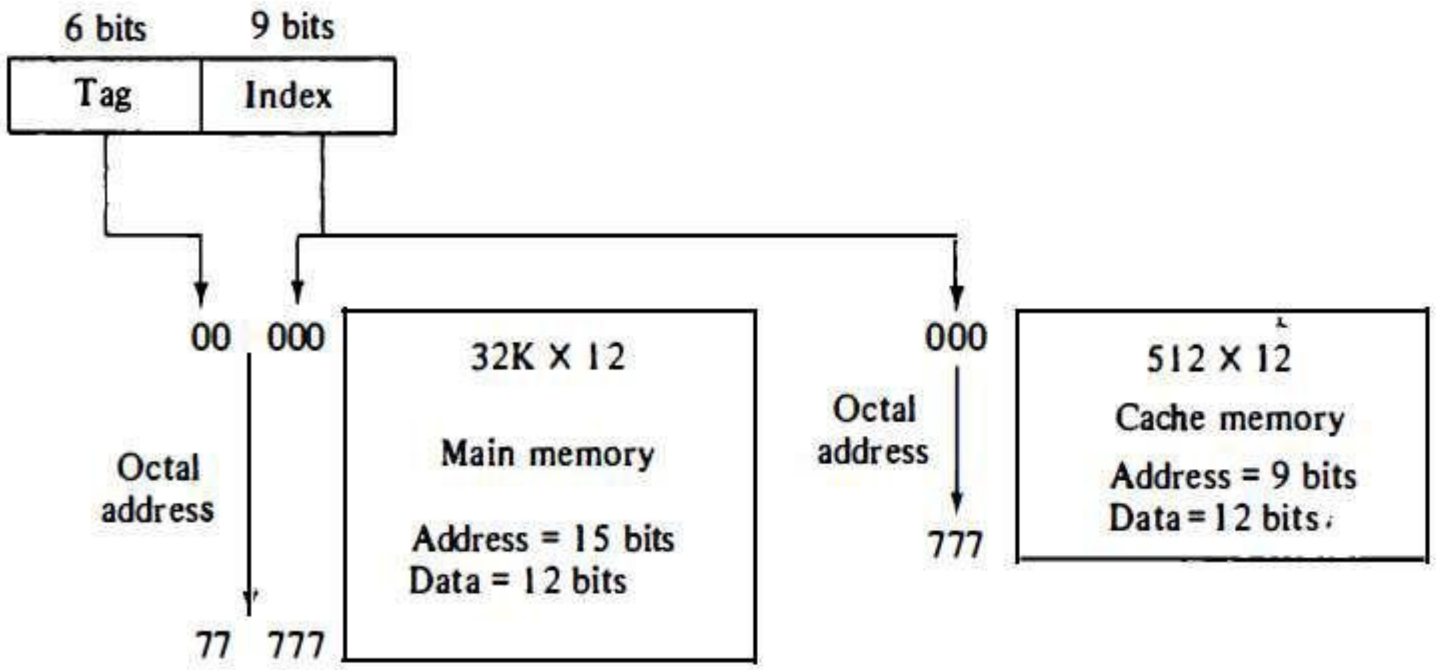


Associative Mapping

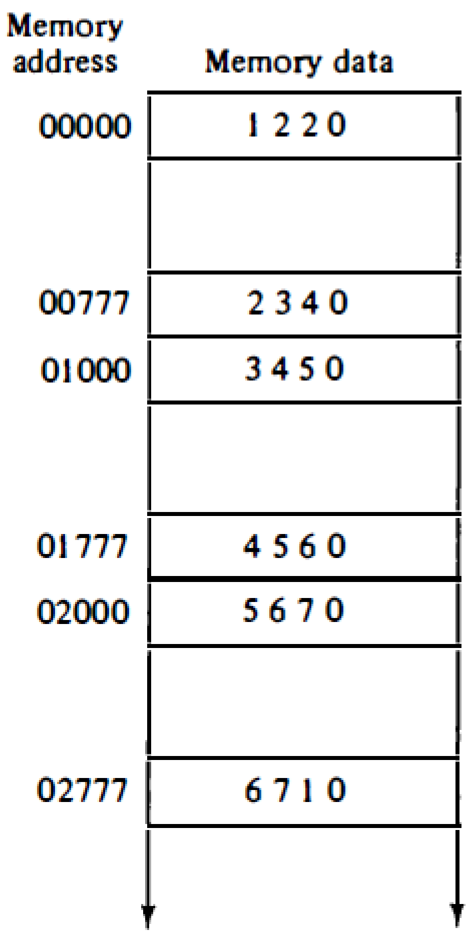


Address	Data
01000	3450
02777	6710
22345	1234

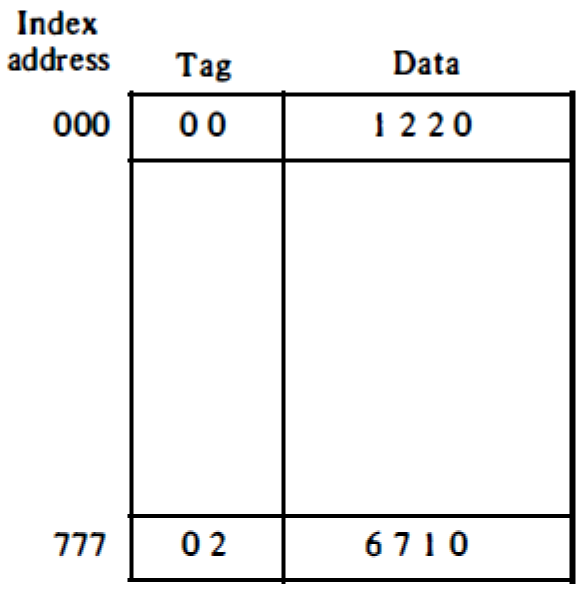
Addressing relationships



Direct Mapping Cache Organization



(a) Main memory



(b) Cache memory

Cache Memory

- **Set-Associative Mapping**
- It was mentioned previously that the disadvantage of direct mapping is that two words with the same index in their address but with different tag values cannot reside in cache memory at the same time.

- A third type of cache organization, called set-associative mapping, is an improvement over the direct mapping organization in that each word of cache can store two or more words of memory under the same index address.

Two way Set-Associative Mapping Cache

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

- 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 x 36.

- It can accommodate 1024 words of main memory since each word of cache contains two data words. In general, a set-associative cache of set size k will accommodate k words of main memory in each word of cache.

- When a miss occurs in a set-associative cache and the set is full, it is necessary to replace one of the tag-data items with a new value.
- The most common **replacement algorithms** used are random replacement, first-in, first out (FIFO), and least recently used (LRU).
- With the random replacement policy the control
  chooses one
- tag-data item for replacement at random.

- **Writing into Cache**

- An important aspect of cache organization is concerned with memory write requests.

- The simplest and most **commonly used procedure** is to update 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.**

- **Cache Initialization**
- The cache is initialized by clearing all the valid bits to 0. The valid bit of a particular cache word is set to 1 the first time this word is loaded from main memory and stays set unless the cache has to be initialized again.

- The introduction of the valid bit means that a word in cache is not replaced by another word unless the valid bit is set to 1 and a mismatch of tags occurs. If the valid bit happens to be 0, the new word automatically replaces the invalid data

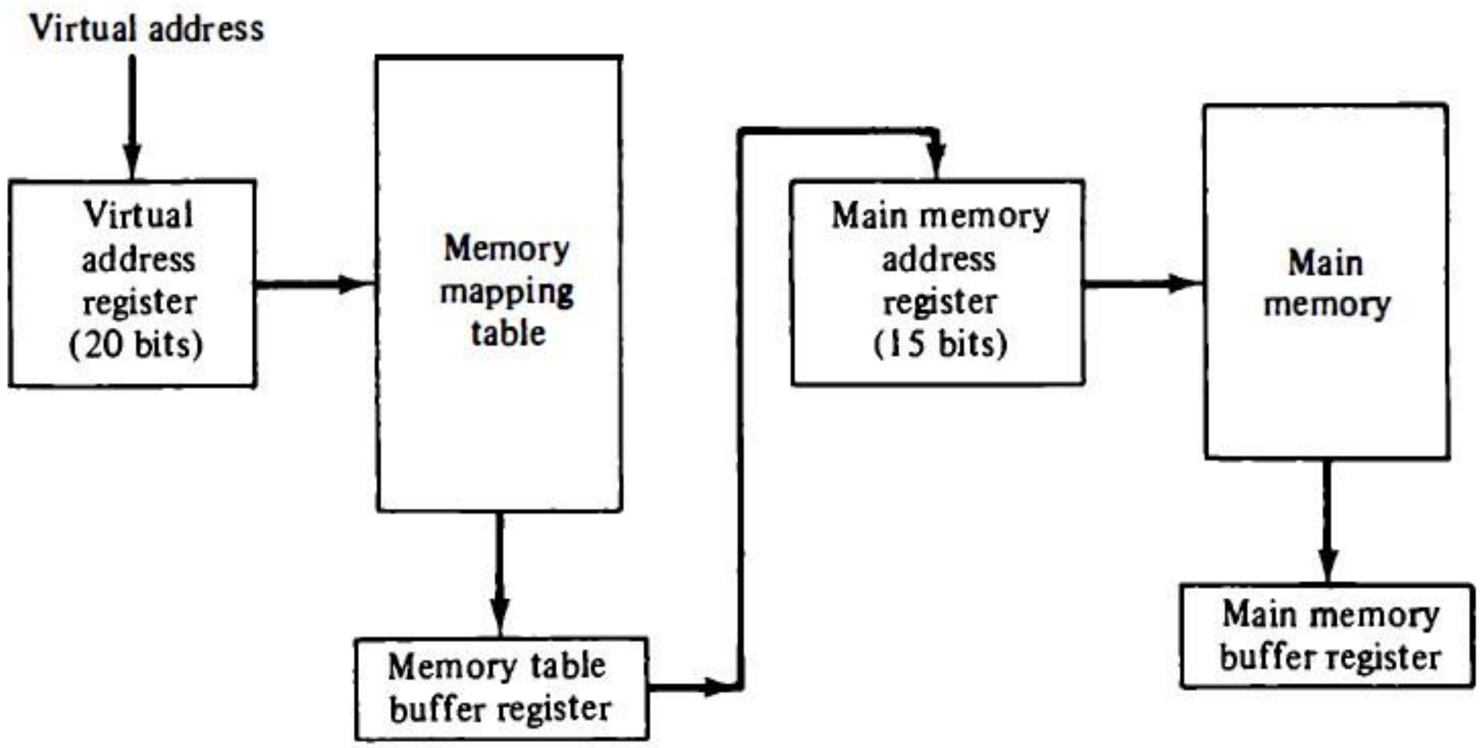
Virtual Memory

- Virtual memory is a concept used in some large computer systems that permit the user to construct programs as though a large memory space were available, equal to the totality of auxiliary memory.

- Each address that is referenced by the CPU goes through an address mapping from the so-called virtual address to a physical address in main memory

- 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 = 215• 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.

Memory Table for mapping a Virtual Address


Virtual address
Virtual address register (20 bits)
Memory mapping table
Memory table buffer register
Main memory address register (15 bits)
Main memory
Main memory buffer register

- The mapping table may be stored in a separate memory as shown in Figure 4.14 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

Input Or Output Interface

- Input-output interface provides a method for transferring information between internal storage and external I/0 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.

- 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.

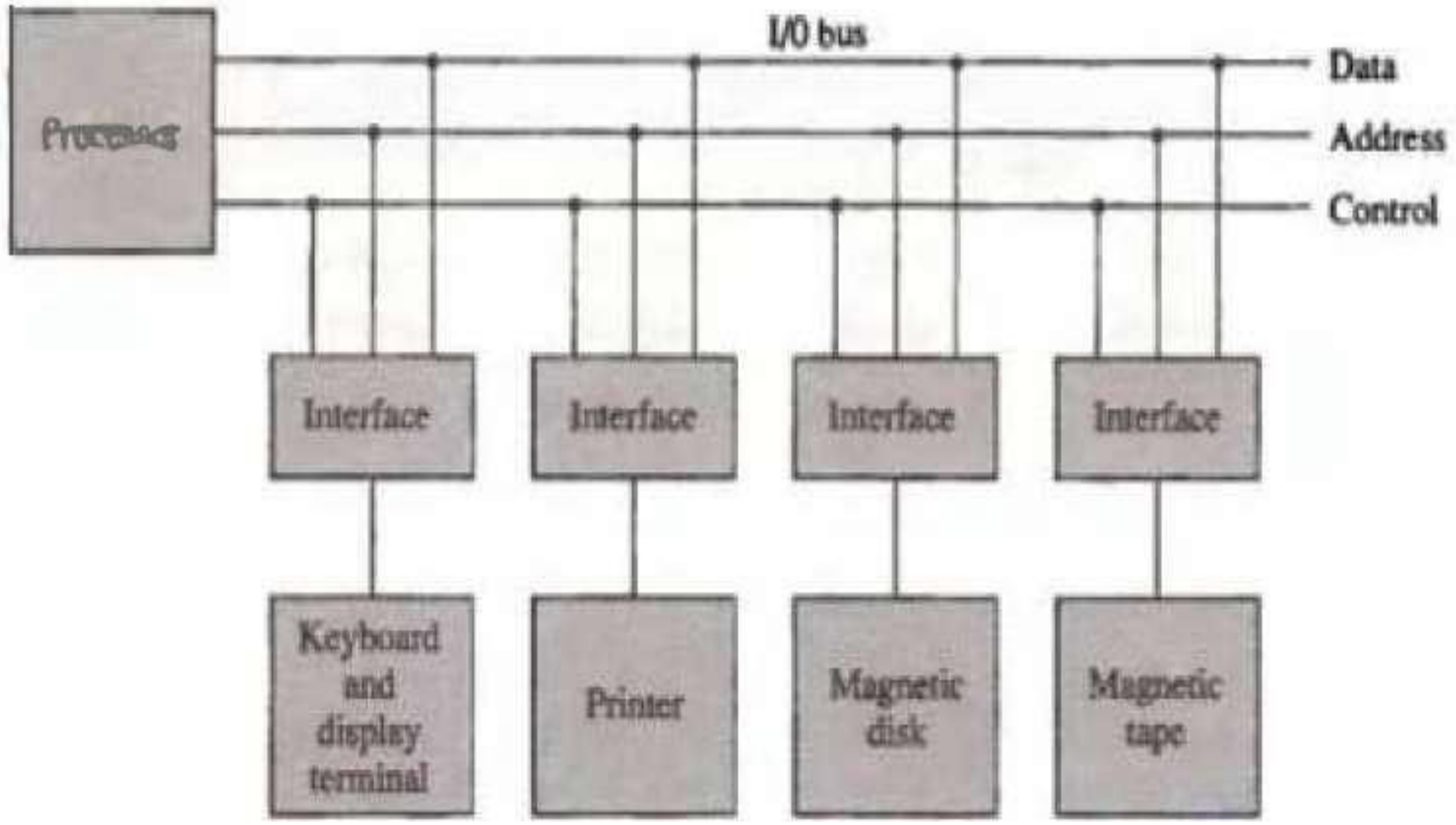
- The data transfer rate of peripherals is usually slower than the transfer rate of the CPU, and consequently, a synchronization mechanism may be needed.

- Data codes and formats in peripherals differ from the word

format in the CPU and memory.

- 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.

Connection of input-output device



- 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.

I/O Memory Bus versus

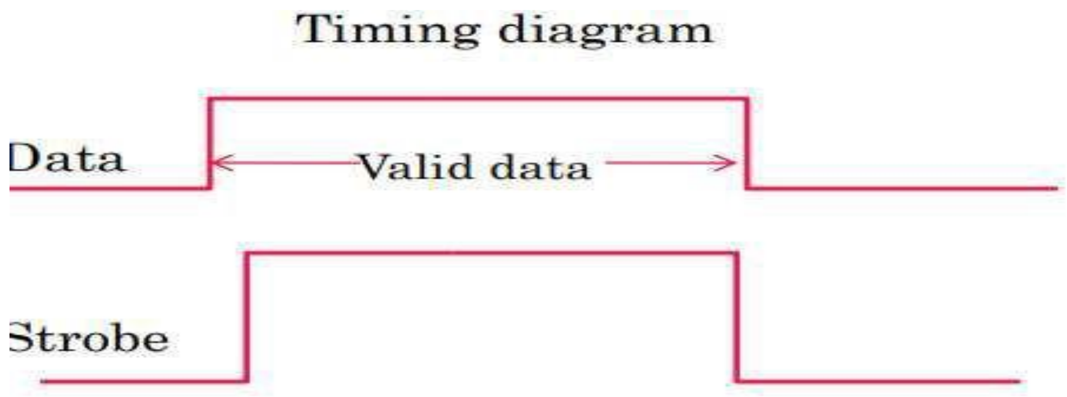
- There are three ways that computer buses can be used to communicate with memory and I/O:

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

Input output organization and memory organization

Source initiated strobe

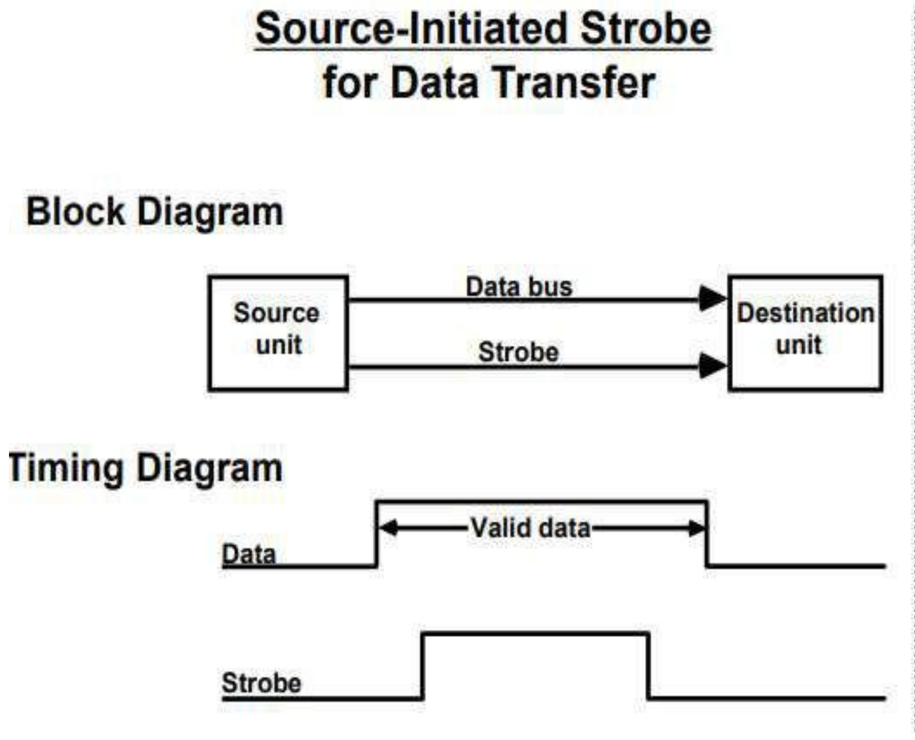
- • The data bus carries the binary information from source unit to the destination unit as shown below.
- • The strobe is a siŶgle liŶe that iŶforŵs the destiŶatioŶ uŶit when a valid data word is available in the bus.

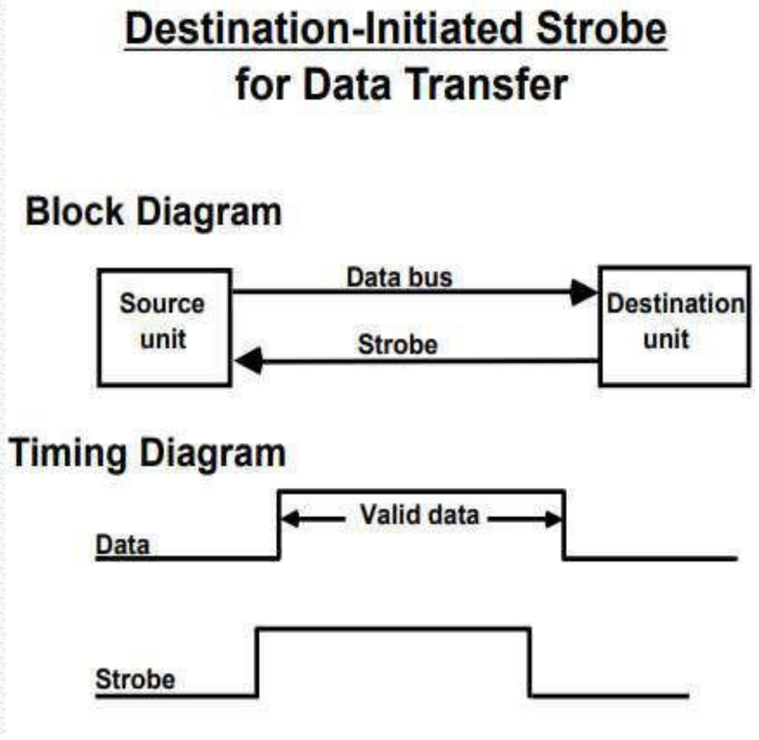


Destination intiated

- First, 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 unit to accept it.
- The data must be valid and remain in the bus long enough for the destination unit to accept it.

Data transfers





Asynchronus Data Transfer

- The transfer of data between two units may be done in parallel or serial. In parallel data transmission, each bit of the message has its own path and the total message is transmitted at the same time.

- This means that an n-bit message must be transmitted through n separate conductor paths. In serial data transmission, each bit in the message is sent in sequence one at a time. This method requires the use of one pair of conductors or one conductor and a common ground.

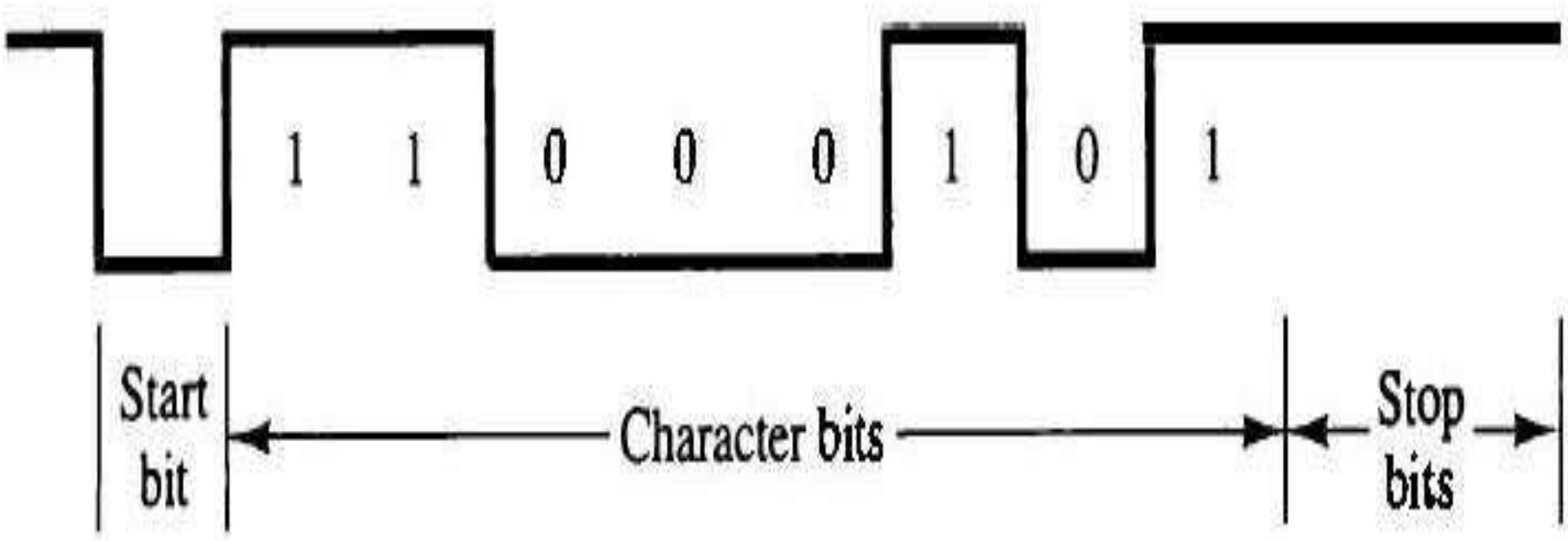
A transmitted character can be detected by the receiver from
knowledge of the transmission rules:

•When a character is not being sent, the line is kept in the 1-state.

•The initiation of a character transmission is detected from the start bit, which is always 0.

•The character bits always follow the start bit.

•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


1
1
0
0
0
1
0
1
Start bit
Character bits
Stop bits

- As an illustration, consider the serial transmission of a terminal whose transfer rate is 10 characters per second.

- Each transmitted character consists of a start bit, eight information bits, and two stop bits, for a total of 11 bits.

- Ten characters per second means that each character takes 0 .1 s for transfer. Since there are 11 bits to be transmitted, it follows that the bit time is 9.09

Input output organization and memory organization

Modes of transfer

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

A. Programmed I/O

B. Interrupt-initiated I/O

C. Direct memory access (DMA)

PROGRAMMED I/0

Programmed I/O operations are the result of I/O instruction 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.

Once a data transfer is initiated, the CPU is required to monitor the interface to see when a transfer can again be made

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

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).

Priority Interrupt

- Data transfer between the CPU and all I/O devices is initiated by the CPU.

- However, the CPU cannot start the transfer unless the device is ready to communicate with the CPU. The readiness of the device can be determined from an interrupt signal.

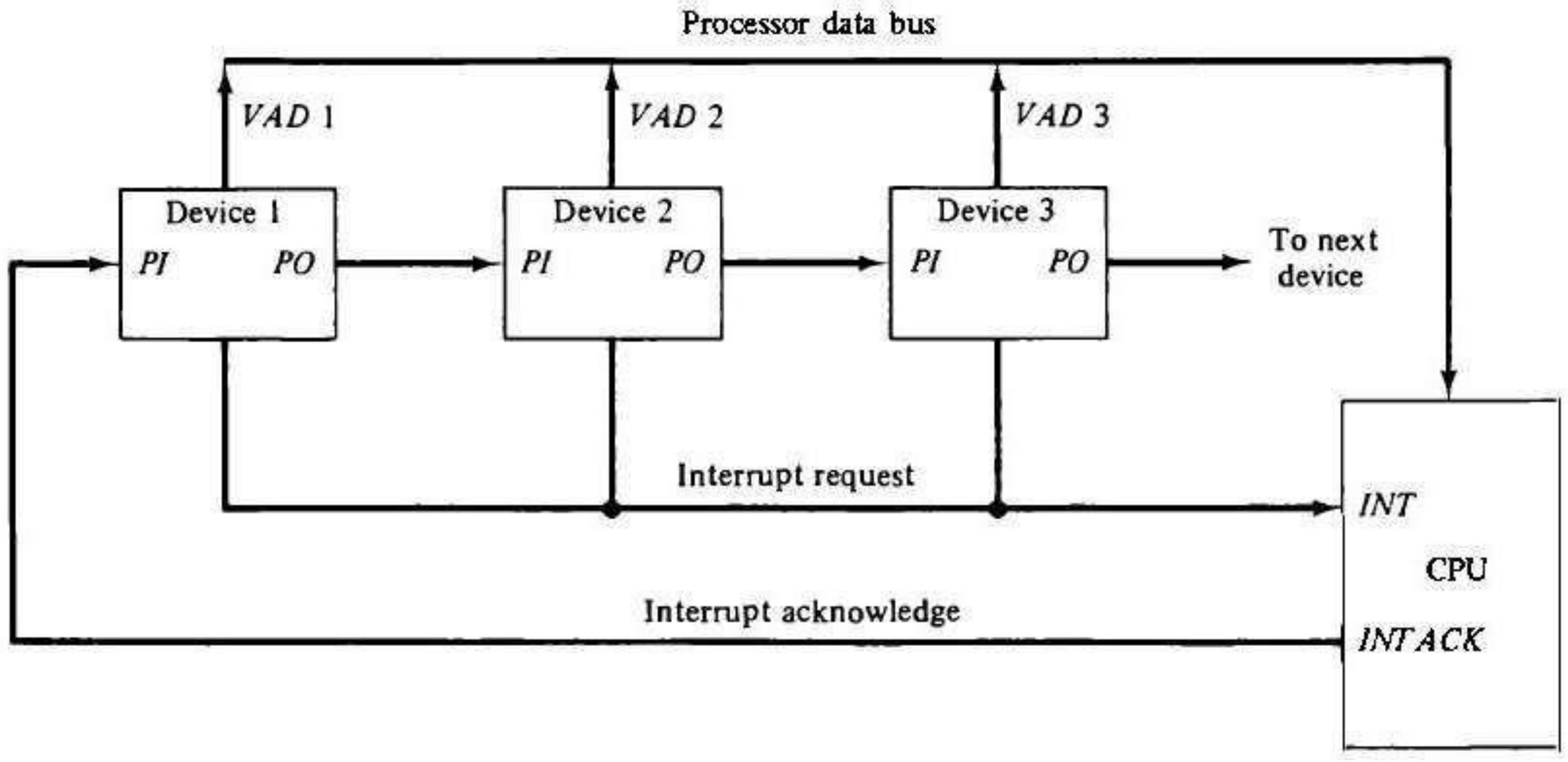
- The CPU responds to the interrupt request by storing the return address from PC into a memory stack and then the program branches to a service routine that processes the required transfer.

- **Daisy-Chaining Priority**

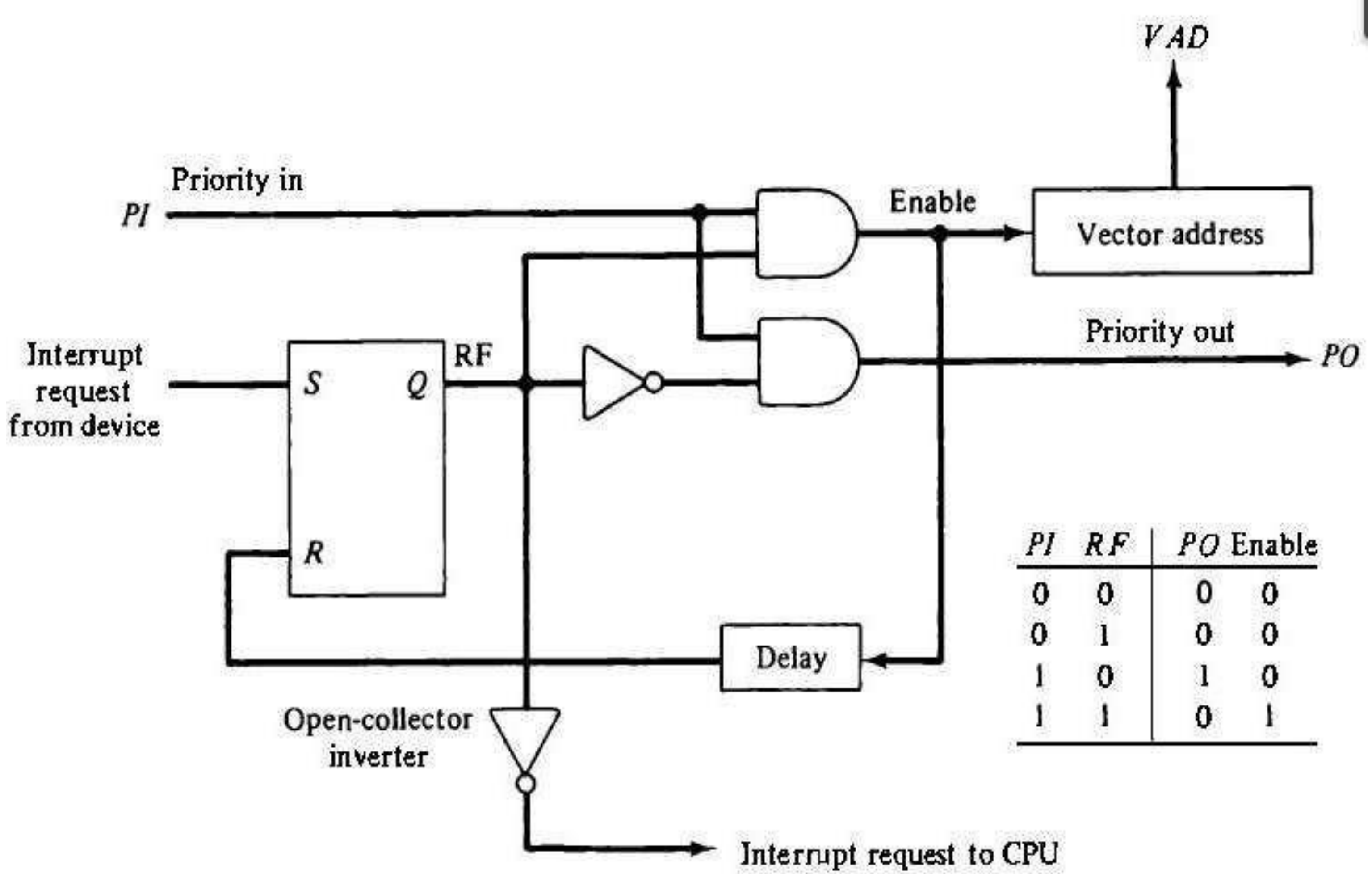
- The daisy-chaining method of establishing priority consists of a serial connection of all devices that request an interrupt.

- The device with the highest priority is placed in the first position, followed by lower-priority devices up to the device with the lowest priority, which is placed last in the chain.

Daisy-Chain Priority Interrupt

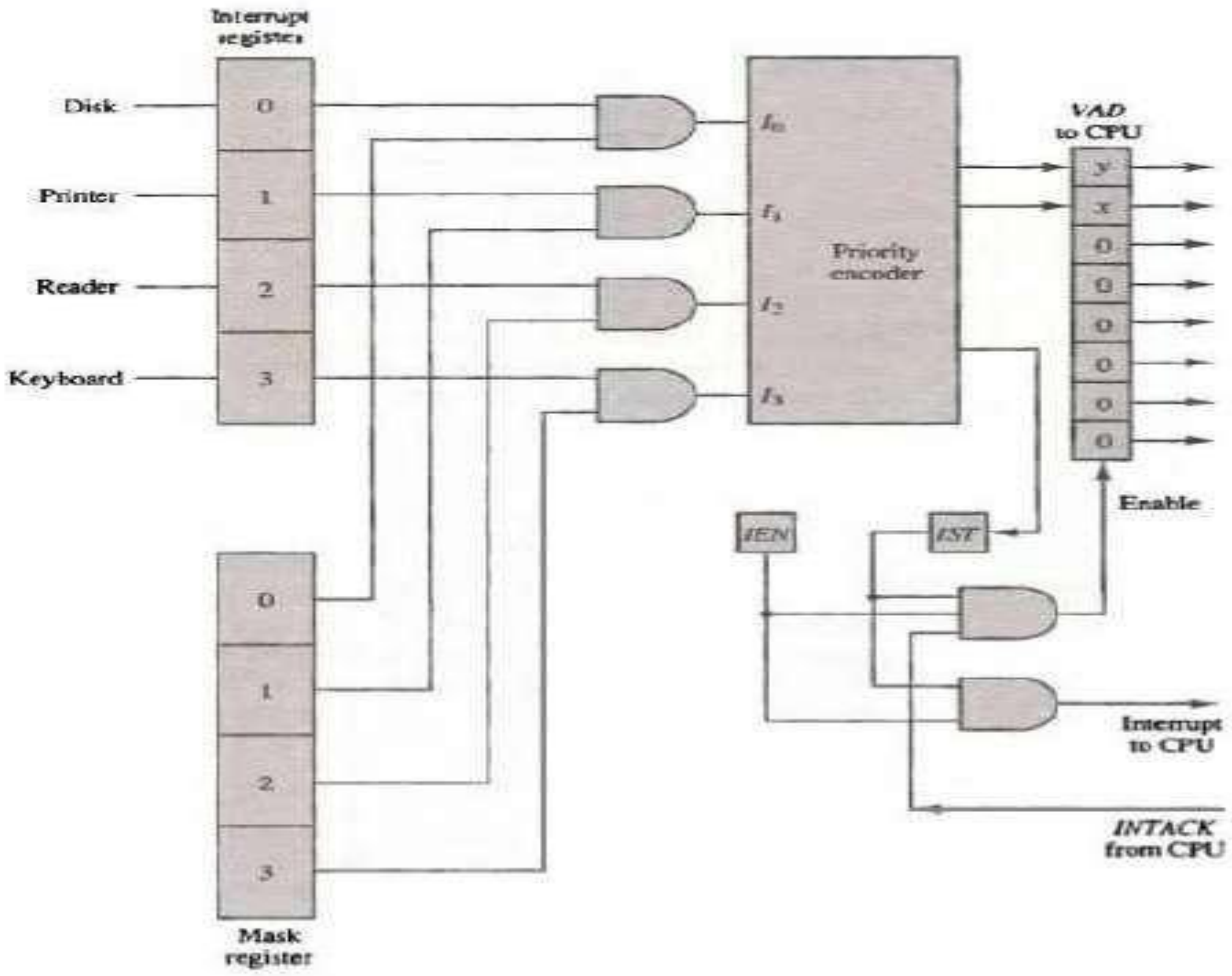


One stage of the Daisy – Chain Priority

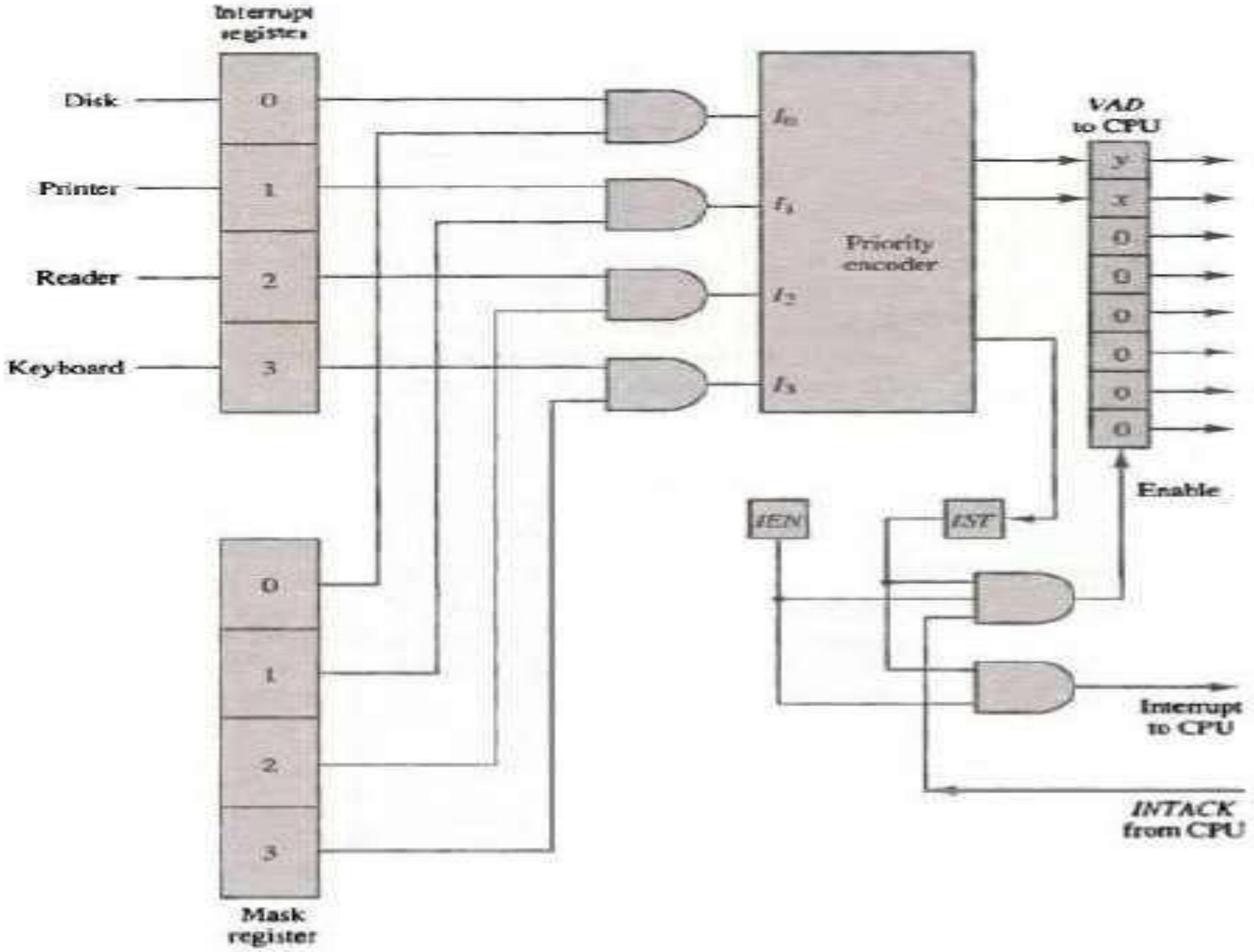


PI	RF	PO	Enable
0	0	0	0
0	1	0	0
1	0	1	0
1	1	0	1

Priority Interrupt Hardware



Priority Interrupt

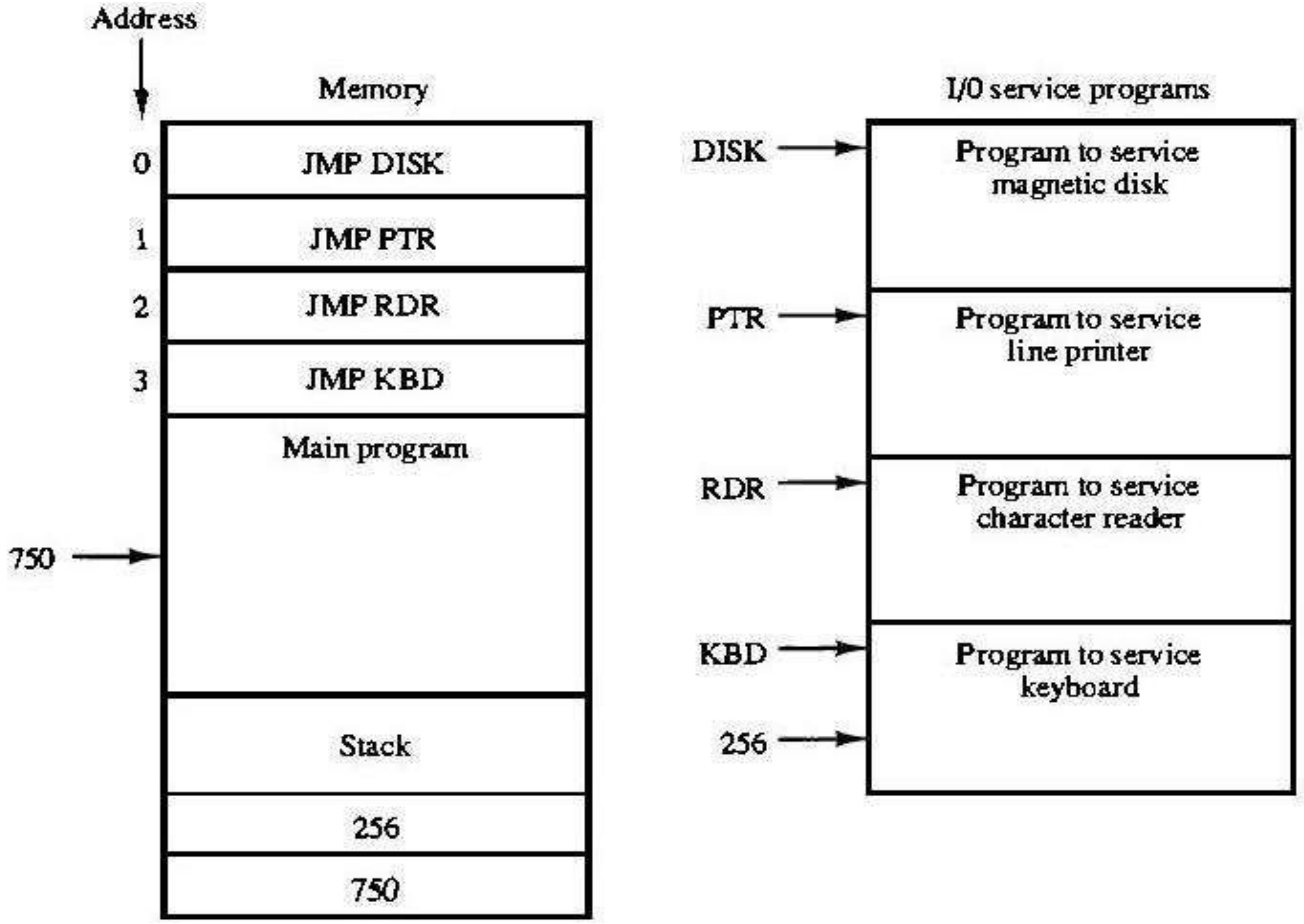

Interrupt register
Disk
Printer
Reader
Keyboard
0
1
2
3
I_0
I_1
I_2
I_3
Priority encoder
VAD to CPU
y
x
0
0
0
0
0
0
Enable
IEN
IST
Interrupt to CPU
INTACK from CPU
Mask register

- 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.
- So far we have discussed the hardware aspects of a priority interrupt system.
- The computer must also have software routines for servicing the interrupt requests and for controlling the interrupt hardware registers.


Address
Memory
0
JMP DISK
1
JMP PTR
2
JMP RDR
3
JMP KBD
Main program
750
Stack
256
750
I/0 service programs
DISK
Program to service magnetic disk
PTR
Program to service line printer
RDR
Program to service character reader
KBD
Program to service keyboard
256

- The initial sequence of each interrupt service routine must have instructions to control the interrupt hardware in the following manner:

- Clear lower-level mask register bits.
- Clear interrupt status bit IST.
- Save contents of processor registers.
- Set interrupt enable bit IEN.
- Proceed with service routine.

- The final sequence in each interrupt service routine must have instructions to control the interrupt hardware in the following manner:

- Gear interrupt enable bit IEN.
- Restore contents of processor registers.
- Clear the bit in the interrupt register belonging to the source

that has been serviced.

- Set lower-level priority bits in the mask register.
- Restore return address into PC and set IEN

Input output organization and memory organization

DMA

The direct memory access (DMA) I/O technique provides direct access to the memory while the microprocessor is temporarily disabled.

I/O devices are connected to system bus via a special interference circuit known as DMA Controller.

In DMA, both CPU and DMA controller have access to main memory via a shared system bus having data, address and control lines.

DMA

- A DMA controller temporarily borrows the address bus, data bus, and control bus from the microprocessor and transfers the data bytes directly between an I/O port and a series of memory locations.

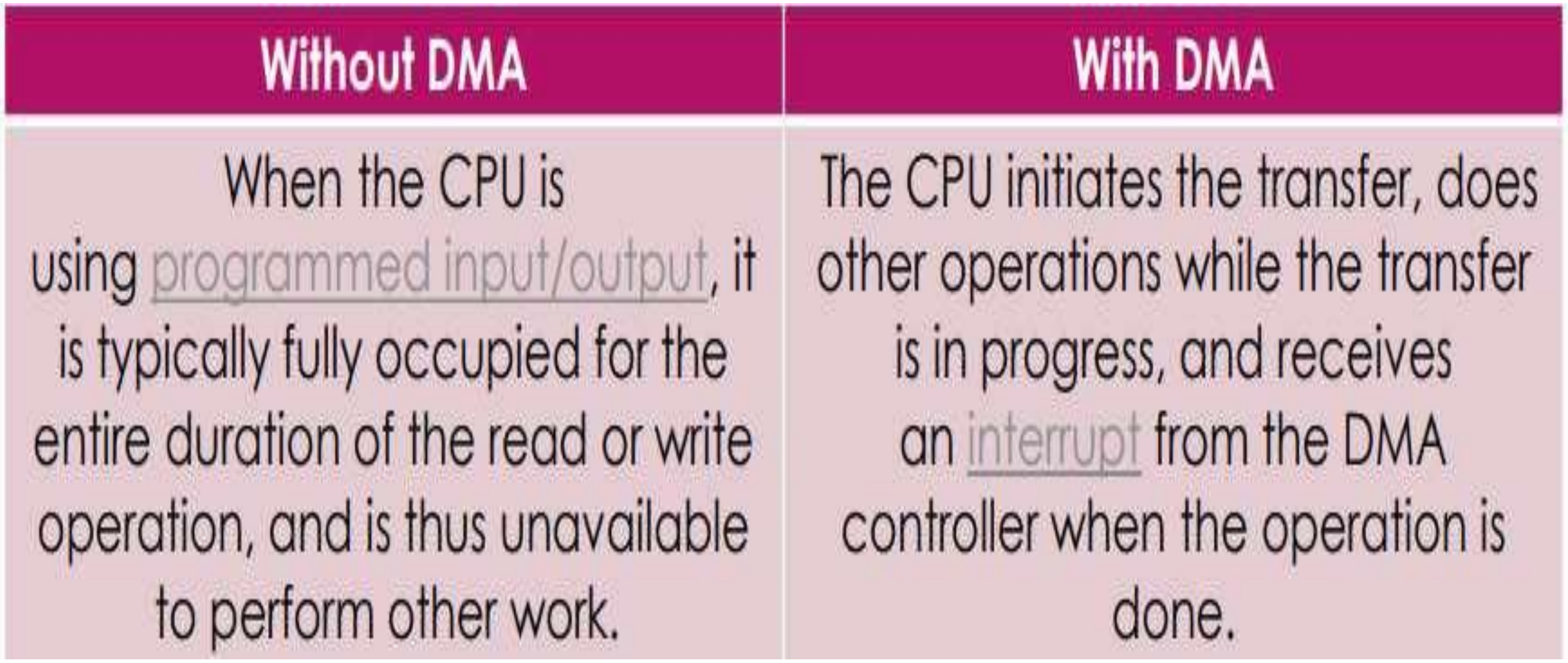
- The DMA transfer is also used to do high-speed memory-to memory transfers.

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).

DMA



Without DMA	With DMA
When the CPU is using programmed input/output, it is typically fully occupied for the entire duration of the read or write operation, and is thus unavailable to perform other work.	The CPU initiates the transfer, does other operations while the transfer is in progress, and receives an interrupt from the DMA controller when the operation is done.

Input output organization and memory organization

Dma Controllers

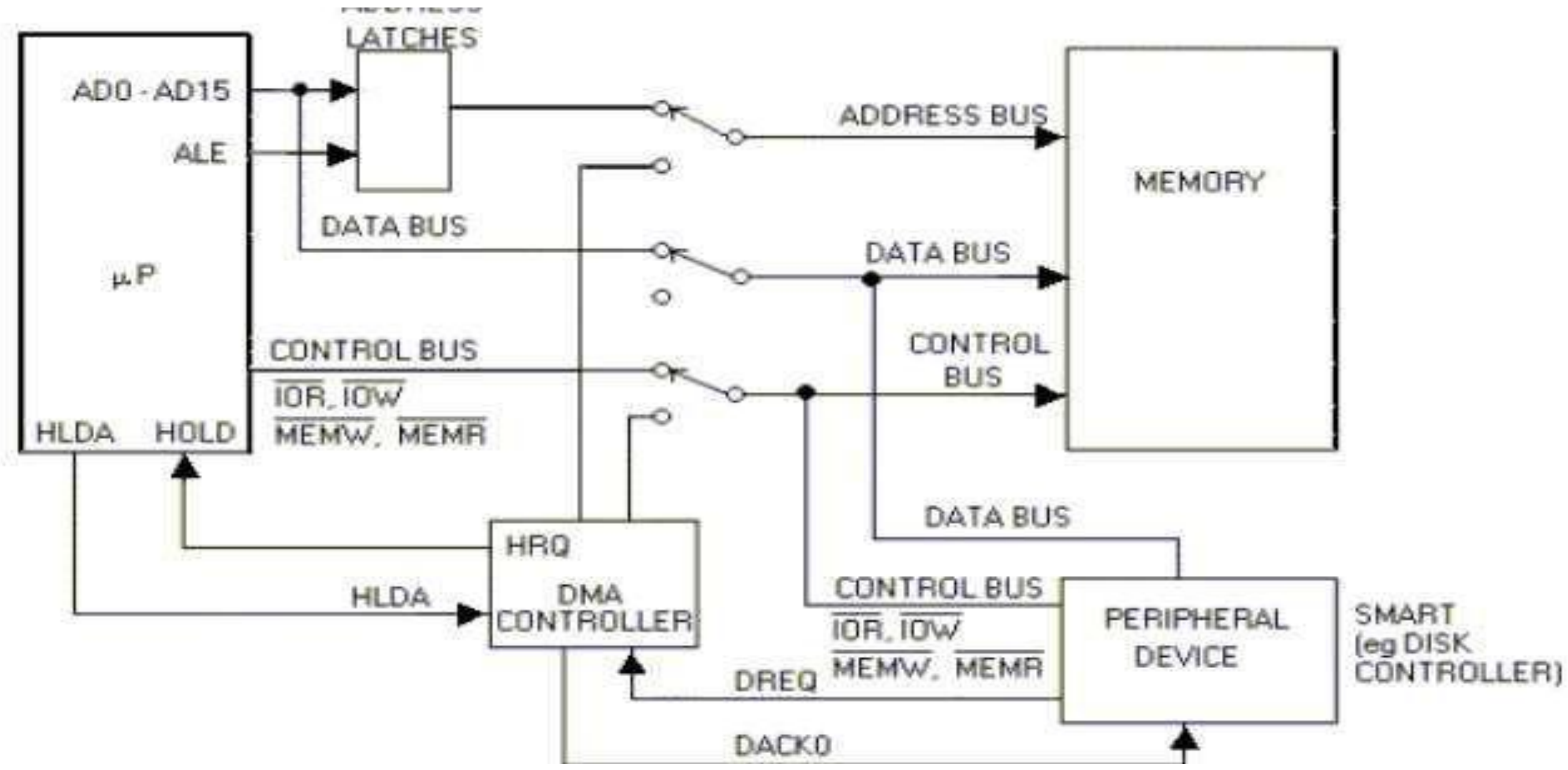
- DMA controllers require initialization by software. Typical setup parameters include the base address of the source area, the base address of the destination area, the length of the block, and whether the DMA controller should generate a processor interrupt once the block transfer is complete.

Dma Transfers

- In this DMA mode , DMA controller is master of memory bus. This mode is needed by the secondary memory like disk drives, that have data transmission and are no to be stopped or slowed without any loss of data transfer of blocks.

- Block DMA transfer supports faster I/O data transfer rates but the CPU remains inactive for relatively long period by tieing up the system bus.

Dma Controller



LATCHES
AD0 - AD15
ALE
μP
DATA BUS
ADDRESS BUS
MEMORY
DATA BUS
CONTROL BUS
$\overline{IOR}$, $\overline{IOW}$
$\overline{MEMW}$, $\overline{MEMR}$
CONTROL BUS
HLDA
HOLD
DATA BUS
HRQ
HLDA
DMA CONTROLLER
CONTROL BUS
$\overline{IOR}$, $\overline{IOW}$
$\overline{MEMW}$, $\overline{MEMR}$
PERIPHERAL DEVICE
SMART (eg DISK CONTROLLER)
DREQ
DACK0

ADVANTAGES

- DMA allows a peripheral device to read from/write to memory without going through the CPU .
- DMA allows for faster processing since the processor can be working on something else while the peripheral can be populating memory.

Disadvantages

- DMA transfer requires a DMA controller to carry out the operation, hence cost of the system increases.
- Cache Coherence problems.

Pipeline: Parallel processing

Pipeline: 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

Pipeline: Parallel processing

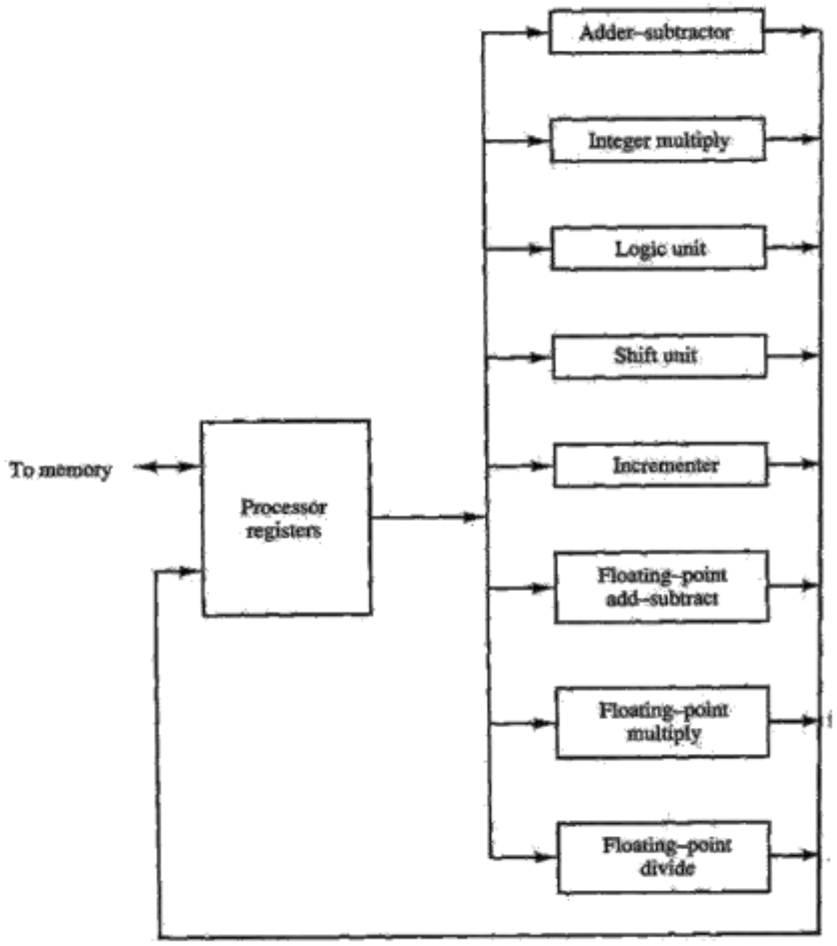
- Parallel processing increases the amount of hardware required

- 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

Pipeline: Parallel processing

Figure 9-1 Processor with multiple functional units.



Pipeline: Parallel processing

Figure 9-2 Example of pipeline processing.

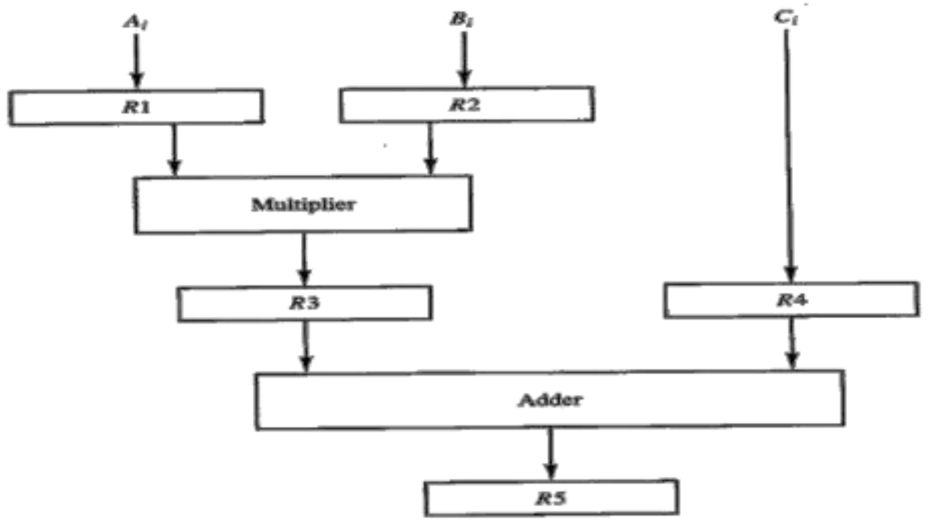


TABLE 9-1 Content of Registers in Pipeline Example

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	$R1$	$R2$	$R3$	$R4$	$R5$
1	A_1	B_1	—	—	—
2	A_2	B_2	$A_1 * B_1$	C_1	—
3	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
5	A_5	B_5	$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$
6	A_6	B_6	$A_5 * B_5$	C_5	$A_4 * B_4 + C_4$
7	A_7	B_7	$A_6 * B_6$	C_6	$A_5 * B_5 + C_5$
8	—	—	$A_7 * B_7$	C_7	$A_6 * B_6 + C_6$
9	—	—	—	—	$A_7 * B_7 + C_7$

Pipelining-arithmetic pipeline

Pipelining-arithmetic pipeline

- **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

Pipelining-arithmetic pipeline

$X = A \times 2^{a}$ $Y = B \times 2^{b}$

•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 o Align the mantissaso Add or subtract the mantissas

•Normalize the result

Pipelining-arithmetic pipeline

Pipelining-arithmetic pipeline

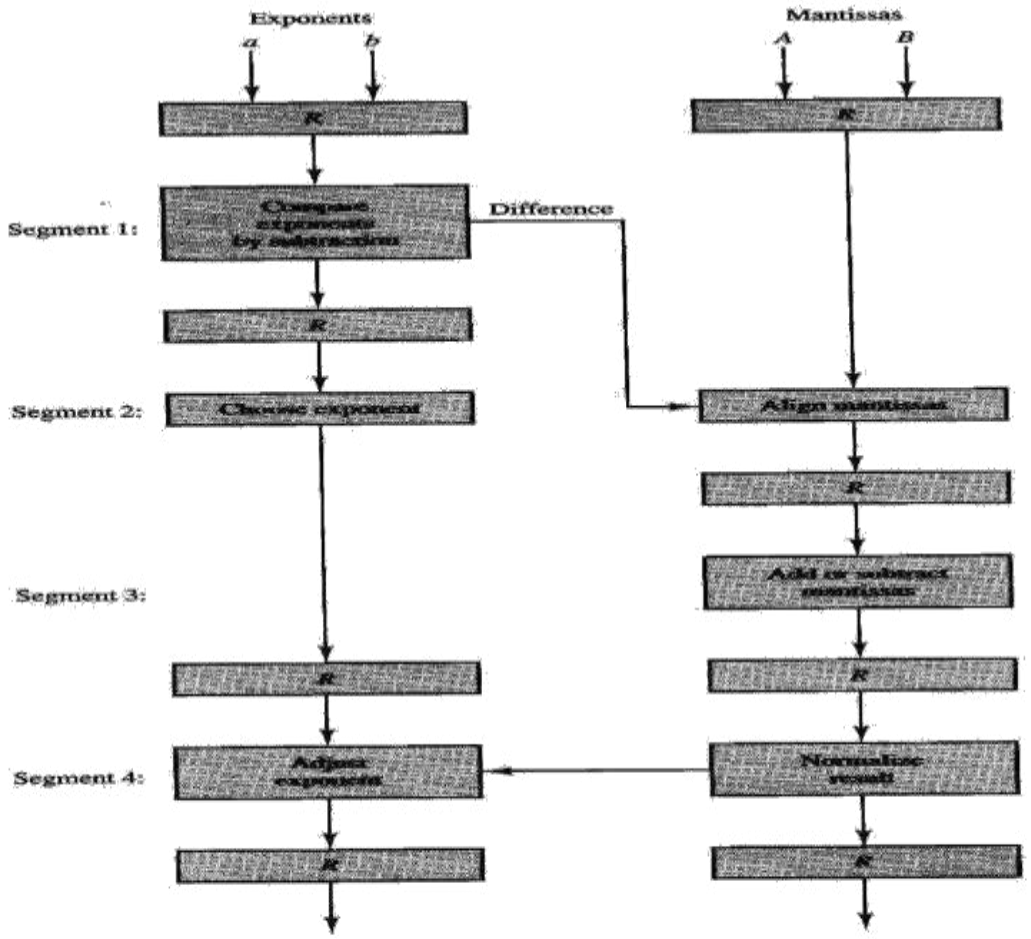


Figure 9-6 Pipeline for floating-point addition and subtraction.

Pipelining-arithmetic pipeline

- $X = 0.9504 \times 10^3$ and $Y = 0.8200 \times 10^2$
- The two exponents are subtracted in the first segment to obtain 3-2=1
- The larger exponent 3 is chosen as the exponent of the result
- Segment 2 shifts the mantissa of Y to the right to obtain $Y = 0.0820 \times 10^3$
- The mantissas are now aligned
- Segment 3 produces the sum $Z = 1.0324 \times 10^3$
- Segment 4 normalizes the result by shifting the mantissa once to the right and incrementing the exponent by one to obtain $Z = 0.10324 \times 10^4$

Unit-V

Inter processor communication and synchronization

Inter processor communication

- Interprocess communication (IPC) is a set of programming interfaces that allow a programmer to coordinate activities among different program processes that can run concurrently in an operating system.

- This allows a program to handle many user requests at the same time. Since even a single user request may result in multiple processes running in the operating system on the user's behalf, the processes need to communicate with each other.

- The IPC interfaces make this possible. Each IPC method has its own advantages and limitations so it is not unusual for a single program to use all of the IPC methods.

Inter processor communication

•In computer science, inter-process communication or interprocess communication (IPC) refers specifically to the mechanisms an operating system provides to allow the processes to manage shared data.

•Typically,applicati ons can use IPC, categorized as clients and servers, wh client requests data and the server responds to client requests.[1] Many a pplications are both clients and servers, as commonly seen in distributed computing.

• Methods for doing IPC are divided into categories which vary based on software requirements, such as performance and modularity requirements, and system circumstances, such as network bandwidth and latency.

Inter processor communication

Interprocessor Arbitration Dynamic Arbitration :

•The IPC mechanism can be classified into pipes, first in, first out (FIFO), and shared memory. Pipes were introduced in the UNIX operating system. In this mechanism, the data flow is unidirectional.

•A pipe can be imagined as a hose pipe in which the data enters through one end and flows out from the other end. A pipe is generally created by invoking the pipe system call, which in turn generates a pair of file descriptors. Descriptors are usually created to point to a pipe node.

- One of the main features of pipes is that the data flowing through a pipe is transient, which means data can be read from the read descriptor only once. If the data is written into the write descriptor, the data can be read only in the order in which the data was written.

Unit-V

Instruction Pipeline

Instruction Pipeline

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

Instruction Pipeline

- The following steps are needed to process each instruction:
    - Fetch the instruction from memory
    - Decode the instruction
    - Calculate the effective address
    - Fetch the operands from memory
    - Execute the instruction
    - Store the result in the proper place

Instruction Pipeline

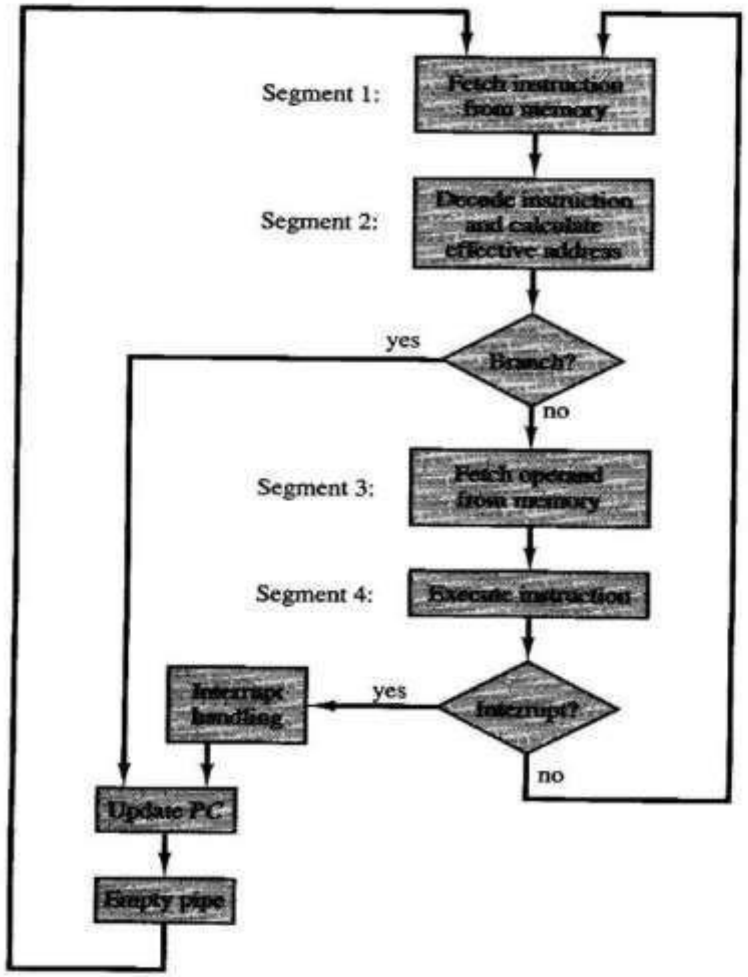


Figure 9-7 Four-segment CPU pipeline.

Unit-V

Characteristics of Multiprocessors

Characteristics of multiprocessors

Characteristics of Multiprocessors:

•A Multiprocessor system is an interconnection of Two or more
CPU’s with ŵeŵory aŶd iŶput-output equipment

•Multiprocessors system are classified as multiple instruction stream, multiple data stream systems(MIMD)

•There exists a distinction between multiprocessor And multi computers that though both supportConcurrent operations.In mult computers several Autonomous computers are connected through a network and they may or may not communicate But in a multiprocessor system there is a single OS Control that provides interaction between processors and all the components of the system To be cooperate in the solution of the problem

Characteristics of multiprocessors

Benefits of Multiprocessing:

•Multiprocessing increases the reliability of the system so that a failur or error in one part has limited effect on the rest of the system. If a fau causes one processor to fail, a second processor can be assigned to perform the functions of the disabled one.

•Improved System performance. System derives high performance from the fact that computations can proceed in parallel in one of the two ways:

- Multiple independent jobs can be made to operate in parallel.

Characteristics of multiprocessors

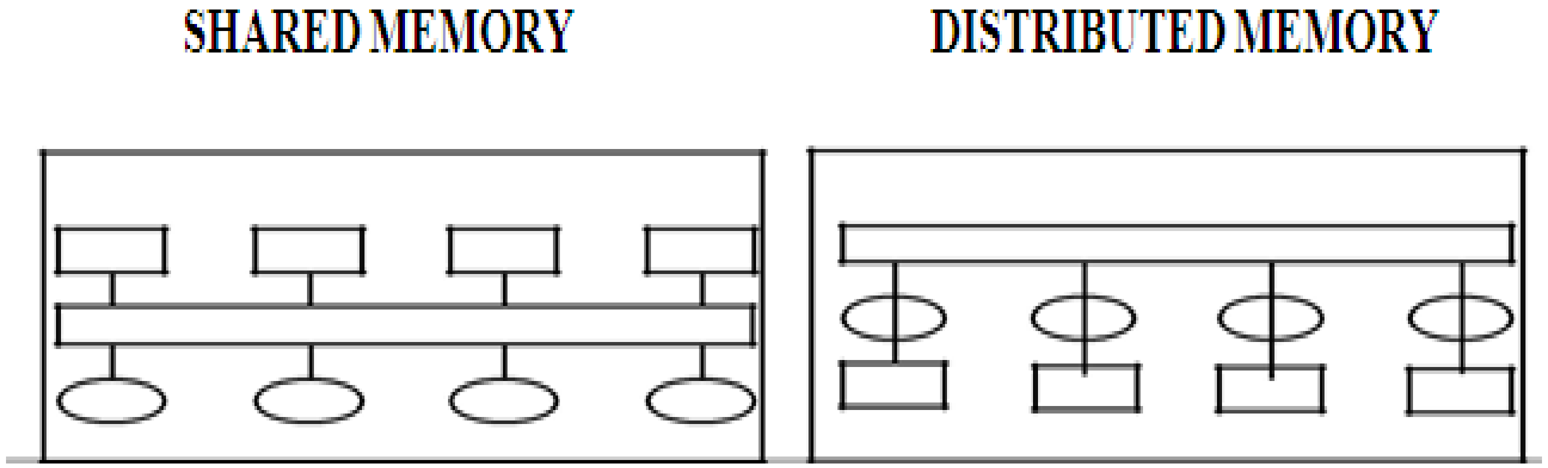
- A single job can be partitioned into multiple parallel tasks. This

can be achieved in two ways:

- The user explicitly declares that the tasks of the program be executed in parallel

- The compiler provided with multiprocessor s/w that can automatically detect parallelism in program. Actually it checks for Data dependency.

Characteristics of multiprocessors



Unit-V

INTER CONNECTION STRUCTURES

INTER CONNECTION STRUCTURES

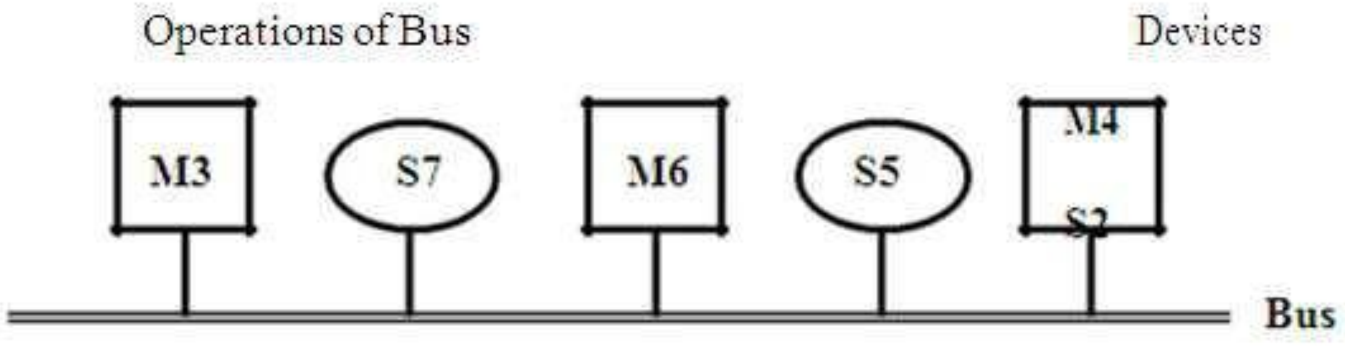
Inter Connection Structures:

•The interconnection between the components of a multiprocessor System can have different physical configurations depending n the number of transfer paths that are available between the processors and memory in a shared memory system and among the processing elements in a loosely coupled system.

•Some of the schemes are as: Time-Shared Common Bus Multiport Memory

INTER CONNECTION STRUCTURES

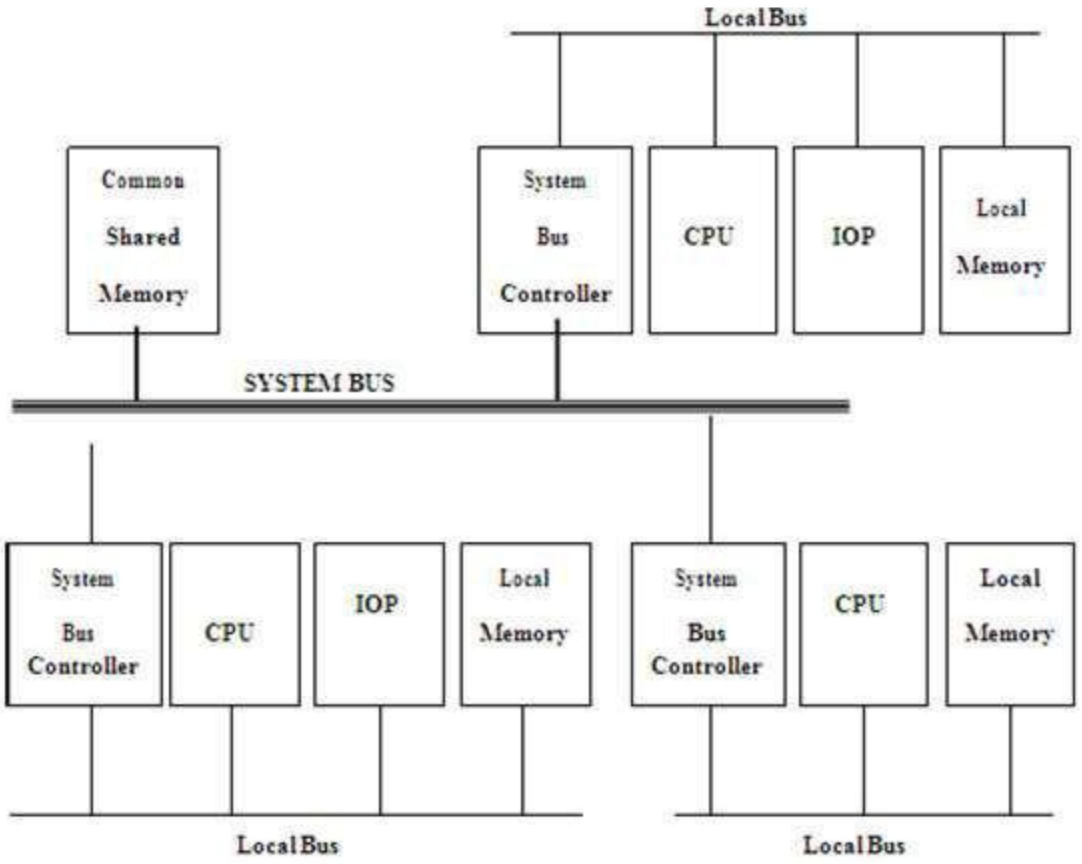
- Crossbar Switch
- Multistage Switching Network
- Hypercube System



M3 wishes to communicate with S5

INTER CONNECTION STRUCTURES

- .



Unit-V

INTER CONNECTION STRUCTURES

INTER CONNECTION STRUCTURES

Inter Connection Structures:

•The interconnection between the components of a multiprocessor system can have different physical configurations depending n the number of transfer paths that are available between the processors and

•memory in a shared memory system and among the processing elements in a loosely coupled system.

•Some of the schemes are as:

•Time-Shared Common Bus

•Multiport Memory

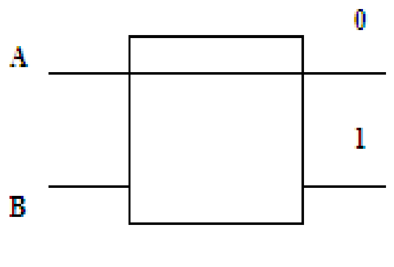
•Crossbar Switch

•Multistage Switching Network

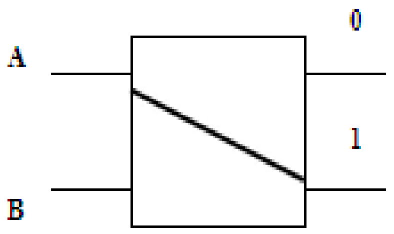
•Hypercube System

INTER CONNECTION STRUCTURES

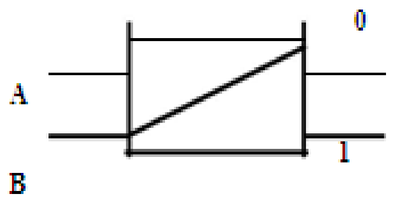
MULTISTAGE SWITCHING NETWORK



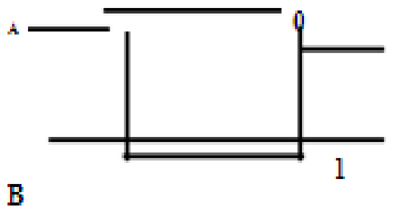
A connected to 0



A connected to 1



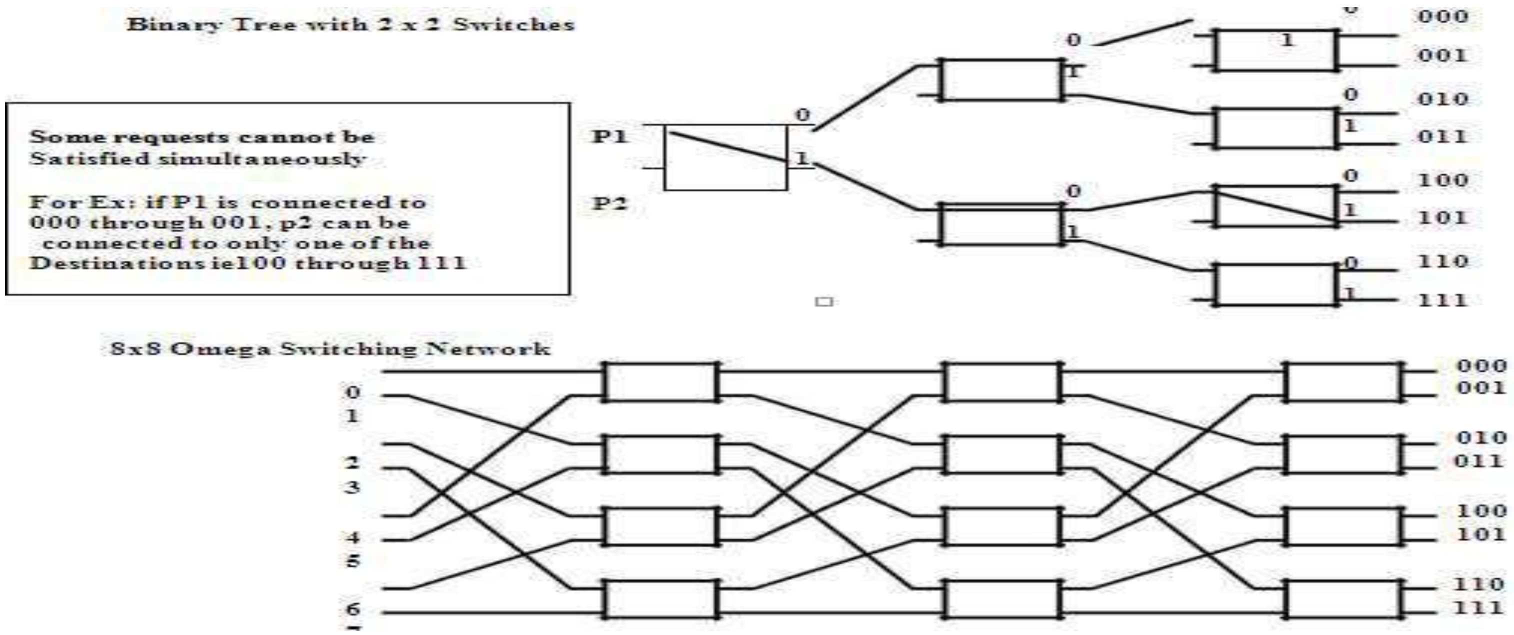
B connected to 0



B connected to 1

INTER CONNECTION STRUCTURES

MULTISTAGE INTERCONNECTION NETWORK



Binary Tree with 2 x 2 Switches
Some requests cannot be Satisfied simultaneously
For Ex: if P1 is connected to 000 through 001, p2 can be connected to only one of the Destinations ie100 through 111
P1
P2
0
1
000
001
010
011
100
101
110
111
8x8 Omega Switching Network
0
1
2
3
4
5
6
7

HYPERCUBE **INTERCONNECTION**

Unit-V

Inter Processor Arbitration

Inter Processor Arbitration

INTER PROCESSOR ARBITRATION:

•Only one of CPU, IOP, and Memory can be granted to use the bus at a time

•Arbitration mechanism is needed to handle multiple requests to the shared resources to resolve multiple contention.

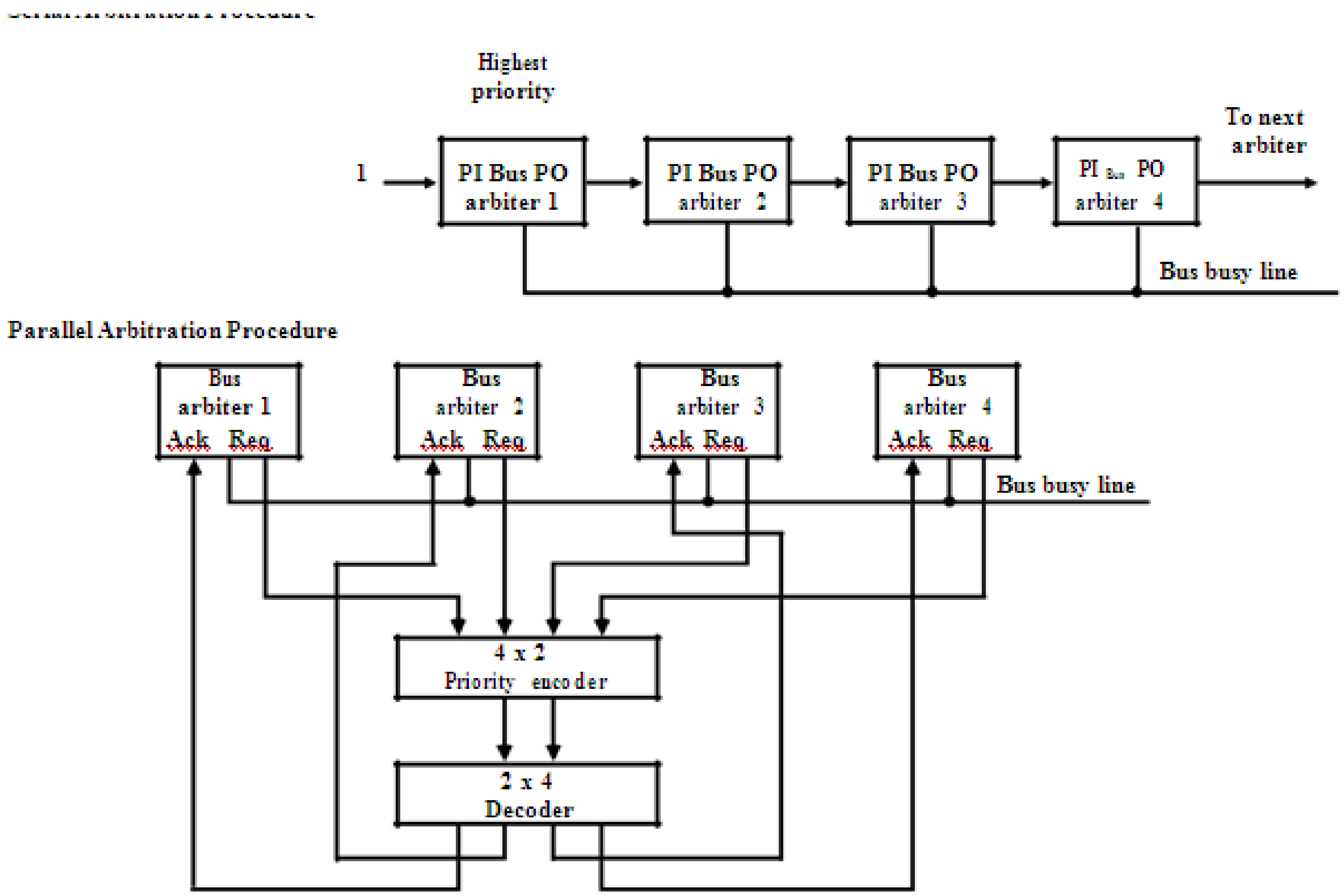
SYSTEM BUS:

•A ďus that ĐoŶŶeĐts the ŵajor ĐoŵpoŶeŶts suĐh as CPU’s, IOP’s and memory

•A typical System bus consists of 100 signal lines divided into three functional groups: data, address and control lines. In addition there are power distribution lines to the components.

Inter Processor Arbitration

Interprocessor Arbitration Static Arbitration:



INTERPROCESSORARBITRATION DYNAMIC ARBITRATION

Inter Processor Arbitration

Interprocessor Arbitration Dynamic Arbitration :

Priorities of the units can be dynamically changeable while the system is in operation

Time Slice

Fixed length time slice is given sequentially to each processor, round-robin fashi on

Polling

Unit address polling - Bus controller advances the address to identify the requesting unit. When processor that requires the access recognizes its address, it activates the bus busy line and then accesses the bus. After a number of bus cycles, the polling continues by choosing a different processor.

Inter processor communication and synchronization

Inter processor communication

•Interprocess communication (IPC) is a set of programming interfaces that allow a programmer to coordinate activities among different program processes that can run concurrently in an operating system.

•This allows a program to handle many user requests at the same time. Since even a single user request may result in multiple processes running in the operating system on the user's behalf, the processes need to communicate with each other.

•The IPC interfaces make this possible. Each IPC method has its own advantages and limitations so it is not unusual for a single program to use all of the IPC methods.

Inter processor communication

•In computer science, inter-process communication or interprocess communication (IPC) refers specifically to the mechanisms an operating system provides to allow the processes to manage shared data.

•Typically,applications can use IPC, categorized as clients and servers, where the client requests data and the server responds to client requests.

•Many applications are both clients and servers, as commonly seen in distributed
computing.

•

Inter processor communication

Interprocessor Arbitration Dynamic Arbitration :

•The IPC mechanism can be classified into pipes, first in, first out (FIFO), and shared memory. Pipes were introduced in the UNIX operating system. In this mechanism, the data flow is unidirectional.

•A pipe can be imagined as a hose pipe in which the data enters through one end and flows out from the other end. A pipe is generally created by invoking the pipe system call, which in turn generates a pair of file descriptors. Descriptors are usually created to point to a pipe node.

- One of the main features of pipes is that the data flowing through a pipe is transient, which means data can be read from the read descriptor only once.

- If the data is written into the write descriptor, the data can be read only in the order in which the data was written.

Inter processor communication and synchronization

Inter processor communication

- **Synchronization :**

- Communication of control information between processors to enforce the correct sequence of processes

- To ensure mutually exclusive access to shared writable data
- Hardware Implementation

- Mutual Exclusion with a Semaphore

- Mutual Exclusion

- One processor to exclude or lock out access to shared resource by other processors when it is in a Critical Section

- Critical Section is a program sequence that, once begun, must complete execution before another processor accesses the same shared resource

Inter processor communication

Semaphore

•binary variable

1: A processor is executing a critical section, that not available to other processors

0: Available to a ny r e qu e st in g p rocessor
software controlled Flag that is stored in memory that all processors can be access

Inter processor communication

- Testing and Setting the Semaphore
- Avoid two or more processors test or set the same semaphore
- May cause two or more processors enter the
- same critical section at the same time
- Must be implemented with an indivisible operation