CS304PC: COMPUTER ORGANIZATION AND ARCHITECTURE

B.TECH II Year I Sem.

L	T	P	C
3	0	0	3

Co-requisite: A Course on "Digital Logic Design and Microprocessors".

Course Objectives:

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

Course Outcomes:

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

UNIT - I

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

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

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

UNIT - II

Microprogrammed Control: Control memory, Address sequencing, micro program example, design of control unit.

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

UNIT - III

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

Computer Arithmetic: Addition and subtraction, multiplication Algorithms, Division Algorithms, Floating – point Arithmetic operations. Decimal Arithmetic unit, Decimal Arithmetic operations.

UNIT - IV

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

Memory Organization: Memory Hierarchy, Main Memory, Auxiliary memory, Associate Memory, Cache Memory.

UNIT - V

Reduced Instruction Set Computer: CISC Characteristics, RISC Characteristics.

Pipeline and Vector Processing: Parallel Processing, Pipelining, Arithmetic Pipeline, Instruction Pipeline, RISC Pipeline, Vector Processing, Array Processor.

Multi Processors: Characteristics of Multiprocessors, Interconnection Structures, Interprocessor arbitration, Interprocessor communication and synchronization, Cache Coherence.

TEXT BOOK:

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

REFERENCE BOOKS:

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

Digital Computer :-

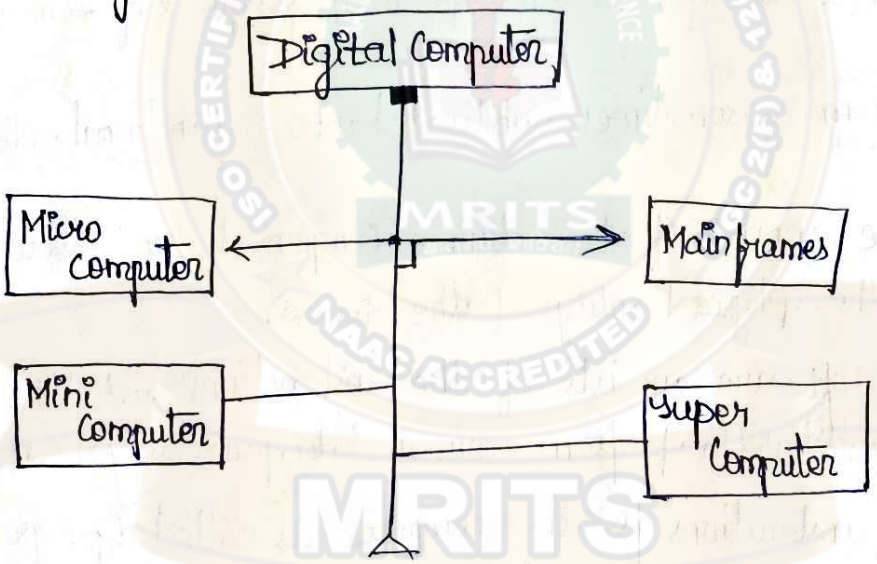
Introduction :-

It is the most commonly used type of Computer and is used to process information with quantities using digits, usually using the binary number system.

* Digital Computers are programmable machines that use Electronic technology to generate, store and process data.

* Digital Computers use the binary number system which has 2 digits 0 and 1. A binary digit is called a bit.

* The two terms positive "1" and nonpositive "0", compose the data into a string.



→ Micro Computer :-

↳ It is a small, relatively inexpensive computer with a microprocessor as its CPU. It includes a microprocessor, memory and I/O devices.

↳ Also known as "personal computer".

↳ Includes workstations, desktops, server, laptop & notebook.

→ Mini Computer :-

↳ Mini computers emerged in the mid-1960s and were first developed by IBM Corporation.

↳ This may also be called a mid-range computer.

↳ Minicomputer may contain 1 or more processors, support multiprocessing & tasking.

→ Mainframes Computer:-

↳ Mainframes are a type of computer that generally are known for their large size, amount of storage, processing power and high level of reliability.

↳ Ability to run (or host) multiple operating systems.

↳ Mainframes first appeared in early 1940's.

→ Super Computer:-

↳ Super Computer consists of tens of thousands of processors that are able to perform billions & trillions of calculations or computations per second.

↳ These are primarily designed to be used in enterprises and organizations that require massive computing power.

↳ It has more than 98,000 processors that allows it to process at a speed of 16,000 trillion calculations per second.

↳ It is a large & very powerful main frame computer called Supercomputer.

↳ Super Computers are applied to the solution of very complex & sophisticated scientific problem & used for national security purposes of some advance nations.

* A Computer System is sometimes subdivided into 2 functional entities : Hardware & software

↳ Hardware consists of all Electronic Components and electromechanical devices that comprise the physical entity of the device.

↳ Computer Software consists of the instructions and data that the computer manipulates to perform various data processing tasks.

* A sequence of instructions for the computer is called a "program".

* The data that are manipulated by the program constitute the "data base".

Block Diagram of Digital Computer:-

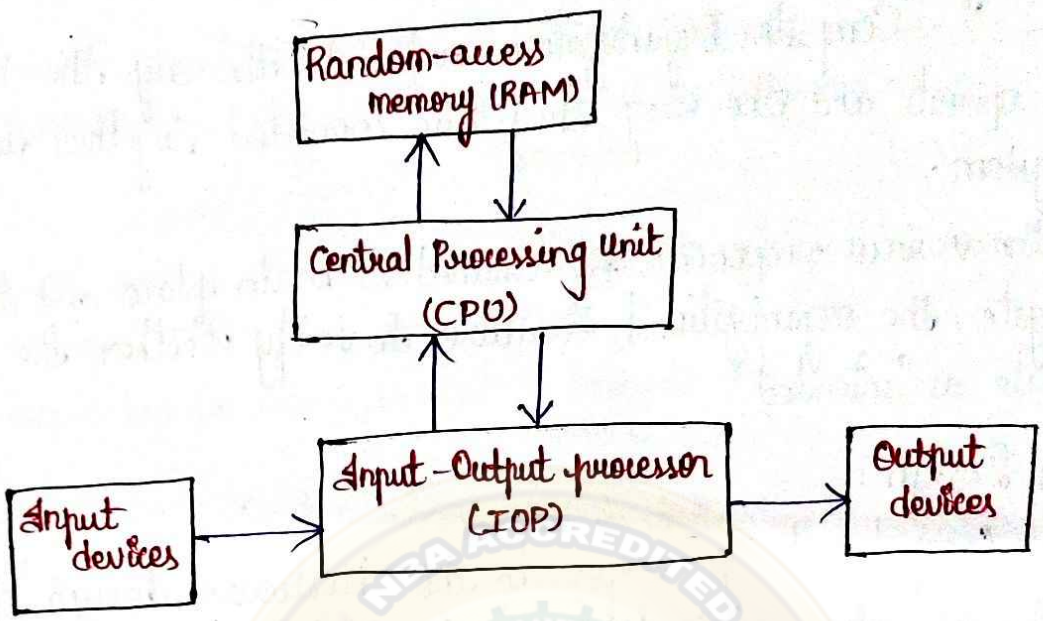


fig: Block diagram of Digital Computer

* The Central processing unit (CPU) contains an arithematic and logic unit for data manipulating data, a number of registers for storing data and control circuits for fetching and executing instructions.
* The memory of a digital Computer contains storage for instructions and data. It is called as Random Access memory (RAM) because the CPU can access any location in memory at random & can retrieve the binary information within a fixed interval of time.
* Input and Output processor (IOP) contains electronic circuits for communicating and controlling the transfer of information b/w the computer and the outside world.
* The Input devices are used by the computer to take input from a user.
  Eg: Mouse, Keyboard, scanner etc;
* The Output devices are used by the computer to give Output to the user.
  Eg:- Printer, Monitor, speaker etc;

Definition of Computer Organization, Computer design & Computer Architecture :-

* Computer Organization :-

Computer Organization refers to the way the hardware components operate and the way they are connected together to form the computer system.

As the various components are assumed to be in place and the task is to investigate the organizational structure to verify whether the computer parts operate as intended.

* Computer Design :-

Computer design refers to the hardware design of the computer

→ Once the computer specifications are formulated it is the task of the designer to develop hardware for the system.

→ The term Computer design is concerned with the determination of what hardware should be used and how parts should be connected.

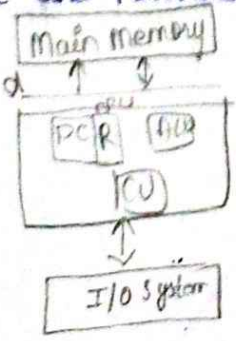
→ As the word computer hardware can be referred to as Computer implementation.

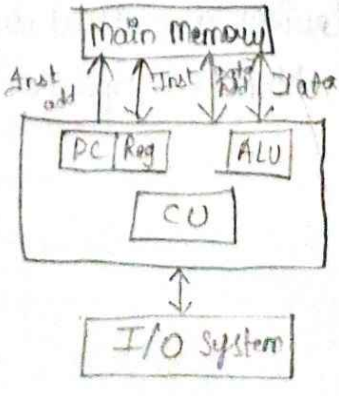
* Computer Architecture :-

Computer Architecture refers to the structure and behavior of the computer as seen by the user. It includes the information, formats, instruction set, and techniques for addressing memory.

The architectural design of a computer system is concerned with the specifications of various functional modules such as processors and memories & structuring them together into a computer system.

There are two basic types of computer architectures are Von Neumann architecture and Harvard architecture





Von Neumann & Harvard Architecture differences

Von-Neumann Architecture	Harvard Architecture
→ It is ancient computer architecture based on stored program computer concept	→ It is modern computer architecture based on Harvard Mark I relay based model
→ Same physical memory address is used for instructions & data.	→ Seperate physical memory address is used for instructions & data.
→ There is common bus for data & instruction transfer.	→ Seperate buses are used for transferring data & instruction.
→ Two clock cycles are required to execute single instruction	→ An instruction is executed in a single cycle.
→ It is cheaper in cost	→ It is costly than Von Neumann architecture.
→ CPU can not access instructions and read/write at the same time.	→ CPU can access instructions and read/write at the same time.
→ It is used in personal computers and small computers	→ It is used in microcontrollers & signal processing.

Difference between Computer Organization and Computer Architecture :-

Computer Organization	Computer Architecture
→ Computer Organization is concerned with the way hardware components are connected together to form a computer system.	→ Computer Architecture is concerned with the structure and behaviour of computer system as seen by the user.
→ It deals with the components of a connection in a system.	→ It acts as interface b/w hardware & Software.
→ It tells us how exactly all the units in the system are arranged and interconnected.	→ It helps us to understand the functionalities of a system.
→ It expresses the realization of architecture	→ A programmer can view architecture in terms of instructions, addressing modes & registers.
→ An organization is done on the basis of architecture	→ While designing a computer system architecture is considered first
→ It deals with low-level design issues	→ It deals with high-level design issues
→ Computer Organization involves physical components (Circuit design, Adders, Signals, Peripherals)	→ Computer Architecture involves logic (Instruction sets, Addressing modes, Data types, Cache optimization).

* Register transfer language:-

→ A digital computer system is an interconnection of digital modules such as registers, decoders, arithematic elements and control logic.

→ These digital modules are interconnected with some common data & control paths to form a complete digital system. Digital modules are best defined by the registers they contain & the operations that are performed on the data stored in them.

→ The operations performed on the data stored in registers are called micro-operations.

→ A micro-operations is an elementary operation performed on the information stored in 1 or more registers. The result of the operation may replace the previous binary information of a register or may be transferred to another register Examples of microoperations are shift, count, clear & load.

→ The internal hardware organization of a digital system is best defined by specifying.

↳ The set of registers and the flow of data between them.

↳ The sequence of microoperations performed on the binary information stored in the registers.

↳ The control that initiates the sequence of microoperations.

* Register Transfer language is the symbolic representation of notations used to specify the sequence of micro-operations.

* In a computer system, data transfer takes place between processor registers and memory and between processor registers & input-output systems. These data transfer can be represented by standard notations given below:

* Notations R0, R1, R2 ---, so on represent processor registers

* Address of memory locations are represented by names such as LOC, PLACE, MEM etc;

* Input-Output registers are represented by names such as DATA IN, DATA OUT and so on.

* The content of register or memory location is denoted by placing square brackets around the name of the register or memory location

*Register Transfer :-

→ The term Register transfer refers to the availability of hardware logic circuits that can perform a given micro-operation and transfer the result of operation to the same (or) another register.

→ The information transformed from one register to another register is represented in symbolic form by replacement operator is called Register transfer.

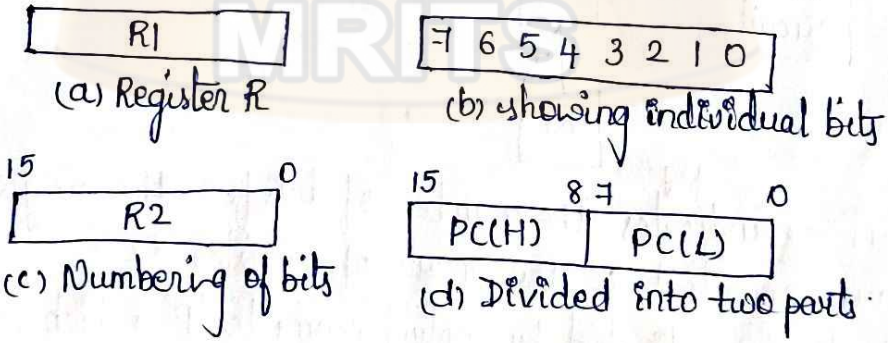
[Replacement Operator :-

→ In the statement $R_2 \leftarrow R_1$, "$\leftarrow$" acts as a replacement operator. This statement defines the transfer of content of register R_1 into register R_2.]

→ most of the standard notations used for specifying operations on various registers are stated below.

↳ The memory address register is designated by "MAR".
↳ Program counter "PC" holds the next instruction's address.
↳ Instruction register "IR" holds the instruction being executed.
↳ R_1 (processor register)

→ The individual flip-flops in an n-bit register are numbered in sequence from 0 through n-1 starting from 0 in the rightmost position & increasing the number toward the left.



(a) Register R (b) showing individual bits
(c) Numbering of bits (d) Divided into two parts.

fig: Block diagram of register.

→ The most common way to represent a register is by a rectangular box with the name of the register inside fig (a).
→ The individual bits can be distinguished in (b)
→ The numbering of bits in a 16-bit register can be marked on top of the box shown in (c)

→ A 16 bit register is partitioned into 2 parts in (d).
→ Bit 0 through 7 are assigned the symbol L (for Low byte) & bit 8 through 15 are assigned the symbol H (for high byte).
→ The name of 16 bit register is PC. The symbol PC(0-7) or PC(L) refers to low order byte & PC(8-15) or PC(H) to the high order byte.
→ Data transfer from one register to another register is represented in symbolic form by means of replacement operator. for instance the following statement denotes a transfer of data of register R_1 into register R_2.

$$R_2 \leftarrow R_1$$

→ Typically most of the users want the transfer to occur only in a Predetermined control condition. As it can be shown by following if-then statement:

If (P=1) then ($R_2 \leftarrow R_1$); Here P is a control signal generated in control section.

→ It is more convenient to specify a control function (P) by seperating the control variables from register transfer operation. for instance, the following statement defines the data transfer operation under a specific control function (P)

$$P: R_2 \leftarrow R_1$$

→ The following fig shows the Block diagram that depicts the transfer of data from R_1 to R_2.

Transfer from R_1 to R_2 when P=1:

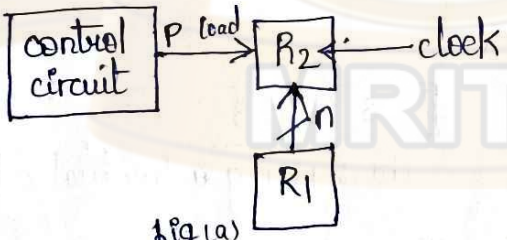


fig (a)

where "n" → indicates the number of bits for the register. The 'n' outputs of register R1 are connected to the inputs of register R2

Load input is activated by control variable 'P' which is transferred to register R_2.

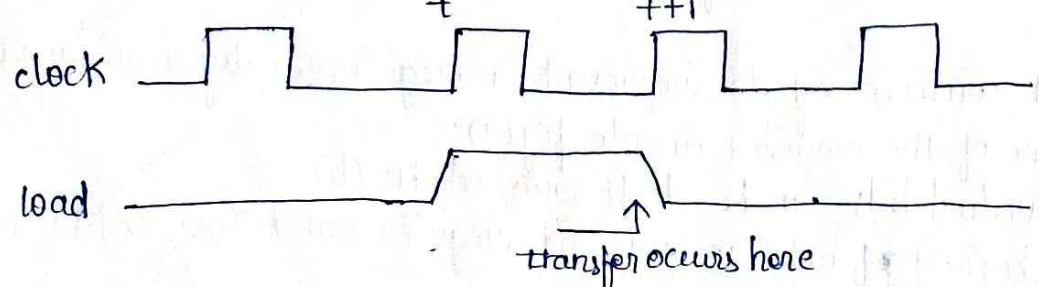


fig (b) Timing diagram

↳ In the timing diagram P is activated in control section by the raising edge of clock pulse at time t.

↳ The next +ve transition of clock at time t+1 finds the load i/p active & data i/p's of R_2 are then loaded into register in parallel.

↳ P may go back to 0 at time t+1; otherwise, tranfer will occur at every clock pulse transition while P remains active.

The basic symbols of register transfer notations are listed in Table ①

Symbol	Description	Examples
(1) Letters (Capital) and numerals	Denote a register	MAR, R2
(2) Parentheses ()	Denotes a part of a register	R2(0-7), R_2(L)
(3) Arrow ←	Denotes transfer of information	$R_2 \leftarrow R_1$
(4) Comma ,	Seperates two microoperations	$R_2 \leftarrow R_1, R_1 \leftarrow R_2$
:	termination of control function	$P: R_2 \leftarrow R_1$ if P=1

* BUS AND MEMORY TRANSFERS :-

*

→ A digital System composed of many registers, and paths must be provided to transfer information from one register to another. The no. of wires will be excessive if seperate lines are used between each register and all other register in the system.

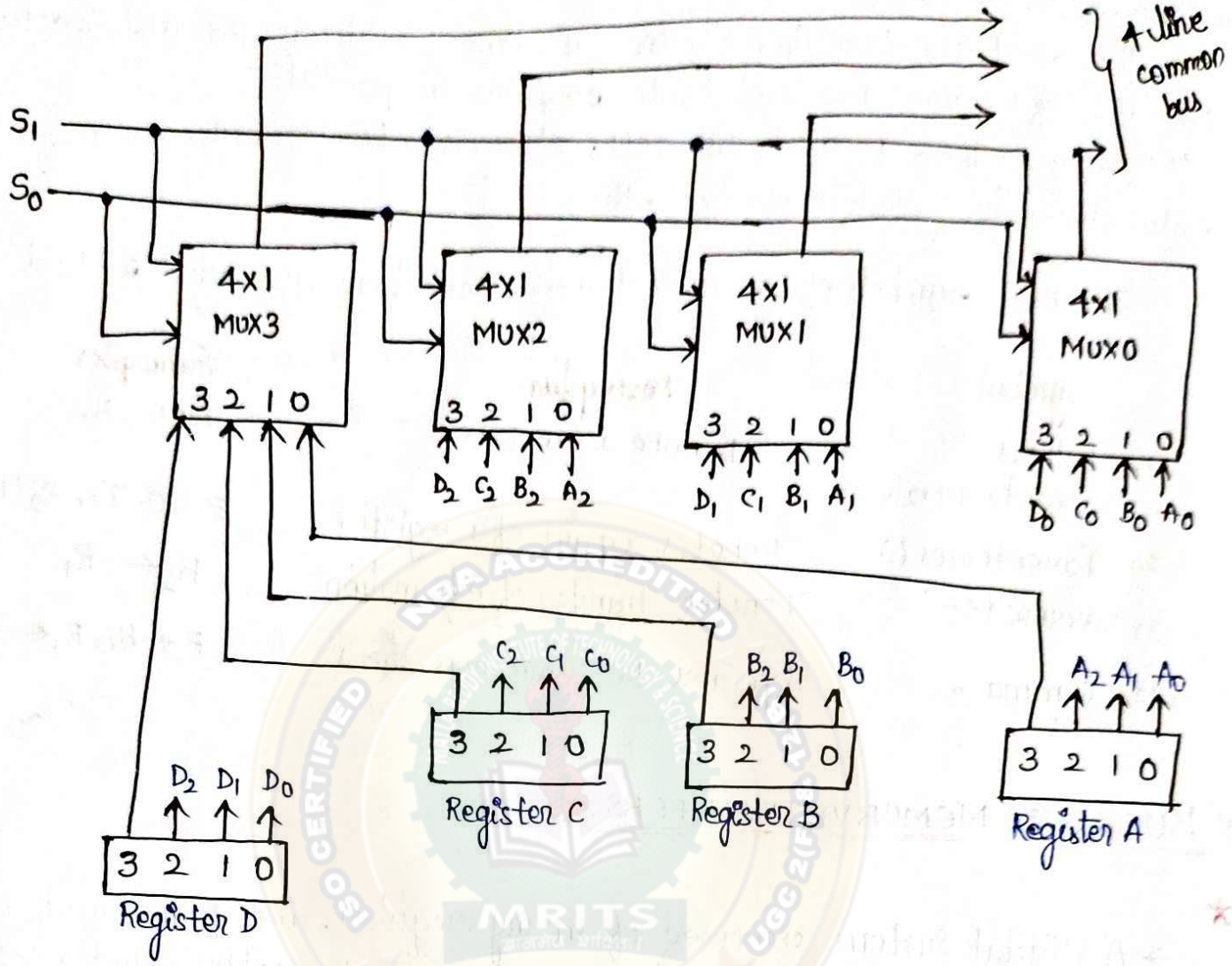
→ A bus structure, on the other hand, is more efficient for transferring information between registers in a multi-register configuration system.

→ A bus consists of a set of common lines, one for each bit of register, through which binary information is transferred one at a time. Control register determine which register is selected by the bus during a particular register transfer.

→ One way of constructing a common bus system is with multiplexers. The following block diagram shows a bus system for 4 registers. It is constructed with the help of four 4×1 Multiplexers each having four data i/p's (0 to 3) & two selection inputs (S_1 and S_2).

→ For instance output 1 of register A is connected to input 0 of MUX 1

Bus system for 4 registers :-



→ The two selection lines S_1 & S_0 are connected to the selected inputs of all 4 multiplexers.

→ The selection lines choose the four bits of 1 register & transfer them into the four common line bus.

→ When both of the select lines are at low logic, i.e; $S_1S_0 = 00$, the 0 data i/ps of all four multiplexers are selected and applied to the o/p's that forms the bus. This in turn, causes the bus lines to receive the content of register A since the output of this register are connected to the 0 data i/ps of the multiplexers

→ Similarly, when $S_1S_0 = 01$, register B is selected, and the bus lines will receive the content provided by register B.

→ The following function table shows the register that is selected by the bus for each of the 4 possible binary values of the selection lines

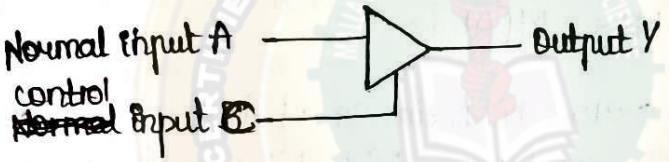
S1	S0	Register Selected
0	0	A
0	1	B
1	0	C
1	1	D

Three state Bus Buffers:-

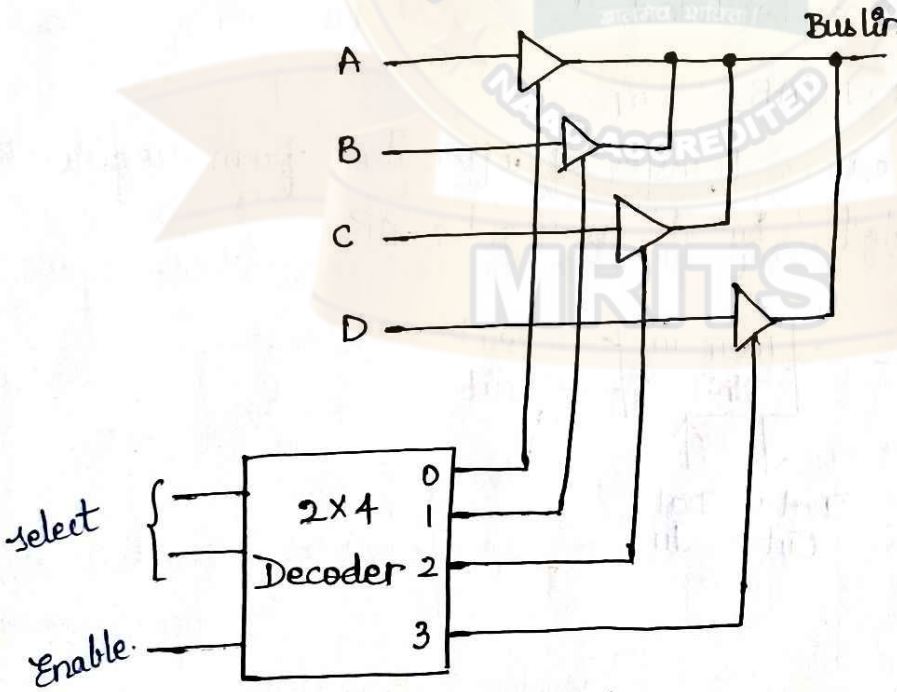
→ A bus system can also be constructed using 3 state gates instead of multiplexers

→ The three state gates can be considered as a digital circuit that has 3 gates, two of which are signals equivalent to logic 1 and 0 as in a conventional gate. However, the 3rd gate exhibits a high impedence state.

→ The most commonly used three state gates in case of the bus system is a buffer gate. The graphical symbol of a three state buffer gate can be represented as



Bus line with 3 state buffer:-



→ The o/p's generated by the 4 buffers are connected to form a single bus line.

→ Only one buffer can be active state at a given point of time.

→ The control i/p's to the buffers determine which of the four normal i/p's will communicate with bus line.

→ A 2x4 decoder ensures that no more than one control i/p is active at any given point of time.

* Memory Transfer:-

Most of the standard notations used for specifying operations on memory transfer are stated below.

→ The transfer of information from a memory unit to the end user is called Read operation.

→ The transfer of new information to be stored in the memory is called write operation

A memory word is designated by letter M.

It is necessary to specify the address of "M" when writing memory transfer operations. This will be done by enclosing the address in square brackets following the letter M.

The address register is designated by AR and the data register by DR.

Thus a read operation can be stated as

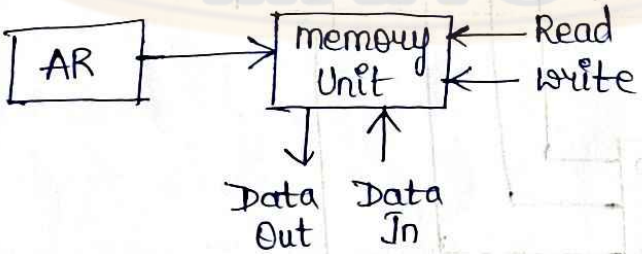
Read: $DR \leftarrow M[AR]$

The read statement causes a transfer of information into the data register (DR) from the memory word (M) selected by address register (AR).

The write operation can be stated as

Write: $M[AR] \leftarrow R_1$

The write statement causes transfer of information from register R_1 into the memory word (M) selected by address register (AR



Micro operations :-

A microoperation is an elementary operation performed with the data stored in registers. In digital computers, microoperations are classified into four categories :

1. Register transfer microopperations transfer binary information from one register to another.
2. Arithematic microoperations perform arithematic operation on numeric data stored in registers.
3. Logic microoperations perform bit manipulation operations on non-numeric data stored in registers.
4. Shift microoperations perform shift operations on data stored in registers.

* Arithematic microoperations :-

Arithematic microoperations deals with the operations performed on numeric data stored in the registers. The basic arithematic microoperations are classified as follows

- ↳ Addition
- ↳ Subtraction
- ↳ Increment
- ↳ decrement.
- ↳ shift.

some additional arithematic microoperations are

- ↳ Add with carry
- ↳ subtract with borrow
- ↳ Transfer/Load, etc.,

The following table shows the symbolic representation of Arithematic micro operations

symbolic designation	Description
$R_3 \leftarrow R_1 + R_2$	The contents of R1 plus R_2 transferred to R_3
$R_3 \leftarrow R_1 - R_2$	The contents of R_1 minus R_2 transferred to R_3
$R_2 \leftarrow \overline{R_2}$	Complement the contents of R_2.

$R_2 \leftarrow \overline{R_2} + 1$	2's complement the contents of R_2
$R_3 \leftarrow R_1 + \overline{R_2} + 1$	R_1 plus the 2's complement of R_2
$R_1 \leftarrow R_1 + 1$	Increment the contents of R_1 by one
$R_1 \leftarrow R_1 - 1$	Decrement the contents of R_1 by one.

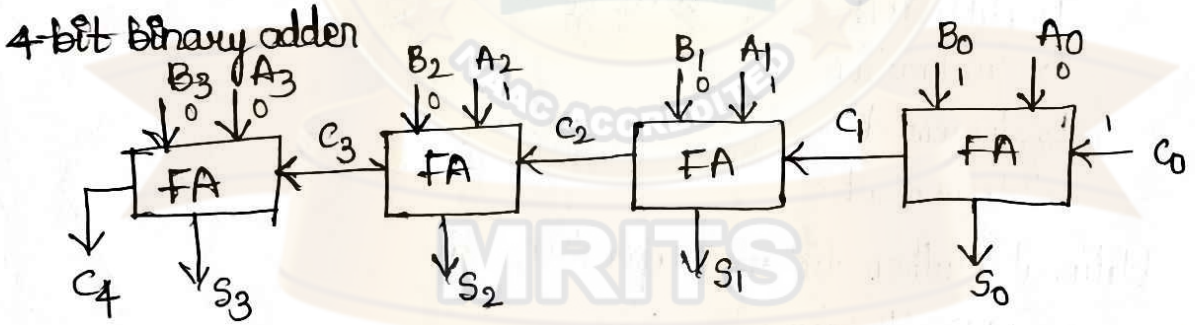
* Binary Adder:-

→ The add microoperation requires registers that can hold the data and the digital components that can perform arithematic addition.

→ A binary adder is a digital circuit that performs the arithematic sum of two binary numbers provided with any length.

→ A Binary Adder is constructed using full-adder circuits connected in series, with the output carry from one full-adder connected to i/p carry of next full-adder.

→ The block diagram shows the interconnections of four full-adder circuits to provide a 4-bit binary adder.



→ The augend bits (A) and addend bits (B) are designated by subscript numbers from right to left, with subscript '0' denoting the low-order bit.

→ The carry i/p's starts from C_0 to C_3 connected in a chain thru the full adders. C_4 is resultant output carry generated by last F.A.

→ The o/p carry from each full-adder is connected to the i/p carry of next-high order F.A.

→ The sum o/p's (S_0 to S_3) generates the required arithematic sum of augend & addend bits.

→ The n data bits for the A & B i/p's come from different source registers. For instance, data bits for A i/p comes from source register R_1 and data bits for B i/p comes from source register ~~(i/p or R_2)~~ R_2.

→ The arithematic sum of the data i/p's of A and B can be transferred to third register or to one of the registers (R_1 or R_2)

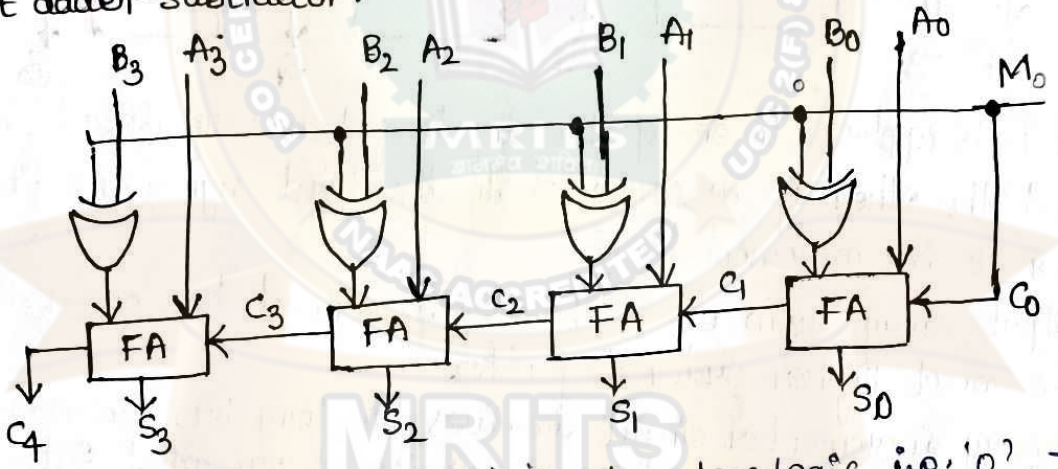
★ Binary Adder-Subtractor:-

→ The subtraction micro operation can be done easily by taking 2's complement of addend bits and adding it to the augend bits

→ The arithematic micro operations like addition and subtraction can be combined into one common circuit by including an exclusive-OR gate with each full adder.

→ The block diagram for a 4-bit adder-subtractor circuit can be represented as

4 bit adder-subtractor:



→ When the mode input (M) is at a low logic i.e; '0', the circuit act as an adder & when the mode i/p is at high logic, i.e; '1', the circuit act as a subtractor.

→ The Exclusive-OR gate connected in series receives i/p M and one of i/p B.

→ When M is at a low logic, we have $B \oplus 0 = B$. The full-adders receive the value of B, the i/p carry is 0 and the circuit performs A+B

→ When M is at high logic, we have $B \oplus 1 = \overline{B}$ and $C0 = 1$

→ The B i/ps are complemented and a 1 is added through the i/p carry. The circuit performs the operation A plus the 2's complement of B.

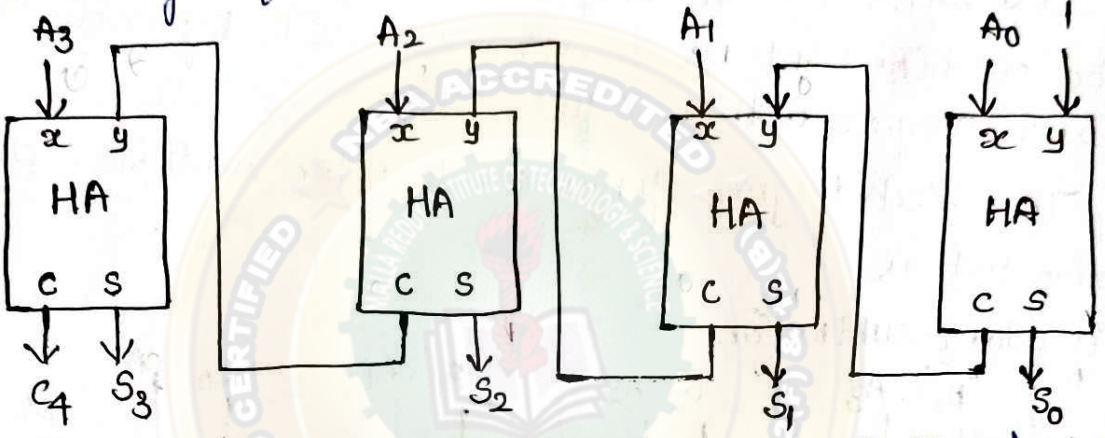
★ Binary Incrementer :-

→ The increment microoperation adds one binary value to the value of binary variables stored in a register.

→ For instance, a 4-bit register has a binary value 0110, when incremented by one - the value becomes 0111.

$$\begin{array}{r} 0110 \\ +\quad\ 1 \\ \hline 0111 \end{array}$$

→ The increment micro-operation is best implemented by a 4-bit combinational circuit incrementer. A 4-bit combinational circuit incrementer can be represented by the following block diagram.



→ A logic-1 is applied to one of the inputs of least significant half-adder, and the other i/p is connected to the least significant bit of the number to be incremented.

→ The output carry from one half-adder is connected to one of the i/p's of the next higher-order half adder.

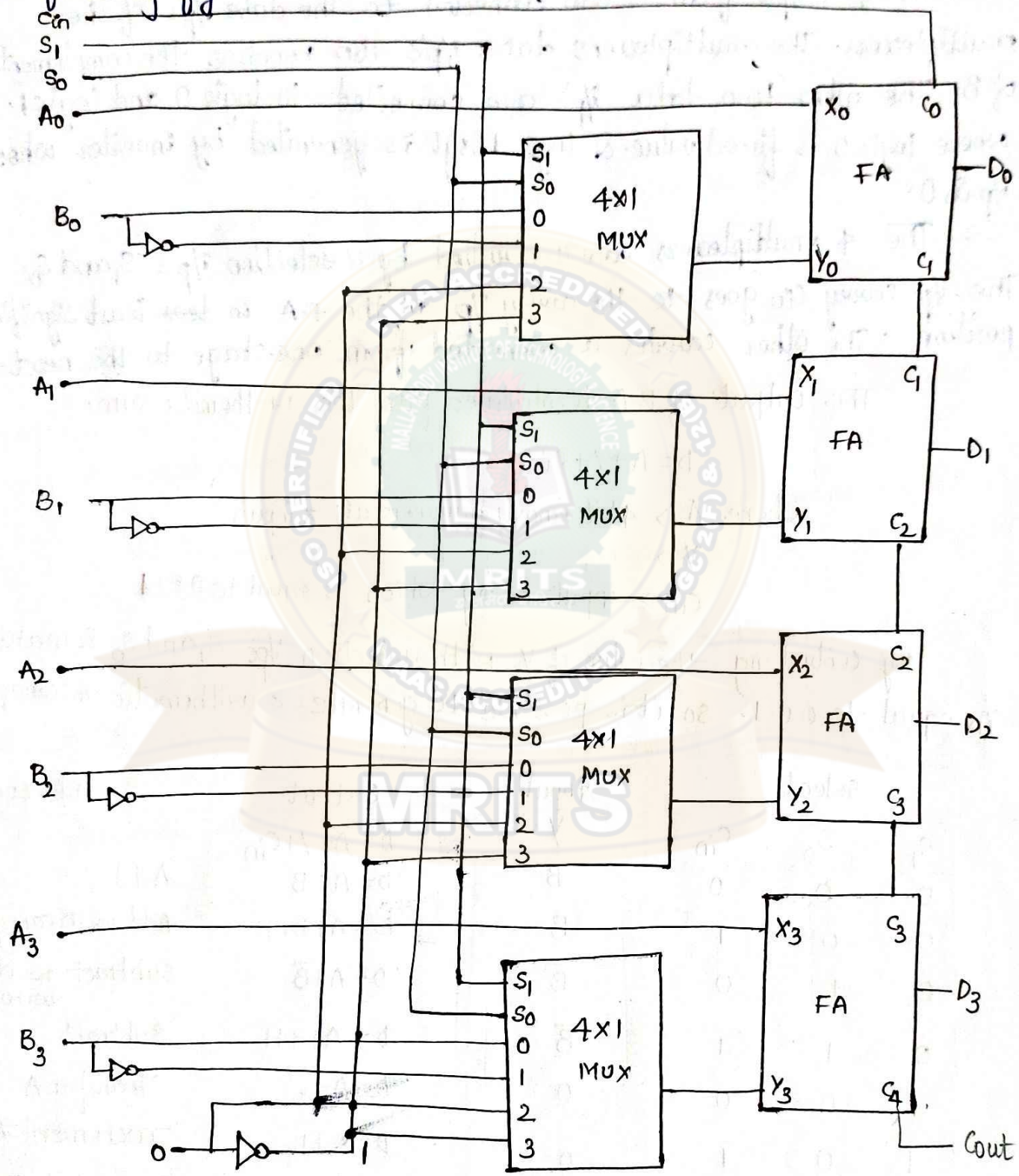
→ The binary incrementer circuit receives the four bits from A_0 through A_3, adds one to it, and generates the incremented output in S_0 through S_3.

→ The output carry C_4 will be 1 only after incrementing binary 1111.

→ Arithmetic Circuit

* Arithematic Circuit :-

The basic component of an arithematic circuit is parallel adder. By controlling the data i/p's to the adder, it is possible to obtain different types of arithematic operations. The diagram of a 4-bit arithematic circuit is shown in the following fig.



→ The block diagram has 4 full adder circuits that constitute the 4 bit adder and four multiplexers for choosing different operations.

→ There are 2 4-bit i/p's A and B and a 4-bit output D. The four i/p's from A go directly to the X i/p's of binary adder.

→ Each of 4 i/p's from B are connected to the data i/p's of the multiplexers. The multiplexers data i/p's also receive the complement of B. The other two data i/p's are connected to logic 0 and logic 1. where logic 0 is fixed value. & logic 1 sgl is generated by inverter whose i/p is 0.

→ The 4 multiplexers are controlled by 2 selection i/p's S_1 and S_0. The i/p carry C_{in} goes to the carry i/p of the FA in least Significant position. The other carries a connected from one stage to the next.

The output of B·A is calculated from the arithematic sum.

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

where A → 4 bit binary numbers at x inputs

Y → " " " " Y "

C_{in} → input carry which is equal to 0 or 1

By controlling the value of Y with selection i/ps S_1 and S_0 & making C_{in} equal to 0 or 1. So it is possible to generate 8 arithematic microoperations

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

Addition:-

→ When $S_1S_0 = 00$ the value B is applied to Y i/ps of the adder.

→ If $C_{in} = 0$ then output $D = A + B$

1 then output $D = A + B + 1$

→ In Both cases Add microoperation with or without adding carry are performed.

Subtraction :-

→ When $S_1S_0 = 01$, the complement of B is applied to Y i/p's of the adder.

→ If $C_{in} = 1$ then o/p $D = A + \overline{B} + 1$ (A + 2's comp of B is equivalent to A-B)

0 then o/p $D = A + \overline{B}$ (This equivalent to subtract with borrow i.e; A-B-1

→ When $S_1S_0 = 10$, the i/p's from B are neglected instead all 0's are inserted into Y i/p's. The o/p becomes $D = A + 0 + C_{in}$

$C_{in} = 0$ then $D = A$ (here direct transfer from i/p A to o/p D

1 then $D = A + 1$ (the value of A is incremented by 1

→ When $S_1S_0 = 11$, all 1's are inserted into Y i/ps of adder to produce the decrement operation ~~D = A - 1 when~~

$D = A - 1$ when $C_{in} = 0$,

This is because a number with all 1's equal to 2's complement of 1 (i.e; 2's comp of binary 0001 is 1111). Adding a number A to 2's comp of 1 produces

$F = A + 2$'s comp of $1 = A - 1$

↳ When $C_{in} = 1$ then $D = A - 1 + 1 = A$, which causes a direct transfer from i/p A to o/p D.

∴ D = A is generated twice, so there are only 7 distinct microoperation in Arithematic circuit.

* Logic microoperations :-

Logic microoperations specify binary operations for strings of bits stored in registers. As operations consider each bit of the register seperately and treat them as binary variables.

Eg:- The Exclusive OR microoperation with the contents of two registers R_1 and R_2 is symbolized by the statement

$$P: R_1 \leftarrow R_1 \oplus R_2$$

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

The Exclusive OR microoperation stated above symbolizes the following logic computation

```
1010    → content of R1
1100    → content of R2
----
0110    → content of R1 after P=1
```

The content of R_1 after the execution of microoperation, is equal to the bit-by-bit Exclusive OR operation on pairs of bits in R_2 & previous values of R_1.

→ The logic microoperations are widely used in scientific computations, but they are very useful for bit manipulation of binary data & for making logical decisions.

Special Symbols.

special symbols will be adopted for the logic microoperations OR, AND and complement, to distinguish them from corresponding symbols used to express boolean functions.

→ The symbol $\vee$ will be used to denote an OR microoperation.

→ The symbol $\wedge$ will be used to denote an AND microoperation.

→ The complement microoperation is same as 1's complement & uses a bar on top of the symbol that denotes the register name.

→ By using different symbols, it will be possible to differentiate b/w a logic microoperation & a control function (Boolean).

→ Another reason for adopting 2 sets of symbols is to be able to distinguish the symbol +, when used to symbolize an arithematic plus, from a logic OR operation.

→ As + symbol has 2 meanings, it will be possible to distinguish b/w them by nothing where the symbol occurs. When the symbol + occurs in a microoperation, it will denote an arithematic plus. When it occurs in a control (or boolean) function, it will denote an OR operation. We will never use it to symbolize an OR microoperation.

For example, $P+Q: R_1 \leftarrow R_2+R_3, R_4 \leftarrow R_5 \vee R_6$ the + b/w P & Q is an OR operation b/w two binary variables of a control function. The + b/w R_2 & R_3 specifies an add microoperation. The OR microoperation is designated by symbol V b/w R_5 & R_6

List of Logic Microoperations:-

There are 16 different logic operations that can be performed with 2 binary variables. They can be determined from all possible truth tables obtained with two binary variables as shown in Table below.

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

In this table, each of 16 columns F_0 to F_{15} represents a T.T of one possible Boolean function for the 2 variables x and y. The functions are determined from 16 binary combinations that can be assigned to F. The 16 boolean functions of 2 variable x & y are expressed in algebraic form in first column of table 2. and logic microoperations in second column using A & B registers

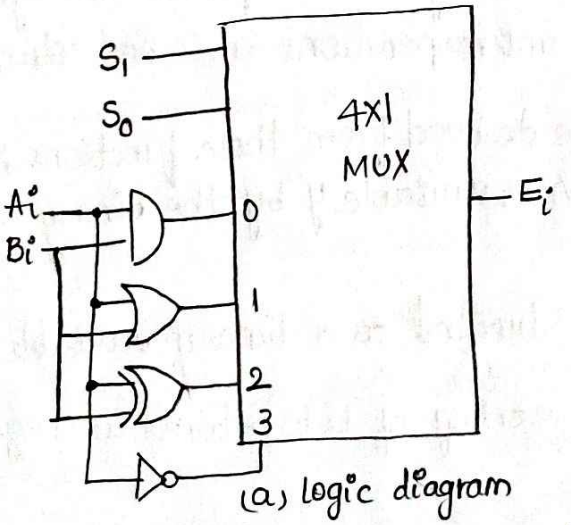
The 16 logic microoperations are derived from these functions by replacing x by the binary content of register A & variable y by the binary content of register B.

Each bit of register is treated as a binary variable & the microoperation is performed on the string of bits stored in registers.

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

1 Hardware Implementation :-

The hardware implementation of logic microoperations requires that logic gates be inserted for each bit or pair of bits in the registers to perform the required logic function.



(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	$E = \bar{A}$	Complement

(b) functional table

fig: One stage of logic circuit.

→ Although there are 16 logic microoperations, most computers use only 4 — AND, OR, XOR (Exclusive-OR), and complement from which all others can be derived.

→ Figure shows one stage of a circuit that generates the 4 basic logic microoperations. It consists of 4 gates & a multiplexer. Each of the 4 logic operations is generated through a gate that performs the required logic.

→ The o/p of the gates are applied to the data i/ps of the multiplexer. The two selection i/p S_1 and S_0 choose one of the data i/ps of multiplexer & direct its value to the o/p.

→ The diagram shows one typical stage with subscript i. for a logic circuit with n bits, the diagram must be repeated n times for $i = 0, 1, 2 \cdots n-1$.

→ The selection variables are applied to all stages. The function table in Fig. lists the logic microoperations obtained for each combination of selection variables.

→ Some applications :-

↳ Logic microoperations are very useful for manipulating individual bits or a portion of word stored in a register.

↳ They can be used to change bit values, delete a group of bits, or insert new bit values into a register.

↳ An typical application, register A is processor register and register B constitute logic operand extracted from memory & placed in reg B.

★ Selective Set :

The selective set operation sets to 1 the bits in reg A where there are corresponding 1's in register B. It doesnot effect bit position that have 0's in B.

for example.

```
  1 0 1 0  → A before
  1 1 0 0  → B (logic operand)
-----------
  1 1 1 0  → A after
```

↳ The 2 leftmost bits of B are 1's, so the corresponding bits of A are set to 1

↳ The bits of A after the operation are obtained from the logic-OR operation of bits in B and previous values of A.

★ Selective complement :-

The selective complement operation complements bits in A where there are selective clear corresponding 1's in B. It doesnot affect bit positions that have 0's in B

for example

```
  1 0 1 0  → A before
  1 1 0 0  → B (logic Operand)
-----------
  0 1 1 0  → A after
```

→ The 2 leftmost bits of B are 1's, so the corresponding bits of A are complemented.
→ The selective complement operation is just an X-OR microoperation.

★ Selective clear:

The selective clear operation clears to 0 the bits in A only where the corresponding 1's in B.

for example:

```
1010   A before
1100   B (logic Operand)
----
0010   A after.
```

→ The leftmost bits of B are 1's, so the corresponding bits of A are cleared to 0.
→ The boolean operation performed on individual bits is AB'. The corresponding logic microoperation is $A \leftarrow A \wedge B'$.

★ Mask Operation:

It is similar to selective clear except that the bits of A are cleared only where there are corresponding 0's in B.

for example,

```
1010   A before
1100   B (logic Operand)
----
1000   A (after masking)
```

The 2 rightmost bits of A are cleared because the corresponding B bits are 0's.

The mask operation is an AND microoperation and it is more convenient than selective clear.

★ Insert:-

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

for example, suppose that an A register contains 8 bits 0110 1010

To replace the 4 leftmost bits by value 1001 we first mask the four unwanted bits:

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

and the insert new value

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

The mask operation is an AND microoperation & insert operation is an OR micro-operation.

* Clear Operation :-

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

```
1010   A
1010   B
----
0000   A ← A⊕B
```

When A & B are equal, the & corresponding bits are either both 0 or both 1. In either case the exclusive-OR operation produces a 0. The all-0's result is then checked to determine if the two numbers were equal.

* SHIFT MICROOPERATIONS :-

Shift microoperations are used for serial transfer of data. They are also used in conjunction with arithematic, logic and other data processing operations.

The contents of a register can be shifted to the left or the right. At the same time the bits are shifted, the first flip-flop receives its binary information from the serial input.

→ During a shift left operation, the serial i/p transfers a bit into the right-most position.

→ During a shift-right operation, the serial i/p transfers a bit into the left most position.

The information transferred through the serial input determines the type of shift. There are 3 types of shifts: logical, circular and arithematic

A logical shift is one that transfers 0 through serial i/p. We will adopt the symbols shl and shr for logical shift left and shift right microoperations.

for example

$R_1 \leftarrow$ shl R1

$R_2 \leftarrow$ shl R2

are 2 microoperations that specify a 1 bit shift to left of the content of register R1 and a 1-bit shift to the right of the content of Register R_2. The register symbol must be same on both sides of the arrow.

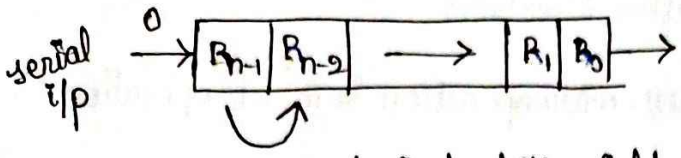


fig: logical shift right

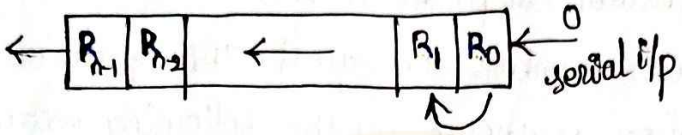


fig: logical shift left.

The bit transferred to the end position through the serial i/p is assumed to be 0 during a logical shift.

Circular shift circulates the bits of the register around the ends without loss of any information. In the case of logical shift, one of the end bits is lost.

Circular shift or rotate operation is performed by connecting the least significant bit to the ~~least~~ most significant bit position. The symbolic representation is shown in fig

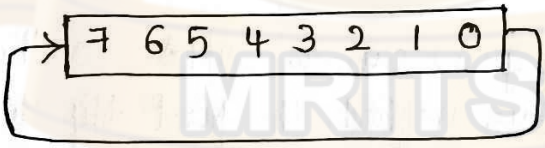


fig:- Rotate left

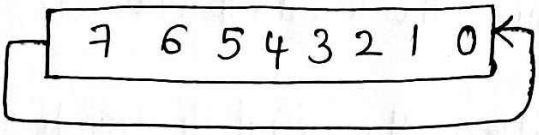


fig:- Rotate right.

we will use the symbols cil and cir for the circular shift left and right.

The symbolic notation for the shift microoperations is shown in below table

symbolic designation	Description
R ← shl R	shift left register R
R ← shr R	shift right register R
R ← cil R	circular left shift left register R
R ← cir R	circular shift right register R
R ← ashl R	arithematic shift left R
R ← ashr R	arithematic shift right R

An arithematic shift is a microoperation that shifts a signed binary number to the left or right. An arithematic shift left multiplies a signed binary number by 2. An arithematic shift right divides the number by 2.

arithematic shift right must leave the sign bit unchanged because the sign of the number remains same.

0 1 0 0 ④
0 0 1 0 ②

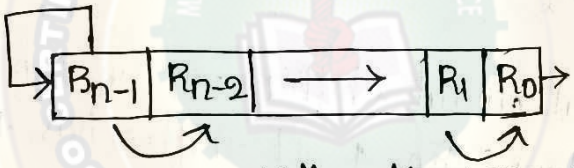


fig: arithematic shift right

This requirement on right shifting distinguish arithematic shifts from logical shifts. Otherwise 2 shift operations are very similar.

The arithematic shift left is same as logic shift left

0 0 1 0 → ②
0 1 0 0 → ④

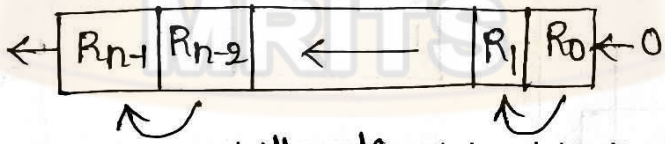


fig: arithematic shift left (ASL)

ASL inserts 0 into R_0 & shifts all other bits to the left. Initial bit of R_{n-1} is lost & replaced by the bit from R_{n-2}. A sign reversal occurs if R_{n-1} changes in value after the shift. This happens if multiplication by 2 causes an overflow. Overflow occurs after an ASL if initially, before the shift $R_{n-1} \neq R_{n-2}$

An overflow flipflop V_S, can be used to detect an ASL overflow

$$V_S = R_{n-1} \oplus R_{n-2}$$

If $V_S = 0$ there is no overflow, but if $V_S = 1$ there is an overflow & a sign reversal after the shift. V_S must be transferred into overflow flipflop with the same clockpulse that shifts the register

* Hardware Implementation :-

→ A possible choice for a shift unit would be a bidirectional shift register with parallel load

→ Information can be transferred to the register in parallel and then shifted to the right or left.

→ In this type of configuration a clock pulse is needed for loading the data into the register and another pulse is needed to initiate the shift.

→ In a processor unit with many registers it is more efficient to implement the shift operations with a combinational circuit.

→ In this way the content of a register that has to be shifted is first placed onto a common bus whose o/p is connected to combinational shifter, & the shifted number is then loaded back into the register

→ It requires only 1 clock pulse for loading the shifted value into the register.

→ A combinational circuit shifter can be constructed with multiplexers as shown in fig. below. The 4-bit shifter has 4 data i/p's, A_0 to A_3 and 4 data o/p's, H_0 to H_3.

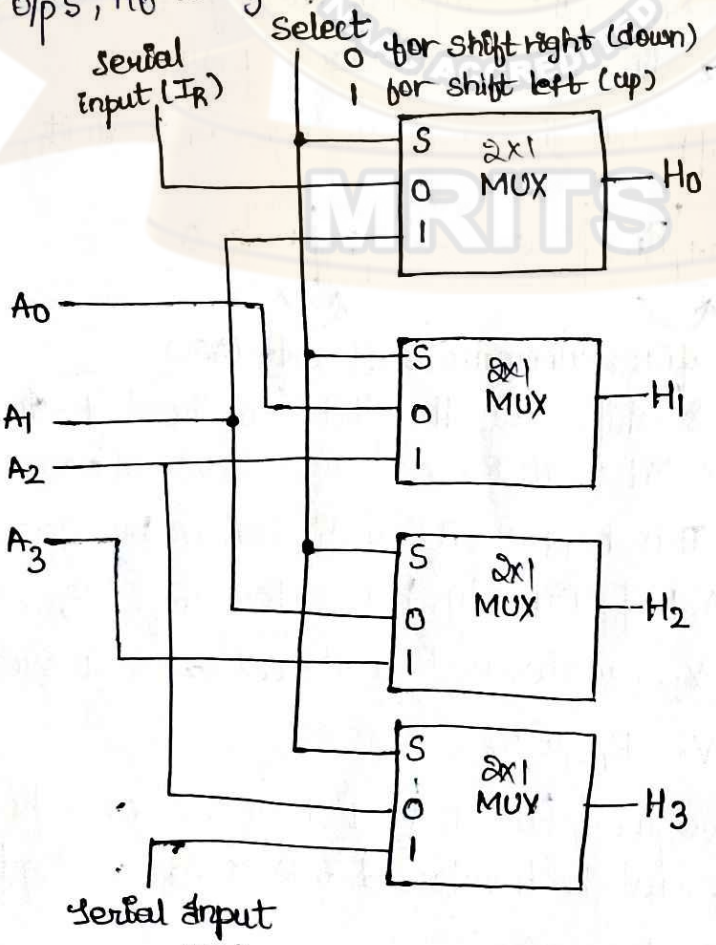


Fig: 4-bit combinational circuit shifter

Functional Table

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

There are two serial i/p's, one for shift left (I_L) and other for shift right (I_R). when the selection i/p S=0, the i/p data are shifted right.
when S=1, the i/p data are shifted left.

A shifter with n data i/p's & o/p's requires n multiplexers. The two serial i/p's can be controlled by another multiplexer to provide the 3 possible types of shifts.

* ARITHMETIC LOGIC SHIFT UNIT:

→ Instead of having individual registers performing the microoperations directly, computer systems employ a number of storage registers connected to a common operational unit called an arithematic logic unit (ALU).

→ To perform a microoperation, the contents of specified registers are placed in the i/p's of the common ALU.

→ The ALU performs an operation & the result is transferred to destination register.

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

→ The shift microoperations are often performed in a seperate unit, but sometimes the shift unit is made part of the ALU.

→ The arithematic, logic and shift circuits ~~introduced~~ in can be combined into one ALU with common selection variables.

→ One stage of an Arithematic logic shift unit shown in below fig. The subscript i designates a typical stage.

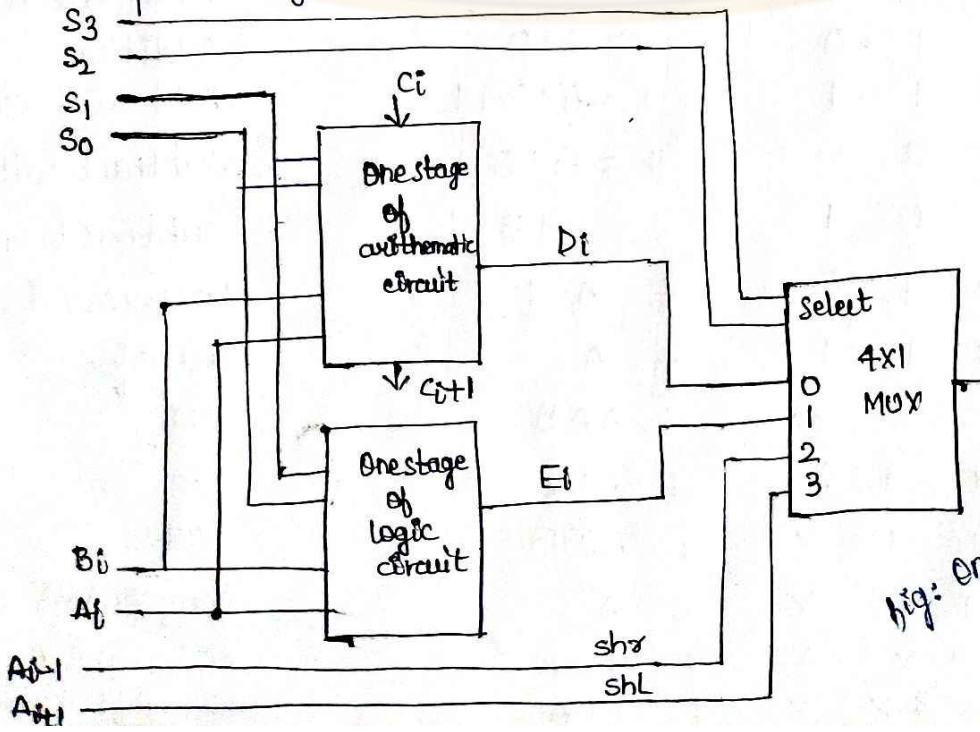


fig: One stage of arithematic logic shift unit

→ Inputs A_i and B_i are applied to both the arithematic & logic units.
→ A particular microoperation is selected with i/p's S_1 and S_0. A 4x1 multiplexer at the o/p chooses b/w an arithematic output E_i and a logic o/p in H_i
→ The data in the multiplexer are selected with i/p's S_3 and S_2.
→ The other 2 data i/p's to the multiplexer receive A_{i-1} for the shift-right operation and A_{i+1} for the shift left operation.
→ The circuit must be repeated n times for an n-bit ALU. The Output carry C_{i+1} of a given arithematic stage must be connected to the input carry C_i of next stage in sequence. The i/p carry to 1st stage is the i/p carry C_{in} which provides a selection variable for the arithematic operations
→ The circuit provides eight arithematic operation, four logic operations, and two shift operations
→ Each operation is selected with 5 variables S_3, S_2, S_1, S_0 and C_{in}
→ The i/p carry C_{in} is used for selecting an arithematic operation only.
→ The first 8 are arithematic operations and are selected with $S_3S_2=00$.
→ The next 4 are logic " " " " " $S_3S_2=01$
→ The i/p carry has no effect during logic operations & marked as don't care x's.
→ The last 2 are shift operations and are selected with $S_3S_2=10$ and 11
→ The other 3 selection i/ps have no effect on shift.

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
0	0	1	1	1	$F=A$	Transfer
0	1	0	0	X	$F=A \wedge B$	AND
0	1	0	1	X	$F=A \vee B$	OR
0	1	1	0	X	$F=A \oplus B$	XOR
0	1	1	1	X	$F=\overline{A}$	complement A
1	0	X	X	X	$F=shr\ A$	shift right A into F
1	1	X	X	X	$F=shl\ A$	shift left A into F

Basic Computer Organization & design

→ The organization of the computer is defined by its internal registers timing & control structures and the set of instructions it uses.

→ The internal organization of a digital system is defined by the sequence of microoperations it performs on data stored in its registers.

→ The general purpose computer is capable of executing various microoperations & can be instructed as to what specific sequence of operations it must perform. The user of a computer can control the process by means of a program.

→ A program is a set of instructions that specify the operations operands & the sequence by which processing has to occur. The dataprocessing task may be altered by specifying a new program with different instructions or specifying the same instructions with different data.

→ A computer instruction is a binary code that specifies a sequence of microoperations for the computer. Instruction code together with data are stored in memory. The computer reads each instruction from memory and places it in a control register.

→ The control then interprets the binary code of instruction & proceeds to execute it by issuing a sequence of microoperations. The ability to store and execute instructions, the stored program concept is the most important property of general purpose computer.

Instruction Codes:-

→ An instruction code is a group of bits that instruct the computer to perform a specific operation.

→ It is usually divided into parts, each having its own interpretation The most basic part of an instruction code is its operation part. The operation code of an instruction is a group of bits that defines such operations as Add, subtract, multiply, shift & complement.

→ The no. of bits required for the operation code of an instruction depends on the total no. of operations available in the computer.

→ The operation part of an instruction code specifies the operation to be performed. An instruction code must be there to specify not only the operation but also the registers & memory words.

Stored Program Organization :-

→ The simplest way to organize a computer is to have one process register and an instruction code format with two parts: Operation & Address.

→ The first part specifies the operation to be performed and second specifies an address.

→ The memory address tells the control where to find an operand in memory.

→ This operand is read from memory and used as the data to be operated on together with the data stored in the processor register

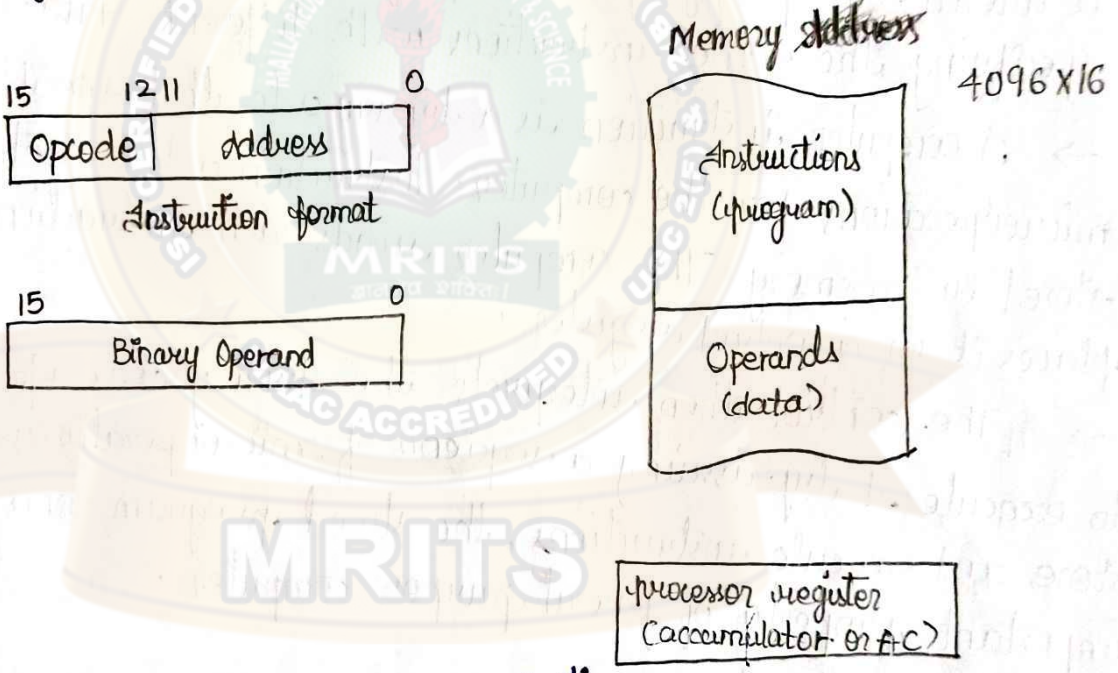


fig: stored program organization

→ Figure depicts this type of organization. Instructions are stored in one section of memory and data in another. For a memory unit with 4096 words. we need 12 bits to specify an address since $2^{12} = 4096$. If we store each instruction code in one 16-bit memory word, we have available four bits for the operation code to specify one out of 16 possible operations and 12 bits to specify the address of an operand.

→ The control reads a 16-bit instruction from the program portion of memory. It uses the 12-bit address part of the instruction to read a 16-bit operand from the data portion of memory. It then executes the operation specified by operation code. computers have a single processor register called Accumulator (AC)

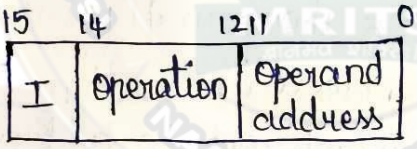
Indirect Address:-

→ It is sometimes convenient to use the address bit of an instruction code not as an address but the actual operand. When the second part of an instruction is said to have an immediate operand.

→ When the second part specifies the address of an operand that instruction is said to have a direct address. This is in contrast to a third possibility called "Indirect address", where the bits in the second part of the instruction designate an address of a memory word in which the address of the operand is formed.

→ One bit of the instruction code can be used to distinguish between a direct & indirect address (MOD bit)

Consider the instruction format as



fig(a): Instruction format.

→ It consists of 3 bit operation code, a 12-bit address and a mode bit designated by I.

If mod bit is 0 ⇒ direct address

If mod bit is 1 ⇒ Indirect address.

A direct address instruction is shown in fig.

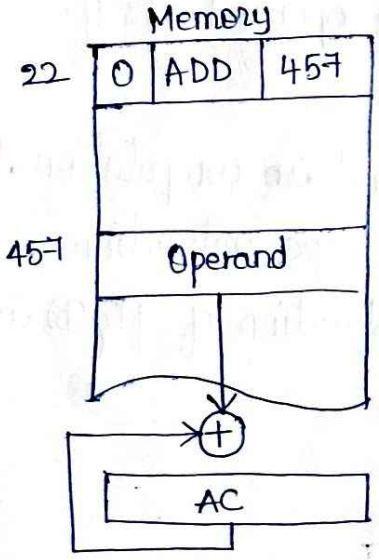


fig: Direct address. (b)

→ As it is placed in address 22 in memory. The I bit is 0, so the instruction is recognized as a direct address instruction. The Opcode specifies an ADD instruction & address part is the binary equivalent of 457. The control finds the operand in memory at address 457 and adds it to the content of AC.

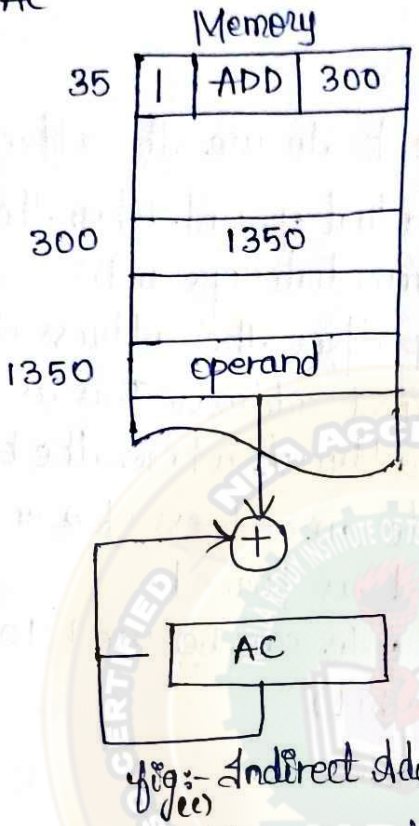


fig:- Indirect Address.

→ The instruction in address 35 shown in fig has a mode bit I=1. Therefore it is recognized as indirect address instruction. The address part is the binary equivalent of 300. The control goes to address 300 to find the address of the operand.

→ The address of the operand in this case is 1350. The operand found in address 1350 is then added to the content of AC.

→ The indirect Address instruction needs 2 references to memory to fetch an operand. The first reference is needed to read the address of the operand, the second is for the operand itself.

Effective address:-

The address of the operand in computation type instruction or the target address in a branch type instruction

The effective address in instruction of fig(b) is 457 & the instruction of fig(b) is 1350.

* Computer Registers :-

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

→ The control reads an instruction from a specific address in memory and executes it. It then continues by reading the next instruction in sequence & executes it & so on.

→ This type of instruction sequencing needs a counter to calculate the address of the next instruction after execution of the current instruction is completed.

→ It is also necessary to provide a register in the control unit for storing the instruction code after it is read from memory.

→ The computer needs processor register for manipulates data and a register for holding a memory address. These requirements dictate the register configuration shown in below fig

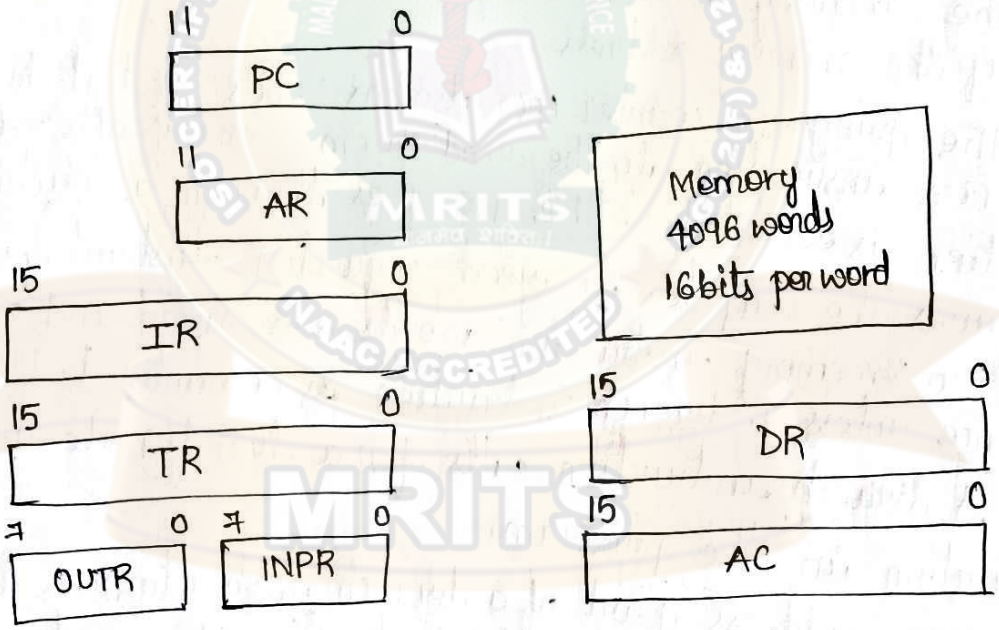


fig:- Basic computer registers and memory

Table:

Register Symbol	No. of bits	Register Name	Function
DR	16	Data Register	Holds memory operand
AR	12	Address register	Holds address for memory
AC	16	Accumulator	Processor registers
IR	16	Instruction register	Holds instruction code
PC	12	Program Counter	Holds address of instruction
TR	16	Temporary register	Holds temporary data
INPR	8	Input register	Holds Input character
OUTR	8	Output register	Holds Output character

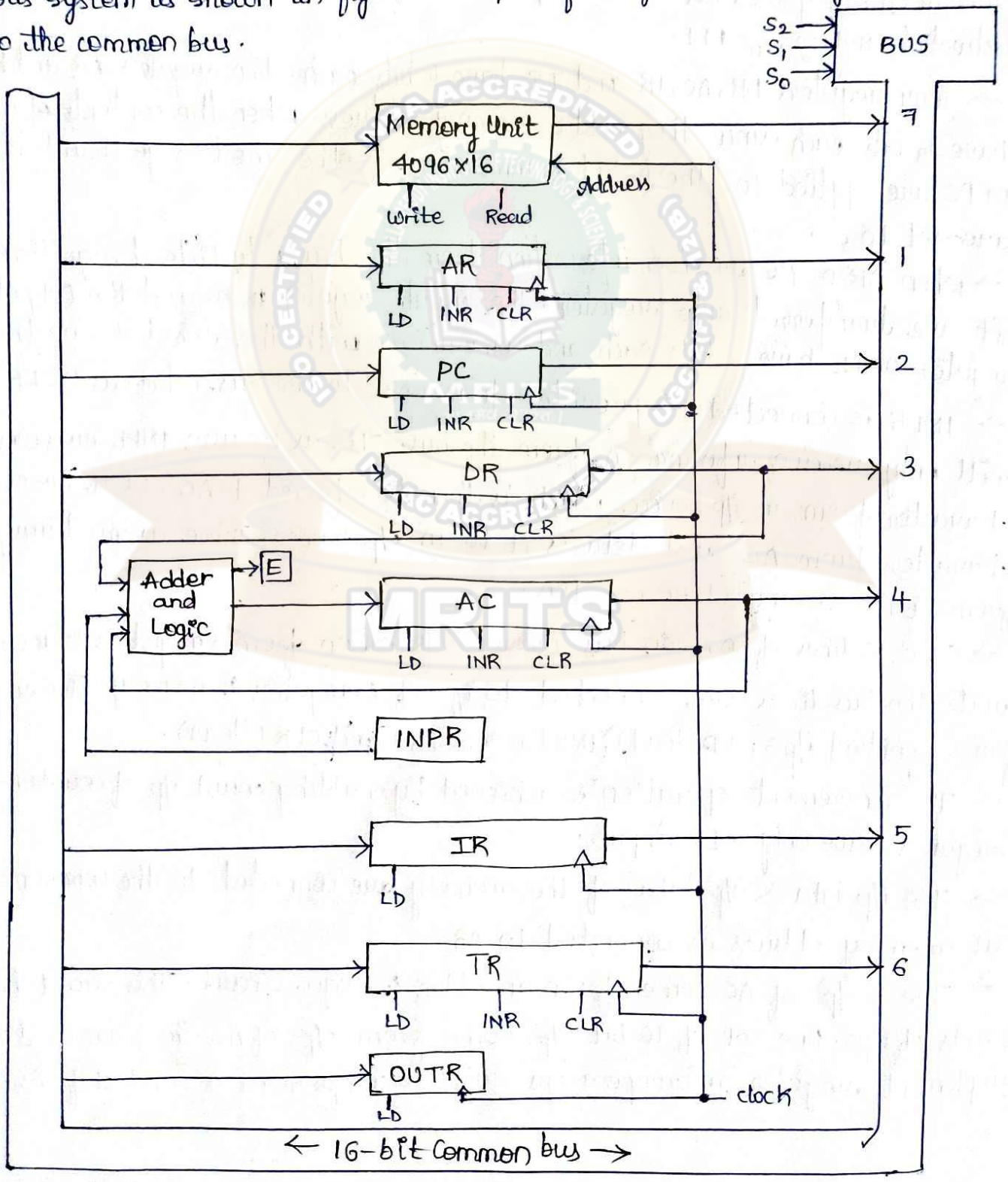
→ The register are listed in the table together with a brief description of their function and the number of bits that they contain.
→ The memory unit has a capacity of 4096 words and each word contains 16 bits. 12 bits of an instruction word are needed to specify the address of an operand.
→ This leaves 3 bits for the operation part of the instruction and a bit to specify a direct or indirect address.
↳ The data register (DR) holds the operand read from memory.
↳ The Accumulator (AC) register is a general purpose processing register.
↳ The instruction read from memory is placed in instruction register (IR).
↳ The temporary register (TR) is used for holding temporary data during the processing.
↳ The memory address register (AR) has 12 bits since this is the width of the memory address.
↳ The program counter (PC) also has 12 bits and it holds the address of the next instruction to be read from memory after the current instruction is executed. The PC goes through a counting sequence and causes the computer to read sequential instructions previously stored in memory instruction words are read and executed in sequence unless a branch instruction is encountered.
→ A branch instruction calls for a transfer to a nonconsecutive instruction in the program.
→ The address part of a branch instruction is transferred to PC to become the address of next instruction. To read an instruction, the content of PC is taken as the address for memory & a memory read cycle is initiated. PC is incremented by one, so it holds the address of the next instruction in sequence.
→ Two registers are used for input and output. The input register (INPR) receives an 8-bit character from an input device. The output register (OUTR) holds an 8-bit character for an output device.

★ Common bus system:-

The basic computer has 8 registers, a memory unit and a control unit. Paths must be provided to transfer information from one register to another and between memory and registers.

The number of wires will be excessive if connections are made b/w the o/p's of each register & the i/p's of other registers. So a more efficient scheme is a common bus system which is employed in 4-bit common bus system using multiplexers & three-state buffers.

The connection of registers & memory of the basic computer to a common bus system is shown in fig. The outputs of 7 registers & memory are connected to the common bus.



→ The specific o/p that is selected for the bus lines at any given time is determined from the binary value of the selection variable S_2, S_1 and S_0.

→ The number along each o/p shows the decimal equivalent of the required binary selection. For example, the number along the output of DR is 3.

→ The 16-bit o/p's of DR are placed on the bus lines when $S_2S_1S_0 = 011$. Since the binary value of decimal 3.

→ The lines of from the common bus are connected to the i/p's of each register and the data i/p's of memory. The particular register whose LD(load) i/p is enabled receives the data from the bus during the next clock pulse transition.

→ The memory receives the contents of the bus when its write i/p is activated. The memory places its 16-bit outputs onto the bus when the ~~i/p~~ read i/p is activated and $S_2S_1S_0 = 111$.

→ Four registers DR, AC, IR and TR, have 16 bits each. Two registers AR and PC, have 12 bits each since they hold a memory address. When the contents of AR or PC are applied to the 16 bit common bus, ~~but~~ the 4 most significant bits are set to 0's.

→ When AR or PC receive information from the bus, only 12 least significant bit are transferred into register. The input register INPR and the output register OUTR have 8 bits each and communicate with the 8 LSB's in the bus.

→ INPR is connected to provide information to the bus. whereas OUTR will only receive information from the bus. This is because INPR receives a character from an i/p device which is then transferred to AC. OUTR receives a character from AC and delivers it to an o/p device. There is no transfer from OUTR to any other registers.

→ The 16 lines of common bus receive information from 6 registers & memory unit. The bus lines are connected to i/ps of 6 registers & memory. Five register have 3 control i/p's: LD(load), INR (increment) and CLR (clear).

→ The increment operation is achieved by enabling count i/p of counter. 2 registers have only a LD input.

→ The i/p data & o/p data of the memory are connected to the common bus but memory address is connected to AR.

→ The 16 i/p's of AC come from an adder & logic circuit. This circuit has 3 sets of i/p's. One set of 16-bit i/p's come from o/p's of AC. They are used to implement register microoperations such as complement AC and shift AC.

→ Another set of 16-bit inputs come from the DR. The i/p's from DR to AC are used for arithematic & logic microoperations, such as add DR to AC or AND DR to AC. The result of addition is transferred to AC and end carry out of the addition is transferred to flip flop E.

→ A third set of 8-bit i/p's come from input register INPR.

For example:- Consider two micro operations

DR ← AC and AC ← DR

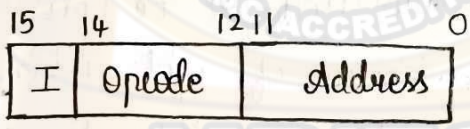
→ The two microoperations can be executed at the same time. This can be done by placing the content of AC on the bus, enabling the LD i/p of DR, transferring the content of DR through adder & logic circuit into AC and enabling the LD (load) i/p of AC, all during same clock cycle.

→ The 2 transfers occur upon the arrival of the clock pulse transition at the end of the clock cycle.

* Computer Instructions:-

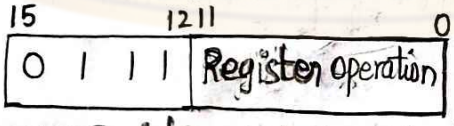
The basic computer has three instruction code formats shown in fig. Each format has 16 bits.

The Operation code (Opcode) part of the instruction contains three bits and the meaning of the remaining 13 bits depends on the operation code encountered.



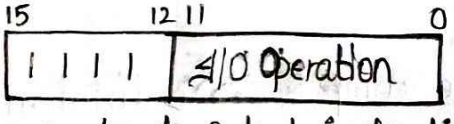
(Opcode = 000 through 110)

(a) Memory-reference instruction



(Opcode = 111, I = 0)

(b) Register-reference instruction



(Opcode = 111, I = 1)

(C) Input-Output instruction

fig: Basic Computer instruction formats

A memory-reference instruction uses 12 bits to specify an address and one bit to specify an address and one bit to specify the addressing mode I.

⇒ I is equal to 0 for direct address
I = 1 for indirect address.

→ The register reference instructions are recognized by the operation code 111 with a 0 in leftmost bit (bit 15) of the instruction. A register reference instruction specifies an operation on or a test of the AC register. An operand from memory is not needed; therefore, the other 12 bits are used to specify the operation or test to be executed.

→ An Input-Output instruction does not need a reference to memory and is recognized by the operation code 111 with a 1 in the leftmost bit of the instruction. The remaining 12 bits are used to specify the type of input-output operation or test performed.

→ The type of instruction is recognized by the computer control from the 4 bits in positions 12 through 15 of the instruction. If the 3 opcode bits in positions 12 through 14 are not equal to 111, the ~~bit~~ in ~~pos~~ instruction is a memory-reference type and the bit in position 15 is taken as the addressing mode I.

→ If 3 bit Opcode is equal to 111, control then inspects the bit in position 15.

If bit = 0 then register reference type
bit = 1 then Input-Output type.

→ Only 3 bits of instruction are used for the Opcode. It seems that the computer is restricted to maximum of eight distinct operations.

→ However, since register-reference and input-output instructions use the remaining 12 bits as part of operation code, the total no. of instructions can exceed eight.

→ In fact, the total number of instructions chosen for the basic computer is equal to 25.

→ The instructions for the computer are listed in Table. The symbol designation is a three-letter word and represents an abbreviation intended for programmers & users. The hexadecimal code is equal to the equivalent hexadecimal number of the binary code used for the instruction.

By using the hexadecimal equivalent we reduce the 16 bits of an instruction code to 4 digits with each hexadecimal digit being equivalent to 4 bits.

Memory reference instructions:-

Symbol	Hexadecimal Code		Description
	I=0	I=1	
AND	0XXX	8XXX	AND memory word to AC
ADD	1XXX	9XXX	Add memory word to AC
LDA	2XXX	AXXX	Load memory word to AC
STA	3XXX	BXXX	Store content of AC in memory
BUN	4XXX	CXXX	Branch unconditionally
BSA	5XXX	DXXX	Branch & save return address
ISZ	6XXX	EXXX	Increment and skip if zero

Register reference instructions :-

Symbol	Hexadecimal Code	Description
CLA	7800	clear AC
CLE	7400	clear E
CMA	7200	complement AC
CME	7100	complement E
CIR	7080	circulate right AC and E
CIL	7040	circulate left AC and E
INC	7020	Increment AC
SPA	7010	skip next instruction if AC positive
SNA	7008	skip next instruction if AC negative
SZA	7004	skip next instruction if AC zero
SZE	7002	skip next instruction if E is 0
HLT	7001	Halt computer

Input-Output instruction:-

Symbol	Hexadecimal code	Description
INP	F800	Input character to AC
OUT	F400	Output character from AC
SKI	F200	Skip on input flag
SKO	F100	Skip on output flag
ION	F080	Interrupt On
IOF	F040	Interrupt Off

A memory reference instruction has an address part of 12 bits. The address part is denoted by three X's and stand for the three hexadecimal digits corresponding to the 12-bit address.

The last bit of instruction is designated by I

when I=0, the last 4 bits of an instruction have a hexadecimal digit equivalent from 0 to 6 since the last bit is 0.

when I=1, the hexadecimal digit equivalent of the last four bits of the instruction ranges from 8 to E since the last bit is 1.

*Instruction Set Completeness:-

→ A computer should have a set of instructions so that the user can construct machine language programs to evaluate any function that is known to be computable.

→ The set of instructions are said to be complete if the ~~complete~~ computer includes a sufficient number of instructions in each of the following categories

① Arithmetic, logical and shift instructions.

② Instructions for moving information to and from memory and process-or registers.

③ Program Control instruction together with instructions that check status conditions.

④ Input and Output instructions.

→ Arithmetic, logical and shift instructions provide computational capabilities for processing the type of data that the user may wish to employ.

The bulk of the binary information in a digital computer is stored in memory, but all computations are done in processor registers. So, the user must have the capability of moving information between these two units.

Program control instructions such as branch instructions are used to change the sequence in which the program is executed.

Input and Output instructions are needed for communication between the computer & user.

* Timing and Control :-

The timing for all registers in the basic computer is controlled by a master clock generator.

The clock pulses are applied to all flipflops and registers in the system, including the flipflops and registers in the control unit.

The clock pulses do not change the state of a register unless the register is enabled by a control signal.

The control signals are generated in the control unit and provide control i/p's for the multiplexers in the common bus, control inputs in processor registers and microoperations for the accumulator.

There are two major types of control organization:

(1) Hardwired control

(2) Microprogrammed control.

In hardwired organization, the control logic is implemented with gates, flipflops, decoders and other digital circuits. It has advantage that it can be optimized to produce a fast mode of operation.

In microprogrammed organization, the control information is stored in a control memory. The control memory is programmed to initiate the required sequence of microoperations.

A hardwired control as the name implies, requires changes in the wiring among the various components if the design has to be modified or changed. In the microprogrammed control, any required changes or modifications can be done by updating the microprogram in control memory.

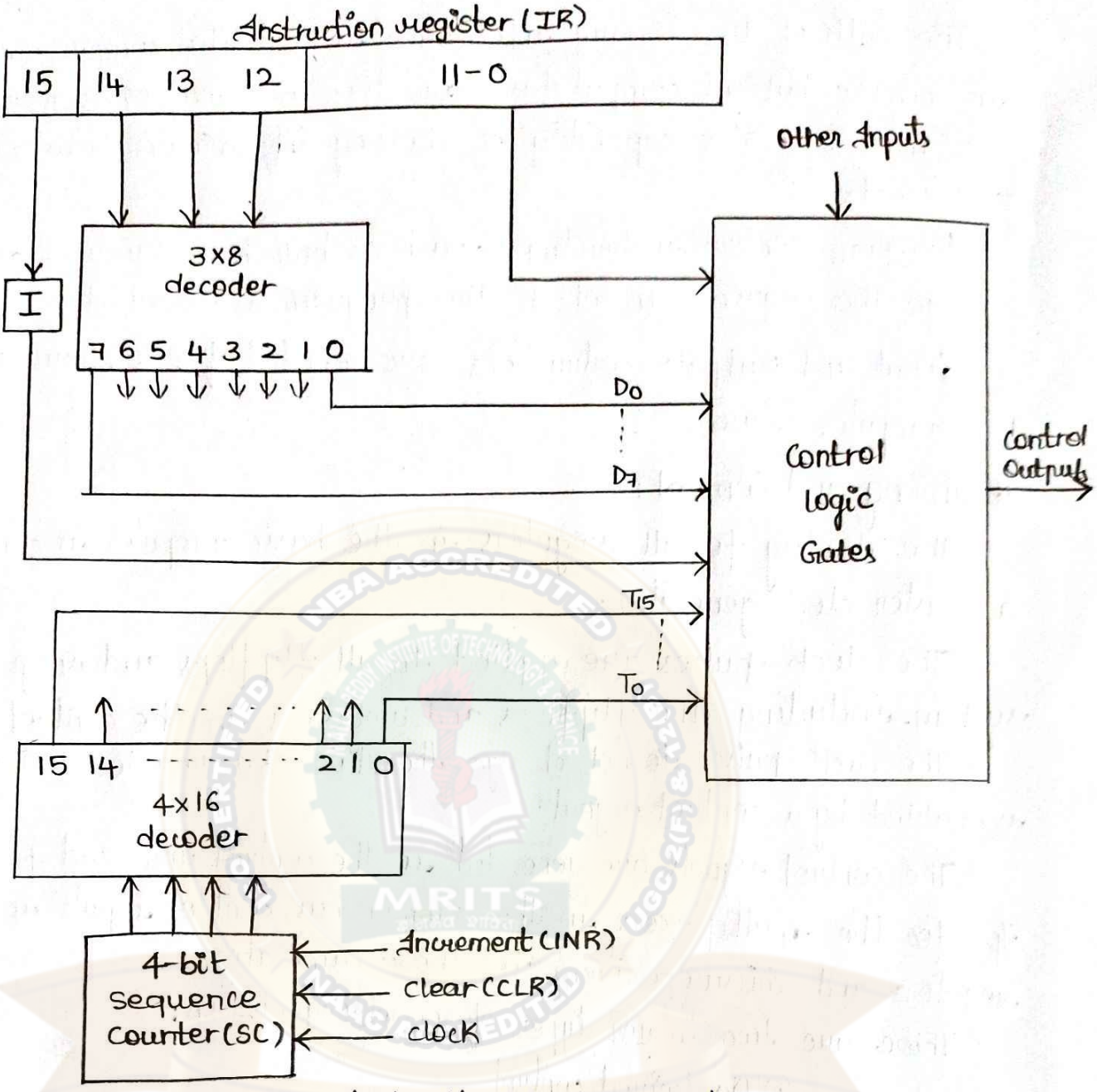


fig: Control unit of basic computer

→ The block diagram of the control unit is shown in fig. It consists of two decoders, a sequence counter and a number of control logic gates. An instruction read from memory is placed in the instruction register (IR).

→ The operation code in bits 12 through 14 are decoded with a 3x8 decoder. The eight outputs of the decoder are designated by symbols D_0 through D7. The subscripted decimal number is equivalent to the binary value of the corresponding operation code. Bit 15 of the instruction is transferred to a flipflop designated by the symbol I.

→ Bits 0 through 11 are applied to the control logic gates. The 4-bit sequence counter can count in binary from 0 through 15. The output of counter are decoded into 16 timing signals T0 through T15.

The sequence counter SC can be incremented or cleared synchronously.

Most of the time, the counter is incremented to provide the sequence of timing signals out of the 4x16 decoder. Once in awhile, the counter is cleared to 0, causing the next active timing signal to be T0.

For example, consider the case where SC is incremented to provide timing signals T_0, T_1, T_2, T_3, T_4 in sequence. At time T_4, SC is cleared to 0, if decoder output D_3 is active. This is symbolically expressed as

$$D_3T_4 : SC \leftarrow 0$$

The timing diagram of figure shows the time relationship of the control signals.

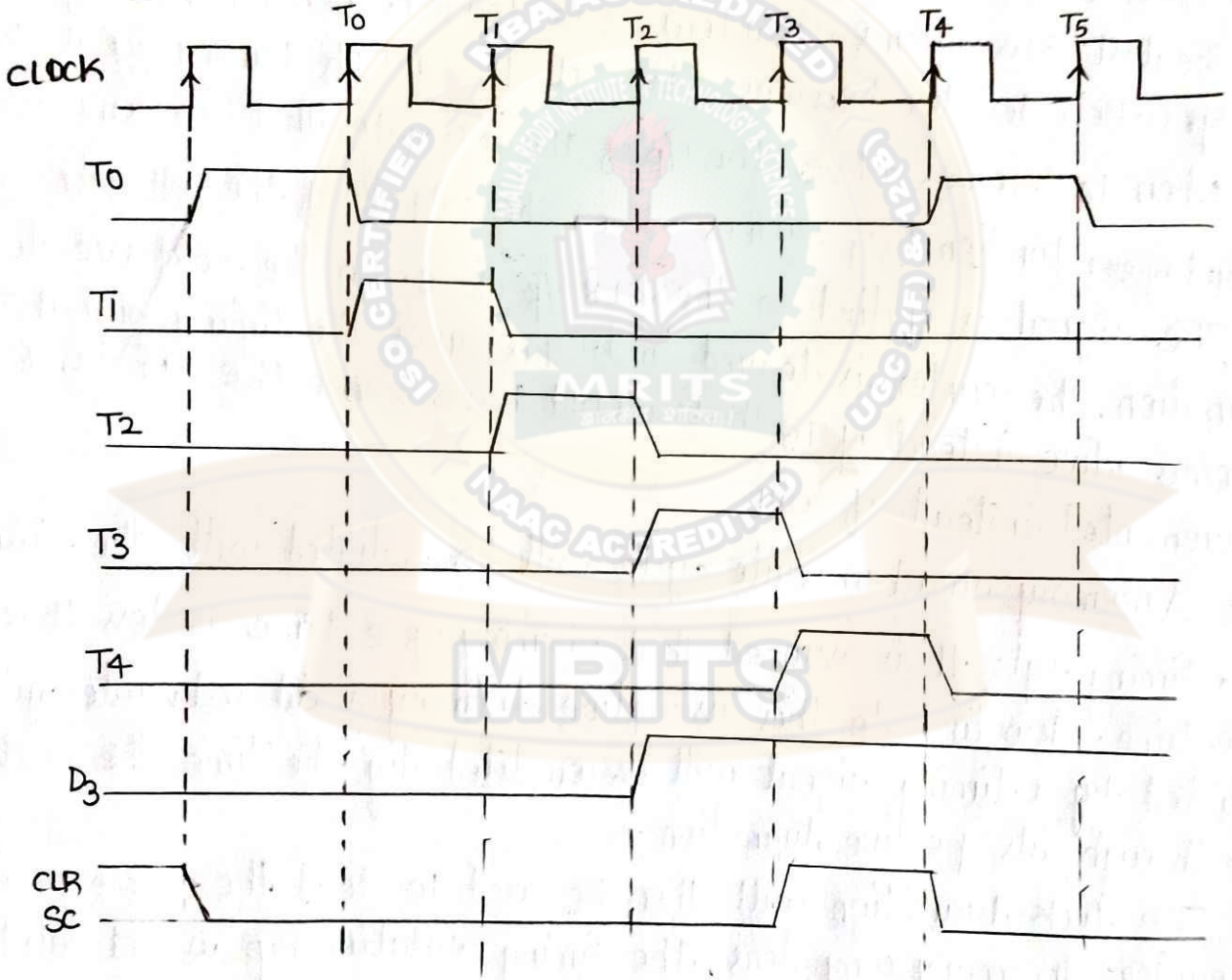


Fig:- Example of control timing signals

→ The sequence counter SC responds to the positive transition of the clock. Initially, the CLR input of SC is active. The first positive transition clock clears SC to 0, which in turn activates the timing signal T0 out of the decoder. T0 is active during one clock cycle.

→ The +ve clock transition labeled T0 in the diagram will trigger only those registers whose control i/p's are connected to timing signal T0.

→ SC is incremented with every positive clock unless its CLR input is active. This produces the sequence of timing signals T_0, T_1, T_2, T_3, T_4 and so on, as shown in diagram.

→ If SC is not cleared, the timing signals will continue with T_5, T_6 upto T_{15} and back to T_0.

→ The last 3 waveforms show how SC is cleared when $D_3T_4 = 1$. O/p D_3 from the operation decoder becomes active at the end of timing signal T2.

→ When T_4 becomes active, the o/p of the AND gate that implements the control ~~opera~~ function D_3T_4 becomes active, ~~the output of the AND gate~~

→ The signal is applied to the CLR i/p of SC. On the next +ve clock transition, the counter is cleared to 0. This causes the timing signal T0 to become active instead of T_5 that would have been active if SC were incremented instead of clear.

→ A memory read or write cycle will be initiated with the rising edge of a timing sgl. It is assumed that memory cycle time is less than clock cycle time. According to this assumption, memory read and write cycle initiated by a timing signal will be completed by the time the next clock goes through its positive transition.

→ The clock transition will then be used to load the memory word into a register. In many computers, the timing relationship is not valid because the memory cycle time is usually longer than processor clock cycle. So in this case, it is necessary to provide wait cycles in the processor until the memory word is available.

→ To fully comprehend the operation of the computer, it is crucial that one understands the timing relationship b/w the clock transition and timing signals.

For example, the register transfer statement

T_0: AR ← PC (specifies a transfer of the content of PC into AR if timing signal T_0 is active)

T_0 is active during an entire clock cycle interval. During this time the content of PC is placed onto the bus (with $S_2S_1S_0 = 010$) and load (LD) input of AR is enabled. The actual transfer does not occur until the end of the clock cycle when the clock goes through a +ve transition.

The same +ve clock transition increments the sequence counter SC from 0000 to 0001. The next clock cycle has T_1 active and T_0 inactive

* Instruction Cycle:-

A program residing in the memory unit of the computer consists of a sequence of instructions. The program is executed in the computer by going through a cycle for each instruction.

Each instruction cycle in turn is subdivided into a sequence of subcycles or phases. In the basic computer each instruction cycle consists of following phases:

1. Fetch an instruction from memory.
2. Decode the instruction
3. Read the effective address from memory address.
4. Execute the instruction.

Upon the execution of step 4, the control goes back to step 1 to fetch, decode and execute the next instruction. This process continues indefinitely unless a HALT instruction is encountered.

Fetch and decode:-

1. Initially, the program counter PC is loaded with the address of the first instruction in the program.
2. The sequence counter SC is cleared to 0 providing a decoded timing signal T_0. After each clock pulse, SC is incremented by one, so that the timing signals go through a sequence T_0, T_1, T_2 and so on.

③ The microoperations for the fetch and decode phases can be specified by the following register transfer statements

T_0 : AR ← PC

T_1 : IR ← M[AR], PC ← PC+1

T_2 : D_0 ..... D_7 ← Decode IR(12-14), AR ← IR(0-11), I ← IR(15)

④ Since only AR is connected to the address inputs of memory, it is necessary to transfer the address from PC to AR during the clock transition associated with timing signal T_0. The instruction read from memory is then placed in the instruction register IR with the clock transition associated with timing signal T_1.

⑤ At the same time, PC is incremented by 1 to prepare it for the address of the next instruction in the program. At time T_2, the operation code in IR is decoded, the indirect bit is transferred to flip-flop I and address part of instruction is transferred to AR.

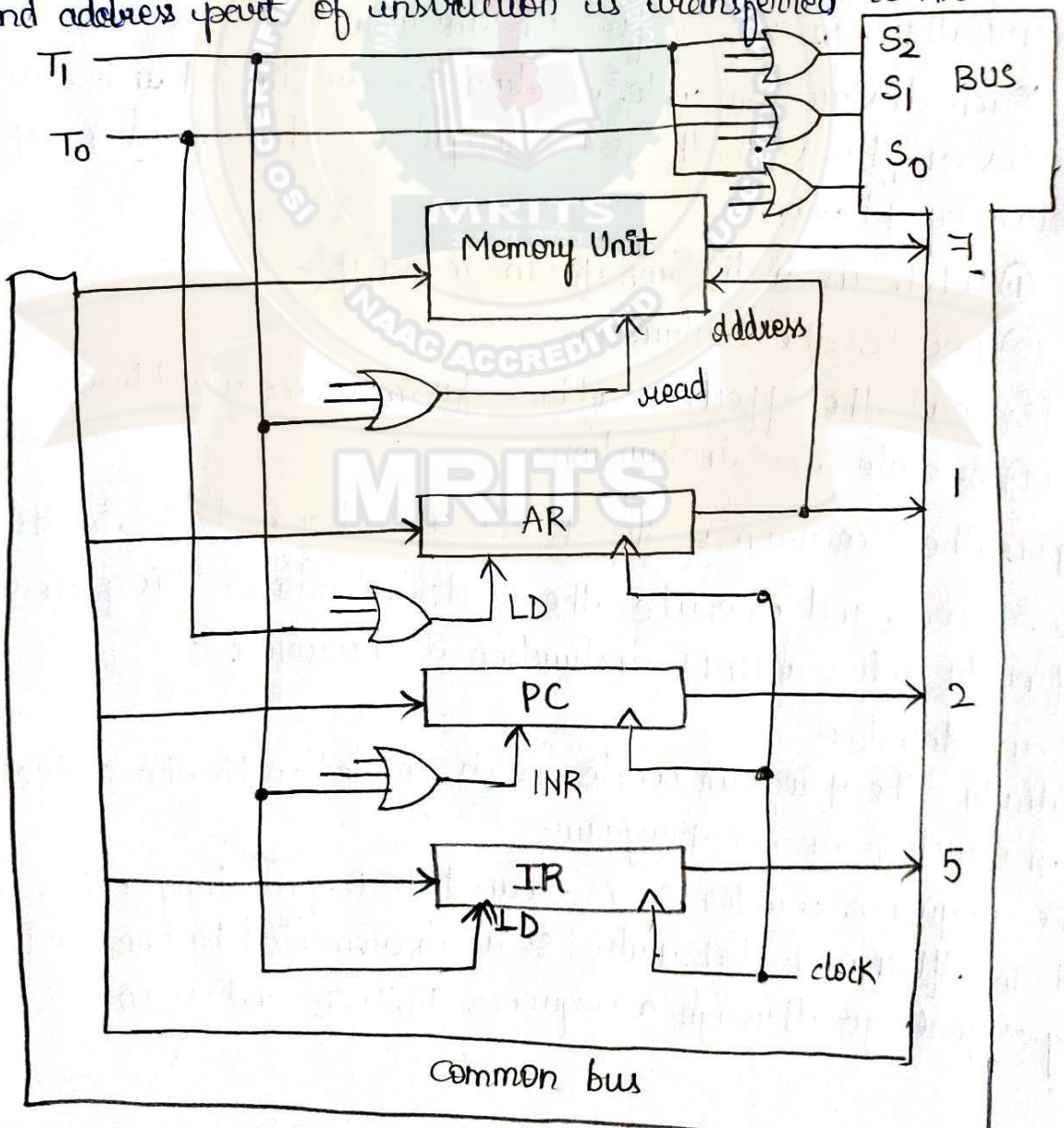


fig: Register transfers for fetch phase

If SC is incremented after each clock pulse to produce the sequence T_0, T_1 and T_2. Fig shows how the first 2 register transfer statements are implemented in bus system.

To provide the data path for the transfer of PC to AR we must apply timing signal T_0 to achieve the following connection :

1. Place the content of PC onto the bus by making bus selection i/p's $S_2S_1S_0 = 010$.
2. Transfer the content of the bus to AR by enabling the LD i/p of AR.

The next clock transition initiates the transfer from PC to AR since $T_0=1$. In order to implement the second statement

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

1. Enable the read i/p of memory.
2. Place the content of memory onto the bus by making $S_2S_1S_0 = 111$
3. Transfer the content of the bus to IR by enabling LD i/p of IR
4. Increment PC by enabling INR i/p of PC.

The next clock transition initiates the read & increment operations since $T_1=1$.

→ Determine the type of instruction:-

The timing signal that is active after the decoding is T_3. During the interval T_3, the control unit determines the type of instruction that was just read from memory.

The flowchart represents an initial configuration for the instruction cycle and shows how the control determines the instruction type after decoding.

Decoder Output $D_7=1$, if the operation is equal to binary 111. If $D_7=1$, the instruction must be an Input or Output (or) register reference.

If $D_7=0$, the operation code must be one of the 7 values 000 through 110 specify memory reference instruction.

The control then inspects the value of first bit of the instruction which is now available in flip-flop I. If $D_7=0$ & $I=1$ we have a memory reference with an indirect address.

At is then necessary to read the effective address from memory. And direct address can be symbolized by

$$AR \leftarrow M[AR]$$

When a memory reference instruction with I=0 is encountered, it is not necessary to do anything. Since effective address is in AR. Then at time interval T_3 the instruction is executed.

$D_7' I T_3$: $AR \leftarrow M[AR]$

$D_7' I T_3$: Nothing

$D_7 I' T_3$: Register reference

$D_7 I T_3$: Input-Output

This is based on the instruction type.

→ ~~Register reference instructions:-~~

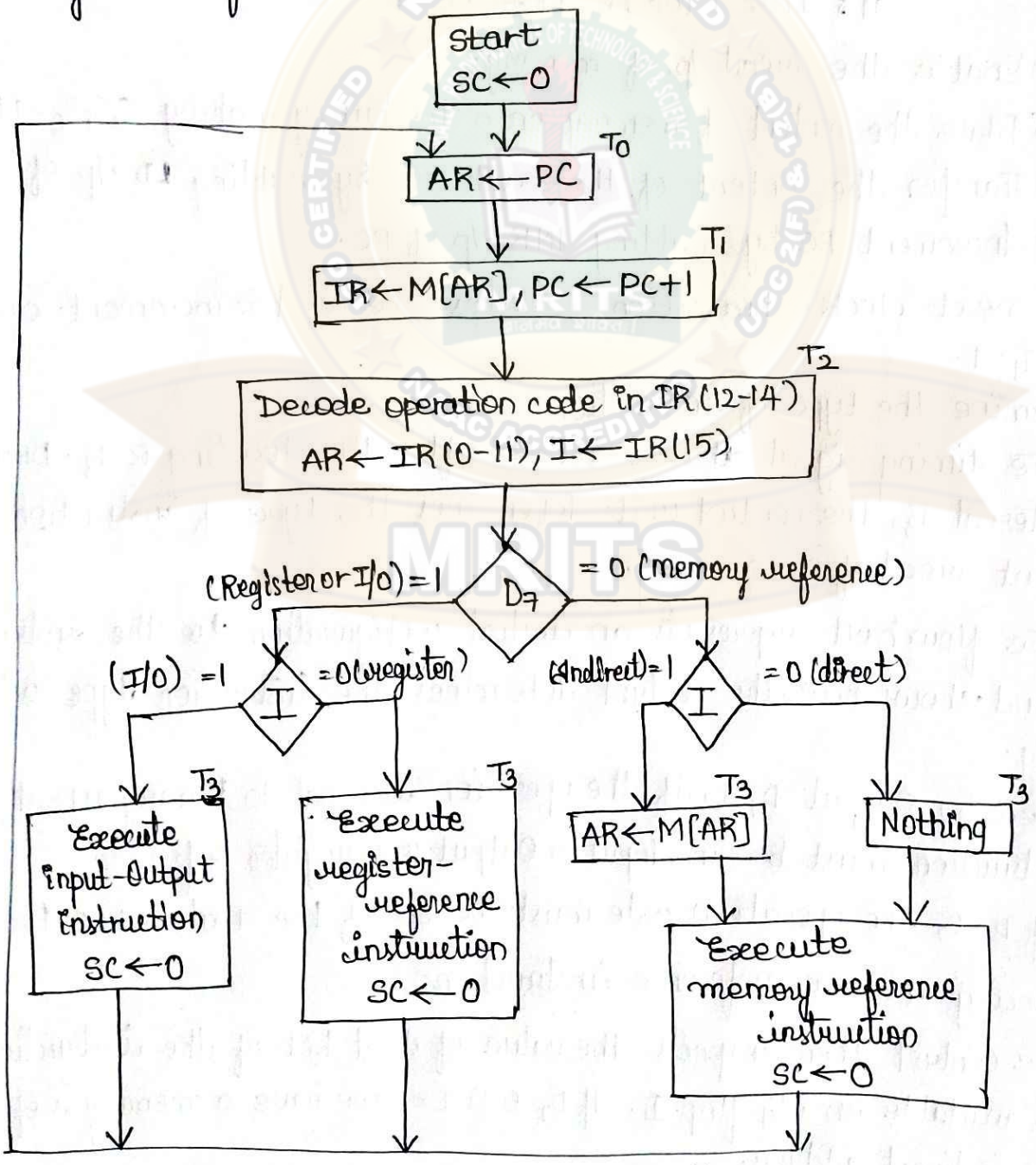


fig:- flowchart for an instruction cycle

→ Register reference Instructions :-

The instructions are recognized by the control when $D_7 = 1$ and $I = 0$. These instructions use bits 0 through 11 of the instruction code to specify one of 12 instructions.

These ~~or~~ 12 bits are available in IR (0 to 11). They were also transferred to AR during T_2.

The control functions and micro operations for the register reference instructions are listed. The control function is distinguished by one of the bits in IR (0 to 11). By assigning the symbol "B" to bit 'i' of IR, all control functions can be simply denoted by rB_i

$D_7 I' T_3 = r$ (common to all register reference instructions).
$IR(i) = B_i$ (bit in IR (0-11) that specifies the operation).

	r: $SC \leftarrow 0$	clear SC
CLA	rB_{11}: $AC \leftarrow 0$	clear AC
CLE	rB_{10}: $E \leftarrow 0$	clear E
CMA	rB_9: $AC \leftarrow \overline{AC}$	complement AC
CME	rB_8: $E \leftarrow \overline{E}$	complement E
CIR	rB_7: $AC \leftarrow shr\ AC$ $AC(15) \leftarrow E$ $E \leftarrow AC(0)$	circular right
CIL	rB_6: $AC \leftarrow shl\ AC$ $AC(0) \leftarrow E$ $E \leftarrow AC(15)$	circular left
INC	rB_5: $AC \leftarrow AC + 1$	Increment AC
SPA	rB_4: If $(AC(15) = 0)$ then $(PC \leftarrow PC + 1)$	skip if Positive
SNA	rB_3: If $(AC(15) = 1)$ then $(PC \leftarrow PC + 1)$	skip if negative
SZA	rB_2: If $(AC = 0)$ then $(PC \leftarrow PC + 1)$	skip if AC zero
SZE	rB_1: If $(E = 0)$ then $(PC \leftarrow PC + 1)$	skip if E zero
HLT	rB_0: $S \leftarrow 0$ (S is a start-stop flip-flop)	Halt computer.

*MEMORY-REFERENCE INSTRUCTION:-

In order to specify the microoperations needed for the execution of each instruction, it is necessary that the function that they are intended to perform be defined precisely.

Memory reference instructions can be defined precisely by means of register transfer notation.

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

→ Table lists the seven memory-reference instructions. The decoded output D_i for $i=0,1,2,3,4,5$ and 6 from operation decoder that belongs to each instruction in table.

→ The effective address of the instruction is in address register AR and was place there during timing signal T_2 when $I=0$ or during timing signal T_3 when $I=1$. The execution of memory-reference instructions starts with timing signal T_4.

→ The actual execution of instruction in the bus system will require sequence of microoperations. This is because data stored in memory cannot be processed directly.

→ The data must be read from memory to a register where they can be operated on with logic circuits

① AND to AC :-

This is the instruction that performs the AND logic operation on pairs of bits in AC & the memory word specified by the effective address. The result of the operation is transferred to AC. The microoperation that executes the instructions are

D_0T_4: $DR \leftarrow M[AR]$

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

② ADD to AC :-

This instruction adds the contents of the memory word specified by the effective address to the value of AC. The sum transferred to AC & the output ~~binary~~ carry C_{out} is transferred to E (Extended Accumulator) flipflop. The microoperations needed to execute this instruction are

D_1T_4 : $DR \leftarrow M[AR]$

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

The same timing signals T_4 & T_5 are used but with operation decoder D_1 instead of D_0.

③ LDA : load to AC :-

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

D_2T_4 : $DR \leftarrow M[AR]$

D_2T_4 : $AC \leftarrow DR$, $SC \leftarrow 0$

④ STA : ~~store~~ AC :-

This instruction stores the contents of AC into memory word specified by the effective address. Since the o/p of AC is applied to the bus & the data i/p of memory is connected to the bus, we can execute this instruction with one microoperation

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

⑤ BUN : Branch Unconditionally :-

This instruction transfers the program to the instruction specified by the effective address. The BUN instruction allows the programmer to specify address.

an instruction out of sequence of the program branches (jumps) unconditionally.

The instruction is executed with one microoperation

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

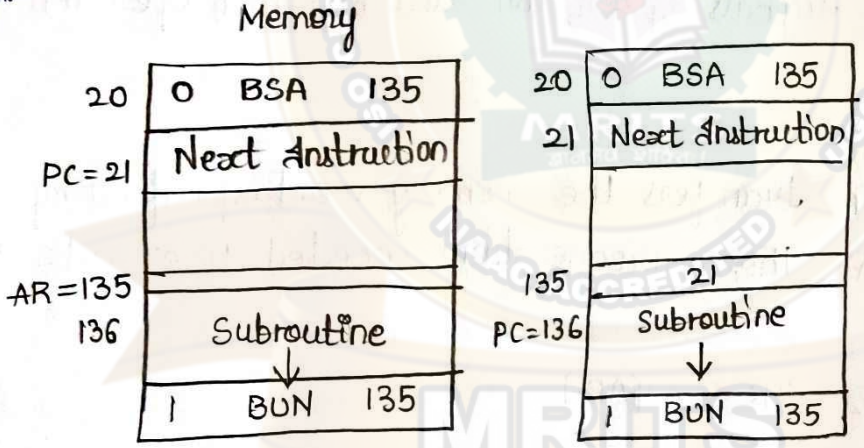
The effective address from AR is transferred through the common bus to PC. Resulting SC to 0 transfers control to T_0.

⑥ BSA: Branch & Save Return Address:-

This instruction is useful for branching to a portion of the program called subroutine (or) procedure. When executed, the BSA instruction stores the address of the next instruction in sequence (PC) into a memory location specified by the effective address. The effective address + one is transferred to PC to serve as the address of the 1st instruction in the subroutine. This operation is specified by the following instruction

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

Eg:-



(a) Memory, PC and AR at time T_4

(b) Memory and PC after Execution.

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

It is not possible to perform the operation of BSA instruction in one clock cycle when we use the bus system. To use the memory bus properly, the BSA instruction must be executed with a sequence of two microoperations

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

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

Timing signal T_4 initiates a memory write operation, places the contents of PC into bus & enables the INR i/p of AR. The memory write operation is completed & AR is incremented by the time the next clock transition occurs. The bus is used at T_5 to transfer the contents of AR to PC.

⑦ ISZ: Increment & skip if zero:-

This instruction increments the word specified by the effective address, and if the incremented value is equal to 0, PC is incremented by 1. The programmer usually stores a negative number in the memory word. As the value is zero.

At the time, PC is incremented by one in order to skip the next instruction in the program.

Since it is not possible to increment a word inside the memory, it is necessary to read the word into DR, increment DR and store the word back into memory. This is done by the following sequence of microoperations

D_6T_4: $DR \leftarrow M[AR]$

D_6T_5: $DR \leftarrow DR+1$

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

★ Control Flowchart:-

→ A flowchart showing all microoperations for the execution of seven memory reference instructions is shown in fig.

→ The control functions are indicated on top of each box.

→ The microoperations that are performed during time T_4, T_5 or T_6 depend on the operation code value. This is indicated in the flowchart by six different paths, one of which the control takes after the instruction is decoded.

→ The sequence counter SC is cleared to 0 with the last timing signal in each case.

→ This causes a transfer of control to timing signal T_0 to start the next instruction cycle.

→ The computer can be designed with a 3-bit sequence counter. The reason for using a 4-bit counter for SC is to provide additional timing

signals for other instructions that are presented in the problems section.

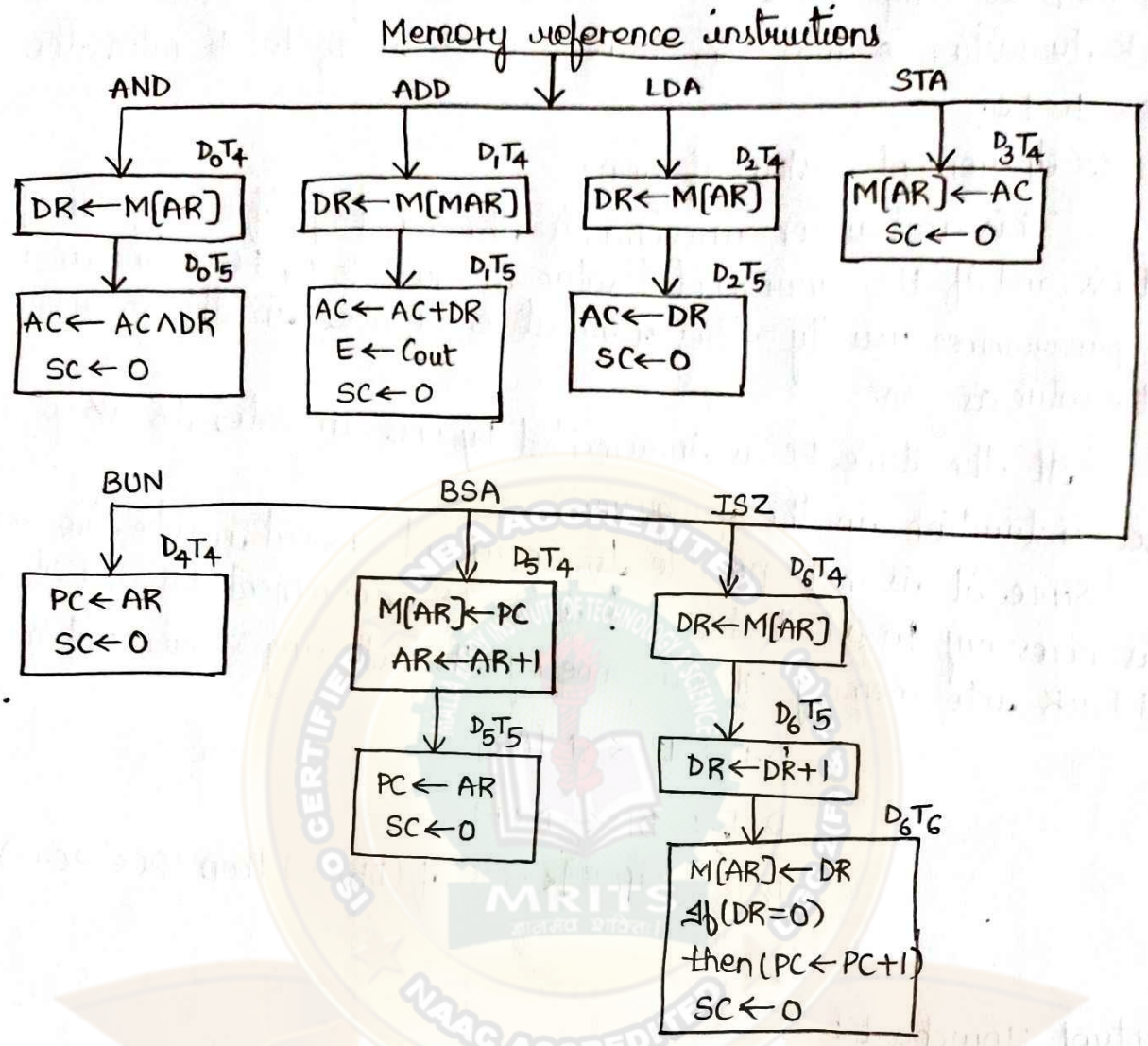


fig: flowchart for memory-reference instructions.

* Input-Output and Interrupt:-

A computer can serve no useful purpose unless it communicates with the external environment. To demonstrate the most basic requirements for input and output communication, we can use a keyboard and printer. Instruction & data stored in memory unit must come from some input device. Computational result (output) must be transmitted to the user through some output device.

* Input-Output Configuration :-

→ The terminal sends and receives serial information. Each quantity of information has 8 bits of an alphanumeric code.

→ The serial information from the keyboard is shifted into the input register INPR

→ The serial information for the printer is stored in the output register OUTR.

→ The 2 registers communicate with a communication interface serially and with the AC in parallel. The input-output configuration is shown in fig. The transmitter interface receives serial information from keyboard and transmits it to INPR.

→ The receiver interface receives information from OUTR and sends it to the printer serially.

→ The input register INPR consists of 8 bits & holds an alphanumeric input information. The 1-bit input flag FGI is a control flip-flop The flag bit is set to 1 when new information is available in the input device & is cleared to 0 when information is accepted by the computer.

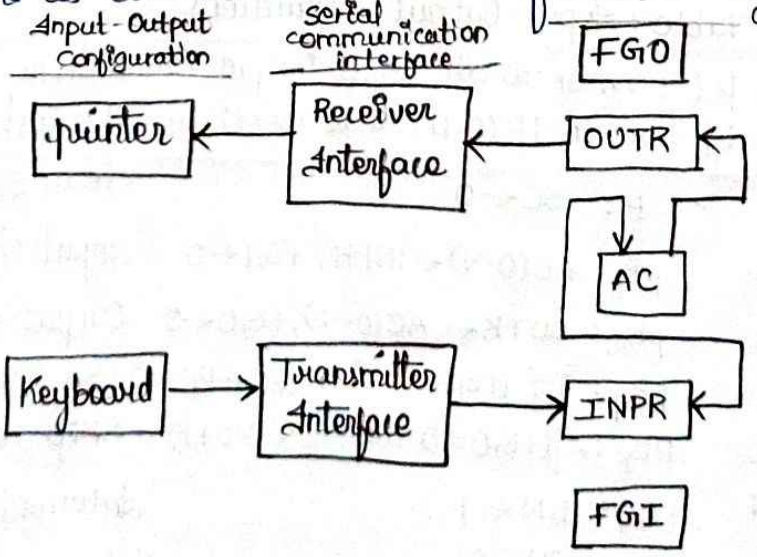


fig:- Input-Output configuration

→ The computer checks the flag bit, if it is 1, the information from INPR is transferred in parallel to AC and FGI is cleared to zero. Once the flag is cleared, new information can be shifted into INPR by striking another key.

→ The Output register OUTR works similarly but the direction of inform-ation is reverse.

→ Initially, the output flag is set to 1. The computer checks the flag bit. If it is 1, the information from AC is transfered in parallel to OUTR and FGO is cleared to zero.

→ The output accepts the coded information, prints the corresponding character and when the operation is completed it sets FGO to 1.

→ The computer doesnot load a new character into OUTR when FGO is zero, because the output device is in the process of printing the character.

★ Input-Output Instruction :-

→ Input-Output instructions are needed for transferring information to and from AC register, for checking the flag bits and for controlling the interrupt facility.

→ Input-Output instructions have an operation code 1111 and are recognized by the control when $D_7 = 1$ and $I = 1$. The remaining bits of the instruction specify the particular operation.

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

Table : Input-Output Instructions

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

→ These instructions are executed with the clock transition associated with timing signal T_3. Each control function needs a boolean relation $D_7 I T_3$, which is designated by symbol p. The control function is distinguished by one of the bits in IR (6-11).

→ By assigning the symbol B_i to bit i of IR, all control functions can be denoted by pB_i for i=6 though 11. The sequence counter SC is cleared to 0 when $p = D_7 I T_3 = 1$

→ The INP instruction transfers the input information from INPR into the eight low-order bits of AC and also clears the input flag to 0.

→ The OUT instruction transfers the eight least significant bits of AC into the output register OUTR and clears the output flag to 0.

→ The SKI and SKO check the status of the flag and causes a skip of the instruction if the flag is 1

→ The last 2 instruction set and clear an interrupt enable flipflop IEN.

★ Program Interrupt :-

→ The process of communication is just referred to as programmed control transfer. The computer keeps checking the flag bit, and when it finds it set, it initiates an information transfer.

→ The difference of information flow-rate b/w the computer and the next of input-output device makes this type of interrupt facility.

→ An alternative to the programmed control procedure is to let the external device inform the computer when it is ready for transfer

→ In the mean time, the computer can be busy with other tasks. This type of transfer uses interrupt facility.

→ While the computer is running a program it doesnot check the flags. However when a flag is set, the computer is momentarily interrupted from proceeding with the current program & it is informed of the fact that a flag has been set.

→ The interrupt enable flipflop IEN can be set and cleared with 2 instructions.

(i) when IEN is cleared to zero (with IOF instruction), the flags cannot interrupt the computer.
(ii) when IEN is set to 1 (with ION instruction), the computer can be interrupted.

→ These instructions provide the programmer with the capability of making a decision as to whether or not to use interrupt facility.

→ The way the interrupt is handled by the computer can be explained by means of the flowchart given.

→ If IEN is 1, the control checks the flag bits. If both flag bits are zero, it indicates that neither the input nor the output registers are ready to transfer information.

→ In this case, control continues with next instruction cycle. If either of the flag is set to 1 while IEN=1, the flipflop is set to 1.

→ At the end of the execute phase, control checks the value of R and if it is equal to 1, it goes to an interrupt cycle instead of instruction.

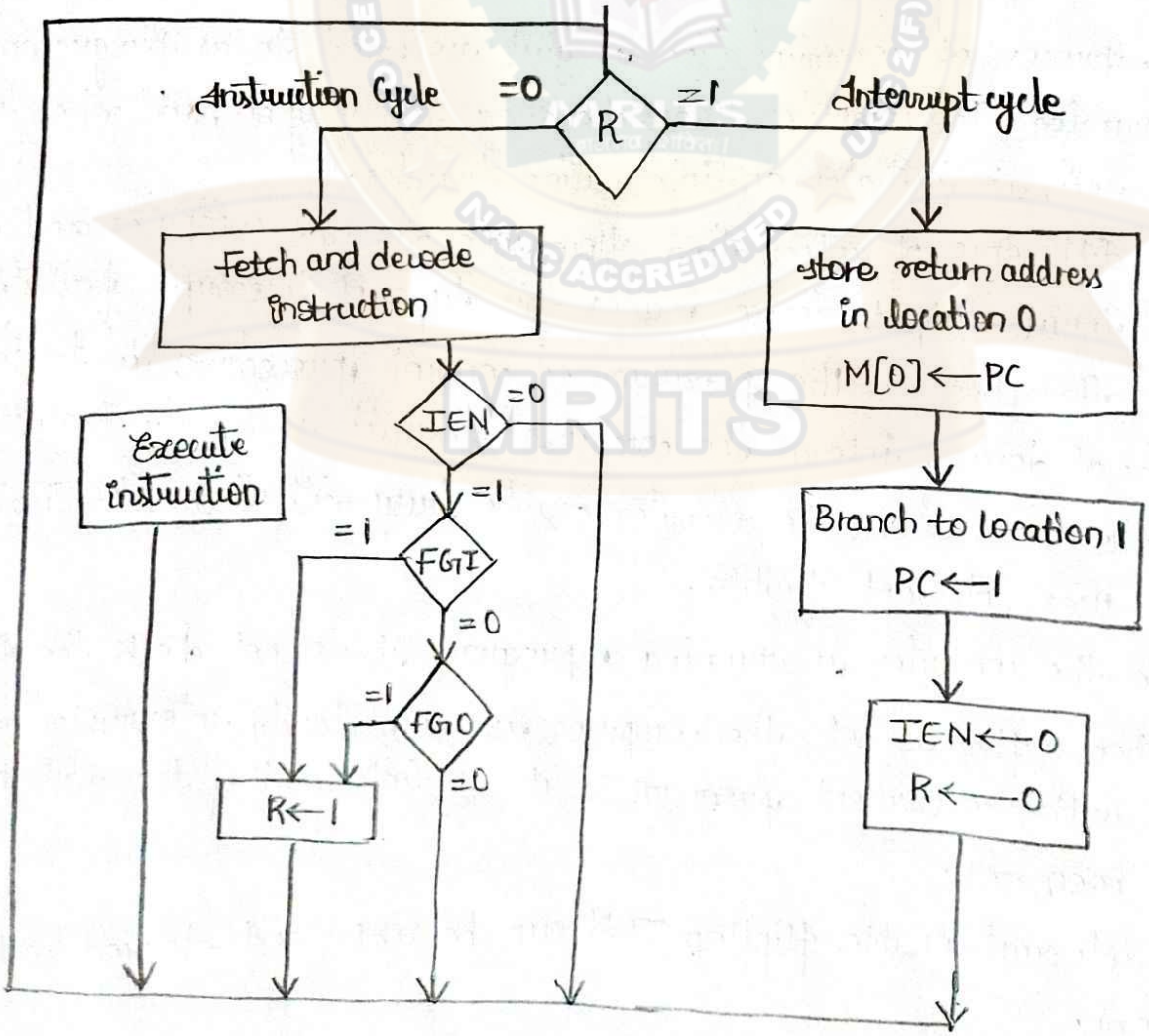


fig:- flowchart for interrupt cycle.

The interrupt cycle is a hardware implementation of a branch and same 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 of which it was interrupted.

An example shows that what happens during the interrupt cycle.

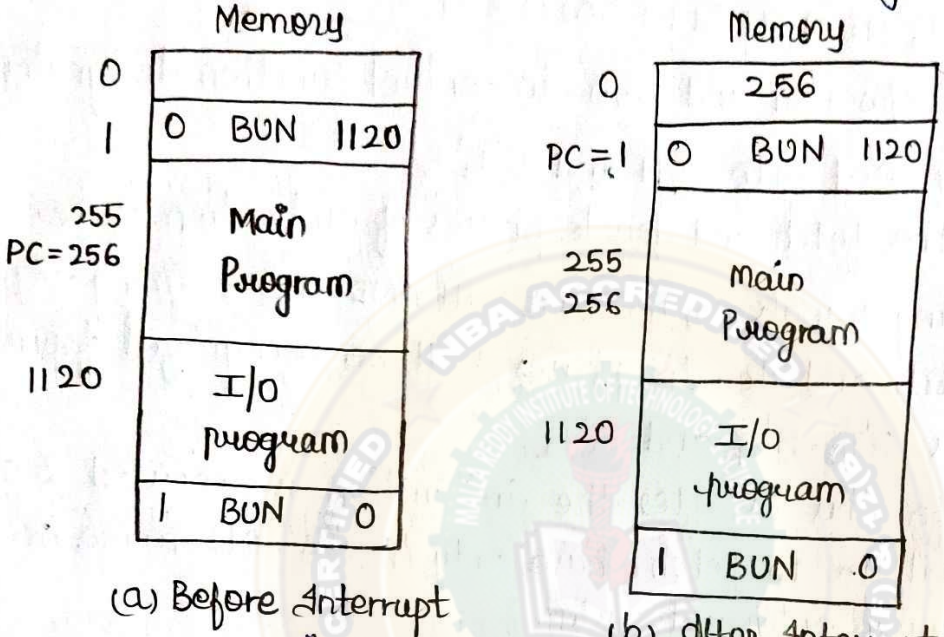


fig: Demonstration of interrupt cycle

Suppose that an interrupt occurs and R is set to 1 while the control is executing the instruction at address 255. At this time, the return address 256 is in PC.

The programmer has previously placed an input-output service program in memory starting from address 1120 and a BUN 1120 instruction at address 1.

When control reads timing signal T_0 & find R=1, it proceeds with interrupt cycle. The content of RC(256) is stored in memory location "0", PC is set to 1 & R is cleared to 0.

At the begining of next instruction cycle, the instruction that is read from memory is in address 1. Since this content is in PC.

The branch instruction at address 1 causes the program to transfer to the I/p-O/p service program at address 1120 & the next instruction is indirect memory.

So, we move to the location 0 & then we get address 256 that is the main program.

* Interrupt cycle:-

→ The interrupt cycle is initiated after the last execute phase if the interrupt flipflop R is equal to 1. This flipflop 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 T_0, T_1 or T_2 are active. The condition for setting flipflop R to 1 can be expressed as

$$T_0'T_1'T_2'(IEN)(FGI + FGO) : R \leftarrow 1$$

→ The symbol "+" b/w FGI and FGO in control function designates logic OR operation. This is ANDed with IEN and $T_0'T_1'T_2'$

→ To modify the fetch and decode phases of instruction cycle. Instead of using only timing signals T_0, T_1, T_2 we will AND the 3 timing signals with R' so that the fetch and decode phases will be recognized from the 3 control functions $R'T_0$, $R'T_1$ and $R'T_2$

→ The reason for this is after the instruction is executed & SC is cleared to 0, the control will go to fetch phase only if R=0. Otherwise if R=1, the control will go through interrupt cycle.

→ The interrupt cycle stores return address into memory location 0, branches to memory location 1 & clears IEN, R & SC to 0. This is done by the sequence of microoperations

RT_0: AR ← 0, TR ← PC

RT_1: M[AR] ← TR, PC ← 0

RT_2: PC ← PC + 1, IEN ← 0, R ← 0, SC ← 0

→ During first timing signal AR is cleared to 0 & content of PC is transferred to TR. With second timing signal, return address is stored in memory at location 0 and PC is cleared to 0. At 3rd timing signal, PC is incremented, clears IEN and R & control goes back to T_0 by clearing SC to 0

→ The beginning of next instruction cycle has the condition $R'T_0$ and the content of PC = 1

→ The control then goes through an instruction cycle that fetches & executes the BUN instruction in location 1.

UNIT: 2 (a)

MICROPROGRAMMED CONTROL

Introduction:-

The major functional parts in a digital computer are central processing unit (CPU), Memory and Input-Output.

The main digital hardware functional units of CPU are control unit, ALU and memory unit.

The function of the control unit in a digital computer is to initiate sequences of microoperations. Two methods of implementing control unit are

↳ Hardwired control

↳ Microprogrammed control.

↳ **Hardwired control :-**

When the control signals are generated by hardware using conventional logic design techniques, the control unit is said to be hardwired.

↳ **Microprogrammed control :-**

A control unit whose binary control variables are stored in memory is called a microprogrammed control.

→ **Dynamic microprogramming :-**

A more advanced development known as dynamic microprogramming permits a microprogram to be loaded initially from an auxiliary memory such as a magnetic disk. Control unit that use dynamic microprogramming employ a writable control memory. This type of memory can be used for writing.

→ **Control Memory.**

Control Memory is the storage in the microprogrammed control unit to store the microprogram.

↳ **Writable control Memory :-**

Control storage whose contents can be modified, allow the change in microprogram and instruction set can be changed or modified is referred as writable control memory.

→ Control word:-

The control variables at any given time can be represented by a control word string of 1's and 0's called a control word.

→ Microoperation:-

An computer central processing units, microoperation (also known as micro-ops or μops) are detailed low level instructions used in some designs to implement complex machine instructions.

→ Microinstruction:-

↳ A computer microprogram can be translated into its binary equivalent by means of an assembler.

↳ Each line of the assembly language microprogram defines a symbolic microinstruction.

↳ Each symbolic microinstruction is divided into five fields:

label, microoperations, CD, BR and AD.

→ Microprogram:-

↳ A sequence of microinstructions constitutes a microprogram

↳ Since alterations of the microprogram are not needed once the control unit is in operation, the control memory can be a Read only Memory (ROM).

↳ ROM ~~are~~ words are made permanent during the hardware production of the unit.

↳ The use of microprogram involves placing all control variables in words of ROM for use by the control unit through successive read operation.

↳ The content of the word in ROM at a given address specifies a microinstruction

→ Micro 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 ~~subroutine~~ memory reference instructions.

★ Organization of micro programmed control Unit:-

→ The general configuration of microprogrammed control unit is demonstrated in block diagram ~~in~~ of following fig.

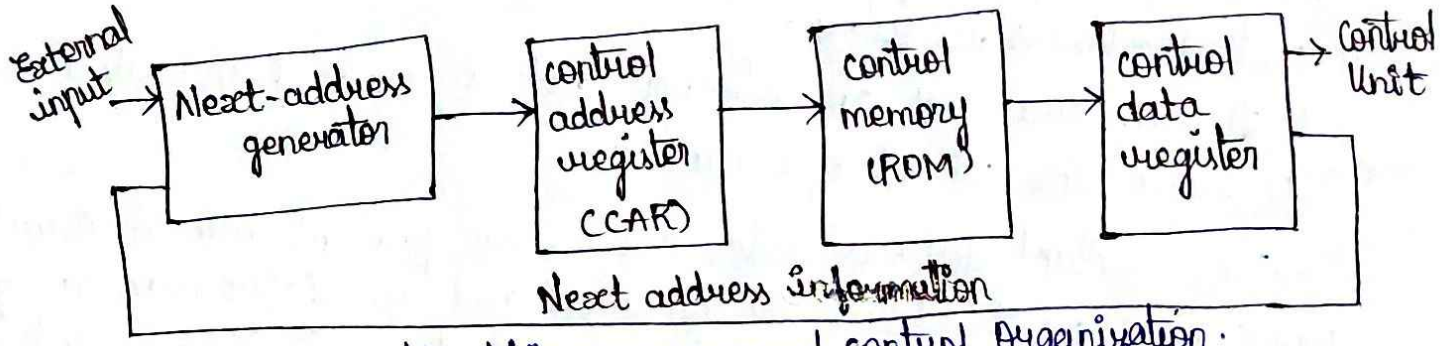


fig: Microprogrammed control Organization.

→ The control memory is assumed to be a ROM, within which all control information is permanently stored.

→ The control ~~word~~ memory address register specifies the address of the microinstruction and the control data register holds the microinstruction read from memory.

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

→ The location of the next microinstruction may be the one next in sequence, or it may be located somewhere else in the control memory.

→ While the microoperations are being executed, the next address is computed in the next address generator circuit and then transferred into the control address register to read the next microinstruction.

→ Thus a microinstruction contains bits for initialising or initiating microoperations in the data processor part and bits that determine the address sequence for the control memory.

→ The next address generator is sometimes called a micro-program sequencer as it determines the address sequence that is read from control memory.

→ The main functions of a microprogram sequencer are as follows-
↳ It can increment the control register by 1.
↳ It can load the address from the control memory to the control address register.
↳ It can transfer an external address or load an initial address to begin the start operation.
→ The control data register holds the present microinstruction while the next address is computed and read from memory.
The data register is sometimes called a pipeline register.
→ It allows the execution of the microoperations specified by the control word simultaneously with the generation of next microinstruction
→ This configuration requires a 2-phase clock, with one clock applied to the address register and other to the data register.
→ The main advantage of the micro programmed control is the fact that once the hardware configuration is established; there should be no need for further hardware or writing changes.
→ If we want to establish a different control sequence for the system, all we need to do is specify a different set of microinstructions for control memory.

* Address Sequencing:-

→ Microinstructions are stored in control memory in groups, with each group specifies a routine. The hardware that controls the address sequencing of the control memory and must be able of sequencing the microinstruction with a routine and be able to branch from one routine to another.

→ An initial address is loaded into the control address register (CAR) when power is turned on. This initial address is the address of the first microinstruction that activates the fetch routine. After the end of fetch routine, the instruction is in the instruction register of the computer.

→ The control memory must go through the routine that determines the EA of the operand. After computing the effective address the address of the operand is available in the memory address register.

→ The next step is to generate the microoperations that execute the instruction fetched from memory. The microoperation steps to be generated in processor registers depend upon the operation code part of instruction

→ Each instruction has its own microprogram routine stored in a given location of the control memory.

→ The transformation from the instruction code bits to an address in the control memory where the routine is located is called as Mapping.

→ After the execution of the instruction control must return to the fetch routine.

→ Address Sequencing Capabilities :-

① Incrementing of the control address register.

② Unconditional branch or conditional branch, depending on status bit conditions.

③ Mapping process (bits of the instruction address for control memory).

④ A facility for subroutine return.

The below figure shows a block diagram of a control ~~word~~ memory and the associated hardware needed for selecting the next microinstruction address.

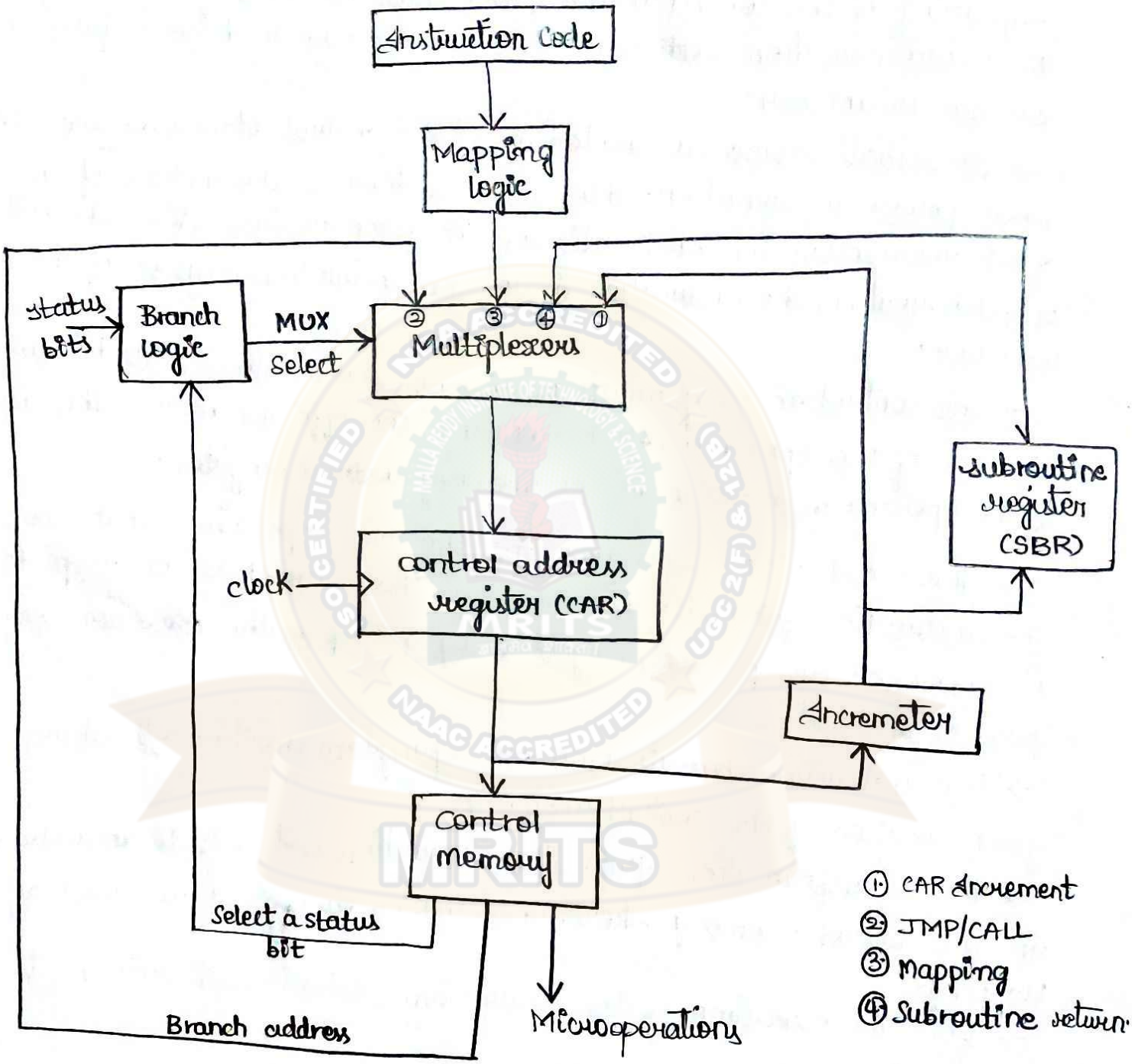


fig: Selection of address for control memory.

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

Microprograms that employ subroutines will require an external register for storing the return address. Return address cannot be stored in ROM.

In the figure, 4 different paths from which the control address register (CAR) receives the address.

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

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

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

↳ An external address is transferred into ~~o~~ control ~~word~~ memory via a mapping logic circuit.

↳ The return address for a subroutine is stored in a special register that value is used when the microprogram wish to return from the subroutine.

★ Conditional Branching :-

→ Conditional branching is obtained by using part of the micro-instruction to select a specific status bit in order to determine its condition.

→ The status conditions are special bits in the system that provide parameter information such as the carry-out of an adder, the sign bit of a number, the mode bits of an instruction and I/O status conditions.

→ The status bits, together with the field in the microcontroller that specifies a branch address, control the branch logic.

→ The branch logic tests the condition, if met then branches, otherwise increments the CAR.

→ If there are 8 status bit conditions, then 3 bits in the microinstruction are used to specify the condition and provide the selection variables for the multiplexer.

→ For unconditional branching, fix the value of one status bit to be one. Load the branch address from control memory into the CAR.

★ Mapping of Instruction :-

→ A special type of branch exists when a microinstruction specifies a branch to the first word in control memory where a microprogram routine is located

→ The status bits for this type of branch are the bits in the Opcode.

→ Assume an opcode of 4 bits and a control memory of 128 locations.

The mapping process converts the 4-bit opcode to a 7-bit address for control memory shown in below figure.

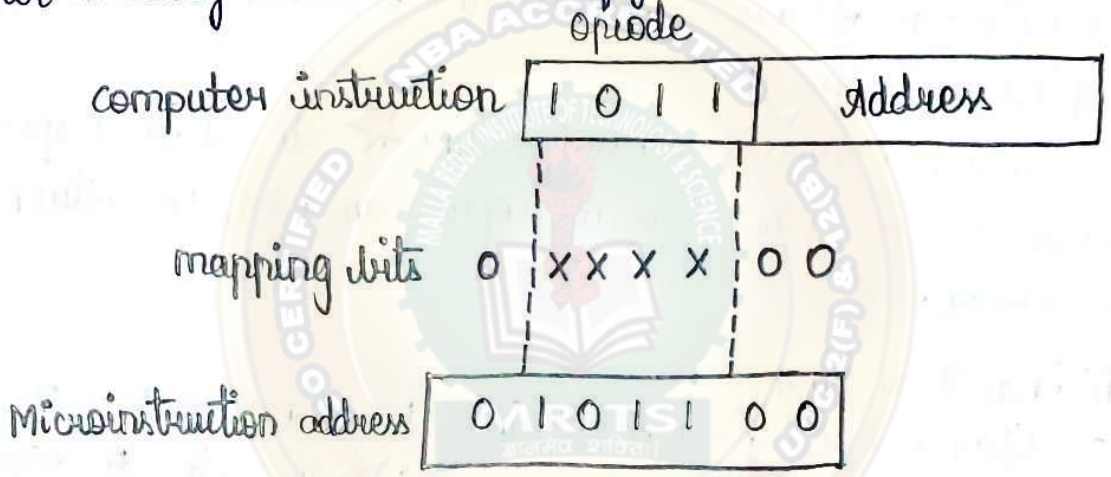
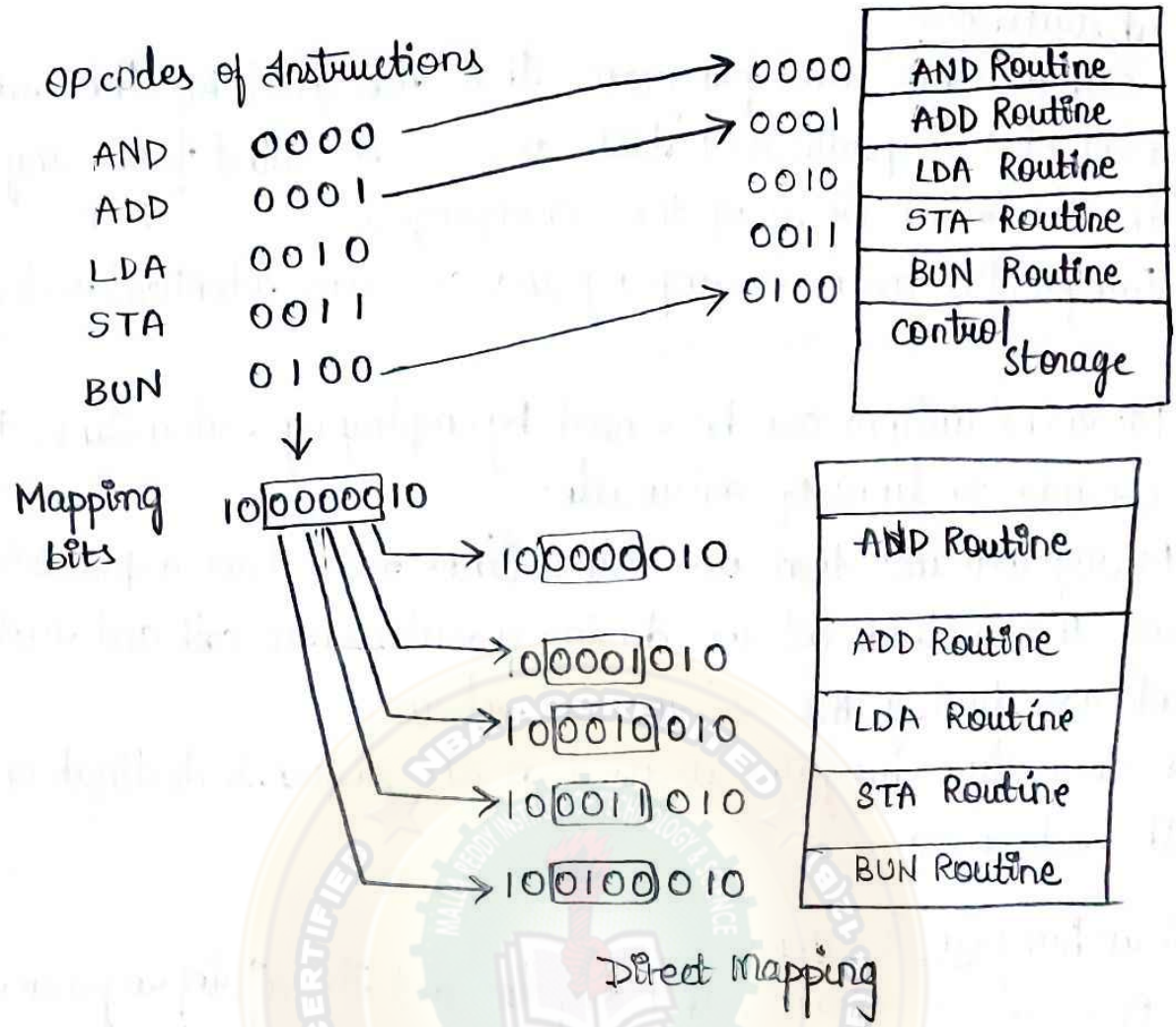


fig: Mapping from instruction code to microinstruction address.

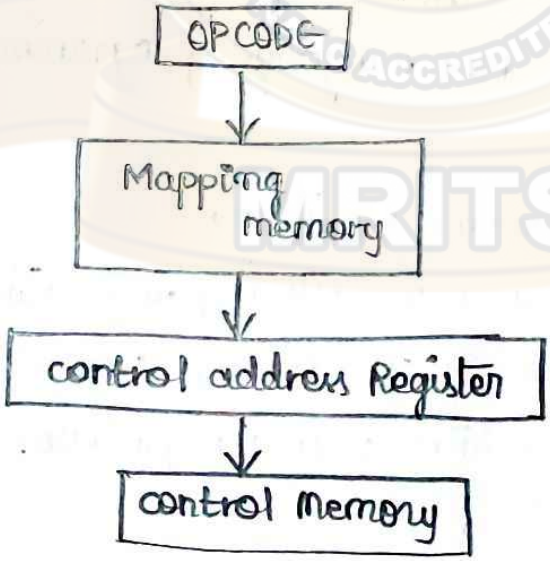
→ Mapping consists of placing a 0 in the most significant bit of the address, transferring the four operation code bits, and clearing the 2 least significant bits of the control address register.

→ With the help of this process, a micro-program will be provided to each computer instruction. The microprogram contains the capacity of four microoperations. If less than 4 microinstructions are used by the routines, the location of unused memory can be used for other routines. If more than 4 microinstructions are used by the routine, it will use the addresses 1000000 through 1111111.



Direct Mapping

This concept can be extended to a more general mapping rule with the help of PLD or ROM.



The above image shows the mapping of address of microinstruction from the OP-code of an instruction. In the execution program, this microinstruction is the starting microinstruction.

* Subroutines:-

→ Subroutines are programs that are used by other routines to accomplish a particular task and can be called from any point within the main body of the microprogram.

→ Frequently main microprograms contains identical section of code.

→ Microinstructions can be saved by employing subroutines that use common sections of microcode.

→ Microprograms that use subroutines must have a provision for storing the return address during a subroutine call and restoring the address during the subroutine return.

→ A subroutine register is used as the source & destination for the addresses.

* Micro Program Example :-

One the configuration of a computer and its microprogrammed control unit is established, the designer's task is to generate the microcode for the control memory.

This code generation is called microprogramming and is a process similar to conventional machine language programming.

* Computer Configuration :-

→ It consists of two memory units : a

↳ A main memory for storing instructions and data.

↳ A control memory for storing the microprogram.

→ Four registers are associated with the processor unit and two with the control unit.

↳ The processor registers are PC, AR, DR and AC.

↳ The control unit has control address register (CAR) and Subroutine register SBR.

↳ The transfer of information among registers in the processor is through Multiplexers rather than a bus.

↳ DR can receive information from AC, PC or memory. AR can receive information from PC or DR. PC can receive information only from AR.
↳ The arithematic, logic and shift unit performs microoperations with data from AC and DR and places the result in AC. Note that memory receives its address from AR.
↳ Input data written to memory come from DR, and data read from memory can go only from AR.

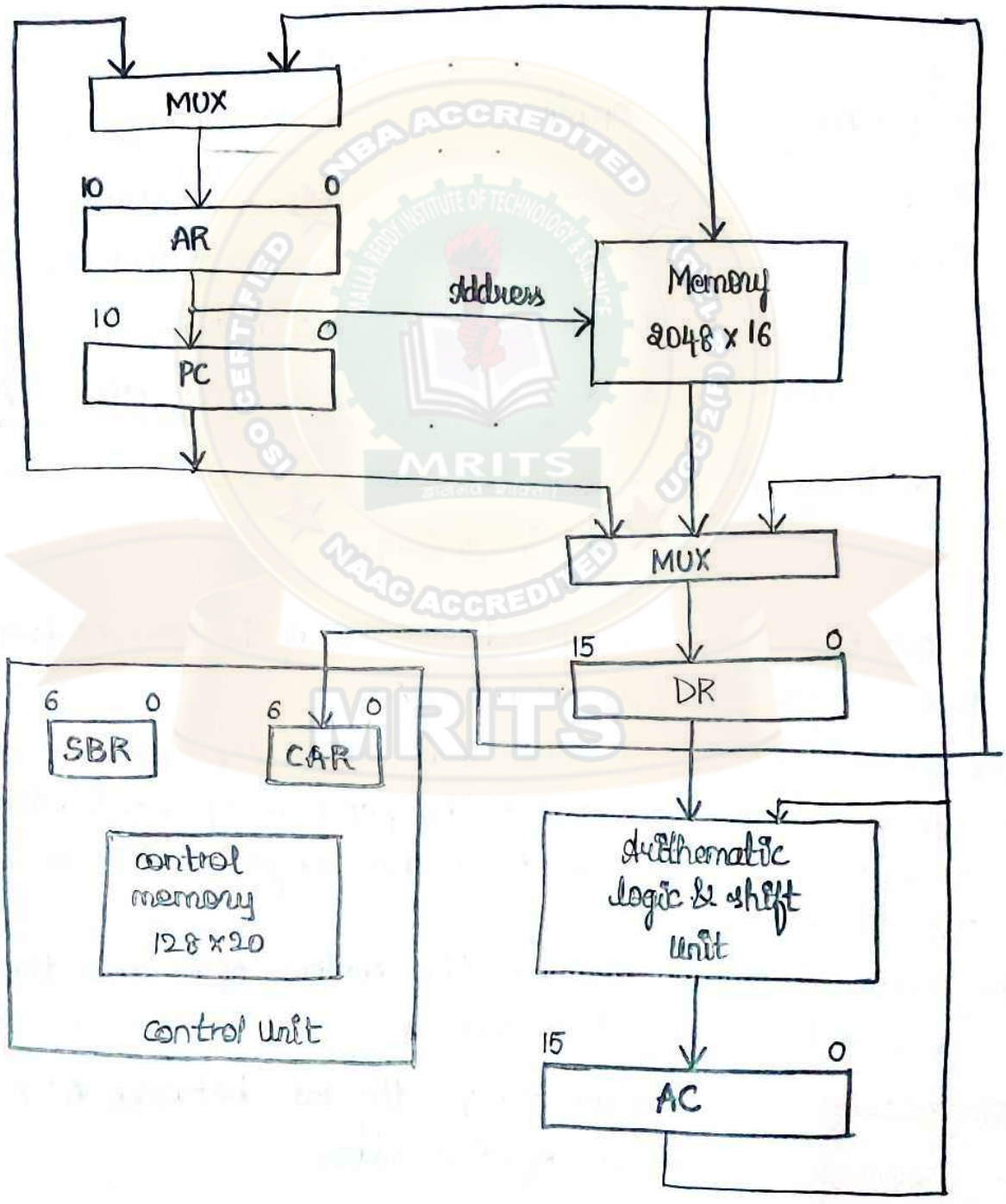


fig: Computer Hardware Configuration

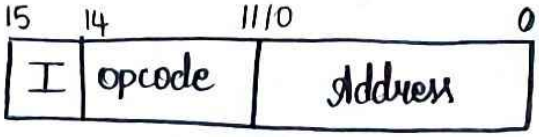
Microinstruction format :-

→ The computer instruction format has three fields :

↳ A 1-bit field for indirect addressing symbolized by I.

↳ A 4-bit operation code (op-code)

↳ An 11-bit address field.



(a) Instruction format

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

EA is Effective address

(b) Four Computer Instructions

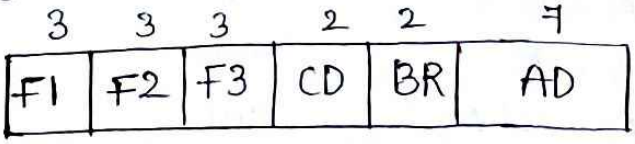
→ The ~~Add~~ ADD instruction adds the content of the operand found in the effective address to the content of AC.

→ The BRANCH instruction causes a branch to the effective address if the operand in AC is negative. The program proceeds with the next consecutive instruction if AC is not negative. The AC is negative if its sign bit is a 1.

→ The STORE instruction transfers the content of AC into the memory word specified by the effective address.

→ The EXCHANGE instruction swaps the data between AC & the memory word specified by the effective address

→ The microinstruction format for the control memory is shown in the below figure.



→ The microinstruction format is composed of 20 bits with four parts to it.

- ↳ Three fields F1, F2, and F3 specify microoperations for the computer.
- ↳ The CD field selects status bit conditions (2 bits)
- ↳ The BR field specifies the type of branch to be used (2 bits)
- ↳ The AD field contains a branch address (7 bits)

→ Each of the 3 microoperation fields can specify one of 7 possibilities
→ No more than three microoperations can be chosen for a microinstruction
→ If fewer than three are needed, the code 000 = NOP.
→ The three bits in each field are encoded to specify 7 distinct microoperations listed in below table.

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

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

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

→ Five letters to specify a transfer-type microoperation.

↳ First 2 designate the source register

↳ Third is 'T'

↳ Last two designate the destination register

AC ← DR F1 = 100 = DRTAC

→ The condition field (CD) is two bits to specify 4 status bit conditions shown below.

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

→ The branch field (BR) consists of two bits and is used with the address field to choose the address of the next microinstruction.

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

Symbolic microinstruction :-

→ Each line of an assembly language microprogram defines a symbolic microinstruction and is divided into 5 parts:

① The label field may be empty or it may specify a symbolic address. Terminate it with a colon (:).

② The microoperations field consists of 1-3 symbols, seperated by commas; Only one symbol from each field. If NOP, then translated to 9 zeros.

③ The condition field specifies one of the four conditions.

④ The branch field has one of the four branch symbols

⑤ The address field has three formats.

(a) A symbolic address - must also be a label.

(b) The symbol NEXT to designate the next address in sequence.

(c) Empty if the branch field is RET or MAP and is converted to 7 zeros.

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

→ ORG 64 - places first microinstruction at control memory 1000000

Fetch Routine :-

→ The control memory has 128 words and each word contains 20 bits To microprogram the control memory, it is necessary to determine the bit values of each of the 128 words.

→ The first 64 words are to be occupied by routines for the 16 instructions. The last 64 words may be used for any other purpose

→ A convenient starting location for the fetch routine is address 64.

→ The microinstructions needed for the fetch routine are

AR ← PC

DR ← M[AR], PC ← PC+1

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

→ The address of the instruction is transferred from PC to AR and the instruction is then read from memory into DR. Since no instruction is available, the instruction code remains DR.

→ The address part is transferred to AR and then control is transferred to one of the 16 routines by mapping the operation code part of the instruction from DR into CAR.

→ The fetch routine needs three microinstructions which are placed in the control memory at address 64, 65 & 66.

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

→ The translation of the symbolic microprogram to binary produces the following microprogram. The bit values are obtained from table of binary code of microinstruction fields.

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

→ The three microinstructions that constitute the fetch routine have been listed in three different representations.

→ The register transfer representation shows the internal register transfer operations that each microinstruction implements

→ The symbolic representation is useful for writing microprograms in an assembly language format.

→ The binary representation is the actual internal content that must be stored in control memory.

→ It is customary to write microprograms in symbolic form & then use an assembler program to obtain a translation to binary.

Symbolic Microprogram:-

→ The execution of the third microinstruction (MAP) in the fetch routine results in a branch to address 0xxxxxx00, where xxxx are the 4 bits of the operation code.

→ In each routine we must provide microinstructions for evaluating the effective address and for executing the instruction. The indirect address mode is associated with all memory-reference instructions.

→ Saving the number of control memory words may be achieved if the microinstructions for the indirect address are stored as a subroutine

→ This subroutine symbolised by INDRCT is located right after the fetch routine is shown in table. The table also shows the symbolic microprogram for the fetch routine and the microinstruction routines that executes four computer instructions

→ To see how the transfer & return from the indirect subroutine occurs, assume that the MAP microinstruction at the end of the fetch routine caused a branch to address 0, where the ADD routine is stored.

The INDRCT subroutine has 2 microinstructions

```
INDRCT: READ  U   JMP   NEXT
        DRTAR U   RET
```

Symbolic microprogram

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

→ The Execution of the ADD instruction is carried out by the microinstruction at address 1 and 2. The first instruction reads an operand from memory into DR. The second microinstruction performs an add microoperation with the content of DR & AC & then jumps back to beginning of fetch routine.

→ The BRANCH instruction should cause a branch to the effective address if AC<0. AC<0 if its sign is -ve which is detected from status bit S being 1. The branch in table starts by checking the value of S. If S=0 no branch occurs & next microinstruction causes a jump back to the fetch routine without altering the content of PC. If S=1, the first JMP microinstruction transfers control to location OVER.

→ The microinstruction at this address location calls the INDRCT subroutine if I=1. The effective address is then transferred from AR to PC & program jumps back to fetch routine.

→ The STORE routine again uses the INDRCT subroutine if I=1 The content of AC is transferred to DR. A memory write operation is initiated to store the content of DR in a location specified by the EA in AR.

→ The EXCHANGE routine reads operand from EA & places it in DR. The contents of DR & AC are interchanged in 3rd microoperation. This is possible when registered are of edge triggered type. The original content of AC that is now stored back in memory.

☆ Binary microprogram:-

→ The symbolic microprogram is a convenient form for writing microprograms in a way that people can read and understand. But this is not the way that the microprogram is stored in memory.

→ The symbolic microprogram must be translated to binary either by means of an assembler program or by the user if the microprogram is simple enough as in this example

The equivalent binary form of the microprogram is listed in Table.

Table: Binary Microprogram for control memory

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

→ The binary microprogram listed in table above specifies the word content of the control memory. When a ROM is used for the control memory, the microprogram binary list provides the truth table for fabricating the unit.

→ Fabrication is a hardware process & consists of creating a mask for the ROM so as to produce 0's & 1's for each word.

→ The bits of ROM are fixed once the internal links are fused during the Hardware production.

→ If a writable control memory is employed, the ROM is replaced by a RAM. The advantage of employing a RAM for the control memory is that the microprogram can be altered by writing new pattern of 0s & 1s without resorting the hardware procedures.

→ A writable control memory posses the flexibility of choosing the instruction set of a computer dynamically by changing the microprogram under process control.

→ However most microprogrammed systems use a ROM for the control memory because it is cheaper & faster than a RAM & also to prevent the occasional user from changing the architecture of system.

Differences between Hardwired Control & Microprogrammed Control Unit

Hardwired Control	Microprogrammed control
① Hardwired control unit generates the control signals needed for the processor using logic circuits	① Microprogrammed control unit generates the control signals with the help of micro-instructions stored in control memory.
② Hardwired control unit is faster when compared to microprogramed control unit as the required control signals generated with the help of hardwares.	② This is slower than the other as microinstructions are used for generating signals here.
③ Difficult to modify as the control signals that need to be generated are hardwired	③ Easy to modify as the modification need to be done only at the instruction level.
④ More costlier as everything has to be realized in terms of logic gates	④ Less costlier than hardwired control as only microinstructions are used for generating control signals.

⑤ It cannot handle complex instructions as the circuit design for it becomes complex.	⑤ It can handle complex instructions
⑥ Only limited no. of instructions are used due to hardware implementation.	⑥ Control signals for many instructions can be generated.
⑦ Used in computer that makes use of reduced instruction set computers (RISC)	⑦ Used in computer that makes use of complex instruction set computers (CISC).

* DESIGN OF CONTROL UNIT:-

→ The microinstruction format usually divided into fields, each field provides control bits to initiate microoperations in the system, special bits to specify the way the next address to be evaluated and an address field for branching.

→ The method of grouping mutually exclusive variables into fields and encoding the k bits in each fields to provide 2^k microoperations is used to reduce the no. of control bits that initiate microoperations. Each field requires a decoder to produce the corresponding control signals.

→ This method has an advantage of reducing the size of the micro-instruction bits with the following drawbacks:-

→ It requires additional hardware external to the control memory.

→ Increases the delay time of the control signals because they must propagate through the decoding circuits.

→ The encoding of control bits was demonstrated in the programming example of the preceding section.

→ The 9 bits of the microoperation field are divided into 3 subfields of 3 bits each.

→ The control memory output of each subfield must be decoded to provide the distinct microoperations.

→ The outputs of the decoders are connected to the appropriate inputs in the processor unit.

→ The following figure shows the three 3×8 decoders. Each decoder is used to decode one field of the microinstruction, presently available in the output of control memory to provide eight outputs.

→ Each of the output must be connected to proper circuit to initiate corresponding microoperations.

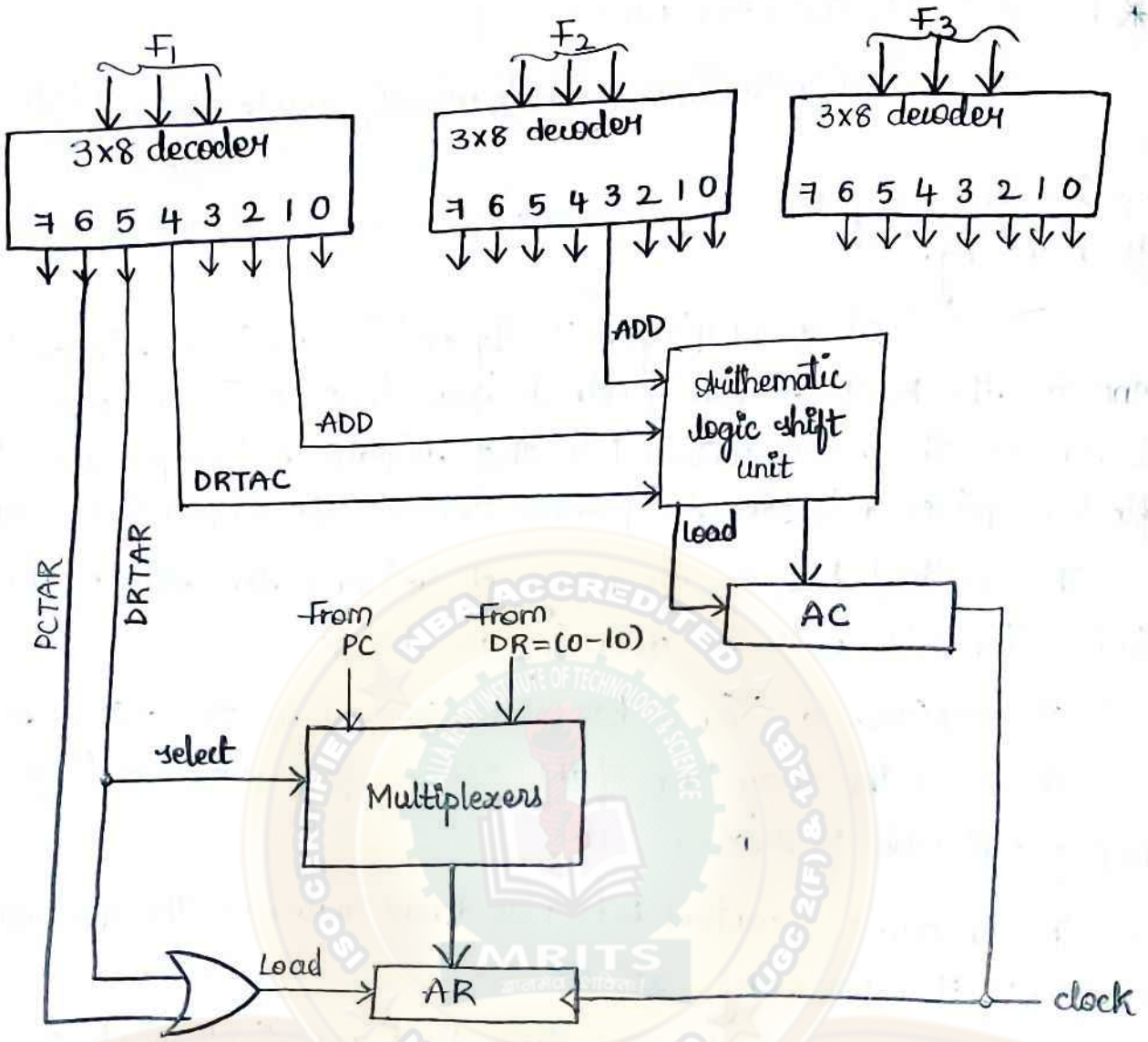


fig: Decoding of microoperation fields

→ For example

↳ when F1 = 101 (binary 5), the next pulse transition transfers the content of DR (0-10) to AR.

↳ Similarly when F1 = 110 (binary 6) there is a transfer from PC to AR (symbolized by PCTAR).

↳ As shown in figure, outputs 5 and 6 of decoder F1 are connected to the load i/p of AR so that when either one of these outputs is active. information from the multiplexers is transferred to AR.

↳ The multiplexers select the information from DR when output 5 is active and from PC when output 5 is inactive.

→ The transfer into AR occurs with the clock transition only when Output 5 or Output 6 of the decoder is active

→ For the arithmatic logic shift unit, the control signals are instead of coming from the logical gates. Now these inputs will now come from the outputs of AND, ADD and DRTAC respectively. The other o/p of decoders that are associated with AC operation must also be connected to arithematic logic shift unit.

★ Microprogram Sequencer:-

→ The basic components of microprogrammed control unit are

↳ The control unit

↳ The circuits that select the next address.

→ The address selection part is called a microprogram sequencer.

→ A microprogram sequencer can be constructed with digital functions to suit a particular application.

→ The purpose of microprogram sequencer is to present an address to the control memory so that a microinstruction may be read and executed.

→ The next-address logic of the sequencer determines the specific address source to be loaded into the control-address register (CAR).

→ The choice of the address source is guided by the next-address information bits that the sequencer receives from the present microinstruction.

→ The internal structure of a typical microprogram sequencer show a particular unit that is suitable for use in the microprogram computer example.

→ The block diagram of the microprogram sequencer is shown in the figure.

→ The control memory is included in the diagram to show the interaction between the sequencer and 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 control address register CAR.
↳ The second multiplexer tests the value of a selected status bit and the result of the test is applied to an input logic circuit.

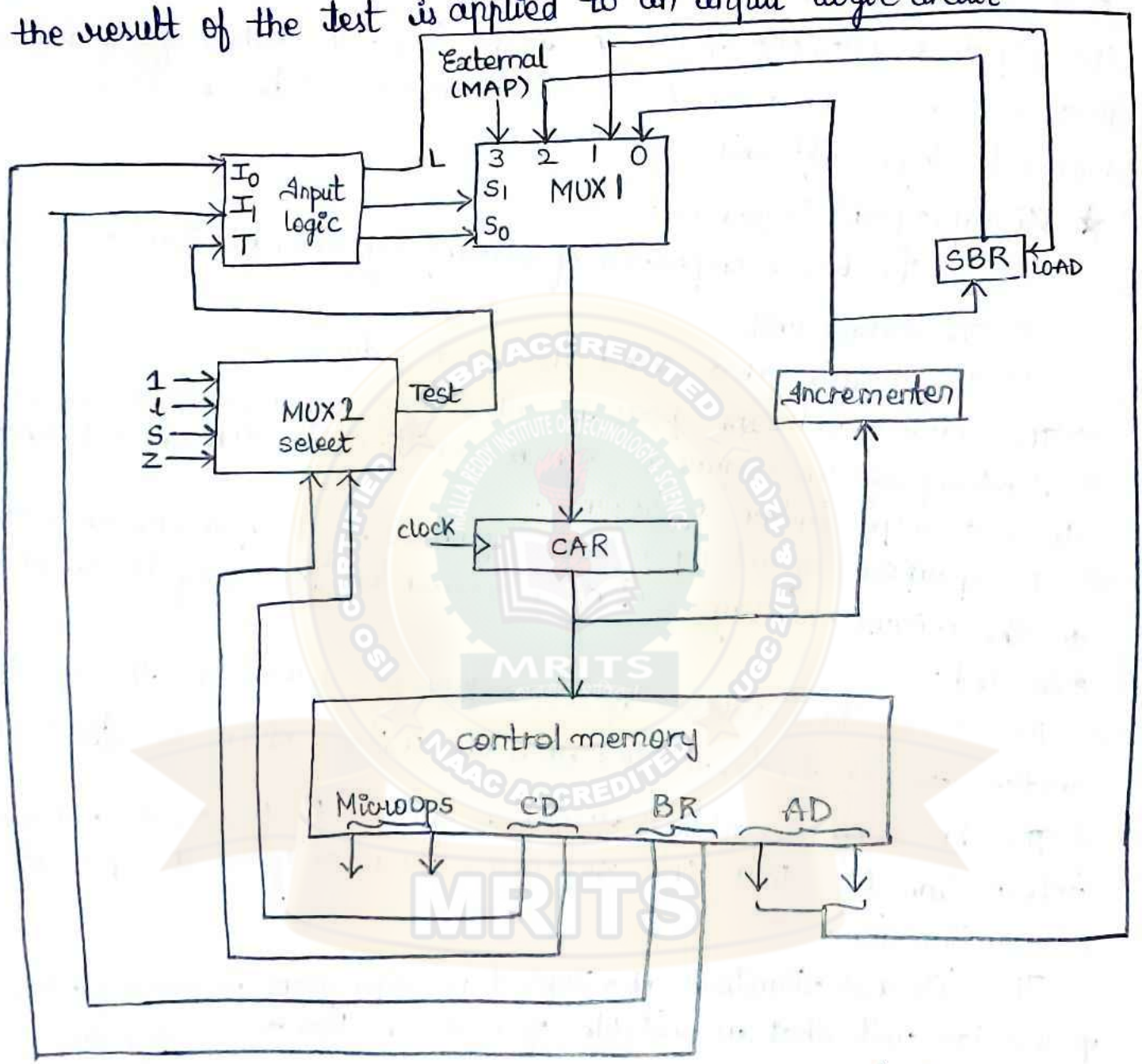


fig:- Microprogram Sequencer for a control memory

→ The output from CAR provides the address for control memory.
→ The content of CAR is incremented and applied to one of the multiplexer inputs and to the subroutine register SBR.

The other three inputs to multiplexer come from
↳ The address field of the present microinstruction
↳ From the out of SBR
↳ From an external source that maps the instruction

→ The CD (condition) field of the microinstruction selects one of the status bits in the second multiplexer.
→ If the bit selected is equal to 1, the T variable is equal to 1; otherwise it is equal to 0.
→ The T value together with two bits from the BR (branch) field goes to an input logic circuit.
→ The input logic in a particular sequencer will determine the type of operations that are available in the unit.
→ The input logic circuit in the figure has three inputs I_0, I_1 and T and three outputs S_0, S_1 and L.
→ Variables S_0 and S_1 select one of the source addresses for CAR. Variable L enables the load i/p in SBR.
→ The binary values of the selection variables determine the path in the multiplexer.
→ For example, with $S_1S_0 = 10$ multiplexer input number 2 is selected & establishes transfer path from SBR to CAR.

The truth table from the input logic circuit is shown in Table below.

BR field	Input			MUX 1		Load SBR
	I_1	I_0	T	S_1	S_0	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	X	1	0	0
1 1	1	1	X	1	1	0

Inputs I_1 and I_0 are identical to the bit values in the BR field.
The bits values for S_1 and S_0 are determined from the stated function and the path in the multiplexer that establishes the required transfer.
The subroutine register is loaded with the incremented value of CAR during a call microinstruction (BR = 01) provided that the status bit condition is satisfied (T = 1).

The truth table can be used to obtain the simplified boolean functions for the input logic circuit:

$$S_1 = I_1$$

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

$$L = I_1' T I_0.$$

UNIT-2 (b)

CENTRAL PROCESSING UNIT

Introduction:-

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

→ The CPU is made up of three major parts as shown in fig.

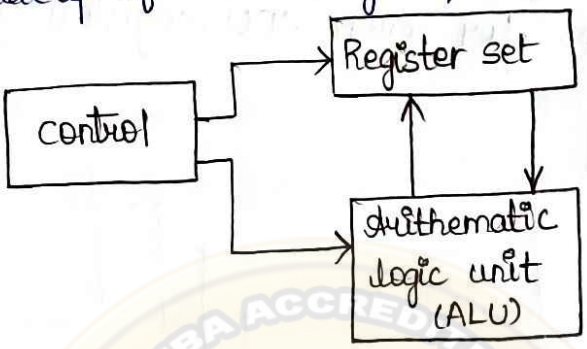


fig: Major Components of CPU

(i) The register set stores intermediate data used during the execution of instructions.

(ii) The arithematic logic unit (ALU) performs the required microoperations for executing the instructions

(iii) The control unit supervises the transfer of information among the registers and instructs the ALU as to which operation to perform.

→ The CPU performs a variety of functions dictated by the type of instructions that are incorporated in the computer.

→ computer architecture is defined as the structure and behaviour as seen by user. As this includes the instruction formats, addressing modes, the instruction set and the general organization of the CPU registers.

* General Register Organization :-

→ Memory locations are needed for storing pointers, counters, return addresses, temporary results and partial products during multiplication.

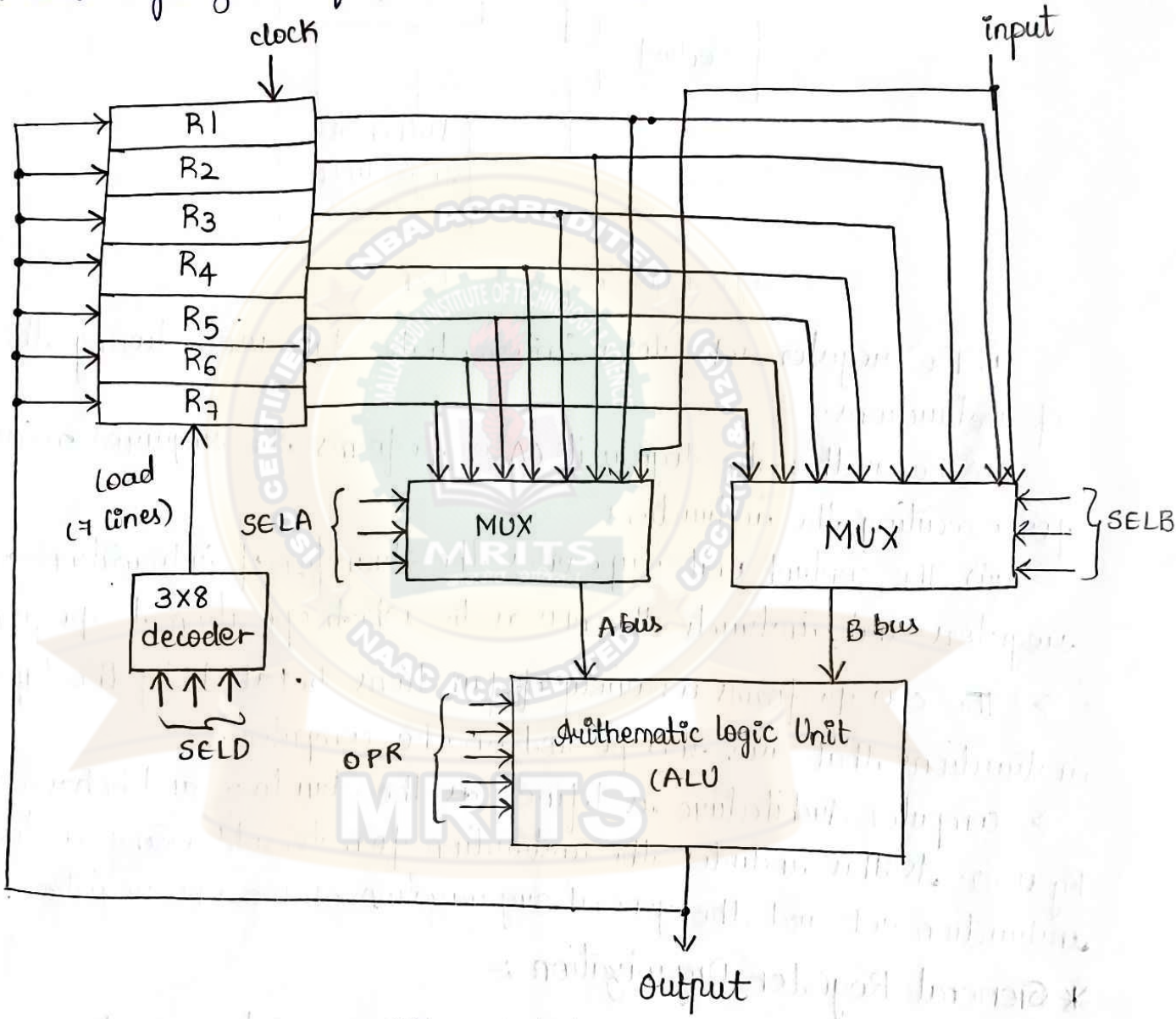
→ Having to refer to memory locations for such applications is time consuming because memory access is the most time consuming, operation in a computer.

→ It is more convenient & more efficient to store these intermediate values in processor registers.

→ When a large no. of registers are included in the CPU. It is most efficient to connect them through a common bus system. The registers communicate with each other not only for direct data transfers, but also while performing various microoperations.

→ Hence it is necessary to provide a common unit that can perform all the arithematic, logic and shift microoperations in the processor.

→ A bus organization for seven CPU registers shown in fig.



(a) Block diagram

3	3	3	5
SELA	SELB	SELD	OPR

(b) Control Word

fig: Register set with common ALU

→ The output of each register is connected to 2 multiplexers (MUX) to form the two buses A and B.
→ The selection lines in each multiplexer select one register or the input data for the particular bus.
→ The A and B buses form the inputs to a common arithematic logic unit (ALU).
→ The operation selected in ALU determines the arithematic or logic microoperation i.e; to be performed.
→ The result of microoperation is available for o/p data & also goes into i/p's of all registers.
→ The register receives the information from the o/p bus. is selected by a decoder.
→ The decoder activates one of the register load i/p's, thus providing a transfer path b/w the data in the o/p bus and the i/p's of the selected destination register.
→ The control unit that operates the CPU bus system directs the information flow through the registers & ALU by selecting various components in the system.
→ For example, to perform $R_1 \leftarrow R_2 + R_3$.

The control must perform binary selection variables to the following selector inputs

(i) MUX A selector (SELA): to place the content of R_2 into bus A.
(ii) MUX B selector (SELB): to place the content of R_3 into bus B.
(iii) ALU operation selector (OPR): to provide the arithematic addition A+B
(iv) Decoder destination selector (SELD): to transfer the content of the output bus into R_1.

→ The 4 control selection variables are generated in control unit & must be available at the beginning of a clock cycle. The data register from 2 source registers propagate through the gates in multiplexers & ALU, to the o/p bus & into i/p's of destination register, all during the clock cycle interval.

→ Then, the next clock transition occurs, the binary information from the output bus is transferred into R_1.

→ To achieve, a fast response time, the ALU is constructed with high speed circuits.

★ Control Word:-

→ There are 14 binary selection inputs in the unit and their combined value specifies a control word. The 14 bit control word is defined in fig(b).

→ It has four fields. Three fields contain 3 bits each and one field has 5 bits.

⇒ 3 bits of SELA : select a source register for the A input of ALU

⇒ 3 bits of SELB : select a register for the B input of ALU

⇒ 3 bits of SELD : select a destination register using decoder & its f load o/p's.

⇒ 5 bits of OPR : select one of the operations in the ALU.

→ The 14 bits control word when applied to the ~~instruction~~ selection inputs specify a particular microoperation. The encoding of the register selection is specified in following table 1.

Table 1 Encoding of register selection fields

Binary Code	SELA	SELB	SELD
000	Input	Input	None
001	R_1	R_1	R_1
010	R_2	R_2	R_2
011	R_3	R_3	R_3
100	R_4	R_4	R_4
101	R_5	R_5	R_5
110	R_6	R_6	R_6
111	R_7	R_7	R_7

→ The 3-bit binary code listed in 1st column of the table specify the binary code for each of the 3 fields.

→ The register selected by fields SELA, SELB and SELD is the one whose decimal number is equivalent to the binary number in the code.

→ When SELA or SELB is 000, the corresponding multiplexer selects the external input data.

→ When SELD=000, no destination register is selected but the contents of the output bus are available in the external output. The ALU provides arithematic & logic unit operations.

→ In addition, the CPU must provide shift operations. The shifter may be placed in the i/p of the ALU to provide a preshift capability or at the output of ALU to provide postshift capability. In some cases, the shift operations are included with ALU.

→ The function table for this ALU is listed in table 2.

Table 2 Encoding of ALU operations

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

Examples of Microoperations:-

→ For Example, the subtract microoperation given by the statement

$$R_1 \leftarrow R_2 - R_3$$

where, R_2 for the A i/p of the ALU

R_3 for the B i/p of the ALU

R_1 for the destination register & an ALU operation to subtract A-B

→ The control word specifies 4 fields & corresponding binary value for each field is obtained from encoding listed tables ① & ②

→ The binary control word for subtract microoperation is 010 011 001 00101 and is obtained as follows

Field	SELA	SELB	SELD	OPR
Symbol	R_2	R_3	R_1	SUB
control word	010	011	001	00101

→ Since the increment & transfer microoperations donot use the B i/p of ALU

Table 3 Examples of Microoperations for the CPU

Microoperation	Symbolic Designation				control Word
	SELA	SELB	SELD	OPR	
$R_1 \leftarrow R_2 - R_3$	R_2	R_3	R_1	SUB	010 011 001 00101
$R_4 \leftarrow R_4 \vee R_5$	R_4	R_5	R_4	OR	100 101 100 01010
$R_6 \leftarrow R_6 + 1$	R_6	–	R_6	INCA	110 000 110 00001
$R_7 \leftarrow R_1$	R_1	–	R_7	TSFA	001 000 111 00000
Output $\leftarrow R_2$	R_2	–	None	TSFA	010 000 000 00000
Output $\leftarrow$ Input	Input	–	None	TSFA	000 000 000 00000
$R_4 \leftarrow$ shl R_4	R_4	–	R_4	SHLA	100 000 100 11000
$R_5 \leftarrow 0$	R_5	R_5	R_5	XOR	101 101 101 01100

→ A register can be cleared to 0 with an Exclusive OR operation. $X \oplus X = 0$

→ The most efficient way to generate control words with a large no of bits is to store them in a memory unit.

→ A memory unit that stores control words is referred as control memory.

→ By reading consecutive control words from memory, it is possible to initiate the desired sequence of microoperations for the CPU. This type of control is referred as microprogrammed control.

* Instruction Formats :-

→ A computer will usually have a variety of instruction code formats. It is the function of the control unit within the CPU to interpret each instruction code and provide the necessary control functions needed to process the instruction.

→ The format of an instruction is usually depicted in a rectangular box symbolizing the bits of the instruction as they appear in memory words or in a control register. The bits of the instruction are divided into groups called fields. The most common fields found in instruction formats are

1. An Operation code field that specifies the operation to be performed.
2. An address field that designates a memory address or a processor register.
3. A mode field that specifies the way the operand or the effective address is determined.

→ The operation code field of an instruction is a group of bits that define various processor operations such as add, subtract, complement and shift. The bits that define the mode field of an instruction code specify a variety of alternatives for choosing the operands from the given address

→ Operation specified by computer instructions are executed on some data stored in memory or processor registers, Operands residing in processor registers are specified with a register address.

→ Computers may have instructions of several different lengths containing varying number of addresses. The number of address fields in the instruction format of a computer depends on internal organization of its registers. Most computers fall into one of three types of CPU organization:

(1.) Single accumulator organization

(2.) General register organization

(3.) Stack organization.

→ An example of an accumulator type organization is the basic computer where all operations performed with an implied accumulator register

ADD X

where X is address of operand. The ADD instruction in this case results in operation $AC \leftarrow AC + M[X]$

where AC → accumulator register

M[X] → symbolizes memory word located at address X.

→ An example of a general register type of organization where the instruction format of computer needs 3 register address fields

ADD R1, R2, R3

to denote the operation $R_1 \leftarrow R_2 + R_3$.

General register type computers employ two or three address fields in their instruction format. Each address field may specify processor register or a memory word ADD R_1, X

An instruction symbolized would specify the operation $R_1 \leftarrow R_1 + M[X]$. where R_1 is register & M other for memory address X.

→ Computers with stack organization would have PUSH and POP instructions which require address field

PUSH X

Thus the instruction will push the word at address X to top of the stack.

→ To illustrate the influence of number of addresses on computer programs, we will evaluate arithematic statement using zero, one, two or three address instructions

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

→ The symbols ADD, SUB, MUL, DIV are used to denote arithematic operations, MOV for transfer type operation and LOAD & STORE for transfer to and from memory and AC register.

* Three Address Instructions :-

→ Computer with three-address instruction formats can use each address field to specify either a processor register or an memory word.

→ The program in assembly language that evaluates

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

is shown below together with comments that explain the register transfer operation of each instruction.

ADD	R_1, A, B	$R_1 \leftarrow M[A] + M[B]$
ADD	R_2, C, D	$R_2 \leftarrow M[C] + M[D]$
MUL	X, R_1, R_2	$M[X] \leftarrow R_1 * R_2$

→ It is assumed that the computer has 2 processor registers R_1 and R_2. The symbol M[A] denotes the operand at memory address symbolized by A.

→ The advantage of three address format is that it results in short programs when evaluating arithematic expressions.

→ The disadvantage is that the binary coded instructions require too many bits to specify three-addresses.

* Two Address Instructions :-

→ Two-address instructions are the most common in commercial computers. Here each address field can specify either a processor register or a memory word.

→ The program to evaluate $X = (A+B) * (C+D)$ is as follows:

MOV	R_1, A	$R_1 \leftarrow M[A]$
ADD	R_1, B	$R_1 \leftarrow R_1 + M[B]$
MOV	R_2, C	$R_2 \leftarrow M[C]$
ADD	R_2, D	$R_2 \leftarrow R_2 + M[D]$
MUL	R_1, R_2	$R_1 \leftarrow R_1 * R_2$
MOV	*, R1	$M[X] \leftarrow R_1$

→ The MOV instruction moves or transfers the operands to and from memory and processor registers.

→ The first symbol listed in an instruction is assumed to be both source & destination where the result of the operation is transferred.

★ One-Address Instructions:-

→ One-Address Instructions use an implied accumulator (AC) register for all data-manipulation.

⇒ For multiplication and division there is a need for second register. However here we will neglect second register and assume that the AC contains the result of all operations.

LOAD	A	$AC \leftarrow M[A]$
ADD	B	$AC \leftarrow AC + M[B]$
STORE	T	$M[T] \leftarrow AC$
LOAD	C	$AC \leftarrow M[C]$
ADD	D	$AC \leftarrow AC + M[D]$
MUL	T	$AC \leftarrow AC * M[T]$
STORE	X	$M[X] \leftarrow AC$.

→ All operations are done b/w the AC register & memory operand. T is the address of temporary memory location required for storing the intermediate result.

★ Zero-Address Instructions:-

A stack organized computer doesnot use an address field for the instructions ADD and MUL. The PUSH and POP instructions, however need an address field to specify the operand that communicates with the stack.

$X = (A+B) * (C+D$ (TOS stands for top of stack).

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

The name "zero address" is given to this type of computer because of absence of address field in compilational instructions.

✱ ADDRESSING MODES:-

The operation field of an instruction specifies the operation to be performed and this operation must be performed on some data.

So each instruction need to specify data on which the operation is to be performed. But the operand (data) may be in accumulator, ~~(or)~~ general purpose register or at some specified memory location.

So, appropriate location (address) of data is need to be specified, and in computer, there are various ways of specifying the address of data.

These various ways of specifying the address of data are known as "Addressing Modes".

(or)

Addressing modes can be defined as the technique for specifying the address of the Operands.

→ Effective Address:-

An Computer, the address of operand i.e; the address where operand is actually found is known as effective address.

An addition to this, the two most prominent reason of why addressing modes are so important:

(1.) First, the way the operand data are chosen during program execution is dependent on the addressing mode of the instruction.

(2.) Second, the address field in a typical instruction format are relatively small and sometimes we would like to be able to reference a large range of locations, to achieve the large range of location in address field, a variety of addressing techniques has been employed. As they reduce the number of field in the addressing field of the instruction.

To understand the various addressing modes, first we need to understant basic operation cycle of computer. The Basic operation cycle has 3 main phases

(1) Fetch the instruction from memory
(2) Decode the instruction
(3) Execute the instruction

Program counter keeps track of instructions in the program stored in memory. PC holds the address of instruction to be executed next and is incremented each time an instruction is fetched from memory.

The decoding done in step 2 determines the operation to be perform-ed, the addressing mode of the instruction & the location of operands. The computer then executes the instruction and returns to step1 to fetch the next instruction in sequence.

An example of an instruction format with distinct addressing mode field is shown in fig.

Opcode	Mode	Address

fig:- Instruction format with mode field.

Where, opcode specifies the operation to be performed

> mode field is used to locate the operands needed for the operation,

Address field specifies the memory address or a processor registers.

Types of Addressing modes:-

Various types of addressing modes are

1. Implied Mode
2. Immediate Mode
3. Direct Address mode
4. Indirect Address mode
5. Register Mode
6. Register Indirect mode
7. Auto increment or Auto decrement mode
8. Relative address mode
9. Indexed addressing mode
10. Base register addressing mode.

1. Implied Mode:-

Implied addressing mode is known as "implicit or inherent" addressing mode is the addressing mode in which, no operand (register or memory location or data) is specified in the instruction.

For example, the instruction "Complement Accumulator" is an Implied mode instruction because the operand in accumulator register is implied in the definition of instruction.

In assembly language it is written as

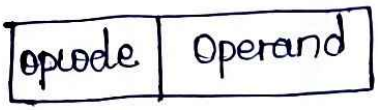
CMA : Take complement of content of AC

RLC : Rotate the content of Accumulator is an implied mode instruction.

② Immediate Addressing Mode :-

An immediate Addressing mode operand is specified in the instruction itself. An otherwords, an immediate mode instruction has an operand field rather than an address field, which contain actual operand to be used in conjunction with the operand specified in the instruction.

An this mode, the format of instruction is



For example,

ADD 05 ADD 05 to the content of accumulator

MOV 06 Move 06 to the accumulator.

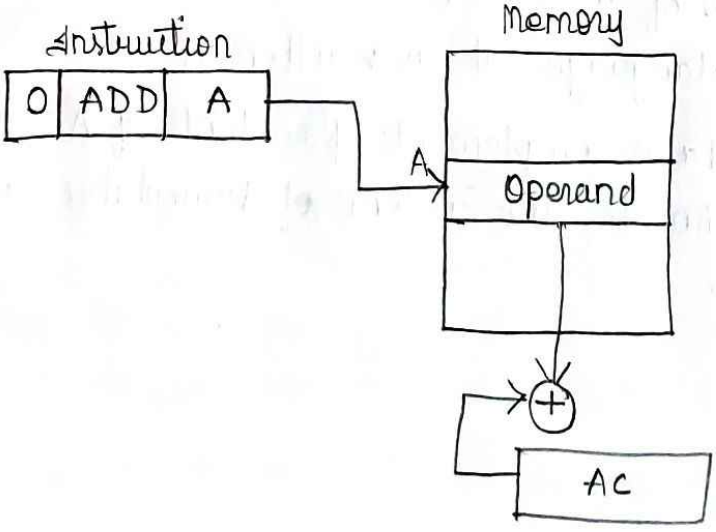
This mode is very useful for initialising the register to a constant value.

③ Direct Addressing Mode :-

Direct addressing mode is also known as Absolute Addressing mode. An this mode, the address of Operand specified in the instruction itself. That is, in this type of mode, the operand resides in memory and its address is given directly by the address field of the instruction.

The address field contain the effective address of operand i.e; EA = A

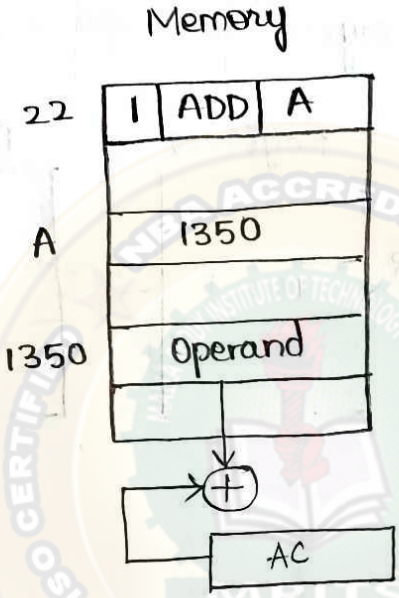
For example, ADD A → means add contents of cell A to accum -ulator.



④ Indirect addressing Mode:-

In this mode, the address field of instruction gives memory address where on, the operand is stored in memory. In this mode the address field of the instruction gives the address where the "effective address" is stored in memory. i.e; EA = (A)

For example, ADD (A) Means adds the content of cell pointed to contents of A to Accumulator



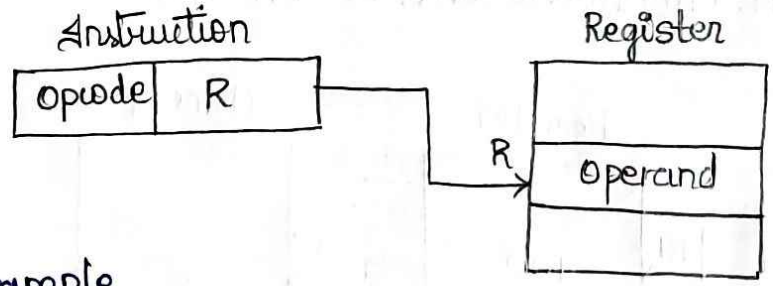
Thus AC ← M[M[A]]

(A) = 1350 = EA

⑤ Register addressing mode:-

In Register addressing mode, the operands are in registers that reside within the CPU. In this mode, instruction specifies a register in CPU, which contain the operand. It is similar to direct addressing mode, the only difference is that the address field refers to a register instead of memory location.

i.e; EA = R



For example

MOV AX, BX ; Move contents of Register BX to AX

ADD AX, BX Add the contents of Register BX to AX

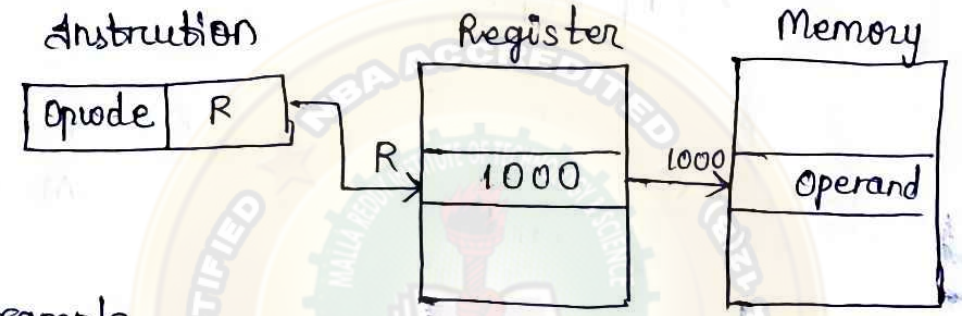
Here, AX, BX are used as register names of each of 16-bit register.

⑥ Register indirect addressing mode :-

In register indirect addressing mode, the instruction specifies a register in CPU whose contents give the operand in memory. In other words, the selected register contain the address of operand rather than the operand itself.

i.e; EA = (R)

Means, control fetches instruction from memory and then uses its address to access register and looks in Register (R) for effective address of operand in memory.



For example,

MOV AL, [BX]

code example:

MOV BX, 1000H

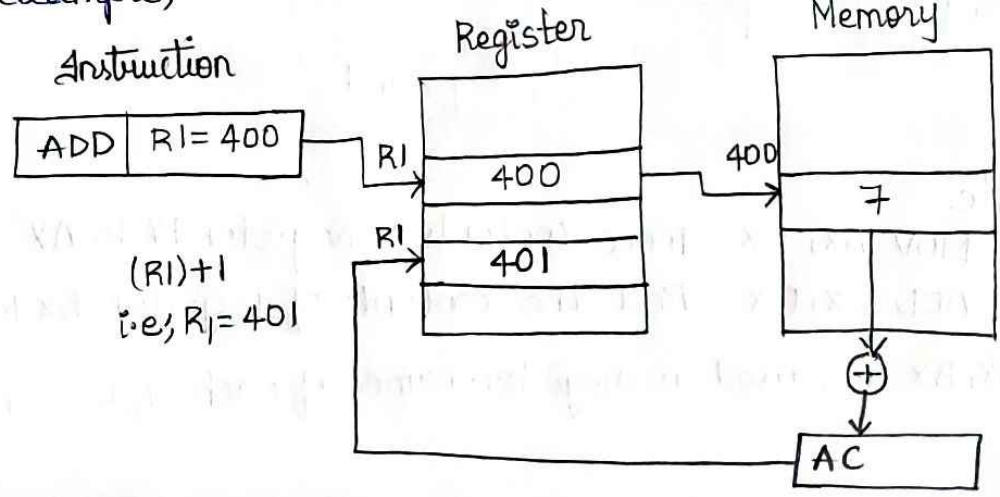
MOV 1000H, operand

⑦ Auto increment and auto decrement addressing modes :-

Auto increment addressing mode :-

Auto increment addressing mode are similar to register indirect addressing mode except that the register is incremented after its value is loaded at another location like accumulator (AC). In this case also the effective address is EA = (R).

For example,



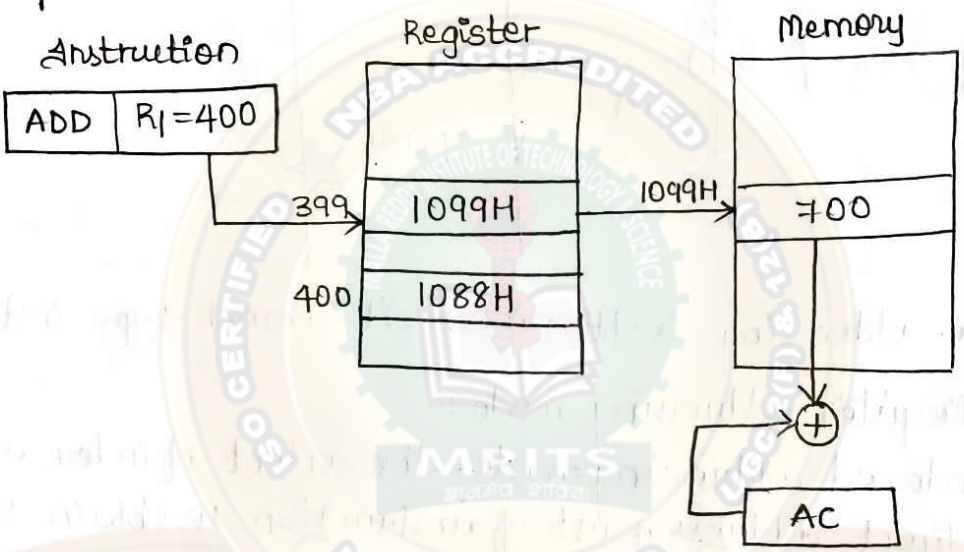
Here, the effective address is (R) = 400 and Operand in AC = 7. After loading R_1 is incremented by 1. It becomes 401.

In the autoincrement mode the R_1 is incremente to 401 after execution of instruction

Auto-decrement addressing mode:-

Auto decrement addressing mode is reverse of auto increment, as it is the register is decremented before the execution of instruction In this case, effective address EA = (R)-1

For example,



Here, in auto-decrement mode, the register R_1 is decremented to 399 prior to execution of instruction, means the operand is loaded to accumulator is of address 1099H in memory instead of 1088H.

EA = 1099H

AC = 700.

(8.) Relative Addressing mode:-

In relative addressing mode, the contents of program counter is added to the address part of instruction to obtain the effective address. In relative addressing mode, the address field of the instruction is added to implicitly reference register program counter to obtain effective address

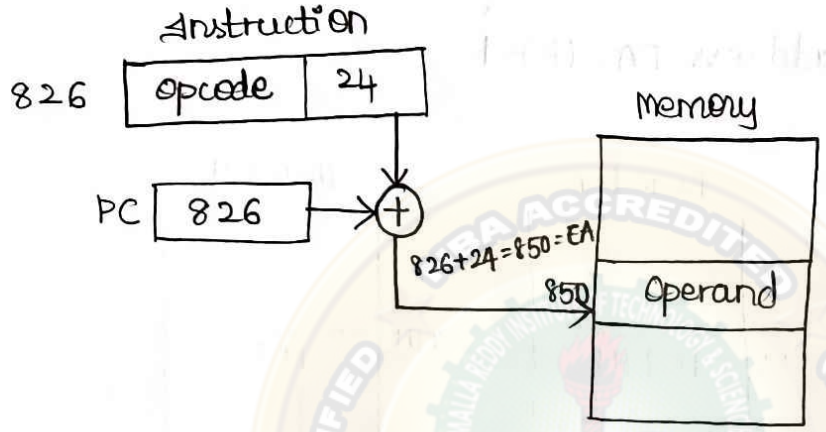
EA = A+PC

For example,

Assume that PC contains a no: 825 and the addrespart of the instruction contain a no: 24, then the instruction at location 825 is read from memory during fetch phase and the program counter is incremented by 1 to 826.

The effective address computation for relative address mode is

826+24 = 850



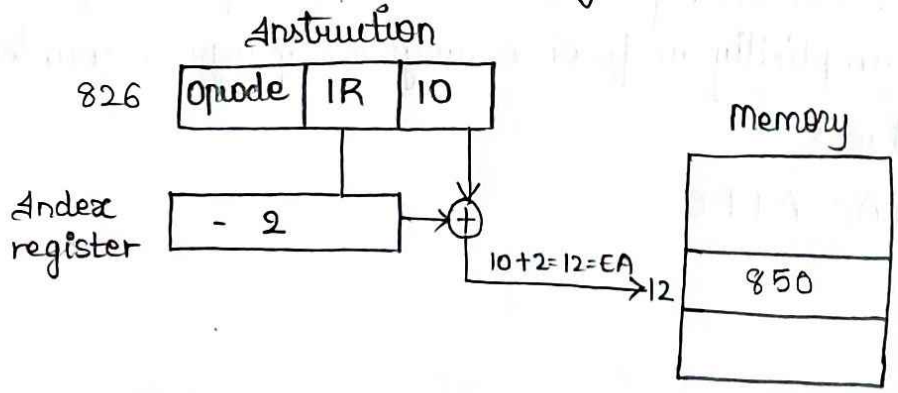
Relative addressing is often used with branch type instruction.

⑨ Index Register Addressing mode :-

An indexed addressing mode, the content of index register is added to direct address part of instruction to obtain the effective address. Means in the register indirect addressing field of instruction point to index register, which is a special CPU register that contain an indexed value & direct addressing field contain base address.

As indexed type instruction make sense that data array is in memory and each operand in the array is stored in memory relative to base address. The distance between the beginning address and the address of operand is the indexed value stored in indexed register.

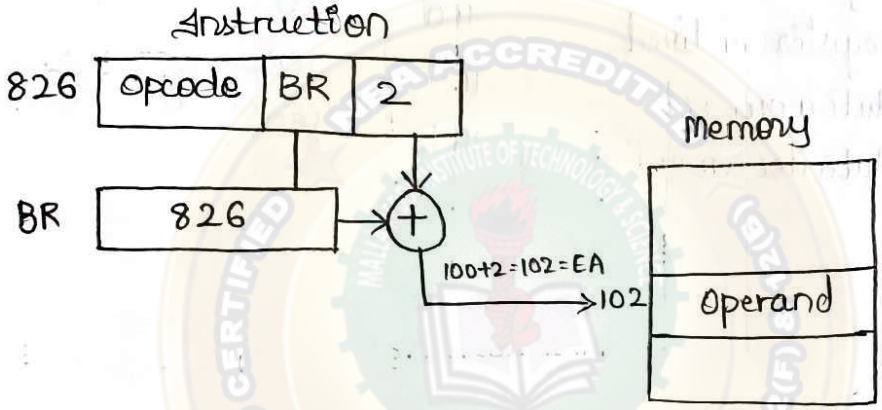
Thus in index addressing mode EA = A + Index



(10) Base register addressing mode:-

In this mode, the content of Base register is added to the direct address part of the instruction to obtain the effective address The register indirect address field point to the Base register and to obtain EA, the content of instruction register is added to direct address part of the instruction. This is similar to indexed addressing mode except that the register is now called as Base register instead of index register

EA = A + Base.



Numerical Example

To show the difference between various addressing modes on the instruction is defined in fig.

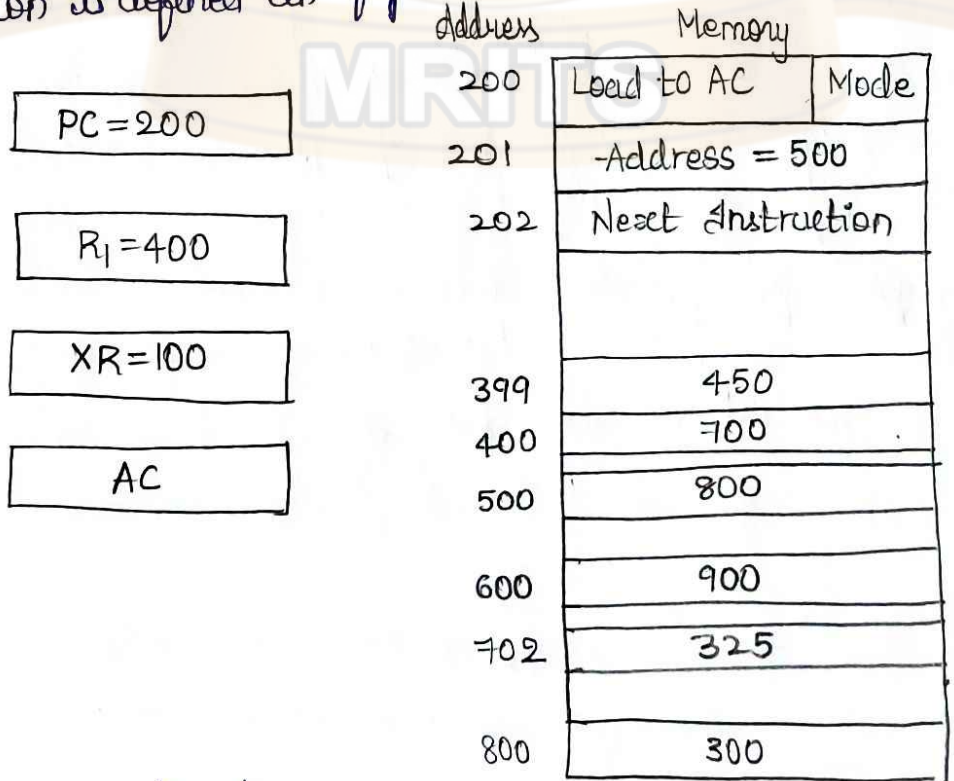


fig: Numerical example for addressing modes.

Tabular list of Numerical Example

Addressing Mode	Effective address	Content of AC
Direct address	500	800
Immediate Operand	201	500
Indirect Address	800	300
Relative address	702	325
Indexed address	600	900
Register	—	400
Register indirect	400	700
Autoincrement	400	700
Auto decrement	399	450

* Data transfer and Manipulation

* Data transfer: Instructions :-

→ Data transfer instructions move data from one place in the computer to another without changing the data content.

→ The most common transfers are b/w memory and processor registers, and b/w processor registers and Input or output and b/w the processor registers itself or themselves.

→ The Eight different data transfer instructions are listed in the table.

Table: Typical Data transfer instructions

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

Each instruction is a mneumonic symbol. It must be realized that different computers use different mneumonics for the same instruction name.

→ The load instruction is used to transfer ofor memory to a processor register, usually an accumulator.

→ The store instruction is used to transfer data to memory.

→ The Move instruction is used to transfer data from one register to other.

→ It has also been used for data transfers between CPU registers and memory or between two memory words.

→ The Exchange instruction swaps information between two registers or a register and a memory word.

→ The input and Output instructions transfer data among processor registers and input or output terminals.

→ The push and pop instructions transfer data between processor registers and memory stack.

→ Some assembly language conventions modify the mnemonic symbol to differentiate between the different addressing modes.

→ For example, the mnemonic for load immediate becomes LDI. and consider the Load to accumulator instruction when used with 8 different addressing modes.

Table: Eight addressing modes for the Load instructions

Mode	Assembly Convention	Register transfer
Direct address	LD ADR	AC ← M[ADR]
Indirect address	LD @ADR	AC ← M[M[ADR]]
Relative address	LD $ ADR	AC ← M[PC+ADR]
Immediate operand	LD # NBR	AC ← NBR
Index addressing	LD ADR(X)	AC ← M[ADR+XR]
Register	~~LD ADR~~ (LDR1	AC ← R1
Register indirect	LD(R1)	AC ← M[R1]
Autoincrement	LD(R1)+	AC ← M[R1], R1 ← R1+1

where, ADR ~~stands~~ for Address
NBR is number or operand
X is index register
R_1 is a processor register
AC is accumulator register
@ symbolizes indirect address
$ address makes the address relative to PC
# immediate-mode instruction.

★ Data Manipulation Instructions :-

Data Manipulation Instructions perform operations on data and provide the computational capabilities for the computer. The data manipulation instructions in a typical computer are usually divided into three basic types:

1. Arithmetic instructions
2. Logical and bit manipulation instructions
3. Shift instructions.

1. Arithmetic Instructions :-

→ The four basic arithmetic operations are addition, subtraction, multiplication and division. Most computers provide instructions for all four operations. Some small computers have only addition and possibly subtraction instructions. The multiplication & division must then be generated by means of software subroutines.

→ The increment instruction adds 1 to the value stored in a register or memory word.

→ The decrement instruction subtracts 1 from a value stored in a register or memory word.

→ The instruction "add with carry" performs the addition on two operands plus the value of the carry from the previous computation

→ Similarly, the "subtract with borrow" instruction subtracts 2 words and a borrow which may have resulted from a previous subtract operation.

→ The negate instruction forms the 2's complement of a number, effectively reversing the sign of an integer when represented in the signed 2's complement form.

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 comp)	NEG

② 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 AND instruction is used to clear a bit or a selected group of bits of an operand.

→ The OR instruction is used to set a bit or a selected group of bits of an operand.

→ Similarly, the XOR instruction is used to selectively complement bits of an operand.

→ Individual bits, such as a carry can be cleared, set or complement -ed with appropriate instructions.

Name	Mnemonic
clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-OR	XOR
clear carry	CLRC
Set carry	SETC
complement carry	COMC
Enable Interrupt	EI
Disable Interrupt	DI

③ 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, arithematic shifts or rotate type operations.

→ In either case the shift may be to the right or to the left.

→ The logical shift inserts 0 to the end bit position. The end position is the leftmost bit for shift right and the rightmost bit position for the shift left.

→ The arithematic shift-right instruction must preserve the sign bit in the leftmost bit position. The sign bit is shifted to the right together with the rest of the number, but the sign bit itself remains unchanged. This is a shift-right operation with the end bit remaining the same. The arithematic shift-left instruction inserts 0 to the end position and is identical to logical shift left instruction.

→ 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.

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

* 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 PC 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 the fetch cycle with the program counter containing the address of instruction next in sequence.

→ On the other hand, a program control type of instruction, when executed, may change the address value in program counter and cause the flow of control to be altered.

→ Program Control Instructions specify conditions for altering the content of program counter, while data transfer and manipulation instructions specify conditions for data-processing operations.

→ The change in value of the program counter as a result of the execution of a program control instruction causes a break in the sequence of instruction execution.

→ This is an important feature in digital computers, as it provides control over the flow of program execution & a capability for branching to different program segments.

Some typical program control instructions are listed in Table.

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

⇒ Branch and Jump instruction will be conditional or unconditional.
↳ The unconditional branch instruction causes a branch to be specified address without any conditions.
↳ The conditional branch instruction specifies a condition such as branch if positive or branch if zero.

⇒ The skip instruction doesnot need an addressfield & therefore a zero-address instruction.

⇒ The call and return instructions are used in conjunction with subroutines.

⇒ The compare instructions performs a subtraction between two operands, but the result of the operation is not retained.

⇒ The test instruction performs the logical AND of two operands and updates certain status bit conditions are set as a result of operation.

★ Status Bit Conditions :-

It is sometimes convenient to supplement the ALU circuit in the CPU with a status register where status bit conditions can be stored for further analysis. Status bits are also called condition-code bits or flag bits.

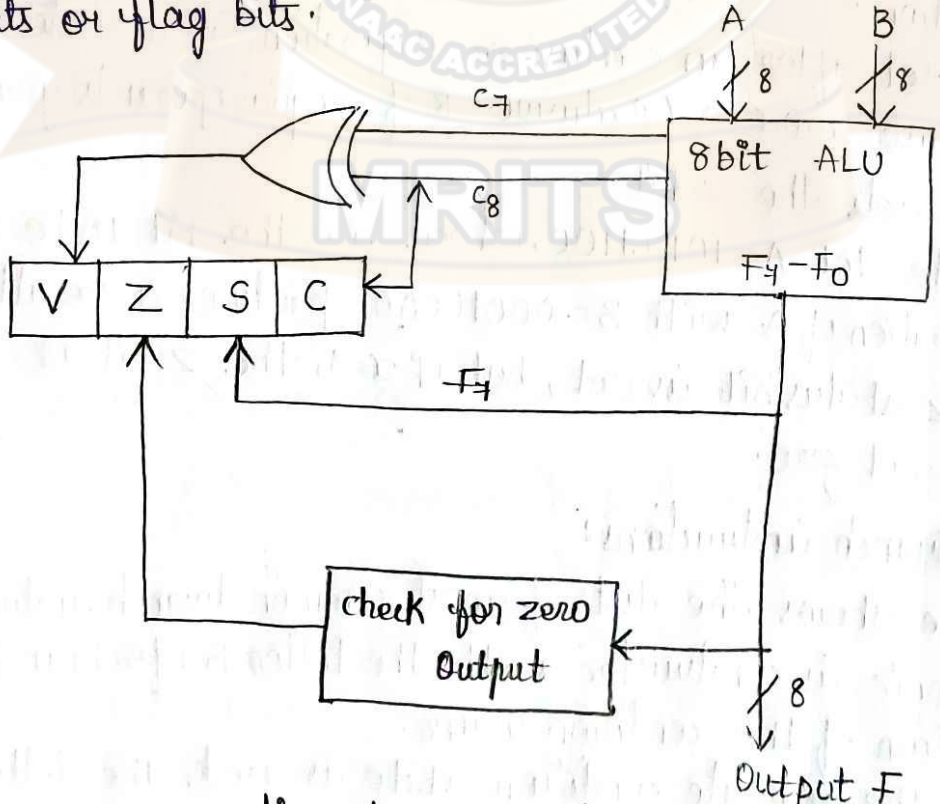


fig: Status register bits.

The figure shows the block diagram of an 8-bit ALU with a 4-bit status register. The 4 status bits are symbolized by C, S, Z and V. The bits are set or cleared as a result of the operation performed in the ALU.

↳ Bit C (carry) is set to 1 if the end carry C_8 is 1. It is cleared to 0 if the carry is 0.

↳ Bit S (Sign) is set to 1 if the highest order bit F_7 is 1. It is set to 0 if the bit is 0.

↳ Bit Z (zero) is set to 1 if the output of ALU contains all 0's. It is cleared to 0 otherwise. In otherwords, Z=1 if the O/p ~~flag~~ is zero and Z=0 if the O/p is not zero.

↳ Bit V (overflow) is set to 1 if the exclusive-OR of the last two carries is equal to 1 and cleared to 0 otherwise. This is the condition for an overflow when negative numbers are in 2's complement. For the 8-bit ALU, V=1 if the output is greater than +127 or less than -128.

→ Status bits can be checked after an ALU operation to determine certain relationships that exist b/w the values of A and B.

→ If bit V is set after the addition of 2 signed numbers, it indicates an overflow condition.

→ If Z is set after an exclusive-OR operation, it indicates that A=B. This is because $x \oplus x = 0$ (Exclusive-OR of 2 equal operands gives an all 0's result which sets the Z-bit.

→ for example let A=101x1100, where x is the bit to be checked. The AND operation of A with B=00010000 produces a result 000x 0000 If x=0, the Z status bit is set, but if x=1, the Z bit is cleared since the result is not zero.

★ Conditional Branch instructions:-

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

→ when the opposite condition state is used, the letter N is inserted to define the 0 state.

Table: Conditional Branch Instructions

Mnemonic	Branch Condition	Tested Condition
BZ	Branch if zero	Z=1
BNZ	Branch if not zero	Z=0
BC	Branch if carry	C=1
BNC	Branch if no carry	C=0
BP	Branch if plus	S=0
BM	Branch if minus	S=1
BV	Branch if overflow	V=1
BNV	Branch if no overflow	V=0

Unsigned Compare conditions (A-B)

BHI	Branch if higher	A>B
BHE	Branch if higher or equal	A≥B
BLO	Branch if lower	A<B
BLOE	Branch if lower or equal	A≤B
BE	Branch if equal	A=B
BNE	Branch if not equal	A≠B

Signed Compare conditions (A-B)

BGT	Branch if greater than	A>B
BGE	Branch if greater or equal	A≥B
BLT	Branch if less than	A<B
BLE	Branch if less or equal	A≤B
BE	Branch if equal	A=B
BNE	Branch if not equal	A≠B

→ A conditional branch instruction is a branch instruction that may or may not cause a transfer of control depending on the value of stored bits in the PSRC (processor status register).

→ Each conditional branch instructions tests a different combination of status bits for a condition.

→ If the condition is true, control is transferred to the effective address (PC ← Address). If the condition is false, the program continues with the next instruction (PC ← PC+1)

↳ 'C' represents the carry or borrow after arithematic addition or subtraction

↳ 'N' represents the leftmost bit of the result of operation i.e; sign bit.

↳ 'V' is for overflow i.e; if the sign of the result is changed (inverted)

↳ 'Z' is for zero i.e; to check whether the result of an operation is zero (Z=1) or not zero (Z=0)

→ Some branch instructions are a combination of compare and condit-ional branch instructions. They are run after the compare instruction has performed the comparision and status bits are updated.

→ Different status bits are checked for signed and unsigned numbers.

→ It is important that if A≥B is complement of A<B and A≤B is complement of A>B. That means if we know the condition of status bits for one, the condition for the other complementary relation is obtained by complement.

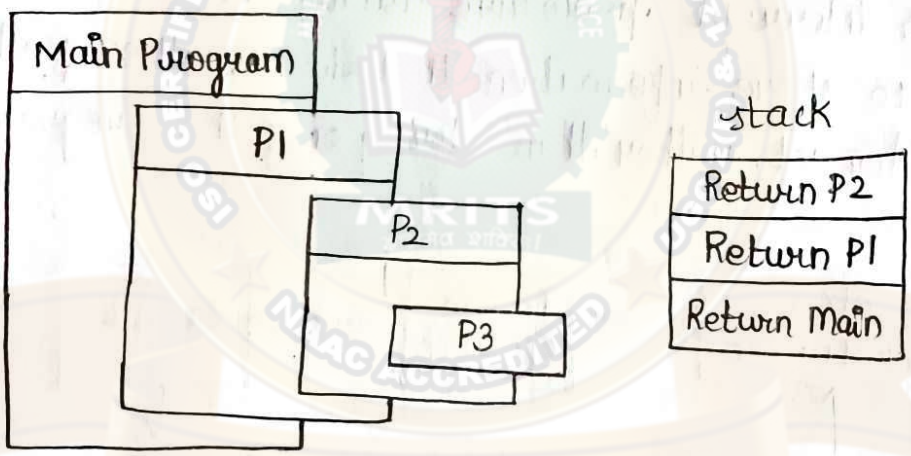
★ Subroutine Call and Return:-

→ A subroutine is a self-contained sequence of instructions that performs a given task (or) a computational task. It is also called a procedure call.

→ When a subroutine is called, the starting address of the subroutine is stored in the PC and the instruction following the current instruction is temporary stored elsewhere.

→ When the subroutine (block of code) is executed, the return is made to the mainprogram by loading the PC with the old value.

→ Instruction following the subroutine call is called continuation point and the corresponding address is called the return address.
→ It is actually a low level form of functions in C++.
→ Subroutine can also be called with another procedure/subroutine.
→ The final instruction of every procedure/subroutine must be return to the calling program.
→ The return address can be stored in memory, register or stack.
→ Stack is preferred because of its ease of access when we need to call a subroutine inside another subroutine. In that case that the return address at the TOS (top of stack) is always to the program which called the current subroutine.



For calling subroutine

SP ← SP − 1	Decrement stack pointer
M[SP] ← PC	Store return address on stack
PC ← Effective address	Transfer control to subroutine

For Return

PC ← M[SP]	Transfer return address to PC
SP ← SP+1	Increment stack pointer

★ Program Interrupts :-

→ An interrupt transfers control from a program that is currently running to another program as a result of externally or internally generated request.

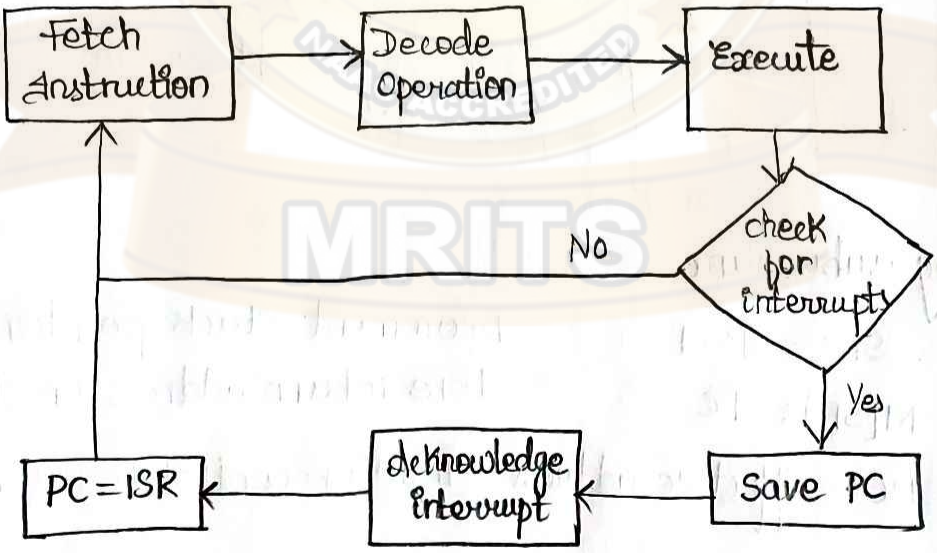
→ The procedure for servicing the interrupt in this case is called the interrupt service routine (ISR).

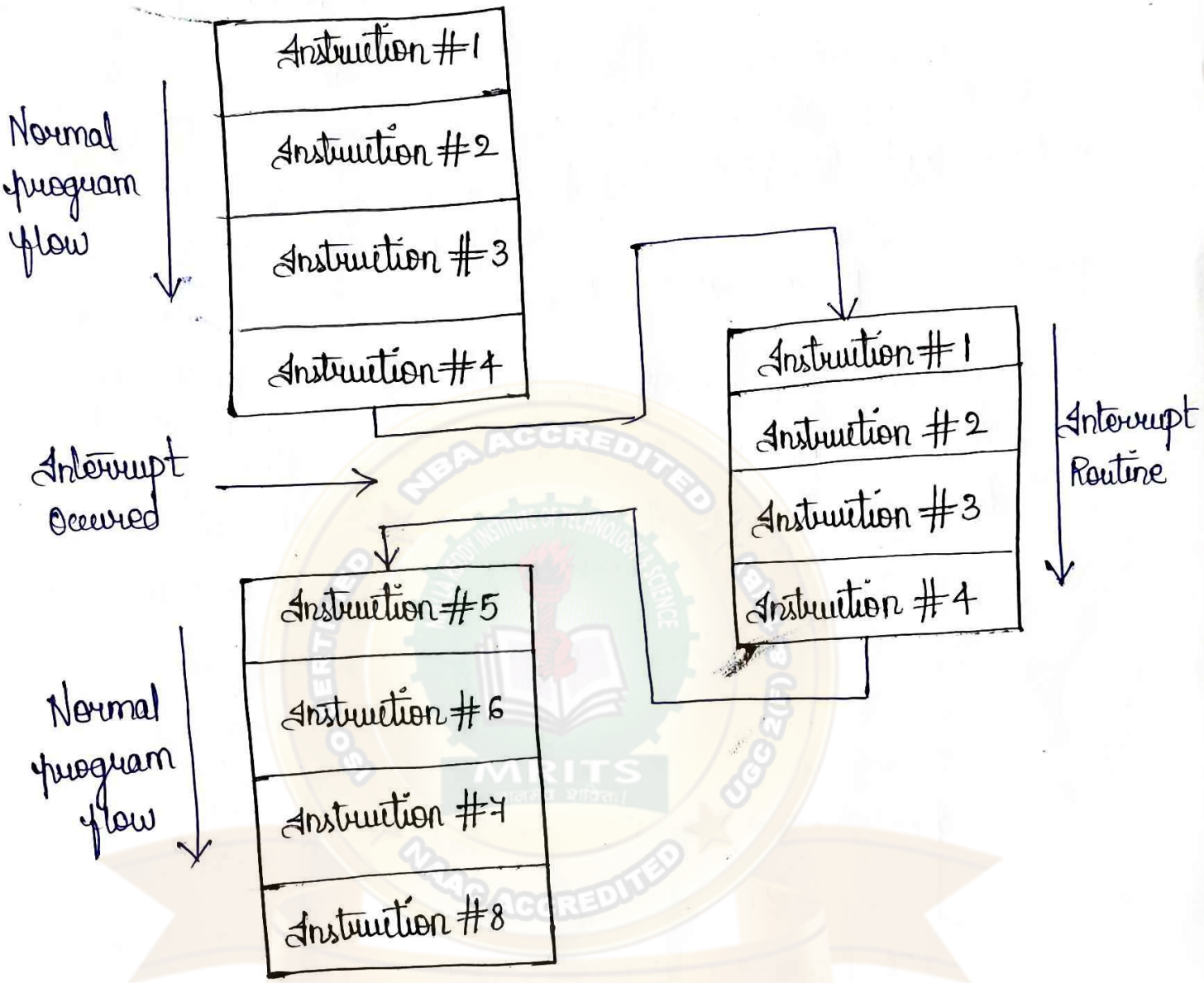
→ The interrupt procedure is quite similar to a subroutine call except for three variations:-

(1.) The interrupt is usually initiated by an external or internal signal rather than from the execution of an instruction.

→(2.) The address of the interrupt service program is determined by the hardware rather than from the address field of an instruction.

(3.) An interrupt procedure usually stores all the information necessary to store information that defines all or part of the contents of the register set, rather than storing only the program counter.




Instruction #1
Instruction #2
Instruction #3
Instruction #4
Normal program flow
Interrupt occured
Instruction #1
Instruction #2
Instruction #3
Instruction #4
Interrupt Routine
Instruction #5
Instruction #6
Instruction #7
Instruction #8
Normal program flow

→ These three procedural concepts are ~~classified~~ clarified further below.

→ After the a program has been interrupted and the service routine been executed, the CPU must return to exactly the same state that it was when the interrupt occured. Only if this happens will the interrupted program be able to resume exactly as if nothing had happened.

→ The state of CPU at the end of execute cycle is determined from

- → The content of the program counter
- → The content of all processor registers
- → The content of certain status conditions.

→ The collection of all status bit conditions in the CPU is sometimes called a program status word.

→ The PSW is stored in a seperate Hardware register and contains the status information that characterizes the state of the CPU.

★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

① External Interrupts :-

External Interrupts come from input or output devices, from timing devices, from a circuit monitoring the power supply or from any other External source.

Examples that cause external interrupts are I/O device requesting transfer of data, I/O device finished transfer of data,

elapsed time of an event, or power failure.

→ Timeout interrupt may result from a program that is an endless loop & thus exceeded its time allocation.

→ Power failure interrupt may have as its service routine a program that transfers the complete state of the CPU into a non-destructive memory in the few milliseconds before power ceases.

② Internal Interrupts:-

Internal interrupts arise from illegal or erroneous use of an instruction or date. Internal interrupts are also called traps.

Examples of Interrupts caused by internal error conditions are register overflow, attempt of to divide by zero, an invalid operation code, stack overflow and protection violation.

These error conditions usually occur as a result of a premature termination of the instruction execution.

The service program that processes the internal interrupt determines the corrective measure to be taken.

③ Software Interrupt:-

Software interrupt is a special call instruction that behaves like an interrupt rather than a subroutine call. It is initiated by executing an instruction.

It can be used by the programer to initiate an interrupt procedure at any desired point in the program.

The most common use of software interrupt is associated with a supervisor user mode to the supervisor mode.

A program written by a user must run in the user mode. When an input-output transfer is required, the supervisor mode is requested by means of a supervisor call instruction. This instruction causes a software interrupt that stores the old CPU state and brings in a new PSW that belongs to the supervisor mode. The calling program must pass information to the O.S in order to specify the particular task requested.

UNIT-3

Data Representation, Computer Arithmetic:

Data Types:-

→ Binary information in digital computers is stored in memory or processor registers. Registers contain either data or control information.

→ Data are numbers and other binary-coded information that are operated on to achieve required computational results.

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

1) Numbers used in arithmetic computations
2) letters of the alphabet used in data processing
3) other discrete (Special) symbols used for specific purposes.

Number systems:-

Radix:- A number system of base, or radix, r is a system that uses distinct symbols for 'r' digits.

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

Decimal:- The decimal number system in every day use employs the radix 10 system. The 10 symbols are 0,1,2,3,4,5,6,7,8,9

→ The string of digits 724.5 is interpreted to represent the quantity.

$$7\times10^2 + 2\times10^1 + 4\times10^0 + 5\times10^{-1}.$$

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

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

$$1\times2^5 + 0\times2^4 + 1\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0 = 45.$$

Octal Hexa decimal:- The octal means radix 8, & hexadecimal means radix 16. The eight symbols of the octal system are 0,1,2,3,4,5,6,7.

→ The 16 symbols of the hexadecimal system are 0,1,2,3,4, 5,6,7,8,9,A,B,C,D,E & F.

→ when we represent hexadecimal digits, the symbols A,B,C,D, E,F correspond to the decimal numbers 10,11,12,13,14,15, respectively.

→ A number in radix 'r' can be converted to the familiar decimal system by forming the sum of the weighted digits.

ex:- octal 736.4 is converted to decimal as:

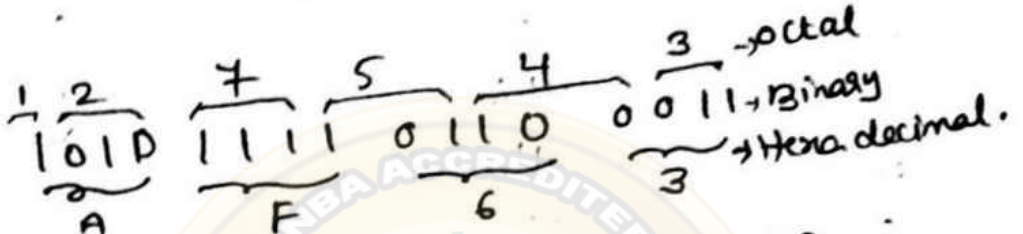
$$(736.4)_8 = 7\times8^2 + 3\times8^1 + 6\times8^0 + 4\times8^{-1}$$
$$= 7\times64 + 24 + 6 + 4/8 = (478.5)_{10}.$$

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

$$(F3)_{16} = F\times16 + 3 = 15\times16 + 3 = (243)_{10}.$$

Octal & Hexadecimal Numbers:- The conversion of binary, octal & Hexadecimal representation plays an important part in digital computers.

-> Since $2^3 = 8$ & $2^4 = 16$, each octal digit corresponds to three binary digits & each hexadecimal digit corresponds to four binary digits.



Binary, octal & Hexadecimal conversion.

Table: Binary Coded - octal Numbers

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

Table: Binary-coded Hexadecimal Numbers.

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

Decimal Representation:- The binary number system is the most natural system for a computer, but people are accustomed to the decimal system. One way to solve this conflict is to convert all input decimal numbers into

binary numbers, let the computer perform all arithmetic operations in binary & then convert the binary results back to decimal for the human user to understand.

Binary code:- A binary code is a group of n bits that assume up to 2^n distinct combinations of 1's & 0's with each combination representing one element of the set that is being coded.

→ For example, a set of four elements can be coded by a 2-bit code with each element assigned one of the following bit combinations: 00, 01, 10 & 11.

→ A set of eight elements requires a 3-bit code, a set of 16 elements requires a 4-bit code & so on.

BCD:- The binary-coded decimal (BCD) is the abbreviation. It is very important to understand the difference between the conversion of decimal numbers into binary & the binary coding of decimal numbers.

→ For example, when converted to a binary number, the decimal number 99 is represented by the string of bits 1100011. But when represented in BCD, it becomes 10011001.

→ The only difference between a decimal number represented by the familiar digit symbols 0, 1, 2 ... 9 & the BCD symbols 0001, 0010, ... 1001 is in the symbols used to represent the digits the number itself is exactly the same.

Table: Binary-coded decimal (BCD) Numbers.

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

↑ code for one decimal digit ↓

Complements:- Complements are used in digital computers for simplifying the subtraction operation & for logical manipulation. There are two types of complements for each base 'r' system:

1) The r's complement
2) (r-1)'s complement.

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

(r-1)'s complement :- Given a number N in base 'r' having 'n' digits, the (r-1)'s complement of N is defined as $(r^n - 1) - N$.

i) 9's complement :- For decimal numbers $r = 10$ & $r - 1 = 9$, so the 9's complement of N is $(10^n - 1) - N$.

ex :- 1) 9's complement of 546700 is

$$999999 - 546700 = 453299.$$

2) 9's complement of 12389 is $99999 - 12389 = 87610$.

→ with $n = 4$ we have $10^4 = 10000$ & $10^4 - 1 = 9999$. It follows that the 9's complement of a decimal number is obtained by subtracting each digit from 9.

ii) 1's complement :- For binary numbers, $r = 2$ & $r - 1 = 1$, so the 1's complement of N is $(2^n - 1) - N$.

→ The 1's complement of a binary number is formed by changing 1's into 0's & 0's into 1's.

ex :- 1) The 1's complement of 1011001 is 0100110.

2) The 1's complement of 0001111 is 1110000.

(r's) complement :-

i) 10's complement :- comparing with the (r-1)'s complement, the r's complement is obtained by adding 1 to the (r-1)'s

Complement, since $r^n - N = [(r^n - 1) - N] + 1$.

ex:- The 10's complement of the decimal 2389 is

$$9999 - 2389 = 7610.$$

$$\Rightarrow 7610 + 1 = 7611.$$

→ This is obtained by adding '1' to the 9's complement value.

ii) 2's complement:- The 2's complement of a binary value is obtained by adding '1' to the 1's complement value.

ex:- The 2's complement of a binary 101100 is

$\Rightarrow$ 010011 (1's complement)

$\Rightarrow$ 010011 + 1 (adding 1 to 1's complement)

$\Rightarrow$ 010100.

Fixed Point Representation:- positive integers, including zero, can be represented as unsigned numbers. However a to represent negative integers, we need a notation for negative values.

→ Because of hardware limitations, computers must represent everything with 1's & 0's, including the sign of a number.

→ The convention is to make the sign bit equal to '0' for positive numbers and '1' for negative numbers.

→ Binary point:- In addition to the sign, a number may have a binary (or decimal) point. The position of the binary point

is needed to represent fractions, integers, or mixed-integer fraction numbers. There are 2 ways of specifying the position of the binary point in a register:

1) Fixed-point representation
2) Floating point representation.

Integer Representation:- When an integer binary number is positive, the sign is represented by '0' & when the number is negative, the sign is represented by '1' but the rest of the number may be represented in one of three posible ways:

i) Signed - magnitude representation
ii) signed - 1's complement representation
iii) signed - 2's complement representation.

→ The signed-magnitude representation of a negative number consists of the magnitude & a negative sign.
→ In the other two representations, the negative number is represented in either the 1's or 2's complement of its positive value.
→ As an example consider the signed number 14 stored in 8-bit register. +14 is represented by a sign bit of '0' in the left-most position followed by the binary equivalent of 14: 00001110
→ Each of the eight bits of the register must have a value & therefore 0's must be inserted in the most significant positions following the sign bit. There is only one way to represent +14.
→ There are 3 different ways to represent -14 with eight bits. they are

i) In signed-magnitude representation 1 0001110.

ii) In signed-1's complement representation 1 1110001

iii) In signed-2's complement representation 1 1110010.

→ The signed-magnitude representation of -14 is obtained from +14 by complementing only the sign bit.

→ The signed-1's complement representation of -14 is obtained by complementing all the bits of +14, including the sign bit.

→ The signed-2's complement representation is obtained by taking the 2's complement of the positive number, including its sign bit.

Arithmetic Addition:- The addition of two numbers in the signed-magnitude system follows the rules of ordinary arithmetic. If the signs are the same, we add the two magnitudes & give the sum the common sign. If the signs are different, we subtract the smaller magnitude from the larger & give the result the sign of the larger magnitude.

2's complement addition:- The rule for adding numbers in the signed 2's complement system does not require a comparison or subtraction, only addition & complementation.

→ The procedure is as follows: Add the two numbers, including their sign bits, & discard any carry out of the sign (leftmost) bit position.

→ Negative numbers must initially be in 2's complement & it is assumed that if the sum obtained after the addition is negative, it is in 2's complement form.

ex:- 1) $+6$ 00000110
$+13$ 00001101
$+19$ 00010011

2) -6 11111010
-13 11110011
-19 11101101

Arithmetic subtraction:- Subtraction of two signed binary numbers when negative numbers are in 2's complement form is as follows: Take the 2's complement form of the subtrahend (including the sign bit) & add it to the minuend (including the sign bit). A carry out of the sign bit position is discarded.

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

ex:- $(-6) - (-13) = +7$.

→ In binary with eight bits this can be written as:

11111010 − 11110011.

→ The subtraction is changed to addition by taking the 2's complement of the (−13) to (+13).

→ In binary 11111010 + 00001101 = 100000111.

→ Removing the end carry, we obtain the correct answer

00000111 ⇒ +7.

Decimal fixed-point representation:- The representation of decimal numbers in registers is a function of the binary code used to represent a decimal digit.

→ A 4-bit decimal code requires four flip-flops for each decimal digit.

→ The representation of 4385 in BCD requires 16 flip-flops four flip-flops for each digit. The number can be represented in a register with 16 flip-flops as follows:

0100 0011 1000 0101.

Floating point Representation:- The floating-point representation has two parts. The first part represents a signed, fixed-point number called the mantissa. The second part designates the position of the decimal (or binary) point called the exponent. The fixed-point mantissa may be a fraction or an integer.

ex:- The decimal number +6132.789 is represented in floating-point with a fraction & an exponent as follows:

Fraction	Exponent
+0.6132789	+04

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

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

$$m \times \beta^e$$

$$\Rightarrow 0.6152789 \times 10^4.$$

Here m is signified number

β is mantissa which holds 10

e is the exponent which holds 4.

Fraction:- A floating point binary number is represented in a similar manner except that it uses base 2 for the exponent.

ex:- The binary number +1001.11 is represented with an 8-bit fraction and 6-bit exponent as follows:

Fraction	Exponent
01001110	000100.

→ The fraction has a '0' in the leftmost position to denote positive. The floating point number is equivalent to

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

Normalization:- A floating-point number is said to be normalized if the most significant digit of the mantissa is non-zero.

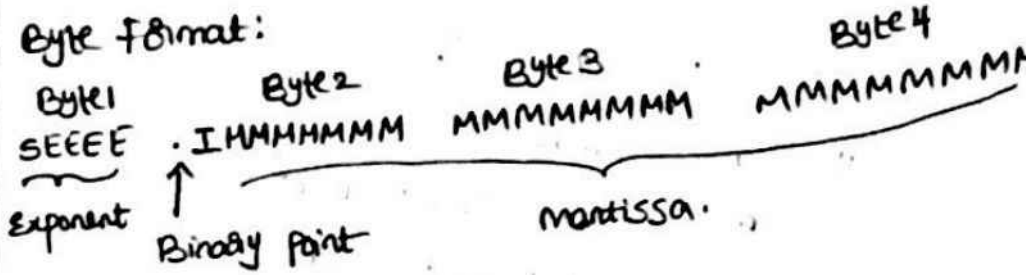
ex:- The decimal number 350 is normalized but 0035 is not.

→ The number is normalized only if its leftmost digit is non-zero.

→ Two main standard forms of floating-point numbers are from the following organizations that decide standards; ANSI (American National standards Institute) & IEEE (Institute of Electrical & Electronic Engineers).

→ The ANSI 32-bit floating-point numbers in byte format with example are:

Byte format:



S = Size of Mantissa

E = Exponent Bits in 2's complement

M = Mantissa Bits.

Computer Arithmetic

Addition & subtraction:- For floating-point operations, most computers use the signed-magnitude representation for the mantissa. In this topic we develop the addition and subtraction algorithms for data represented in signed-magnitude and again for data represented in signed-2's complement.

Addition & subtraction with signed-magnitude data:- The representation of numbers in signed-magnitude is familiar because it is used in everyday arithmetic calculations.

→ We designate the magnitude of the two numbers by A and B. When the signed numbers are added or subtracted, we find that there are eight different conditions to consider, depending on the sign of the numbers and the operation performed.

→ These conditions are listed in the first column of the table.

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).

Table 10.1 : Addition & subtraction of signed-magnitude Numbers.

→ The other columns in the table show the actual operation to be performed with the magnitude of the numbers. The last column is needed to prevent a negative zero. In other words, when two equal numbers are subtracted, the result should be +0 not -0.

→ The algorithms for addition & subtraction are derived from the table can be stated as follows (the words inside parantheses should be used for the subtraction algorithm).

Addition (subtraction) algorithm:- When the signs of A & B are identical (different), add the two magnitudes and attach the sign of A to the result.

→ When the signs of A & 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.

→ The two algorithms are similar except for the sign comparison. The procedure to be followed for identical signs in the addition algorithm is the same as for different signs in the subtraction algorithm & vice versa.

Addition & subtraction with signed-2's complement data:-

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

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

→ when two numbers of n digits each are added and the sum occupies n+1 digits, we say that an overflow occured.

→ ~~The result of an overflow on the sum of two signed 2's complement numbers~~

→ An overflow can be detected by inspecting the last 2 carries out of the addition. when the two carries are applied to an exclusive-OR gate, the overflow is detected when the output of the gate is equal to 1.

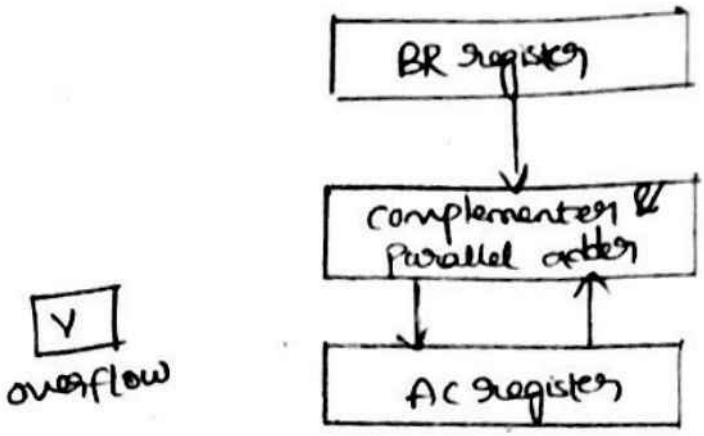


Fig 10.3 : Hardware for signed 2's complement addition & subtraction

→ We have the A register AC (accumulator) and the B register BR. The leftmost bit in AC & BR represent the sign bits of the numbers. The two sign bits are added or subtracted together with the other bits in the complementer & parallel adder.

→ The overflow flip-flop 'V' is set to '1' if there is an overflow.

→ The output carry in this case is discarded.

→ The algorithm for adding & subtracting two binary numbers in signed-2's complement representation is shown below diagram.

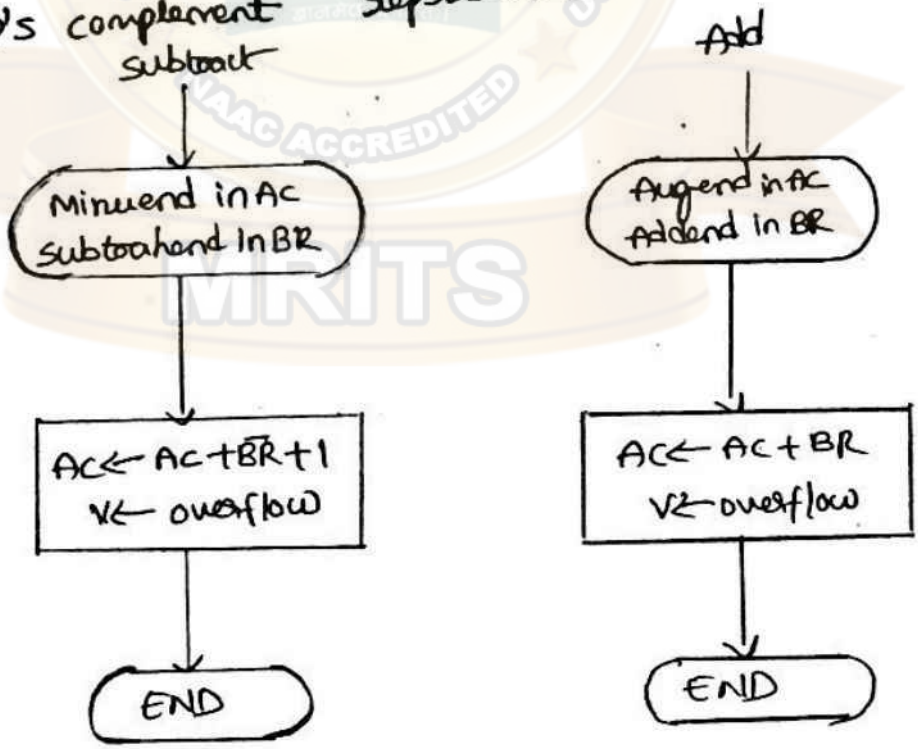


fig 10.4 Algorithm for adding & subtracting numbers in signed 2's complement representation.

→ The sum is obtained by adding the contents of AC & BR (including their signs). The overflow bit 'V' is set to '1', if the exclusive-OR of the last two carries is 1, & it is cleared to '0' otherwise.

→ The subtraction operation is accomplished by adding the content of AC to the 2's complement of BR. Taking the 2's complement of BR has the effect of changing a positive number to negative, & vice versa.

→ An overflow must be checked during this operation because the two numbers added could have the same sign.

Multiplication Algorithms:- Multiplication of two fixed-point binary numbers in signed-magnitude representation, is done with a process of successive shift & add operations.

→ The sign of the product is determined from the signs of the multiplicand & multiplier. If they are alike, the sign of the product is positive. If they are unlike, the sign of the product is negative.

Hardware Implementation for Signed-magnitude data:-

The hardware for multiplication consists of the equipment shown below:

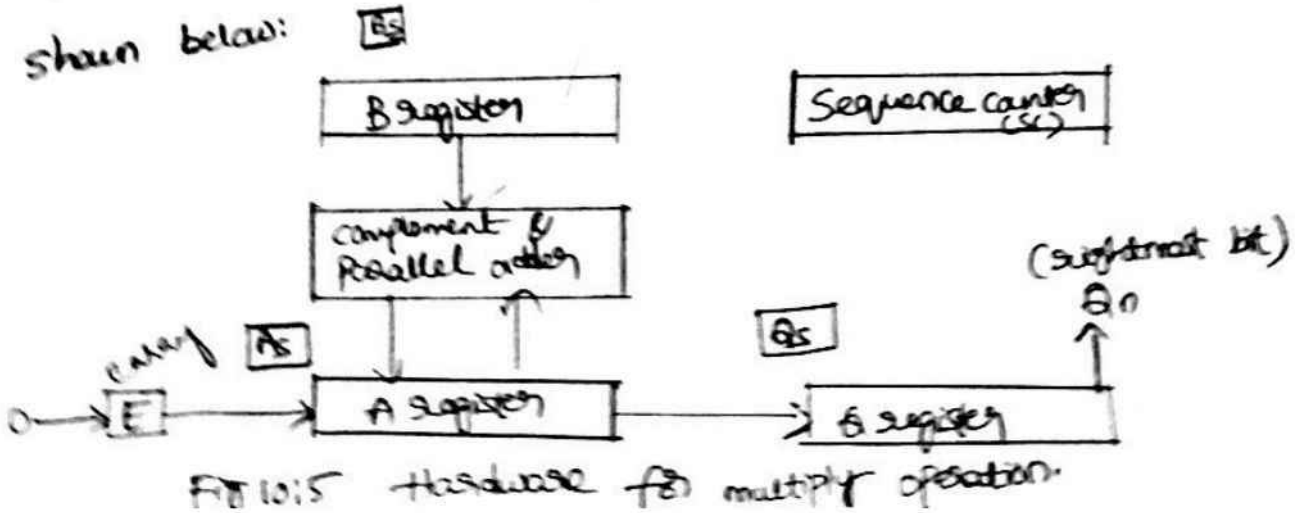


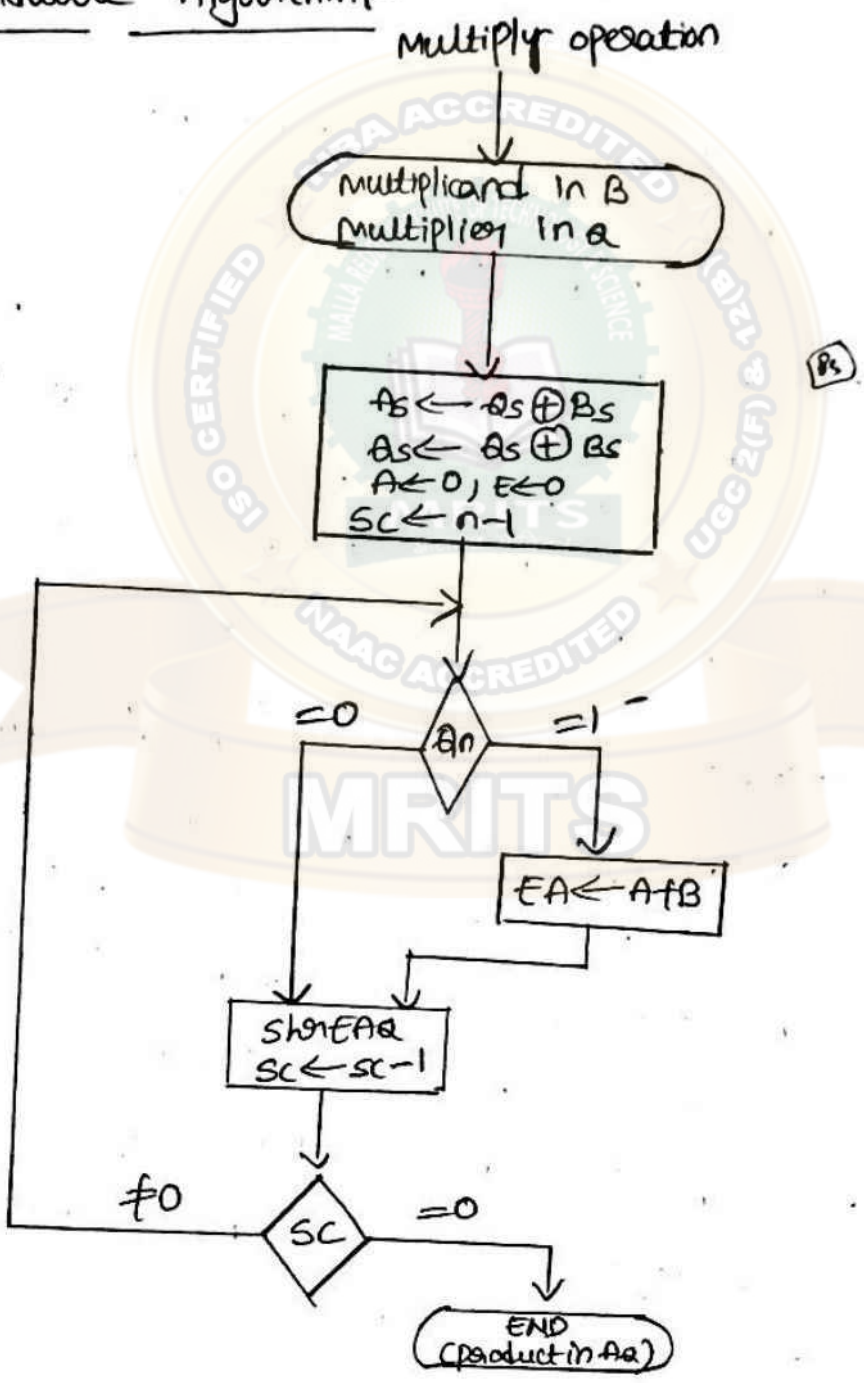
Fig 10.5 Hardware for multiply operation.

-> The multiplier is stored in the Q-register & its sign in Q_s

-> The sequence counter SC is initially set to a number equal to the number of bits in the multiplier. The counter is decremented by 1 after forming each partial product.

-> When the content of the counter reaches to zero, the product is formed and the process stops.

Hardware Algorithm:-



$x \times y = z$

B Q A A_s

Q_n SC

E = Carry

(+) (-) = ·

$A_s \leftarrow Q_s + B_s$

$Q_s \leftarrow Q_s + B_s$

$$\begin{array}{r} 1 \\ 2\overline{)25 \times 35} \\ \hline 1\ 25 \\ 6\ 5\times \\ \hline 775 \end{array}$$

Fig 10.6 Flowchart for multiply operation

→ Initially the multiplicand is in B & the multiplier in Q. Their corresponding signs are in Bs & Qs respectively. The signs are compared, & both A and Q are set to correspond to the sign of the product since a double-length product will be stored in registers A & Q.

→ Registers A & E are cleared & the sequence counter SC is set to a number equal to the number of bits of the multiplier.

→ After the initialization, the lower-order bit of the multiplier in Q_n is tested. If it is '1', the multiplicand in B is added to the present partial product in A. If it is '0' nothing is done.

→ Register EAQ is then shifted once to the right to form the new partial product. The sequence counter is decremented by 1 & its new value checked. if it is not equal to zero, the process is repeated & a new partial product is formed. The process stops when SC = 0.

Booth Multiplication Algorithm:— Booth algorithm gives a procedure for multiplying binary integers in signed-2's complement representation. It operates on the fact that strings of 0's in the multiplier require no addition but just shifting, & a string of 1's in the multiplier from bit weight 2^k to weight 2^m can be treated as $2^{k+1} - 2^m$.

→ As in all multiplication schemes, Booth algorithm requires examination of the multiplier bits & shifting of the partial product

→ Prior to the shifting, the multiplicand may be added to the partial product, subtracted from the partial product, or left uncharged according to the following rules:

1) The multiplicand is subtracted from the partial product upon encountering the first least significant 1 in a string of 1's in the multiplier.

2) The multiplicand is added to the partial product upon encountering the first 0 in a string of 0's in the multiplier.

3) The partial product does not change when the multiplier bit is identical to the previous multiplier bit.

→ The algorithm works for positive or negative multiplier's in 2's complement representation.

→ In the flow diagram 10.8, AC & the appended bit Q_{n+1} are initially cleared to 0 and the sequence counter SC is set to a number n equal to the number of bits in the multiplier.

→ The two bits of the multiplier in Q_n & Q_{n+1} are inspected. If the two bits are equal to 10, it means that the first '1' in a string of 1's has been encountered. This requires a subtraction of the multiplicand from the partial product in AC.

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

→ When the two bits are equal, the partial product does not change. An overflow cannot occur because the addition & subtraction of the multiplicand follow each other.

→ The next step is to shift right the partial product & the multiplier (including bit Qn+1). This is an arithmetic shift right (ashr) operation which shifts AC & QR to the right & leaves the sign bit in AC unchanged. The sequence counter is decremented & the computational loop repeated n times.

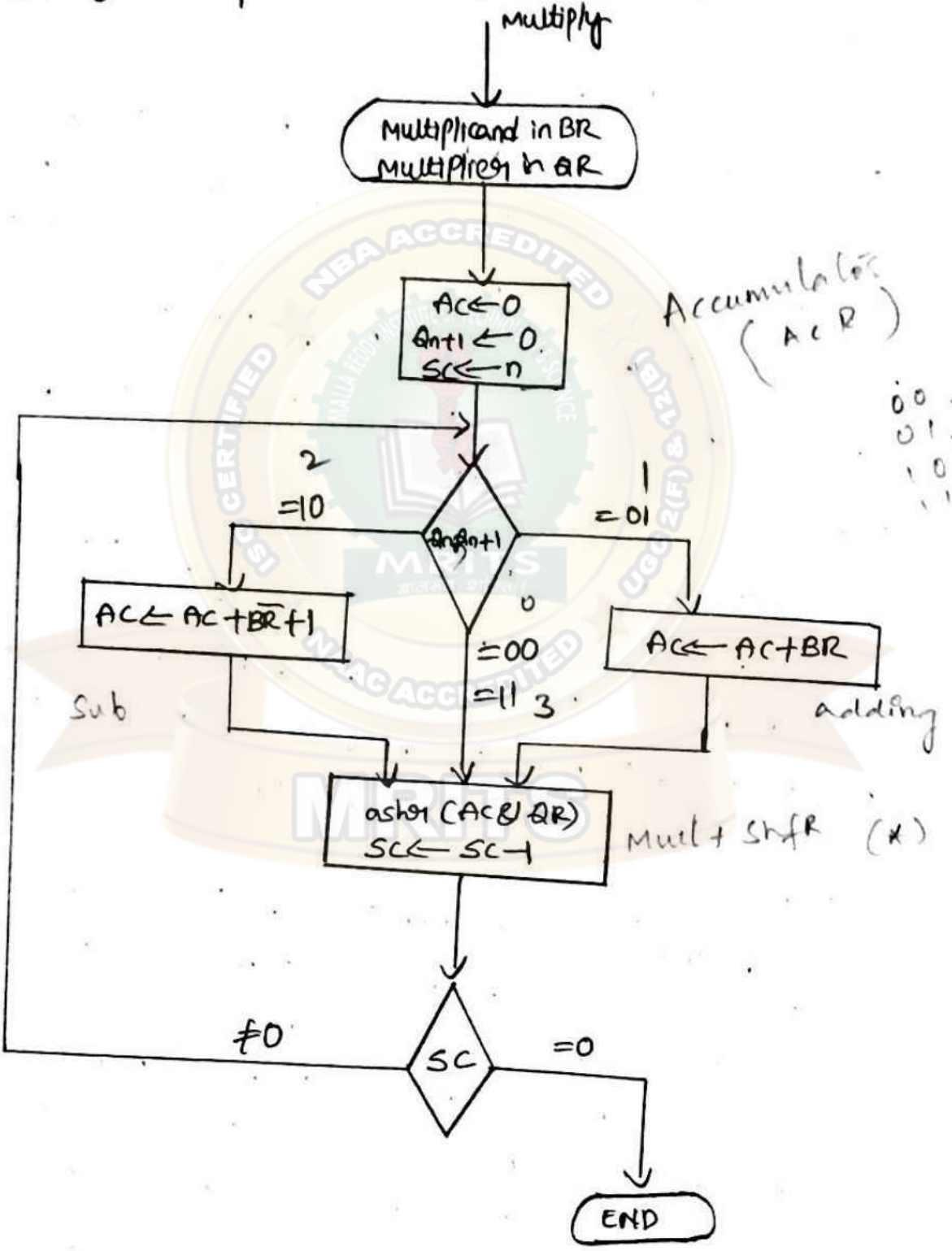


Fig 10-8 Booth algorithm for multiplication of signed-2's complement numbers.

Division Algorithms :- Binary division is simpler than decimal division because the quotient digits are either 0 or 1 & there is no need to estimate how many times the dividend or partial remainder fits into the divisor.

Hardware Algorithm :-

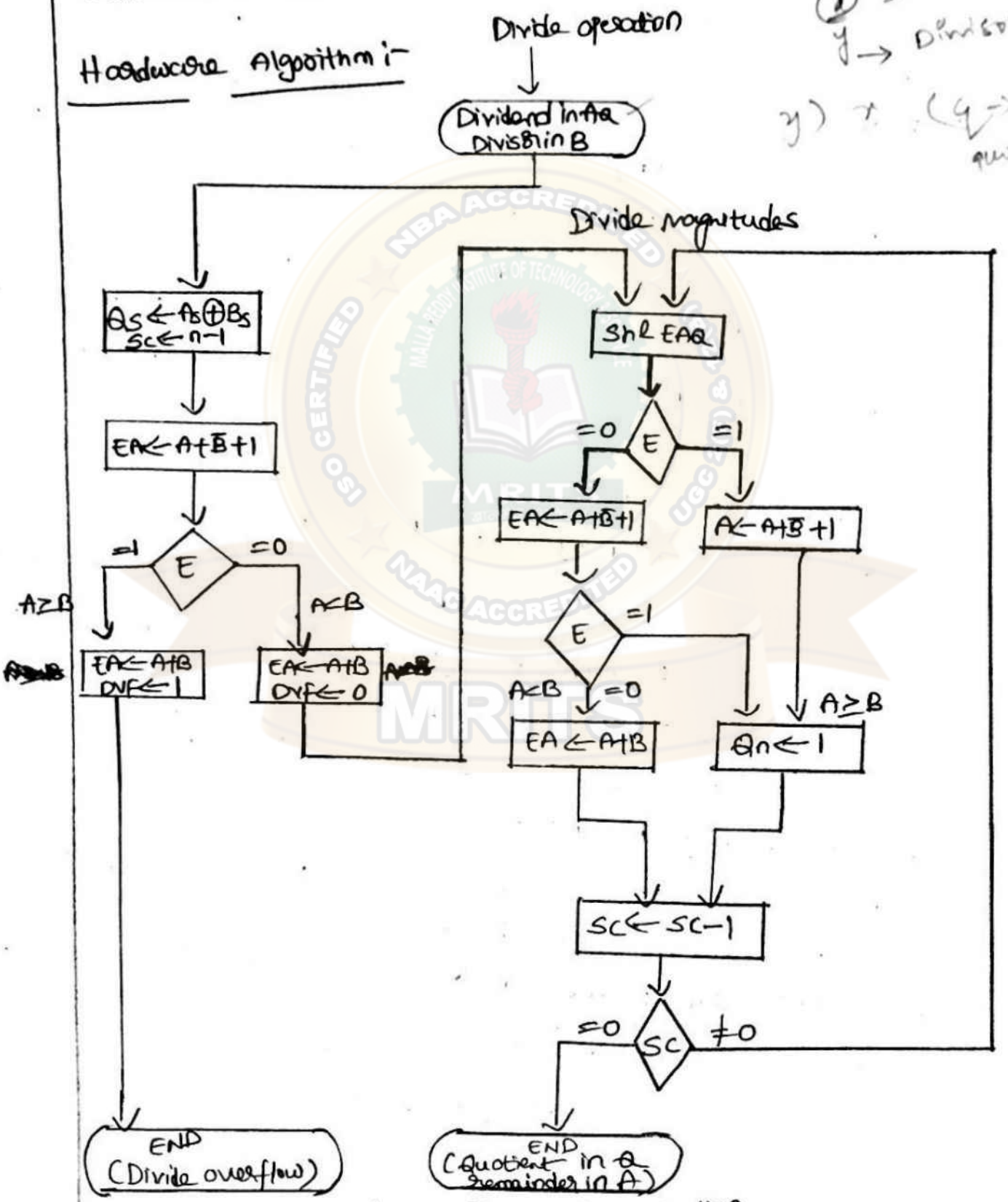


Fig 10.13 Flowchart for divide operation

→ In the above algorithm the dividend is in A & Q & the divisor in B. The sign of the result is transferred into Qs to be part of the quotient. A constant is set into the sequence counter SC to specify the number of bits in the quotient.

→ As in multiplication, we assume that operands are transferred to registers from a memory unit that has words of n bits. Since an operand must be stored with its sign, one bit of the word will be occupied by the sign & the magnitude will consists of n-1 bits.

→ A divide-overflow condition is tested by subtracting the divisor in B from half of the bits of the dividend stored in A. If A ≥ B, the divide-overflow flip-flop DVF is set & the operation is terminated prematurely.

→ If A < B, no divide overflow occurs so the value of the dividend is restored by adding B to A.

→ The division of the magnitudes starts by shifting the dividend in AQ to the left with the high-order bit shifted into E.

→ If the bit shifted into E, is 1, we know that EA > B because EA consists of a 1 followed by n-1 bits while B consists of only n-1 bits.

→ In this case, B must be subtracted from EA & 1, inserted into Q for the quotient bit. Since register A is missing the high-order bit of the dividend, its value is $EA - 2^{n-1}$

→ Adding to this value the 2's complement of B resulting

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

→ The carry from this addition is not transferred to E if we want E to remain a '1'.

→ If the shift-left operation inserts a 0 into E, the divisor is subtracted by adding its 2's complement value & the carry is transferred into E. if E=1, it signifies that A≥B; therefore, Q_n is set to '1'. If E=0, it signifies that A<B, & the original number is restored by Adding B to A. In the latter case we leave a 0 in Q_n.

→ This process is repeated again with register A holding the partial remainder. After n-1 times, the quotient magnitude is formed in register Q & the remainder is found in register.

Floating Point Arithmetic operations:-

Basic considerations:- A floating point number in computers registers consists of two parts: a mantissa m & an exponent e.

$$m \times 10^e$$

→ The decimal number 537.25 is represented in a register with m = 53725 & e=3 & is interpreted to represent the floating-point number

$$.53725 \times 10^3.$$

Register configuration:- The register configuration for floating-point operations is quite similar to the layout for fixed-point operations. As a general rule, the same registers & adders used for fixed-point arithmetic are used for processing the mantissas. The difference lies in the way the exponents are handled.

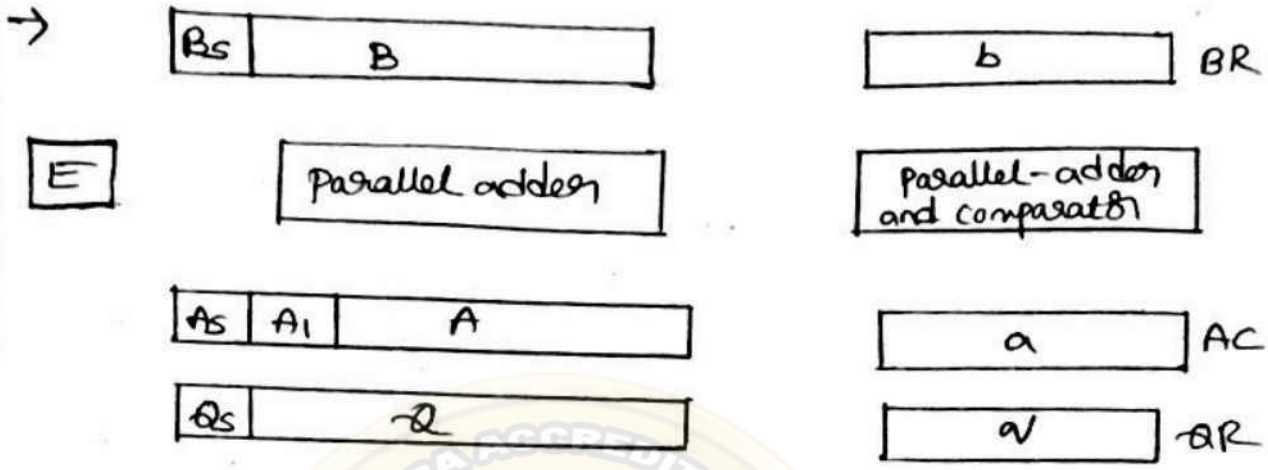


Fig 10.14 Registers for floating-point arithmetic operations.

→ In the above figure, there are three registers, BR, AC & QR. Each register is subdivided into two parts.

→ The mantissa part has the same uppercase letter symbols as in fixed-point representation. The exponent part uses the corresponding lowercase letter symbol.

→ It is assumed that each floating-point number has a mantissa in signed magnitude representation & a biased exponent. Thus a AC has a mantissa whose sign is in As and a magnitude that is in A.

→ The exponent is in the part of the register denoted by the lowercase letter symbol 'a'. The diagram shows explicitly the most significant bit of A, labeled by A1. The bit in this position must be a '1' for the number to be normalized. The symbol AC represents the entire register, the concatenation of As, A and a.

→ similarly register BR is subdivided into Bs, B & b, and QR into Qs, Q, and q. A parallel-adder adds the two mantissas & transfers the sum into A, & the carry into E.

→ A separate parallel-adder is used for the exponents.
→ The numbers in the mantissa will be taken as a fraction, so the binary point is assumed to reside to the left of the magnitude part.
→ The numbers in the registers are assumed to be initially normalized. After each arithmetic operation, the result will be normalized.

Addition & subtraction:- During addition or subtraction, the two floating-point operands are in AC and BR. The sum or difference is formed in AC. The algorithm can be divided into 4 consecutive parts:-

1) Check for zero's
2) Align the mantissas.
3) Add or subtract the mantissas.
4) Normalize the result.

→ A floating-point number that is zero cannot be normalized. if this number is used during the computation, the result may also be zero. The alignment of the mantissas must be carried out prior to their operation.
→ The flowchart, represents if BR is equal to zero, the operation is terminated, with the value in the AC being the result. If AC is equal to zero, we transfer the content of BR into AC & also complement its sign if the numbers are to be subtracted. If neither number is equal to zero, we proceed to align the mantissas.
→ The magnitude comparator attached to exponents a & b provides three outputs that indicate their relative magnitude

→ If the two exponents are equal, we go to perform the arithmetic operation. If the exponents are not equal the mantissa having the smaller exponent is shifted to the right and its exponent incremented. This process is repeated until the two exponents are equal.

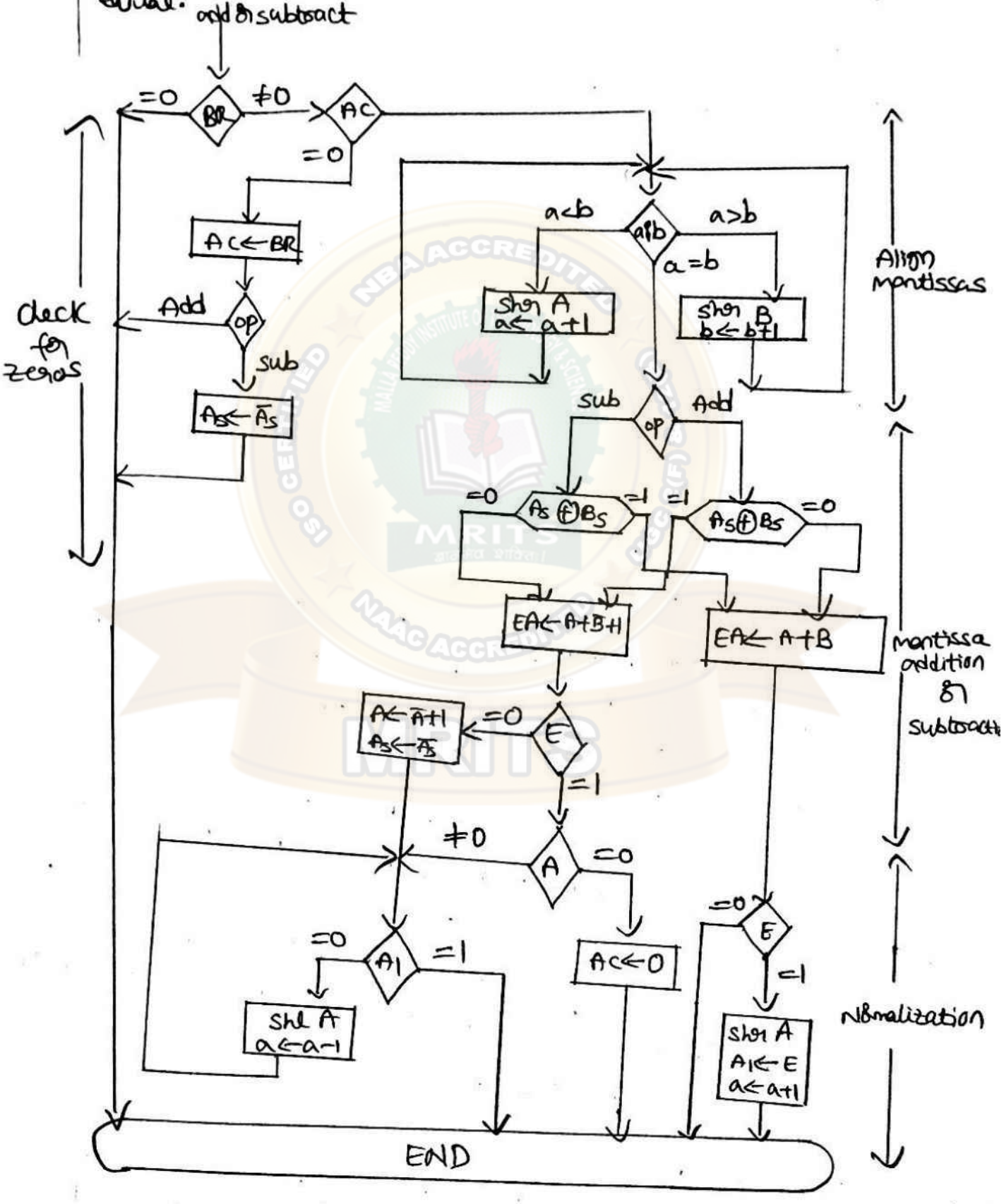


Fig 10.15 Addition & subtraction of floating-point numbers.

→ The magnitude part is added or subtracted depending on the operation and the signs of the two mantissas. if an overflow occurs when the magnitudes are added, it is transferred into flip-flop E.

→ If E is equal to 1, the bit is transferred into A_1 and all other bits of A are shifted right. The exponent must be incremented to maintain the correct number. NO underflow may occur in this case because the original mantissa that was not shifted during the alignment was already in a normalized position.

→ If the magnitudes were subtracted, the result may be zero, or may have an underflow. If the mantissa is zero, the entire floating-point number in the AC is made zero. otherwise, the mantissa must have at least one bit that is equal to 1. The mantissa has an underflow if the most significant bit in position A_1 is 0.

→ In that case, the mantissa is shifted left & the exponent decremented. The bit in A_1 is checked again & the process is repeated until it is equal to 1. when $A_1 = 1$, the mantissa is normalized & the operation is completed.

Multiplication:- The multiplication of two floating-point numbers requires that we multiply the mantissas & add the exponents. No comparison of exponents or alignment of mantissas is necessary. The multiplication of the mantissas is performed in the same way as in fixed-point to provide a double-precision product. The double precision answer is used in fixed-point numbers to increase the accuracy of the product.

→ The multiplication algorithm can be subdivided into 4 parts:

1) check for zeros
2) add the exponents.
3) multiply the mantissas
4) normalize the product.

→ Step 2 & 3 can be done simultaneously if separate adders are available for the mantissas & exponents.

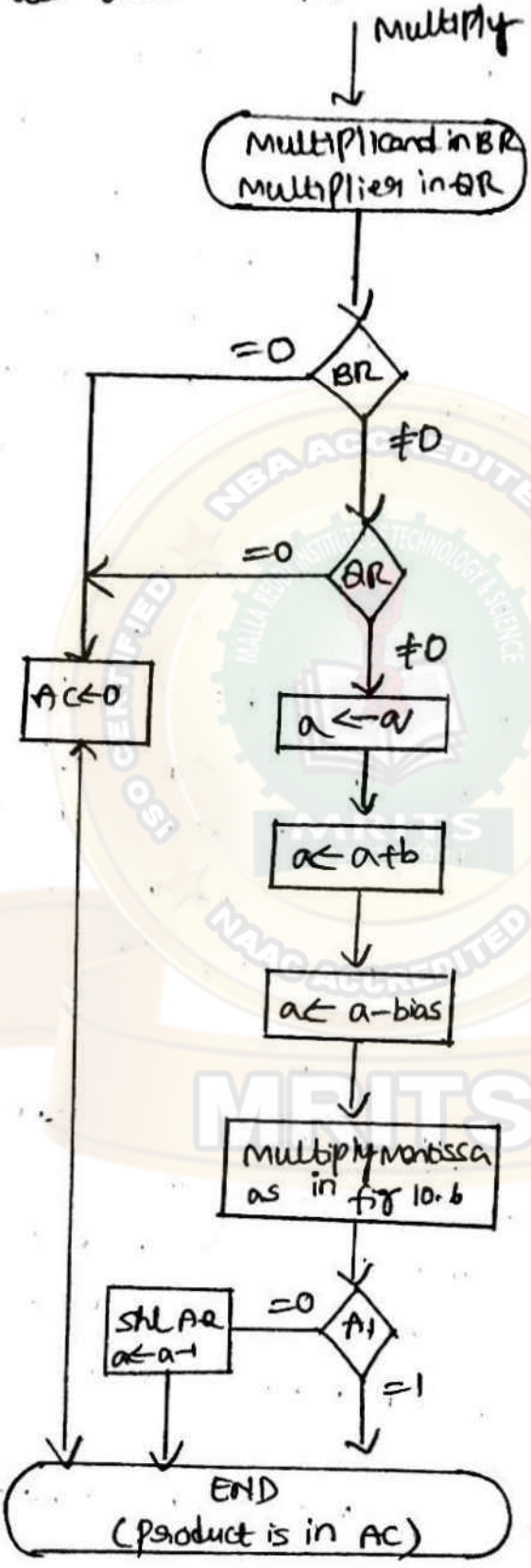


Fig 10.16 Multiplication of floating-point numbers.

→ the flowchart, represents if either operand is equal to zero, the product in the AC is set to zero & the operation is terminated. If neither of the operands is equal to zero, the process continues with the exponent addition.

→ The exponent of the multiplier is in Q & the adder is between exponents a & b. It is necessary to transfer the exponents from Q to a, add the two exponents, & transfer the sum into a. Since both exponents are biased by the addition of a constant, the exponent sum will have double this bias. The correct biased exponent for the product is obtained by subtracting the bias number from the sum.

→ The product may have an underflow, so the most significant bit in A is checked. If it is a 1, the product is already normalized. If it is a 0, the mantissa in AQ is shifted left & the exponent decremented. Note that only one normalization shift is necessary. The multiplier & multiplicand were orginally normalized & contained fractions. The smallest normalized operand is 0.1, so the smallest possible product is 0.01. So, only one leading zero may occur.

Division :— Floating point division requires that the exponents be subtracted & the mantissa divided. The mantissa division is done as in fixed-point except that the dividend has a single-precision mantissa that is placed in the AC. Remember that the mantissa dividend is a fraction & not an integer.

→ For integer representation, a single-precision dividend must be placed in register a & register A must be cleared.

→ The zeros in A are to the left of the binary point and have no significance. In fraction representation, a single-precision dividend is placed in register A and register Q is cleared. The zeros in Q are to the right of the binary point and have no significance.

→ The check for divide-overflow is the same as in fixed-point representation. For normalized operands this is a sufficient operation to ensure that no mantissa divide-overflow will occur. The operation above is referred as a dividend alignment

→ The division of two normalized ~~operation~~ floating-point numbers will always result in a normalized quotient provided that a dividend alignment is carried out before the division.

→ The division algorithm can be subdivided into five parts:

1) check for zeros.
2) Initialize registers & evaluate the sign
3) Align the dividend.
4) Subtract the exponents
5) Divide the mantissas.

→ In the diagram the two operands are checked for zero. If the divisor is zero, it indicates an attempt to divide by zero, which is an illegal operation. The operation is terminated with an error message. An alternative procedure would be to set the quotient in QR to the most positive number possible or to the most negative possible.

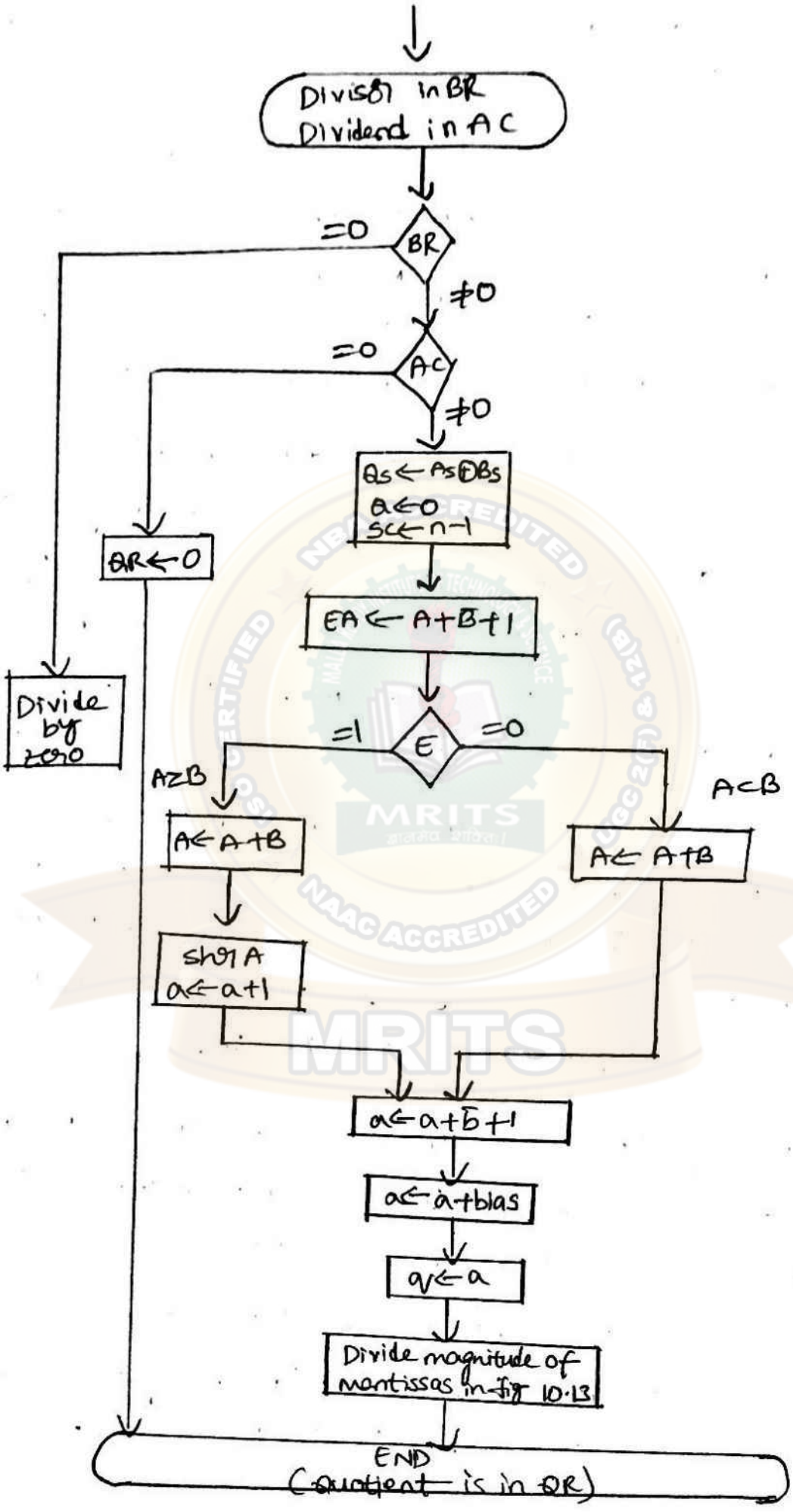


Fig 10.17 Division of floating point numbers.

→ If the dividend in AC is zero, the quotient in QR is made zero & the operation terminates.

→ If the operands are not zero, we proceed to determine the sign of the quotient and store it in Qs. The sign of the dividend in As is left unchanged to be the sign of the remainder. The Q register is cleared & the sequence counter SC is set to a number equal to the number of bits in the quotient.

→ The two fractions are compared by a subtraction test. The carry in E determines their relative magnitude. The dividend fraction is restored to its original value by adding the divisor. If $A \geq B$, it is necessary to shift A once to the right & increment the divident exponent. Since both operands are normalized, this alignment ensure that $A < B$.

→ Next, the divisor exponent is subtracted from the dividend exponent. Since both exponents were originally biased, the subtraction operation gives the difference without the bias. The bias is then added & the result transferred into 'Q' because the quotient is formed in QR.

→ The magnitudes of the mantissas are divided as in the fixed-point case. After the operation, the mantissa quotient resides in Q & the remainder in A.

→ The remainder can be converted to a normalized fraction by subtracting 1 from the dividend exponent & by shift & decrement until the bit in A_1 is equal 1.

Decimal Arithmetic Unit:- The user of a computer prepares data with decimal numbers and receives result 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, & to convert the results into decimal. This may be an efficient method in applications requiring a large number of calculations & a relatively smaller amount of input & output data.

→ Electronic calculators invariably use an internal decimal arithmetic unit, since inputs & outputs are frequent. There does not seem to be a reason for converting the keyboard input numbers to binary & again converting the displayed results to decimal, since this process requires special circuits & also takes a longer time to execute.

→ A decimal arithmetic unit is a digital function that performs decimal microoperations. It can add or subtract decimal numbers, usually by forming the 9's or 10's complement of the subtrahend.

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 doesnot exceed 9, the output sum cannot be [redacted] 9+9+1=19, the '1' in the sum being an inp[redacted]

→ Suppose that we apply two BCD digits to a 4-bit binary adder. The adder will form the sum in binary & produce a result that may range from 0 to 19.

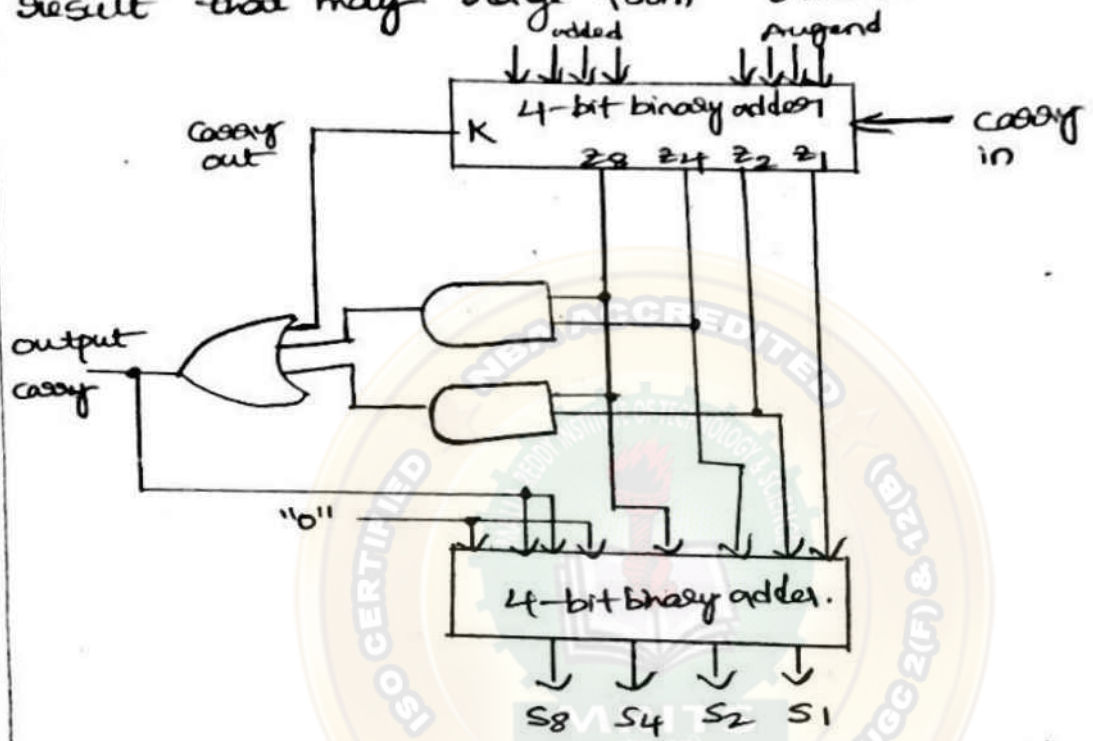


Fig 10.8 Block diagram of BCD adder.

→ In the above table the binary numbers are labeled by symbols K, Z_8, Z_4, Z_2 and Z_1. K is the carry & the subscripts under the letter Z represent the weights 8, 4, 2 & 1 that can be assigned to the four bits in the BCD code.

→ The first column in the table lists the binary sums as they appear in the outputs of a 4-bit binary adder. The output sum of two decimal numbers must be represented in BCD & should appear in the form listed in the second column of the table.

→ The problem is to find a simple rule by which the binary number in the first column can be converted to the correct BCD digit representation of numbers in the second column.

→ when the binary sum is equal to or less than 1001, the corresponding BCD number is identical & therefore no correction is needed.

→ when the binary sum is greater than 1001, we obtain a nonvalid BCD representation.

→ One method of adding decimal numbers in BCD would be to employ one 4-bit binary adder and perform the arithmetic operation one digit at a time. The low-order pair of BCD digits is first added to produce a binary sum. If the result is equal or greater than 1010, it is corrected by adding 0110 to the binary sum.

→ The condition for a correction and an output-carry can be expressed by the Boolean function.

$$C = K + Z_8 Z_4 + Z_8 Z_2.$$

when $C = 1$, it is necessary to add 0110 to the binary sum & provide an output-carry for the next stage.

→ A BCD adder is a circuit that adds two BCD digits in parallel and produces a sum digit also in BCD. A BCD adder must include the correction logic in its internal construction. To add 0110 to the binary sum, we use a second 4-bit binary adder.

→ In diagram shows the block diagram of BCD adder, The two decimal digits, together with the input-carry, are first added in the top 4-bit binary adder to produce the binary sum.

→ when the output-carry is equal to '0', nothing is added to the binary sum. when it is equal to 1, binary 0110 is added to the binary sum through the bottom 4-bit binary adder.

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

Table 10.4 Derivation of BCD Adder.

→ The output-carry generated from the bottom binary-adder may be ignored, since it supplies information already available in the output-carry terminal.

→ A decimal parallel-adder that adds 'n' decimal digits needs 'n' BCD adder stages with the output-carry from one stage connected to the input-carry of the next-higher-order stage.

BCD subtraction:- It is more economical to perform the subtraction by taking the 9's or 10's complement of the subtrahend & adding it to the minuend. Since the BCD is not a self-complementing code, the 9's complement cannot be obtained by complementing each bit in the code. It must be formed by a circuit that subtracts each BCD digit from 9.

→ The 9's complement of a decimal digit represented in BCD may be obtained by complementing the bits in the coded representation of the digit provided a correction is included.

→ There are two possible correction methods. In the first method, binary 1010 (decimal 10) is added to each complemented digit & the carry discarded after each addition.

→ In second method, binary 0110 (decimal 6) is added before the digit is complemented. As a numerical example, the 9's complement of BCD 0111 (decimal 7) is computed by first complementing each bit to obtain 1000. Adding binary 1010 & discarding the carry, we obtain 0010 (decimal 2).

→ By second method, we add 0110 to 0111 to obtain 1101. complement each bit, we obtain the required result of 0010. Complementing each bit of a 4-bit binary number 'N' is identical to the subtraction of the number from 1111 (decimal 15).

→ Adding the binary equivalent of decimal 10 gives $15-N+10 = 9-N+16$. But 16 signifies the carry that is discarded, so the result is $9-N$.

→ Adding the binary equivalent of decimal 6 & then complementing gives $15-(N+6) = 9-N$ as required.

→ The 9's complement of a BCD digit can also be obtained through a combinational circuit. When this circuit is attached to a BCD adder, the result is a BCD adder/subtractor.

→ Let the subtrahend or (addend) digit be denoted by the four binary variables B_8, B_4, B_2 & B_1. Let M be a mode bit that controls the add/subtract operation. When M=0, the two digits are added; when M=1, the digits are subtracted.

→ Let the binary variables x_8, x_4, x_2 & x_1 be the outputs of the 9's complemented circuit. B_1 should always be complemented; B_2 is always the same in the 9's complement as in the original digit; x_4 is '1' when exclusive-OR of B_2 & B_4 is '1'.

→ And x_8 is '1' when $B_8 B_4 B_2 = 000$. The Boolean functions for the 9's complementer circuit are

$$x_1 = B_1 M' + B_1' M$$
$$x_2 = B_2$$
$$x_4 = B_4 M' + (B_4' B_2 + B_4 B_2') M$$
$$x_8 = B_8 M' + B_8 B_4' B_2' M.$$

From these equations we see that $x = B$ when $M = 0$. When $M = 1$, the x outputs produce the 9's complement of B.

Decimal Arithmetic operations:- The algorithms for arithmetic operations with decimal data are similar to the algorithms for the corresponding operations with binary data. In fact, except for a slight modification in the multiplication & division algorithms, the same flowcharts can be used for both types of data provided that we interpret the microoperation symbols properly. Decimal numbers in BCD are stored in computer registers in groups of four bits. Each 4-bit group represents a decimal digit & must be taken as a unit when performing decimal microoperations.

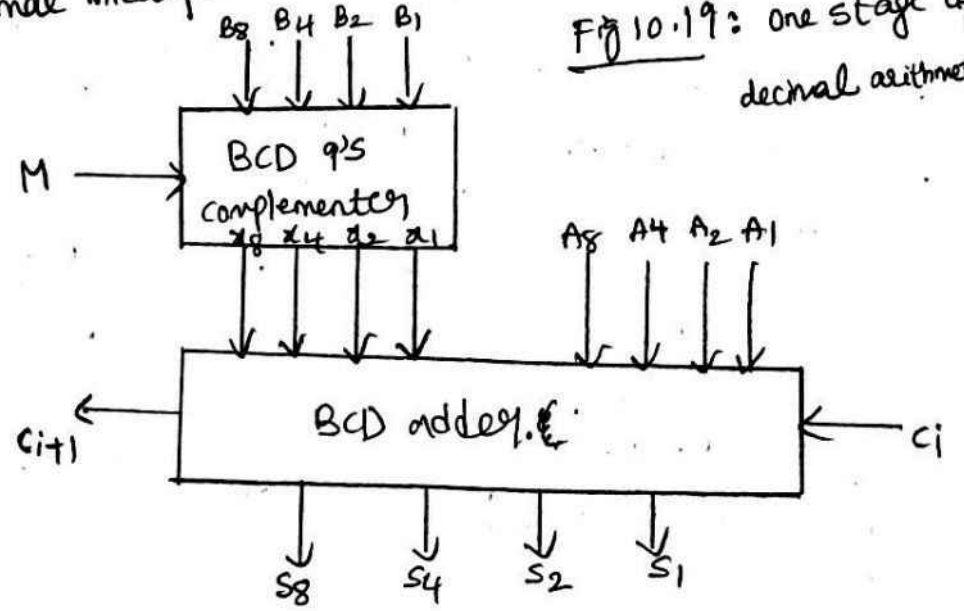


Fig 10.19: One stage of a decimal arithmetic unit

→ we will use the same symbols for binary & decimal arithmetic microoperations but give them a different interpretation. As shown in table below, a bar over the register letter symbol denotes the 9's complement of the decimal number stored in the register.

Table 10.5 Decimal arithmetic Microoperations symbol

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

→ Adding '1' to the 9's complement produces the 10's complement. Thus for decimal numbers the symbol $A \leftarrow A+\overline{B}+1$ denotes a transfer of the decimal sum formed by adding the original content A to 10's complement of B.

→ Incrementing or decrementing a register is the same for binary and decimal except for the number of states that the register is allowed to have. A binary counter goes through 16 states, from 0000 to 1111, when incremented. A decimal counter goes through 10 states from 0000 to 1001 & back to 0000, since 9 is the last count. Similarly, a binary counter sequences from 1111 to 0000 when decremented. A decimal counter goes from 1001 to 0000.

Addition & subtraction :- The algorithm for addition and subtraction of binary signed-magnitude numbers applies also to decimal signed-magnitudes numbers provided that we interpret the microoperation symbols in the proper manner. Similarly the algorithm for binary signed-2's complement numbers applies to decimal signed-10's complement numbers.

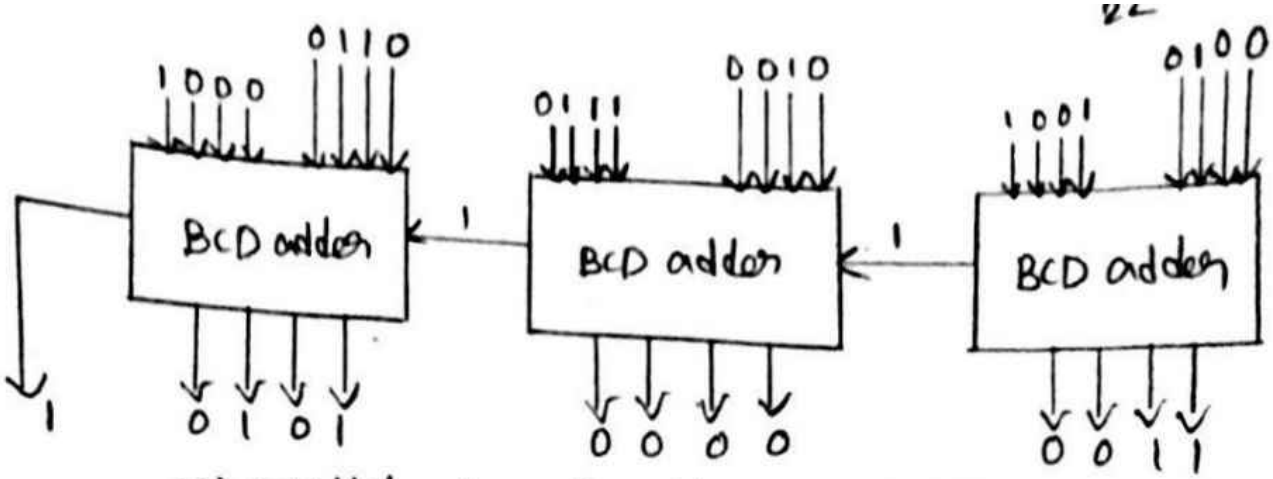
→ The binary data must employ a binary adder and a complementer. The decimal data must employ a decimal arithmetic unit capable of adding two BCD numbers & forming the 9's complement of the subtrahend, as shown in figure one stage of a decimal arithmetic unit.

→ Decimal data can be added in three different ways, the parallel method uses a decimal arithmetic unit composed of as many BCD adders as there are digits in the number. The sum is formed in parallel & requires only one microoperation.

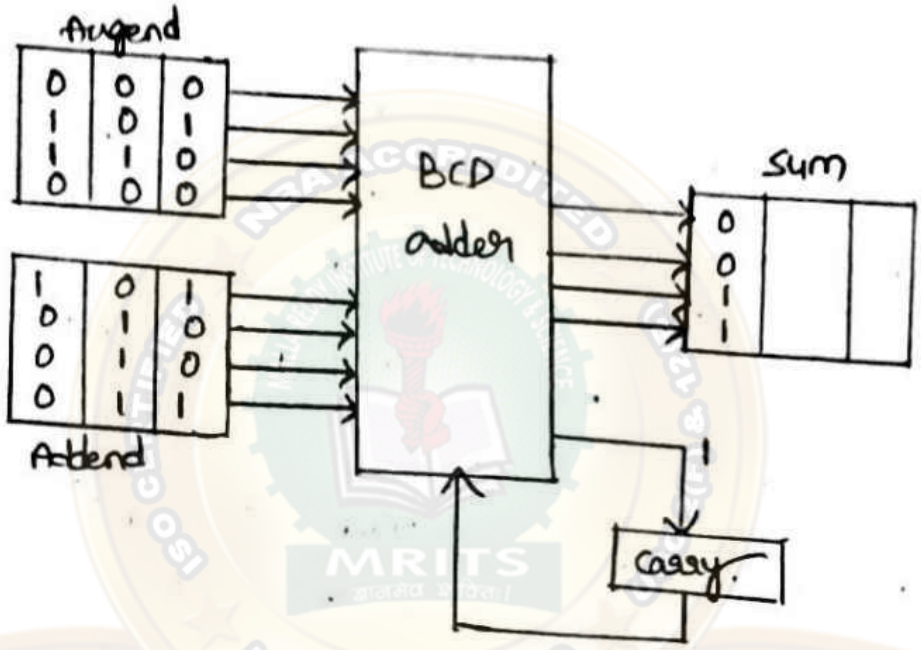
→ In digit-serial bit-parallel method, the digits are applied to a single BCD adder serially, while the bits of each coded digit are transferred in parallel. The sum is formed by shifting the decimal numbers through the BCD adder one at a time.

→ For K decimal digits, this configuration requires K microoperations, one for each decimal shift. In all the serial adders, the bits are shifted one at a time through a full-adder.

→ The binary sum formed after four shifts must be corrected into a valid BCD digit.



(a) parallel decimal addition : 624+879 = 1503.



b) Digit-Serial, bit-parallel decimal addition.

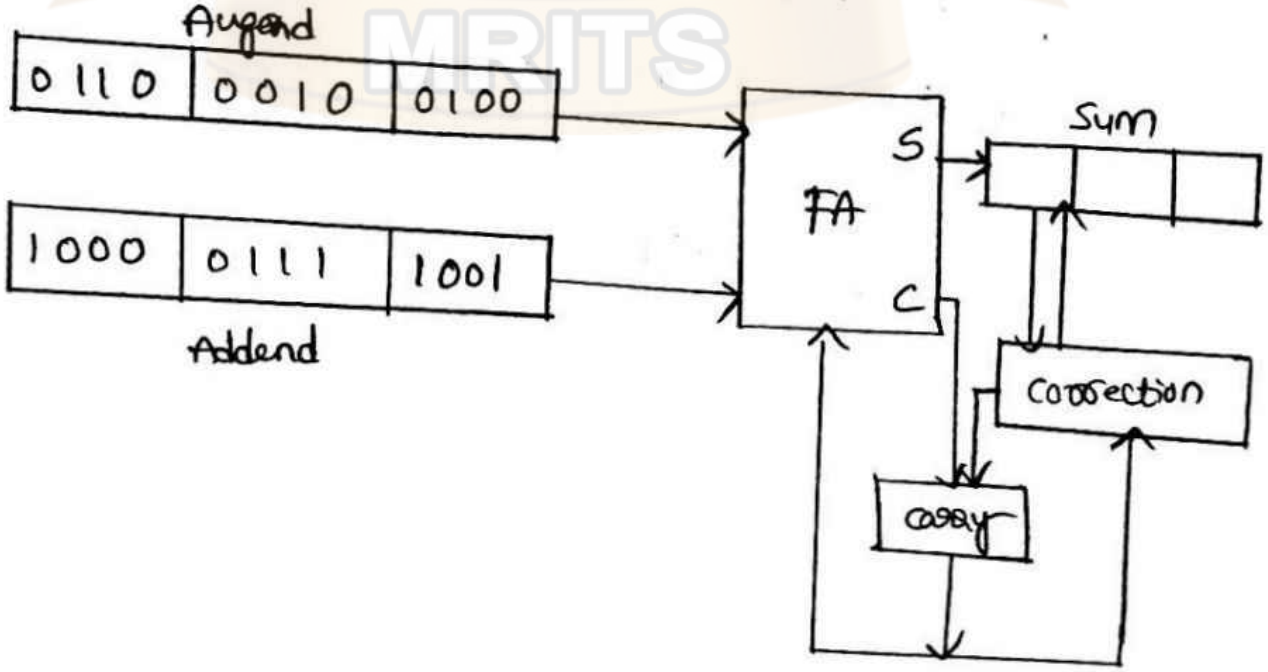


Figure 10.20 Three ways of adding decimal numbers.

→ The parallel method is fast but requires a large number of adders. The digit-serial bit-parallel method requires only one BCD adder, which is shared by all the digits.

→ It is slower than the parallel method because of the time required to shift the digits. The all serial method requires a minimum amount of equipment but is very slow.

Multiplication:- The multiplication of fixed-point decimal numbers is similar to binary except for the way the partial products are formed. A decimal multiplier has digits that range in value from 0 to 9, whereas a binary multiplier has only 0 & 1 digits.

→ In the binary case, the multiplicand is added to the partial product if the multiplier bit is 1. In the decimal case, the multiplicand must be multiplied by the digit multiplier & the result added to the partial product.

→ This operation can be accomplished by adding the multiplicand to the ~~partial~~ Partial product a number of times equal to the value of the multiplier digit.

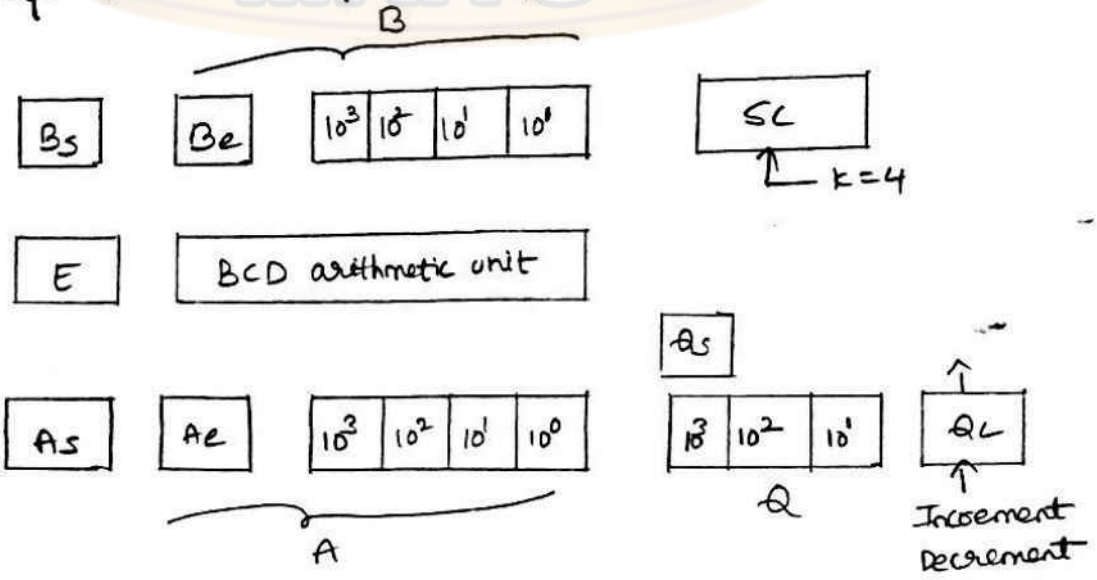


Fig 10.21 Registers for decimal arithmetic multiplication & division.

-) In the figure shows the register organization for the decimal multiplication. we are assuming have four-digit numbers with each digit occupying 4 bits, for a total of 16 bits for each number.

-> There are three registers, A, B & Q, each having a corresponding sign flip-flop As, Bs, & Qs. Registers A & B have four more bits designated by Ae & Be that provide an extension of one more digit, to the registers. The BCD arithmetic unit adds the five digits in parallel & places the sum in the five-digit A register. The end-carry goes to flip-flop E.

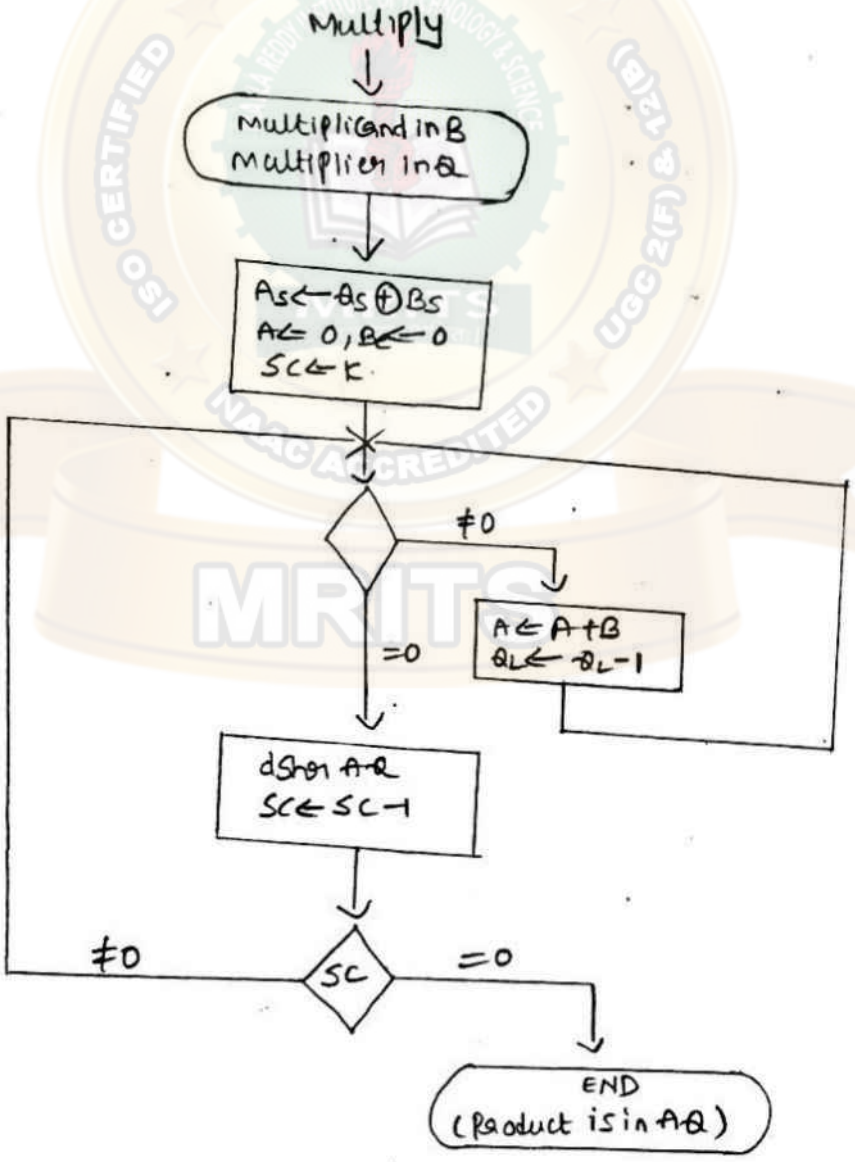


Fig 10.22 Flowchart for decimal multiplication.

→ A decimal operand coming from memory consists of 17 bits. One bit (the sign) is transferred to Bs & the magnitude of the operand is placed in the lowest 16 bits of B. Both Be & Ao are cleared initially. The result of the operation is also 17 bits long & does not use the Ae part of the A register.

→ The decimal multiplication algorithm shown in diagram.

→ Initially, the entire A register & Be are cleared & the sequence counter SC is set to a number K equal to the number of digits in the multiplier. The low-order digit of the multiplier in Q_L is checked. If it is not equal to 0, the multiplicand in B is added to the partial product in A once & Q_L is decremented.

→ Q_L is checked again & the process is repeated until it is equal to '0'. In this way, the multiplicand in B is added to the partial product a number of times equal to the multiplier digit. Any temporary overflow digit will reside in Ae & can range in value from 0 to 9.

→ Next, the partial product & the multiplier are shifted once to the right. This places zero in Ae & transfers the next multiplier quotient into Q_L. The process is then repeated K times to form a double-length product in AQ.

Division:- Decimal division is similar to binary division except of course that the quotient digits may have any of the 10 values from 0 to 9. In the restoring division method, the divisor is subtracted from the dividend or partial remainder as many times as necessary until a negative remainder results.

→ The correct remainder is then restored by adding the divisor. The digit in the quotient reflects the number of subtractions up to but excluding the one that caused the negative difference.

→

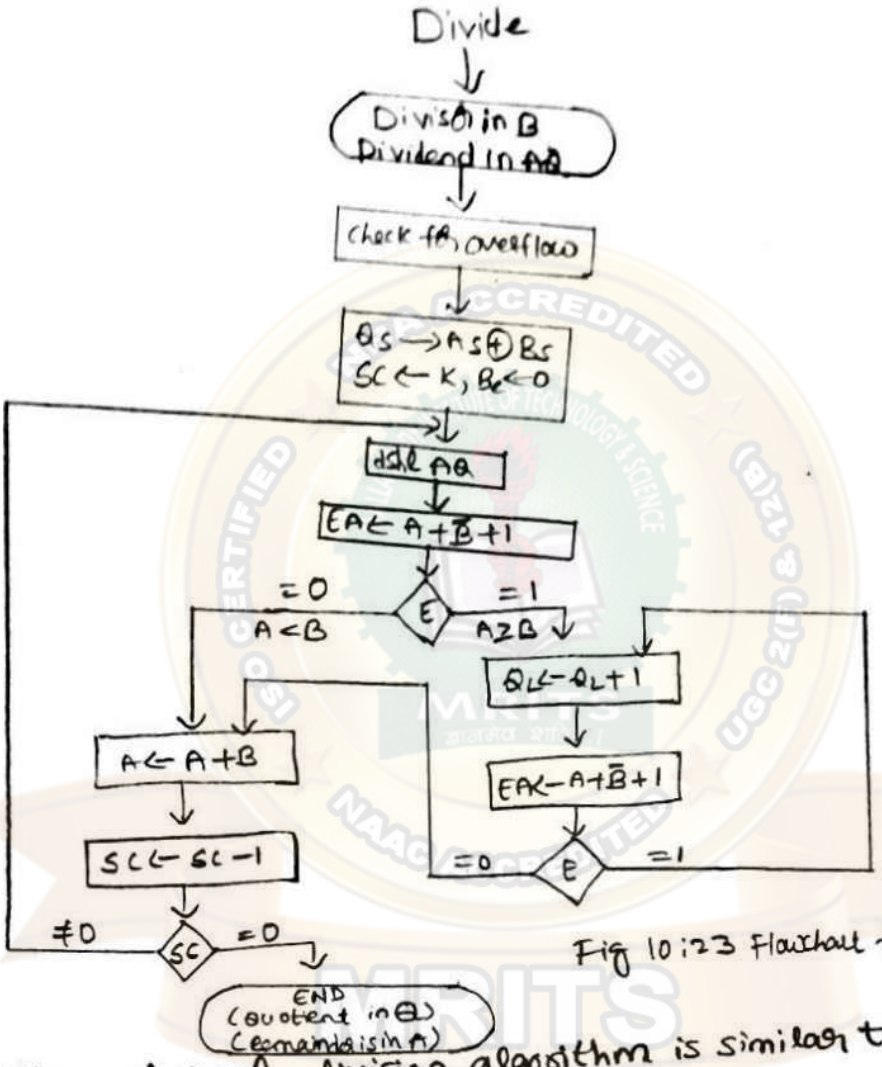


Fig 10:23 Flowchart for decimal division.

→ The decimal division algorithm is similar to the algorithm with binary data except for the way the quotient bits are formed. The dividend (or partial remainder) is shifted to the left, with its most significant digit placed in Ae.

→ The divisor then subtracted by adding its 10's complement value. Since Be is initially cleared, its complement value is 9 as required. The carry in E determines relative magnitude of A & B.

→ If E=0, it signifies that A<B. In this case the divisor is added to restore the partial remainder & Q_L stays at 0. If E=1, it signifies that A≥B. The quotient digit in Q_L is incremented once & the divisor subtracted again.

→ This process is repeated until the subtraction results in a negative difference which is recognised by E being 0.

→ When this occurs, the quotient digit is not incremented but the divisor is added to restore the positive remainder. In this way, the quotient digits is made equal to the number of times that the partial remainder "goes" into the divisor.

Input-Output organization

Peripheral devices

- In addition to the processor and a set of memory modules, the third key element of a computer system is a set of input-output subsystem referred to as I/O, provides an efficient mode of communication between the central system and the outside environment.
- Programs and data must be entered into computer memory for processing and results obtained from computations must be recorded or displayed for the user.
- Devices that are under the direct control of the computer are said to be connected on-line. These devices are designed to read information into or out of the memory unit upon command from CPU.
- Input or output devices attached to the computer are also called peripherals.
- Among the most common peripherals are keyboards, display units, and printers.
- Perhaps those provide auxiliary storage for the systems are magnetic disks and tapes.
- Peripherals are electromechanical and electromagnetic devices of some complexity.
- We can broadly classify peripheral devices into three categories:
    - **Human Readable**: Communicating with the computer users, e.g. video display terminal, printers etc.
    - **Machine Readable**: Communicating with equipments, e.g. magnetic disk, magnetic tape, sensor, actuators used in robotics etc.
    - **Communication**: Communicating with remote devices means exchanging data with that, e.g. modem, NIC (network interface Card) etc.

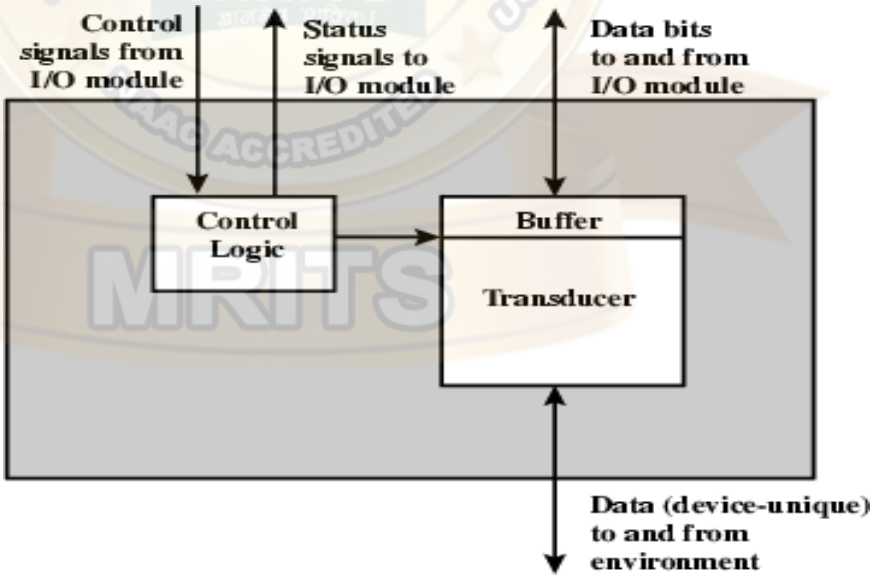


Fig: Block diagram of Peripheral device

- Control signals determine the function that the device will perform such as send data to I/O module, accept data from I/O module.
- Status signals indicate the state of the device i.e. device is ready or not.
- Data bits are actual data transformation.

- Control logic associated with the device controls the device's operation in response to direction from the I/O module.
- The transducer converts data from electrical to other forms of energy during output and from other forms to electrical during input.
- Buffer is associated with the transducer to temporarily hold data being transferred between the I/O module and external devices i.e. peripheral environment.

Input Device

- Keyboard
- Optical input devices
  - Card Reader
  - Paper Tape Reader
  - Optical Character Recognition (OCR)
  - Optical Bar code reader (OBR)
  - Digitizer
  - Optical Mark Reader
- Magnetic Input Devices
  - Magnetic Stripe Reader
  - Magnetic Ink Character Recognition (MICR)
- Screen Input Devices
  - Touch Screen
  - Light Pen
  - Mouse
- Analog Input Devices

Output Device

- Card Puncher, Paper Tape Puncher
- Monitor (CRT, LCD, LED)
- Printer (Impact, Ink Jet, Laser, Dot Matrix)
- Plotter
- Analog
- Voice

I/O modules

- I/O modules interface to the system bus or central switch (CPU and Memory), interfaces and controls to one or more peripheral devices. I/O operations are accomplished through a wide assortment of external devices that provide a means of exchanging data between external environment and computer by a link to an I/O module. The link is used to exchange control status and data between I/O module and the external devices.

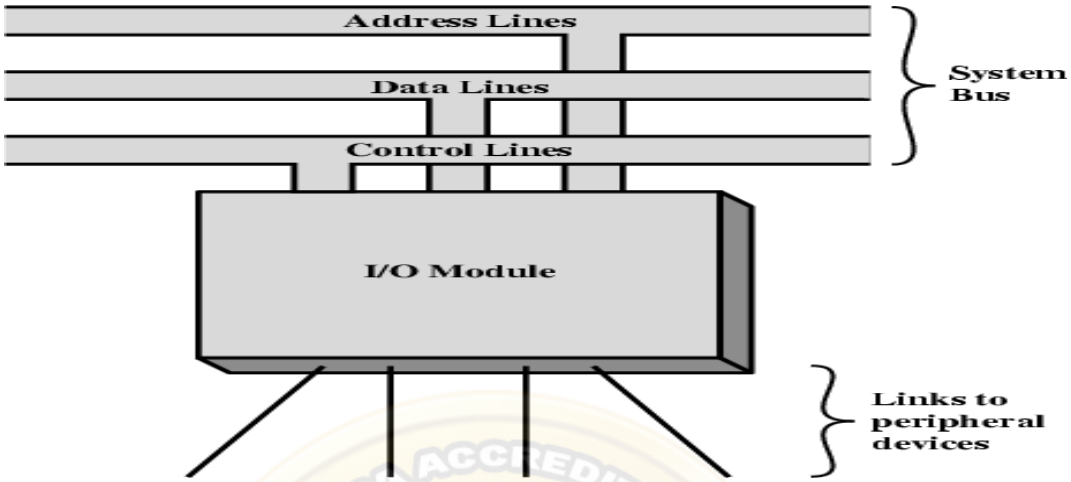


Fig: Model of I/O module

- Peripherals are not directly connected to the system bus instead an I/O module is used which contains logic for performing a communication between the peripherals and the system bus. The reasons due to which peripherals do not directly connected to the system bus are:
    - There are a wide variety of peripherals with various methods of operation. It would be impractical to incorporate the necessary logic within the processor to control a range of devices.
    - The data transfer rate of peripherals is often much slower than that of the memory or processor. Thus, it is impractical to use high speed system bus to communicate directly with a peripheral and vice versa.
    - Peripherals often use different data format and word length than the computer to which they are connected.
- Thus an I/O module is required which performs two major functions.
    - Interface to the processor and memory via the system bus
    - Interface to one or more peripherals by tailored data links

I/O Module Functions

- The I/O module is a special hardware component interface between the CPU and peripherals to supervise and synchronize all I/O transformation The detailed functions of I/O modules are;

  Control & Timing: I/O module includes control and timing to coordinate the flow of traffic between internal resources and external devices. The control of the transfer of data from external devices to processor consists following steps:
    - The processor interrogates the I/O module to check status of the attached device.
    - The I/O module returns the device status.
    - If the device is operational and ready to transmit, the processor requests the transfer of data by means of a command to I/O module.
    - The I/O module obtains the unit of data from the external device.
    - The data are transferred from the I/O module to the processor.

  Processor Communication: I/O module communicates with the processor which involves:

- Command decoding: I/O module accepts commands from the processor.
- Data: Data are exchanged between the processor and I/O module over the bus.
- Status reporting: Peripherals are too slow and it is important to know the status of I/O module.
- Address recognition: I/O module must recognize one unique address for each peripheral it controls.

Device Communication: It involves commands, status information and data.

Data Buffering: I/O module must be able to operate at both device and memory speeds. If the I/O device operates at a rate higher than the memory access rate, then the I/O module performs data buffering. If I/O devices rate slower than memory, it buffers data so as not to tie up the memory in slower transfer operation.

Error Detection: I/O module is responsible for error detection such as mechanical and electrical malfunction reported by device e.g. paper jam, bad ink track & unintentional changes to the bit pattern and transmission error.

I/O Module Structure

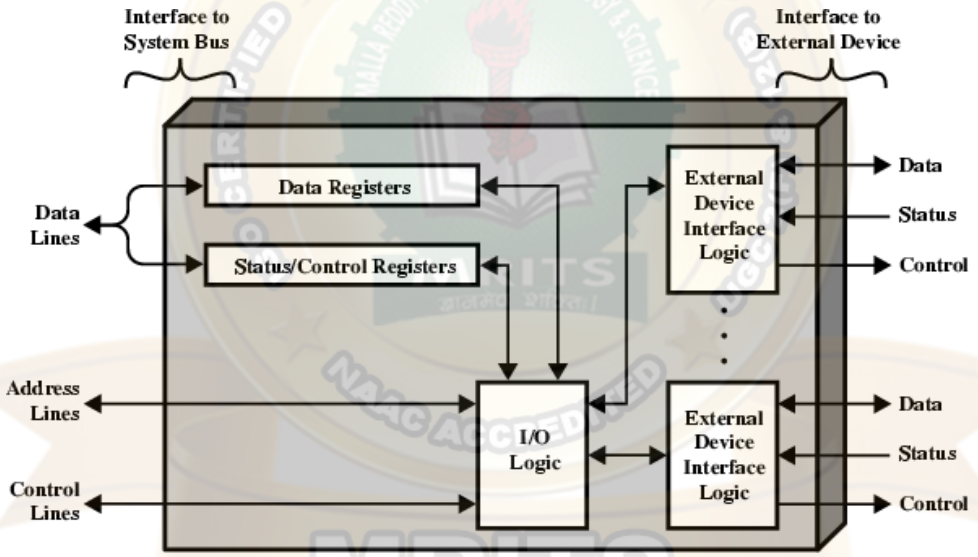


Fig: Block diagram of I/O Module

- The I/O bus from the processor is attached to all peripheral interfaces
- To communicate with the particular devices, the processor places a device address on the address bus.
- Each interface contains an address decoder that monitors the address line. When the interface detects the particular device address, it activates the path between the data line and devices that it controls.
- At the same time that the address is made available in the address line, the processor provides a function code in the control way includes control command, output data and input data.

I/O Module Decisions

- Hide or reveal device properties to CPU
- Support multiple or single device

- Control device functions or leave for CPU
- Also O/S decisions
    - e.g. Unix treats everything it can as a file

Input-Output interface

- Input-Output interface provides a method for transferring information between internal storage (such as memory and CPU registers) and external I/O devices.
- Peripherals connected to a computer need special communication l nks for interfacing them with the central processing unit.
- The communication link resolves the following *differences* between the computer and peripheral devices.
    - Devices and signals
      Peripherals - Electromechanical Devices
      CPU or Memory - Electronic Device
    - Data Transfer Rate
      Peripherals - Usually slower
      CPU or Memory - Usually faster than peripherals
      Some kinds of Synchronization mechanism may be needed
    - Unit of Information
      Peripherals - Byte
      CPU or Memory - Word
    - Operating Modes
      Peripherals - Autonomous, Asynchronous
      CPU or Memory – Synchronous
- To resolve these differences, computer systems include special hardware components (Interfaces) between the CPU and peripherals to supervise and synchronize all input and output interfaces.

I/O Bus and Interface Modules

- The I/O bus consists of data lines, address lines and control lines.

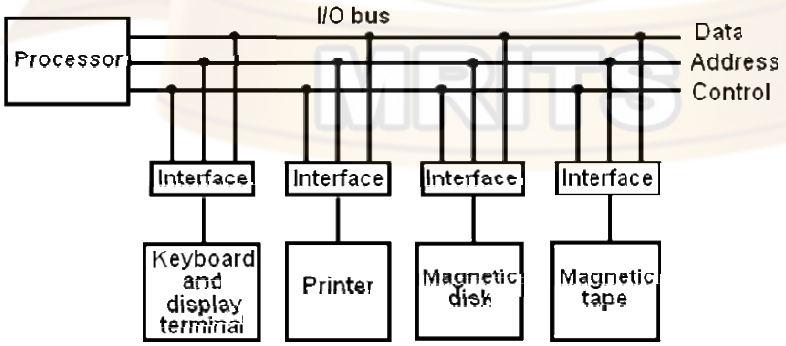


Fig: Connection of I/O bus to input-output devices

- Interface performs the following:
    - Decodes the device address (device code)
    - Decodes the commands (operation)
    - Provides signals for the peripheral controller

- Synchronizes the data flow and supervises the transfer rate between peripheral and CPU or Memory

- I/O commands that the interface may receive:
  - Control command: issued to activate the peripheral and to inform it what to do.
  - Status command: used to test various status conditions in the interface and the peripheral.
  - Output data: causes the interface to respond by transferring data from the bus into one of its registers.
  - Input data: is the opposite of the data output.

I/O versus Memory Bus

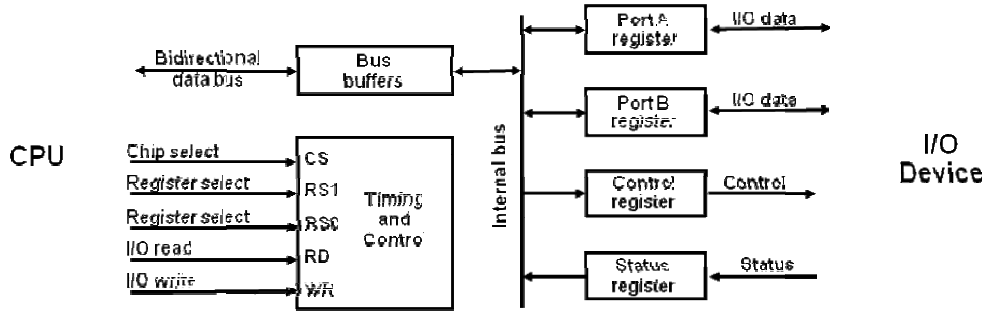
- Computer buses can be used to communicate with memory and I/O in three ways:
  - Use two separate buses, one for memory and other for I/O. In this method, all data, address and control lines would be separate for memory and I/O.
  - Use one common bus for both memory and I/O but have separate control lines. There is a separate read and write lines; I/O read and I/O write for I/O and memory read and memory write for memory.
  - Use a common bus for memory and I/O with common control line. This I/O configuration is called memory mapped.

Isolated I/O versus Memory Mapped I/O

- **Isolated I/O**
  - Separate I/O read/write control lines in addition to memory read/write control lines
  - Separate (isolated) memory and I/O address spaces
  - Distinct input and output instructions

- **Memory-mapped I/O**
  - A single set of read/write control lines (no distinction between memory and I/O transfer)
  - Memory and I/O addresses share the common address space which reduces memory address range available
  - No specific input or output instruction so the same memory reference instructions can be used for I/O transfers
  - Considerable flexibility in handling I/O operations

Example of I/O Interface

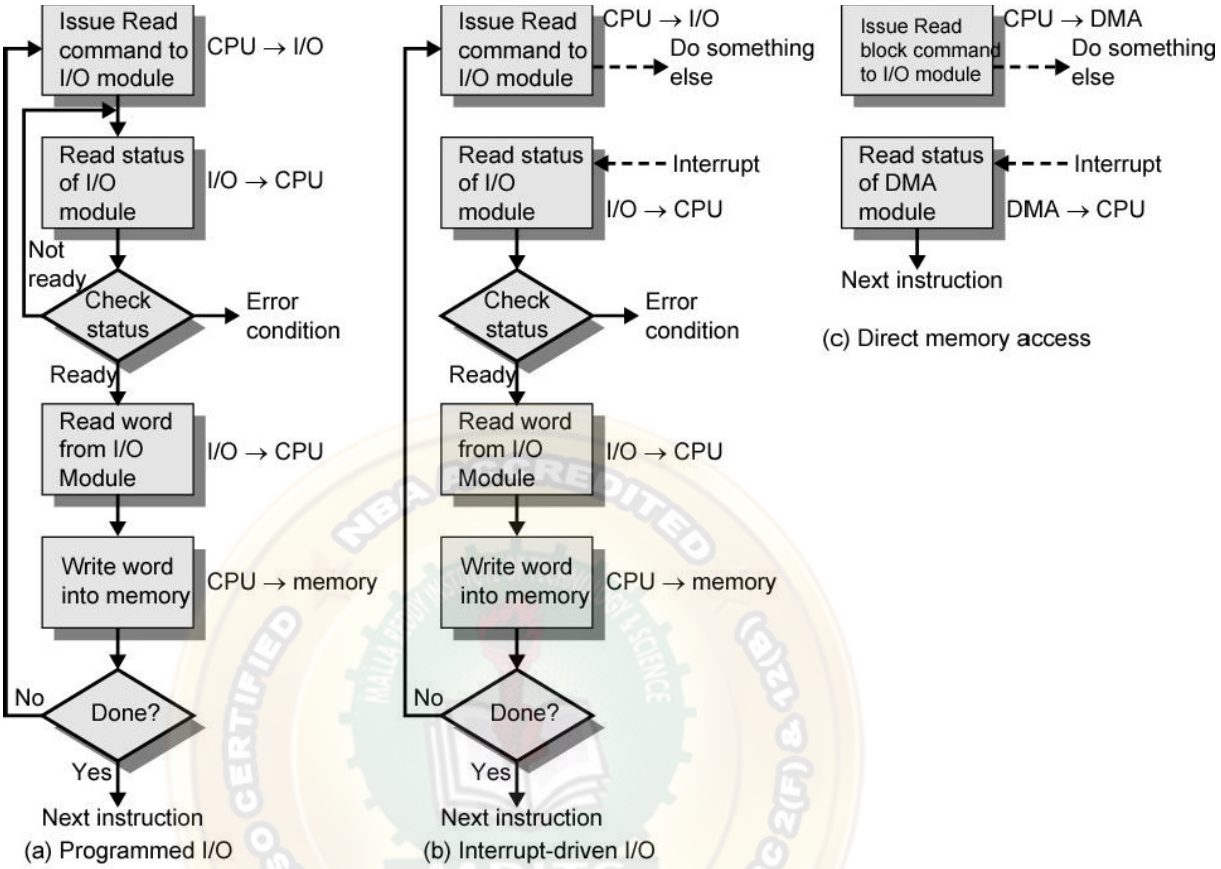


CS	RS1	RS0	Register selected
0	x	x	None - data bus in high-impedence
1	0	0	Port A register
1	0	1	Port B register
1	1	0	Control register
1	1	1	Status register

- Information in each port can be assigned a meaning depending on the mode of operation of the I/O device
  - Port A = Data; Port B = Command; Port C = Status
- CPU initializes (loads) each port by transferring a byte to the Control Register
  - Allows CPU can define the mode of operation of each port
  - *Programmable Port*: By changing the bits in the control register, it is possible to change the interface characteristics

Modes of transfer

- Data Transfer between the central computer and I/O devices may be handled in a variety of modes.
- Some modes use CPU as an intermediate path, others transfer the data directly to and from the memory unit.
- Data transfer to and from peripherals may be handled in one of three possible modes.
  - Programmed I/O
  - Interrupt Driven I/O
  - Direct Memory Access (DMA)



Programmed I/O

- Programmed I/O operations are the result of I/O instructions written in the computer program.
- In programmed I/O, each data transfer in initiated by the instructions in the CPU and hence the CPU is in the continuous monitoring of the interface.
- Input instruction is used to transfer data from I/O device to CPU, store instruction is used to transfer data from CPU to memory and output instruction is used to transfer data from CPU to I/O device.
- This technique is generally used in very slow speed computer and is not a efficient method if the speed of the CPU and I/O is differen .

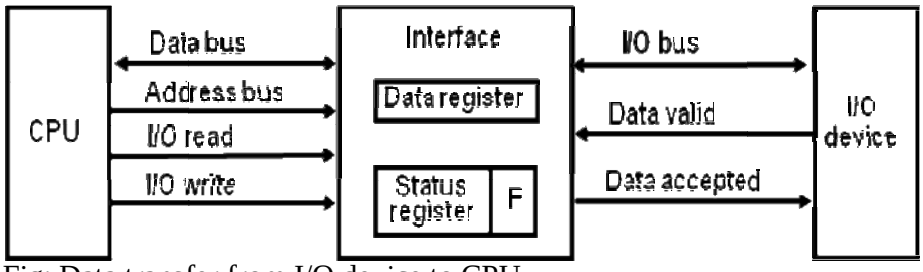


Fig: Data transfer from I/O device to CPU

- I/O device places the data on the I/O bus and enables its data valid signal
- The interface accepts the data in the data register and sets the F bit of status register and also enables the data accepted signal.
- Data valid line is disables by I/O device.
- CPU is in a continuous monitoring of the interface in which it checks the F bit of the status register.
    - If it is set i.e. 1, then the CPU reads the data from data register and sets F bit to zero
    - If it is reset i.e. 0, then the CPU remains monitoring the interface.
- Interface disables the data accepted signal and the system goes to initial state where next item of data is placed on the data bus.

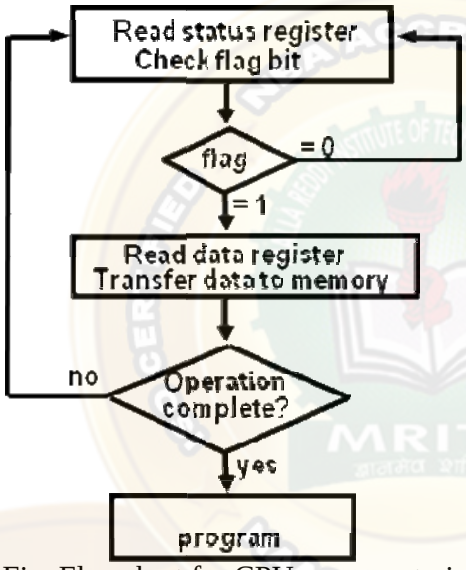


Fig: Flowchart for CPU program to input data

Characteristics:

- Continuous CPU involvement
- CPU slowed down to I/O speed
- Simple
- Least hardware

Polling, or polled operation, in computer science, refers to actively sampling the status of an external device by a client program as a synchronous activity. Polling is most often used in terms of input/output (I/O), and is also referred to as **polled I/O or software driven I/O.**

Interrupt-driven I/O

- Polling takes valuable CPU time
- Open communication only when some data has to be passed -> *Interrupt*.
- I/O interface, instead of the CPU, monitors the I/O device
- When the interface determines that the I/O device is ready for data transfer, it generates an *Interrupt Request* to the CPU
- Upon detecting an interrupt, CPU stops momentarily the task it is doing, branches to the service routine to process the data transfer, and then returns to the task it was performing

The problem with programmed I/O is that the processor has to wait a long time for the I/O module of concern to be ready for either reception or transmission of data. The processor, while waiting, must repeatedly interrogate the status of the I/O module. As a result, the level of the performance of the entire system is severely degraded. An alternative is for the processor to issue an I/O command to a module and then go on to do some other useful work. The I/O module will then interrupt the processor to request service when it is ready to exchange data with processor. The processor then executes the data transfer, and then resumes its former processing. The interrupt can be initiated either by software or by hardware.

Interrupt Driven I/O basic operation

- CPU issues read command
- I/O module gets data from peripheral whilst CPU does other work
- I/O module interrupts CPU
- CPU requests data
- I/O module transfers data

Interrupt Processing from CPU viewpoint

- Issue read command
- Do other work
- Check for interrupt at end of each instruction cycle
- If interrupted:-
    - Save context (registers)
    - Process interrupt
    - Fetch data & store

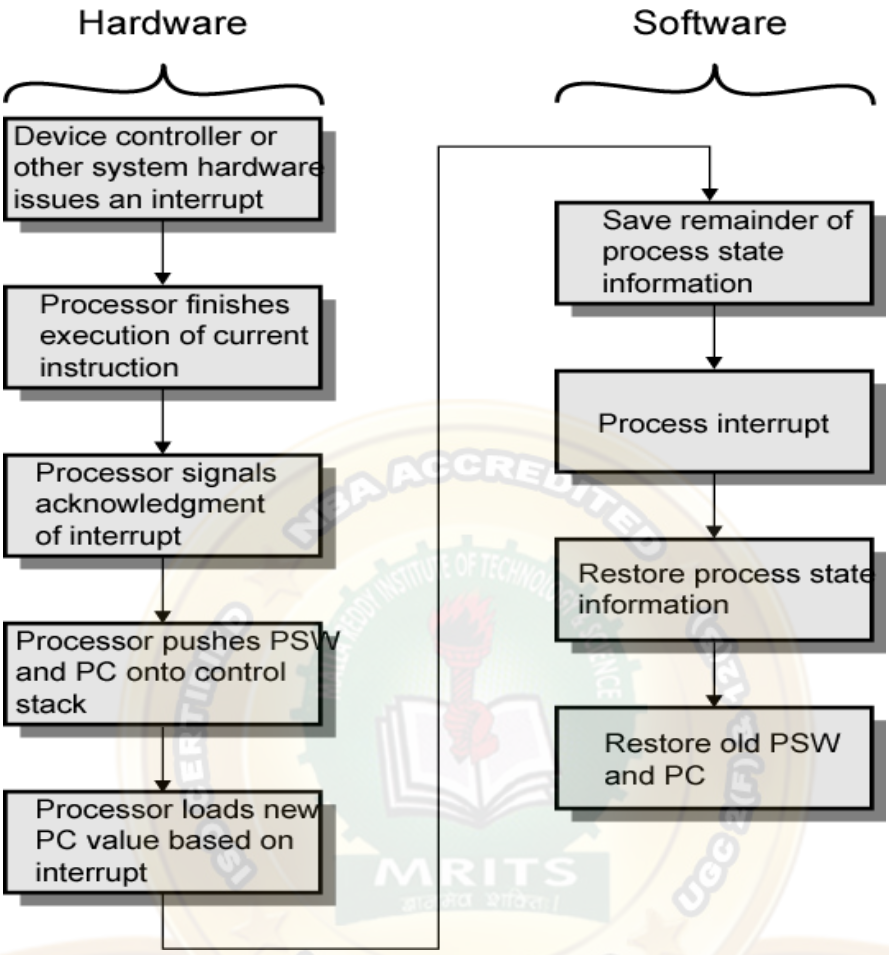


Fig: Simple Interrupt Processing

Priority Interrupt

- Determines which interrupt is to be served first when two or more requests are made simultaneously
- Also determines which interrupts are permitted to interrupt the computer while another is being serviced
- Higher priority interrupts can make requests while servicing a lower priority interrupt

Priority Interrupt by Software (Polling)

- Priority is established by the order of polling the devices (interrupt sources), that is identify the highest-priority source by software means
- One common branch address is used for all interrupts
- Program polls the interrupt sources in sequence
- The highest-priority source is tested first
- Flexible since it is established by software
- Low cost since it needs a very little hardware
- Very slow

Priority Interrupt by Hardware

- Require a priority interrupt manager which accepts all the interrupt requests to determine the highest priority request
- Fast since identification of the highest priority interrupt request is identified by the hardware
- Fast since each interrupt source has its own interrupt vector to access directly to its own service routine

1. Daisy Chain Priority (Serial)

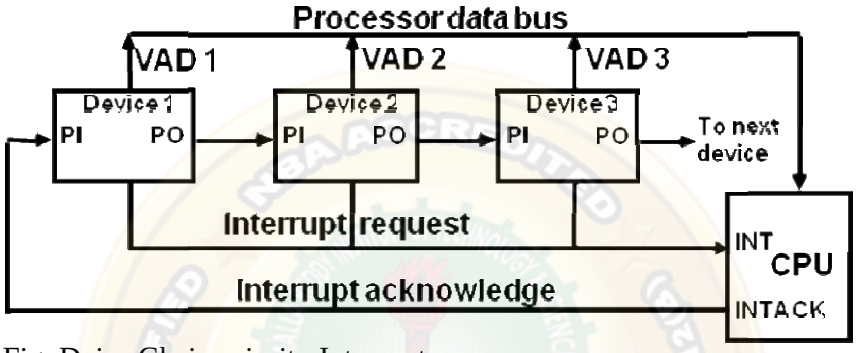
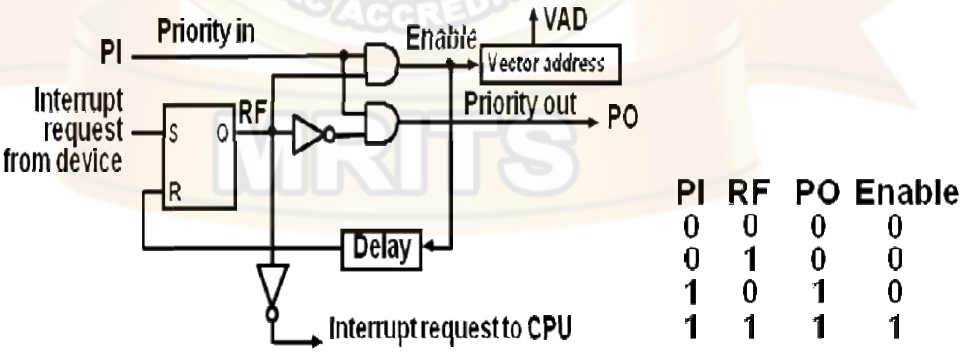


Fig: Daisy Chain priority Interrupt

- Interrupt Request from any device
- CPU responds by INTACK
- Any device receives signal(INTACK) at PI puts the VAD on the bus
- Among interrupt requesting devices the only device which is physically closest to CPU gets INTACK and it blocks INTACK to propagate to the next device



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

Fig: One stage of Daisy chain priority arrangement

2. Parallel Priority

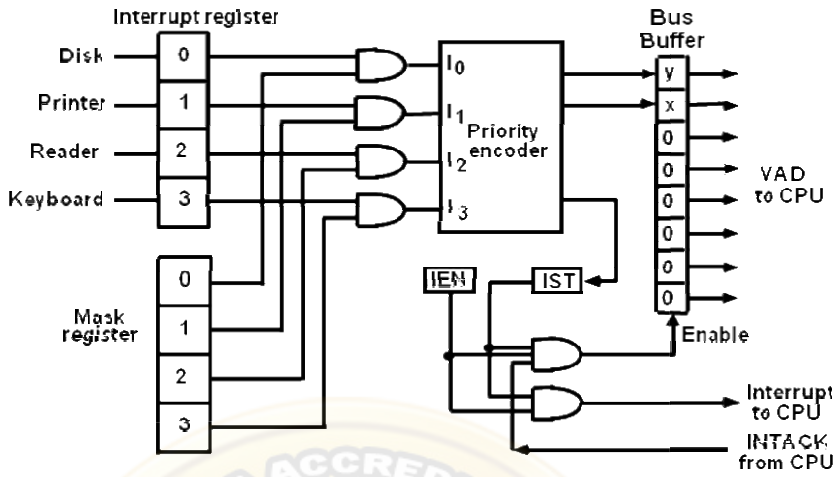


Fig: Parallel priority interrupts hardware

- IEN: Set or Clear by instructions ION or IOF
- IST: Represents an unmasked interrupt has occurred. INTACK enables tristate Bus Buffer to load VAD generated by the Priority Logic
- Interrupt Register:
  - Each bit is associated with an Interrupt Request from different Interrupt Source - different priority level
  - Each bit can be cleared by a program instruction
- Mask Register:
  - Mask Register is associated with Interrupt Register
  - Each bit can be set or cleared by an Instruction

Priority Encoder

- Determines the highest priority interrupt when more than one interrupts take place

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

Fig: Priority Encoder Truth Table

Interrupt Cycle

At the end of each Ins ruction cycle

- CPU checks IEN and IST
- If IEN and IST = 1, CPU -> Interrupt Cycle
    - 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 to execute the first instruction in the in errupt service routine

Direct Memory access

- Large blocks of data transferred at a high speed to or from high speed devices, magnetic drums, disks, tapes, etc.
- DMA controller Interface that provides I/O transfer of data directly to and from the memory and the I/O device
- CPU initializes the DMA controller by sending a memory address and the number of words to be transferred
- Actual transfer of data is done directly between the device and memory through DMA controller -> Freeing CPU for other tasks

The transfer of data between the peripheral and memory without the interaction of CPU and letting the peripheral device manage the memory bus directly is termed as Direct Memory Access (DMA).

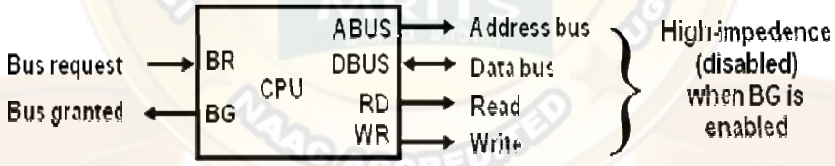


Fig: CPU bus signal for DMA transfer

The two control signals Bus Request and Bus Grant are used to fascinate the DMA transfer. The bus request input is used by the DMA controller to request the CPU for the control of the buses. When BR signal is high, the CPU terminates the execution of the current instructions and then places the address, data, read and write lines to the high impedance state and sends the bus grant signal. The DMA controller now takes the control of the buses and transfers the data directly between memory and I/O without processor interaction. When the transfer is completed, the bus request signal is made low by DMA. In response to which CPU disables the bus grant and again CPU takes the control of address, data, read and write lines.

The transfer of data between the memory and I/O of course facilitates in two ways which are DMA Burst and Cycle Stealing.

DMA Burst: The block of data consisting a number of memory words is transferred at a time.

Cycle Stealing: DMA transfers one data word at a time after which it must return control of the buses to the CPU.

- CPU is usually much faster than I/O (DMA), thus CPU uses the most of the memory cycles
- DMA Controller steals the memory cycles from CPU
- For those stolen cycles, CPU remains idle
- For those slow CPU, DMA Controller may steal most of the memory cycles which may cause CPU remain idle long time

DMA Controller

The DMA controller communicates with the CPU through the data bus and control lines. DMA select signal is used for selecting the controller, the register select is for selecting the register. When the bus grant signal is zero, the CPU communicates through the data bus to read or write into the DMA register. When bus grant is one, the DMA controller takes the control of buses and transfers the data between the memory and I/O.

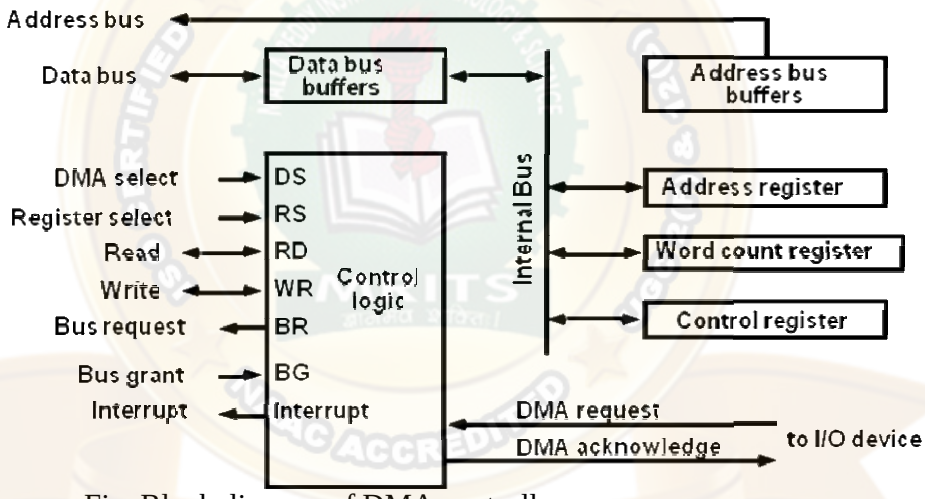


Fig: Block diagram of DMA controller

The address register specifies the desired location of the memory which is incremented after each word is transferred to the memory. The word count register holds the number of words to be transferred which is decremented after each transfer until it is zero. When it is zero, it indicates the end of transfer. After which the bus grant signal from CPU is made low and CPU returns to its normal operation. The control register specifies the mode of transfer which is Read or Write.

DMA Transfer

- DMA request signal is given from I/O device to DMA controller.

- DMA sends the bus request signal to CPU in response to which CPU disables its current instructions and initialize the DMA by sending the following information.
    - The starting address of the memory block where the data are available (for read) and where data to be stored (for write)
    - The word count which is the number of words in the memory block
    - Control to specify the mode of transfer
    - Sends a bust grant as 1 so that DMA controller can take the control of the buses
    - DMA sends the DMA acknowledge signal in response to which peripheral device puts the words in the data bus (for write) or receives a word from the data bus (for read).

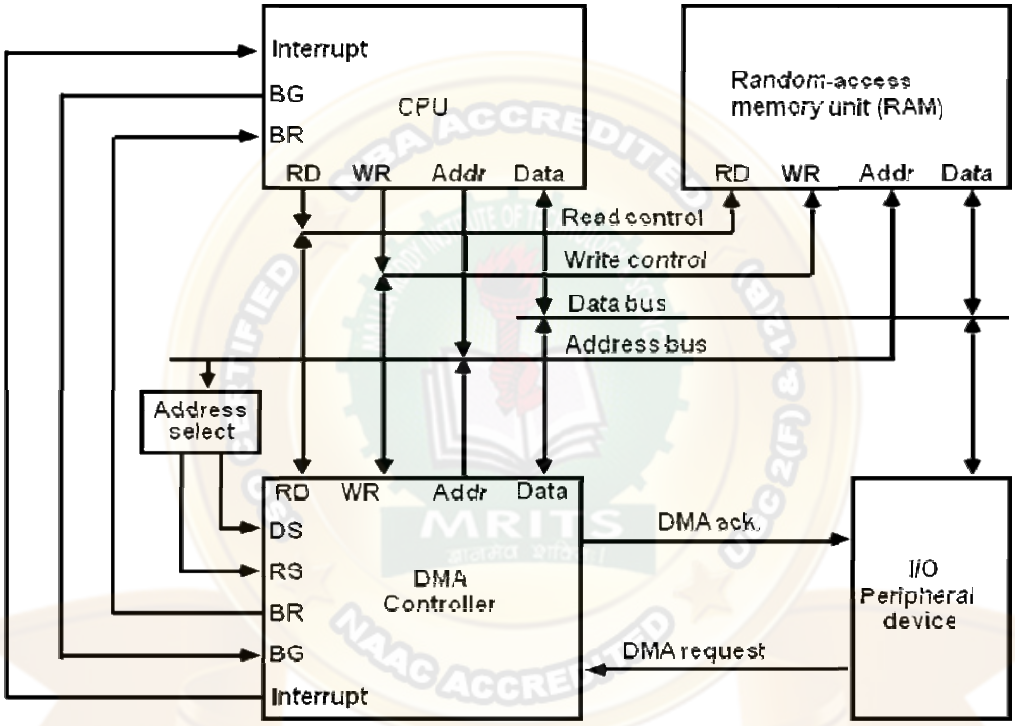


Fig: DMA transfer in a computer system

DMA Operation

- CPU tells DMA controller:-
    - Read/Write
    - Device address
    - Starting address of memory block for data
    - Amount of data to be transferred
- CPU carries on with other work
- DMA controller deals with transfer
- DMA controller sends interrupt when finished

I/O Processors

- Processor with direct memory access capability that communicates with I/O devices
- Channel accesses memory by cycle stealing

- Channel can execute a Channel Program
- Stored in the main memory
- Consists of Channel Command Word(CCW)
- Each CCW specifies the parameters needed by the channel to control the I/O devices and perform data transfer operations
- CPU initiates the channel by executing a channel I/O class instruction and once initiated, channel operates independently of the CPU

A computer may incorporate one or more external processors and assign them the task of communicating directly with the I/O devices so that no each interface need to communicate with the CPU. An I/O processor (IOP) is a processor with direct memory access capability that communicates with I/O devices. IOP instructions are specifically designed to facilitate I/O transfer. The IOP can perform other processing tasks such as arithmetic logic, branching and code translation.

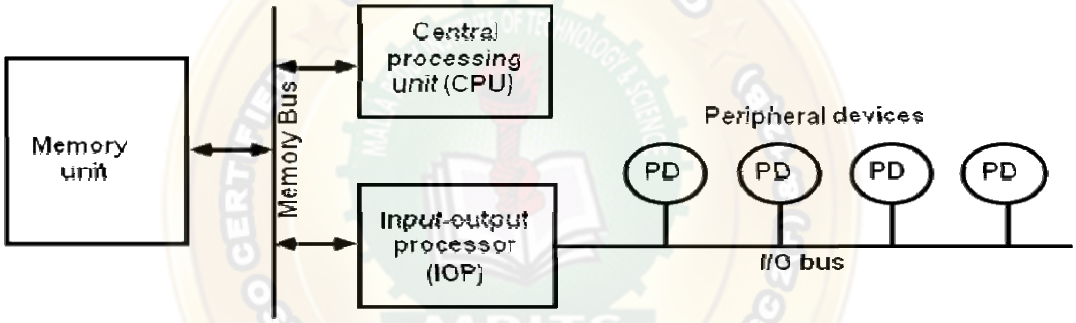


Fig: Block diagram of a computer with I/O Processor

The memory unit occupies a central position and can communicate with each processor by means of direct memory access. The CPU is responsible for processing data needed in the solution of computational tasks. The IOP provides a path for transferring data between various peripheral devices and memory unit.

In most computer systems, the CPU is the master while the IOP is a slave processor. The CPU initiates the IOP and after which the IOP operates independent of CPU and transfer data between the peripheral and memory. For example, the IOP receives 5 bytes from an input device at the device rate and bit capacity. After which the IOP packs them into one block of 40 bits and transfer them to memory. Similarly the O/P word transfer from memory to IOP is directed from the IOP to the O/P device at the device rate and bit capacity.

CPU – IOP Communication

The memory unit acts as a message center where each processor leaves information for the other. The operation of typical IOP is appreciated with the example by which the CPU and IOP communication.

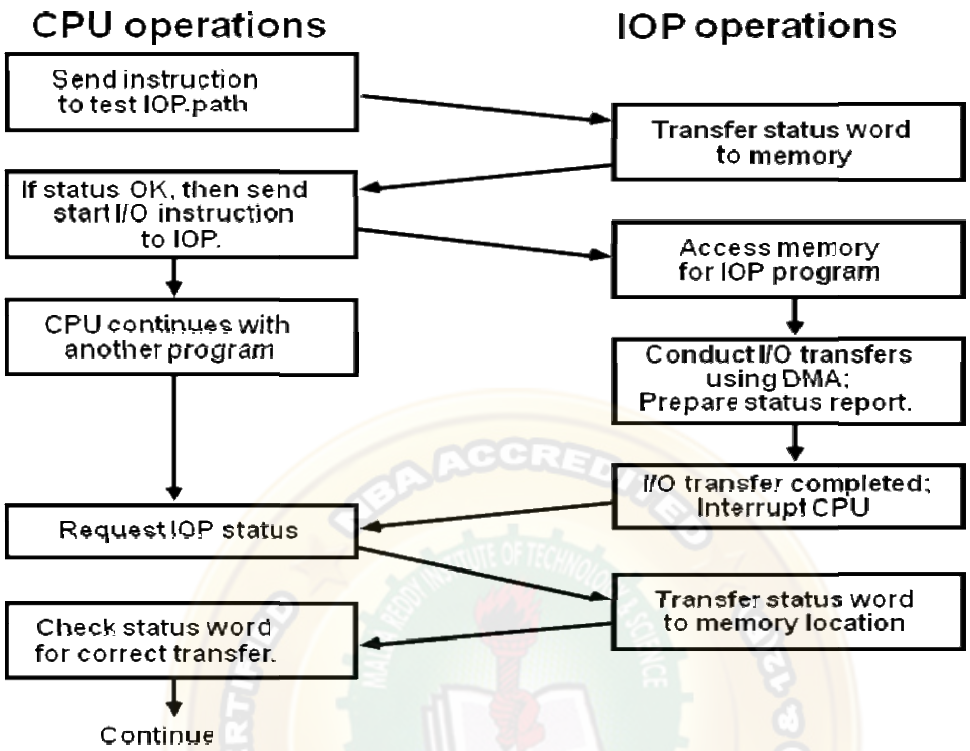


Fig: CPU – IOP communication

- The CPU sends an instruction to test the IOP path.
- The IOP responds by inserting a status word in memory for the CPU to check.
- The bits of the status word indicate the condition of the IOP and I/O device, such as IOP overload condition, device busy with another transfer or device ready for I/O transfer.
- The CPU refers to the status word in in memory to decide what to do next.
- If all right up to this, the CPU sends the instruction to start I/O transfer.
- The CPU now continues with another program while IOP is busy with I/O program.
- When IOP terminates the execution, it sends an interrupt request to CPU.
- CPU responds by issuing an instruction to read the status from the IOP.
- IOP responds by placing the contents to its status report into specified memory location.
- Status word indicates whether the transfer has been completed or with error.

Data Communication Processor

- Distributes and collects data from many remote terminals connected through telephone and other communication lines.
- Transmission:
  - Synchronous
  - Asynchronous
- Transmission Error:
  - Parity
  - Checksum
  - Cyclic Redundancy Check

    - Longitudinal Redundancy Check
- Transmission Modes:
    - Simples
    - Half Duplex
    - Full Duplex
- Data Link & Protocol

A data communication (command) processor is an I/O processor that distributes and collects data from remote terminals connected through telephone and other communication lines. In processor communication, processor communicates with the I/O device through a common bus i.e. data and control with sharing by each peripherals. In data communication, processor communicates with each terminal through a single pair of wires.

The way that remote terminals are connected to a data communication processor is via telephone lines or other public or private communication facilities. The data communication may be either through synchronous transmission or through asynchronous transmission. One of the functions of data communication processor is check for transmission errors. An error can be detected by checking the parity in each character received. The other ways are checksum, longitudinal redundancy check (LRC) and cyclic redundancy check (CRC).

Data can be transmitted between two points through three different modes. First is simplex where data can be transmitted in only one direction such as TV broadcasting. Second is half duplex where data can be transmitted in both directions at a time such as walkie-talkie. The third is full duplex where data can be transmitted in both directions simultaneously such as telephone.

The communication lines, modems and other equipment used in the transmission of information between two or more stations is called data link. The orderly transfer of information in a data link is accomplished by means of a protocol.

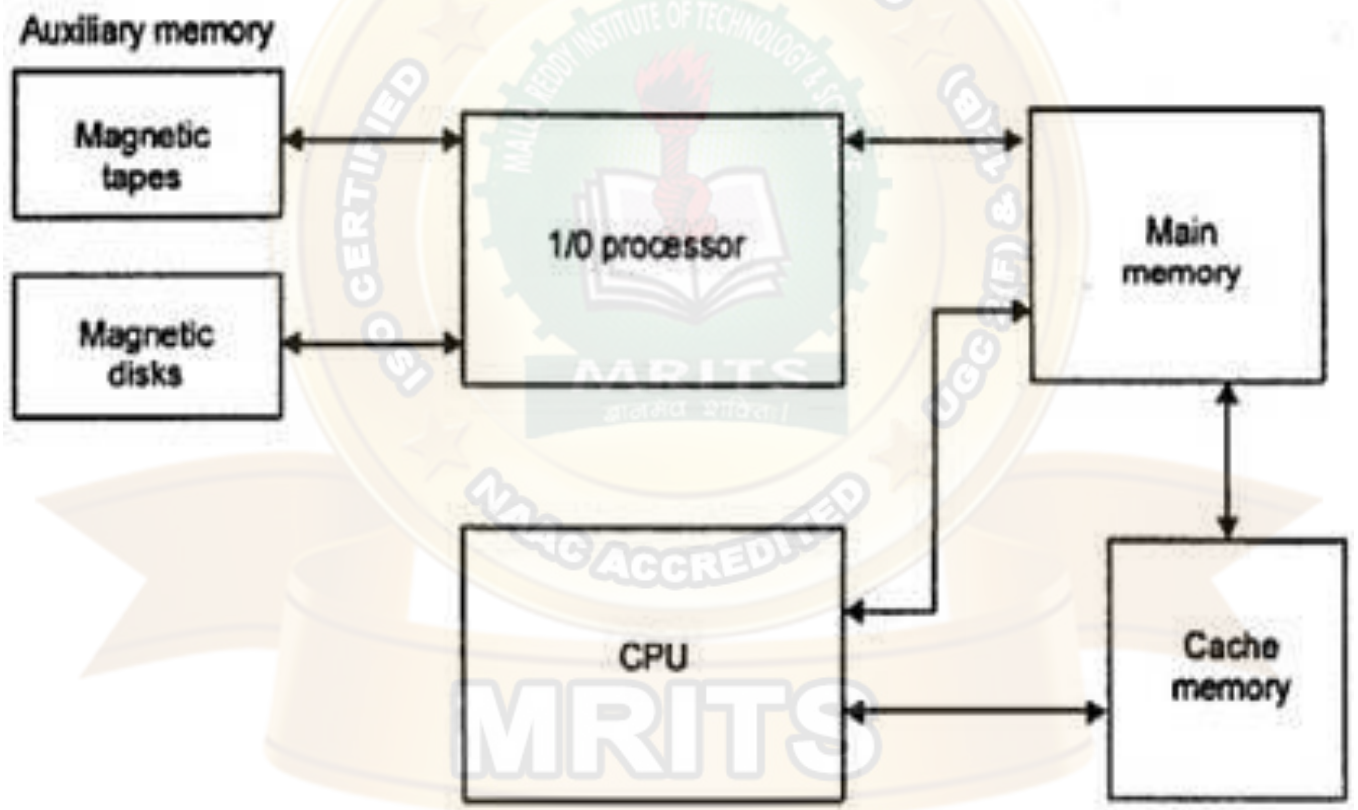
Memory Hierarchy

Memory is used for storing programs and data that are required to perform a specific task.

For CPU to operate at its maximum speed, it required an uninterrupted and high speed access to these **memories** that contain programs and data. Some of the criteria need to be taken into consideration while deciding which **memory** is to be used:

- Cost
- Speed
- Memory access time
- Data transfer rate, Reliability

How Memories attached to CPU



A computer system contains various types of memories like auxiliary memory, cache memory, and main memory.

- **Auxiliary Memory**

The auxiliary memory is at the bottom and is not connected with the CPU directly. However, being slow, it is present in large volume in the system due to its low pricing. This memory is basically used for storing the programs that are not needed in the main memory. This helps in freeing the main memory which can be utilized by other programs that needs main memory. The main function of this memory is to provide parallel searching that can be used for performing a search on an entire word.

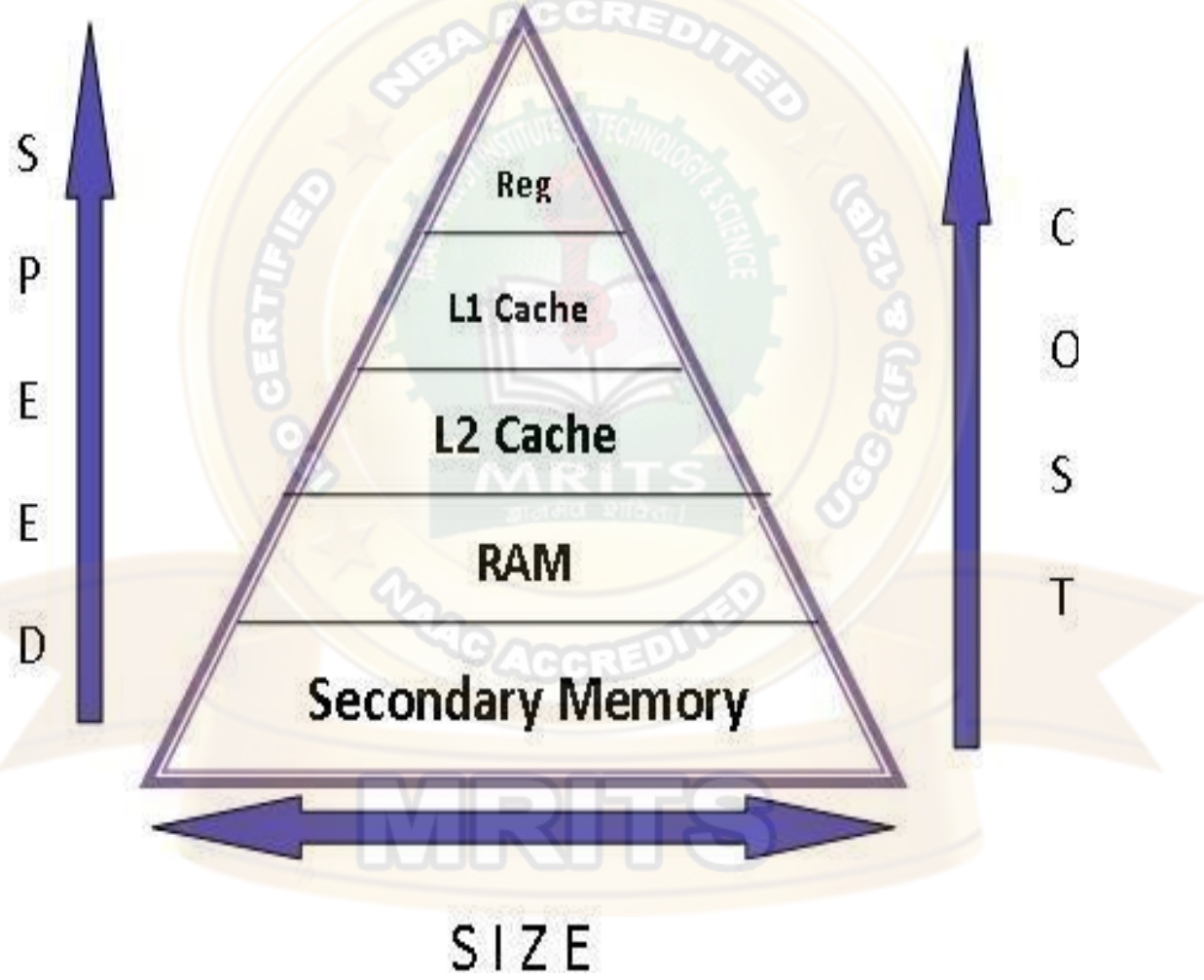
- **Main Memory**

The main memory is at the second level of the hierarchy. Due to its direct connection with the CPU, it is also known as central memory. The main memory holds the data and the programs that are needed by the CPU. The main memory mainly consists of RAM, which is available in static and dynamic mode.

- **Cache Memory**

Cache memory is at the top level of the memory hierarchy. This is a high speed

memory used to increase the speed of processing by making current programs and data available to the CPU at a rapid rate. Cache memory is usually placed between the CPU and the main memory.


SPEED
Reg
L1 Cache
L2 Cache
RAM
Secondary Memory
COST
SIZE

Main Memory

- Central storage unit in a computer system
- Large memory
- Made up of Integrated chips
- Types:

  RAM (Random access memory)

  ROM (Read only memory)

RAM (Random Access Memory)

Random access memory (RAM) is the best known form of computer memory. RAM is considered "random access" because you can access any memory cell directly if you know the row and column that intersect at that cell.

Types of RAM:-

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

- **Static RAM (SRAM)**
  - a bit of data is stored using the state of a flip-flop.
  - Retains value indefinitely, as long as it is kept powered.
  - Mostly uses to create cache memory of CPU.
  - Faster and more expensive than DRAM.

- **Dynamic RAM (DRAM)**
  - Each cell stores bit with a capacitor and transistor.
  - Large storage capacity
  - Needs to be refreshed frequently.
  - Used to create main memory.

- Slower and cheaper than SRAM.

ROM

ROM is used for storing programs that are **Permanently** resident in the computer and for tables of constants that do not change in value once the production of the computer is completed

The ROM portion of main memory is needed for storing an initial program called *bootstrap loader,* witch is to start the computer software operating when power is turned on.

There are five basic ROM types:

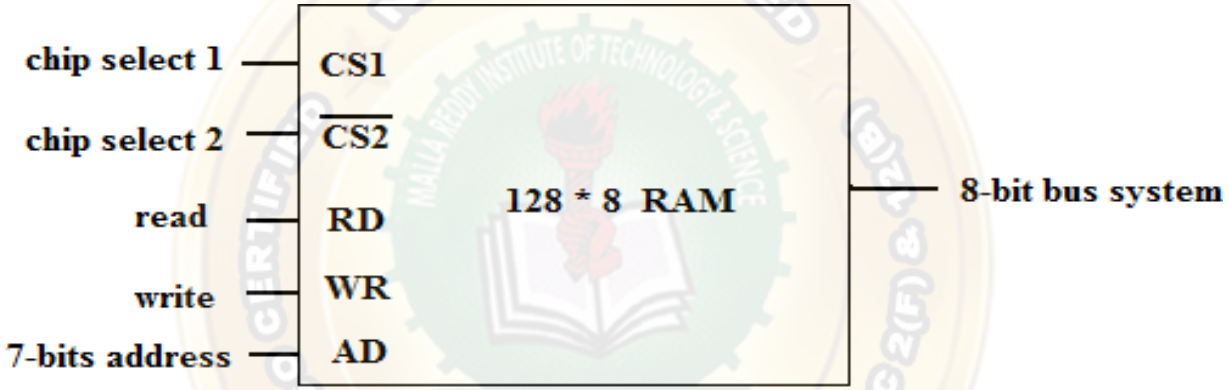
- ROM - Read Only Memory
- PROM - Programmable Read Only Memory
- EPROM - Erasable Programmable Read Only Memory
- EEPROM - Electrically Erasable Programmable Read Only Memory
- Flash EEPROM memory

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 when needed

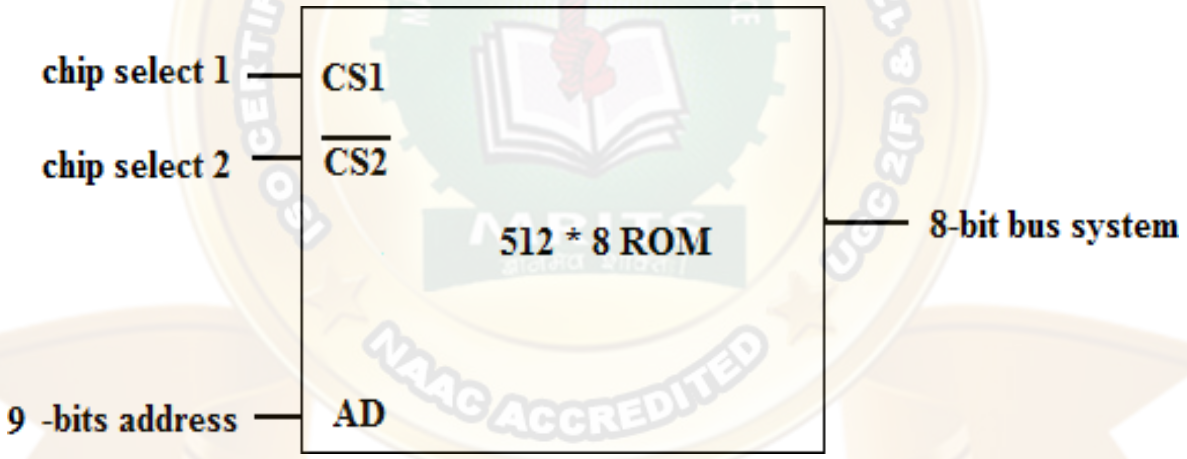
- The Block diagram of a RAM chip is shown next slide, the capacity of the memory is 128 words of 8 bits (one byte) per word

RAM



CS1	$\overline{CS2}$	RD	WD	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

ROM

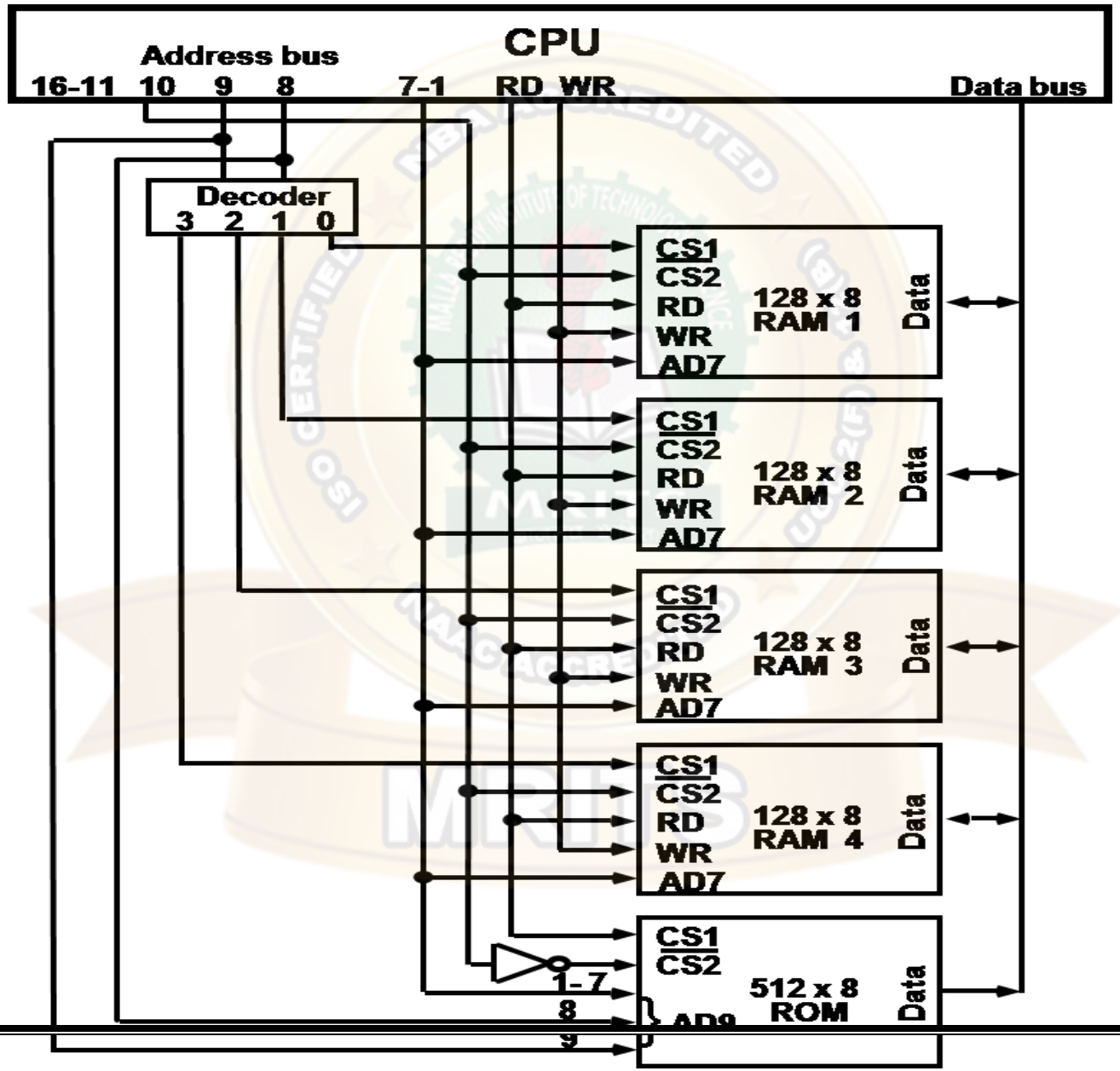

chip select 1
CS1
chip select 2
CS2
512 * 8 ROM
8-bit bus system
9 -bits address
AD

Memory Address Map

- Memory Address Map is a pictorial representation of assigned address space for each chip in the system
- To demonstrate an example, assume that a computer system needs 512 bytes of RAM and 512 bytes of ROM
- The RAM have 128 byte and need seven address lines, where the ROM have 512 bytes and need 9 address lines

Component	Hexadecimal Address	10	9	8	7	6	5	4	3	2	1
RAM1	0000-007F	0	0	0	*	*	*	*	*	*	*
RAM2	0080-00FF	0	0	1	*	*	*	*	*	*	*
RAM3	0100-017F	0	1	0	*	*	*	*	*	*	*
RAM4	0180-01FF	0	1	1	*	*	*	*	*	*	*
ROM	0200-03FF	1	*	*	*	*	*	*	*	*	*

- The hexadecimal address assigns a range of hexadecimal equivalent address for each chip
- Line 8 and 9 represent four distinct binary combination to specify which RAM we chose
- When line 10 is 0, CPU selects a RAM. And when it’s 1, it selects the ROM


CPU
Address bus
16-11 10 9 8
7-1
RD WR
Data bus
Decoder
3 2 1 0
CS1
CS2
RD
WR
AD7
128 x 8
RAM 1
Data
CS1
CS2
RD
WR
AD7
128 x 8
RAM 2
Data
CS1
CS2
RD
WR
AD7
128 x 8
RAM 3
Data
CS1
CS2
RD
WR
AD7
128 x 8
RAM 4
Data
CS1
CS2
512 x 8
ROM
Data
1-7
8
9
AD9

Memory connection to the CPU

Cache memory

- If the active portions of the program and data are placed in a fast small memory, the average memory access time can be reduced
- Thus reducing the total execution time of the program
- Such a fast small memory is referred to as cache memory
- The cache is the fastest component in the memory hierarchy and approaches the speed of CPU component

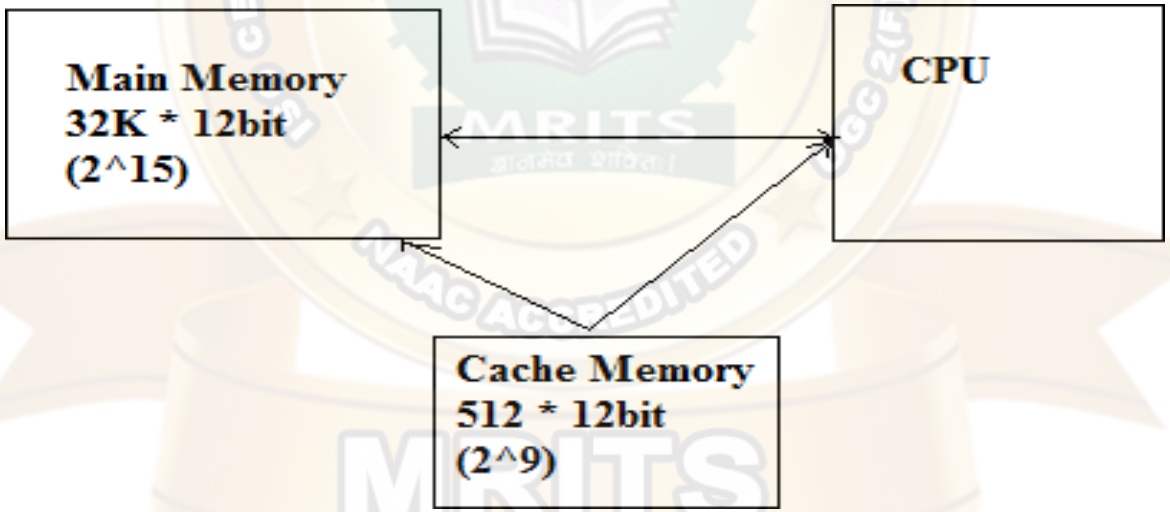
- When CPU needs to access memory, the cache is examined
- If the word is found in the cache, it is read from the fast memory
- If the word addressed by the CPU is not found in the cache, the main memory is accessed to read the word

- When the CPU refers to memory and finds the word in cache, it is said to produce a **hit**
- Otherwise, it is a **miss**
- The performance of cache memory is frequently measured in terms of a quantity called **hit ratio**

Hit ratio = hit / (hit+miss)

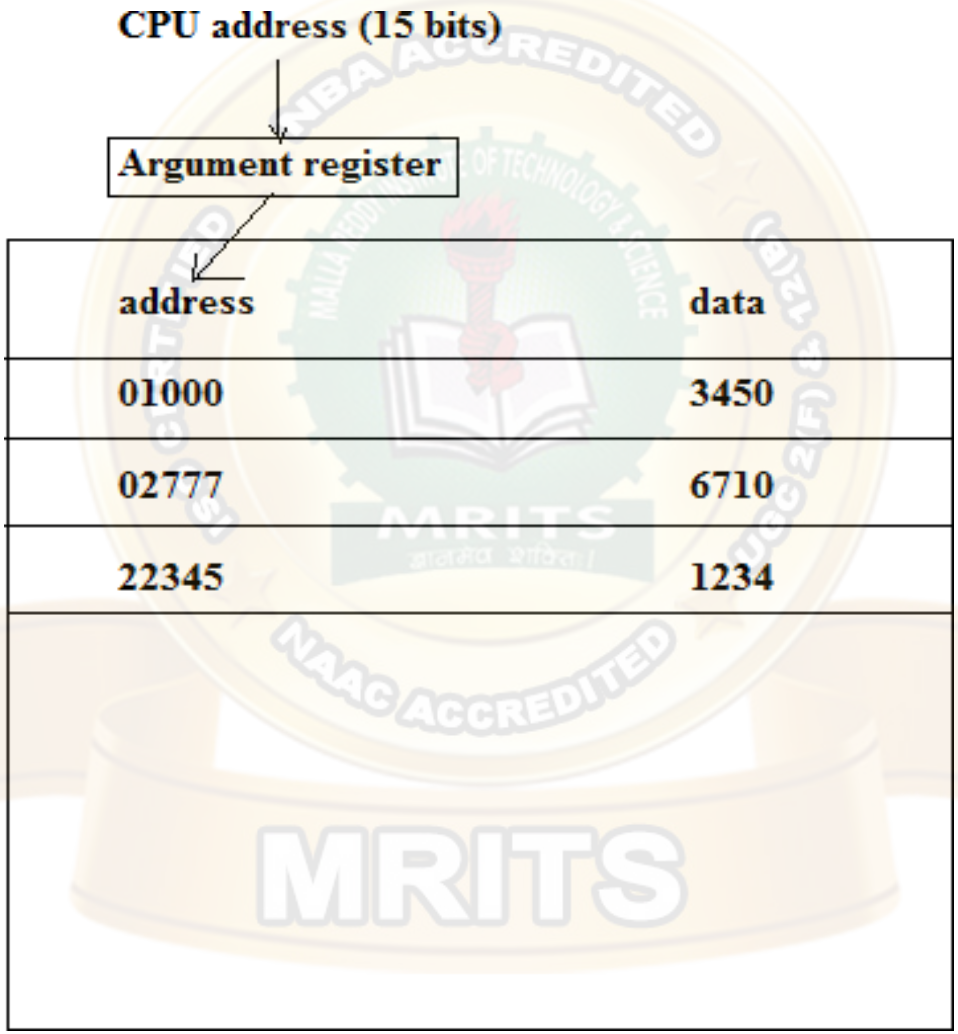
- The basic characteristic of cache memory is its fast access time
- Therefore, very little or no time must be wasted when searching the words in the cache
- The transformation of data from main memory to cache memory is referred to as a **mapping** process, there are three types of mapping:
  - **Associative mapping**
  - **Direct mapping**
  - **Set-associative mapping**

- To help understand the mapping procedure, we have the following example:



Associative mapping

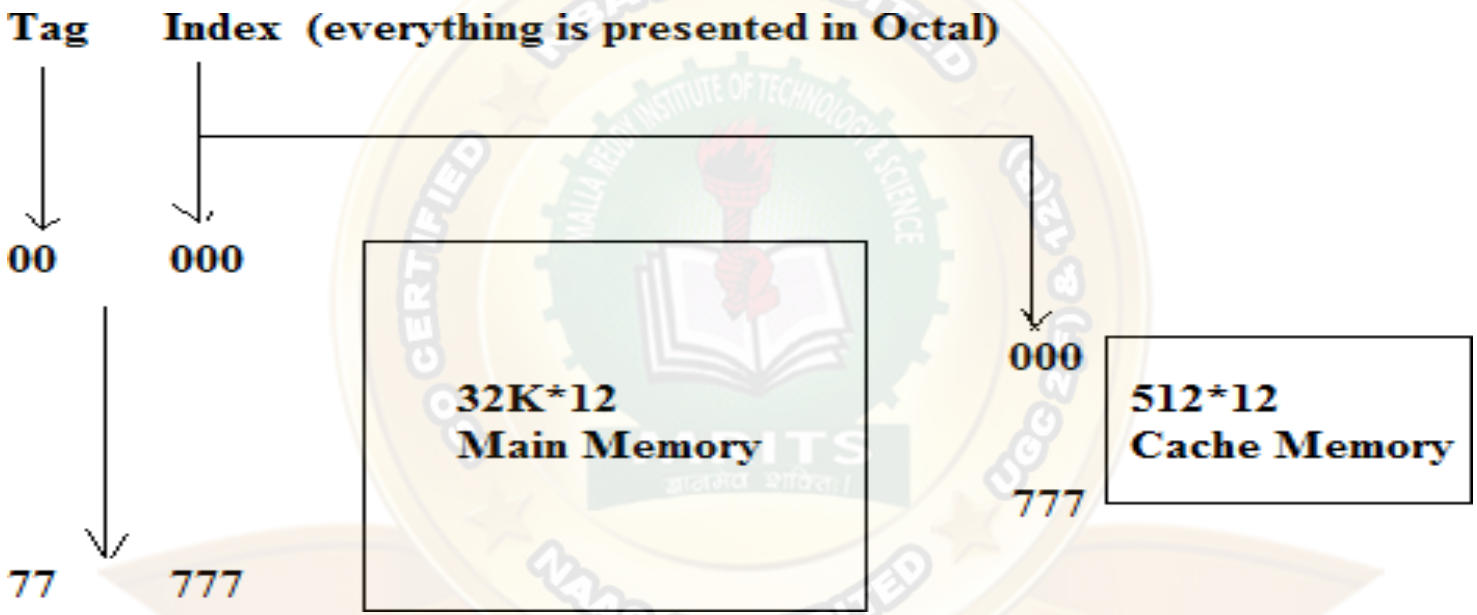
- The fastest and most flexible cache organization uses an associative memory
- The associative memory stores both the address and data of the memory word
- This permits any location in cache to store ant word from main memory
- The address value of 15 bits is shown as a five-digit **octal** number and its corresponding 12-bit word is shown as a four-digit octal number


CPU address (15 bits)
Argument register
address
data
01000
3450
02777
6710
22345
1234

- A CPU address of 15 bits is places in the argument register and the associative memory us searched for a matching address
- If the address is found, the corresponding 12-bits data is read and sent to the CPU
- If not, the main memory is accessed for the word
- If the cache is full, an address-data pair must be displaced to make room for a pair that is needed and not presently in the cache

Direct Mapping

- Associative memory is expensive compared to RAM
- In general case, there are 2^k words in cache memory and 2^n words in main memory (in our case, k=9, n=15)
- The n bit memory address is divided into two fields: k-bits for the index and n-k bits for the tag field



Addressing relationships between main and cache memories

Memory Address	Memory Data
00000	1220
00777	2340
01000	3450
01111	2222
01777	4560
02000	5670
02777	6710

Index Address	Tag	Data
000	00	1220
111	01	2222
777	02	6710

Set-Associative Mapping

- 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

- Set-Associative Mapping is an improvement over the direct-mapping in that each word of cache can store two or more word of memory

under the same index address

NBA ACCREDITED
ISO CERTIFIED
UGC 2(F) & 12(B)
NAAC ACCREDITED
MRITS
MRITS

Memory Address	Memory Data
00000	1220
00777	2340
01000	3450
01111	2222
01777	4560
02000	5670
02777	6710

Index Address	Tag	Data	Tag	Data
000	01	3450	02	5670
111	01	2222		
777	02	6710	00	2340

- Each index address refers to two data words and their associated tags
- Each tag requires six bits and each data word has 12 bits, so the word length is 2*(6+12) = 36 bits

UNIT-5

Reduced Set Instruction Set Architecture (RISC) – The main idea behind is to make hardware simpler by using an instruction set composed of a few basic steps for loading, evaluating and storing operations just like a load command will load data, store command will store the data.

Complex Instruction Set Architecture (CISC) – The main idea is that a single instruction will do all loading, evaluating and storing operations just like a multiplication command will do stuff like loading data, evaluating and storing it, hence it's complex.

Both approaches try to increase the CPU performance

- **RISC:** Reduce the cycles per instruction at the cost of the number of instructions per program.
- **CISC:** The CISC approach attempts to minimize the number of instructions per program but at the cost of increase in number of cycles per instruction.
- $$CPU\ Time = \frac{Seconds}{Program} = \frac{Instructions}{Program} X \frac{Cycles}{Instructions} X \frac{Seconds}{Cycle}$$

  Earlier when programming was done using assembly language, a need was felt to make instruction do more task because programming in assembly was tedious and error prone due to which CISC architecture evolved but with up rise of high level language dependency on assembly reduced RISC architecture prevailed.

Characteristic of RISC –

1. Simpler instruction, hence simple instruction decoding.
2. Instruction come under size of one word.
3. Instruction take single clock cycle to get executed.
4. More number of general purpose register.
5. Simple Addressing Modes.
6. Less Data types.
7. Pipeline can be achieved.

Characteristic of CISC –

1. Complex instruction, hence complex instruction decoding.
2. Instruction are larger than one word size.

3. Instruction may take more than single clock cycle to get executed.
4. Less number of general purpose register as operation get performed in memory itself.
5. Complex Addressing Modes.
6. More Data types.

Example – Suppose we have to add two 8-bit number:

- **CISC approach:** There will be a single command or instruction for this like ADD which will perform the task.
- **RISC approach:** Here programmer will write first load command to load data in registers then it will use suitable operator and then it will store result in desired location.

So, add operation is divided into parts i.e. load, operate, store due to which RISC programs are longer and require more memory to get stored but require less transistors due to less complex command.

Difference –

RISC	CISC
Focus on software	Focus on hardware
Uses only Hardwired control unit	Uses both hardwired and micro programmed control unit
Transistors are used for more registers	Transistors are used for storing complex Instructions
Fixed sized instructions	Variable sized instructions
Can perform only Register to	Can perform REG to REG or REG to

RISC	CISC
Register Arithmetic operations	MEM or MEM to MEM
Requires more number of registers	Requires less number of registers
Code size is large	Code size is small
A instruction execute in single clock cycle	Instruction take more than one clock cycle
A instruction fit in one word	Instruction are larger than size of one

The general definition of a processor or a microprocessor is: A small chip that is placed inside computer as well as other electronic devices.

In very simple terms, the main job a processor is to receive input and then provide the appropriate output (depending on the input).

Modern day processors, have become so advanced that they can handle trillions of calculations per second, increasing efficiency and performance.

Both RISC and CISC architectures have been developed largely as a breakthrough to cover the semantic gap. The semantic gap, is the gap which is present between machine language and high level language.

Therefore the main objective of creating these two architectures is to improve the efficiency of software development, and by doing so, there has been several programming languages which have been developed as a result, such as Ada, C++, C, and Java etc.

These programming languages provide a high level of power and abstraction.

Therefore to allow for efficient compilation of these high level language programs, RISC and CISC are used.

What are RISC processors?

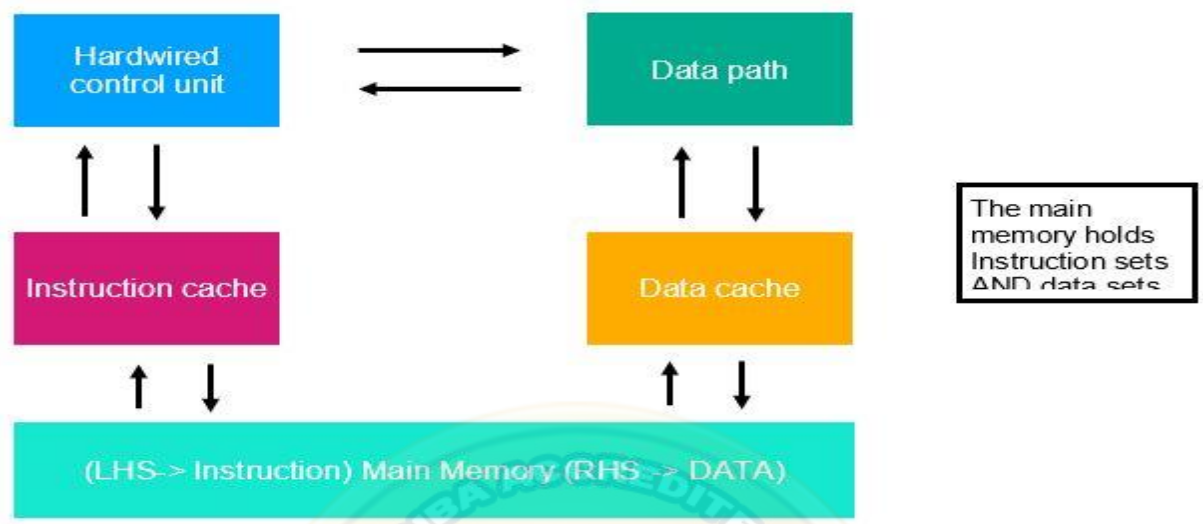
Reduced Instruction Set Computer (RISC), is a type of computer architecture which operates on small, highly optimised set of instructions, instead of a more specialised set of instructions, which can be found in other types of architectures. This architecture means that the computer microprocessor will have fewer cycles per instruction.

The word "Reduced Instruction Set" may be incorrectly interpreted to refer to "reduced number of instructions". Though this is not the case, the term actually means that the amount of work done by each instruction is decreased in terms of number of cycles.

Due to the design of Alan Turing 1946 Automatic Computing Engine, it had many characteristics that resembled RISC architecture, furthermore many traits of RISC architectures were seen in the 1960s due to them embodying the load/store approach.

That being said the term RISC had first been used by David Patterson of "Berkeley RISC project", who is considered to be a pioneer in his RISC processor designs. Patterson is currently the Vice Chair of Board of Directors of the RISC-V Foundation.

A RICS chip doesn't require many transistors, which makes them less costly to design and to produce. One of RISCs main characteristics is that the instruction set contains relatively simple and basic instruction from which more complex instructions can be produced.



Some the terminology which can be handy to understand:

- **LOAD:** Moves data from the memory bank to a register.
- **PROD:** Finds product of two operands located within the register.
- **STORE:** Moves data from a register to the memory banks.

Addressing modes: An address mode is an aspect of instruction set architecture in most CPU designs.

- The RISC architecture utilises simple instructions.
- RISC synthesises complex data types and supports few simple data types.
- RISC makes use of simple addressing modes and fixed length instructions for pipelining.
- RISC allows any register to be used in any context.
- RISC has only one cycle for execution time.
- The work load of a computer that has to be performed is reduced by operating the “LOAD” and “STORE” instructions.
- RISC prevents various interactions with memory, it does this by have a large number of registers.
- Pipelining in RISC is carried out relatively simply. This is due to the execution of instructions being done in a uniform interval of time (i.e. one click).
- More RAM is required to store assembly level instructions.
- Reduced instructions need a smaller number of transistors in RISC.
- RISC utilises the Harvard architecture
- To execute the conversion operation, a compiler is used. This allows the conversion of high-level language statements into code of its form.

- RISC processors utilise pipelining.
  - Pipelining is a process that involves improving the performance of the CPU. The process is completed by fetching, decoding, and executing cycles of three separate instructions at the same time.

A RISC architecture systems contains a small core logic processor, which enables engineers to increase the register set and increase internal parallelism by using the following techniques:

Thread Level Parallelism:
Thread level parallelism increases the number of parallel threads executed by the CPU.

Thread level parallelism can also be identified as "Task Parallelism", which is a form of parallel computing for multiple computer processors, using a technique for distributing the execution of processes and threads across different parallel processor nodes. This type of parallelism is mostly used in multitasking operating systems, as well as applications that depend on processes and threads.

Instruction Level Parallelism:
Instructions level parallelism increases the speed of the CPU in executing instructions. This type of parallelism that measures how many of the instructions in a computer can be executed simultaneously.

However Instruction level parallelism is not to be confused with concurrency. Instruction level parallelism is about the parallel election of a sequence of instructions, which belong to a specific thread of execution of a process.

Whereas concurrency is about threads of one or different processes being assigned by the CPU's core in a mannered and strict alteration or in true parallelism (provided that there are enough CPU cores).

Advantages of RISC processors

- Due to the architecture having a set of instructions, this allows high level language compilers to produce more efficient code.
- This RISC architecture allows simplicity, which therefore means that it allows developers the freedom to utilise the space on the microprocessor.
- RISC processors make use of the registers to pass arguments and to hold local variables.

- RISC makes use of only a few parameters, furthermore RISC processors cannot call instructions, and therefore, use a fixed length instruction, which is easy to pipeline.
- Using RISC, allows the execution time to be minimised, whilst increasing the speed of the overall operation, maximising efficiency.
- As mentioned above, RISC is relatively simple, this is due to having very few instructional formats, and a small number of instructions and a few addressing modes required.

Disadvantages of RISC processors

- The performance of RISC processors depends on the compiler or the programmer. The following instructions might rely on the previous instruction to finish their execution.
- RISC processors require very fast memory systems to feed various instructions, thus a large memory cache is required.

What are CISC processors?

CISC, which stands for "Complex Instruction Set Computer", is computer architecture where single instructions can execute several low level operations, for instance, "load from memory an arithmetic operation, and a memory store). CISC processors are also capable of executing multi-step operations or addressing modes with single instructions.

CISC, as with RISC, is a type of microprocessor that contains specialised simple/complex instructions.

Until recent times, all major manufacturers of microprocessors had used CISC based designs to develop their products. The reason for that was because, CISC was introduced around the early 1970's, where it was used for simple electronic platforms, such as stereos, calculators, video games, **not personal computers**, therefore allowing the CISC technology to be used for these types of applications, as it was more suitable.
However, eventually, CISC microprocessors found their way into personal computers, this was to meet the increasing need of PC users. CISC manufactures started to focus their efforts from general-purpose designs to a high performance computing orientation.

Advantageously, CISC processors helped in simplifying the code and making it shorter in order to reduce the memory requirements.

In CISC processors, each single instruction has several low level operations. Yes, this makes CISC instructions short, but complex.

Some examples of CISC processors are:

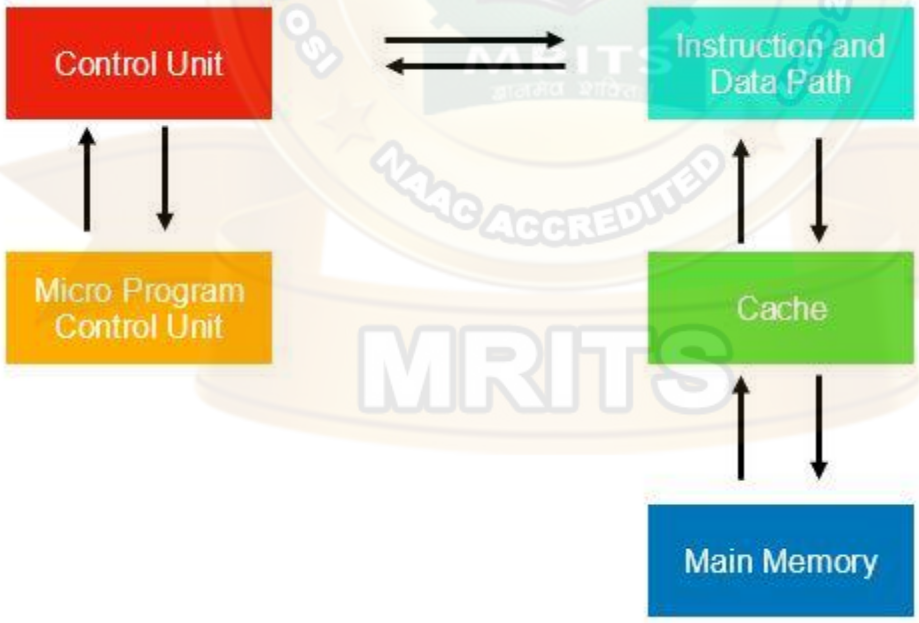
- IBM 370/168 and Intel 80486
- Also non-trivial items such as government databases were built using a CISC processor

The characteristics of CISC processors

As mentioned above, the main objective of CISC processors is to minimise the program size by decreasing the number of instructions in a program.

However to do this, CISC has to embed some of the low level instructions in a single complex instruction. Moreover, this means that when it is decoded, this instruction generates several microinstructions to execute.

The complex architecture of CISC is below:



Microprogram Control Unit:

The microprogram control unit uses a series of microinstructions of the microprogram stored in the "control memory" of the microprogram control unit and generate control signals.

Control Unit:
The control units access the control signals, which are produced by the microprogram control unit, moreover they operate the functioning of processors hardware.

Instructions and data path:
The instructions and the data path retrieve/fetches the opcode and operands of the instructions from the memory.

Cache and main memory:
This is the location where the program instructors and operands are stored.
Instructions in CISC are complex, and they occupy more than a single word in memory. Like we saw in RISC, CISC also uses LOAD/STORE to access the memory operands, however CISC also has a "MOVE" instruction attribute, which is used to gain access to memory operands.

Though one advantageous characteristic of the "MOVE" operation, is that it has a wider scope. This allows the CISC instructions to directly access memory operands.

CISC instruction sets also have additional addressing modes:

- **Auto-increment mode:**
  - The address of an operand is the content of the register. It is automatically incremented after accessing the registers content, in order to point to the memory location of the next operand.
- **Auto-decrement mode:**
  - Like "auto-increment", the address of an operand is the content of the register. However with auto-decrement, initially the contest of register is decremented, moreover then the content of the register is used as an address for an operand.
- **Relative Mode:**
  - The program counter is used instead of a general-purpose register. This allows to refer large range of area in memory.

Advantages of CISC processors

- Memory requirement is minimised due to code size.
- The execution of a single instruction will also execute and complete several low level tasks.
- Memory access is more flexible due to the complex addressing mode.

- Memory locations can be directly accessed by CISC instructions.
- Microprogramming is easy to implement and less expensive than wiring a control unit.
- If new commands are to be added to the chip, the structure of the instruction set does not need to be changed. This is because the CISC architecture uses general purpose hardware to carry out commands.
- The compiler doesn't have to be complicated, as the microprogram instruction sets can be written to match the high-level language constructs.

Disadvantages of CISC processors

- Although the code size is minimised, the code requires several clock cycles to execute a single instruction. Therefore decreasing the efficiency of the system.
- The implementation of pipelining in CISC is regarded to be complicated.
- In order to simplify the software, the hardware structure needs to be more complex.
- CISC was designed to minimise the memory requirement when memory was smaller and more expensive. However nowadays memory is inexpensive and the majority of new computer systems have a large amount of memory, compared to the 1970's when CISC first emerged.

RISC vs. CISC

RISC	**CISC**
RISC focuses on software	CISC focuses on hardware
Single clock, reduced instruction only, which means the instructions are simple compared to CISC	Multi-clock complex instructions

RISC	CISC
Operates on Register to Register. However "LOAD" and "STORE" are independent instructions	CISC operates from Memory to Memory: The "LOAD" and "STORE" incorporated in instructions. Also uses MOVE
RISC has large code sizes, which means it operates low cycles per second	CISC has small code sizes, high cycles per second
Spends more transistors on memory registers	The transistors in a CISC processor are used to store complex instructions
Less memory access	More memory access
Implementing pipelining on RISC is easier	Due to CISC instructions being of variable length, and having multiple operands, as well as complex addressing modes and complex instructions this increases complexity. Furthermore, CISC as defined above, occupies more than a memory word. Thus taking several cycles to execute operand fetch. Implementing pipelining on CISC is complicated

Although the above showcases differences between the two architectures, the main difference between RISC and CISC is the CPU time taken to execute a given program.

CPU execution time is calculated using this formula:

CPU time = (number of instruction) x (average cycles per instruction) x (seconds per cycle)

RISC architectures will shorten the execution time by reducing the average clock cycle per one instruction.

However, CISC architectures try to reduce execution time by reducing the number of instructions per program.

Summary and Facts

A reduced Instruction Set Computer (RISC), can be considered as an evolution of the alternative to Complex Instruction Set Computing (CISC). With RISC, in simple terms, its function is to have simple instructions that do less but execute very quickly to provide better performance.

What are RISC processors?

- Reduced Instruction Set Computer (RISC), is a type of computer architecture which operates on small, highly optimised set of instructions, instead of a more specialised set of instructions, which can be found in other types of architectures. This architecture means that the computer microprocessor will have fewer cycles per instruction.
- RISC processors/architectures are used across a wide range of platforms nowadays, ranging from tablet computers to smartphones, as well as supercomputers
- **Thread Level Parallelism:**
  - Thread level parallelism increases the number of parallel threads executed by the CPU.
- **Instruction Level Parallelism:**
  - Instructions level parallelism increases the speed of the CPU's executing instructions.

Advantages and Disadvantages of RISC processors

Advantages:

- Greater performance due to simplified instruction set
- Uses pipelining efficiently
- RISC can be easily designed in compared to CISC
- Less expensive, as they use smaller chips

Disadvantages:

- Performance of the processor will depend on the code being executed
- RISC processors require very fast memory systems to feed different instructions. This requires a large memory cache.

The characteristics of RISC processor structure:

- Hardwired Control Unit
- Data Path
- Instruction Cache

- Data Cache
- Main Memory
- Only Load and store instructions have access to memory
- Fewer number of addressing modes
- RISC includes a less complex pipelining architecture compared to CISC

What are CISC processors?

- CISC, which stands for "Complex Instruction Set Computer", is computer architecture where single instructions can execute several low level operations. CISC processors are also capable of executing multi-step operations or addressing modes with single instructions.
- CISC, as with RISC, is a type of microprocessor that contains specialised simple/complex instructions.
- The primary objective for CISC processors is to complete a task in as few lines of assembly as possible. To accomplish this, processor hardware must be built able to comprehend and execute a series of operations.

Advantages and disadvantages of CISC processors:

Advantages:

- Allows for simple small scripts
- Using CISC, complex commands are readable
- Most code is built to be implemented on CISC

Disadvantages:

- CISC processors are larger as they contain more transistors
- May take multiple cycles per line of code, decreasing efficiency
- Lower clock speed
- Complex use of pipelining
- Compared to RISC, they are more complex, which means they are more expensive

The characteristics of CISC processor structure:

- Microprogram Control Unit
- Control Unit
- Instructions and data path
- Cache and main memory

CISC instruction sets also have additional addressing modes:

- Auto-increment mode
- Auto-decrement mode
- Relative Mode
- CISC uses STORE/LOAD/MOVE

Unit- 5 (b) Pipelining and Vector Processing

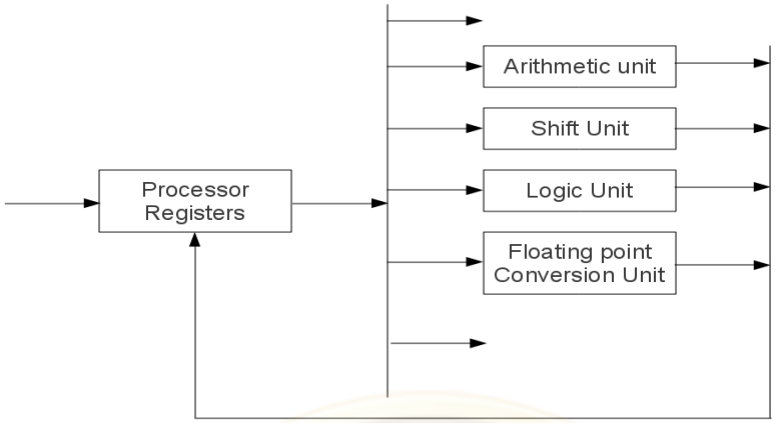
Parallel Processing:

The term parallel processing indicates that the system is able to perform several operations in a single time. Now we will elaborate the scenario, in a CPU we will be having only one Accumulator which will be storing the results obtained from the current operation. Now if we are giving only one command such that "a+b" then the CPU performs the operation and stores the result in the accumulator. Now we are talking about parallel processing, therefore we will be issuing two instructions "a+b" and "c-d" in the same time, now if the result of "a+b" operation is stored in the accumulator, then "c-d" result cannot be stored in the accumulator in the same time. Therefore the term parallel processing in not only based on the Arithmetic, logic or shift operations. The above problem can be solved in the following manner. Consider the registers R1 and R2 which will be storing the operands before operation and R3 is the register which will be storing the results after the operations. Now the above two instructions "a+b" and "c-d" will be done in parallel as follows.

- Values of "a" and "b" are fetched in to the registers R1 and R2
- The values of R1 and R2 will be sent into the ALU unit to perform the addition
- The result will be stored in the Accumulator
- When the ALU unit is performing the calculation, the next data "c" and "d" are brought into R1 and R2.
- Finally the value of Accumulator obtained from "a+b" will be transferred into the R3
- Next the values of C and D from R1 and R2 will be brought into the ALU to perform the "c-d" operation.
- Since the accumulator value of the previous operation is present in R3, the result of "c-d" can be safely stored in the Accumulator.

This is the process of parallel processing of only one CPU. Consider several such CPU performing the calculations separately. This is the concept of parallel processing.

Concept of Parallel Processing



In the above figure we can see that the data stored in the processor registers is being sent to separate devices basing on the operation needed on the data. If the data inside the processor registers is requesting for an arithmetic operation, then the data will be sent to the arithmetic unit and if in the same time another data is requested in the logic unit, then the data will be sent to logic unit for logical operations. Now in the same time both arithmetic operations and logical operations are executing in parallel. This is called as parallel processing.

Instruction Stream: The sequence of instructions read from the memory is called as an Instruction Stream

Data Stream: The operations performed on the data in the processor is called as a Data Stream.

The computers are classified into 4 types based on the Instruction Stream and Data Stream. They are called as the Flynn's Classification of computers.

Flynn's Classification of Computers:

- Single Instruction Stream and Single Data Stream (SISD)
- Single Instruction Stream and Multiple Data Stream (SIMD)
- Multiple Instruction Stream and Single Data Stream (MISD)
- Multiple Instruction Stream and Multiple Data Stream (MIMD)

SISD represents the organization of a single computer containing a control unit, a processor unit and a memory unit. Instructions are executed sequentially and the system may or may not have internal parallel processing capabilities. Parallel processing in this case may be achieved by means of multiple functional units or by pipeline processing.

SIMD represents an organization that includes many processing units under the supervision of a common control unit. All processors receive the same instruction from the control unit but operate on different items of data. The shared memory unit must contain multiple modules so that it can communicate with all the processors simultaneously.

MISD structure is only of theoretical interest since no practical system has been constructed using this organization because Multiple instruction streams means more no of instructions, therefore we have to perform multiple instructions on same data at a time. This is practically impossible.

MIMD structure refers to a computer system capable of processing several programs at the same time operating on different data.

Pipelining: Pipelining is a technique of decomposing a sequential process into sub operations, with each sub process being executed in a special dedicated segment that operates concurrently with all other segments. We can consider the pipelining concept as a collection of several segments of data processing programs which will be processing the data and sending the results to the next segment until the end of the processing is reached. We can visualize the concept of pipelining in the example below.

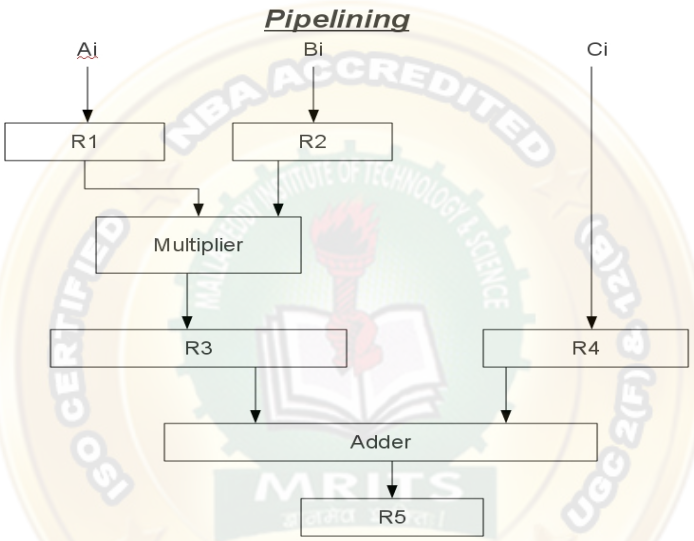
Consider the following operation: Result=(A+B)*C

- First the A and B values are Fetched which is nothing but a “Fetch Operation”.
- The result of the Fetch operations is given as input to the Addition operation, which is an Arithmetic operation.

- The result of the Arithmetic operation is again given to the Data operand C which is fetched from the memory and using another arithmetic operation which is Multiplication in this scenario is executed.
- Finally the Result is again stored in the "Result" variable.

In this process we are using up-to 5 pipelines which are the

→ Fetch Operation (A)| Fetch Operation(B) | Addition of (A & B) | Fetch Operation(C) | Multiplication of ((A+B), C) | Load ((A+B)*C), Result);

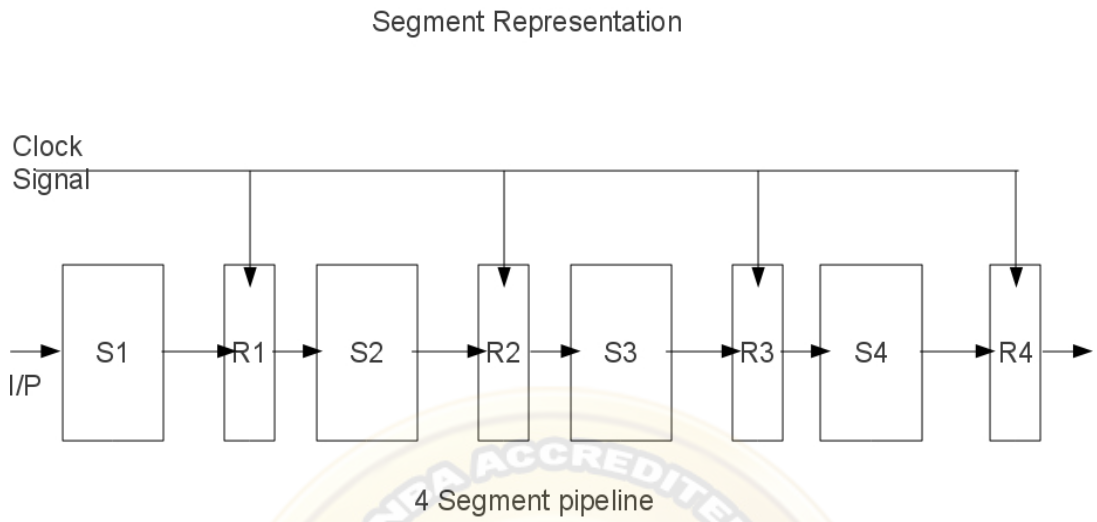


The contents of the Registers in the above pipeline concept are given below. We are considering the implementation of A[7] array with B[7] array.

Clock Pulse Number	Segment1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	A1	B1	-	-	-
2	A2	B2	A1*B1	C1	-
3	A3	B3	A2*B2	C2	A1*B1+C1
4	A4	B4	A3*B3	C3	A2*B2+C2
5	A5	B5	A4*B4	C4	A3*B3+C3
6	A6	B6	A5*B5	C5	A4*B4+C4
7	A7	B7	A6*B6	C6	A5*B5+C5
8			A7*B7	C7	A6*B6+C6
9					A7*B7+C7

If the above concept is executed with out the pipelining, then each data operation will be taking 5 cycles, totally they are 35 cycles of CPU are needed to perform the operation. But if are using the concept of pipeline, we will be cutting off many cycles. Like given in the table below when the values of A1 and B1 are coming into the registers R1 and R2, the registers R3, R4 and R5 are empty. Now in the second cycle the multiplication of A1 and B1 is transferred to register R3, now in this point the contents of the register R1 and R2 are empty. Therefore the next two values A2 and B2 can be brought into the registers. Again in the third cycle after fetching the C1 value the operation (A1*B1)+C1 will be performed. So in this way we can achieve the total concept in only 9 cycles. Here we are assuming that the clock cycle timing is fixed. This is the concept of pipelining.

Below is the diagram of 4 segment pipeline.

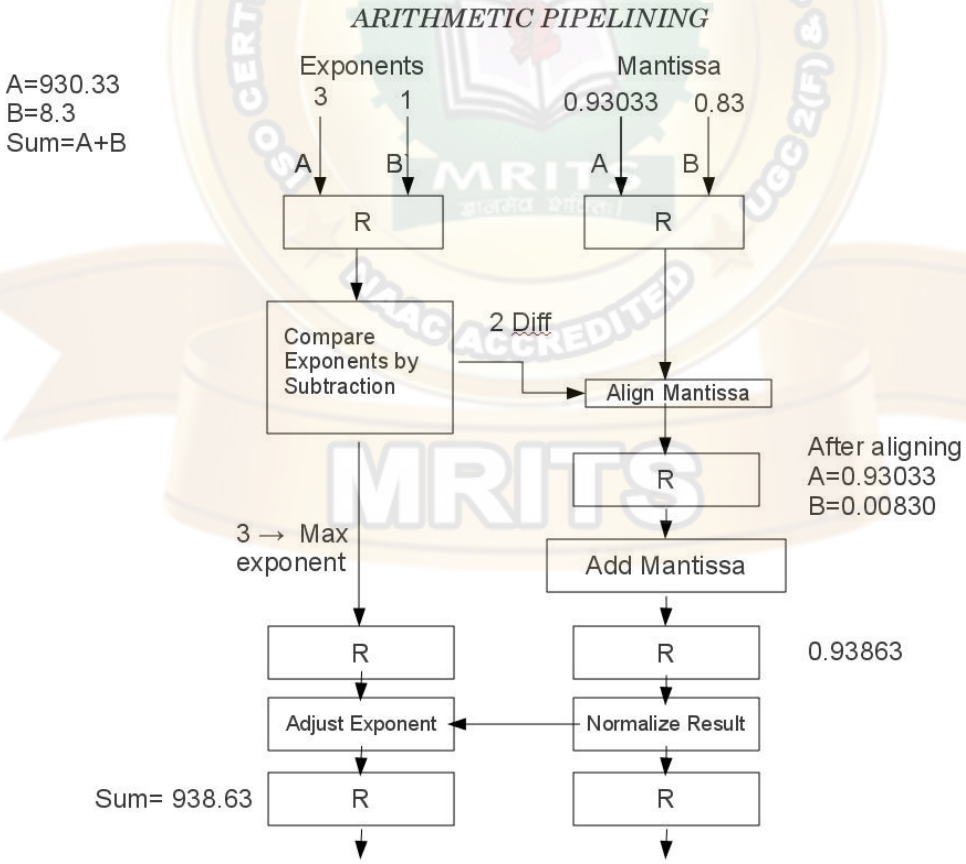


Space and Time Diagram

Seg/ clock	C1	C2	C3	C4	C5	C6	C7	C8	C9
S1	T1	T2	T3	T4	T5	T6			
S2		T1	T2	T3	T4	T5	T6		
S3			T1	T2	T3	T4	T5	T6	
S4				T1	T2	T3	T4	T5	T6

S= nTn/(K+n-1)*tp

The below table is the space time diagram for the execution of 6 tasks in the 4 segment pipeline.



Arithmetic pipeline:

The above diagram represents the implementation of arithmetic pipeline in the area of floating point arithmetic operations. In the diagram, we can see that two numbers A and B are added together. Now the values of A and B are not normalized, therefore we must normalize them before start to do

any operations. The first thing is we have to fetch the values of A and B into the registers. Here R denote a set of registers. After that the values of A and B are normalized, therefore the values of the exponents will be compared in the comparator. After that the alignment of mantissa will be taking place. Finally, we will be performing addition, since an addition is happening in the adder circuit. The source registers will be free and the second set of values can be brought. Like wise when the normalizing of the result is taking place, addition of the new values will be added in the adder

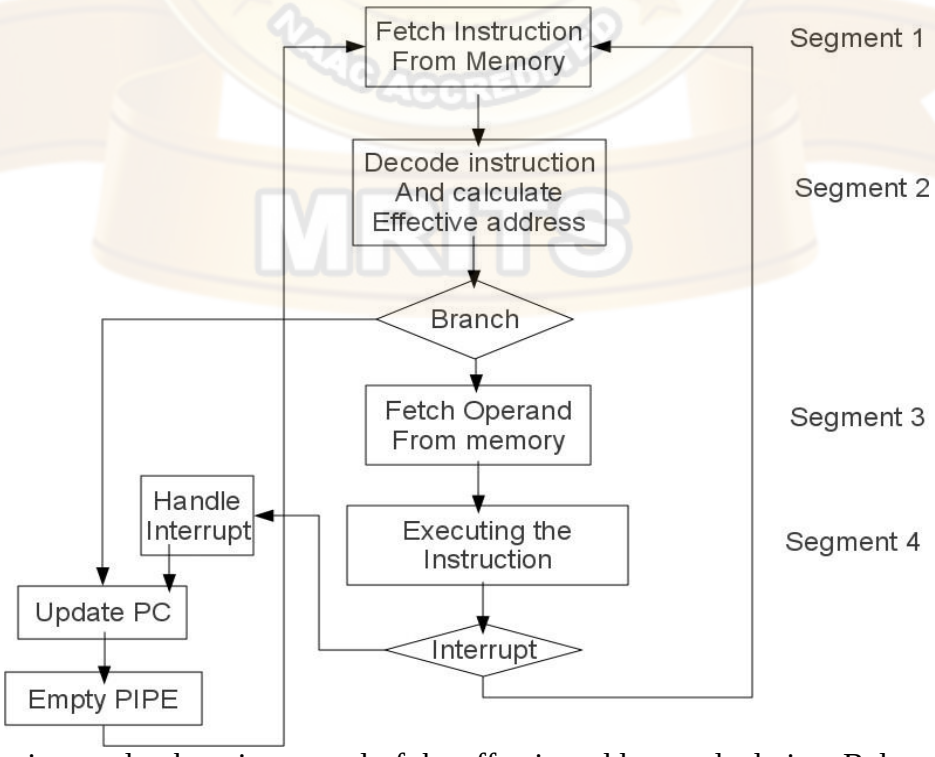
circuit and when addition is going on, the new data values will be brought into the registers in the start of the implementation. We can see how the addition is being performed in the diagram.

Instruction Pipeline: Pipelining concept is not only limited to the data stream, but can also be applied on the instruction stream. The instruction pipeline execution will be like the queue execution. In the queue the data that is entered first, will be the data first retrieved. Therefore when an instruction is first coming, the instruction will be placed in the queue and will be executed in the system. Finally the results will be passing on to the next instruction in the queue. This scenario is called as Instruction pipelining. The instruction cycle is given below

- Fetch the instruction from the memory
- Decode the instruction
- calculate the effective address
- Fetch the operands from the memory
- Execute the instruction
- Store the result in the proper place.

In a computer system each and every instruction need not necessary to execute all the above phases. In

Instruction pipelining



a Register addressing mode, there is no need of the effective address calculation. Below is the example

of the four segment instruction pipeline.

In the above diagram we can see that the instruction which is first executing has to be fetched from the memory, there after we are decoding the instruction and we are calculating the effective address. Now we have two ways to execute the instruction. Suppose we are using a normal instruction like ADD, then the operands for that instruction will be fetched and the instruction will be executed. Suppose we are executing an instruction such as Fetch command. The fetch command itself has internally three more commands which are like ACTDR, ARTDR etc.., therefore we have to jump to that particular location to execute the command, so we are using the branch operation. So in a branch operation, again other instructions will be executed. That means we will be updating the PC value such that the instruction can be executed. Suppose we are fetching the operands to perform the original operation such as ADD, we need to fetch the data. The data can be fetched in two ways, either from the main memory or else from an input output devices. Therefore in order to use the input output devices, the devices must generate the interrupts which should be handled by the CPU. Therefore the handling of interrupts is also a kind of program execution. Therefore we again have to start from the starting of the program and execute the interrupt cycle.

The different instruction cycles are given below:

- FI → FI is a segment that fetches an instruction
- DA → DA is a segment that decodes the instruction and identifies the effective address.
- FO → FO is a segment that fetches the operand.
- EX → EX is a segment that executes the instruction with the operand.

Timing of Instruction Pipeline

FI → Fetch Instruction
FO → Fetch Operand
DA → Decode instruction and Fetch Effective Address
EX → Execute the Instruction

Step	1	2	3	4	5	6	7	8	9	10	11	12	13
1	FI	DA	FO	EX									
2		FI	DA	FO	EX								
3			FI	DA	FO	EX							
4				FI	-	-	FI	DA	FO	EX			
5					-	-	-	FI	DA	FO	EX		
6									FI	DA	FO	EX	
7										FI	DA	FO	EX

Pipelining Conflicts: There are different conflicts that are caused by using the pipeline concept. They are

- Resource Conflicts: These are caused by access to memory by two or more segments at the same time. Most of these conflicts can be resolved by using separate instruction and data memories
- Data Dependency: These conflicts arise when an instruction depends on the result of a previous instruction, but this result is not yet available.
- Branch difficulties: These difficulties arise from branch and other instructions that change the value of PC.

Data Dependency Conflict: The data dependency conflict can be solved by using the following methods.

- Hardware Interlocks: The most straight forward method is to insert hardware interlocks. An interlock is a circuit that detects instructions whose source operands are destination of instructions farther up in the pipeline. Detection of this situation causes the instruction whose source is not available to be delayed by enough clock cycles to resolve the conflict. This approach maintains the program sequence by using hardware to insert the required delay.
- Operand Forwarding: Another technique called operand forwarding uses special hardware to detect a conflict and avoid the conflict path by using a special path to forward the values between the pipeline segments.
- Delayed Load: The delayed load operation is nothing but when executing an instruction in the pipeline, simply delay the execution starting of the instruction such that all the data that is needed for the instruction can be successfully updated before execution.

Branch Conflicts:

The following are the solutions for solving the branch conflicts that are obtained in the pipelining concept.

- Pre-fetch Target Instruction: In this the branch instructions which are to be executed are pre-fetched to detect if any errors are present in the branch before execution.
- Branch Target Buffer: BTB is the associative memory implementation of the branch conditions.
- Loop buffer: The loop buffer is a very high speed memory device. Whenever a loop is to be executed in the computer. The complete loop will be transferred in to the loop buffer memory and will be executed as in the cache memory.

- Branch Prediction: The use of branch prediction is such that, before a branch is to be executed, the instructions along with the error checking conditions are checked. Therefore we will not be going into any unnecessary branch loops.
- Delayed Branch: The delayed branch concept is same as the delayed load process in which we are delaying the execution of a branch process, before all the data is fetched by the system for beginning the CPU.

RISC Pipeline:

The ability to use the instruction pipelining concept in the RISC architecture is very efficient. The simplicity of the instruction set can be utilized to implement an instruction pipeline using a small number of sub operations, with each being executed in one clock cycle. Due to fixed length instruction format, the decoding of the operation can occur at the same time as the register selection. Since the arithmetic, logic and shift operations are done on register basis, there is no need for extra fetching or effective address decoding steps to perform the operation. So pipelining concept can be effectively used in this scenario. Therefore the total operations can be categorized as one segment will be fetching the instruction from program memory, the other segment executes the instruction in the ALU and the third segment may be used to store the result of the ALU operation in a destination register. The data transfer instructions in RISC are limited to only Load and Store instructions. To prevent conflicts in data transfer, we will be using two separate buses one for storing the instructions and other for storing the data.

Example of three segment instruction pipeline:
We want to perform a operation in which there is some arithmetic, logic or shift operations. Therefore as per the instruction cycle, we will be having the following steps:

- I: Instruction Fetch
- A: ALU Operation
- E: Execute Instruction.

The I segment will be fetching the instruction from program memory. The instruction is decoded and an ALU operation is performed in the A segment. In the A segment the ALU operation instruction will be fetched and the effective address will be retrieved and finally in the E segment the instruction will be executed.

Delayed Load:

Consider the following instructions:

1. LOAD: R1 ← M[address 1]
2. LOAD: R2 ← M[address 2]
3. ADD: R3 ← R1 + R2
4. STORE: M[address 3] ← R3

Pipeline timing with data conflict

Clock Cycles	1	2	3	4	5	6
1.Load R1	I	A	E			
2.Load R2		I	A	E		
3. Add R1+R2			I	A	E	
4. Store R3				I	A	E

Pipeline timing with delayed load

Clock Cycles	1	2	3	4	5	6	7
1.Load R1	I	A	E				
2.Load R2		I	A	E			
3. No Operation			I	A	E		
4. Add R1+R2				I	A	E	
5. Store R3					I	A	E

The below tables will be showing the pipelining concept with the data conflict and without data conflict.

Vector Processing:

Normal computational systems are not enough in some special processing requirements. Such as, in special processing systems like artificial intelligence systems and some weather forecasting systems, terrain analysis, the normal systems are not sufficient. In such systems the data processing will be involving on very high amount of data, we can classify the large data as a very big arrays. Now if we want to process this data, naturally we will need new methods of data processing. The vectors are considered as the large one dimensional array of data. The term vector processing involves the data processing on the vectors of such large data.

The vector processing system can be understand by the example below. Consider a program which is adding two arrays A and B of length 100;

Machine level program

```
        Initialize I=0
20      Read
        A(I)
        Read
        B(I)
        Store C(I)=A(I)+B(I)
        Increment I=I+1
        If I<=100 go to
        20 continue
```

so in this above program we can see that the two arrays are being added in a loop format. First we are starting from the value of 0 and then we are continuing the loop with the addition operation until the I value has reached to 100. In the above program there are 5 loop statements which will be executing 100 times. Therefore the total cycles of the CPU taken is 500 cycles. But if we use the concept of vector processing then we can reduce the unnecessary fetch cycles, since the fetch cycles are used in the creation of the vector. The same program written in the vector processing statement is given below.

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

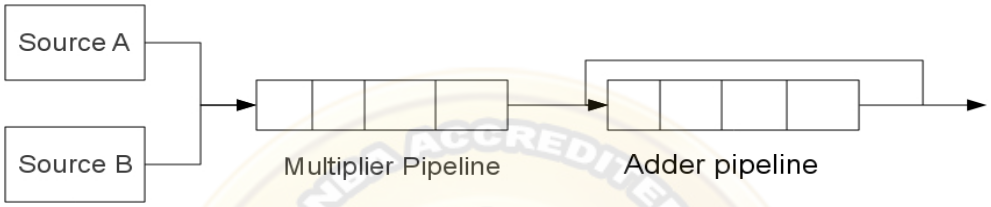
In the above statement, when the system is creating a vector like this the original source values are fetched from the memory into the vector, therefore the data is readily available in the vector. So when a

Operation Code	Base Address SRC 1	Base Address SRC 2	Base Address DST	Vector length

operation is initiated on the data, naturally the operation will be performed directly on the data and will not wait for the fetch cycle. So the total no of CPU Cycles taken by the above instruction is only 100.

Instruction format of Vector Instruction

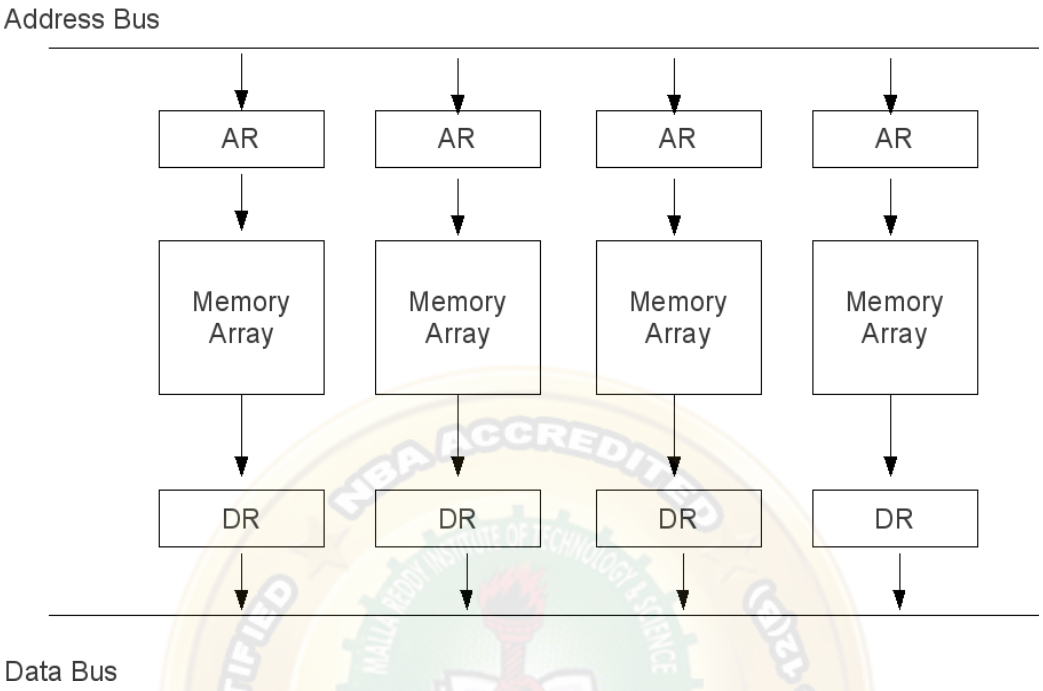
Below we can see the implementation of the vector processing concept on the following matrix



multiplication. In the matrix multiplication, we will be multiplying the row of A matrix with the column of the B matrix elements individually finally we will be adding the results.

In the above diagram we can see that how the values of A vector and B Vector which represents the matrix are being multiplied. Here we will be considering a 4x4 matrix A and B. Now the from the source A vector we will be taking the first 4 values and will be sending to the multiplier pipeline along with the 4 values from the vector B. The resultant 1 value is stored in the adder pipeline. Like wise remaining values from a row and column multiplication will be brought into the adder pipeline, which will be performing the addition of all the things finally we will have the result of one row to column multiplication. When addition operation is taking place in the adder pipeline the next set of values will be brought into the multiplier pipeline, so that all the operations can be performed simultaneously using the parallel processing concepts by the implementation of pipeline.

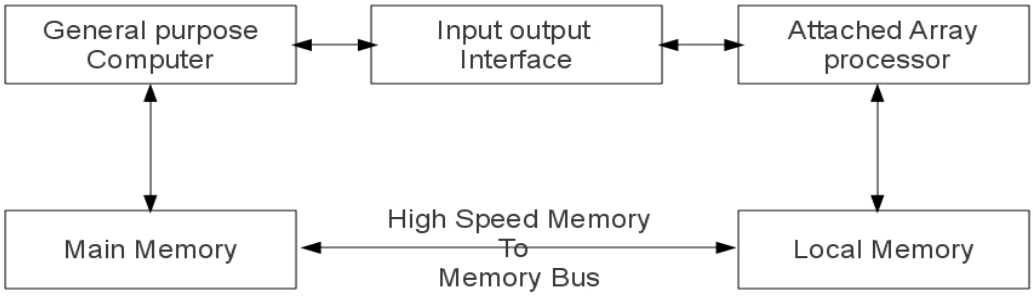
Memory Interleaving



Memory Interleaving:

Pipelining and vector processing naturally requires the several data elements for processing. So instead of using the same memory and selecting one at a time, we will be using several modules of the memory such that we can have separate data for each processing unit. As we can see in the above in the diagram each memory array is designed independently of the next memory array. Such that when the data needed for a operation is stored in the first memory array, another data for another operation can be safely stored in the next memory array, so that the operations can be performed concurrently. This process is called as memory interleaving.

Array Processors: In a distributed computing we will be having several computers working on the same task such that their processing power will be shared among all the systems so that they can perform the task fast. But the disadvantage of the distributed computing is that we have to give separate resources for each system and every system need to be controlled by a task initiating system or can be

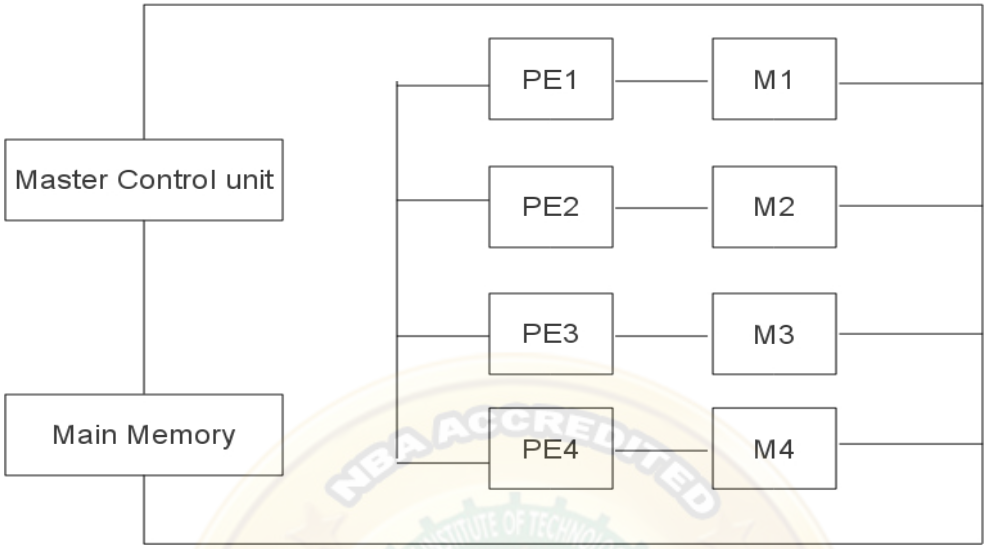


called as a central control unit. The management of this kind of systems is very hard. In order to perform a specific operation involving a large processing there is no need of distributed computing. The alternate for this kind of scenarios is array processors or attached array processors. The simplest is the SIMD Attached array processor.

Attached Array processor

The above diagram shows that the system is attached a separate processor which will be used for operation specific purpose. If the array processor is designed for solving floating point arithmetic, then it will only perform that operations. The detailed figure of the attached array processor is given in the diagram below. This will be having the SIMD architecture. In this we will be having a master control unit which will be coordinating all the process in the array processor. Each processing unit in the array processor is having a local memory unit as in the memory interleaving concept on which it performs the operations. Finally we will be having a main memory in which the original source data and the results that are obtained from the array processor will be stored. This

the working principle of the SIMD array processor technology.



SIMD Array Processor Technology

MULTIPROCESSORS

Multiprocessor:

- A set of processors connected by a communications network

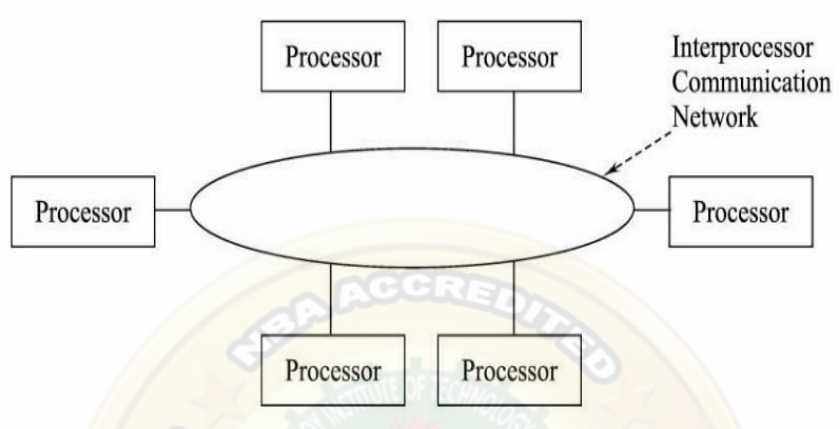


Fig. 5.1 Basic multiprocessor architecure

- A multiprocessor system is an interconnection of two or more CPU's with memory and input-output equipment.
- Multiprocessors system are classified as multiple instruction stream, multiple data stream systems(MIMD).
- There exists a distinction between multiprocessor and multicomputers that though both support concurrent operations.
- In multicomputers 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 cooperate in the solution of the problem.
- VLSI circuit technology has reduced the cost of the computers to such a low Level that the concept of applying multiple processors to meet system performance requirements has become an attractive design possibility.

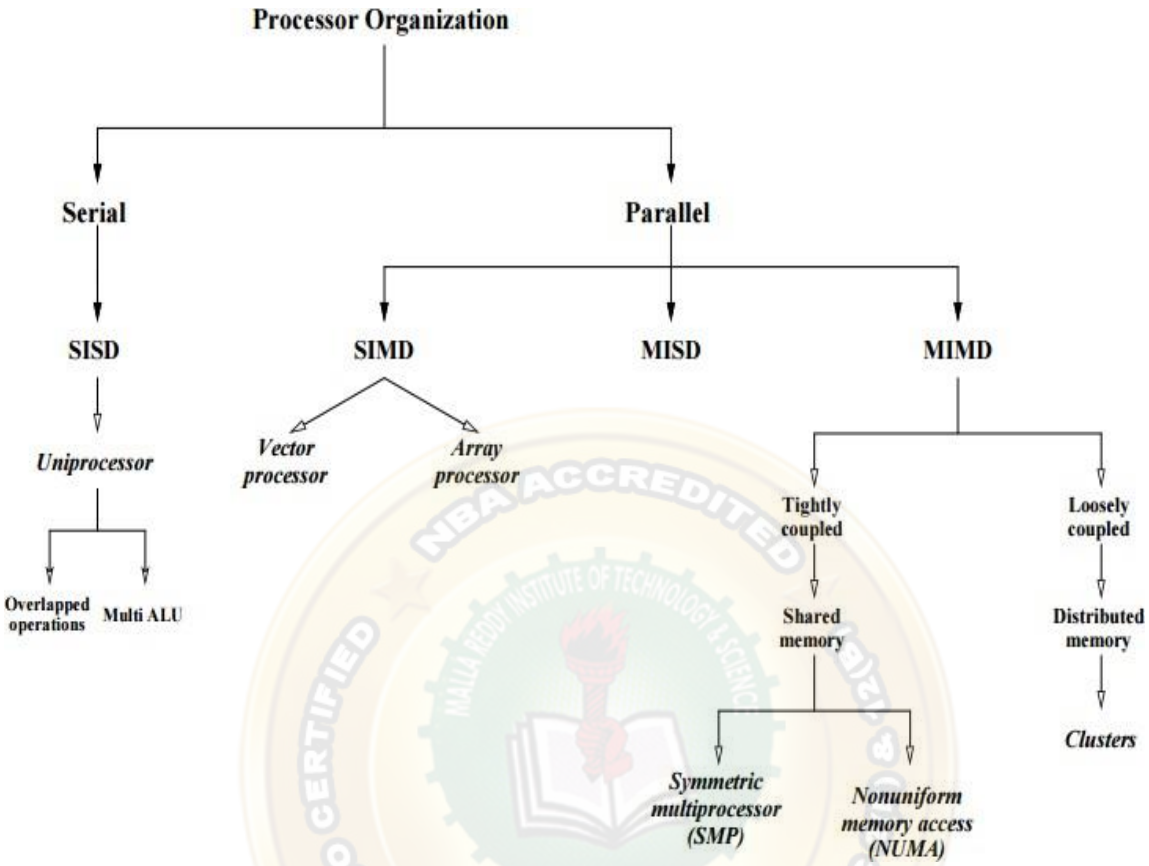


Fig. 5.2 Taxonomy of mono- mulitporcessor organizations

Characteristics of Multiprocessors:

Benefits of Multiprocessing:

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

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

a) Multiple independent jobs can be made to operate in parallel.

b) 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

COUPLING OF PROCESSORS

Tightly Coupled System/Shared Memory:

- Tasks and/or processors communicate in a highly synchronized fashion
- Communicates through a common global shared memory
- Shared memory system. This doesn't preclude each processor from having its own local memory(cache memory)

Loosely Coupled System/Distributed Memory

- Tasks or processors do not communicate in a synchronized fashion.
- Communicates by message passing packets consisting of an address, the data content, and some error detection code.
- Overhead for data exchange is high
- Distributed memory system

Loosely coupled systems are more efficient when the interaction between tasks is minimal, whereas tightly coupled system can tolerate a higher degree of interaction between tasks.

Shared (Global) Memory

- A Global Memory Space accessible by all processors
- Processors may also have some local memory

Distributed (Local, Message-Passing) Memory

- All memory units are associated with processors
- To retrieve information from another processor's memory a message must be sent there

Uniform Memory

- All processors take the same time to reach all memory locations Non-uniform (NUMA) Memory
- Memory access is not uniform

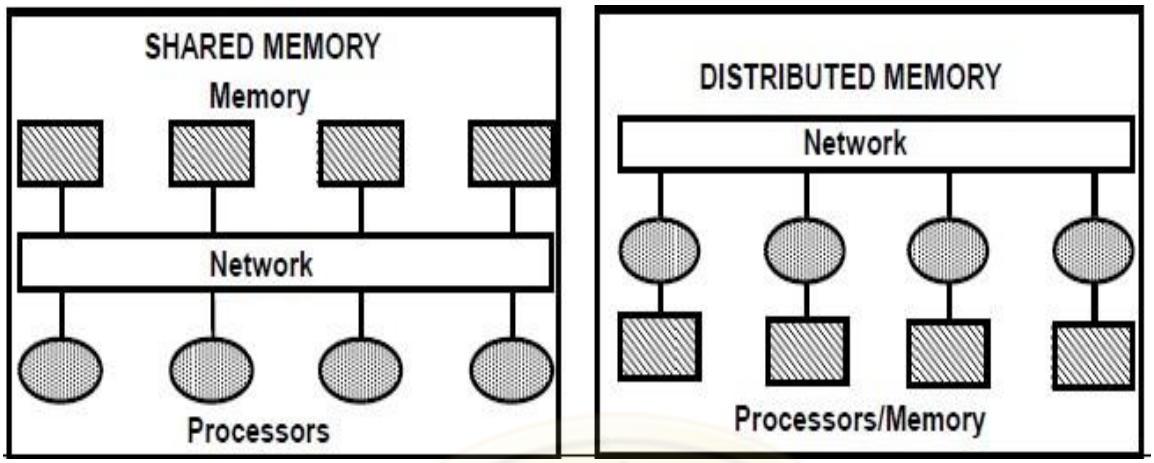


Fig. 5.3 Shared and distributed memory

Shared memory multiprocessor:

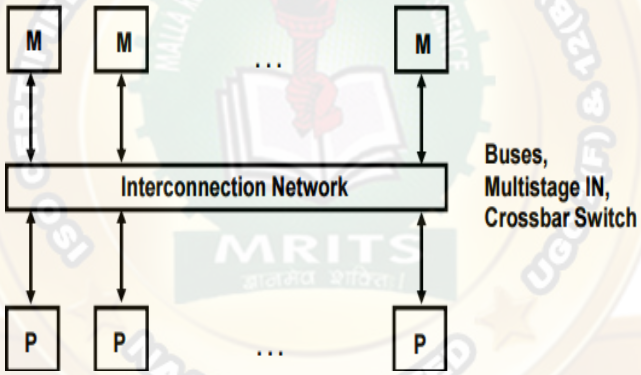


Fig 5.4 Shared memory multiprocessor

Characteristics

- All processors have equally direct access to one large memory address space

Limitations

- Memory access latency; Hot spot problem

Interconnection 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

a. Time shared common Bus

- All processors (and memory) are connected to a common bus or busses
- Memory access is fairly uniform, but not very scalable
- A collection of signal lines that carry module-to-module communication
- Data highways connecting several digital system elements
- Operations of Bus

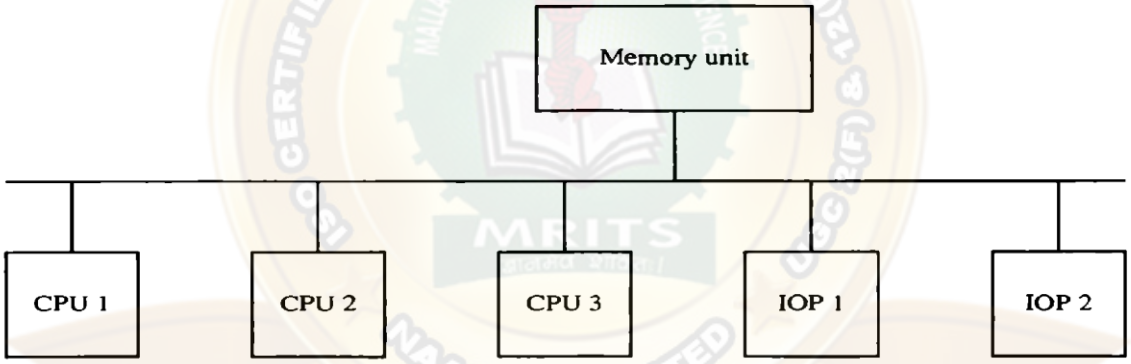


Fig. 5.5 Time shared common bus organization

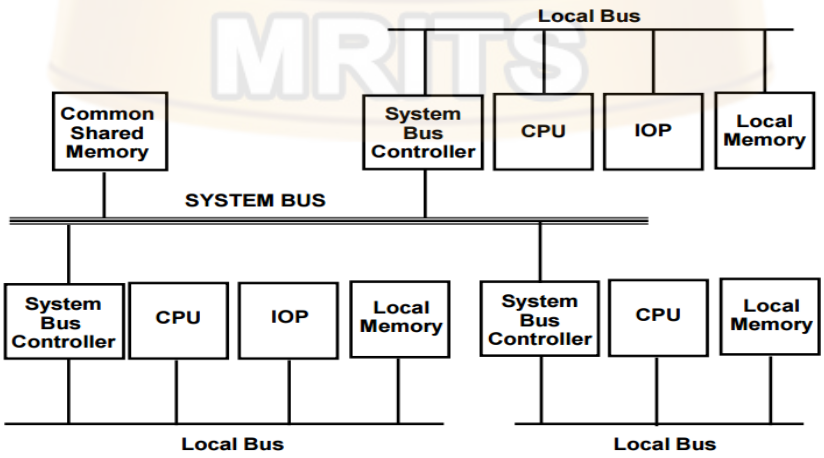


Fig. 5.6 system bus structure for multiprocessor

In the above figure we have number of local buses to its own local memory and to one or more processors. Each local bus may be connected to a CPU, an IOP, or any combinations of processors. A system bus controller links each local bus to a common system bus. The I/O devices connected to the local IOP, as well as the local memory, are available to the local processor. The memory connected to the common system bus is shared by all processors. If an IOP is connected directly to the system bus the I/O devices attached to it may be made available to all processors

Disadvantage.:

- Only one processor can communicate with the memory or another processor at any given time.
- As a consequence, the total overall transfer rate within the system is limited by the speed of the single path

b. Multiport Memory:

Multiport Memory Module

- Each port serves a CPU

Memory Module Control Logic

- Each memory module has control logic
- Resolve memory module conflicts Fixed priority among CPUs

Advantages

- The high transfer rate can be achieved because of the multiple paths.

Disadvantages:

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

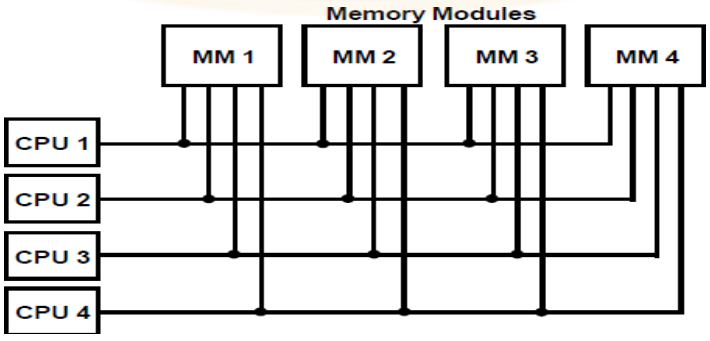


Fig. 5.7 Multiport memory

c. Crossbar switch:

- Each switch point has control logic to set up the transfer path between a processor and a memory.
- It also resolves the multiple requests for access to the same memory on the predetermined priority basis.
- Though this organization supports simultaneous transfers from all memory modules because there is a separate path associated with each Module.
- The H/w required to implement the switch can become quite large and complex

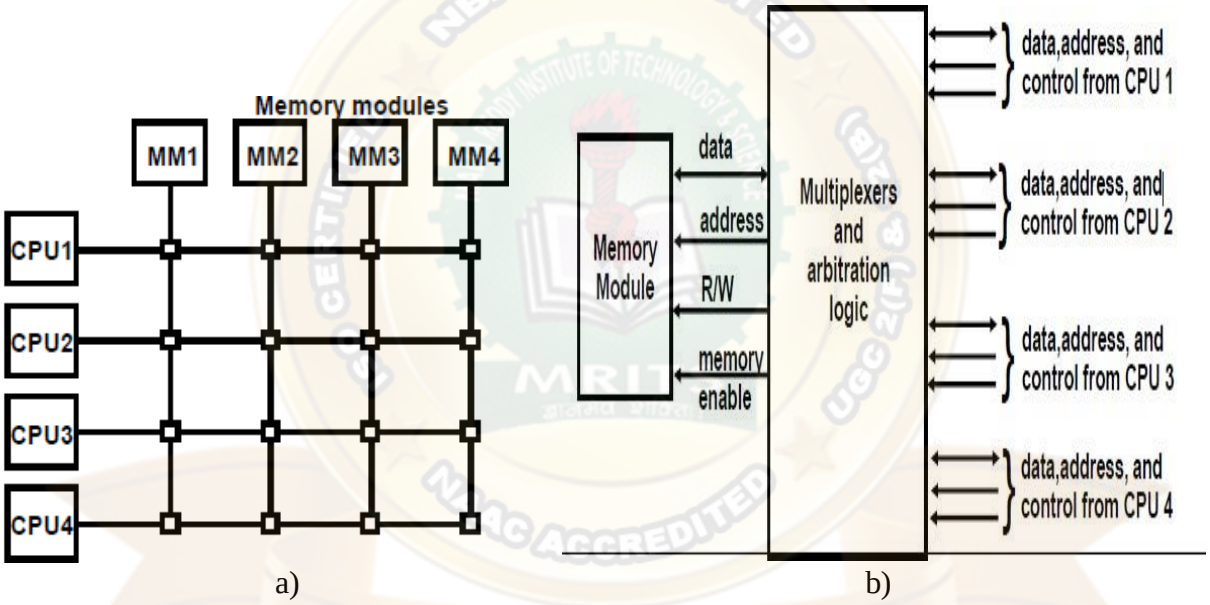


Fig. 5.8 a) cross bar switch b) Block diagram of cross bar switch

Advantage:

- Supports simultaneous transfers from all memory modules

Disadvantage:

- The hardware required to implement the switch can become quite large and complex.

d. Multistage Switching Network:

- The basic component of a multi stage switching network is a two-input, two- output interchange switch.

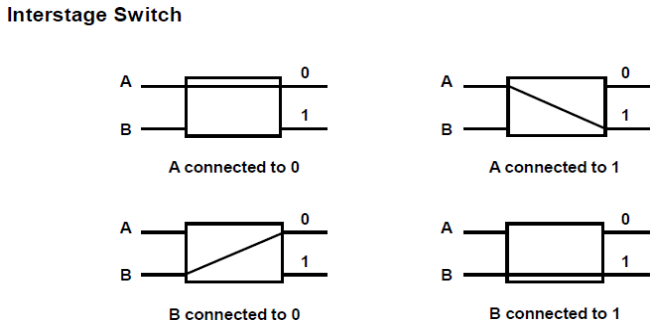


Fig. 5.9 operation of 2X2 interconnection switch

Using the 2x2 switch as a building block, it is possible to build a multistage network to control the communication between a number of sources and destinations.

- To see how this is done, consider the binary tree shown in Fig. below.
- Certain request patterns cannot be satisfied simultaneously. i.e., if

P1 € 000~011, then P_2 € 100~111

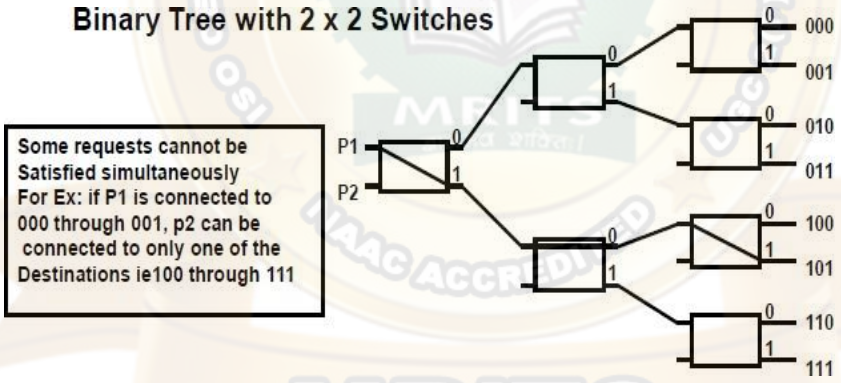


Fig 5.10 Binary tree with 2x2 switches

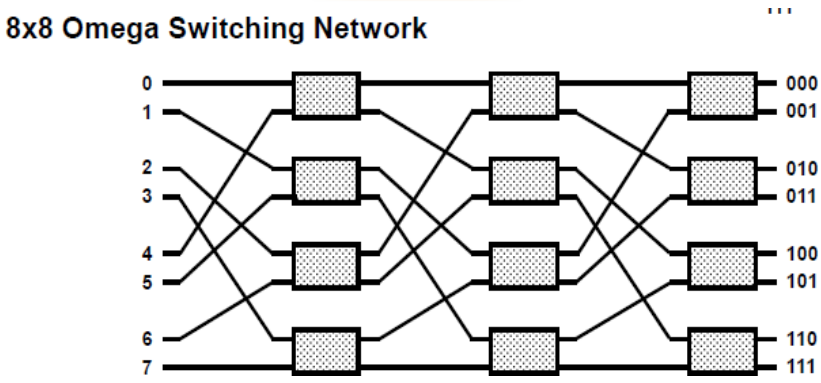


Fig. 5.11 8X8 Omega switching network

- Some request patterns cannot be connected simultaneously. i.e., any two sources cannot be connected simultaneously to destination 000 and 001
- In a tightly coupled multiprocessor system, the source is a processor and the destination is a memory module.
- Set up the path € transfer the address into memory € transfer the data
- In a loosely coupled multiprocessor system, both the source and destination are Processsing elements.

e. Hypercube System:

The hypercube or binary n-cube multiprocessor structure is a loosely coupled system composed of N=2n processors interconnected in an n-dimensional binary cube.

- Each processor forms a node of the cube, in effect it contains not only a CPU but also local memory and I/O interface.
- Each processor address differs from that of each of its n neighbors by exactly one bit position.
- Fig. below shows the hypercube structure for n=1, 2, and 3.
- Routing messages through an *n*-cube structure may take from one to *n* links from a source node to a destination node.
- A routing procedure can be developed by computing the exclusive-OR of the source node address with the destination node address.
- The message is then sent along any one of the axes that the resulting binary value will have 1 bits corresponding to the axes on which the two nodes differ.
- A representative of the hypercube architecture is the Intel iPSC computer complex.
- It consists of 128(*n*=7) microcomputers, each node consists of a CPU, a floating point processor, local memory, and serial communication interface units

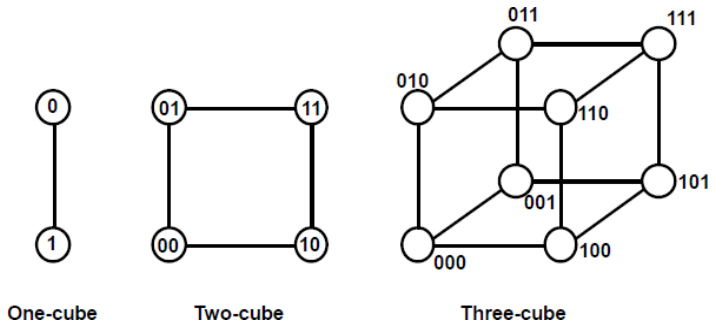


Fig. 5.12 Hypercube structures for n=1,2,3

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 bus that connects the major components such 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.
- Synchronous Bus
    - Each data item is transferred over a time slice
    - known to both source and destination unit
    - Common clock source or separate clock and synchronization signal is transmitted periodically to synchronize the clocks in the system
- Asynchronous Bus
    - Each data item is transferred by Handshake mechanism
        - Unit that transmits the data transmits a control signal that indicates the presence of data
        - Unit that receiving the data responds with another control signal to acknowledge the receipt of the data

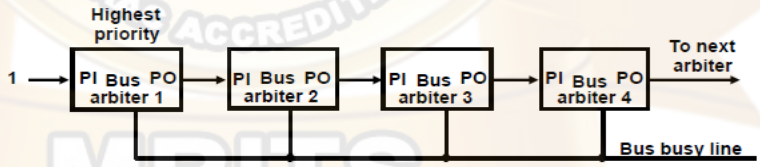
- Strobe pulse -supplied by one of the units to indicate to the other unit when the data transfer has to occur

Table 5.1 IEEE standard 796 multibus signals

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

INTERPROCESSOR ARBITRATION STATIC ARBITRATION

Serial Arbitration Procedure



Parallel Arbitration Procedure

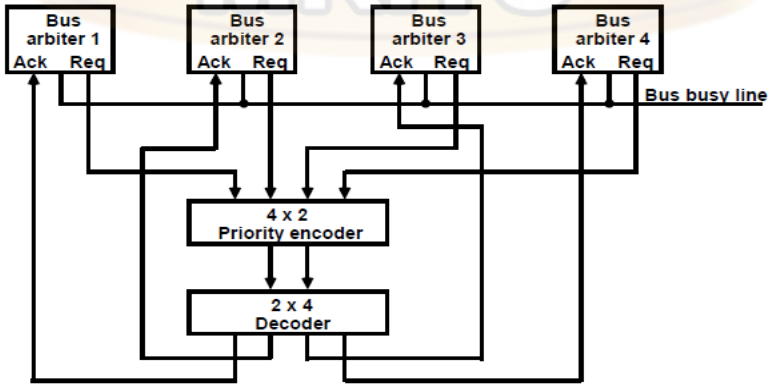


Fig. 5.13 Inter-processor arbitration static arbitration

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 fashion
- 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.
- LRU
    - The least recently used algorithm gives the highest priority to the requesting device that has not used bus for the longest interval.
- FIFO
    - The first come first serve scheme requests are served in the order received. The bus controller here maintains a queue data structure.
- Rotating Daisy Chain
    - Conventional Daisy Chain -Highest priority to the nearest unit to the bus controller
    - Rotating Daisy Chain –The PO output of the last device is connected to the PI of the first one. Highest priority to the unit that is nearest to the unit that has most recently accessed the bus(it becomes the bus controller)

Inter processor communication and synchronization:

- The various processors in a multiprocessor system must be provided with a facility for *communicating* with each other.
    - A communication path can be established through *a portion of memory* or *a common input-output channels*.

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

Loosely Coupled System

- There is *no shared memory* for passing information.
- The communication between processors is by means of message passing through *I/O channels*.
- The communication is initiated by one processor calling a *procedure* that resides in the memory of the processor with which it wishes to communicate.

- The communication efficiency of the interprocessor network depends on the *communication routing protocol, processor speed, data link speed, and the topology of the network.*

Interprocess Synchronization

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

Mutual Exclusion with Semaphore

- A properly functioning multiprocessor system must provide a mechanism that will guarantee orderly access to shared memory and other shared resources.
    - Mutual exclusion: This is necessary to protect data from being changed simultaneously by two or more processors.
    - Critical section: is a program sequence that must complete execution before another processor accesses the same shared resource.
- A *binary variable* called a *semaphore* is often used to indicate whether or not a processor is executing a critical section.

- Testing and setting the semaphore is itself a critical operation and must be performed as a single indivisible operation.
- A semaphore can be initialized by means of a *test and set instruction* in conjunction with a hardware *lock* mechanism.
- The instruction TSL SEM will be executed in two memory cycles (the first to read and the second to write) as follows:

R □ M[*SEM*], M[*SEM*] □ 1

Cache Coherence

cache coherence is the consistency of shared resource data that ends up stored in multiple local **caches**. When clients in a system maintain **caches** of a common memory resource, problems may arise with inconsistent data, which is particularly the case with CPUs in a multiprocessing system.

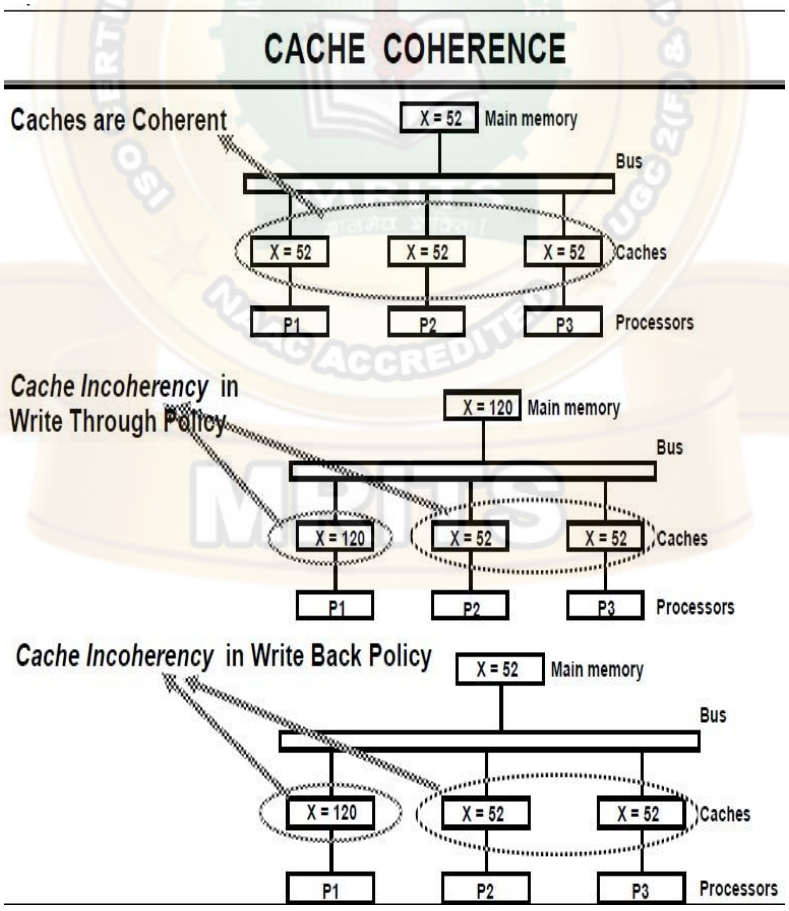


Fig. 5.14 cache coherence

Shared Cache

-Disallow private cache

-Access time delay

Software Approaches

* Read-Only Data are Cacheable

- Private Cache is for Read-Only data
- Shared Writable Data are not cacheable
- Compiler tags data as cacheable and noncacheable
- Degrade performance due to software overhead

* Centralized Global Table

- Status of each memory block is maintained in CGT: RO(Read-Only); RW(Read and Write)
- All caches can have copies of RO blocks
- Only one cache can have a copy of RW block
- Hardware Approaches

* Snoopy Cache Controller

- Cache Controllers monitor all the bus requests from CPUs and IOPs
- All caches attached to the bus monitor the write operations
- When a word in a cache is written, memory is also updated (write through)
- Local snoopy controllers in all other caches check their memory to determine if they have a copy of that word; If they have, that location is marked invalid(future reference to this location causes cache miss)