# LECTURE MATERIAL

**Name of the subject : Computer Organization and Architecture**
**Name of the Course Coordinator : Mrs. J. Sravanthi**
**&Faculty Member :Dr. M. Sandhya Rani**
**Academic Year : 2026-27**
**Year & SEM : II - I SEM**

**DEPARTMENT OF CSE (Data Science)**

TKR College of Engineering & Technology, Meerpet,
Balapur, Saroornagar, Hyderabad - 500097, Telangana.
Accredited by NAAC with A+

# UNIT-1

## Boolean algebra and Logic Gates

## Digital logic gates, Data Representation

## Digital Computers

## Data Representation:

## Data Types:

### Number System :-

Number System defines a set of values used to represent the quantity.

* Let us consider a quantity that is represented by a decimal number $(7392)_{10}$. We can also represent the same quantity with any other number system like binary, octal, Hexa decimal.
* Decimal number System is most popular and common Number System, used in our daily-life. If we have to count any thing we use decimal number system from 0 to 9, such as 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
* In case of binary Number System, two different digits are there. And 2 is the base for the binary System.
* Base of the Number System is also called as radix which is denoted by 'r'. The base will tell us how many distinguish numbers are in Number System.

| Name | Base(r) | digits |
|---|---|---|
| Binary | 2 | 0, 1 |
| Octal | 8 | 0, 1, 2, 3, 4, 5, 6, 7 (0 to (r-1)) |
| Decimal | 10 | 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 |
| Hexadecimal | 16 | 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F |

* Digits of a number system start from '0', incremented by '1' until the value (r-1).

## Boolean Algebra and Logic Gates:

**Boolean Algebra:** It is the mathematical foundation of digital Electronics, enabling the design of logic gates, binary codes, storage Elements and registers.

→ It is a branch of Algebra.

1) dealing with binary variables (0, 1) & logical operations

2) The values of the variables are the truth values which are denoted by 0 & 1 [0 → False, 1 → True]

→ It was introduced by George Boole in 1847 for Mathematical Analysis of logic.

→ It is the fundamental in the development of digital electronic & is provided for all in programming languages.

## Rules of Boolean Algebra: —

1) In Boolean Algebra, variables are two values 0 & 1

→ Binary 1 is for denoted to "High" & the complement of '1' is '0'

→ Binary 0 is for "low" & the complement of '0' is '1'

→ In Boolean Algebra, there are two logical operations are used in 1) Logical 'OR' operation

2) Logical 'AND' operation.

→ Logical 'OR' operation is represented by '+' (addition) b/w two variables. Ex: A+B

→ logical 'AND' operation is represented by '.' (multiplication) b/w two variables. Ex: A.B

## Basic Boolean Operations:-

The basic boolean operations are AND, OR, NOT.

AND → Multiplication

OR → Addition

NOT → Complement

→ For 'n' variables → There are $2^n$ combinations.

| Two variables A | B | (operation) A+B | A.B (mul) |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 |

Combinations →

| Three variables A | B | C | A+B | A.B |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

NOT → Complement

A → $\bar{A}$

0 → 1

1 → 0

| A | $\bar{A}$ |
|---|---|
| 0 | 1 |
| 1 | 0 |

## Basic laws:—

The basic laws of boolean algebra is applicable for both logic operations such as addition & multiplication.

1) Identity laws

$A+0=A$ , $A\cdot 0=0$

$A+1=1$ , $A\cdot 1=A$

2) Idempotent law

$A+A=A$

$A\cdot A=A$

3) Complementation law

$A+\overline{A}=1$

$A\cdot\overline{A}=0$

4) Commutative law

$A+B=B+A$

$A\cdot B=B\cdot A$

5) Associative : law

$(A+B)+C=A+(B+C)$

$(A\cdot B)\cdot C=A\cdot(B\cdot C)$

6) Distributive law

$A+(B\cdot C)=(A+B)\cdot(A+C)$

$A\cdot(B+C)=(A\cdot B)+(A\cdot C)$

7) Absorption law

$X\cdot(X+Y)=X$

$X+(X\cdot Y)=X$

8) Demorgan's law

$\overline{A+B}=\overline{A}\cdot\overline{B}$

$\overline{A\cdot B}=\overline{A}+\overline{B}$

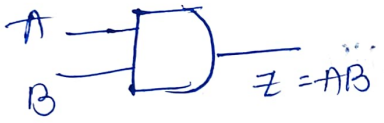
**Logic Gates:** Logic gates are the fundamental building blocks of digital System.

- Logic Gates
  - Basic Gates
    - AND
    - OR
    - NOT
  - Universal Gates
    - NAND
    - NOR
  - Other Gates
    - EX-OR
    - EX-NOR

**AND Gate:-**

* 'AND' gate has two or more inputs but only one o/p
* The output assumes 'Logic 1' state when two i/ps are at 'logic 1' state.

Symbol:

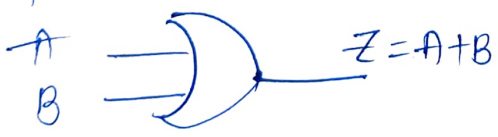



Truth Table

| A | B | Z |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

**OR Gate:-**

* Have two or more inputs and only one output
* Output is '1' when any one of the i/p is '1'




Symbol

| A | B | Z |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Truth Table

## NOT Gate:-

* Also called as 'inverter'
* Has only one i/p & one o/p
* When i/p is '0' & o/p is '1', when i/p is '1' & o/p is '0'

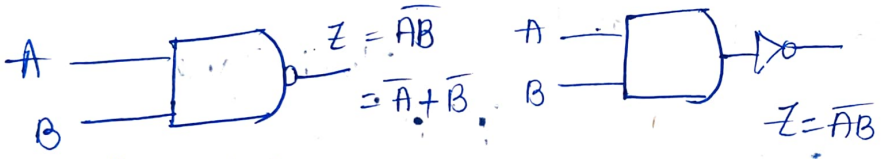



| A | Z = Ā |
|---|---|
| 0 | 1 |
| 1 | 0 |

## Universal Gates :-

* By using this universal gates, we can implement all the gates (AND, OR..)

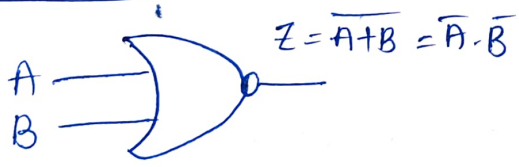
## NAND Gate :

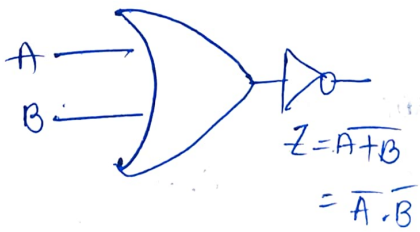



'NAND' means NOT + AND

| i/Ps A | i/Ps B | o/P Z |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

## NOR Gate :-



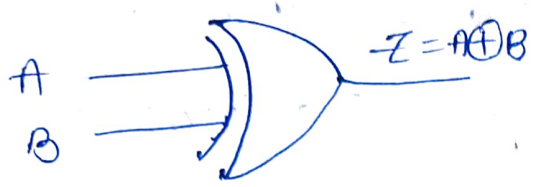




| A | B | Z |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

• NOR Gate = OR Gate + NOT Gate

## Other Gates (Special Gates):-

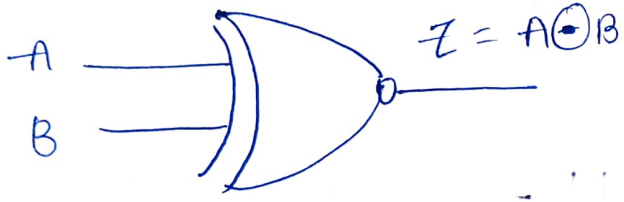
### 1. Ex-OR Gate:




| A | B | Z |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

* When any one of the input is logic '1' then the O/P is logic '1'.

### 2. Ex-NOR Gate:-




| A | B | Z |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

* When two i/Ps are same that means either logic '0' or logic '1' then the O/P is logic '1'.

## Binary Codes:

→ Binary codes are methods of representing information using binary digits (bits: 0 and 1)

→ It is essential for communication b/w hardware and software, data storage and Error detection.

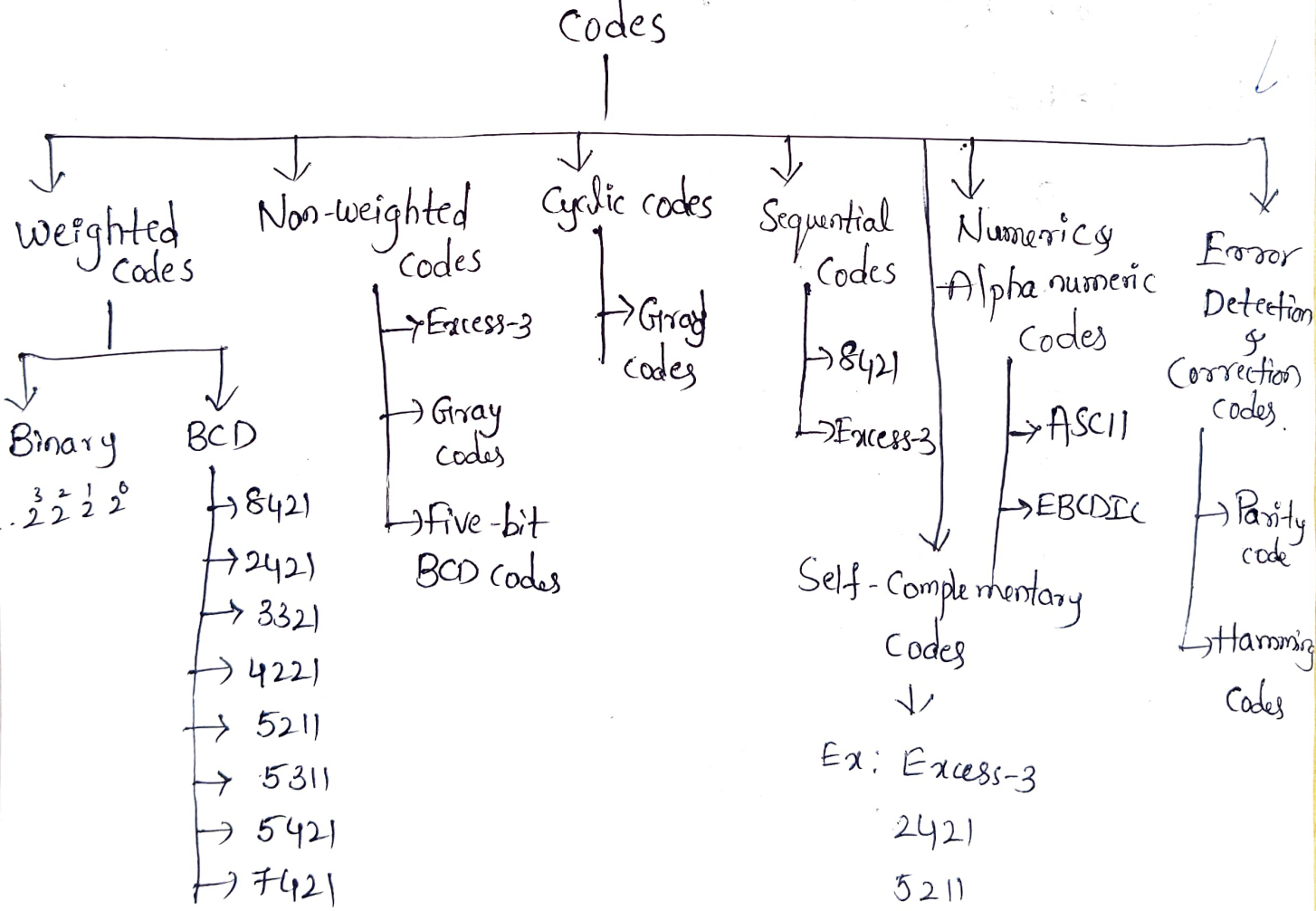
## Binary codes :—

→ Binary codes are methods of representing information using binary digits (bits: 0 and 1)

→ Binary codes are used to represent text, numbers, images or other types of information in the form of '0' or '1'.

→ It is essential for communication b/w hardware and software, data storage and error detection etc.

## Classification of Binary Codes :—



## Weighted codes:-

→ In weighted codes, each digit position of the number represents a specific weight.

Ex:- Decimal No: 9,8

| 8421 representation | 2421 representation |
|---|---|
| 1001 | 1111 |
| $(8\times1)+(4\times0)+(2\times0)+(1\times1)$ | $(2\times1)+(4\times1)+(2\times1)+(1\times1)$ |
| $8+0+0+1$ | $2+4+2+1$ |
| $=9$ | $=9$ |

## Non-weighted codes:-

→ Non-weighted codes are not assigned with any weight of each digit position.

Ex:-

| Decimal | Excess-3 | Gray code | 5-bit BCD code |
|---|---|---|---|
| 0 | 0011 | 0000 | 0,0000 |
| 1 | 0100 | 0001 | 0,0001 |
| 2 | 0101 | 0011 | 0,0010 |
| 3 | 0110 | 0010 | 0,0011 |
| 4 | 0111 | 0110 | 0,0100 |
| 5 | 1000 | 0111 | 0,0101 |
| 6 | 1001 | 0101 | 0,0110 |
| 7 | 1010 | 0100 | 0,0111 |
| 8 | 1011 | 1100 | 0,1000 |
| 9 | 1100 | | 0,1001 |

2) Cyclic codes :-

Cyclic codes are those in which each successive code word differs from the preceding, only one bit position.

Ex:- Gray code.

3) Sequential codes :-

→ In sequential codes, each succeeding code is one binary number greater than its preceeding code.

Ex:- 8421 & Excess-3 codes.

4) Self-complementary codes :-

→ Self complementary code is a binary-coded decimal (BCD) code, in which the 9's complement of a decimal digit can be obtained by simply complementing each bit (changing 0 to 1 and 1 to 0) of its code word.

Ex: Excess-3.

Decimal digit 2

→ Excess-3 code for 2 is : 2+3 = 5.

Binary conversion for '5' is 0101

Complementing each bit ie 1010 → in decimal it is '7'

9's complement of '2' is

9 - 2 = '7'

## 5) Numeric & Alphanumeric codes :

**Numeric code:** It is code that uses only numbers to represent specific information.

→ Numeric codes can only show digits from '0' to '9'.

Ex:- 8421, 2421, 5211

**Alphanumeric codes:-**

The codes which consists of both numbers & alphabetic characters are called alphanumeric codes.

Ex:- ASCII, EBCDIC

**ASCII :-** (American Standard Code for Information Interchange). It translates human-readable text into binary numbers that computer understand. It assigns a unique numeric value from '0' to '127'

Numbers 0 to 9 : 48 to 57 (0110000 to 0111001)

Uppercase A to Z : 65 to 90 (1000001 to 1011010)

lowercase a to z : 97 to 122 (1100001 to 1111010)

**EBCDIC :-** (Extended Binary Coded Decimal Interchange code). It is an alphanumeric code as 8-bit binary coded digital systems.

→ Total $2^8 = 256$ Possible characters.

→ This was developed by IBM (International Business Machine) & mainly used in IBM's mainframe Computer Systems. 0-9 numbers : 240 to 249 ; A-I : 193-201 etc

## Error Detection & Correction codes :—

* When the digital information in the binary form is transmitted from one circuit (or) System to another circuit (or) System an Error may occur. This means that the signal corresponding to '0' may change to '1' (or) Viceversa due to the presence of noise.
* Codes which allow only Error detection are called "Error detecting codes" & Codes which allow Error detection & correction are called "Error detection & Correction Codes".
* Ex: Parity code, Hamming code.
* By using Parity bit, we can detect Error in the message.
* Hamming code is a type of linear Error-correcting code capable of detecting up to two-bit Errors and automatically correcting single-bit Errors in digital data transmission.

## Binary Storage & Registers :—

* Binary Storage is the method by which computer store data by using binary digits (bits)
* Bit is the Smallest unit of data & can have only two values ie '0' or '1'.

Bit → 1 bit → 0 or 1

Nibble → 4 bits

Byte → 8 bits

Word → A group of bits processed by the CPU (Commonly 16, 32 (or) 64 bits).

## Registers :-

* A register is a group of flip-flops, with each flip-flop capable of storing one bit of information.
* An n-bit register has a group of n-flip-flops & is capable of storing any binary information of n-bits.
* Registers are small, high-speed storage locations inside the CPU.
* They temporarily hold data, instructions (or) memory address while the processor executes programs.

## Types of Registers :-

(CPU)

Program Counter → Stores the address of the next instruction

Instruction Register (IR) → Holds the current instruction being executed

Memory address Register (MAR) → Stores the address of the memory location to access

Memory Data / Buffer Register (MDR) → Stores data being transferred to / b/w Processor & Main memory.

Accumulator (AC) → Holds intermediate arithmetic & logical results.

General Purpose Registers (GPRs) → Store operands and temporary data

Stack Pointer (SP) → Points to the top of the stack

Flag Register → Stores condition flags such as Zero, carry, Sign and overflow.

## Data Representation :-

A computer understands only binary digits i.e 0 & 1. All types of data such as numbers, characters, Symbols, images, audio & video must be converted into binary form before they can be stored (or) processed.

* Data representation is the method of converting data into binary form so that it can be stored, processed and transmitted by a digital computer.

## Data types :-

### Decimal number System :-

0, 1, 2, 3, 4, 5, 6, 7, 8 & 9

Ex. 724.5 is represented as

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

## Binary number System:-

The binary number System uses the radix 2. The 2 Symbols are '0' & '1'

$$(45)_{10} = 1\times 2^5 + 0\times 2^4 + 1\times 2^3 + 1\times 2^2 + 0\times 2^1 + 1\times 2^0$$

The string of digits 101101 is interpreted to represent the quantity as above.

## Octal number System:-

The octal number System uses radix 8. The 8 Symbols are 0, 1, 2, 3, 4, 5, 6 & 7

Ex:- Octal 736.4 is converted to decimal as

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

## Hexadecimal number System:-

The Hexadecimal number System uses the radix 16. The 16 Symbols are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E & F. The Symbols A, B, C, D, E, F represents decimal numbers 10, 11, 12, 13, 14 & 15 respectively.

Ex:- The hexa decimal number F3 is converted to decimal as

$$(F3)_{16} = F\times 16^1 + 3\times 16^0 = 15\times 16 + 3\times 1 = (243)_{10}$$

## Conversion :-

Conversion from decimal to its equivalent represented in the radix 'r' system is carried out by separating the number into its integer & fraction parts & converting each part separately. The conversion of a decimal integer into a base r representation is done by successive divisions by r & accumulation of the remainders. The conversion of decimal fraction to radix r representation is accomplished by successive multiplications by r. finally two parts are combined to give the total conversion required.

* Conversion is needed because humans, hlw & slw speak diff. languages & need to understand each other.

Ex :- Conversion of decimal 41.6875 into binary

Integer = 41,

| | |
|---|---|
| 2 \| 41 | |
| 2 \| 20 - 1 | |
| 2 \| 10 - 0 | |
| 2 \| 5 - 0 | |
| 2 \| 2 - 1 | |
| 1 - 0 | |

$(41)_{10} - (101001)_2$

Fraction = 0.6875

$0.6875 \times 2 = 1.3750 = 1$

$0.3750 \times 2 = 0.7500 = 0$

$0.75 \times 2 = 1.5 = 1$

$0.5 \times 2 = 1 = 1$

$(0.6875)_{10} = (0.1011)_2$

$\therefore (41.6875)_{10} = (101001.1011)$

* The conversion from & to binary, octal & hexadecimal representation plays an important role in digital computers. Since $2^3 = 8$ & $2^4 = 16$, each octal digit corresponds three binary digits & each hexadecimal digit corresponds to four binary digits.

Ex:- 1 2 7 5 4 3 → octal

1010 1111 0110 0011 → Binary

A F 6 3 → Hexadecimal

**Complements :-** Complements are needed to easily perform Subtraction

Complements are used in digital computers for simplifying the Subtraction operation and for logical manipulation. There are two types of complements for each base 'r' system. The r's complement and (r-1)'s complement. For binary numbers, there are 2's complement & 1's complement. For decimal numbers, there are 10's complement and 9's complement.

**(r-1)'s Complement :-**

* For a given number 'N' in base 'r' having n digits, the (r-1)'s complement of N is defined as $(r^n - 1) - N$.
* For decimal numbers, $r = 10$ & $(r-1) = 9$,

So the 9's complement of 'N' is $(10^n - 1) - N$

For Ex:- if $n = 4$ then $10^4 - 1 = 10000 - 1 = 9999$,

it follows that the 9's complement of a decimal number is obtained by subtracting each digit from 9.

Ex: 9's Complement of 546700 is Subtracting each digit from '9' i.e 999999 - 546700 = 453299.

for binary numbers, $r=2$, & $(r-1)=1$, So, the 1's complement of N is $(2^n-1)-N$.

* Again $2^n$ is the representation by a binary number that consists of a 1 followed by 'n' number of 0's.

Ex:- if $n=4$ ; $2^4 = (10000)_2$ then $2^4-1 = (1111)_2$.

Thus, 1's complement is obtained by Subtracting Each digit from 1.

However, the Subtraction of a binary digit from 1 causes the bit to change from 0 to 1 (or) from 1 to 0. Therefore 1's complement of a binary number is formed by changing 0's to 1's & 1's into 0's.

for Ex:- 1's complement of 1011001 is 0100110 & 1's complement of 111100010110 is 000011101001.

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

## r's Complement:-

The r's complement of an n-digit number N in base r is defined as $r^n-N$ for $n \neq 0$ & 0 for $N=0$.

* Comparing $(r-1)$'s Complement, the $r$'s Complement is obtained by adding 1 to the $(r-1)$'s Complement Since, $r^n - N = [(r^n - 1) - N] + 1$.

* Thus 10's Complement of the decimal 2839 is 7160 + 1 = 7161 & is obtained by adding '1' to the 9's complement. The 2's Complement of binary 101100 is 010011 + 1 = 010100 & is obtained by adding '1' to the 1's Complement value.

* The Complement of the Complement restores the number to its original value. The $r$'s complement of the number 'N' is $r^n - N$. The complement of the $r$'Complement is $r^n - (r^n - N) = r^n - r^n + N = N$.
i.e Giving back the original number 'N'.

## Subtraction of Unsigned Numbers :—

* The Subtraction of two n-digit unsigned numbers $M - N$ $(N \neq 0)$ in base 'r' can be done as follows,
* Add the minued M to the r's Complement of the Subtrahend N. This performs $M + (r^n - N) = M - N + r^n$
* If $M \geq N$, the Sum will produce an end Carry $r^n$ which is discarded, and what is left is the result $M - N$.

* If $M<N$, the Sum does not produce an end carry and is Equal to $r^n - (N-M)$, which is the r's Complement of $(N-m)$. To obtain the answer in a familiar form, take the r's complement of the Sum and place a negative sign in front.

Ex!. Subtract $72532 - 13250 = 59282$ $(M>N)$

The 10's Complement of 13250 is 86750

$M = 72532$

10's Complement of $N = 86750$

Sum $= 159282$

Discard end carry $10^5 = 100000$ from Sum 159282

i.e. $159282 - 100000$

Answer $= 59282$

Ex!. Subtract $13250 - 72532 = -59282$ $(M<N)$

$M = 13250$

10's Complement of $N = 27468$

Sum $= 40718$

There is no End Carry

Answer is 10's complement of 40718 with negative sign.

$\therefore$ Aswer $= -59282$

## Subtraction for binary numbers:—

Ex:- $X = 1010100$ & $Y = 1000011$. Perform $X-Y$ & $Y-X$

(i) $X-Y$

$X = 1010100$

2's complement of $Y$: $0111101 + 1 = 0111101 + 1$

Sum $= 10010001$

Discard end carry

$\therefore$ Answer $X-Y = 0010001$

(ii) $Y-X$

$Y = 1000011$

2's complement of $X = 0101100$

Sum $1101111$

No carry $\therefore$ Answer is 2' complement of Sum

$\therefore Y-X = -0010001$

## Fixed - Point Representation:—

* Positive integers, including zero, can be represented as unsigned numbers.
* We need a notation for negative integers.
* In general, positive numbers are represented by '+' sign & negative numbers with '−' sign.
* Because of computer of hardware limitations, computer must represent everything with 1's & 0's including sign of a number.
* To represent the sign, a bit is placed in the left most position of the number.
* If the sign bit is equal to '0' then positive & '1' then negative number.

* In addition to 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 two ways of specifying the position of the binary point in a register:
  1) by giving it a fixed position.
  2) by employing a floating point representation.
* The fixed-point method assumes that the binary point is always fixed in one position. The two fixed positions most widely used are
  1). a binary point in the extreme left of the register to make the stored number a fraction.
  2) a binary point in the extreme right of the register to make the stored number an integer.

## Integer Representation:-

* When an integer binary number is positive, the sign is represented by '0' & magnitude by a positive binary number. When the number is negative, the sign is represented by '1' but the rest of the number may be represented in one of three possible ways:
  1) Signed - magnitude representation
  2) Signed - 1's complement representation
  3) Signed - 2's complement representation

* The Signed-magnitude representation of a negative number consists of the magnitude and a negative sign. In the other two representations, the negative number is represented in either the 1's (or) 2's Complement of its positive value.

Ex:- +14 is represented by a sign bit '0' in the left most position followed by binary Equivalent to be stored in an 8-bit register, then

+14 : 0 0001110

* There are three different representations for -14 with 8-bit register.

→ In Signed-magnitude representation : 1 0001110

→ In Signed-1's Complement representation: 1 1110001

→ In Signed-2's Complement representation: 1 1110010

* The Signed-magnitude System is used in ordinary arithmetic but awkward when Employed in Computer arithmetic.

* The 1's Complement imposes difficulties because it has two representations of 0 (+0 & -0). The Signed 2's Complement representation of the negative numbers is used for the Signed binary arithmetic.

## Arithmetic Addition:-

* The addition of two numbers in the Signed-magnitude system follows the rules of ordinary arithmetic.
* If the signs are the same, we add the two magnitudes and give the sum the common sign.
* If the signs are different, we subtract smaller magnitude from the larger & give the result the sign of the larger magnitude.

Eg: $(+25) + (-37) = -(37 - 25) = -12$

* This process requires comparison of the signs and the magnitudes and then performing either addition (or) subtraction.
* In Signed-2's complement, there is no need of subtraction or comparison, only addition & complementation.
* The process of adding two numbers, including their sign bits and discard any carry out of the sign (leftmost) bit position.

Note: Negative numbers must be in 2's complement & if the sum obtain after the addition is negative, it is in 2's complement form.

Ex:

| | | |
|---|---|---|
| +6 | 00000110 | |
| +13 | 00001101 | |
| +19 | 00010011 | |

| | | |
|---|---|---|
| -6 | 11111010 | |
| +13 | 00001101 | |
| discard | 1111 | |
| +7 | ① 00000111 | |

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

* In each of the four cases, the operation performed is always addition including sign bit & any carryout of the sign bit position is discarded & negative results are automatically in 2's complement form
* To get the signed 2's complement of a negative number, take the positive number & apply 2's complement form of it & this is the required answer.

## Arithmetic Subtraction :

* Subtraction of two signed binary numbers when negative numbers are in 2's complement form is very simple and can be stated as follows:

→ Take the 2's complement of the subtrahend (including the sign bit) and add it to the minuend (including the sign bit), and a A carry out of the sign bit position is discarded.

* The subtraction operation can be changed to an addition operation, if the sign of the subtrahend is changed. This is demonstrated by the following relationship.

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

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

Ex:- $(-6)-(-13) = -6+13 = +7$

$-6$ : 11111010 ; $+13$ : 00001101

```
  11111010
+ 00001101
  --------
  111111
 (1)00000111   =>  +7 : 00000111
  --------
  ↓
  1 is discarded
```

Overflow:-

* When two numbers of n-digits each are added & the sum occupies n+1 digits, we say that an overflow occured. A result that contains n+1 bits can not be accommodated in a register with a standard length of 'n' bits.
* In 8-bit, the range of numbers can accommodate is from binary +127 to -128.

Ex:-

```
  +70   0 1000110        -70   1 0111010
  +80   0 1010000        -80   1 0110000
 ----  ----------      ----  ----------
 +150   1 0010110       -150  (1)0 1 1010
                              ↳ discard.
```

* Here, the 8-bit result that should have been positive has a negative signe & 8-bit result that should have been negative has a positive sign. The carryout of the sign bit position is taken as sign bit of the result, the 9-bit answer. So obtained will be correct. Since the answer cannot be accommodated within 8-bits, we say that an overflow occured.

* An overflow condition can be detected by observing the carry into the sign bit position & the carry out of the sign bit position are not equal, an overflow has occured. This can be detected by using XOR gate.

## Floating - Point Representation:

* The floating - point representation of a number has two parts. The first part represents a signed fixed - point number called the mantissa. The second part designates the position of the decimal (or binary) point & is called the Exponent. The fixed - point 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 the actual position of the decimal point is four positions to the right of the indicated decimal point in fraction. This represen-tation is equivalent to the scientific notation

$+0.6132789 \times 10^{+4}$

* floating - point is always interpreted as follows:

$$m \times r^{e}$$

$m \rightarrow$ mantissa ; e - Exponent ; r - radix-point.

* A floating-point binary number is represented in a similar manner except that it uses base 2 for exponent.

Ex:- The binary number +1001.11 is represented with an 8-bit fraction & 6-bit exponent as follows.

| Fraction | Exponent |
|---|---|
| 01001110 | 000100 |

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

* The floating-point number is equivalent to

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

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

Ex:- The decimal number 350 is normalized but 000350 is not. Regardless of where the position of the radix-point is assumed to be in the mantissa, the number is normalized only if its leftmost digit is non-zero. For example, the 8-bit binary number 00011010 is not normalized because of the three leading 0's. The number can be normalized by shifting three positions to the left & discarding the leading 0's to obtain 11010000.

* The three shifts multiply the number by $2^3 = 8$. To keep the same value for the floating-point number, the exponent must be substracted by 3. Normalized numbers provide the maximum possible precision for the floating point numbers. A zero cannot be normalized because it doesnot have a nonzero digit. It usually represented in floating point by all 0's in the mantissa and exponent.

## Digital Computers :—

### Introduction :—

* It is a digital system that performs various computational tasks. The word digital implies that the information in the computer is represented by variables that take a limited number of discrete values.
* The first electronic digital computers, developed in the 1940's, were used primarily for numerical computations. In this case, the discrete elements are the digits. From this application, the term digital computer has emerged. In practice, digital computers function more reliably if only two states are used.
* Digital computers use the binary number system, which has two digits : 0 and 1. A binary digit is called a bit.
* Information is represented in digital computers in group of bits.

### Block diagram of Digital Computer :—

* A computer system is sub-divided into two functional entities : hardware and software.
* The hardware of the computer consists of all the electronic components and electromechanical devices that comprise the physical entity of the device.
* Computer software consists of the instructions and data that the computer manipulates to perform various data-processing tasks.
* A sequence of instructions for the computer is called a program.

* The data that are manipulated by the program constitute the data base.
* The System Software of a computer consists of a Collection of programs, whose purpose is to make more Effective use of the computer.
* The programs includes in a systems software package are referred to as the operating System. They are distinguished from application programs written by the user for the purpose of Solving particular problems.
* The hardware of the Computer is usually divided into three major parts, as shown in fig. below.

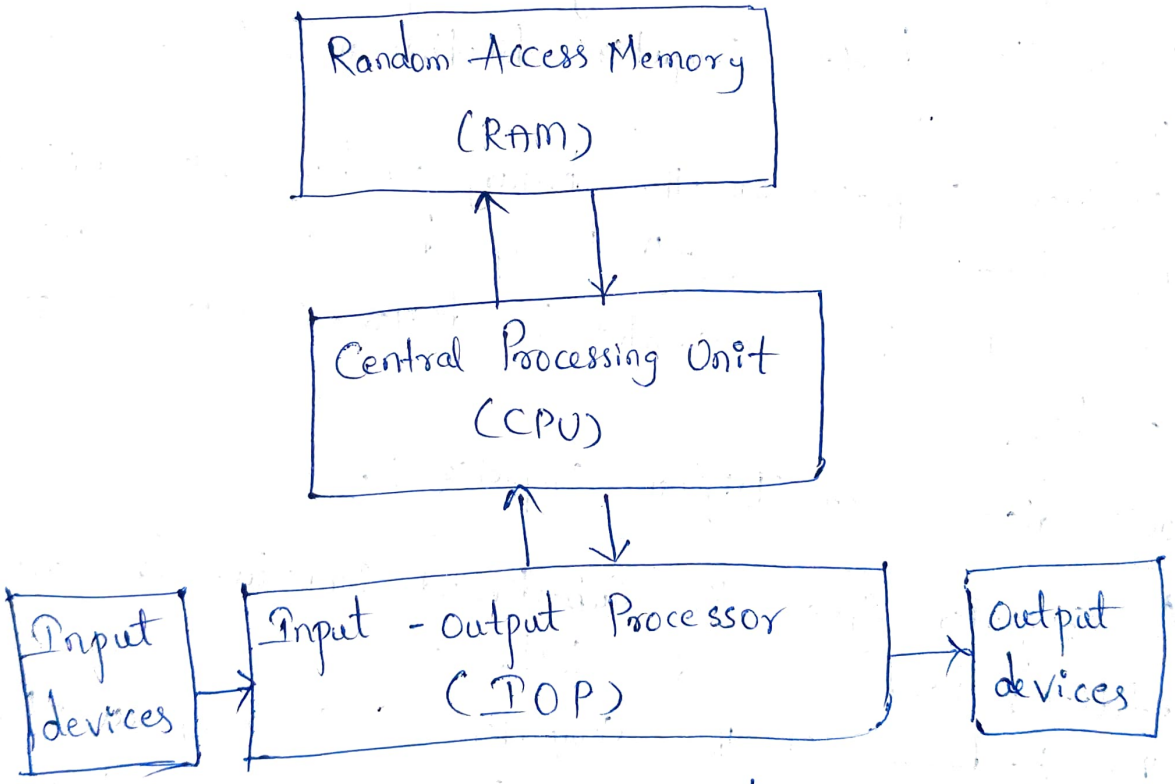



Fig : Block Diagram of a digital Computer.

Input devices :—

* Used to enter the data and instructions into the computer.

Ex:- Keyboard, mouse, Scanner, microphone etc.

* The input data is Sent to the IOP.

## Input - Output Processor (IOP) :—

* A Specialized processor that manages communication between input / output devices and the CPU.
* It reduces the CPU's workload by handling I/O operations independently.

## Central Processing Unit (CPU) :—

* CPU is the brain of the computer. It performs all arithmetic, logical and control operations.

### Main Components of the CPU:

**Arithmetic and Logical Unit (ALU) :-**

Performs arithmetic & logical operations.

**Control Unit (CU) :—**

Controls and coordinates the operations of the computer

**Registers :-** High - Speed Storage locations for temporary data and instructions.

### Functions of CPU:—

→ Fetches instructions from memory
→ Decode the instructions
→ Executes the instructions.
→ Sends results back to memory (or) output devices.

## Random Access Memory (RAM) :—

* The primary (main) memory of the computer. It Stores program (instructions), input data, intermediate results, final results (temporarily)

* RAM is volatile, meaning its contents are lost when the power is turned off.

Output Devices:-

* Present the processed information to the user.

Ex:- Monitor, Printer, Speakers, Projector.

Types of digital Computer:-

1) Micro Computer
2) Mini Computer
3) Main-frames
4) Super Computer

1) MicroComputer:-

* It is a small, relatively, inexpensive Computer with a microprocessor as its CPU. It includes a microprocessor, memory & I/O devices. Also known as "Personal Computer".

Ex:- Desktops, Laptops.

2) Mini Computer:-

It is emerged in the mid 1960's and were first dereloped by IBM Corporation. This may also be called a mid-range Computer. It may contain 1 or more processors, Support multi-processing & tasking.

3) Main frame Computer:

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

* Ability to run multiple operating system.

## 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 & organisation that require massive computing power.

## Definition of Computer Organisation, Computer Design and Computer Architecture :

* **Computer Organisation :**
  It 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 that the computer parts operate as intended.
* It includes :
  1) Control signals
  2) Processor Organisation
  3) Register "
  4) Memory "
  5) I/O "
  6) Bus Structure
  7) Data transfer Mechanisms.

Ex:- RAM communicates with CPU, register store data.

## Computer Design:—

* Computer Design is concerned with the hardware design of the Computer. Once the Computer specifications are formulated, it is the task of the designer to develop hardware ~~for~~ the System.
* Computer design is concerned with the determination of what hardware should be used and how the parts should be connected. This aspect of computer hardware is sometimes referred to as Computer implementation.
* It includes: 1) Logic Gate design
  2) ALU design
  3) Register design
  4) Control Unit design
  5) Cache implementation
  6) Circuit design
  7) Processor implementation

## Computer Architecture:—

Computer Architecture is concerned with the Structure and behaviour of the computer as seen by the user. It includes information formats, the instruction set and the techniques for addressing memory. The architectural design of a computer System is concerned with the specifications of the various functional modules, such as processors and

# UNIT - II

## Combinational Logic

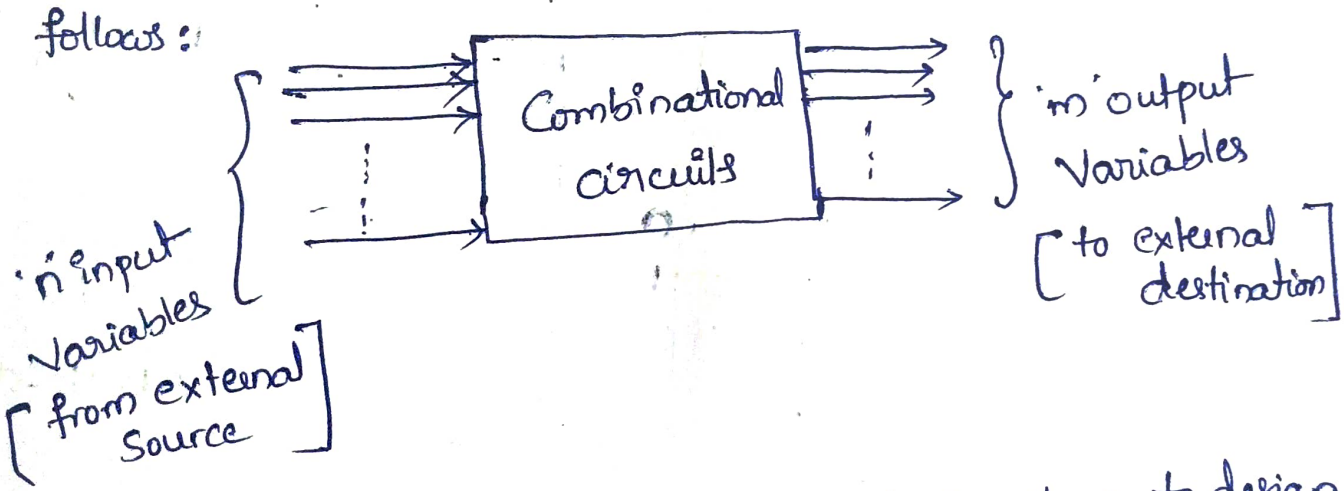
## Sequential Logic

# Unit-II

## Combinational Logic

### Combinational Circuits:

* The logic circuits for digital systems may be combination of sequential.
* The output of the circuit is depend only on the present i/p.
* It performs a specific information processing operation fully specified logically by set of boolean functions.
* It consists of i/p Variables, o/p Variable and logic Gates. The logic gates accept signal from input and generate signals to the output.
* The blockdiagram of the Combinational circuit is as follows:




**Design Procedure:** The following are the basic steps to design a Combinational circuits:

1. The problem is stated
2. Define the number of input variables and required o/p variables
3. Fix a letter symbols to the input and the outputs.
   [Ex: A, B, C, x, w, Y, F ....]

4. Get the relationship between input and output from the truth table

5. Obtain the simplified boolean expression for the outputs by using k-map

6. Draw the logic diagram using Gates.

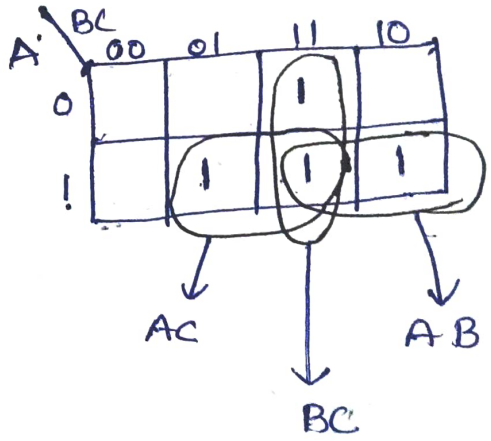
Example: Design a Combinational logic circuit with three inputs, the output is at logic 1 when more than one inputs are at logic 1.

Sol: No. of inputs = 3 → A, B, C
Output = 1 → Y

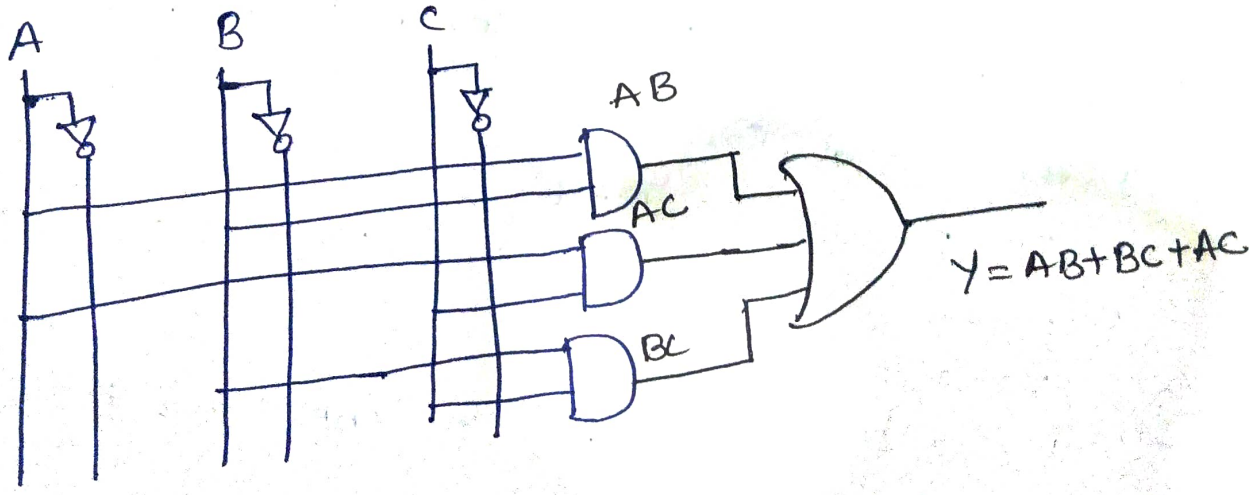
Condition: o/p is at logic 1 when more than one inputs are at logic 1.

Truth Table:

| A | B | C | Y |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |



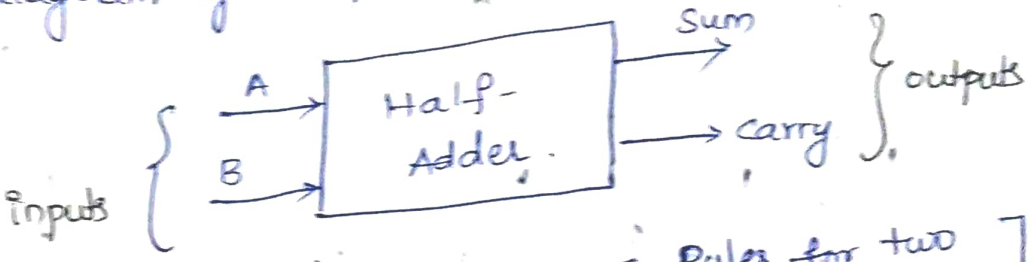

$Y = AB + BC + AC$



# Binary Adder - Subtractor:

## 1. Half - Adder:

* The basic operation of digital computer is binary addition. The circuit which perform the addition of binary bits are called as Adder.
* The logic circuit which perform the addition of two bit is called Half adder.
* The block diagram of Half - adder is shown in the figure.



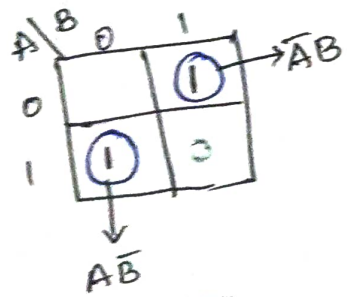

* Truth table for Half adder

| Inputs | | Outputs | |
|---|---|---|---|
| A | B | Sum | Carry |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

Rules for two bit addition

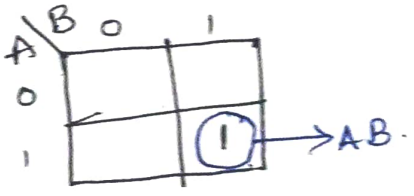
$0+0 = 0$
$0+1 = 1$
$1+0 = 1$
$1+1 = (10)_2$

Sum

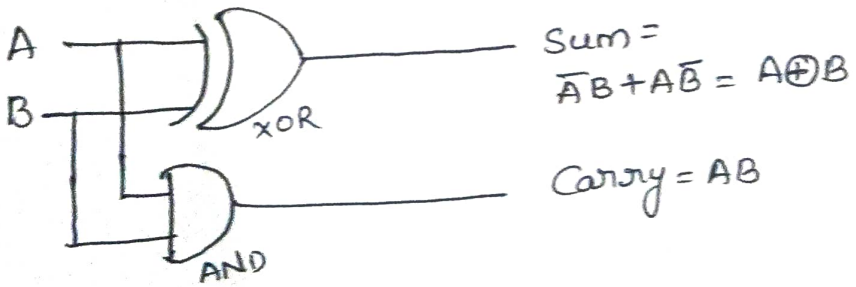



$Sum = \bar{A}B + A\bar{B}$

Carry

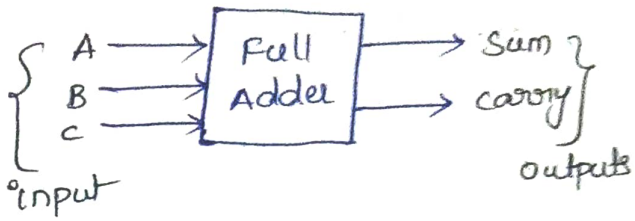



$Carry = AB$



* Full Adder The logic circuit which perform the addition of three bit is called Full Adder

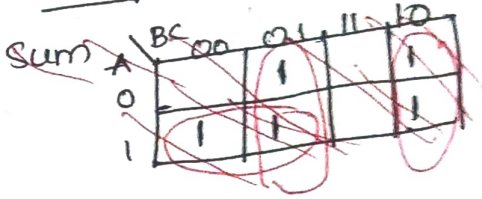
* The block diagram of the full-adder is shown in the figure.

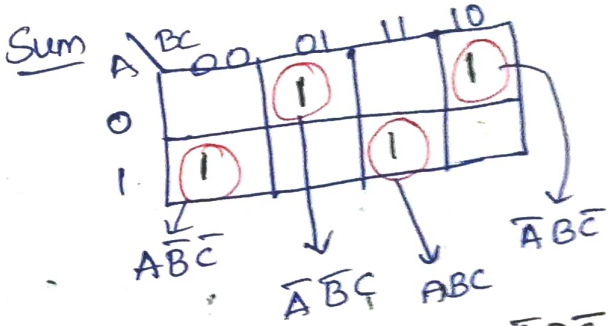



* The Truth Table for the full adder is

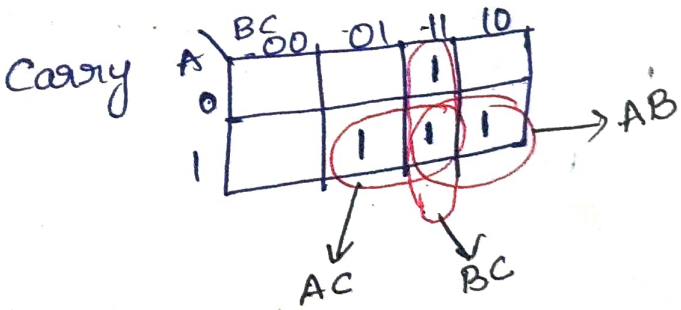
| Inputs | | | outputs | |
|---|---|---|---|---|
| A | B | C | Sum | Carry |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

k-map

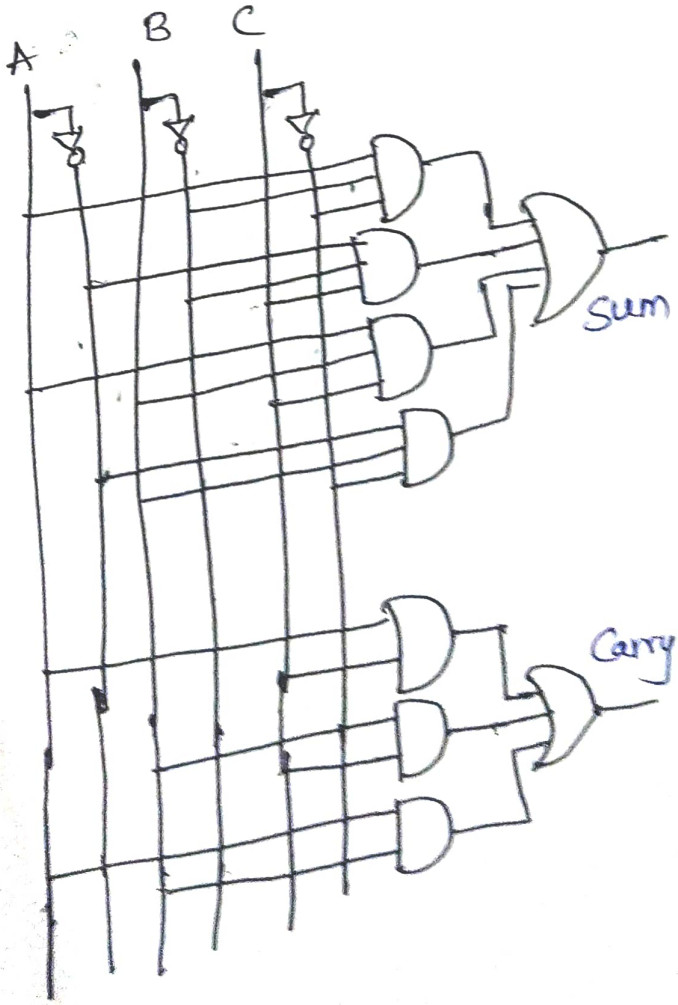



$Sum = A\bar{B}\bar{C} + \bar{A}\bar{B}C + ABC + \bar{A}BC$

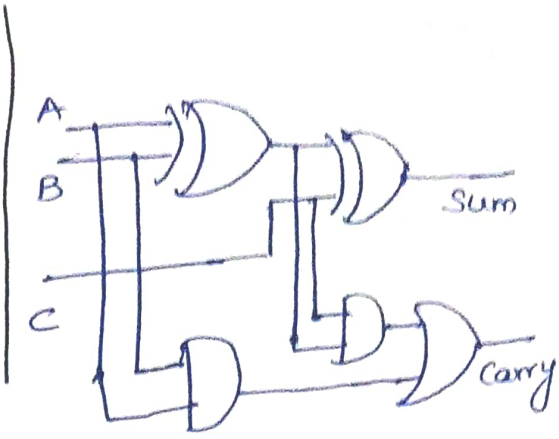



$Carry = AB + AC + BC$



By simplifying the expressions

$$\text{Sum} = \bar{A}\bar{B}C + \bar{A}B\bar{C} + A\bar{B}\bar{C} + ABC$$
$$= C(\bar{A}\bar{B} + AB) + \bar{C}(\bar{A}B + A\bar{B})$$
$$= C(A \odot B) + \bar{C}(A \oplus B)$$
$$= C(\overline{A \oplus B}) + \bar{C}(A \oplus B)$$
$$= C \oplus (A \oplus B)$$




$$\text{Carry} = AB + AC + BC$$
$$= AB + AC + (A + \bar{A})BC$$
$$= AB + AC + ABC + \bar{A}BC$$
$$= ABC(1 + C) + AC + \bar{A}BC \quad [\because 1 + A = 1]$$
$$= AB + AC + \bar{A}BC$$
$$= AB + AC(B + \bar{B}) + \bar{A}BC$$
$$= AB + ABC + A\bar{B}C + \bar{A}BC$$
$$= ABC(1 + C) + A\bar{B}C + \bar{A}BC$$
$$= AB + C(A\bar{B} + \bar{A}B)$$
$$= AB + C(A \oplus B)$$

## Subtractor:

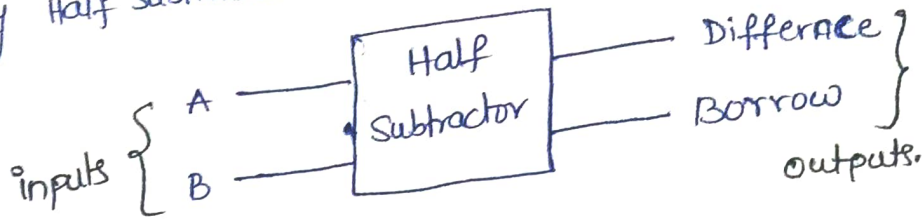
* Subtractor is the logic circuit which is used to subtract two binary number and provide difference and borrow as a output.
* Two types of Subtractor: 1) Half Subtractor
  2) Full Subtractor
* Rules for two bit Subtraction

| A | B | Result |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 (with borrow 1) |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

* Truth Table

## Half subtractor:

* Half subtractor is used for subtracting one single bit binary digit from another single bit binary digit.
* The block diagram of Half subtractor is shown below

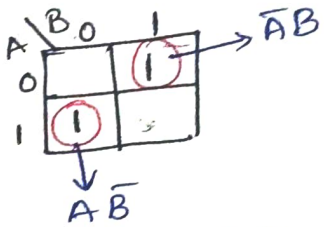



* The truth Table is

| A | B | Diff | Borrow |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |

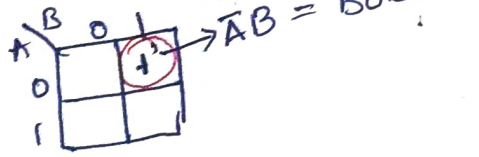
K-map

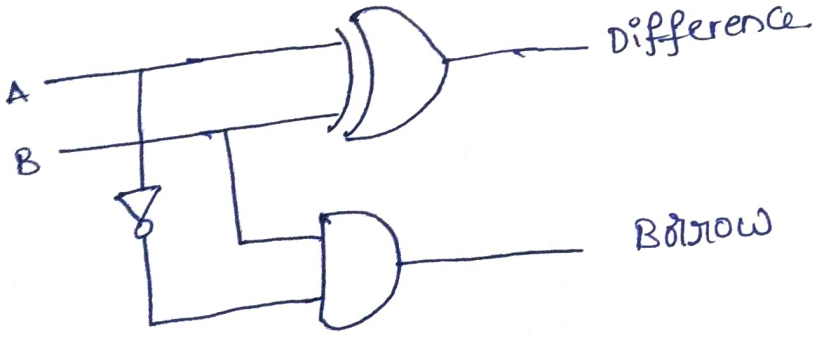
Difference




$Diff = A\bar{B} + \bar{A}B = A \oplus B$

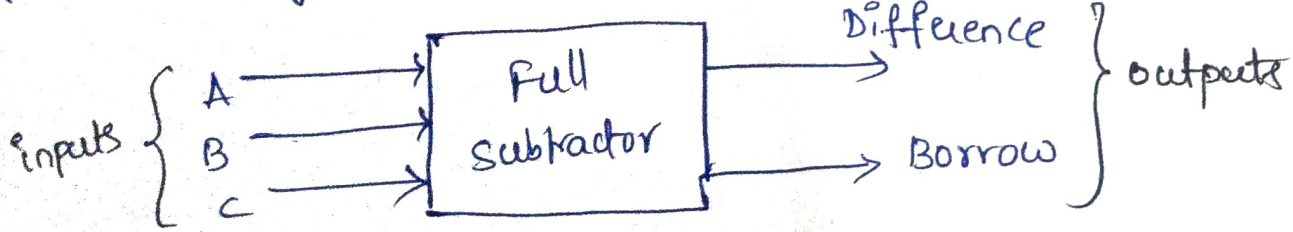
Borrow




**Full subtractor**: A logic circuit which is used for subtracting three single bit binary digit is known as full subtractor.

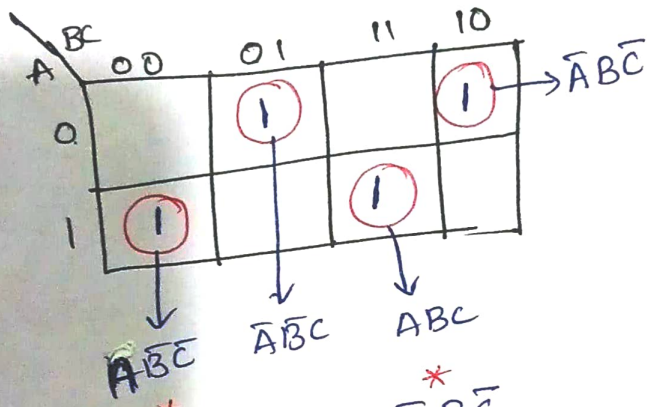
* The blockdiagram of full subtractor is shown in figure



* Truth Table

| Inputs | | | Outputs | |
|---|---|---|---|---|
| A | B | C | Diff | Borr |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |

K-map for Difference




$= A\bar{B}\bar{C} + \bar{A}\bar{B}C + ABC + \bar{A}B\bar{C}$

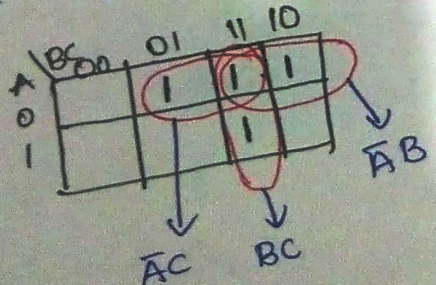
$= \bar{A}(\bar{B}C + B\bar{C}) + AC(BC + \bar{B}\bar{C})$

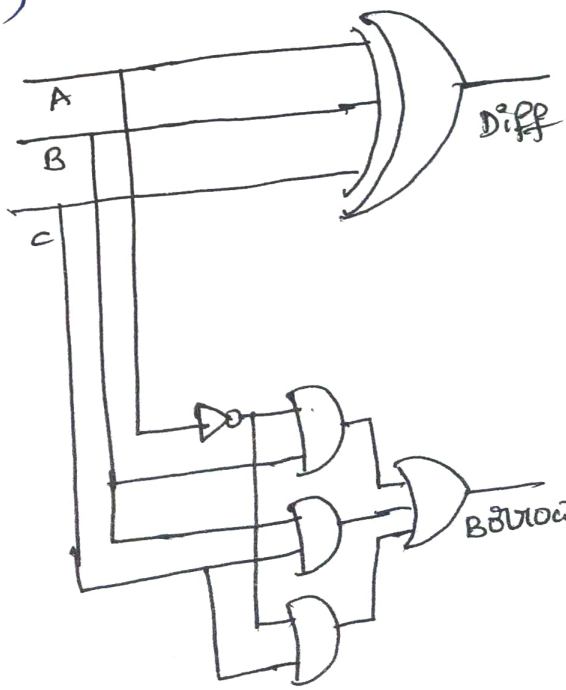
$= \bar{A}(B \oplus C) + A(B \odot C)$

$= \bar{A}(B \oplus C) + A(\overline{B \oplus C})$

$= A \oplus (B \oplus C)$

for borrow



## Decoder

- It is a combinational ckt with N i/p's & $2^N$ o/p's
- it selects one out of $2^N$ o/p's, based on the binary code applied

of the N i/p's

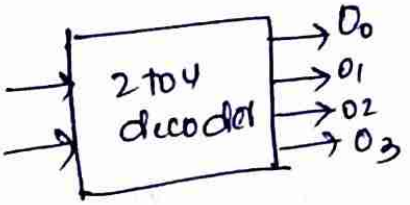
∴ Decoders are used to select one out of many hence are called 1 of M selector

Applications:

1. when several peripherals are connected to cpu, to select one among them, Decoders are used.
2. when the cpu store data in memory, it selects one out of several memory locations.

Decoders are always a part of the memory ✱

Available decoders: 1 to 2 decoders, 2 to 4, 3 to 8 and so on




(a) block diagram

| i/p's | | o/ps | | | |
|---|---|---|---|---|---|
| a | b | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 |

(b) Truth table

$D_0 = \text{Em}(0) = \bar{a}\bar{b}$

$D_1 = \text{Em}(1) = \bar{a}b$

$D_2 = \text{Em}(2) = a\bar{b}$

$D_3 = \text{Em}(3) = ab$

(c) design Eqn's

Active high o/p of 2 to 4 decoder

∴ Depending on the i/p combination 1 o/p is selected in active high o/p 2 to 4 decoder the selected o/p will go high. But in case of active low 2 to 4 decoder, the selected o/p will go low.




(a) block diagram

| i/ps | o/p's | | | |
|---|---|---|---|---|
| a b | $O_4$ | $O_3$ | $O_2$ | $O_1$ |
| 0 0 | 1 | 1 | 1 | 0 |
| 0 1 | 1 | 1 | 0 | 1 |
| 1 0 | 1 | 0 | 1 | 1 |
| 1 1 | 0 | 1 | 1 | 1 |

(b) truth table

$O_1 = \pi M(0) = a+b$

$O_2 = \pi M(1) = a+\bar{b}$

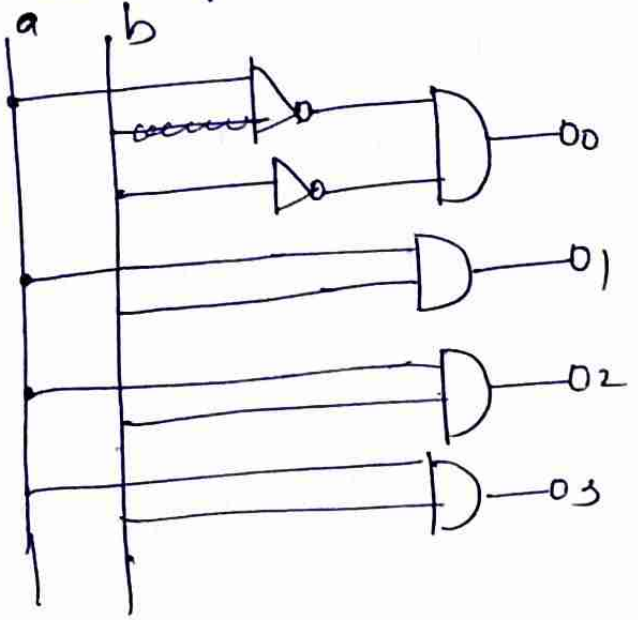
$O_3 = \pi M(2) = \bar{a}+b$

$O_4 = \pi M(3) = \bar{a}+\bar{b}$

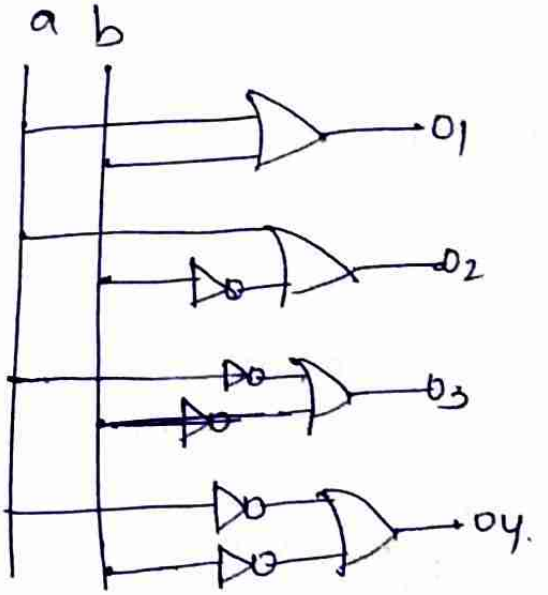
(c) design Equation

logic ckt of 2 to 4 decoder

Active-high o/p




Active-low o/p

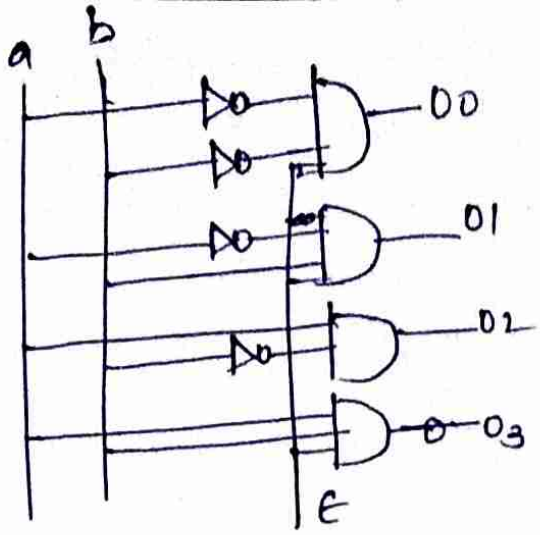



Decoder with Enable line

this Enable pin (E) is used to active the circuit if E=1 then the ckt is activated and the o/p's are selected accordingly. of $E_1=0$, the decoder is 'Disabled'. the Decoders are available in IC form

Advatage of An enable pin

the un-used Ic's on a mother board of a computers can be disabled using the Enable signal so that the power consumption can be reduced.




| I/p's | O/p's |
|---|---|
| e ab | $O_3$ $O_2$ $O_1$ $O_0$ |
| 0 x x | 0 0 0 0 |
| 1 0 0 | 0 0 0 1 |
| 1 0 1 | 0 0 1 0 |
| 1 1 0 | 0 1 0 0 |
| 1 1 1 | 1 0 0 0 |

$O_0 = e\bar{a}\bar{b}$

$O_1 = e\bar{a}b$

$O_2 = ea\bar{b}$

$O_3 = eab$

## logic function generation using a Decoder

→ A combinational logic ckt is defined by the following Boolean fn's $f_1 = a\bar{b}\bar{c} + ac$, $f_2 = ab\bar{c} + \bar{a}b$, $f_3 = ab\bar{c} + ab$

Design the ckt with a decoder and external gates.

Sol The Combinational logic ckt to be designed is a 3-i/p 3-o/p ckt

$f_1 = a\bar{b}\bar{c} + ac = \bar{a} + b + \bar{c} + ac = \bar{a}(b+\bar{b}) + b(\bar{a}+a)(\bar{c}+c)$

$+\bar{c}(a+\bar{a})(b+\bar{b}) + ac(b+\bar{b})$

$= \bar{a}bc + \bar{a}b\bar{c} + \bar{a}bc + \bar{a}\bar{b}\bar{c} + ab\bar{c} + a\bar{b}\bar{c} + abc$

$= \Sigma m(0,1,2,3,4,5,6,7) = 1$

thus also can be written as:

$f_1 = a\bar{c}\bar{b} = a\bar{c} + b + ac = 1 + b = 1 \Rightarrow f_1 = 1$

$f_2 = ab\bar{c} + \bar{a}b = a(\bar{b}+\bar{c}) + \bar{a}b = a\bar{b}(c+\bar{c}) + a\bar{c}(b+\bar{b}) + \bar{a}b(c+\bar{c})$.

$= abc + a\bar{b}\bar{c} + ab\bar{c} + \bar{a}bc + \bar{a}b\bar{c} = \Sigma m(2,3,4,5,6)$

$f_3 = a\bar{b}c + ab = abc + ab(c+\bar{c}) = abc + abc + ab\bar{c}$

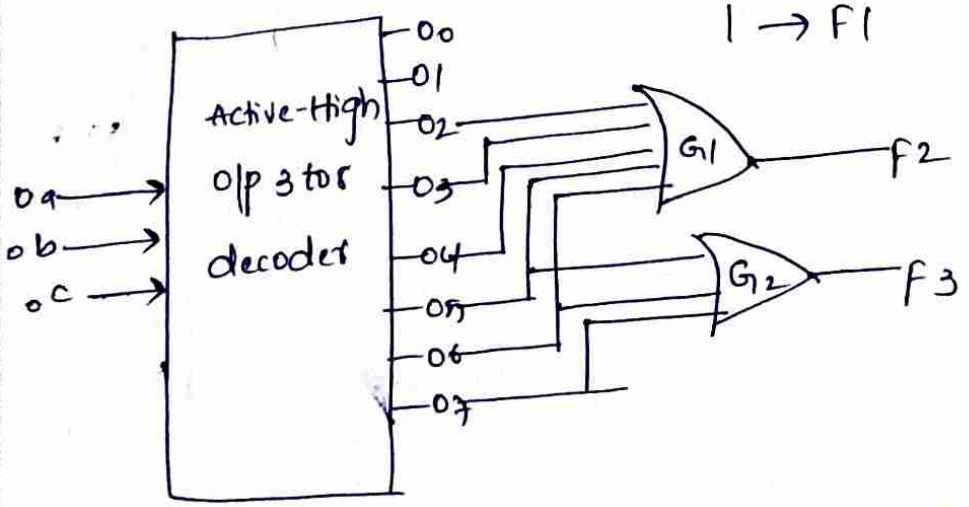
$= \Sigma m(5,6,7)$

| min term | I/p's a | b | c | O/p's f1 | f2 | f3 | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 00 |
| 1 | 0 | 0 | 1 | 1 | 0 | 0 | 01 |
| 2 | 0 | 1 | 0 | 1 | 1 | 0 | 02 |
| 3 | 0 | 1 | 1 | 1 | 1 | 0 | 03 |
| 4 | 1 | 0 | 0 | 1 | 1 | 0 | 04 |
| 5 | 1 | 0 | 1 | 1 | 1 | 1 | 05 |
| 6 | 1 | 1 | 0 | 1 | 1 | 1 | 06 |
| 7 | 1 | 1 | 1 | 1 | 0 | 1 | 07 |

Truth table for given fn

| I/p's a | b | c | 07 | 06 | 05 | 04 | 03 | 02 | 01 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

Truth table for active-high 3 to 8 decoder



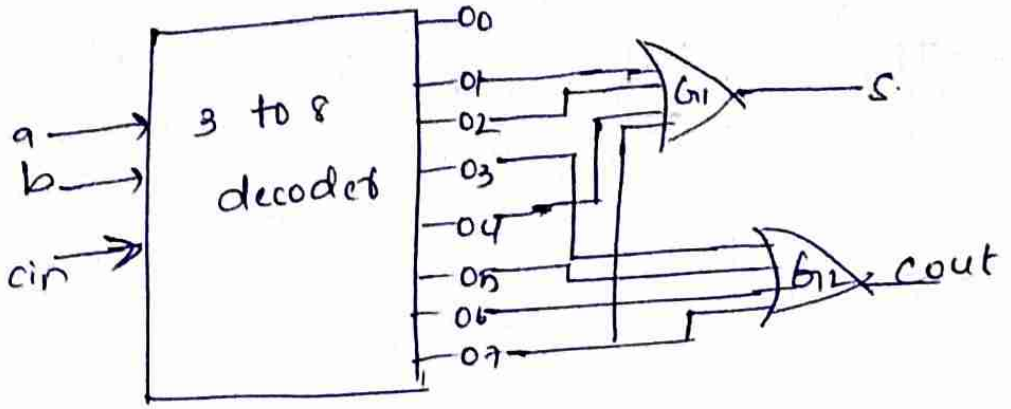

| a | b | c | i/p's to OR Gate (G1) 02 | 03 | 04 | 05 | 06 | i/p's to OR gate 05 | 06 (G2) | 07 | o/p's | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |

→ Implement a FA using decoder and OR gate

FA is a 3-I/p 2-O/p logic ckt

$S = \bar{a}b\,cin + \bar{a}b\,c\bar{in} + ab\,\bar{c}in + ab\,cin = \Sigma m(1,2,4,7)$

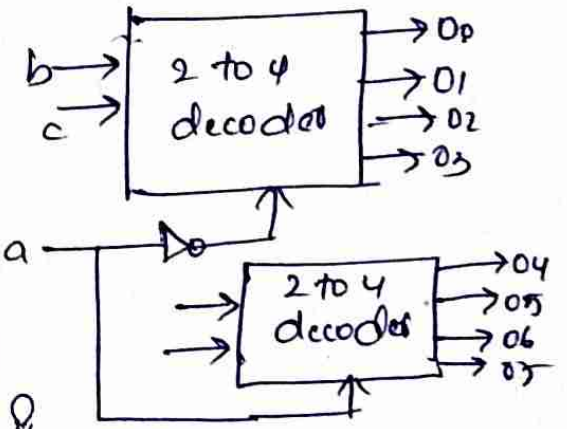
$Cout = ab + b\,cin + c\,ain = \Sigma m(3,5,6,7)$




→ Decoder tree

Enable pin can be used to construct a larger decoder from a smaller decoder. construction of multiple decoders from smaller size is called a decoder tree.

→ design a 3 to 8 decoder using 2 to 4 decoder and other required gate

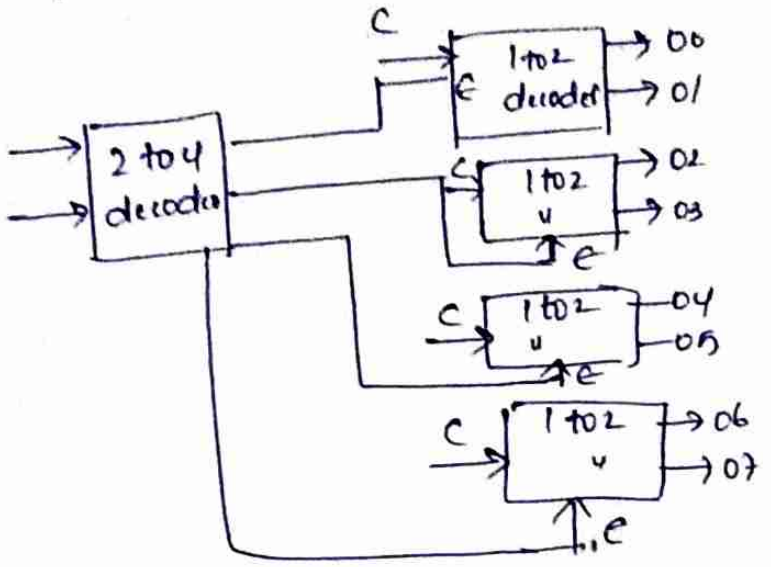



If $a=0 \rightarrow E_1=1$ and 1st 2 to 4 decoder is activated and $E_2=0$ so it is disabled

If $a=1 \rightarrow E_1=0$ & $E_2=1$ so second decoder is activated and first one is disabled.

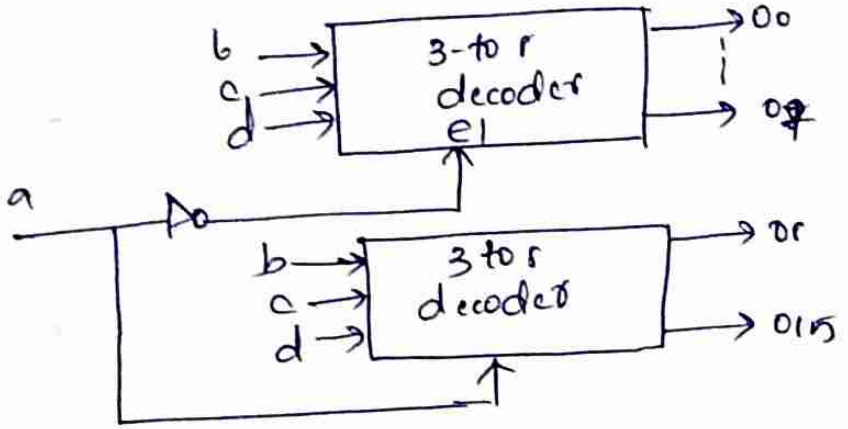
| i/p | | | o/p | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| a | b | c | O7 | O6 | O5 | O4 | O3 | O2 | O1 | O0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

→ Design a 3 to 8 decoder using 1 to 2 decoder




| a | b | c | o/p |
|---|---|---|---|
| 0 | 0 | 0 | 00 |
| 0 | 0 | 1 | 01 |
| 0 | 1 | 0 | 02 |
| 0 | 1 | 1 | 03 |
| 1 | 0 | 0 | 04 |
| 1 | 0 | 1 | 05 |
| 1 | 1 | 0 | 06 |
| 1 | 1 | 1 | 07 |

Construct a 4×16 decoder with 3×8 decoder




→ Encoder

the function of an encoder is just the opposite of that of a decoder

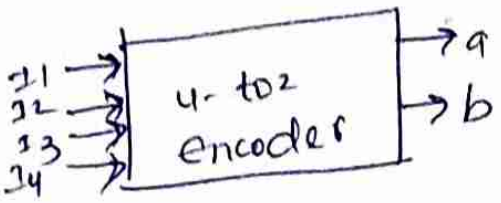
→ It is a combinational ckt with $2^N$ o/ps and N outputs

→ it generate binary code (o/p) for a selected i/p among $2^N$ inputs.

→ Encoders are used in A to D converters.

4 to 2, 8 to 3 and 16 to 4 encoders are available for 4 i/p, 16 i/p combinations are there but only 4

are used Here. So remaining 12 are invalid combinations

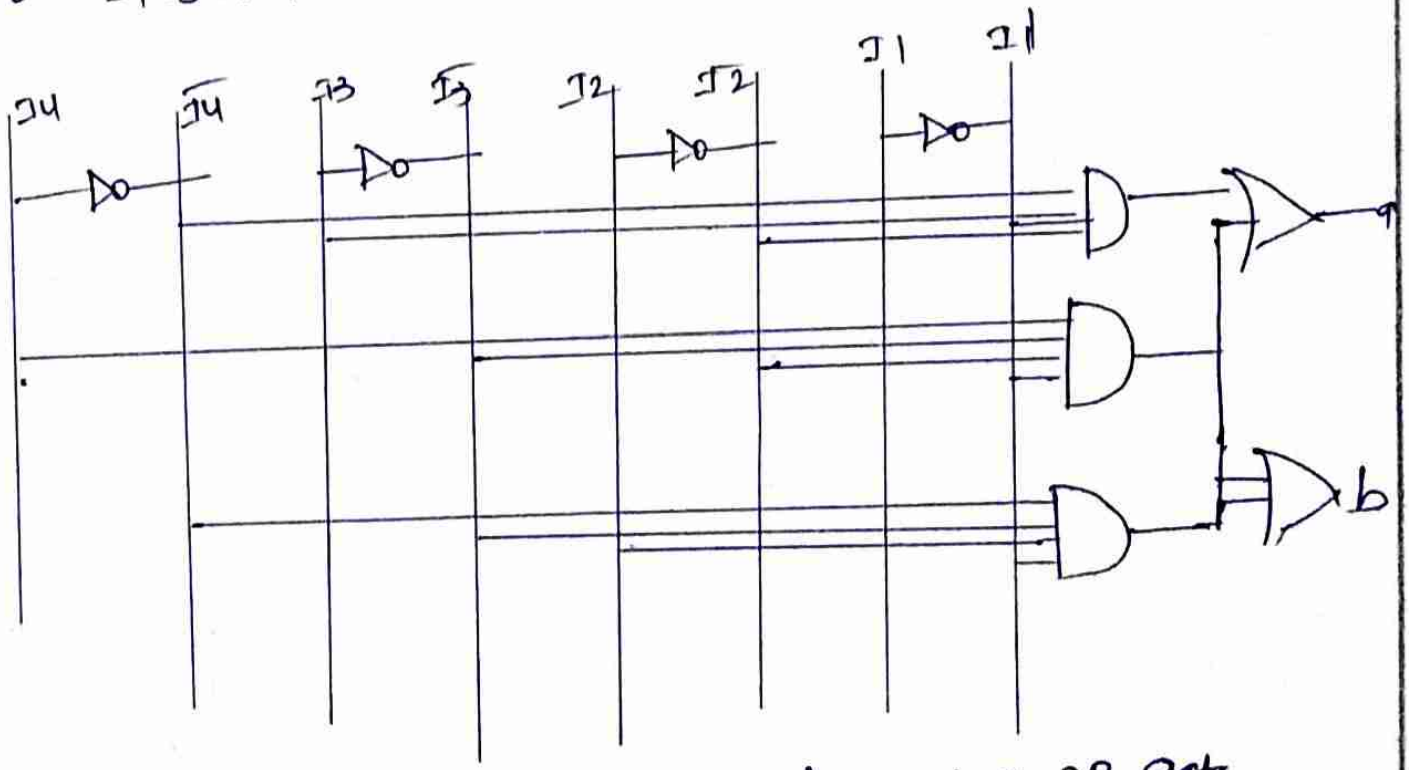



block diagram

| I/Ps | | | | O/Ps | |
|---|---|---|---|---|---|
| I4 | I3 | I2 | I1 | a | b |
| 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 |

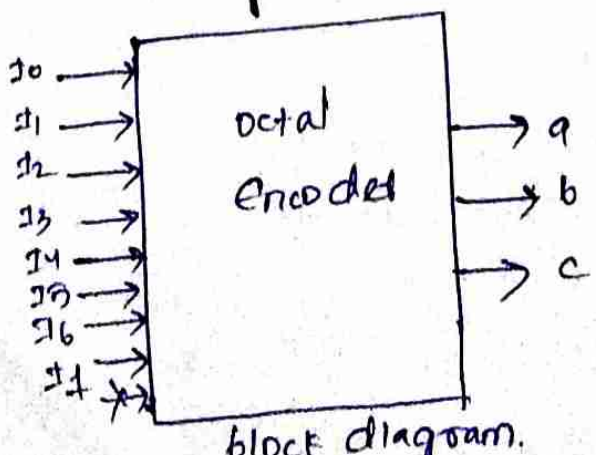
$a = I_4 \bar{I}_3 \bar{I}_2 \bar{I}_1 + I_4 \bar{I}_3 \bar{I}_2 \bar{I}_1 = \Sigma m(4,8)$

$b = I_4 \bar{I}_3 I_2 \bar{I}_1 + I_4 \bar{I}_3 \bar{I}_2 \bar{I}_1 = \Sigma m(2,8)$




→ Design of an octal to binary encoder using OR gate

this is nothing but 8-to-3 encoder




block diagram.

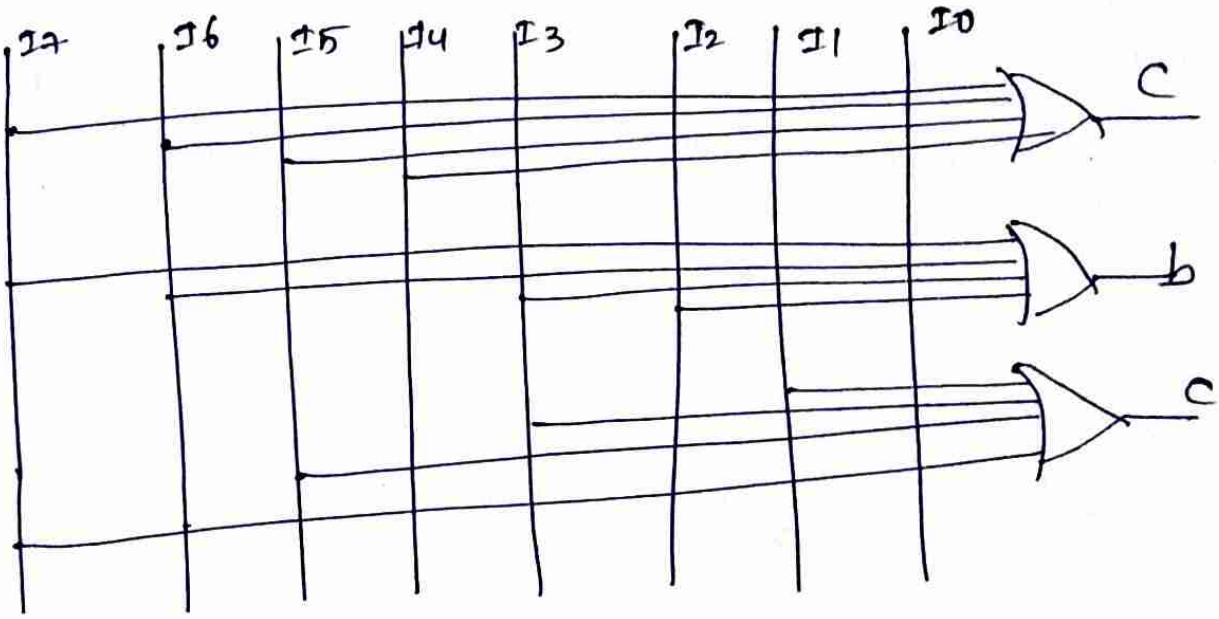
| octal digit | I/p | outputs a | b | c |
|---|---|---|---|---|
| 0 | I0 | 0 | 0 | 0 |
| 1 | I1 | 0 | 0 | 1 |
| 2 | I2 | 0 | 1 | 0 |
| 3 | I3 | 0 | 1 | 1 |
| 4 | I4 | 1 | 0 | 0 |
| 5 | I5 | 1 | 0 | 1 |
| 6 | I6 | 1 | 1 | 0 |
| 7 | I7 | 1 | 1 | 1 |

Truth table

$a = I4 + I5 + I6 + I7$

$b = I7 + I6 + I3 + I2$

$c = I7 + I5 + I3 + I1$



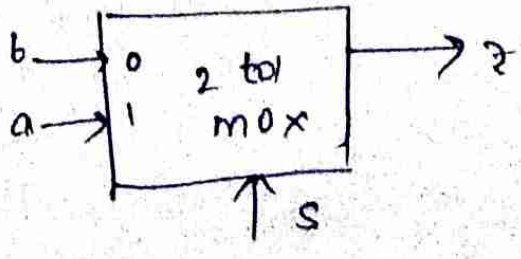

→ Multiplexer (Data selector) (or) mux

It is a combinational ckt with N select lines $2^N$ I/p's and one o/p depending on the binary code of N selected it select one of the $2^N$ I/p's and connect it to the o/p

(N to-1 mux)

2-to-1 mux, 4 to 1 mux and 8 to 1 mux are available

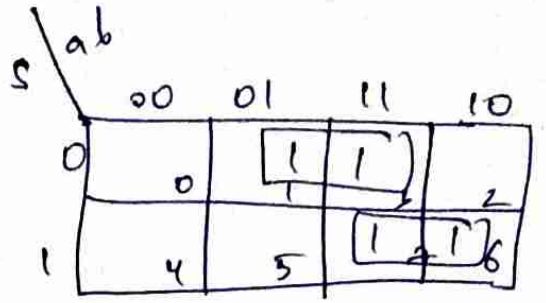
in IC form



| i/p | | | o/p |
|---|---|---|---|
| s | a | b | z |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

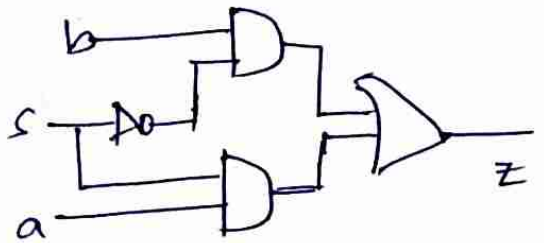
s=0, z=1

s=1, z=q

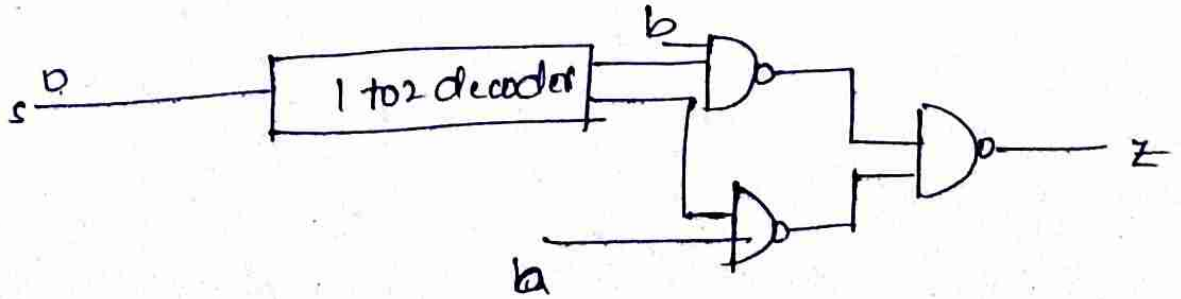



when s=0, for the 4 combinations of i/p, the LSB bit b is selected when s=1 for the 4 combinations of i/p a is selected

∴ o/p z = sa + $\bar{s}$b → characteristic eqn of 2-to-1 mux

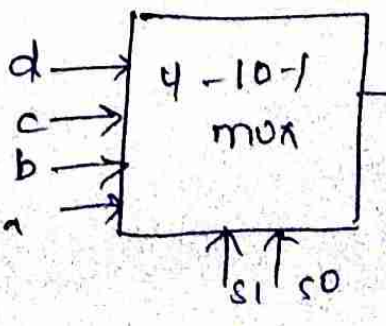



logic ckt of 2 to 1 mux




4 to 1 mux

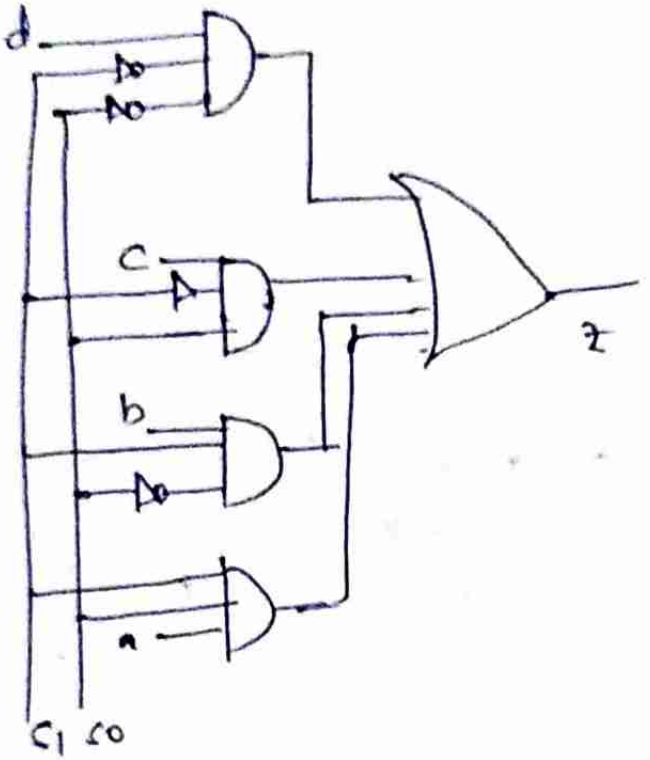
| selection line | | o/p(z) |
|---|---|---|
| $s_1$ | $s_0$ | |
| 0 | 0 | d |
| 0 | 1 | c |
| 1 | 0 | b |
| 1 | 1 | a |




block diagram

for 00 i/p 'd' is selected, for 01 i/p 'c' is selected, for 10 i/p 'b' is selected and for 11 i/p, 'a' is selected.

characteristic eqn: $z = \bar{s_1}\bar{s_0}\,d + \bar{s_1}s_0\,c + s_1\bar{s_0}\,b + s_1 s_0\,a$.




Decoder truth table

| a | b | $O_3$ | $O_2$ | $O_1$ | $O_0$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 |

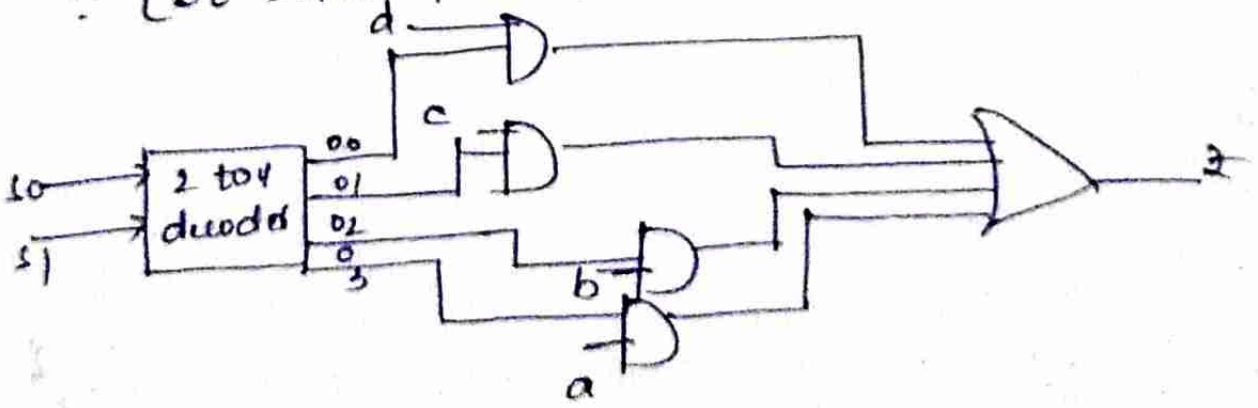
$O_0 = \bar{a}\bar{b}$, $O_1 = \bar{a}b$, $O_2 = a\bar{b}$

$O_3 = ab$

logic ckt of 4 to 1 mux

Ac to decoder ckt, $\bar{s_1}\bar{s_0}$, $\bar{s_1}s_0$, $s_1\bar{s_0} + s_1 s_0$ selection lines can be selected using decoder

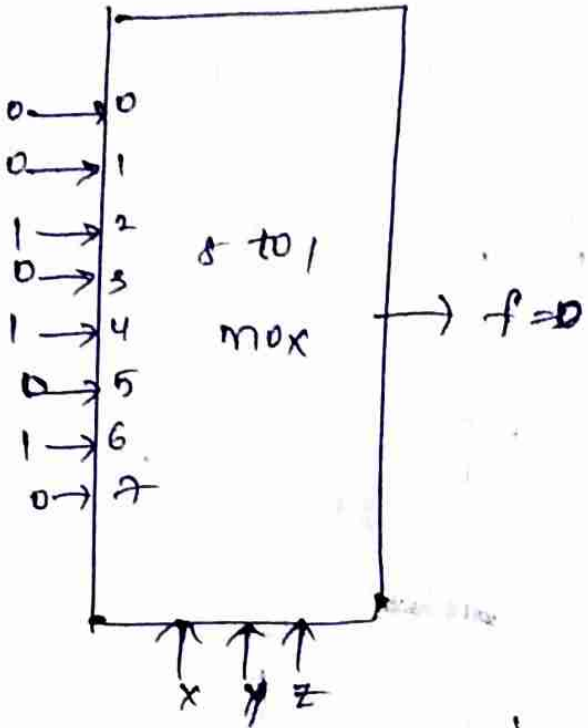
$\therefore C_{00} = \bar{s_1}\bar{s_0}$, $O_1 = \bar{s_1}s_0$, $O_2 = s_1\bar{s_0}$, $O_3 = s_1 s_0$



→ Implement the boolean function $F(x, y, z) = \Sigma m(2, 4, 6)$ with a mux

Sol

| minterm | I/p's x y z | o/p f |
|---|---|---|
| 0 | 0 0 0 | 0 |
| 1 | 0 0 1 | 0 |
| 2 | 0 1 0 | 1 |
| 3 | 0 1 1 | 0 |
| 4 | 1 0 0 | 1 |
| 5 | 1 0 1 | 0 |
| 6 | 1 1 0 | 1 |
| 7 | 1 1 1 | 0 |




To implement any function, the I/p's of a given fn should be considered as selected lines of a mux and the o/p (f) can be conidered as i/p's to the mux

If $xyz = 101$ then 5th minterm is selected and it is directly connected to o/p so $f = 0$ 11y, the remaining o/p's can be generated for different combinations of selective lines

→ Realize a full subtractor using multiplexers

It is a 3-i/p (x, y, bin) and 2 o/p

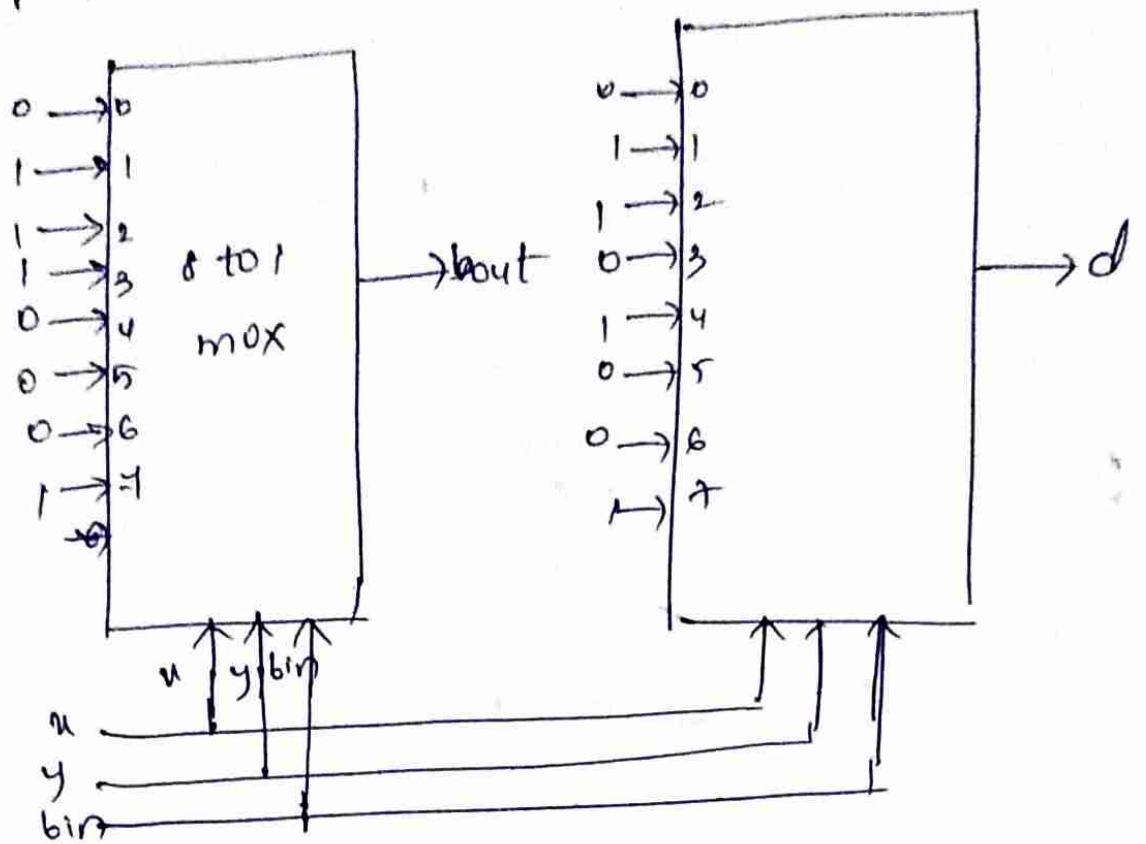
$b_{out} = \Sigma m(1, 2, 3, 7) = \bar{x}\,bin + \bar{x}y + ybin$

$d = \Sigma m(1, 2, 4, 7) = \bar{x}\bar{y}\,bin + \bar{x}y\,\overline{bin} + x\bar{y}bin + xybin$

$b_{out}$ can be implemented by using one 8 to 1 mux

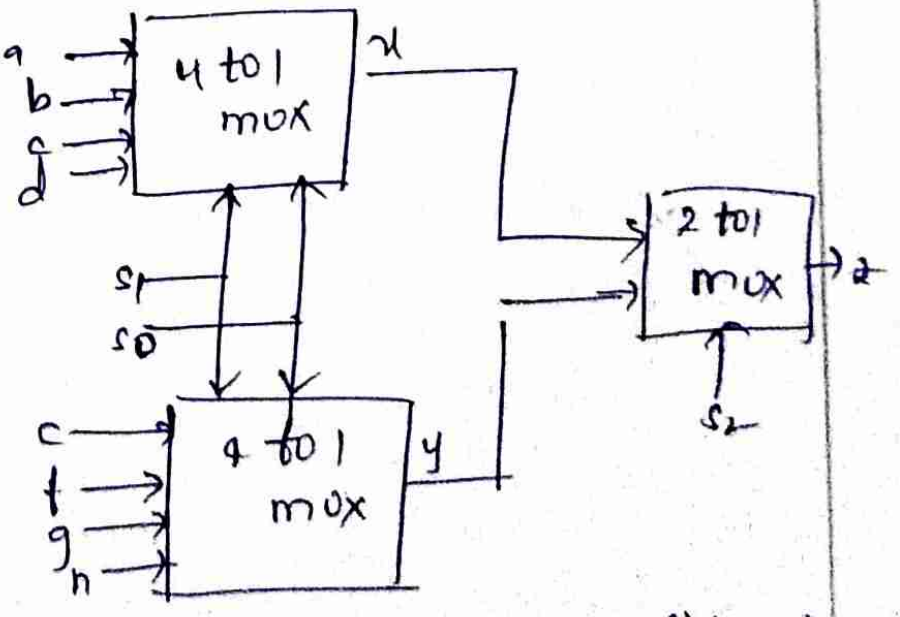
d u u 0 0 7

→ Implement the boolean function $F(x,y,z) = \Sigma m(2,4,6)$




→ modular design using IC chips (multiplexer true) construction of 8 to 1 mux using 4 to 1 mux

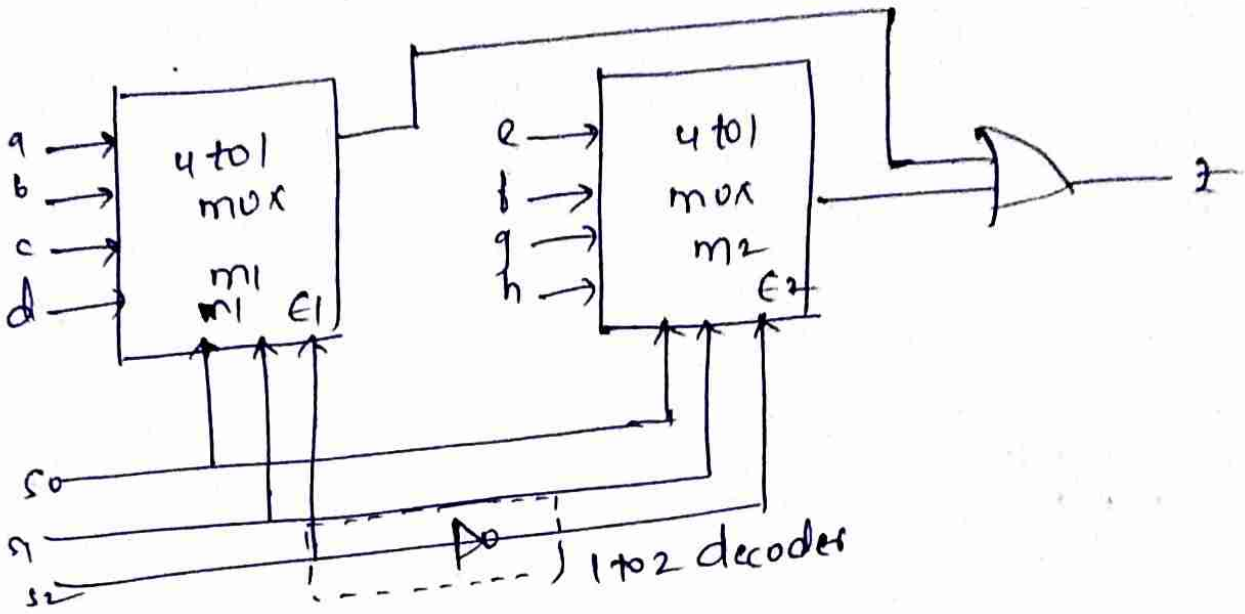
| select lines | o/p: |
|---|---|
| $s_2$ $s_1$ $s_0$ | x y z |
| 0 0 0 | a e a |
| 0 0 1 | b f b |
| 0 1 0 | c g c |
| 0 1 1 | d h d |
| 1 0 0 | a e e |
| 1 0 1 | b f f |
| 1 1 0 | c g g |
| 1 1 1 | d h h |




for $s_1$ $s_0$ combination, whatever may the $s_2$ value ('0' or '1') $x = a, b, c, d$ only, for $s_1$ $s_0$ combination, whatever may be the $s_2$ value, $y = e, f, g, h$

If $S_2 = 0$, x is selected and if $S_2 = 1$, y is selected

→ multiplex free using smaller mux and decoder



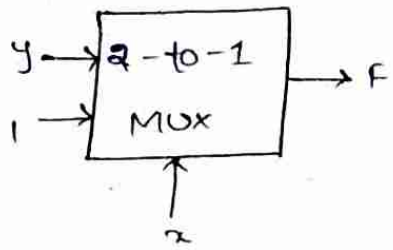

Here, $S_2$ acts as enable signal.

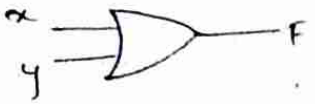
If $S_2 = 0$, $E_1 = \overline{0} = 1$ (active-low signal) so, m is selected and the i/p's of x are i/p's of so, if $S_2 = 1$, $E_2$ is activated and the o/p's of y are e, f, g & h finally by giving x and y signals to the OR gate it generate the signal z.

| Selection line i/p's | | | Enable i/p's | | o/p's | | |
|---|---|---|---|---|---|---|---|
| $S_2$ | $S_1$ | $S_0$ | $E_1$ | $E_2$ | x | y | $z = x+y$ |
| 0 | 0 | 0 | 0 | 1 | a | 0 | a |
| 0 | 0 | 1 | 0 | 1 | b | 0 | b |
| 0 | 1 | 0 | 0 | 1 | c | 0 | c |
| 0 | 1 | 1 | 0 | 1 | d | 0 | d |
| 1 | 0 | 0 | 1 | 0 | 0 | e | e |
| 1 | 0 | 1 | 1 | 0 | 0 | f | f |
| 1 | 1 | 0 | 1 | 0 | 0 | g | g |
| 1 | 1 | 1 | 1 | 0 | 0 | h | h |

# Multiplexer Realization of switching functions.

→ MUX as OR gate




if $x=0$, $F=y$
if $x=1$, $F=1$

| x | y | F |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

AC to truth table, $F = \bar{x}y + x\bar{y} + xy = \bar{x}y + x(y+\bar{y})$

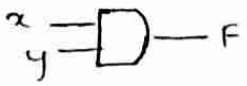
$\therefore F = \bar{x}y + x \cdot 1$ —① this equation is in the form of $z = \bar{s}b + s \cdot a$ —②

(characteristic equation of 2-to-1 MUX).

from ① & ② $F = z$, $s = x$, $y = b$ and $a = 1$.

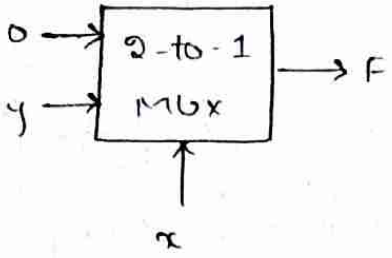
→ MUX as AND gate :-

$F = xy = \bar{x} \cdot 0 + x \cdot y$




| x | y | F |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

→ $z = F$, $s = x$, $b = 0$ and $a = y$.

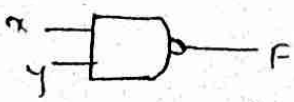



if $x=0$, $F=0$
if $x=1$, $F=y$

→ MUX as NAND gate :-

$F = \bar{x}\bar{y} + \bar{x}y + x\bar{y} = \bar{x}(y+\bar{y}) + x\bar{y}$

$= \bar{x} \cdot 1 + x\bar{y}$ ($\because s = x$, $b = 1$ & $a = \bar{y}$)




| x | y | F |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

## Comparison b/w sequential circuit & combinational circuit

| combinational circuits | Sequential circuit |
|---|---|
| - o/p's at any instant of the depends only on present i/p | - o/p is depend on present i/p & present states. |
| - Memory unit is not required | - memory is required to store present state values |
| - Easy to design | - Harder to design |
| - Faster than sequential | - slower than combinational |

## comparison b/w synchronous and Asynchronous sequential circuits.

| Synchronous Seq circuits | Asynchronous Seq. circuits |
|---|---|
| - In synchronous circuits memory elements are clocked flipflops) flipflops. | - Memory elements are un-clocked flipflops (Latches) |
| - Easier to design | - more difficult to design |
| - clock signal is required to active the circuit. | - clock is not required |
| - Max operating speed of clock depends on time-delays involved | - The operating speed is high due to the absence of clock signal. |

## Latches and flipflops:

- Latch or flipflop is nothing but a single bit storage element flipflops are the basic building blocks of most sequential circuits.

- A flip flop is known as bistable multivibrator has 2 stable states.
- The flipflop is said to be in high state (or) logic 1 state or SET state when Q=1, in low state logic 0 state or RESET state or CLEAR state when Q=0. A flipflop contains one or more i/p's.

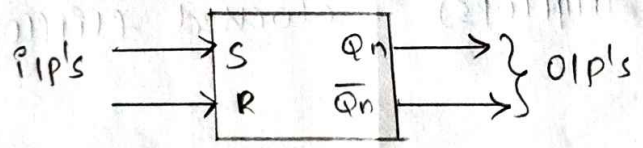
Latch → non-clocked flipflop

flipflop → gated latch

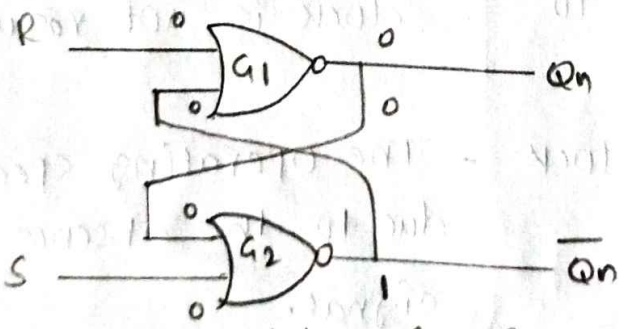
S-R (or) RS latch (or) Asynchronous latch:

It can be constructed using either 2 cross-coupled NAND gates or two cross coupled NOR gates.

The NOR gate S-R latch (Active high SR latch)




a) logic symbol

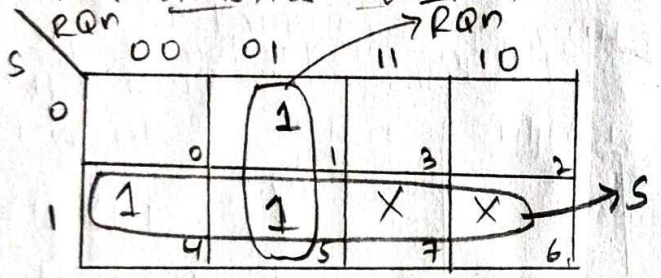



b) logic diagram.

| i/p's | P.s | N.s | state |
|---|---|---|---|
| S R | Qn | Qn+1 | |
| 0 0 | 0 | 0 | NOchange |
| 0 0 | 1 | 1 | |
| 0 1 | 0 | 0 | Reset |
| 0 1 | 1 | 0 | |
| 1 0 | 0 | 1 | set |
| 1 0 | 1 | 1 | |
| 1 1 | 0 | X | Indetermin-ate |
| 1 1 | 1 | X | (Invalid) |

c) Truth Table

d) characteristics equation

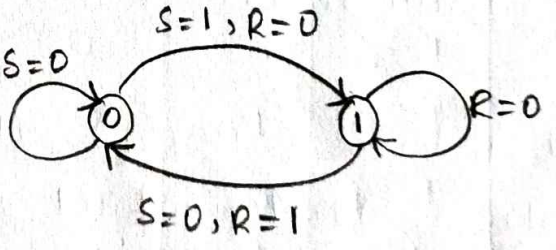



$Q_{n+1} = S + \bar{R}Q_n$

e) excitation table

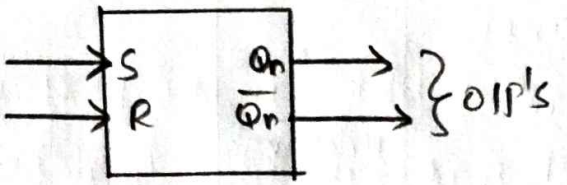
| Qn | Qn+1 | S | R | S R |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 X |
| | | 0 | 1 | |
| 0 | 1 | 1 | 0 | 1 0 |
| 1 | 0 | 0 | 1 | 0 1 |
| 1 | 1 | 0 | 0 | X 0 |
| | | 1 | 0 | |

f) state Diagram

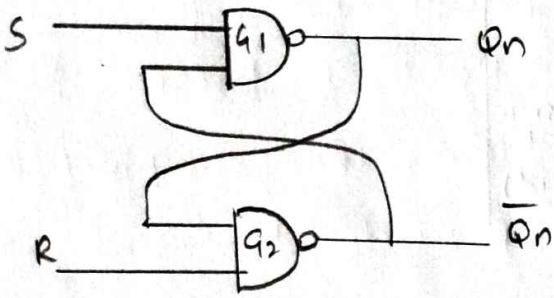


# S-R NAND Latch

a) Logic symbol




b) Logic diagram



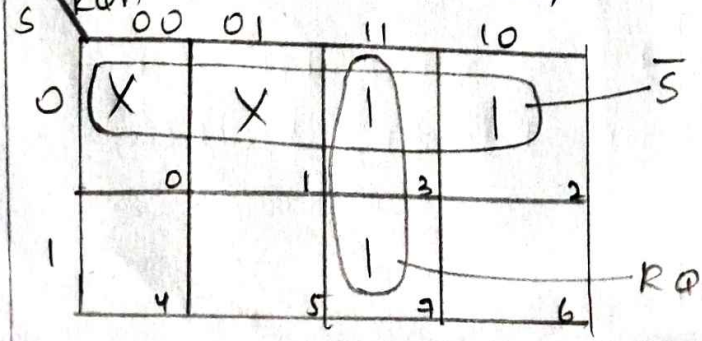

S=0, R=1, set

S=1, R=0, Reset

c) Truth table.

| i/p's | | P.s | N.s | state |
|---|---|---|---|---|
| S | R | Qn | Qn+1 | |
| 0 | 0 | 0 | X | Indetermin-ate (Invalid) |
| 0 | 0 | 1 | X | |
| 0 | 1 | 0 | 1 | set |
| 0 | 1 | 1 | 1 | |
| 1 | 0 | 0 | 0 | Reset |
| 1 | 0 | 1 | 0 | |
| 1 | 1 | 0 | 0 | No change |
| 1 | 1 | 1 | 1 | |

d) Characteristic equation

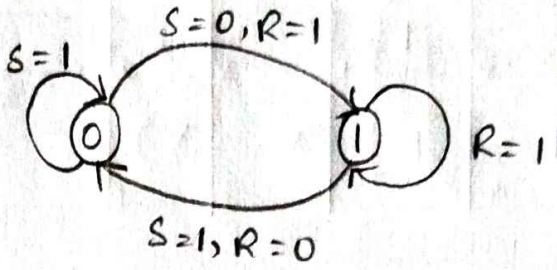



$Q_{n+1} = \overline{S} + RQ_n$

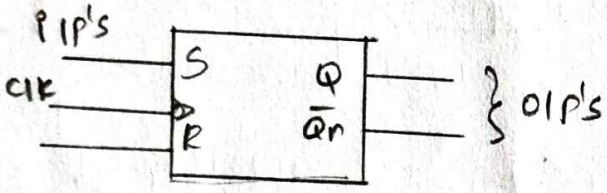
e) Excitation table

| Qn | Qn+1 | S | R |
|---|---|---|---|
| 0 | 0 | 1 | X |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | X | 1 |

f) state diagram

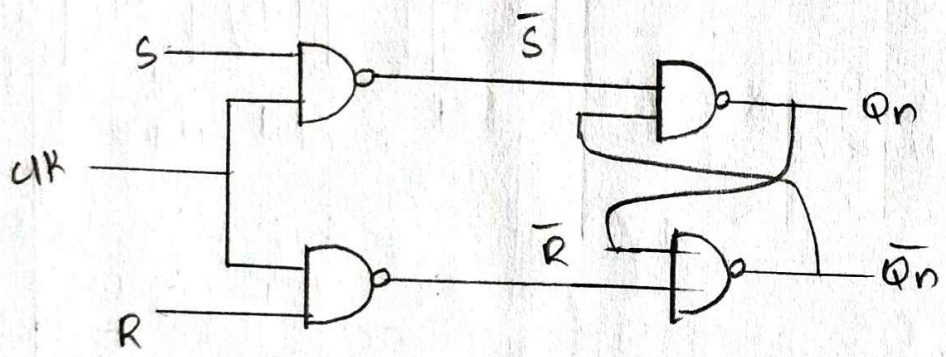



## SR flip flop:




a) logic symbol

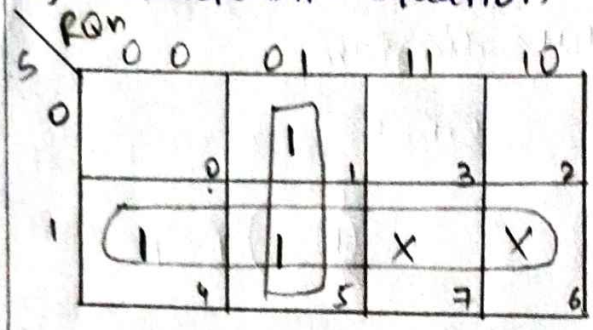
b) logic diagram.




c) Truth Table

| clk | S | R | Qn | Qn+1 | State |
|---|---|---|---|---|---|
| ↑ | 0 | 0 | 0 | 0 | No change |
| ↑ | 0 | 0 | 1 | 1 | |
| ↑ | 0 | 1 | 0 | 0 | Reset |
| ↑ | 0 | 1 | 1 | 0 | |
| ↑ | 1 | 0 | 0 | 1 | Set |
| ↑ | 1 | 0 | 1 | 1 | |
| ↑ | 1 | 1 | 0 | x | Invalid |
| ↑ | 1 | 1 | 1 | x | |
| ↓ | X | X | 0 | 0 | NO change |
| ↓ | X | X | 1 | 1 | |

d) characteristic equation

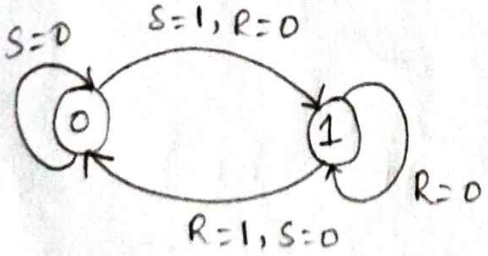



$$Q_{n+1} = S + \bar{R}Q_n$$

e) excitation table

| $Q_n$ | $Q_{n+1}$ | S | R | S | R |
|---|---|---|---|---|---|
| 0 | 0 | 0<br>0 | 0<br>1 | 0 | X |
| 0 | 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0<br>1 | 0<br>0 | X | 0 |

f) State diagram

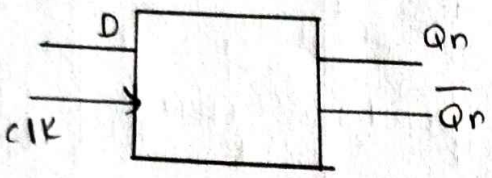



f) excitation table.

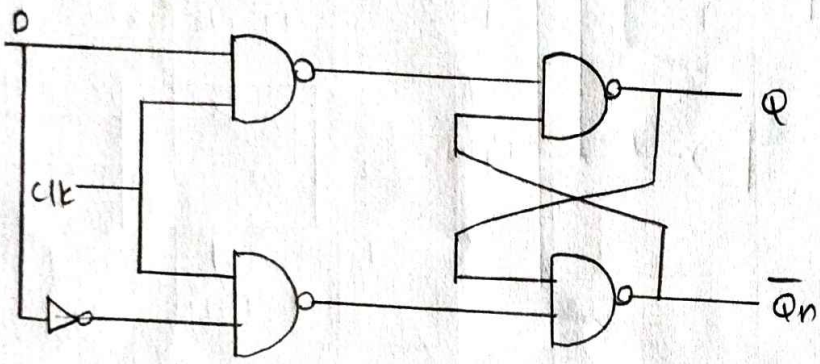
# D-flipflop (Data flip flop)

- It has only 1 input D.

a) Logic symbol




b) Logic diagram



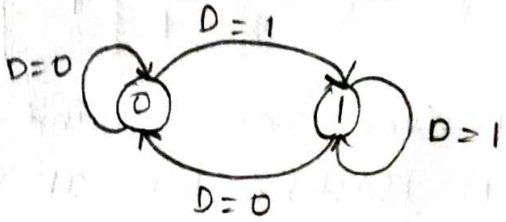

c) Truth Table:

| clk | D | Qn | Qn+1 | state |
|---|---|---|---|---|
| ↑ | 0 | 0 | 0 | Reset |
| ↑ | 0 | 1 | 0 | |
| ↑ | 1 | 0 | 1 | Set |
| ↑ | 1 | 1 | 1 | |
| ↓ 0 | X | 0 | 0 | No |
| ↓ 0 | X | 1 | 1 | change |

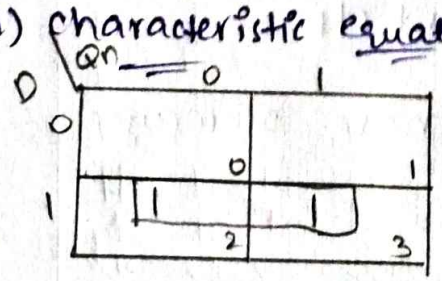
e) Excitation table

| Qn | Qn+1 | D |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

f) State diagram



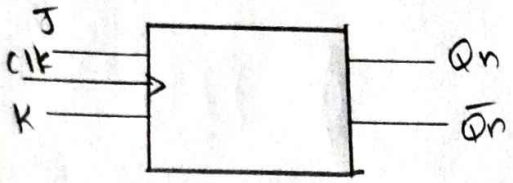

d) Characteristic equation



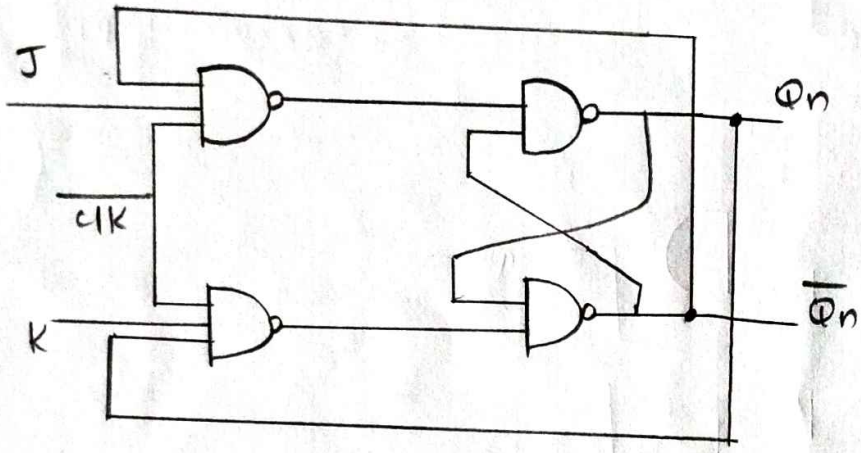

$Q_{n+1} = D$

## J-K flip flop :-

a) Logic symbol




b) Logic Diagram




c) Truth Table :-

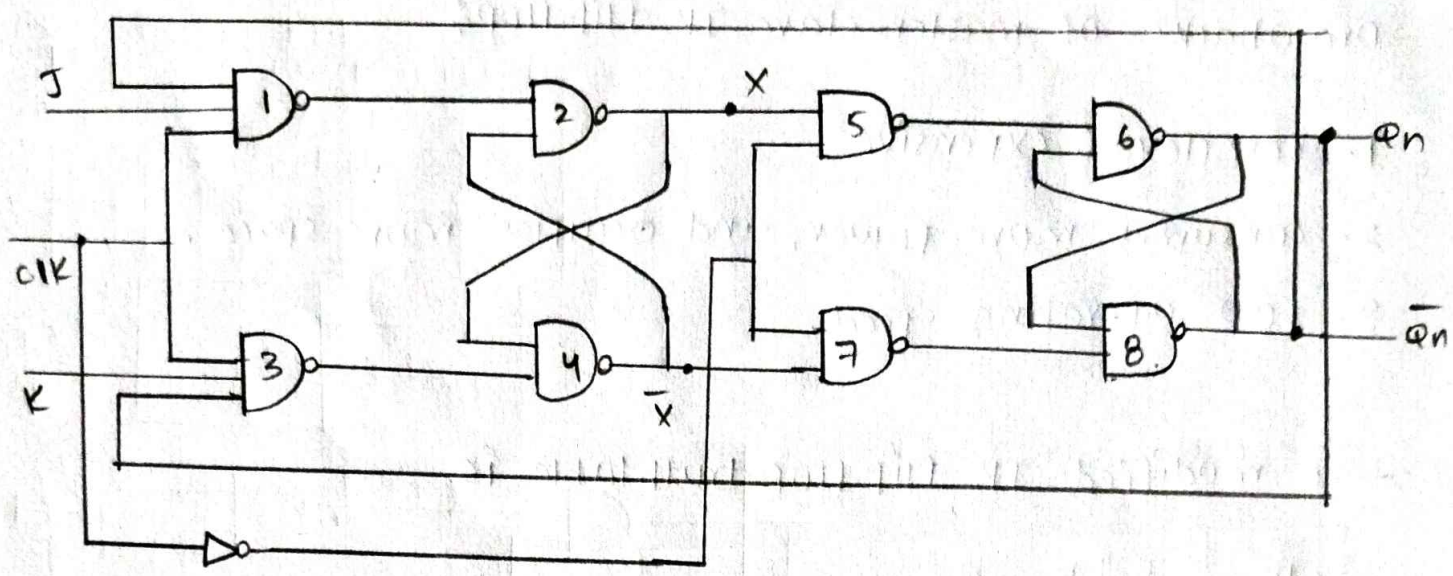
| clk | J | K | Qn | Qn+1 | State |
|---|---|---|---|---|---|
| ↑ | 0 | 0 | 0 | 0 | N.C |
| ↑ | 0 | 0 | 1 | 1 | |
| ↑ | 0 | 1 | 0 | 0 | Reset |
| ↑ | 0 | 1 | 1 | 0 | |
| ↑ | 1 | 0 | 0 | 1 | Set |
| ↑ | 1 | 0 | 1 | 1 | |
| ↑ | 1 | 1 | 0 | 1 0 | Toggle |
| ↑ | 1 | 1 | 1 | 0 1 | |
| ↓ | X | X | 0 | 0 | N.C |
| ↓ | X | X | 1 | 1 | |

for J=K=1, case if clk is high o/p is changed from 1 to 0 and 0 to 1 state continuously. This is called Toggle state and this problem is called "Race around condition". To avoid this we have to make clk is slow if clk is low. Next state is equal to present state. so it maintains the constant o/p.

To avoid race around condition, we have 2 ways

1) Master - slave JK flip flop
2) Edge triggered JK flip flop.

⇒ Master-slave JK flip flop:




It consists of 2 JK flip flop's one acting like a master and the other like a slave. The o/p of the master drive the i/p's of second one. so if is a slave circuit. The o/p's of slave circuit connected to a master in a cross-coupled fashion clk i/p is directly connected to the master. But for slave, it is connected to an inverter and then to slave. For 000 to 101 case. It works same as JK flip flop.

J K $Q_n$

→ For JK$Q_n$=110 case, assume clk is high then NAND gate '1' o/p is 1. NAND gate '3' o/p is 0. if inputs of NAND gate '2' is 10, then o/p is 1. i.e X=1, $\bar{X}$=0.

Next X, $\bar{X}$ are i/p's to slave circuit. But In slave circuit clk is low. so it maintains the previous state o/p i.e Q=1, and $\bar{Q}_1$=0. Even though we got different o/p for Q=1, $\bar{Q}_1$=0, because of disabling slave JK flip flop. It always maintains previous state o/p.

→ similarly JK $Q_n$ = 111 case, we got $Q_{n+1}$ = 0, $\overline{Q_{n+1}}$ = 1

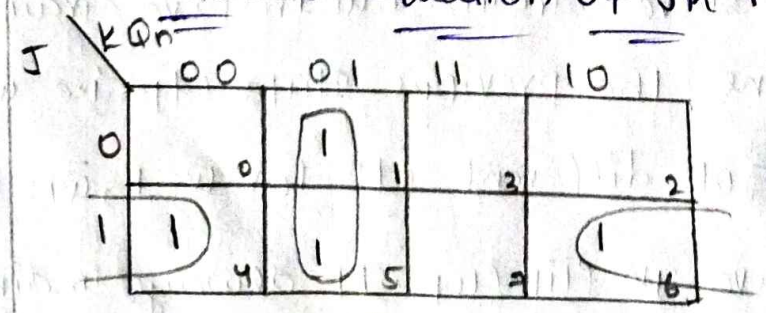
## Drawbacks of master-slave JK flip-flop:

1. complex & expensive
2. consumes more power, and occupies more space.
3. Low operating speed.

— so modified JK flip-flop truthtable is.

| clk | J | K | $Q_n$ | $Q_{n+1}$ | state |
|---|---|---|---|---|---|
| ↑ | 0 | 0 | 0 | 0 | N.C |
| | 0 | 0 | 1 | 1 | |
| ↑ | 0 | 1 | 0 | 0 | Reset |
| | 0 | 1 | 1 | 0 | |
| ↑ | 1 | 0 | 0 | 1 | Set |
| | 1 | 0 | 1 | 1 | |
| ↑ | 1 | 1 | 0 | 1 | Inverse present state |
| | 1 | 1 | 1 | 0 | |
| ↓ | X | X | 0 | 0 | N.C |
| | X | X | 1 | 1 | |

## characteristic equation of JK flip-flop:



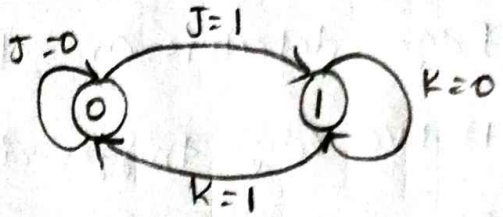

$Q_{n+1} = J\overline{Q_n} + \overline{K}Q_n$

(Ig $\overline{Q_{n+1}} = KQ_n + \overline{J}Q_n$

## Excitation Table of JK flipflop:

| $Q_n$ | $Q_{n+1}$ | J | K | J | K |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | X |
| | | 0 | 1 | | |
| 0 | 1 | 1 | 0 | 1 | X |
| | | 1 | 1 | | |
| 1 | 0 | 0 | 1 | X | 1 |
| | | 1 | 1 | | |
| 1 | 1 | 0 | 0 | X | 0 |
| | | 1 | 0 | | |

## State diagram




## CLOCK SKEW

One of the most common th timing problems in synchronous circuits is clock skew. It happened because of propagation delay of flipflop.

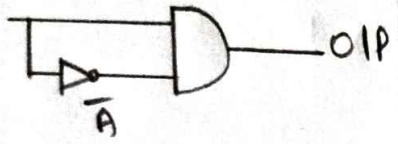
→ The clock signal which is applied simultaneously to all the flipflop's in a synchronous system may undergo varying degrees of delay caused by wiring blw components, and arrive at the clk i/p's of different flipflops at different times. This delay is called clock skew.

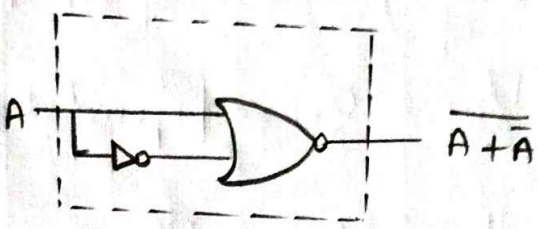
## Edge detector:

Edge detector is used to generate the narrow spikes which are very helpful is avoiding the race-around condition.

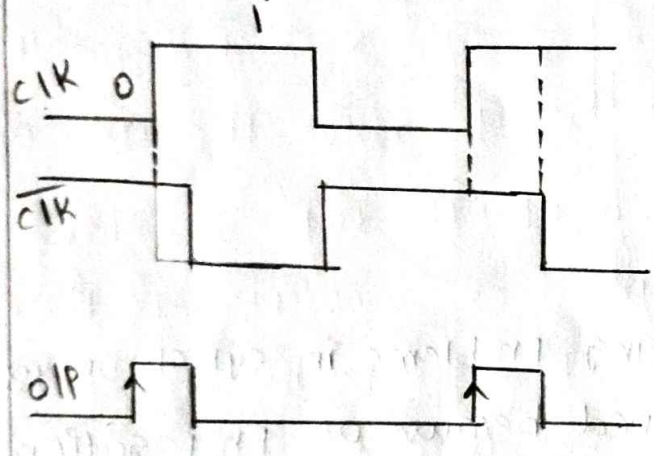
Edge detectors are classified into 2 types.

1) +ve edge trigger circuit.
2) -ve edge trigger circuit.

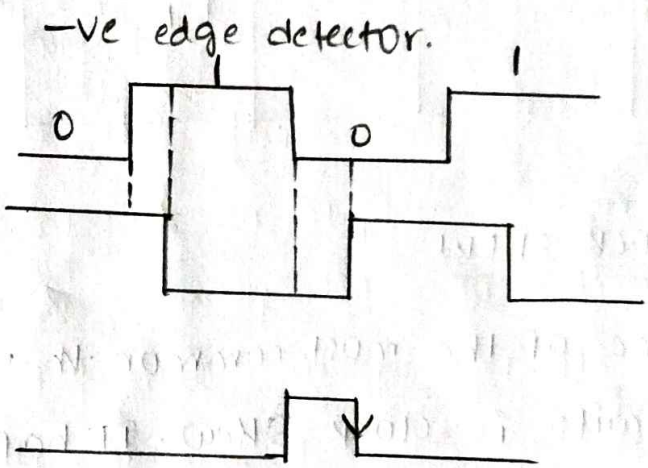



+ve edge detector

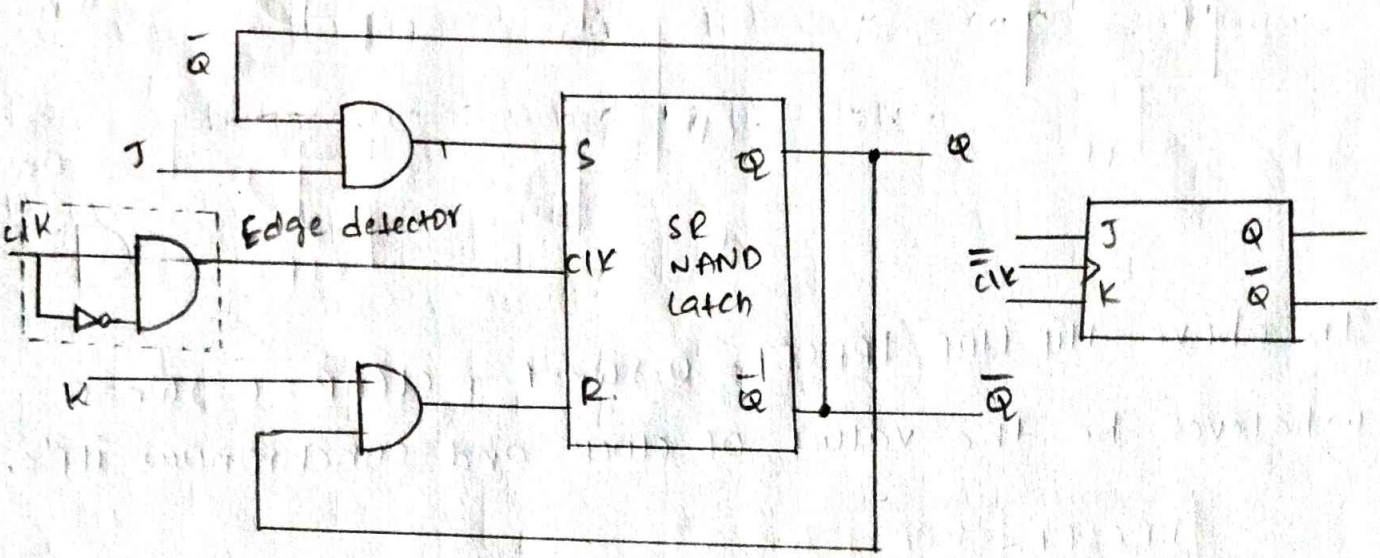



-ve edge detector.




- Because of the propagation delay of not gate, the AND gate doesn't responds very quickly for the given ilp. so due to this delay, a small spikes are presented at the OIP of AND gates. i.e the narrow pulses are presented only for the leading edge of clk pulse similarly the -ve edge detector generate spikes at the trailing edge of clk pulse.
- coming to the operation, if we give the +ve spike (⌐) then only the circuit responds for the given ilp. If we give the -ve voltage (⌙), the circuit is disabled and it maintains the previous state.

- By using this type of clock signals, we can design edge triggered D-flip flop, J-K flip-flop and SR flipflop.




- In this ckt, clock is given to edge detector and then to SR NAND latch. So narrow pulses are generated.
- If J=1, K=1, & Q=0 then Qn+1=1, CLK=1 for short interval and after that it becomes 0.
- If CLK=0, It maintain previous state as o/p. So CLK=1 is continued so for 110 i/p, Qn+1=1, for 111 i/p, Qn+1=0

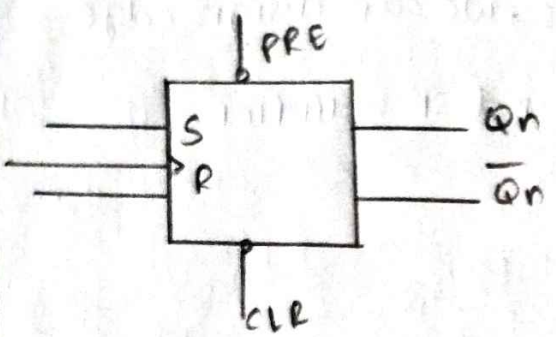
⇒ PRESET, CLEAR:

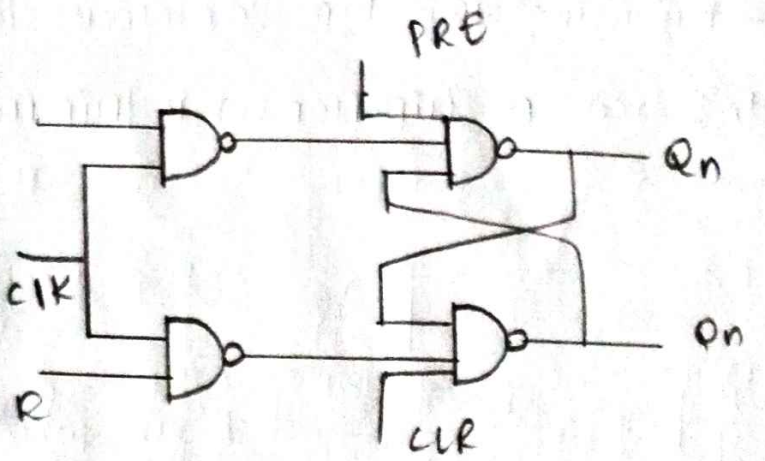
→ They are direct inputs / overriding inputs, also asynchronous i/p's.

→ for the clocked flip-flops S-R, D, J-K & T inputs are called synchronous i/p's, because their effect on the flip flop o/p is synchronized with the clock i/p.

→ These asynchronous i/p's affect the flip-flop o/p independently of the synchronous i/p's and clock i/p.

→ These asynchronous i/p's kept the flip-flop in 'SET 1' or 'RESET 0' state.



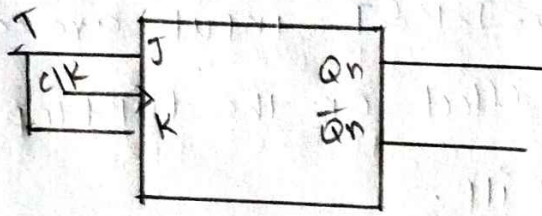

In above flip flop PRESET = 0 ⇒ Qn = 1 , CLEAR = 0 ⇒ Qn = 0 whateves be the values of clock and synchronous i/p's.

| PRESET | CLEAR | Qn |
|---|---|---|
| 0 | 0 | not used |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | flipflop will perform normally. |

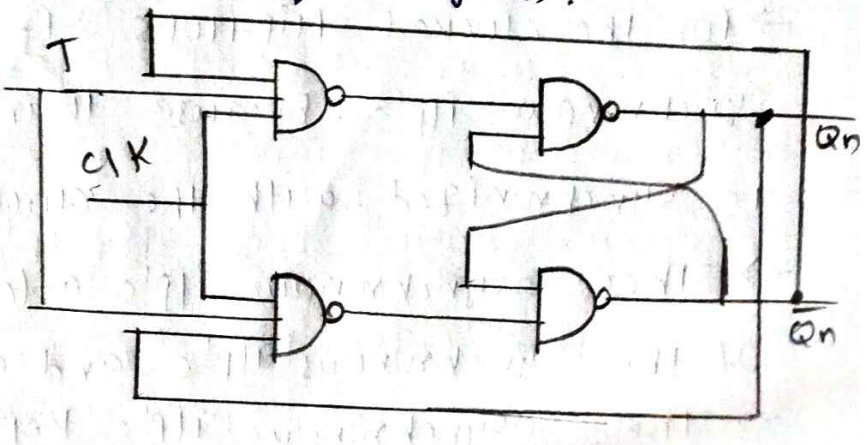
→ T - flip flop (Toggle flipflop) :-

when the J and K i/p's of a JK flipflop are shorted together, a T flip flop is formed.

a) Logic symbol :-




b) logic diagram.

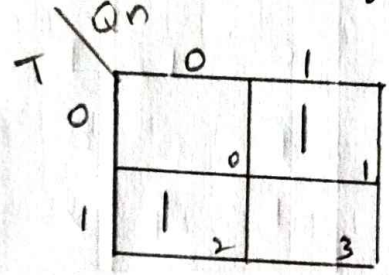



b

c) Truth Table.

| I/p's | | | o/p | state |
|---|---|---|---|---|
| Clk | T | Qn | Qn+1 | |
| ↑ | 0 | 0 | 0 | N.C |
| ↑ | 0 | 1 | 1 | |
| ↑ | 1 | 0 | 1 | Toggle |
| ↑ | 1 | 1 | 0 | |
| ↑ | X | 0 | 0 | N.C |
| ↑ | X | 1 | 1 | |

d) characteristic equation.

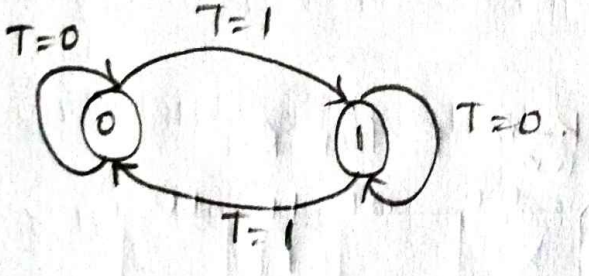



$Q_{n+1} = T\bar{Q}_n + \bar{T}Q_n$

e) Excitation Table.

| P.S | N.S | F/F's |
|---|---|---|
| Qn | Qn+1 | T |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

f) State diagram:-



## Finite state machines.

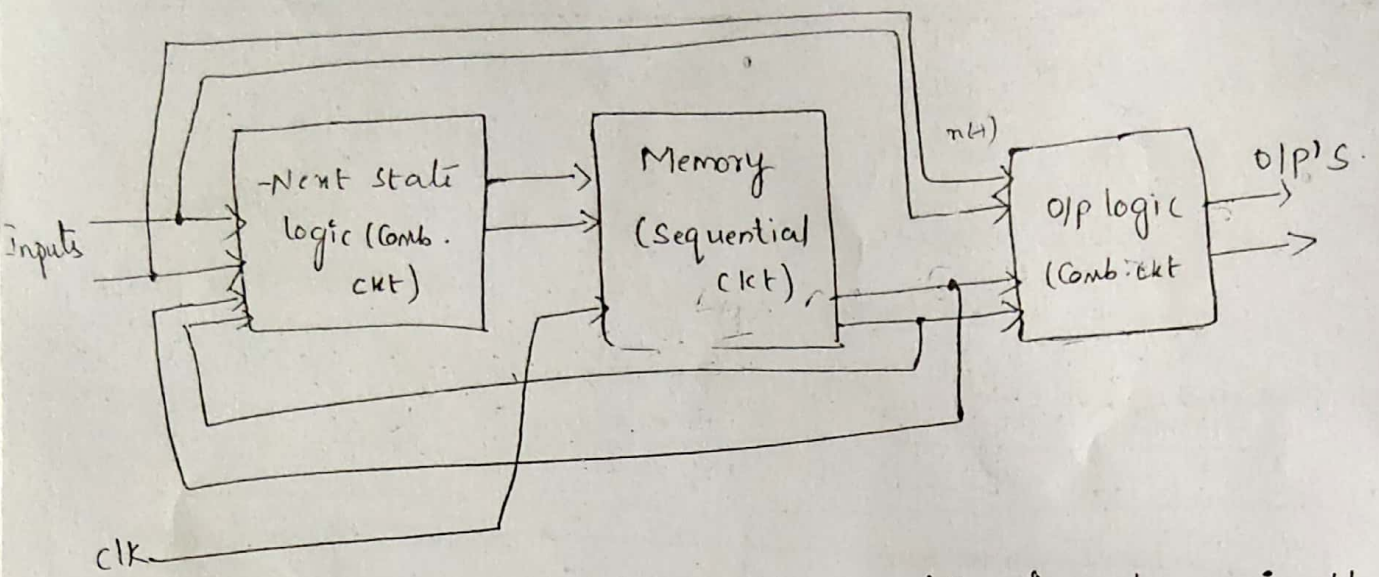
A clocked sequential ckt, in general, is also called a 'FSM'. i.e. it contains finite no. of states.

FSM's are broadly classified into 2 categories called

(a) Mealy - type (b) Moore - type..

Scanned by CamScanner

1. Melay machine.




The general structure of Melay machine is shown in the fig. It consists of a Seq. ckt in b/w & Combinational n/w's. The i/p's are applied to the Next state Comb. ckt along with the present i/p's.

The O/P of Next state logic is given as i/p to the memory elements. This memory elements stores this Next state value and the next state acts as P.S. for the Next i/p's. Here, O/p Combinational logic-ckt also depends on i/p's and p.s.

$\therefore$ Next state = $F$ (P.S, I/P's).

O/P = $G$ (P.S, I/P's). (∵ F & G are 2 different fn's).

Depending on the i/p, o/p is changed but not on clock.

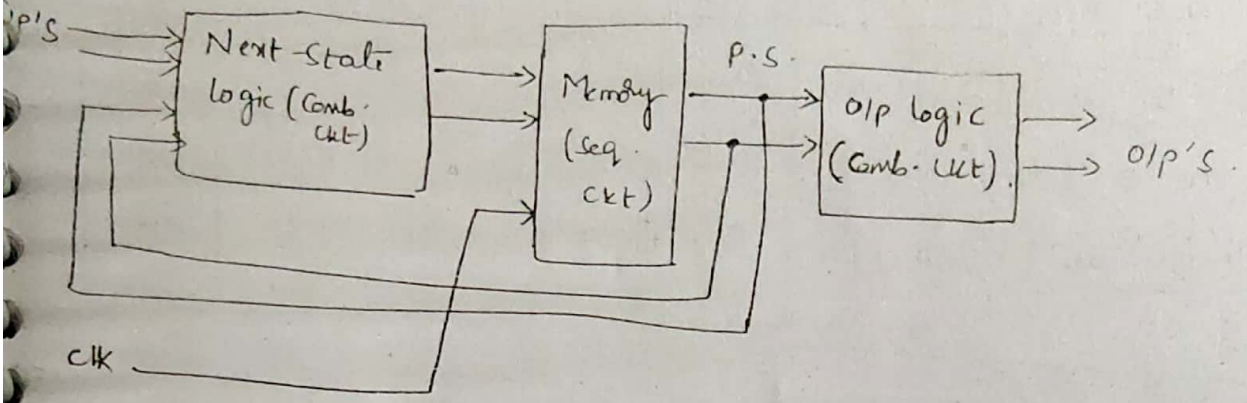
→ Moore Machine

$N.S = F(P.S, I/P's)$.

$O/P = G(P.S)$.

Here, the o/p depends only on p.s. but not on i/p's. So depending on the clock pulse only, the o/p's can be changed

Scanned by CamScanner




→ Differences b/w Mealy & Moore machines

| Mealy | Moore |
|---|---|
| 1. o/p is a fn of both p.s. and i/p's. | 1. The o/p is a fn of p.s only |
| 2. o/p depends only on clk but not on ~~clock~~ i/p | 2. o/p depends only on clk, but not on i/p |
| 3. These are not preferred to operate in a noisy environment | 3. They are used to operate in a noisy environment |
| 4. It requires less no. of states in the corresponding state diagram. so cost is low | 4. P.S. produces more no. of gates. so it requires more no. of states leads to high cost. |

→ A Sequential ckt with 2 D f/f's A and B, two i/p's x, y and one o/p z, is specified by the following next-state and o/p eqns.

$$A(t+1) = \bar{x}y + xA(t), \quad B(t+1) = \bar{x}B(t) + xA(t)$$

$$z = B(t)$$

(a) Draw the logic diagram of the ckt

(b) List the state table for the seq. ckt

(c) Draw the corresponding state diagram.

Scanned by CamScanner

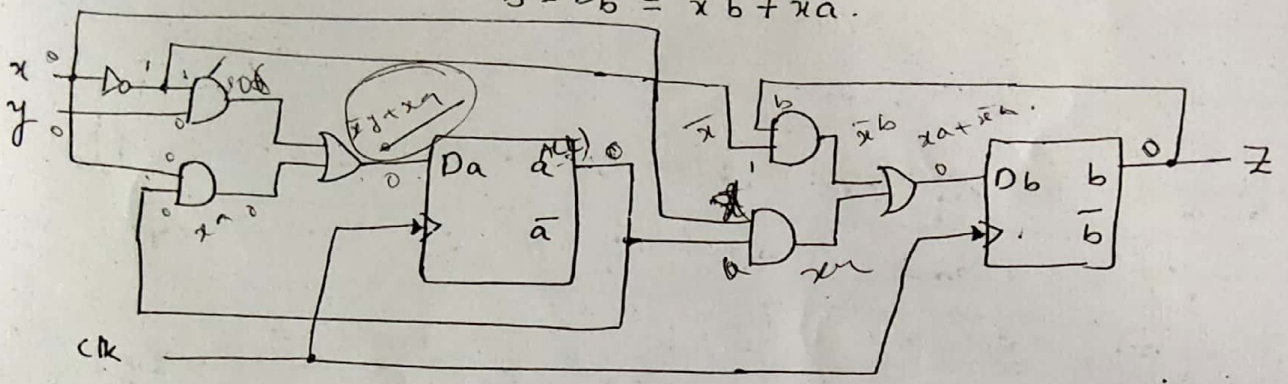
Sol: Let N.S. $A(t+1) = A$, $B(t+1) = B$, P.S. $A(t) = a$, $B(t) = b$.

Then the N.S. eqns are modified as follows.

For the D flip flop, next state is same as the P.S.

The eqns are: $A = Da = \bar{x}y + xa$.

$B = Db = \bar{x}b + xa$.




Logic diagram.

From the given data, we have 2 i/p's x, y & 2 P.S. i/p's a and b. So we have totally 4 i/p's.

4 i/p means 16 Combinations (0 to 15) are there.

| Min term | P.S. a b | I/P's x y | N.S. A B | O/P z = b |
|---|---|---|---|---|
| 0 | 0 0 | 0 0 | 0 0 | 0 |
| 1 | 0 0 | 0 1 | 1 0 | 0 |
| 2 | 0 0 | 1 0 | 0 0 | 0 |
| 3 | 0 0 | 1 1 | 0 0 | 0 |
| 4 | 0 1 | 0 0 | 0 1 | 1 |
| 5 | 0 1 | 0 1 | 1 1 | 1 |
| 6 | 0 1 | 1 0 | 0 0 | 1 |
| 7 | 0 1 | 1 1 | 0 0 | 1 |
| 8 | 1 0 | 0 0 | 0 0 | 0 |
| 9 | 1 0 | 0 1 | 1 0 | 0 |
| 10 | 1 0 | 1 0 | 1 1 | 0 |
| 11 | 1 0 | 1 1 | 1 1 | 0 |
| 12 | 1 1 | 0 0 | 0 1 | 1 |
| 13 | 1 1 | 0 1 | 1 1 | 1 |
| 14 | 1 1 | 1 0 | 1 1 | 1 |
| 15 | 1 1 | 1 1 | 1 1 | 1 |

Truth Table.

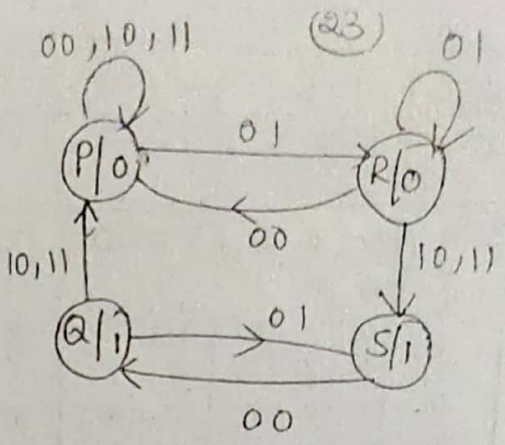
| P.S. ab | N.S. AB xy=00 | xy=01 | xy=10 | xy=11 | O/P z |
|---|---|---|---|---|---|
| 0 0 | 00 | 10 | 00 | 00 | 0 |
| 0 1 | 01 | 11 | 00 | 00 | 1 |
| 1 0 | 00 | 10 | 11 | 11 | 0 |
| 1 1 | 01 | 11 | 11 | 11 | 1 |

Binary transition table.

Scanned by CamScanner

| P.S. | N.S. $xy=00$ | $xy=01$ | $xy=10$ | $xy=11$ | O/P z |
|---|---|---|---|---|---|
| P | P | R | P | P | 0 |
| Q | Q | S | P | P | 1 |
| R | P | R | S | S | 0 |
| S | Q | S | S | S | 1 |

Coded state table

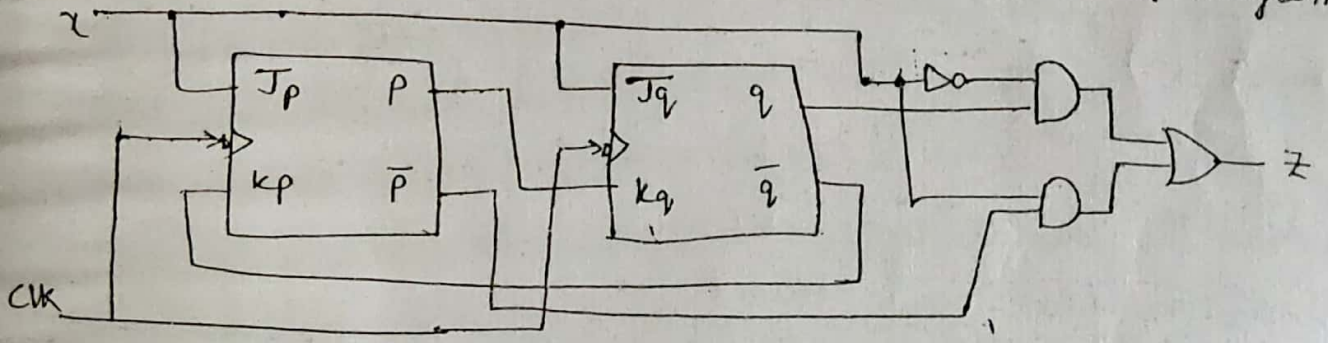



State diagram

→ Analyse the seq. ckt that has 2 JK flf's (P and q), a single i/p x and a single o/p z. Its o/p and i/p eqns are governed by:

$J_P = x$, $K_P = \bar{q}$, $J_q = x$, $K_q = P$, $Z = x\bar{P} + \bar{x}q$.

Sol: Draw the logic ckt & plot the state table & diagram.




Sequential ckt.

Ac. to the ch. eqns of JK flf, the Next state eqns are given by: $P = J_P\bar{P} + \bar{K}_P P$. $(\because Q = J\bar{q} + \bar{K}q)$.

$Q = J_q\bar{q} + \bar{K}_q q$.

$\Rightarrow P = x\bar{P} + qP$ and $Q = x\bar{q} + \bar{P}q$.

From this, p.s. are: P, q and I/p is x.

∴ for 3-bit i/p, 8 combinations are there.

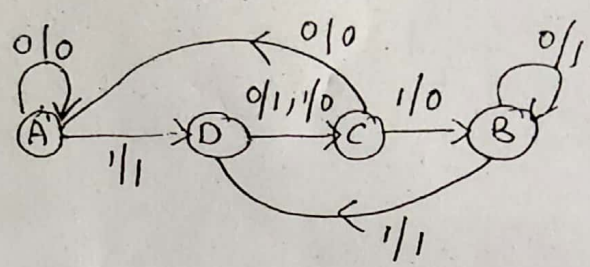
Scanned by CamScanner

| Min tim | P.S P | q | x | N.S P | Q | o/p z |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 1 | 1 |
| 2 | 0 | 1 | 0 | 0 | 1 | 1 |
| 3 | 0 | 1 | 1 | 1 | 1 | 1 |
| 4 | 1 | 0 | 0 | 0 | 0 | 0 |
| 5 | 1 | 0 | 1 | 0 | 1 | 0 |
| 6 | 1 | 1 | 0 | 1 | 0 | 1 |
| 7 | 1 | 1 | 1 | 1 | 0 | 0 |

Binary transition table.

| P.S. | N.S. X=0 | N.S. X=1 | o/p z X=0 | o/p z X=1 |
|---|---|---|---|---|
| A | A | D | 0 | 1 |
| B | B | D | 1 | 1 |
| C | A | B | 0 | 0 |
| D | C | C | 1 | 0 |

coded state table.




State diagram.

—> A clocked seq. ckt with single i/p x and single o/p z is described by the following T f/f i/p eqns and o/p eqns of z. $T_a = \bar{a}b + \bar{x}b$, $T_b = \bar{x}a + \bar{x}\bar{b} + x\bar{a}b$.

$$z = x(\bar{a}+b) = x\bar{a} + xb.$$

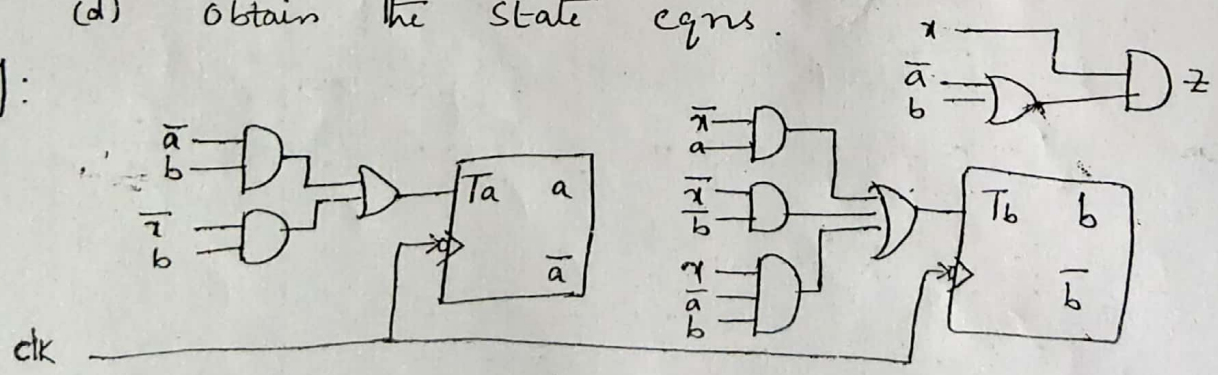
(a) Draw the schematic logic ckt

(b) Obtain the state table

(c) Obtain the state diagram

(d) Obtain the state eqns.

Sol:




from the ch. eqns of T f/f,

$$A = T_a\bar{a} + \bar{T}_a a = (\bar{a}b + \bar{x}b)\bar{a} + (\overline{\bar{a}b + \bar{x}b})\cdot a.$$

$$\therefore A = \bar{a}b + \bar{x}b\bar{a} + (\overline{\bar{a}b}\cdot\overline{\bar{x}b})a = \bar{a}b + \bar{x}b\bar{a} + (a+\bar{b})(x+\bar{b})a$$

$$= \bar{a}b + \bar{x}b\bar{a} + ax + a\bar{b} + \bar{b}xa + \bar{b}a.$$

Scanned by CamScanner

$A = \bar{a}b + a\bar{b} + x a(1+\bar{b}) + \bar{x} b \bar{a} = \bar{a}b + a\bar{b} + xa + \bar{x} b \bar{a}$.

$B = \overline{\bar{b}} \cdot b + \bar{b}\bar{b} = \overline{(\bar{x}a + \bar{x}\bar{b} + x\bar{a}b)}\, b + (\bar{x}a + \bar{x}\bar{b} + x\bar{a}b)\bar{b}$

$= \left(\overline{\bar{x}a} \cdot \overline{\bar{x}\bar{b}} \cdot \overline{x\bar{a}b}\right) b + \bar{x}a\bar{b} + \bar{x}\bar{b} + \overline{\bar{x}}\,\bar{a}b\bar{b}$

$= (x+\bar{a})(x+b)(\bar{x}+a+\bar{b})\, b + \bar{x}\bar{b}(1+a)$

$= (x + xb + \bar{a}x + \bar{a}b)(\bar{x} + a) + \bar{x}\bar{b}$

$= xa + xab + \bar{a}b\bar{x} + \bar{x}\bar{b} = xa(1+b) + \bar{x}\bar{a}b + \bar{x}\bar{b}$

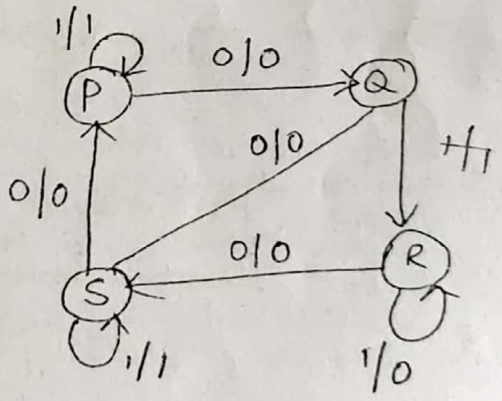
$= xa + \bar{x}\bar{a}b + \bar{x}\bar{b}$.

$Z = x\bar{a} + xb$.

| Min term | P·S a | P·S b | i/p X | N·S A | N·S B | o/p Z |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 2 | 0 | 1 | 0 | 1 | 0 | 0 |
| 3 | 0 | 1 | 1 | 1 | 1 | 0 |
| 4 | 1 | 0 | 0 | 0 | 0 | 1 |
| 5 | 1 | 0 | 1 | 0 | 0 | 1 |
| 6 | 1 | 1 | 0 | 1 | 1 | 0 |
| 7 | 1 | 1 | 1 | 0 | 0 | 1 |

Binary transition table.

| P·S· | N·S & o/p x=0 | x=1 |
|---|---|---|
| P | Q, 0 | P, 1 |
| Q | S, 0 | R, 1 |
| R | S, 0 | R, 0 |
| S | P, 0 | S, 1 |




state diagram.

→ Capabilities and limitations of finite state machines.

1. Periodic sequence of finite states:

With n-state machine, we can generate a periodic sequence of n or <n states.

For ex: a 6-state machine can generate the states from 0 to 5 (i.e. 0,1,2,3,4,5) 0,1,2,3,4,5,0,1...

Scanned by CamScanner

2. No Infinite Sequence:

Assume an infinite sequence whose O/P = 1 for the i/p values of $\frac{P(P+1)}{2}$ (P -> i/p). i.e. for 1, 3, 6 ...

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| I/P | x | x | x | x | x | x | x | x | x | x | x |
| O/P | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 - - |

Such an infinite sequence can't be produced by the FSM.

3. Limited Memory: It has limited memory so it can't produce certain O/P's. Of on the binary multiplication ($N \times N = 2N$), a large no. is produced. So it is not possible to store the result becz of finite memory.

Steps in Analysing of a sequential ckt.

1. Understand the specification.
2. Draw the logic diagram.
3. Draw Binary transition table
4. prepare state table
5. Draw the state diagram.

Synthesis of finite state machines

1. Understand the Specification.
2. state reduction (Reducing the states from state table).
3. State assignment.
4. prepare binary transition table.
5. Draw the logic circuit.

Scanned by CamScanner

## State reduction.

If 2 or more states are having same Next states and o/p's then one is redundant among them. and the redundant states are replaced with the existed state.

Ex:

| P.S. | N.S, O/P(Z) x=0 | x=1 |
|---|---|---|
| a | b, 0 | f, 1 |
| b | e, 0 | c, 1 |
| c | a, 1 | d, 1 |
| d | b, 0 | f, 1 |
| e | a, 0 | f, 0 |
| f | a, 1 | d, 1 |

state table

| P.S. | N.S, O/P x=0 | x=1 |
|---|---|---|
| a | b, 0 | c, 1 |
| b | e, 0 | c, 1 |
| c | a, 1 | a, 1 |
| e | a, 0 | c, 0 |

Redundant state table

a = d & c = f. so d and f are redundant.

and substitute d by a and f by c in the redundant state table.

The partition technique is used in the state reduction. which will be explained later.

## State assignment

State assignment can be done in the following 3 ways.

1. In a straight away manner (i.e. 00, 01, 10, 11 :-).
2. Jumbling manner (01, 11, 10, 00).
3. Using Gray code representation (00, 01, 11, 10).

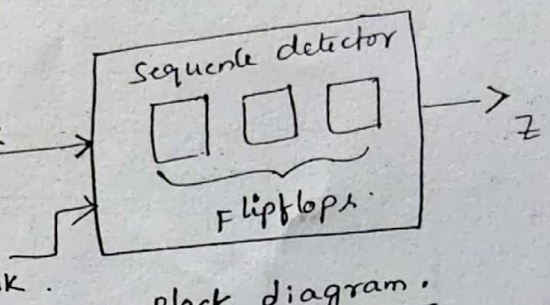
Generally, Gray code representation is used.

Scanned by CamScanner

## Synthesising of Sequence detector.

Ex: Design a sequence detector that recognises the occurence of a 3-bit sequence 101.

Sol: The no. of f/f's are assumed to be 3. Because we have to detect the 3-bit sequence 101.




Block diagram.

To prepare the state diagram, the following conditions should be taken into account:

1. If the first bit is 1 then the state should be changed from a to b. (∵ a → Initial state). If the first bit is zero, it should be in the same state a.
2. If the second bit is zero then only the state should be changed from b to c.
3. If the 3rd bit is 1 then the state should be changed from c to a. but for 1 i/p, the o/p should be 1. Because whenever the o/p sequence is detected, the o/p should be = 0. 0.w. o/p = 0.

Ex1: I/p Seq (X): 1 0 0 1 0 1 0 0 1 0 1 0 0 1 1 1 1 . .

O/p Seq (Z): 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 .

(a) Non-overlapping Case

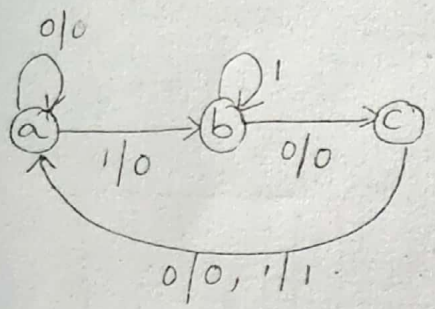
Ex: 2 I/p Seq (x): 0 1 0 1 0 1 0 0 0

O/p Seq (Z): 0 0 0 1 0 1 0 0 0 .

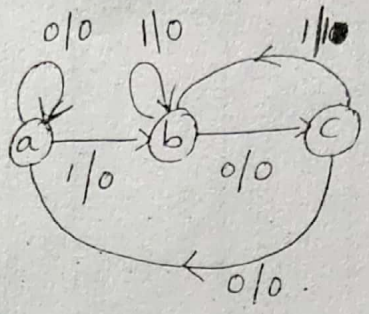
(b) Overlapping Case.

Scanned by CamScanner

In overlapping seq. (1011), middle '1' is common to both the sequences.




Non-overlapping case.




Overlapping case.

State diagram of '101' Sequence detector.

Synthesis of Sequence detector using T flip-flops.

| P.S | N.S, O/P: x=0 | x=1 |
|---|---|---|
| a(00) | a,0 | b,0 |
| b(01) | c,0 | b,0 |
| c(11) | a,0 | a,1 |

Reduced state diagram.

| P.S | | i/p | N.S | | o/p |
|---|---|---|---|---|---|
| p | q | x | P | Q | Z |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | x | x | x |
| 1 | 0 | 1 | x | x | x |
| 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 1 |

Assign codes: 00 to a, 01 to b and 11 to c. i.e. Gray code representation of binary codes is used.

As the binary code contains 2 bits, the p.s are: p and Q. we want to design it using T f/f's. so draw the excitation table and according to that prepare binary transition table including Excitation inputs.

By using k-map method, findout the values of i/p's: $T_p$, $T_q$ and o/p Z.

10 state → Don't care state.

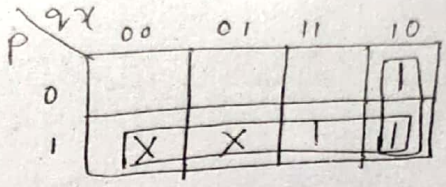
Excitation table.

| q | Q | $T_q$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

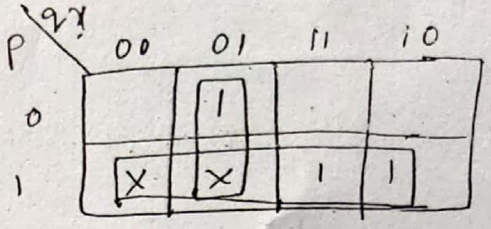
Scanned by CamScanner

| P.S. | I/p | N.S. | | Excitation i/p's | | O/p |
|---|---|---|---|---|---|---|
| P | q x | P | Q | $T_P$ | $T_q$ | Z |
| 0 | 0 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 1 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 1 1 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 0 | x | x | x | x | x |
| 1 | 0 1 | x | x | x | x | x |
| 1 | 1 0 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 1 | 0 | 0 | 1 | 1 | 1 |

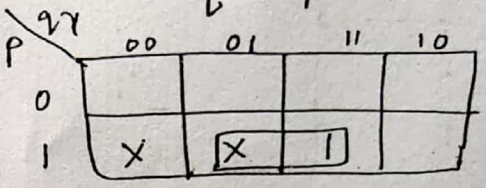
Binary transition table including Excitation i/p's.



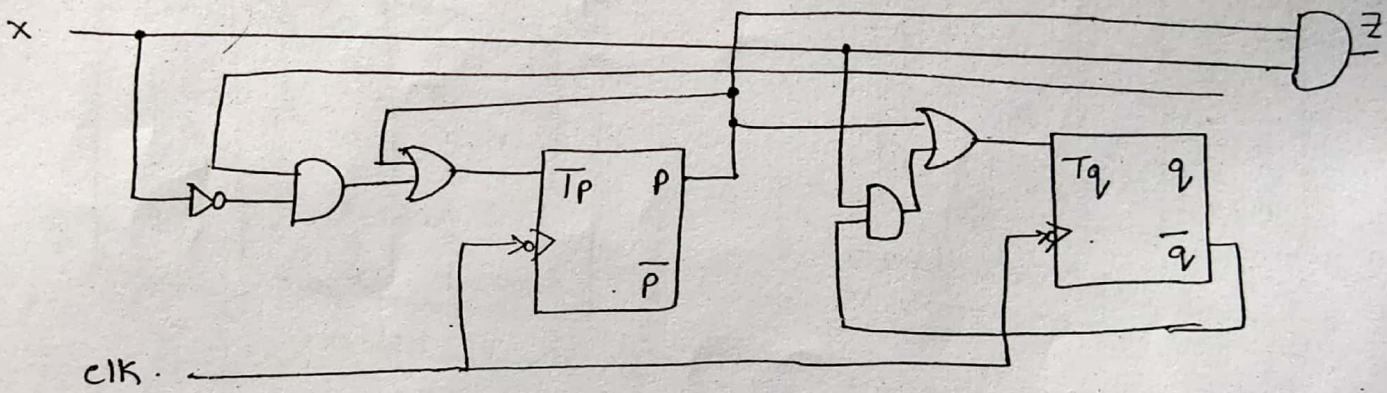

$T_P = P + q\bar{x}$




$T_q = P + \bar{q}x$.




$Z = Px$.




Sequential logic ckt.

Synthesis using JK flf's.

| P.S. | | I/P | N.S. | | Excitation I/p's | | | | Z |
|---|---|---|---|---|---|---|---|---|---|
| P | q | x | P | Q | $J_P$ | $k_P$ | $J_q$ | $k_q$ | |
| 0 | 0 | 0 | 0 | 0 | 0 | x | 0 | x | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | x | 1 | x | 0 |
| 0 | 1 | 0 | 1 | 1 | 1 | x | x | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 | x | x | 0 | 0 |
| 1 | 0 | 0 | x | x | x | x | x | x | x |
| 1 | 0 | 1 | x | x | x | x | x | x | x |
| 1 | 1 | 0 | 0 | 0 | x | 1 | x | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | x | 1 | x | 1 | 1 |

| $q_t$ | Q | J | k |
|---|---|---|---|
| 0 | 0 | 0 | x |
| 0 | 1 | 1 | x |
| 1 | 0 | x | 1 |
| 1 | 1 | x | 0 |

Excitation table.

Scanned by CamScanner

(27)

| p \ qx | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | | | | 1 |
| 1 | x | x | x | x |

$J_p = q\bar{x}$

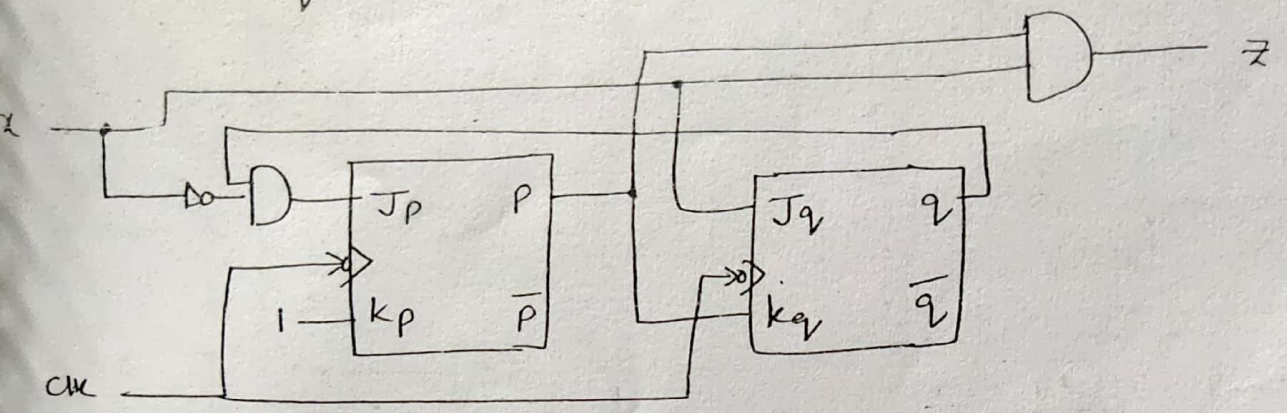
| p \ qx | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | x | x | | |
| 1 | x | x | 1 | 1 |

$K_q = p$

| p \ qx | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | x | x | 1 | 0 |

$z = px$.

(or) $J_q = x$, $K_p = 1$




Seq. logic ckt with JK flf's for N.O. Calc.

Scanned by CamScanner

## Shift registers:-

Register: It is a group of flipflops connected in a particular way to store a group of bits.

→ If a register contain N flipflops, it will store N bits since registers are made of flipflops and because of the fact that flipflops operate under the control of a clock, they fall under the category of synchronous sequential circuits.

→ Register can be constructed with SR, T, D, JK or any other type of flipflops.

→ The information loading into and reading from the flipflop can be done either in series fashion or parallel fashion

serial → bit by bit transmission

parallel → entire data transmission.

shift register:- A register which provides a facility for the data to be shifted towards right or left is called a shift register. Based on the way we write data in to register and also the way the data is read from it, the registers are classified in various ways

→ Even though $D_2$, $D_3$ & $D_4$ flipflops are activated due to -ve edge triggered clk signal, the o/p's are not available at the $D_2$, $D_3$ & $D_4$ flipflops. During the 2nd clk pulse, $D_2$ generates the next state i.e., (x=1 reached to B). At this time, the second o/p of x reached as next state of A.

a, b, c, d → present states

A, B, C, D → Next state

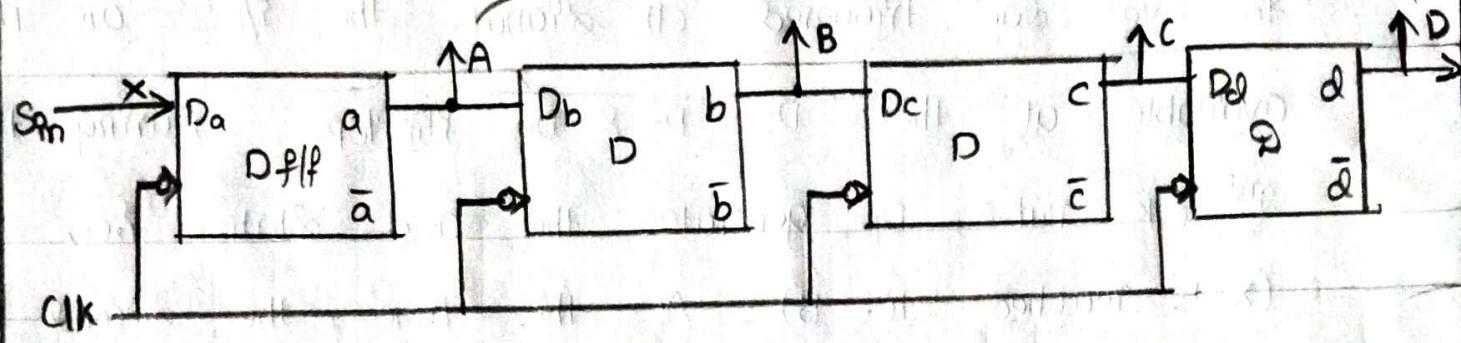
Time = (N+N-1)T

= (2N-1)T

→ The first bit entered at x came out on z after the fourth clk pulse, because the register has 4 flipflops

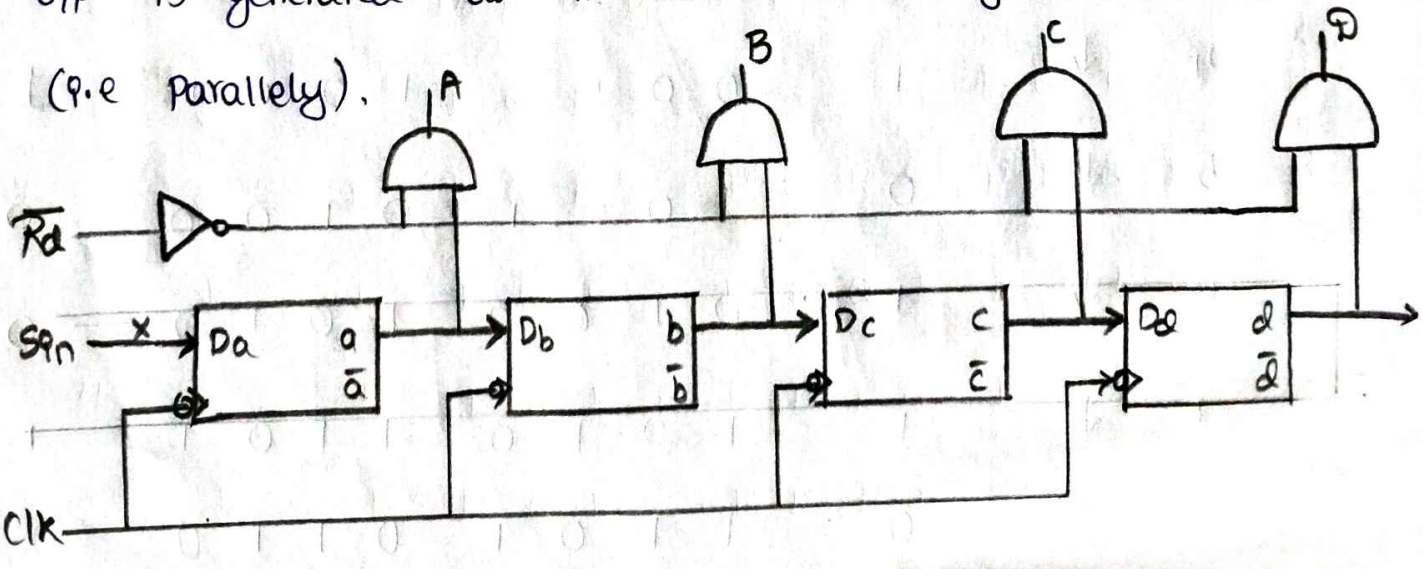
| S.No | Clk | Serial o/p | Present state a b c d | Next state A B C D | Serial o/p x=D |
|---|---|---|---|---|---|
| 1 | ↓ | 1 | - - - - | 1 - - - | - |
| 2 | ↓ | 0 | 1 - - - | 0 1 - - | - |
| 3 | ↓ | 0 | 0 1 - - | 0 0 1 - | - |
| 4 | ↓ | 1 | 0 0 1 - | 1 0 0 1 | 1 |
| 5 | ↓ | 0 | 1 0 0 1 | 0 1 0 0 | 0 |
| 6 | ↓ | 1 | 0 1 0 0 | 1 0 1 0 | 0 |
| 7 | ↓ | 1 | 1 0 1 0 | 1 1 0 1 | 1 |
| 8 | ↓ | 0 | 1 1 0 1 | 0 1 1 0 | 0 |

→ Serial In Parallel out shift register




→ In SIPO shift register, i/p is entered serially but o/p is taken in parallely as shown in figure. The operation performed by a SIPO register is basically a memory read function; so we introduce read ($\overline{Rd}$) control signal so that the ckt is modified as show in fig.

→ If $\overline{Rd}=1$, what ever may be the Next state, the Parallel $dP=0$. So to get Parallel o/p, the $\overline{Rd}=1$ for N clock pulses for $N+1^{th}$ clk pulse, if we issued $\overline{Rd}=0$, o/p is generated at the end of every 0 flip flop (i.e Parallely).



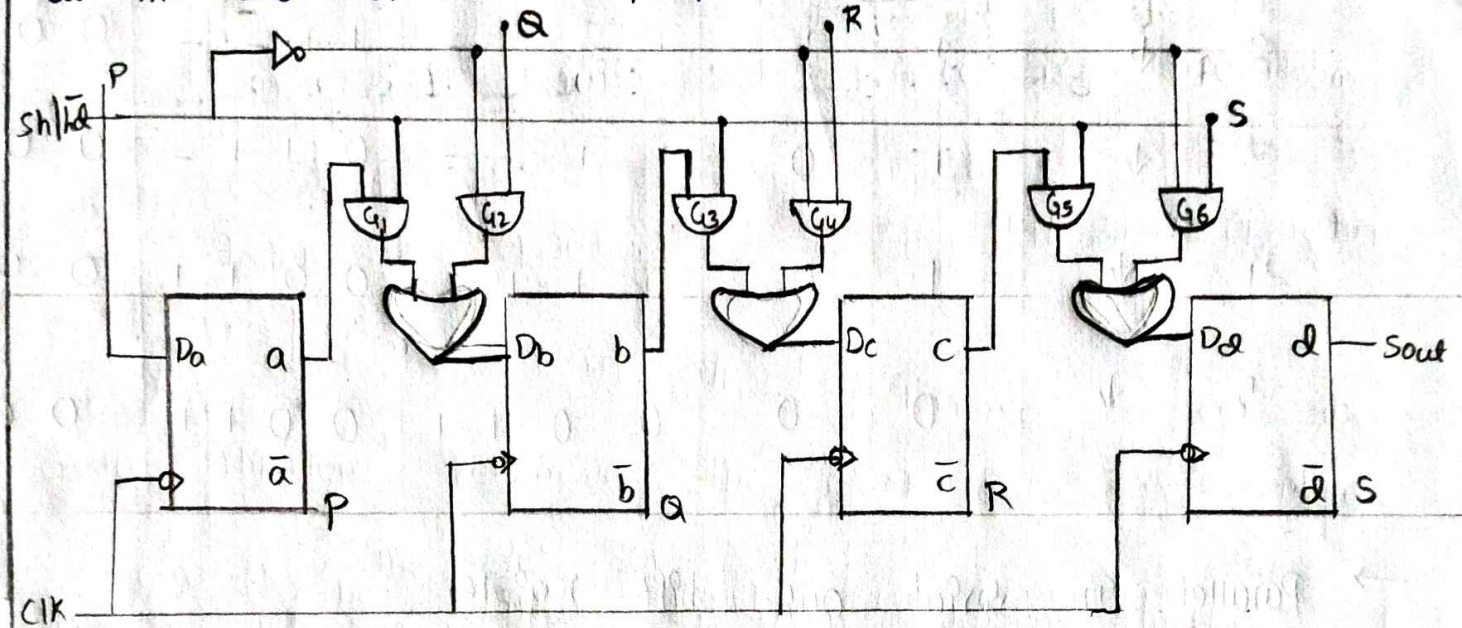
| S.No | Clk | $\overline{Rd}$ | Serial o/p | Present State | | | | Next State | | | | Parallel o/p | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | X | a | b | c | d | A | B | C | D | A | B | C | D |
| 1 | ↓ | 1 | 1 | - | - | - | - | 1 | - | - | - | 0 | 0 | 0 | 0 |
| 2 | ↓ | 1 | 1 | 1 | - | - | - | 1 | 1 | - | - | 0 | 0 | 0 | 0 |
| 3 | ↓ | 1 | 0 | 1 | 1 | - | - | 0 | 1 | 1 | - | 0 | 0 | 0 | 0 |
| 4 | ↓ | 1 | 0 | 0 | 1 | 1 | - | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 5 | ↓ | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |

→ Parallel In Serial Out Shift register

It converts parallel data in to serial form. It depends on control signal $Sh/\overline{Ld}$. If $Sh/\overline{Ld} = 0$ means loading operation is done because $G_1$, $G_3$ & $G_5$ are disabled and $Sh/\overline{Ld}$ i/p to AN gates $G_2$, $G_4$, $G_6$ is through NOT gate. so its value is 1. Hence the i/p's are directly loaded in to D flipflop.

→ If $Sh/\overline{Ld} = 1$ means loading is stopped & shifting operation should be done i.e. $G_1$, $G_3$ & $G_5$ are enabeld. so the o/p's of D flipflops are directly connected to succeeding flipflops.

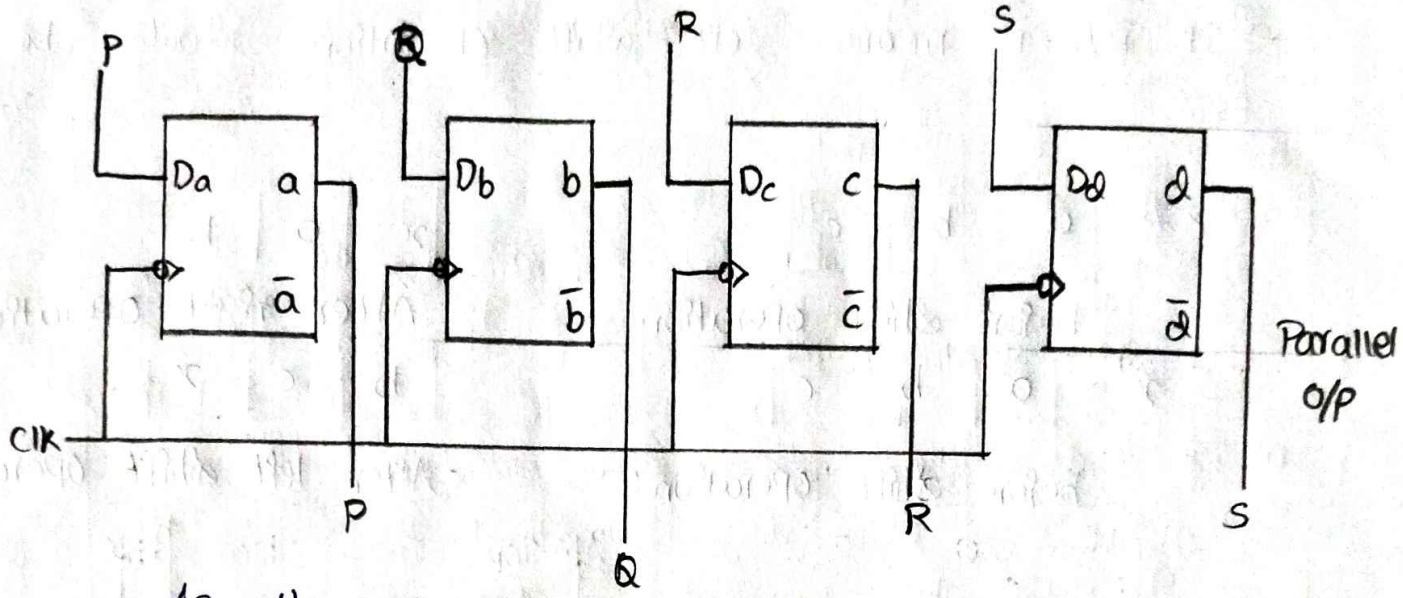
→ Initially $Sh/\overline{ld} = 0$ control signal is applied to load the data into flipflops. After that issue $Sh/\overline{ld} = 1$ signal then shifting operation is done & serial o/p is obtained at the end of 4th flipflop.




| Sh/ld | Control action | Db | Dc | Dd |
|---|---|---|---|---|
| 0 | G1, G3, G5 disabled | Q | R | S |
| 1 | G2, G4, G6 disabled | a | b | c |

| S.No | CLK | Sh/ld | flipflop i/ps | | | | Present state | | | | Next state | | | | Serial o/p |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Da | Db | Dc | Dd | a | b | c | d | A | B | C | D | |
| 1 | ↓ | 0 | P̄ | Q̄ | R̄ | S̄ | - | - | - | - | P | Q | R | S | S |
| 2 | ↓ | 1 | P | Q | R | S | P | Q | R | S | P | P | Q | R | R |
| 3 | ↓ | 1 | P | Q | R | S | P | P | Q | R | P | P | P | Q | Q |
| 4 | ↓ | 1 | P | Q | R | S | P | P | P | Q | P | P | P | P | P |
| 5 | ↓ | 1 | P | Q | R | S | P | P | P | P | P | P | P | P | P |

→ Parallel In Parallel Out shift register




As shown in fig, i/p's are applied to all the data flipflops whenever the clock pulse is applied, we can get the o/p's of each flipflops as PQRS. It doesn't perform the shifting operation between flipflops. so it should not be called as shift register.

→ Parallel transmission → short distance communication

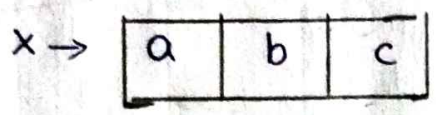
→ serial transmission → long distance communication.

Bi-directional shift register:-

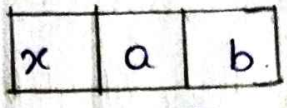
It is having the provision of to shift the data in both the positions i.e towards left or right. The command signal in bi directional shift register is $\bar{R}/L$. Multiplexers are used to select the left or right shift operations.

→ If $\overline{R}/L = 0$ means right shift operation should be done

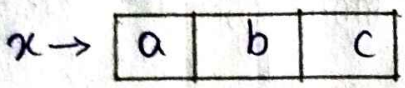
→ If $\overline{R}/L = 1$ means left shift operation should be done.



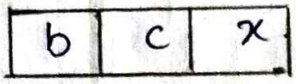

before shift operation



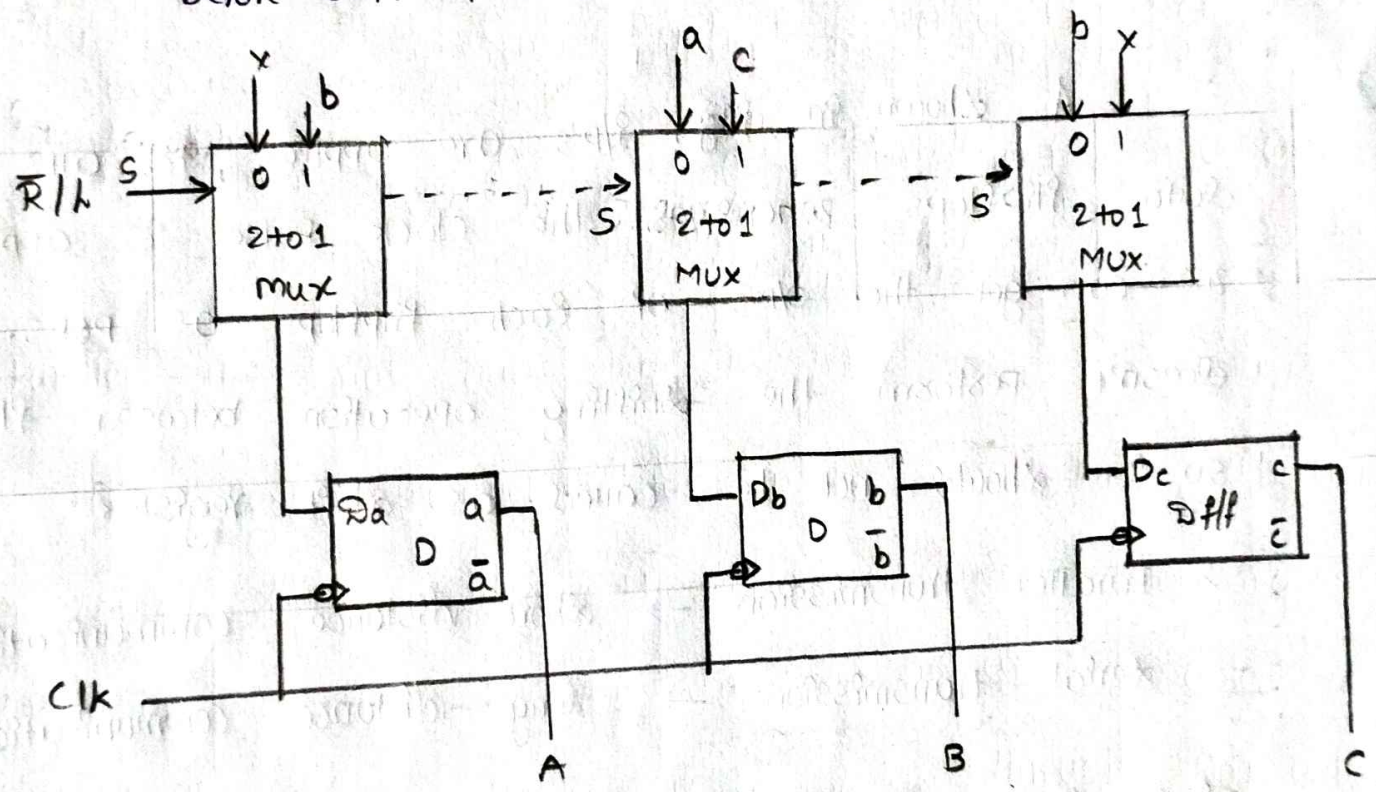

After right operation




before shift operation




After left shift operation.



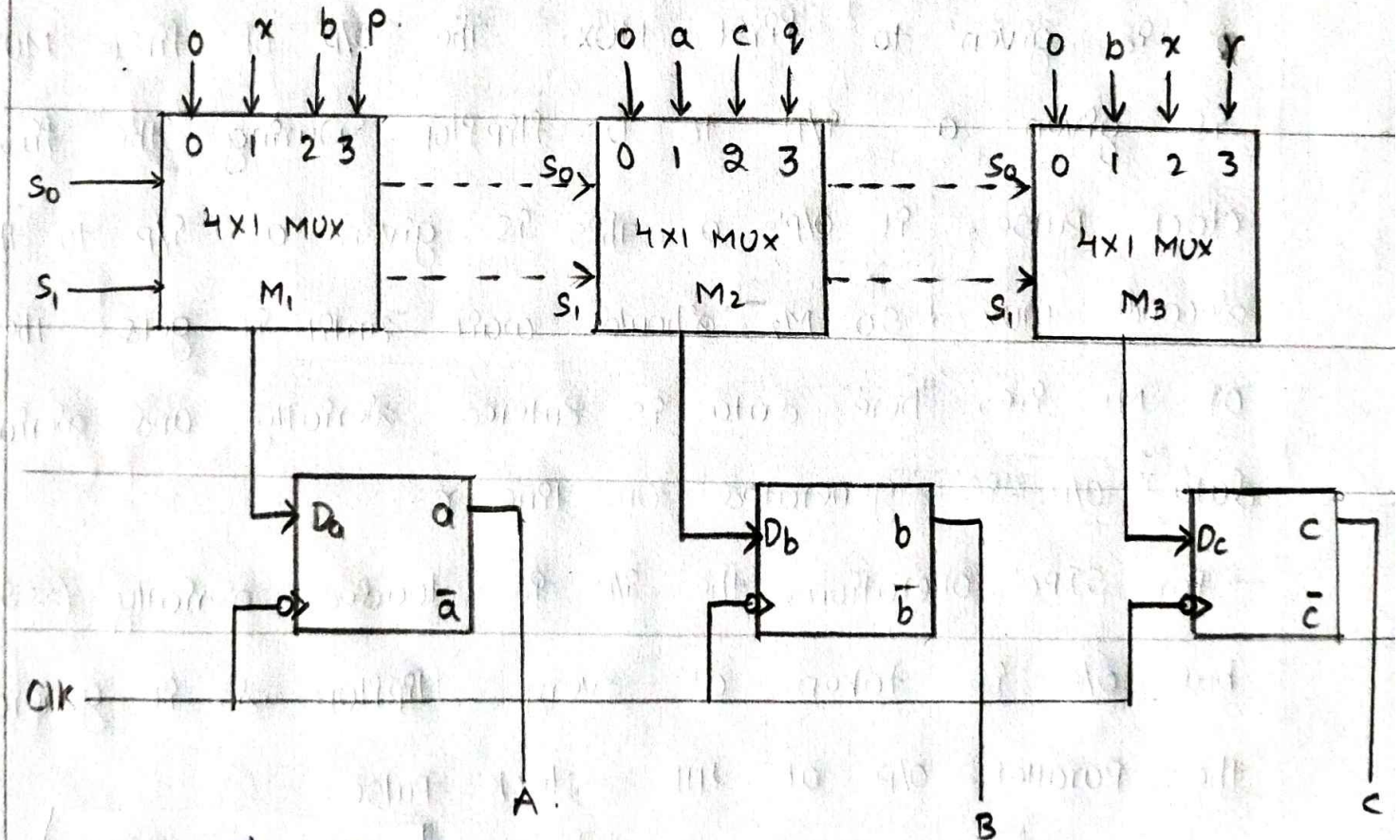

— Selection line I/P : S - $\overline{R}/L$, if $L=0$ - operation is right shift register

if $L=1$ - operation is left shift register

→ Universal shift register:

The universal shift register is designed to perform the following functions.

1) SIPO, SISO, PISO, PIPO operations.
2) shift data right
3) shift data left.
4) load a '0' in each flipflops i.e., clear the register contents.




The operation of MUX depends on the select line input.

| Select lines | | Present state of register | | | MUX O/P's | | | Next state of register | | | Operation |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $S_1$ | $S_0$ | | | | $D_a$ | $D_b$ | $D_c$ | A | B | C | |
| 0 | 0 | a | b | c | 0 | 0 | 0 | 0 | 0 | 0 | clear register |
| 0 | 1 | a | b | c | x | a | b | x | a | b | Right shift |
| 1 | 0 | a | b | c | b | c | x | b | c | x | left shift |
| 1 | 1 | a | b | c | p | q | r | p | q | r | Parallel load. |

a) If $S_1S_0 = 01$, SISO operation is done.

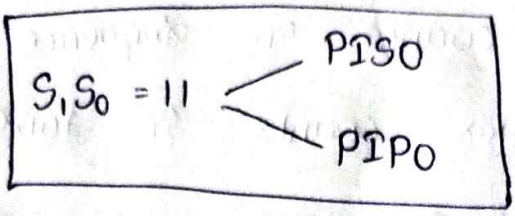
Because I/P is entered serially i.e the first I/P x is given to first MUX. The O/P of first MUX is given as I/P to D flipflop. During the first clock Pulse, it O/P's a. This is given as I/P to the second mux. So $M_2$ should wait until it gets the O/P of $M_1$ i.e., here data is entered serially and serial data O/P is generated on line c.

→ For SIPO operation, the I/P is loaded serially ($\because S_1S_0 = 01$) but O/P is taken at every flipflop so it generates the Parallel O/P at 4th clock pulse.

$S_1S_0 = 01$ < SISO, SIPO

$S_1S_0 = 11$ < PISO, PIPO

b) If $S_1S_0 = 11$, I/P is loaded Parallely & O/P is taken parallely from ABC & O/P is taken serially from O/P line c




→ Counters:-

A digital counter counts the no. of clock pulses applied to it. It is a collection of flipflops that change their states, when trigerred by clock pulses.

Applications:-

1. Digital wrist watches 2) digital oscilloscopes, freq counters, digital multimeters etc.

- counters are broadly clarified in to 2-types.

a) Asynchronous as ripple counters

b) synchronous counters

Based on their operation, they are also classified as.

a) up b) down c) up/down d) sequential

e) Non-sequential counters.

→ An up counter counts in the upward direction like 0, 1, 2, 3 - - -

→ Down counter counts down like 4, 3, 2, 1, 0.

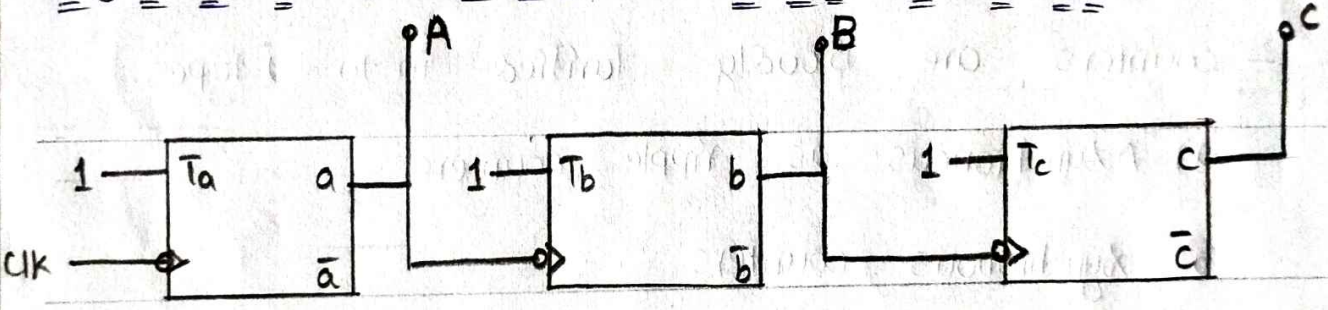
→ up/down counter can counts in both directions either up or down depending on status of control signal.

→ A sequential counter counts in sequence where as a non-sequential counter counts in random fashion.
[Counter counts → 2, 1, 7, 5, 0 means non-sequences]

→ Modulus Counter:- The total no. of distinct states through which a counter programes is called modulus of a counter

for eg: the counter that goes through 5 states 2, 1, 7, 5, 0 is called Modulus-5 or Mod-5 counter

→ Asynchronous UP counter (or) Ripple UP counter



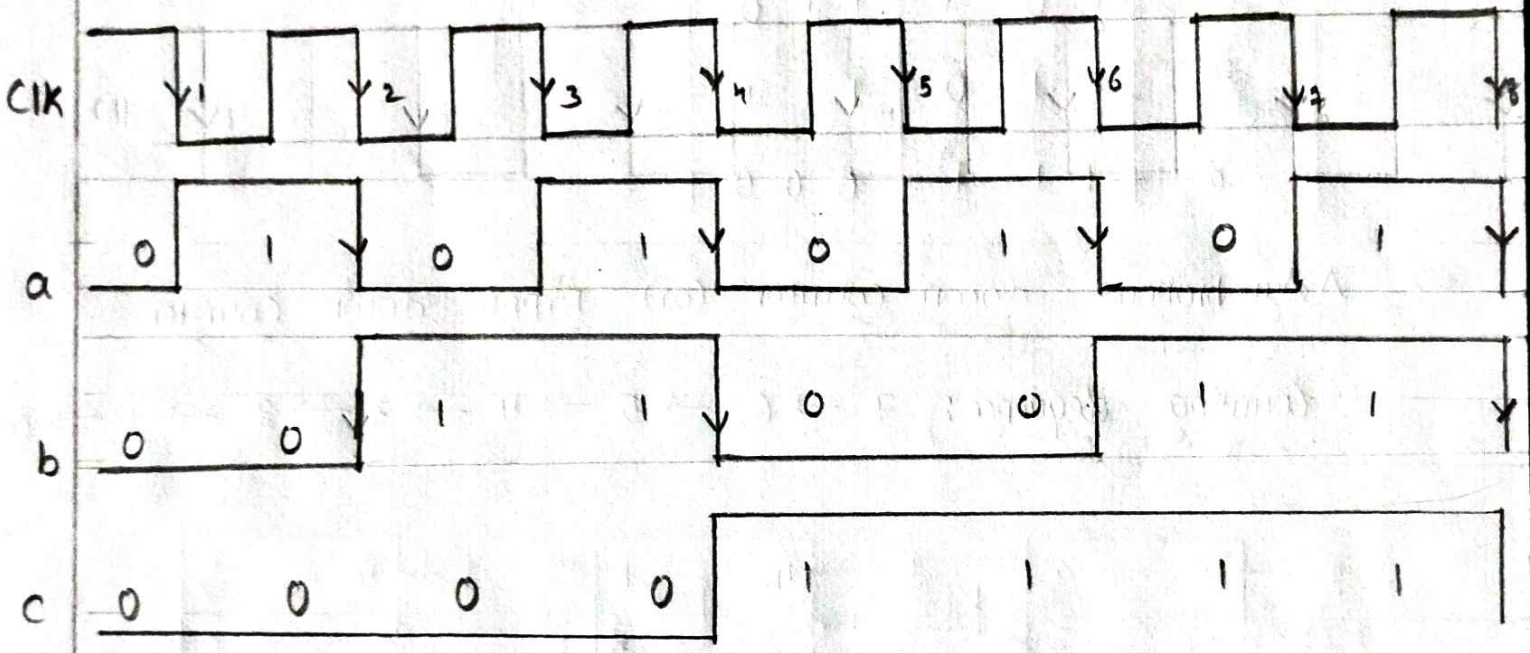

Consider the seq ckt shown in the fig. All the i/p's to T f/f are tied to '1' permanently. The clock (master clock), is applied to the left most f/f to generate LSB. The o/p 'a' of the first or LSB f/f drives the clock i/p of second f/f 'b' and the o/p of second drives the clk i/p of the third or MSB f/f 'c'.

The same clock is not applied to all the f/f's simultaneously in ripple counters. The binary value of 'CBA' at any given time together is called the

state of the Counter.

For every -ve clock edge of the clk. T is activate of T=1 means next state = Inverse Present state.

Initially, abc = 000 so next state of first f/f = 1. The -ve edge of 'a' acts as clk i/p to the 2nd f/f. so during -ve edge of 'a'. 2nd f/f is activated and its o/p = 1.

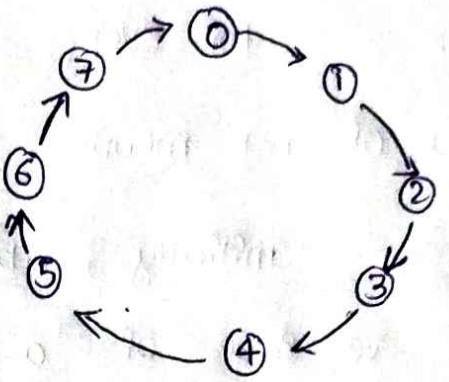



If a Propagation delay of a f/f is considered, the necessary condition to be satisfied is

$$T_{clock} \geq N \times t_{pd}$$

Truth Table

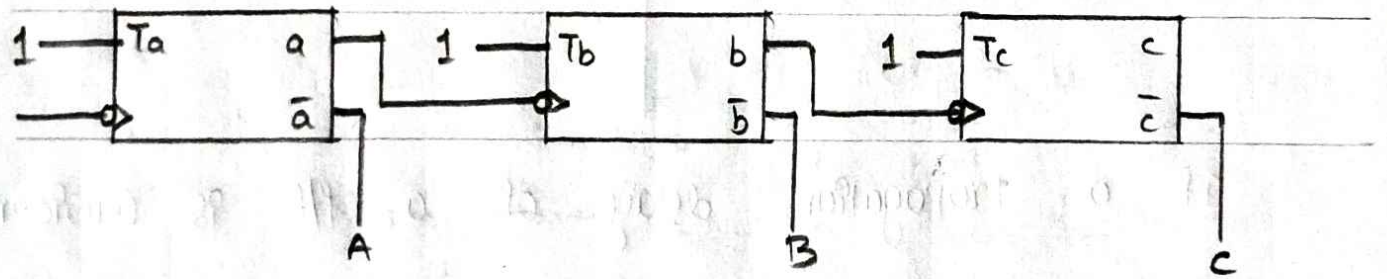
| Clock Edge | Present State c b a | Next State C B A |
|---|---|---|
| 1 | 0 0 0 | 0 0 1 |
| 2 | 0 0 1 | 0 1 0 |
| 3 | 0 1 0 | 0 1 1 |
| 4 | 0 1 1 | 1 0 0 |
| 5 | 1 0 0 | 1 0 1 |
| 6 | 1 0 1 | 1 1 0 |
| 7 | 1 1 0 | 1 1 1 |
| 8 | 1 1 1 | 0 0 0 |

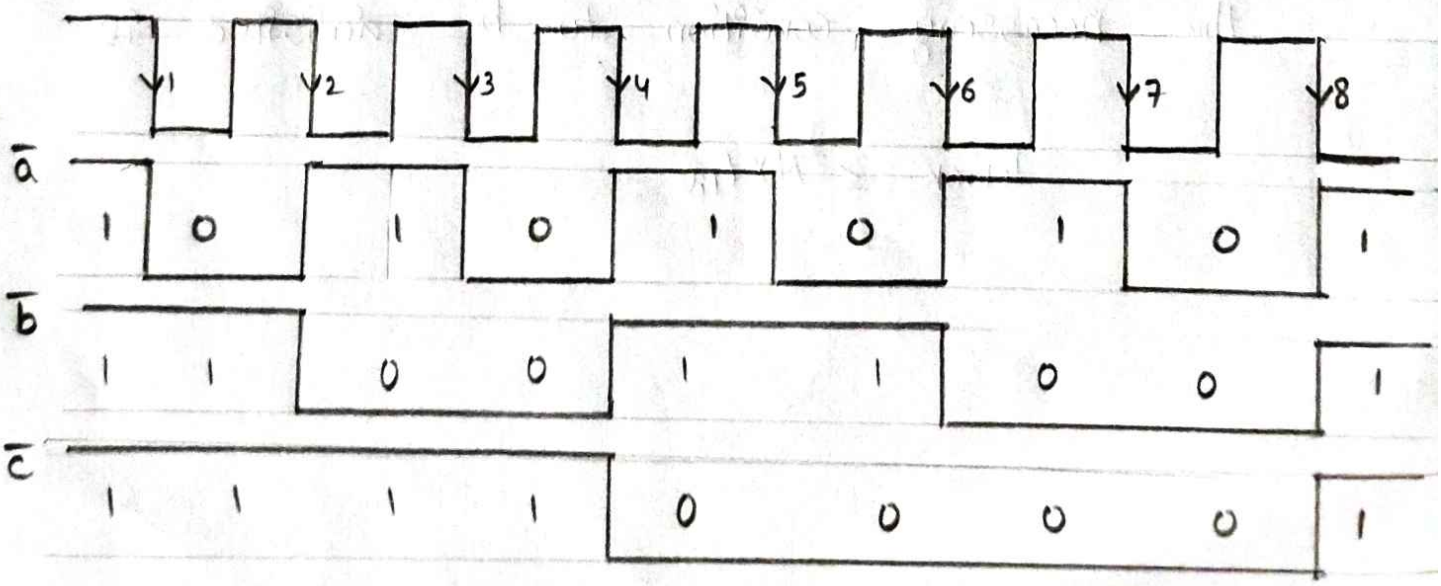



State Transition diagram.

⇒ Asynchronous down counter (or) Ripple down counter

Counting sequence: 7 → 6 → 5 → 4 → 3 → 2 → 1 → 0.



→ If a +ve edge triggered clock is given to T f/f it acts as up counter

→ If a -ve edge triggered clock is given to T f/f it acts as down counter.

→ The no. of f/f's required to count the required no. of states is $2^n \geq m$

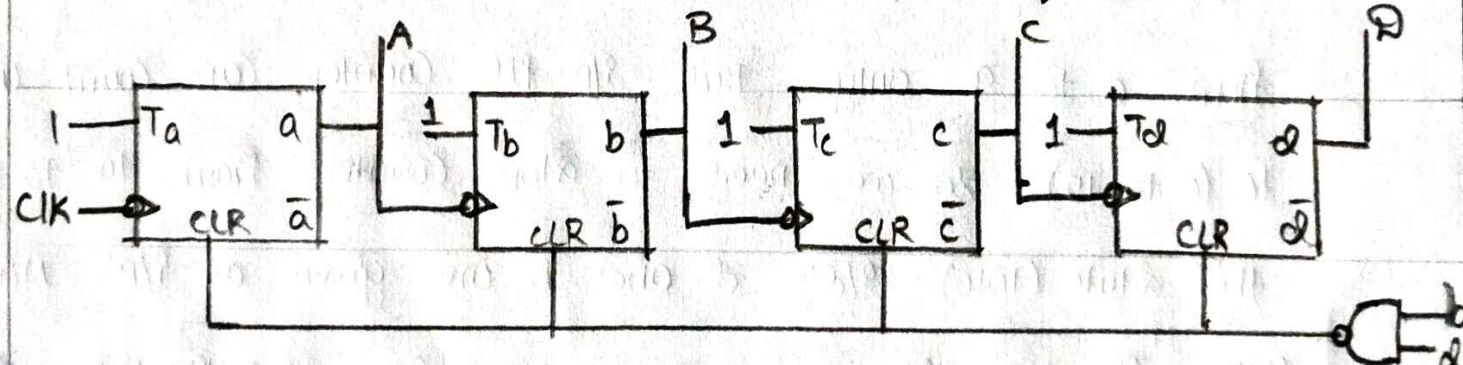
n → no. of f/f's

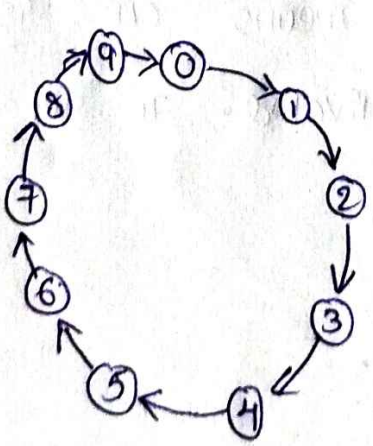
m → no. of states.

→ Decade Counter (or) Mod-10 counter :-

Suppose the required states are from 0 to 9 then it is better to use 4 f/f's. The remaining states i.e. 10 to 15 are skipped

$2^n \geq 10 \Rightarrow 2^4 \geq 10$ i.e 4 f/f's are required.




State Transition diagram

## Mod-10 Counter Truth Table

| Clk edges | Present state | | | | Next state | | | |
|---|---|---|---|---|---|---|---|---|
| | d | c | b | a | D | C | B | A |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 2 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 3 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| 4 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |
| 5 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| 6 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |
| 7 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 |
| 8 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| 9 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 10 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |

As it is sequence counter, it can counts the states from 0 to 9 only But i/p flf Counter can count upto 16 (0 to 15). so we need to stop Counter from 10 to 15, the state (1010) i/p's d and b are given as i/p's NAND gate. If CLR i/p = 1 means CLR is de-activated. If CLR = 0, is CLR activated. To get this state b, d are given to NAND gate.

⇒ Synchronous Counters (or) Parallel Counters:-

Design procedure:-

1. Determine the no. of f/f's needed, based on the states of a counter ($2^n \geq M$)
2. Draw the state diagram and a binary transition table in which the next state is specified for each present state.
3. Also determine the i/p's based on the next state transitions.
4. The design equations are obtained for the i/p's of all the f/f's using k-maps
5. The ckt is designed and then verified with design specifications.

⇒ Design a 3-bit synchronous up counter using JK f/f's

3-bit up counter → no. of f/f's required = 3

states are: $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 0$

Assuming the f/f's to be a, b and c, the binary transition table can be drawn as shown in the fig.

| Min term | Present state $Q_c$ $Q_b$ $Q_a$ | Next state $Q_{c+1}$ C | $Q_{B+1}$ B | $Q_{A+1}$ A | Excitation inputs $J_c$ | $K_c$ | $J_b$ | $K_b$ | $J_a$ | $K_a$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 0 0 | 0 | 0 | 1 | 0 | x | 0 | x | 1 | x |
| 1 | 0 0 1 | 0 | 1 | 0 | 0 | x | 1 | x | x | 1 |
| 2 | 0 1 0 | 0 | 1 | 1 | 0 | x | x | 0 | 1 | x |
| 3 | 0 1 1 | 1 | 0 | 0 | 1 | x | x | 1 | x | 1 |
| 4 | 1 0 0 | 1 | 0 | 1 | x | 0 | 0 | x | 1 | x |
| 5 | 1 0 1 | 1 | 1 | 0 | x | 0 | 1 | x | x | 1 |
| | 1 1 0 | 1 | 1 | 1 | x | 0 | x | 0 | 1 | x |
| | 1 1 1 | 0 | 0 | 0 | x | 1 | x | 1 | x | 1 |

JK f/f excitation Table

| $Q_n$ | $Q_{n+1}$ | J | K |
|---|---|---|---|
| 0 | 0 | 0 | x |
| 0 | 1 | 1 | x |
| 1 | 0 | x | 1 |
| 1 | 1 | x | 0 |

$J_a = \Sigma m(0,2,4,6) + \Sigma d(1,3,5,7)$

$K_a = \Sigma m(1,3,5,7) + \Sigma d(0,2,4,6)$

$J_b = \Sigma m(1,5) + \Sigma d(2,3,6,7)$

$K_b = \Sigma m(3,7) + \Sigma d(0,1,4,5)$

$J_c = \Sigma m(3) + \Sigma d(4,5,6,7)$

$K_c = \Sigma m(7) + \Sigma d(0,1,2,3)$

For $J_a$ :-

| c \ ba | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 1 | x | x | 1 |
| 1 | 1 | x | x | 1 |

$J_a = 1$

// y $K_a = 1$

K-map for $J_b$

| c \ ba | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | | 1 | x | x |
| 1 | | 1 | x | x |

$J_b = a$

K-map for $K_b$

| c \ ba | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | x | x | 1 | |
| 1 | x | x | 1 | |

$K_b = a$

K-map for $J_c$

| c \ ba | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | | | 1 | |
| 1 | x | x | x | x |

$J_c = ab$

K-map for $K_c$

| c \ ba | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | X | X | X | X |
| 1 | | | 1 | |

$K_c = ab$

$J_a = K_a = 1$ , $J_b = K_b = a$ , $J_c = K_c = ab$

Design Equations

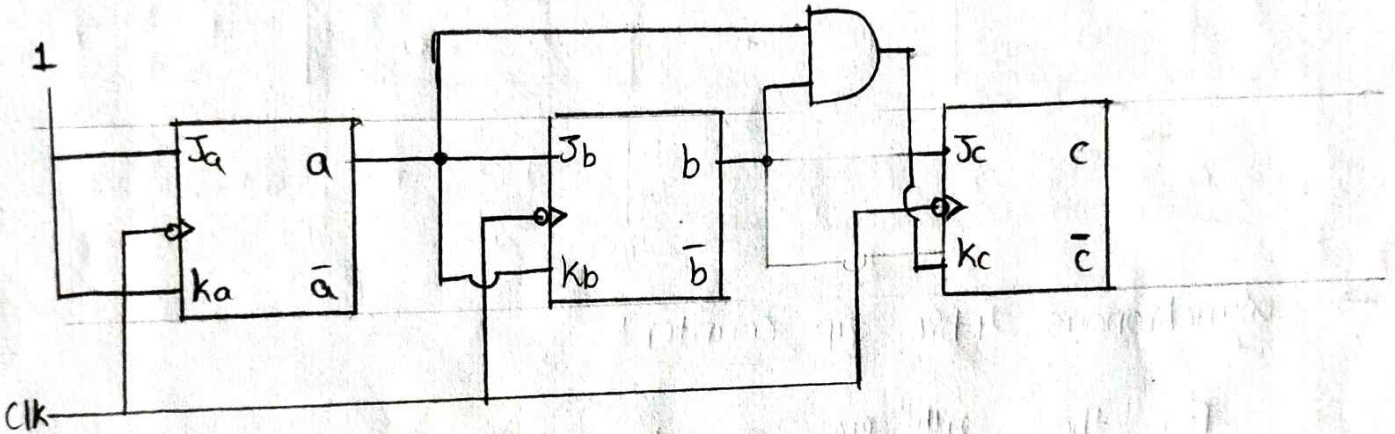



Fig: Synchronous 3-bit Up Counter

Assume the propagation delay of each flip flop is 20ns and that of the AND gate is 5ns

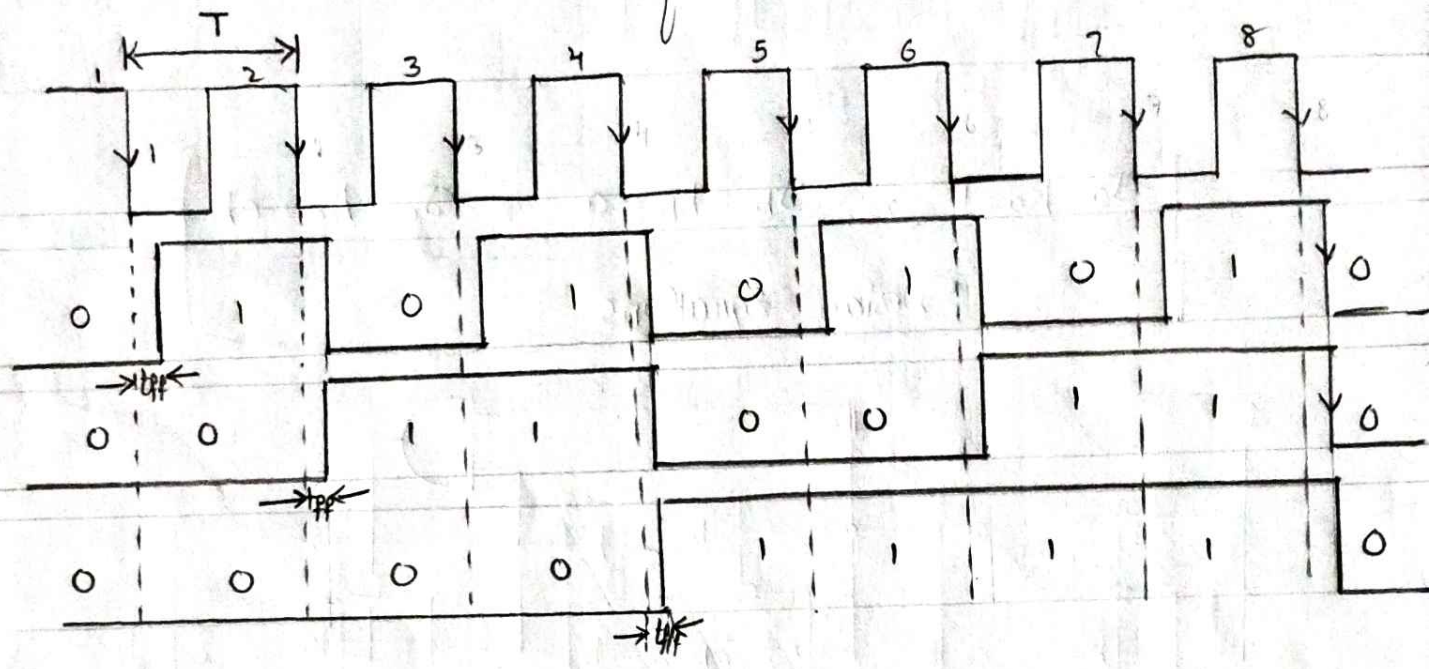
J and K i/p's are connected together. So it acts as a T-flipflop

If $J_a = K_a = 1$ means Next State = Inverse present state.

1. If $J_a = K_a = 1$ & for every -ve edge of the clock, 'a' flip flop will toggle.

2. If $J_b = K_b = a = 1$ & for every -ve edge of the clock, 'b' flip flop will toggle.

3. If $J_c = K_c = ab = 1$ i.e ($a=1$ & $b=1$) & for every -ve edge of the clk, 'c' f/f will toggle.




→ Synchronous 4 bit up counter:

For the 4th f/f, $J_d = K_d = abc$

$J_c = K_c = ab$, $J_b = K_b = a$, $J_a = K_a = 1$

→ Synchronous Non-sequential counters:-

The procedure is same as that of synchronous sequential But we have to write Next state as per the state diagram given (or) states given.

Problems:-

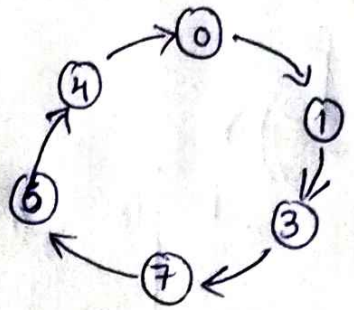
⇒

→ Design a synchronous counter with T f/f's that goes through the binary repeated sequence.

0, 1, 3, 7, 6, 4, 0, 1 . . .

Sol:- Largest decimal count is 7. so

no. of states $M = 7$, no. of f/f's $n = 3$




state diagram

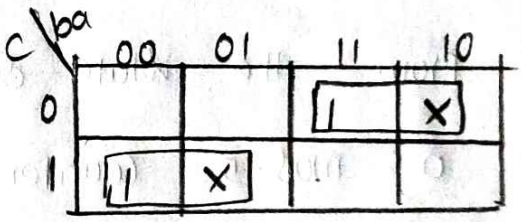
| Present state | Next state |
|---|---|
| 0 | 1 |
| 1 | 3 |
| 3 | 7 |
| 7 | 6 |
| 6 | 4 |
| 4 | 0 |

| Q | $Q_{n+1}$ | T |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Excitation Table of T f/f.

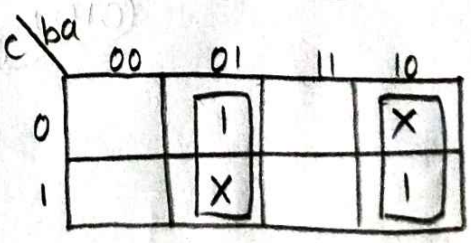
| Minterms | Present state | Next state | F/F i/ps | | |
|---|---|---|---|---|---|
| | c b a | C B A | $T_c$ | $T_b$ | $T_a$ |
| 0 | 0 0 0 | 0 0 1 | 0 | 0 | 1 |
| 1 | 0 0 1 | 0 1 1 | 0 | 1 | 0 |
| 2 | 0 1 0 | x x x | x | x | x |
| 3 | 0 1 1 | 1 1 1 | 1 | 0 | 0 |
| 4 | 1 0 0 | 0 0 0 | 1 | 0 | 0 |
| 5 | 1 0 1 | x x x | x | x | x |
| 6 | 1 1 0 | 1 0 0 | 0 | 1 | 0 |
| 7 | 1 1 1 | 1 1 0 | 0 | 0 | 1 |

K-Map for $T_c$



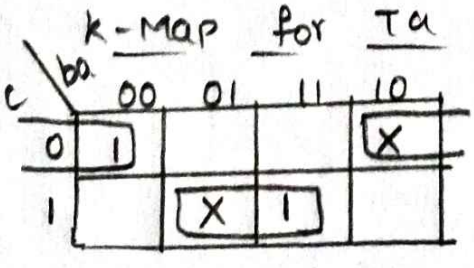

$T_c = c \oplus b$.

K-Map for $T_b$




$T_b = a \oplus b$.

K-Map for $T_a$




$T_a = c \oplus a$

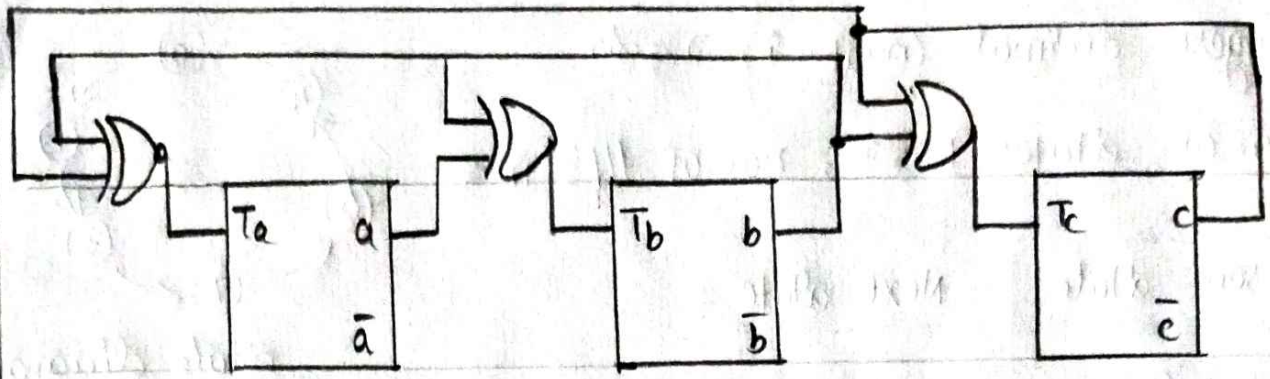


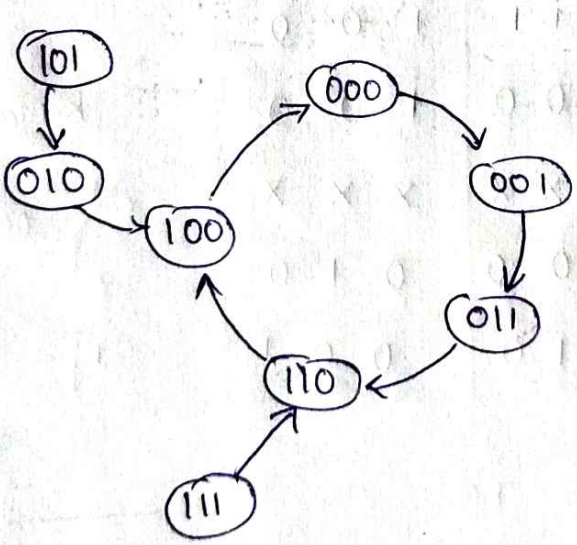

Fig: Mod-6 Synchronous Counter

Lock-out Condition

The two unused states are: abc = 010 & 101

| Present State c b a | $T_c = c \oplus b$ | $T_b = a \oplus b$ | $T_a = c \odot b$ | Next state C B A |
|---|---|---|---|---|
| 0 1 0 | 1 | 1 | 0 1 | 1 0 1 |
| 1 0 1 | 1 | 1 | 0 1 | 0 1 0 |

→ From the state diagram shown, design and draw a mod-5 counter using D ff's.



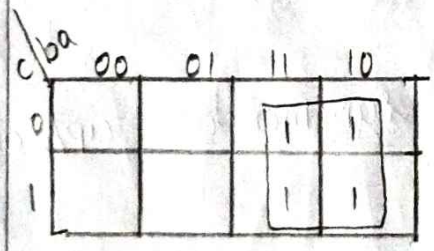
Sol:- From the given state diagram, if present state is 000, Next state is 001. If present state is 110, Next state is 100 and so on.

| Minterm | Present State c b a | Next State C B A | Flip flop i/p's $D_C$ $D_b$ $D_a$ |
|---|---|---|---|
| 0 | 0 0 0 | 0 0 1 | 0 0 1 |
| 1 | 0 0 1 | 0 1 1 | 0 1 1 |
| 2 | 0 1 0 | 1 0 0 | 1 0 0 |
| 3 | 0 1 1 | 1 1 0 | 1 1 0 |
| 4 | 1 0 0 | 0 0 0 | 0 0 0 |
| 5 | 1 0 1 | 0 1 0 | 0 1 0 |
| 6 | 1 1 0 | 1 0 0 | 1 0 0 |
| 7 | 1 1 1 | 1 1 0 | 1 1 0 |

| $Q_n$ | $Q_{n+1}$ | D |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

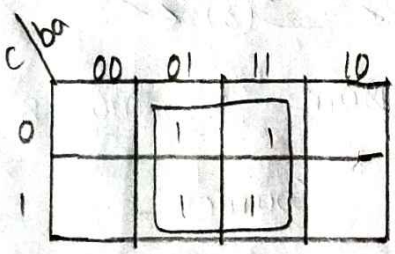
Excitation table of D f/f

K-Map for $D_C$



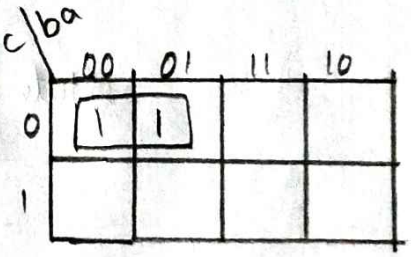

$D_C = b$

K-Map for $D_b$

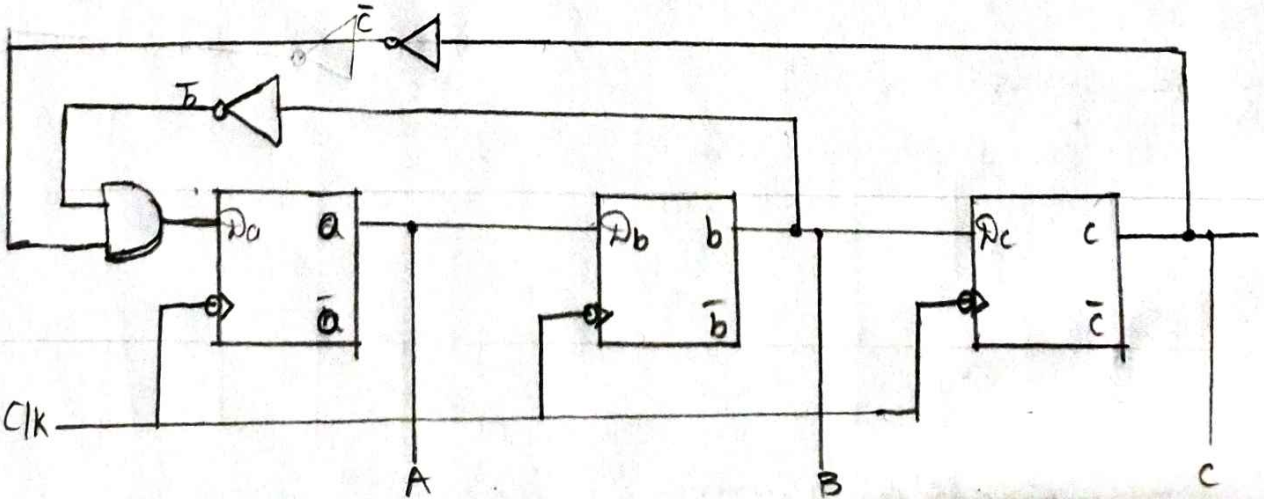



$D_b = a$

K-map of $D_a$



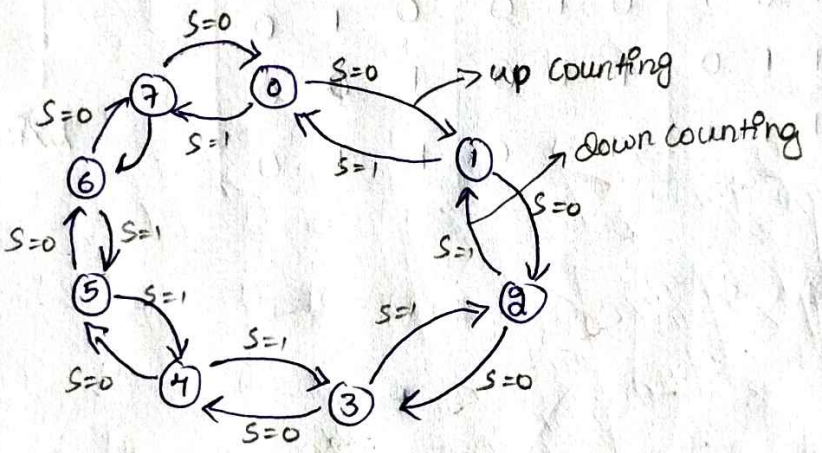

$D_a = \bar{c}\,\bar{b}$



→ Synchronous Up/Down Counters :-

It is having the features of both up & down counting depending on the status of the control signal(s). If S=0, the counter should count up or else if S=1, it should count down.

→ Design a 3-bit synchronous up/down counter, having a control signal S, using JK f/f's. It should count up with S=0 or else count down if S=1.




State diagram of mod-8 synchronous up/down counter

| Control Signal | Present State c | b | a | Next State C | B | A | Flip-flop Inputs $J_c$ | $K_c$ | $J_b$ | $K_b$ | $J_a$ | $K_a$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | X | 0 | X | 1 | X |
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | X | 1 | X | X | 1 |
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | X | X | 0 | 1 | X |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | X | X | 1 | X | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 | 1 | X | 0 | 0 | X | 1 | X |
| 0 | 1 | 0 | 1 | 1 | 1 | 0 | X | 0 | 1 | X | X | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 | 1 | X | 0 | X | 0 | 1 | X |
| 0 | 1 | 1 | 1 | 0 | 0 | 0 | X | 1 | X | 1 | X | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | X | 1 | X | 1 | X |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | X | 0 | X | X | 1 |
| 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | X | X | 1 | 1 | X |
| 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | X | X | 0 | X | 1 |
| 1 | 1 | 0 | 0 | 0 | 1 | 1 | X | 1 | 1 | X | 1 | X |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | X | 0 | 0 | X | X | 1 |
| 1 | 1 | 1 | 0 | 1 | 0 | 1 | X | 0 | X | 1 | 1 | X |
| 1 | 1 | 1 | 1 | 1 | 1 | 0 | X | 0 | X | 0 | X | 1 |

K-Map for $k_c$

| Sc \ ba | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | | | 1 | |
| 01 | X | X | X | X |
| 11 | X | X | X | X |
| 10 | 1 | | | |

$J_c = \bar{s}ba + s\bar{a}\bar{b}$

K-Map for $k_c$ :-

| c \ ba | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | X | X | X | X |
| 01 | | | 1 | |
| 11 | 1 | | | |
| 10 | X | X | X | X |

$K_c = \bar{s}ba + s\bar{a}\bar{b}$

K-Map for $J_b$ :

| c \ ba | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | | 1 | X | X |
| 01 | | 1 | X | X |
| 11 | 1 | | X | X |
| 10 | 1 | | X | X |

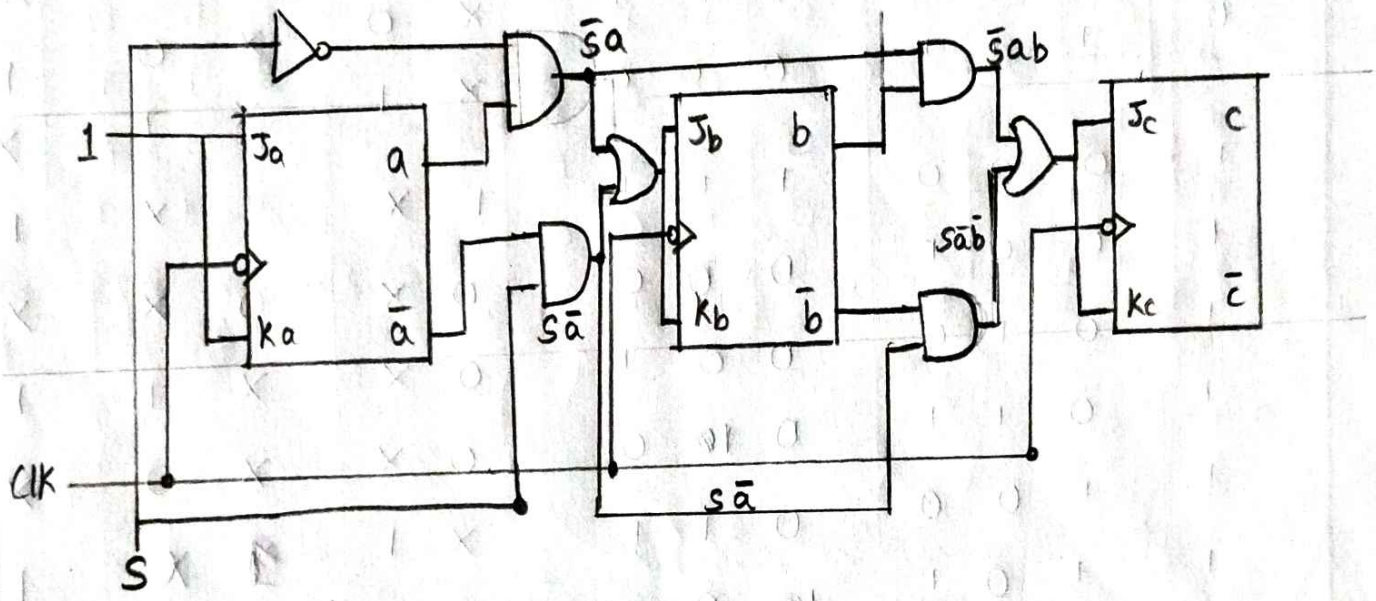
$J_b = \bar{s}a + s\bar{a}$

K-Map for $K_b$ :

| c \ ba | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | X | X | 1 | |
| 01 | X | X | 1 | |
| 11 | X | X | | 1 |
| 10 | X | X | | 1 |

$K_b = \bar{s}a + s\bar{a}$

$\therefore J_a = K_a = 1$.




→ Shift Register Counters :-

Shift Register Counters come under Synchronous Non-sequential type. There are classified into 2 types

(1) Ring Counter

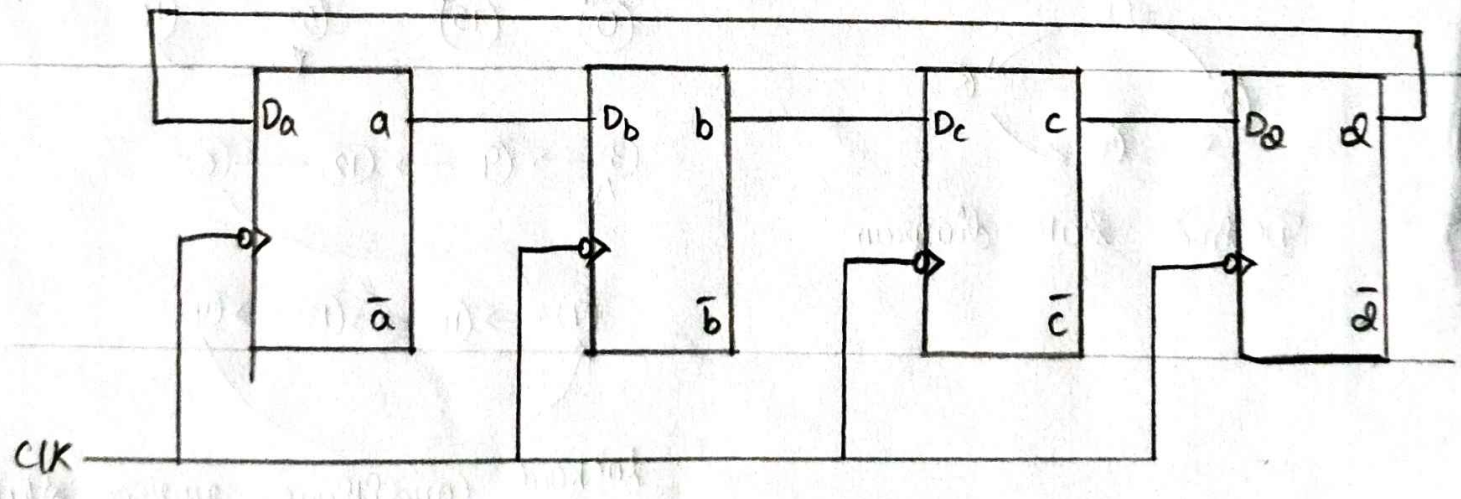
(2) Twisted ring counter.

## Ring Counter:-

It can be constructed using D flip-flops in a shift register arrangement using SISO concept. The o/p of each f/f is connected as i/p to the next f/f in a ring fashion.

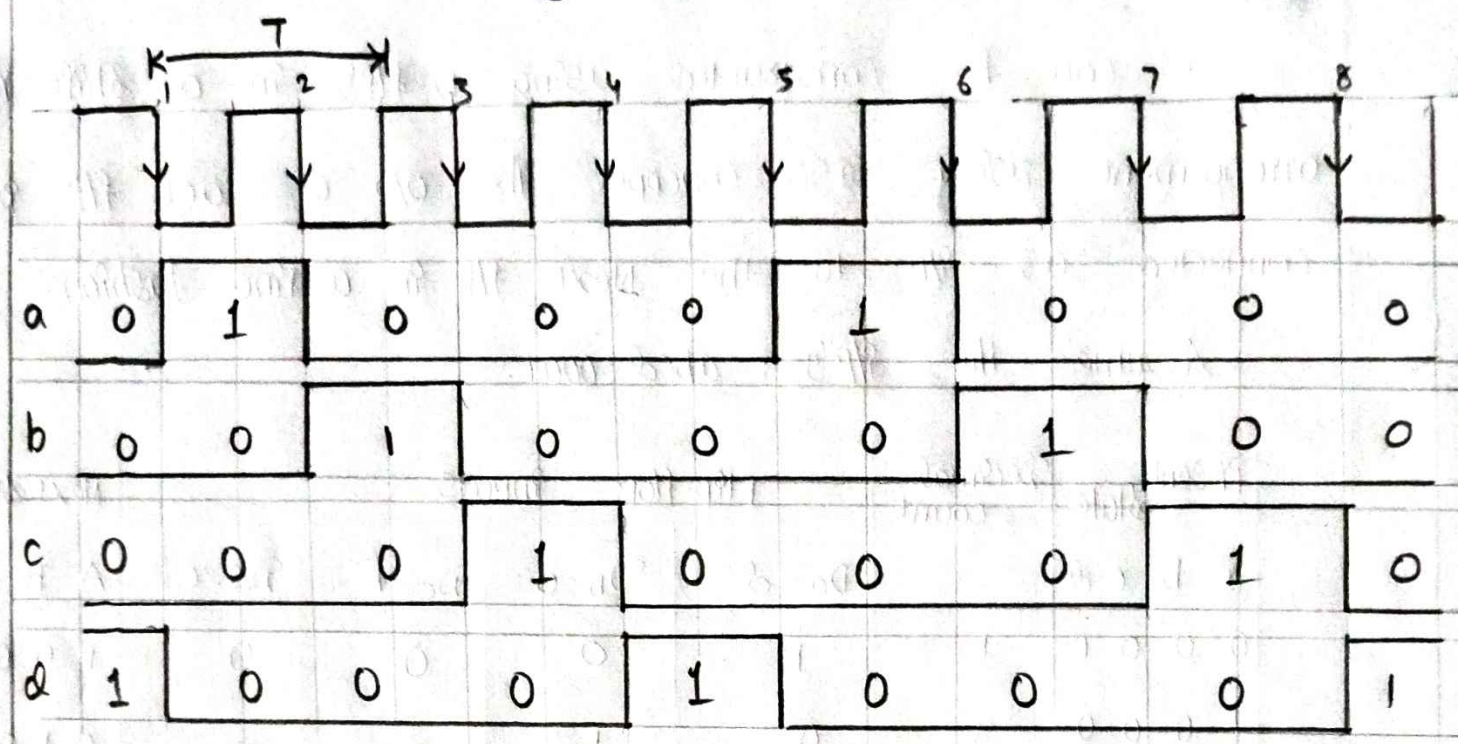
Assume the i/p's : abcd = 0001.

| Present state | | | | Decimal count | Flip-flop inputs | | | | Next state | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | b | c | d | | $D_a = d$ | $D_b = a$ | $D_c = b$ | $D_d = c$ | A | B | C | D |
| 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 8 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 4 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 2 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |



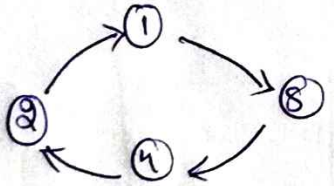

Ring Counter.

The waveforms are generated as shown in the fig.

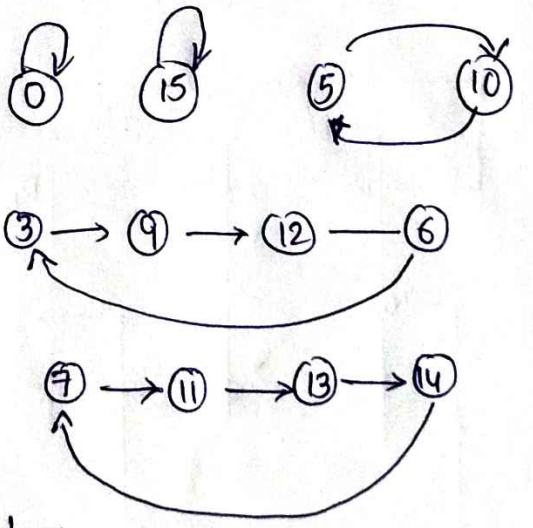



A Ring counter contains only 4 states : 1, 8, 4 and 2 After that it is Repeating.

It have un desirable states also




Desired state diagram




Lockout conditions - unused states

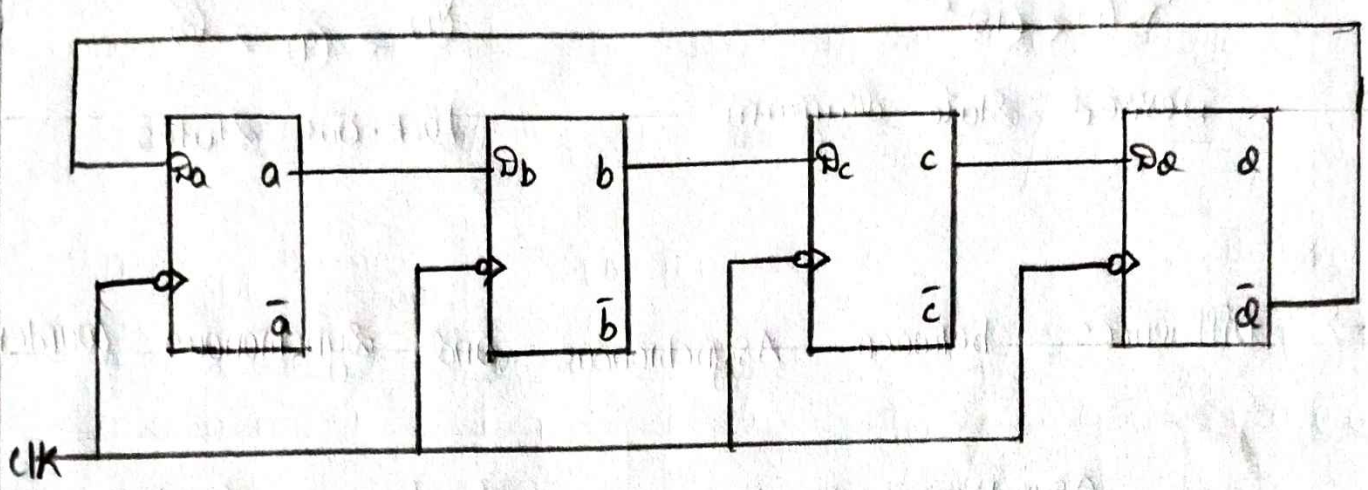
As shown in the fig, the lockout states are: 0, 15 ....

If the persent state = 0101 (5), Next State = 1010 (10).

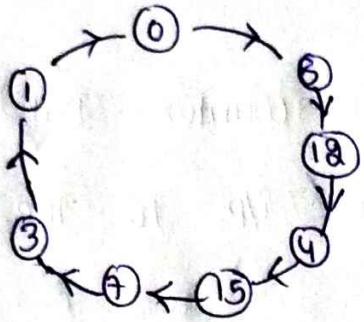
If Present state = 10, Next state = 5 : so it is repeating continuously.

→ Twisted - Ring (Johnson) Counter:

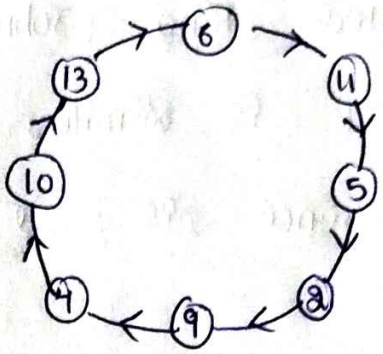
It is similar to the Ring counter. But the difference is: $\bar{d}$ is given as i/p to the 'a' flf.




| Present State | | | | Decimal | flip flop i/p's | | | | Next State | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | b | c | d | Count | $D_a = \bar{d}$ | $D_b = a$ | $D_c = b$ | $D_d = c$ | A | B | C | D |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 8 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 12 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 0 | 14 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 15 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 | 7 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 1 | 3 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |




Desired state diagram




Lock-out states

→ Differences between Asynchronous and Synchronous Counters :-

| Asynchronous Counters | Synchronous Counters |
|---|---|
| 1. In this the o/p of first f/f drives the correct clk for second one and so on. | 1. There is no connection between the o/p of first f/f & clock i/p of next f/f and so on. |
| 2. All the f/f's are not clocked simultaneously | 2. All the f/f's are clocked simultaneously |
| 3. Design & implementation is very simple | 3. Design & implementation is complex as the no. of states ↑es |
| 4. Drawback: low speed | 4. They are very fastes |

# UNIT - III

## Register Transfer Language and Micro operations

## Basic Computer Organization and Design

* 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.

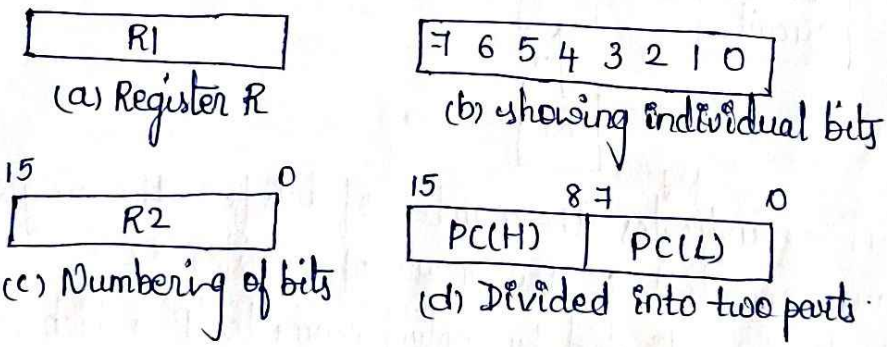



fig: Block diagram of register.

→ 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:

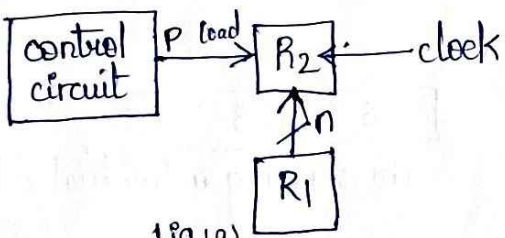



fig (a)

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 $R_2$.

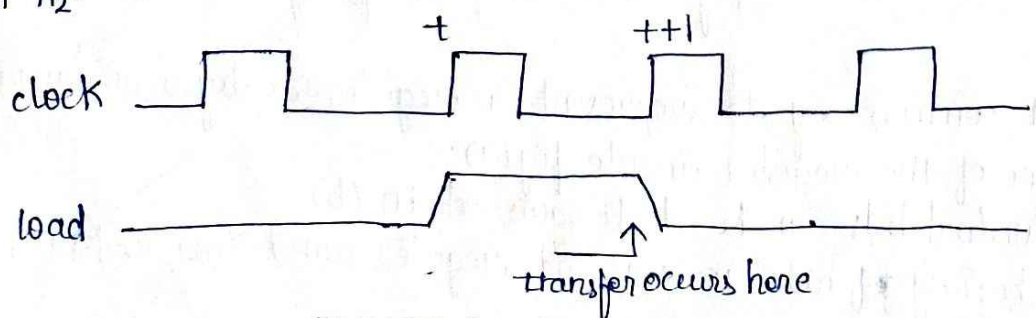



fig (b) Timing diagram

↳ In the timing diagram P is activated in control section by the raising edge of clock pulse at time t. (3)

↳ 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 (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 :-

*

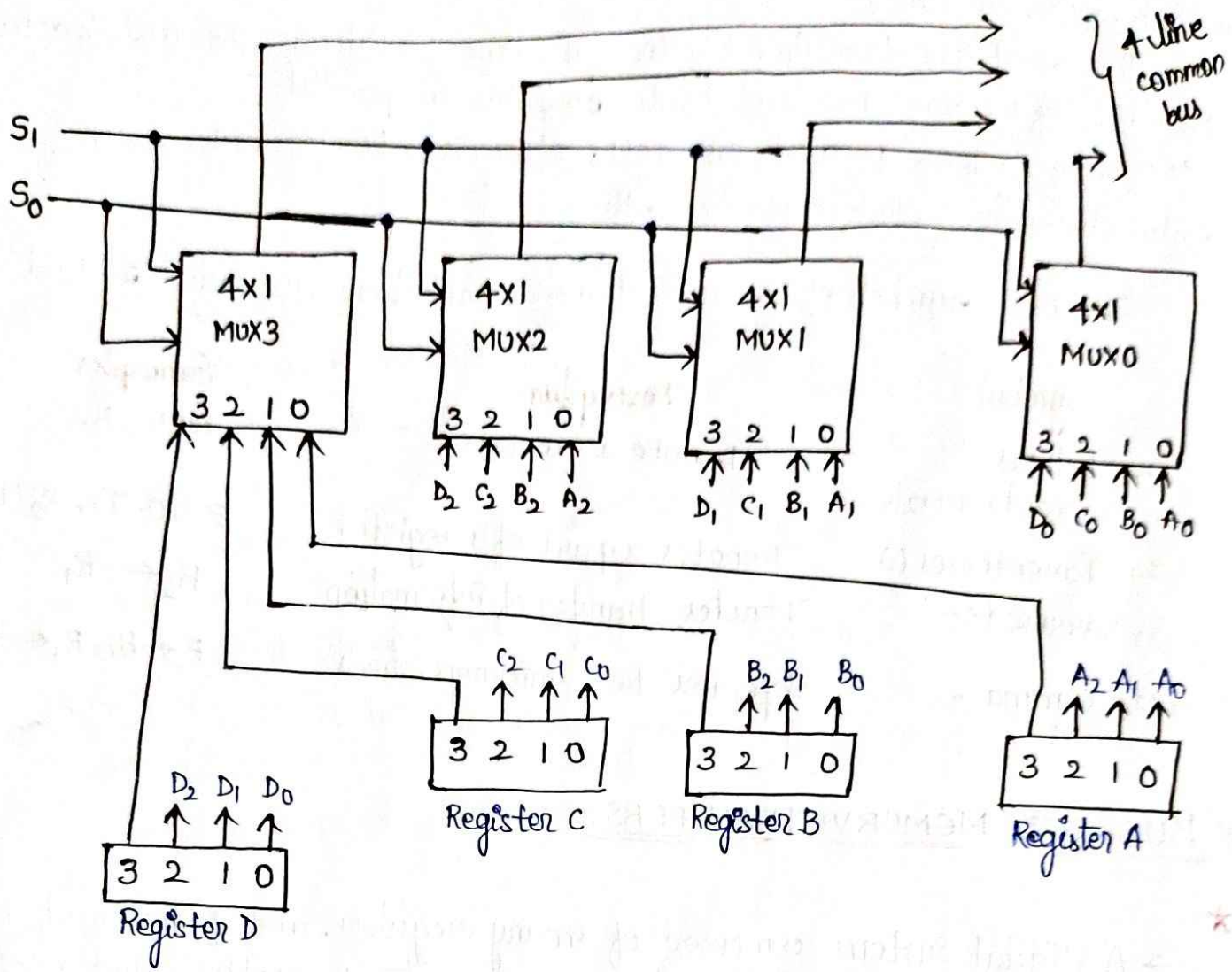
→ A digital system composed of many registers, and paths must be provided to transfer information from one register to another. The no. of wires will be excessive if seperate lines are used between each register and all other register in the system.

→ 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




→ 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

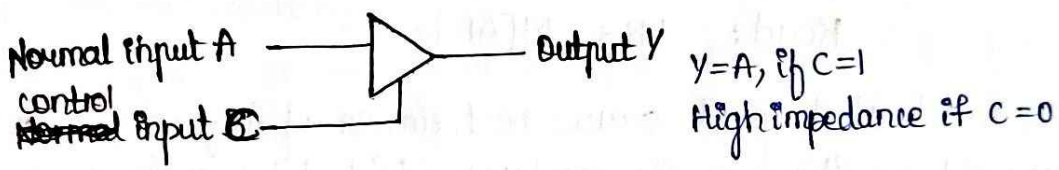
| S1 | S0 | Register Selected |
|---|---|---|
| 0 | 0 | A |
| 0 | 1 | B |
| 1 | 0 | C |
| 1 | 1 | D |

Three state Bus Buffers:-

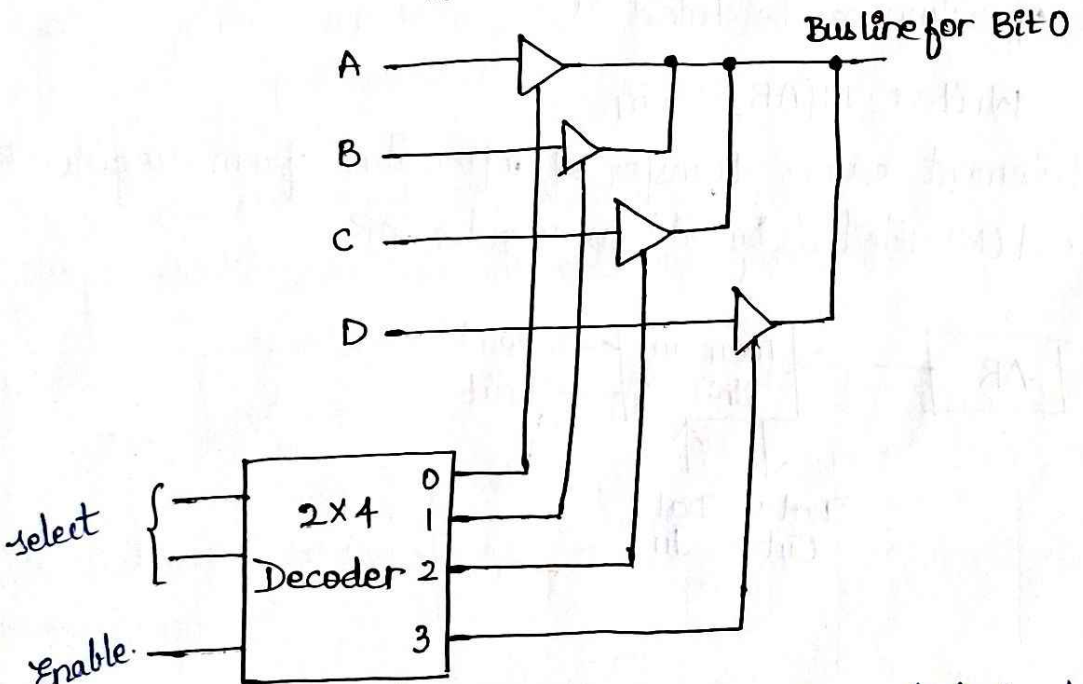
→ A bus system can also be constructed using 3 state gates instead of multiplexers

→ 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 2×4 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

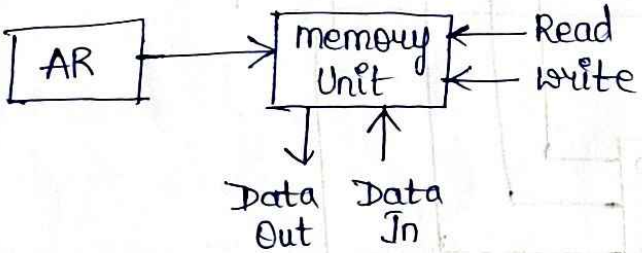
Read: $DR \leftarrow M[AR]$

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:-

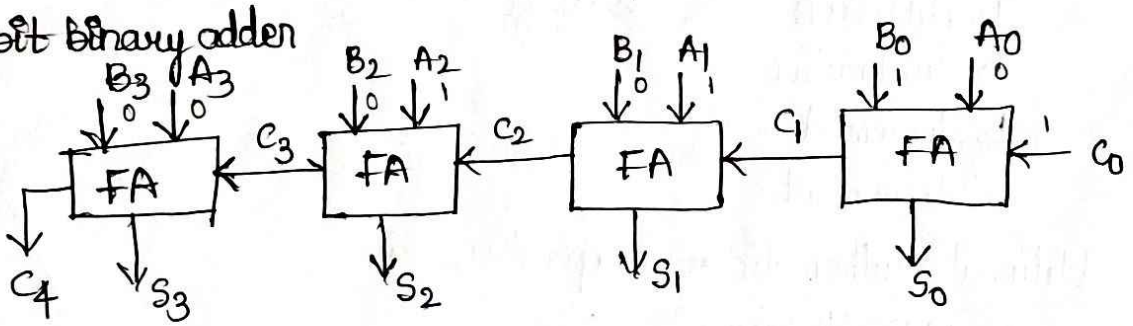
→ The add microoperation requires registers that can hold the data and the digital components that can perform arithematic addition.

→ 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.

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 ~~(A or B)~~ $R_2$.

→ 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 $R_2$)

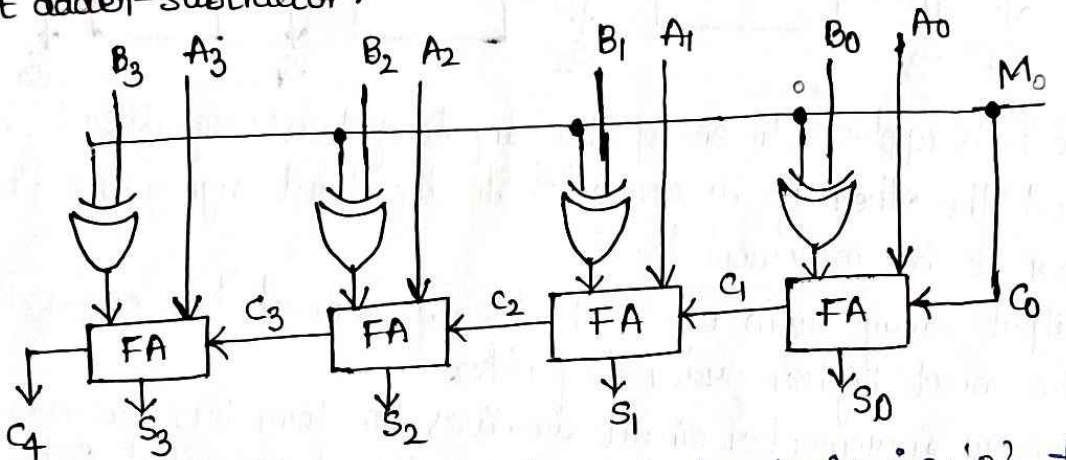
★ Binary Adder-Subtractor:-

→ 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

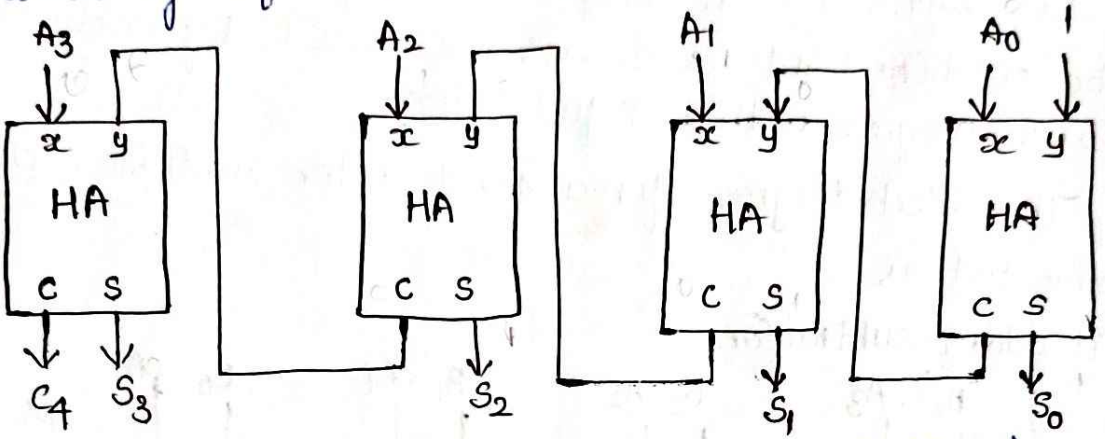
★ Binary Incrementer :-

→ 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.

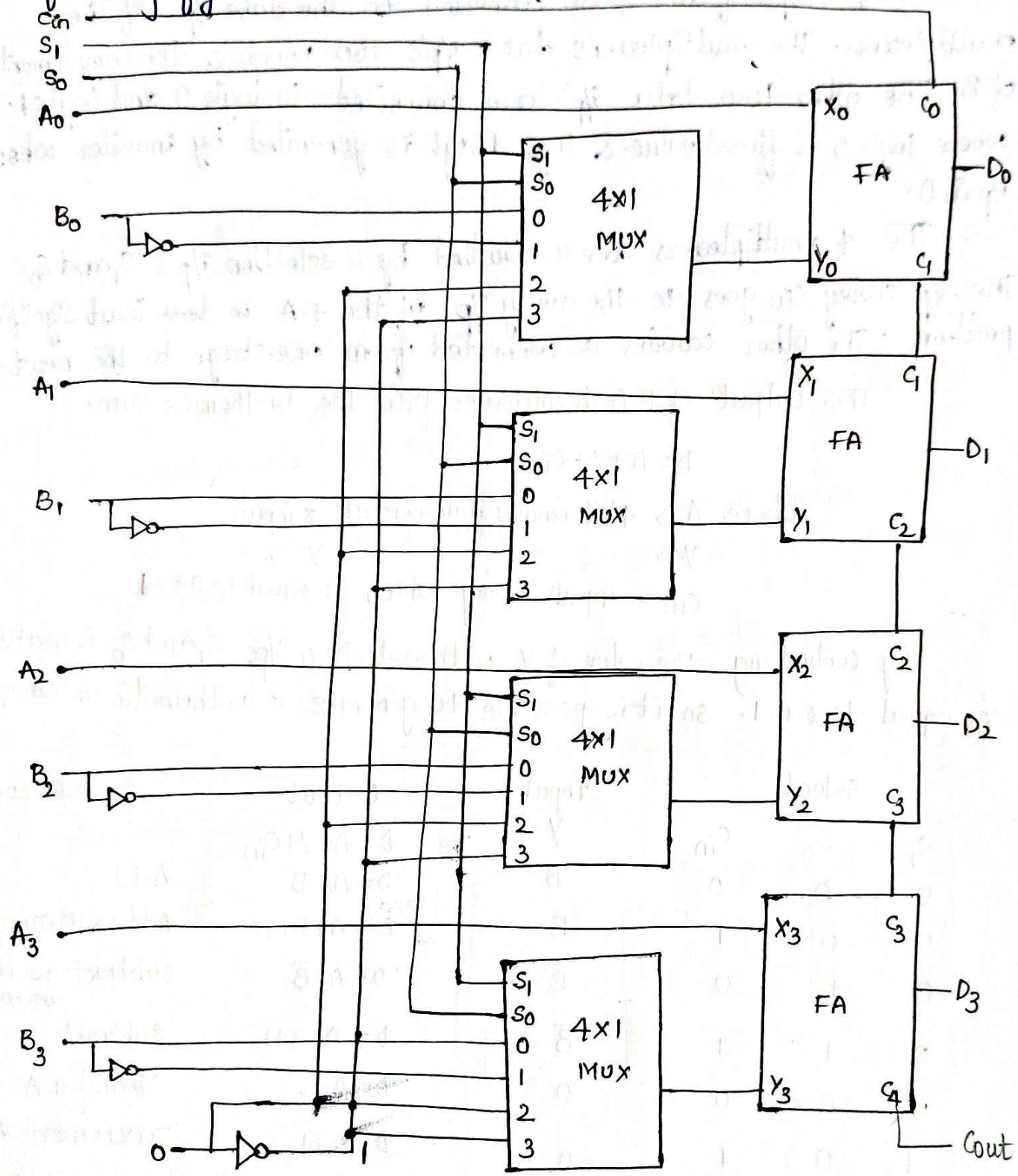
→ The binary incrementer circuit receives the four bits from $A_0$ through $A_3$, adds one to it, and generates the incremented output in $S_0$ through $S_3$.

→ 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 $\vee$ 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.

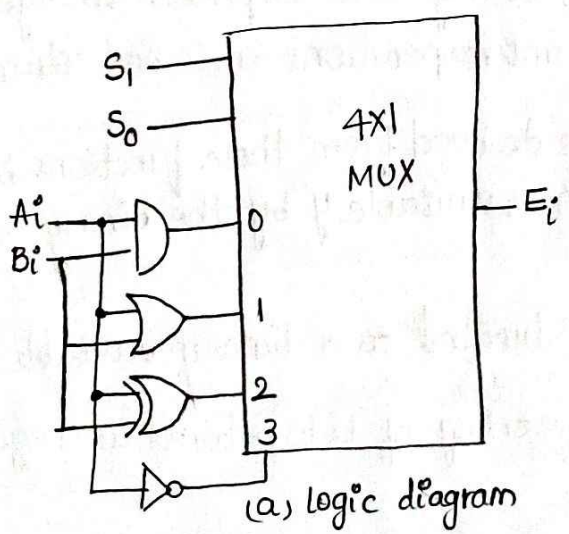
The 16 logic microoperations are derived from these functions by replacing x by the binary content of register A & variable y by the binary content of register B.

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)
---------
```

→ 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

$R_1 \leftarrow shl\ R1$

are 2 microoperations that specify a 1 bit shift to left of the content of register $R_1$ 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.

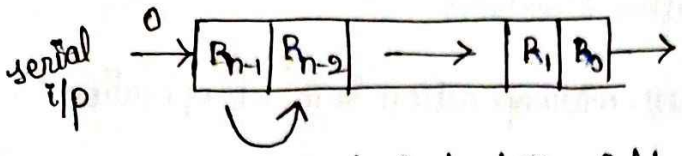



fig: logical shift right

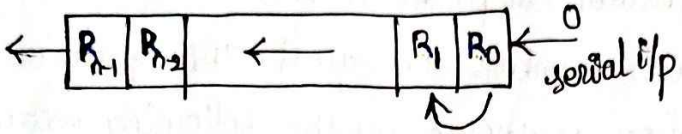



fig: logical shift left.

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

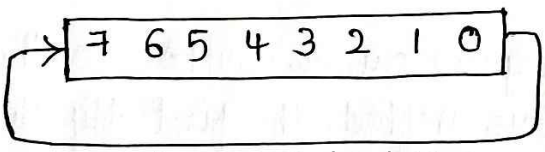



fig:- Rotate left

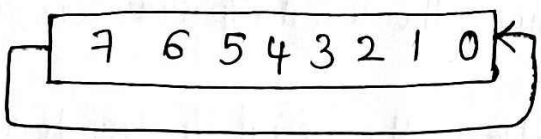



fig:- Rotate right.

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

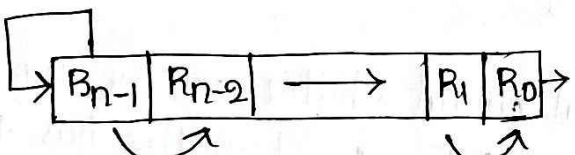



fig: arithematic shift right

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

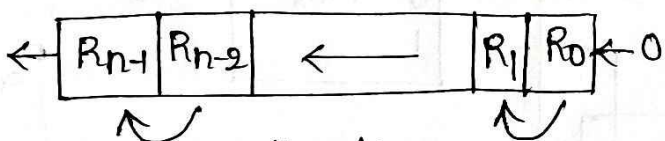



fig: arithematic shift left (ASL)

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

* 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.

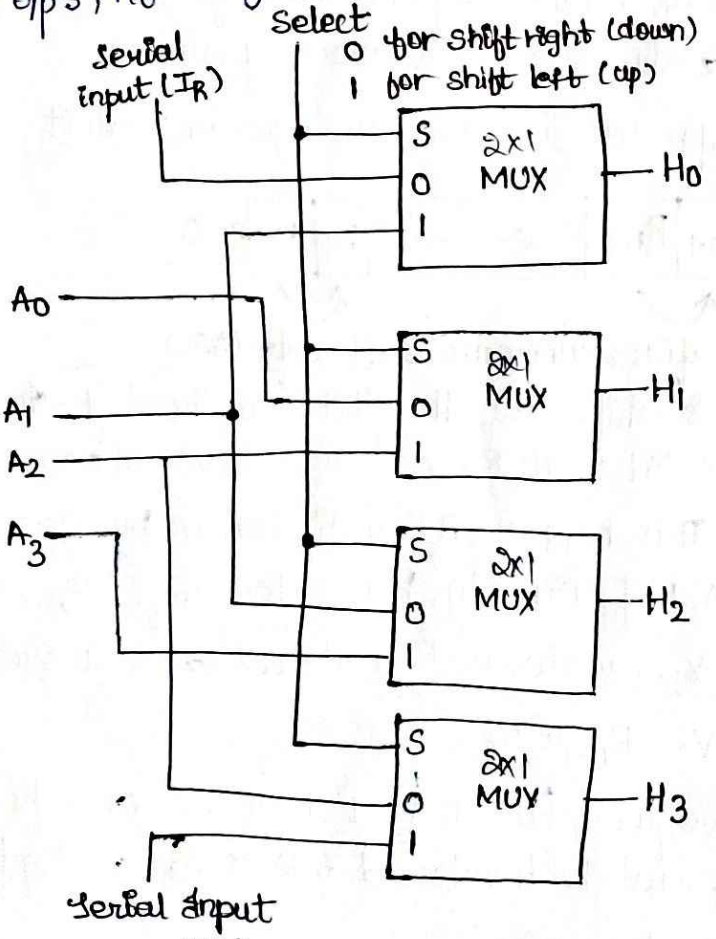
→ In this type of configuration a clock pulse is needed for loading the data into the register and another pulse is needed to initiate the shift.

→ 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 $H_3$.




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.

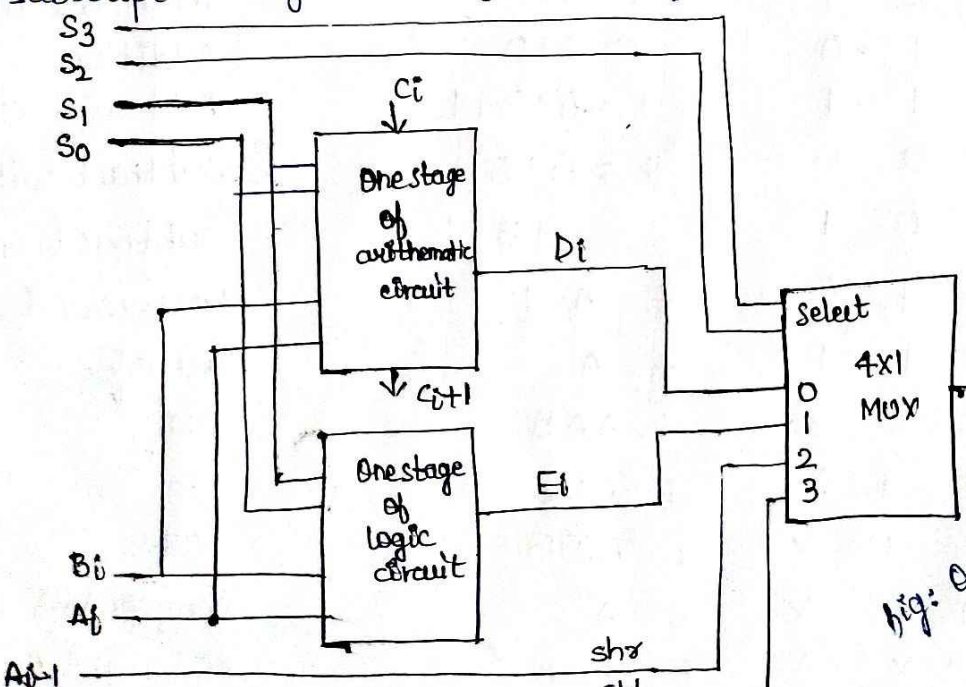



fig: One stage of arithematic logic shift unit

→ 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 |

# 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 processor 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

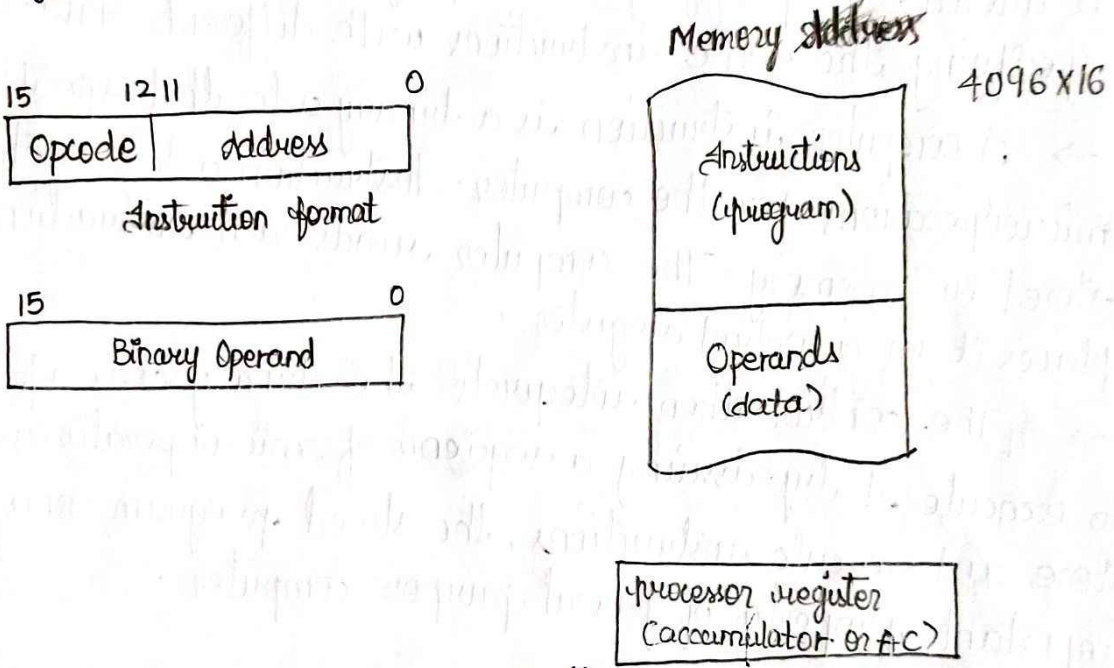



fig: stored program organization

→ 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)

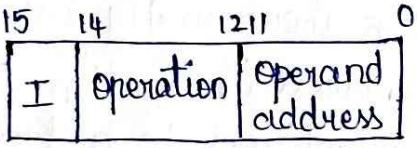
Indirect Address:-

→ 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



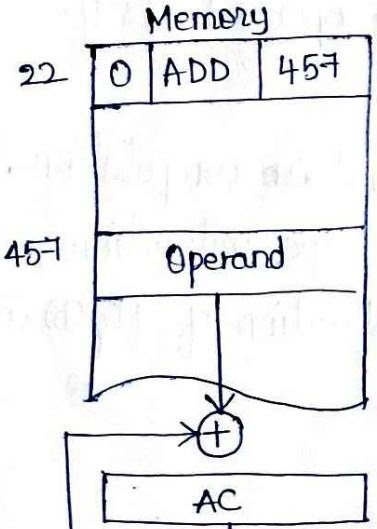

fig(a): Instruction format.

→ 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(b): Direct address.

→ as it is placed in address
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.

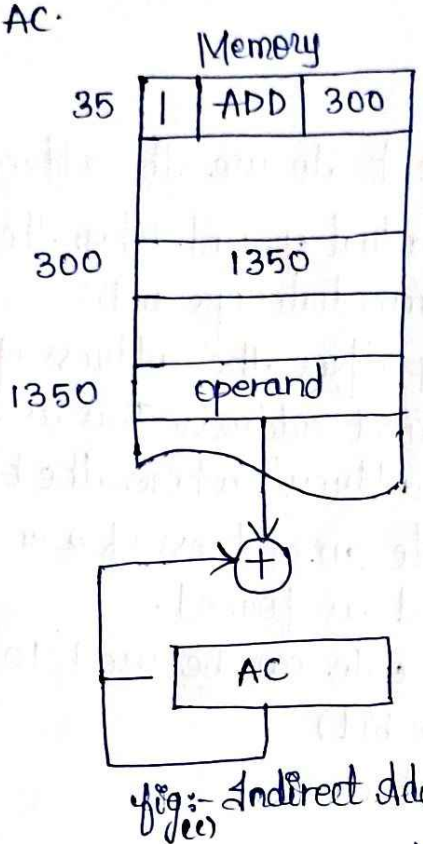



fig:- Indirect Address.

→ 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 dicatate the register configuration shown in below fig

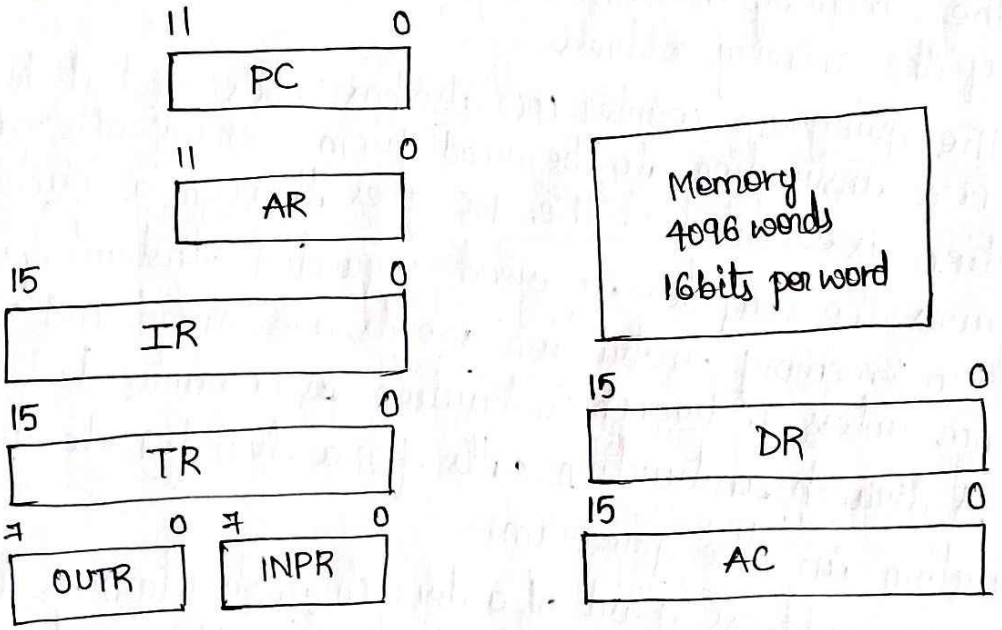



fig:- Basic computer registers and memory

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 |

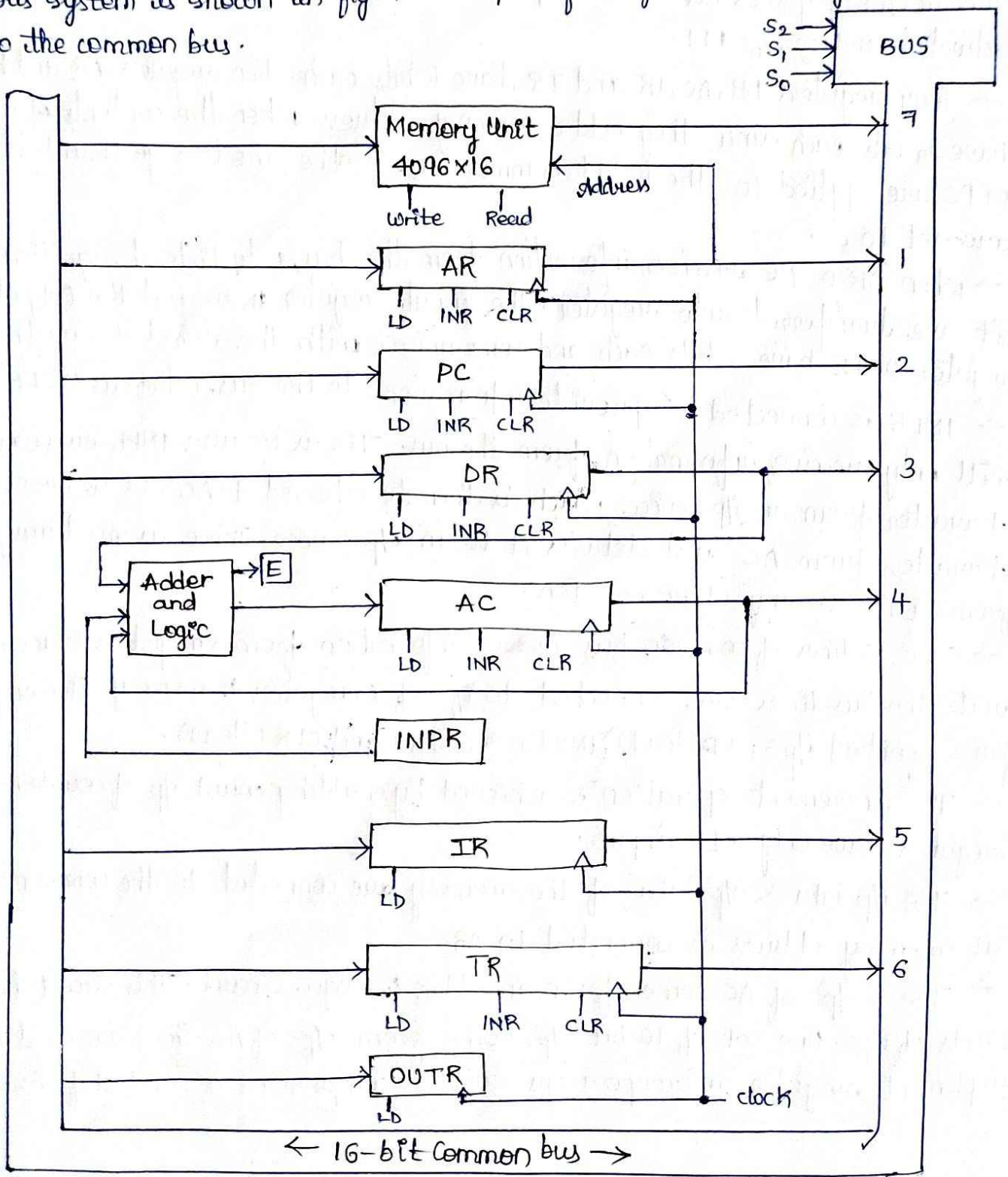
→ The register are listed in the table together with a brief description of their function and the number of bits that they contain.
→ The memory unit has a capacity of 4096 words and each word contains 16 bits. 12 bits of an instruction word are needed to specify the address of an operand.
→ This leaves 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 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, the 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

$$DR \leftarrow AC \text{ and } AC \leftarrow DR$$

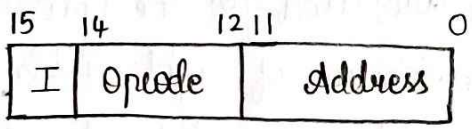
→ The two microoperations can be executed at the same time. This can be done by placing the content of AC on the bus, enabling the LD i/p of DR, transferring the content of DR through adder & logic circuit into AC and enabling the LD (load) i/p of AC, all during same clock cycle.

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

## * Computer Instructions:-

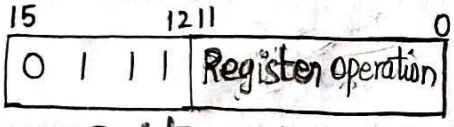
The basic computer has three instruction code formats shown in fig. Each format has 16 bits.

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.



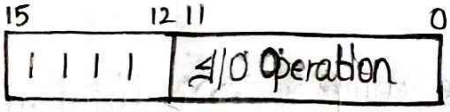

(Opcode = 000 through 110)

(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 Arithematic, 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.

→ Arithematic, 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.

An 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.

An 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. An the microprogrammed control, any required changes or modifications can be done by updating the microprogram in control memory.

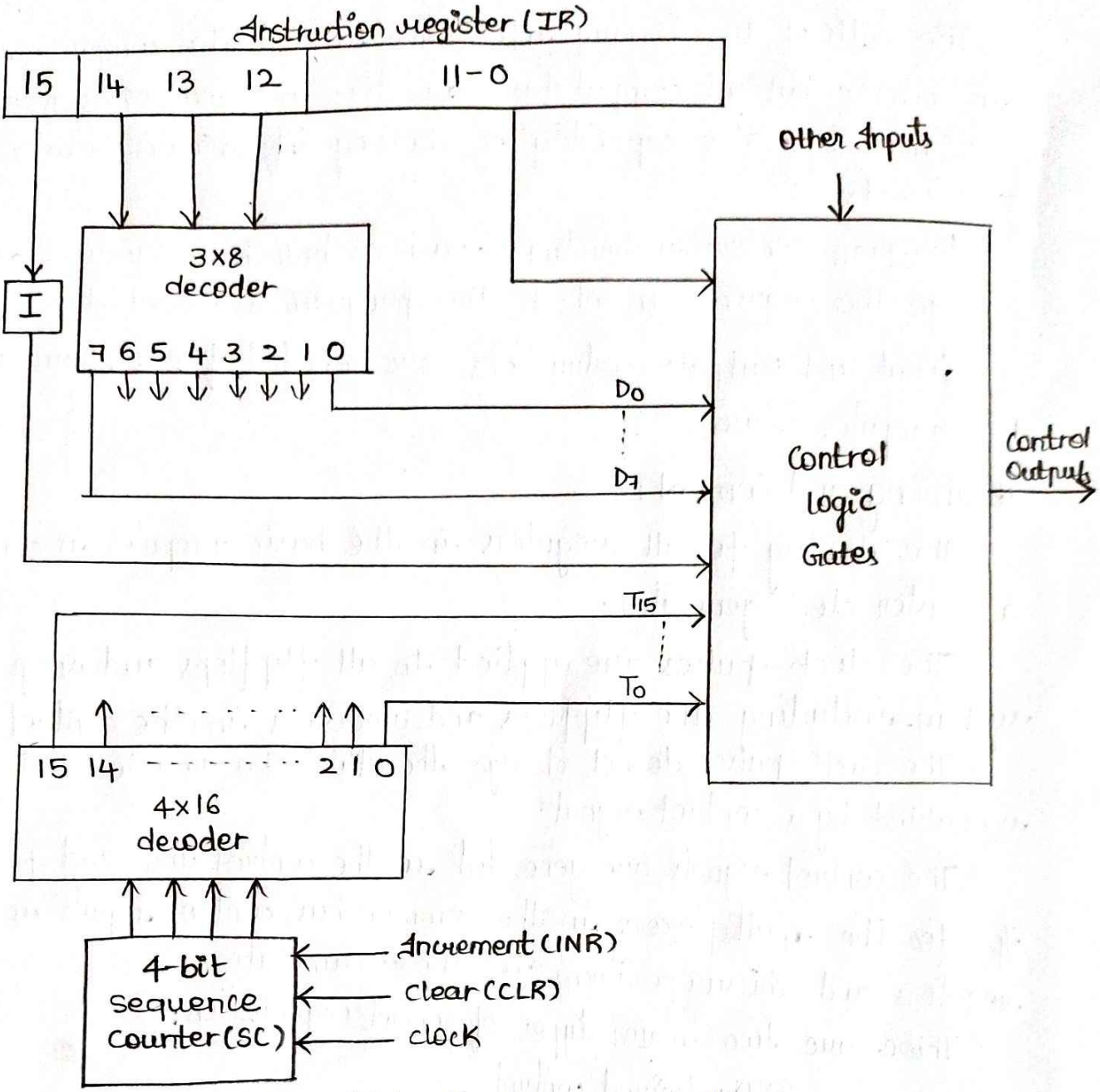



fig: Control unit of basic computer

→ 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 $D_7$. 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

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 T4, 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.

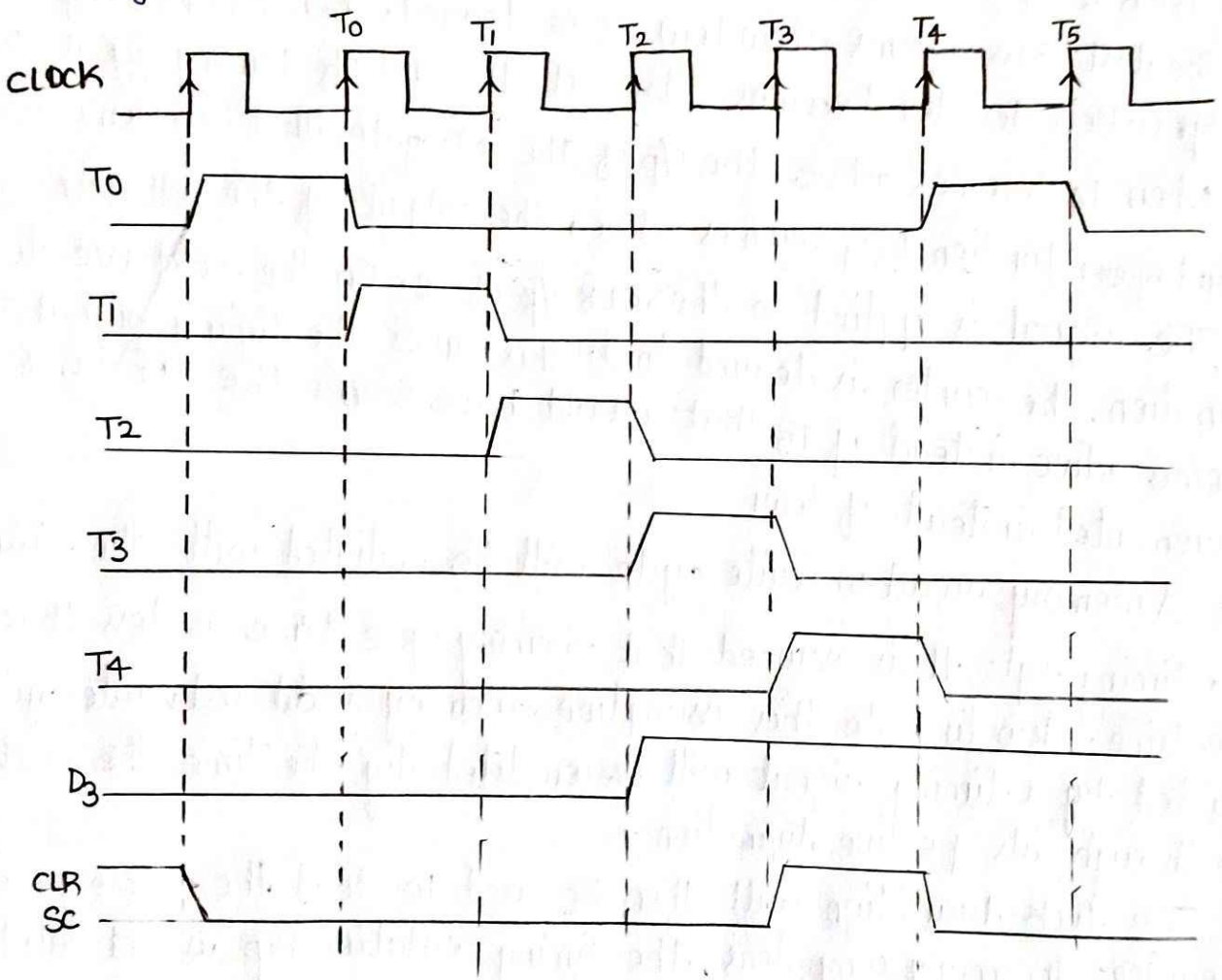



Fig:- Example of control timing signals

→ 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

$T_0 : AR \leftarrow PC$

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

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

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.

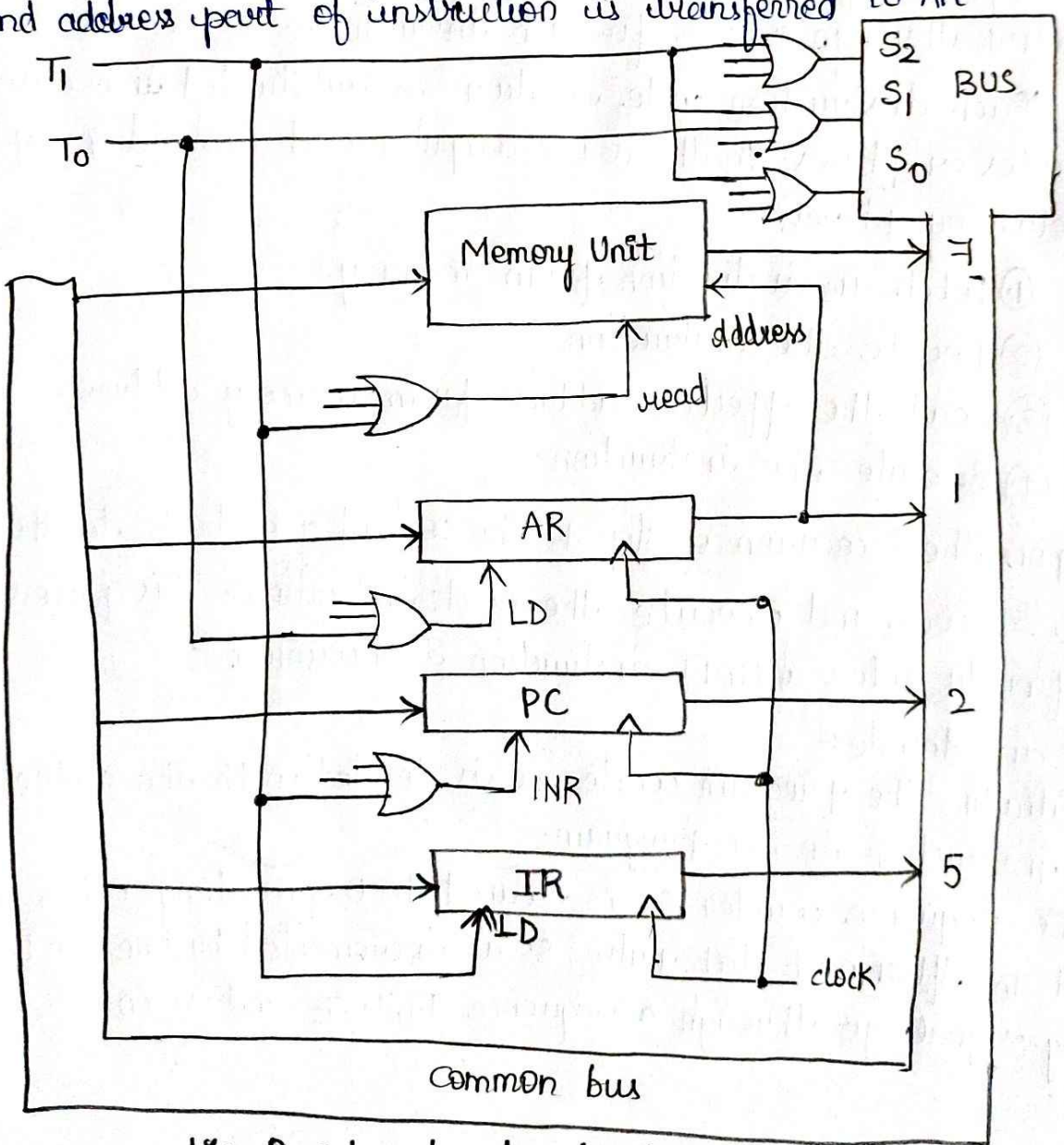



fig: Register transfers for fetch phase

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.

It 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:-

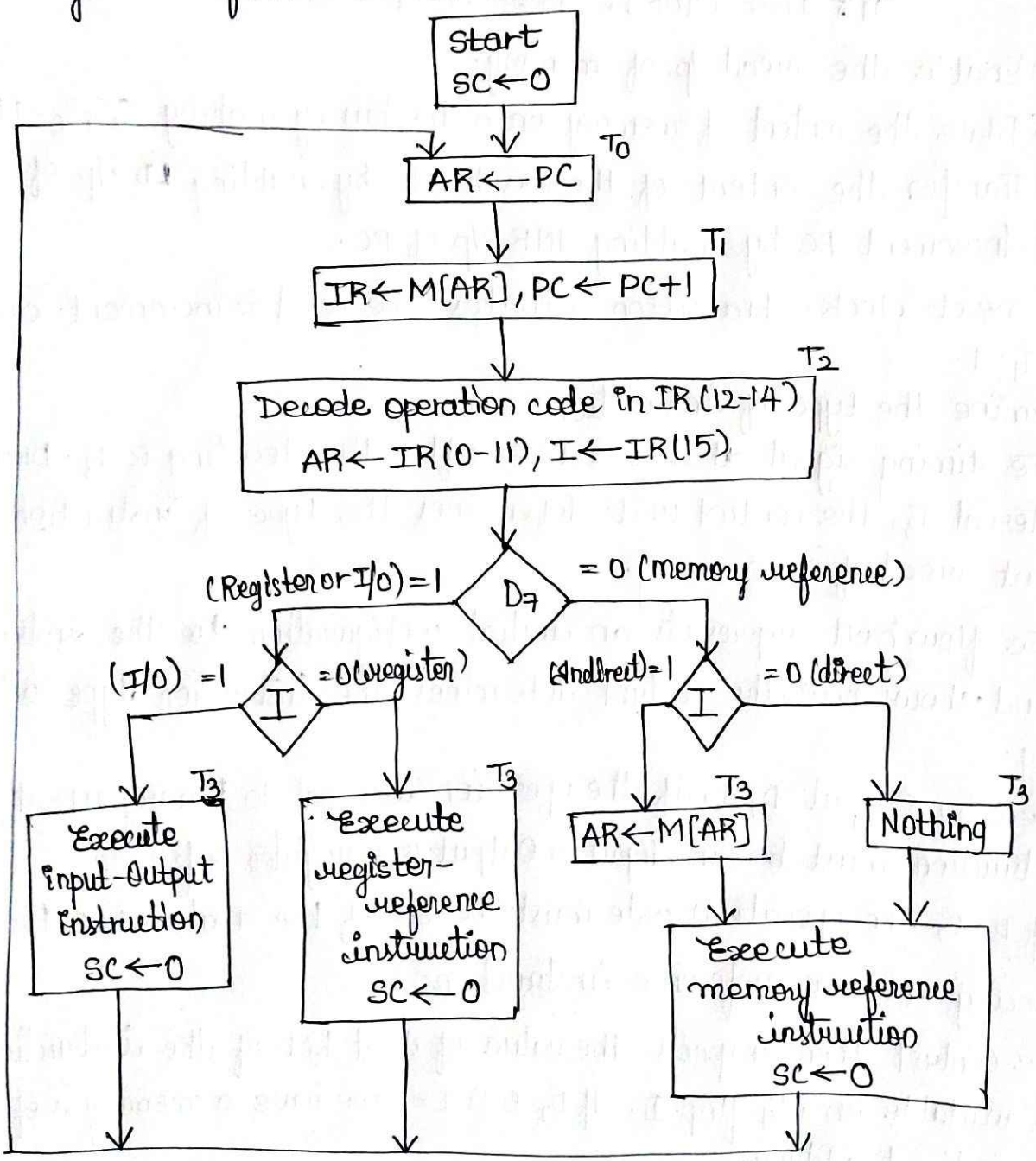



fig:- flowchart for an instruction cycle

→ 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 | $rB_7$: $AC \leftarrow shr\ AC$<br>$AC(15) \leftarrow E$<br>$E \leftarrow AC(0)$ | circular right |
| CIL | $rB_6$: $AC \leftarrow shl\ AC$<br>$AC(0) \leftarrow E$<br>$E \leftarrow AC(15)$ | circular left |
| INC | $rB_5$: $AC \leftarrow AC + 1$ | Increment AC |
| SPA | $rB_4$: If $(AC(15) = 0)$<br>then $(PC \leftarrow PC + 1)$ | skip if Positive |
| SNA | $rB_3$: If $(AC(15) = 1)$ then<br>$(PC \leftarrow PC + 1)$ | 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 | $D_6$ | $M[AR] \leftarrow M[AR]+1$<br>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 microoperations that executes the instructions are

$D_0T_4$: DR ← M[AR]

$D_0T_5$: AC ← AC ∧ DR, SC ← 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 Cout is transferred to E (Extended Accumulator) flipflop. The microoperations needed to execute this instruction are

$D_1T_4$ : DR ← M[AR]

$D_1T_5$ : AC ← AC + DR, E ← Cout, SC ← 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 ← M[AR]

$D_2T_4$ : AC ← DR, SC ← 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] ⇚ AC, SC ← 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

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

The instruction is executed with one microoperation

$D_4T_4$: PC ← AR, SC ← 0

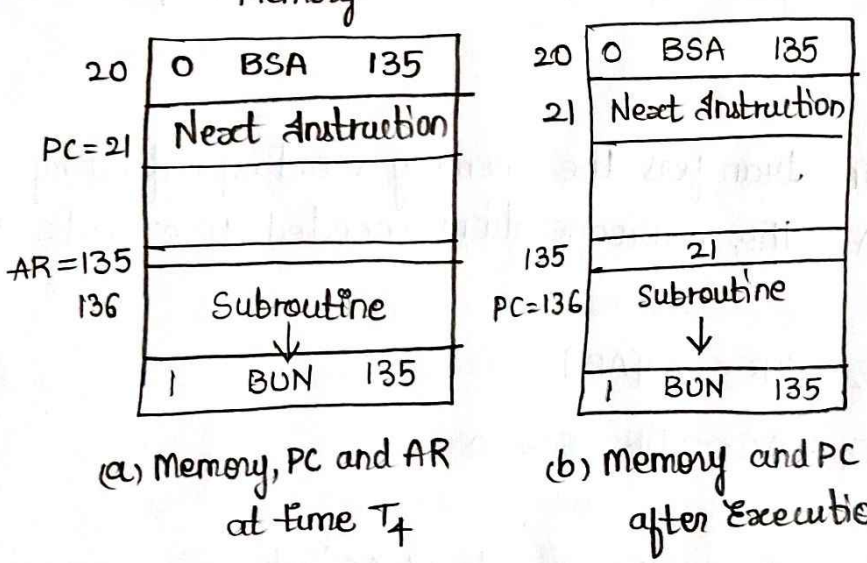
The effective address from AR is transferred through the common bus to PC. Resulting SC to 0 transfers control to $T_0$.

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] ← PC , PC ← AR+1

Eg:-




(a) Memory, PC and AR at time $T_4$

(b) Memory and PC after Execution.

M[135] ← 21, PC ← 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

~~$D_5D_4$~~ $D_5T_4$ : M[AR] ← PC, AR ← AR+1

$D_5T_5$ : PC ← AR, SC ← 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.

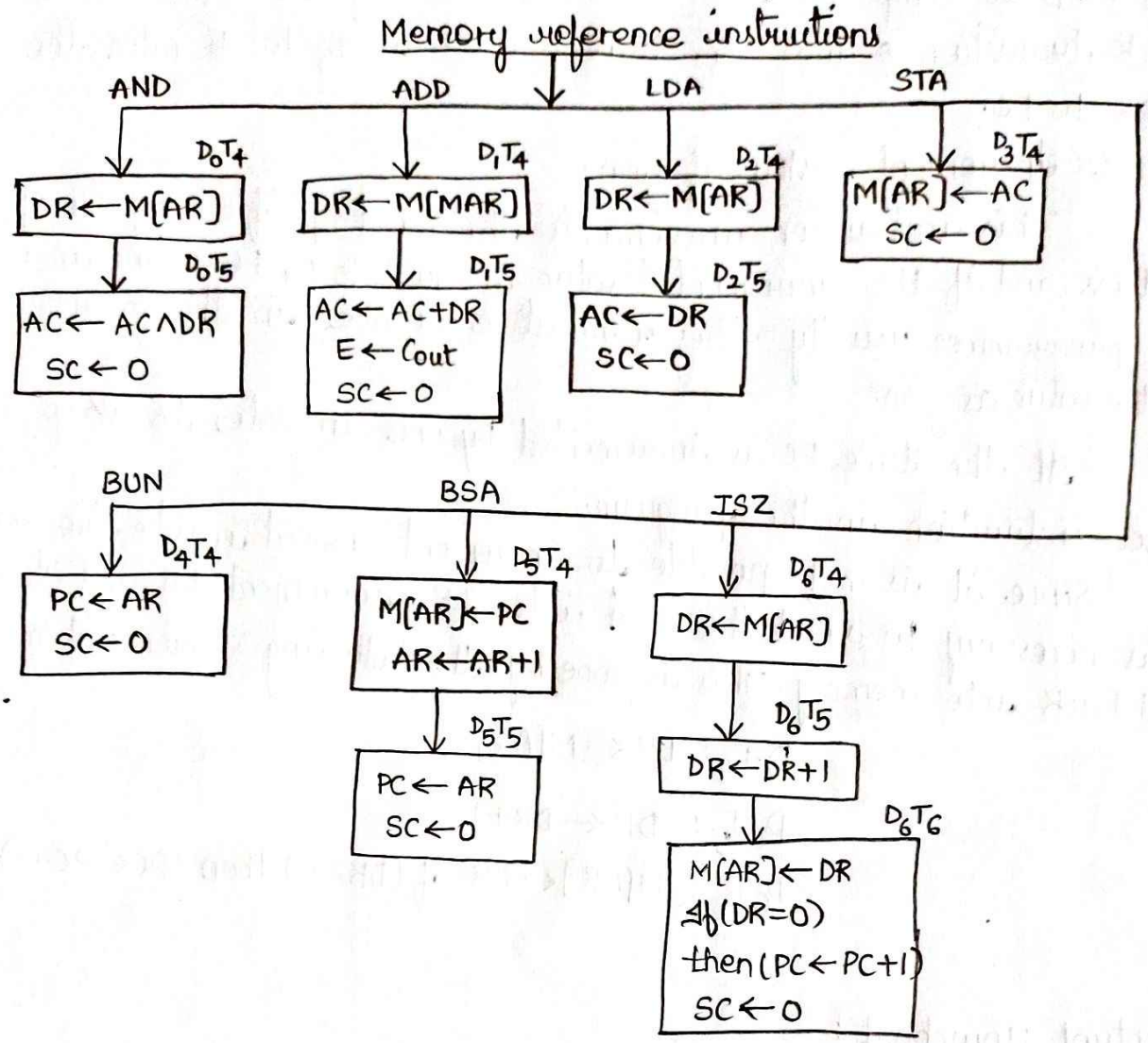



fig: flowchart for memory-reference instructions.

## * 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.

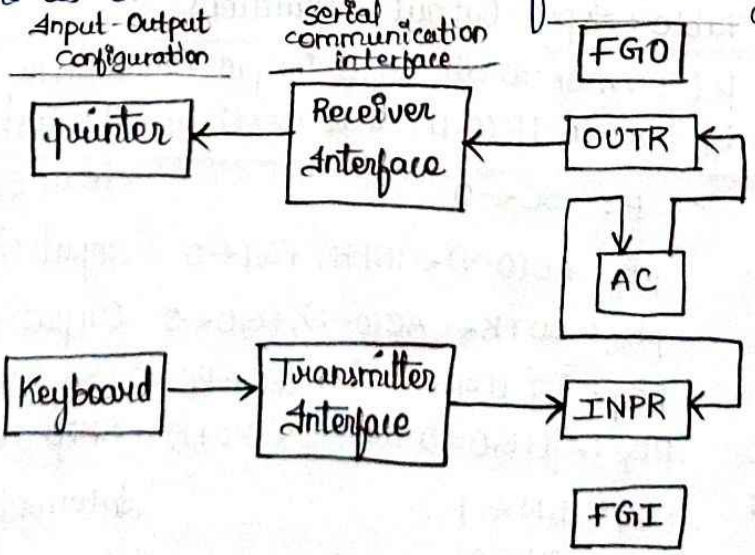



fig:- Input-Output configuration

→ 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

| $D_7 I T_3 = p$ (common to all input-Output instructions<br>$IR(i) = B_i$ [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 ICF 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.

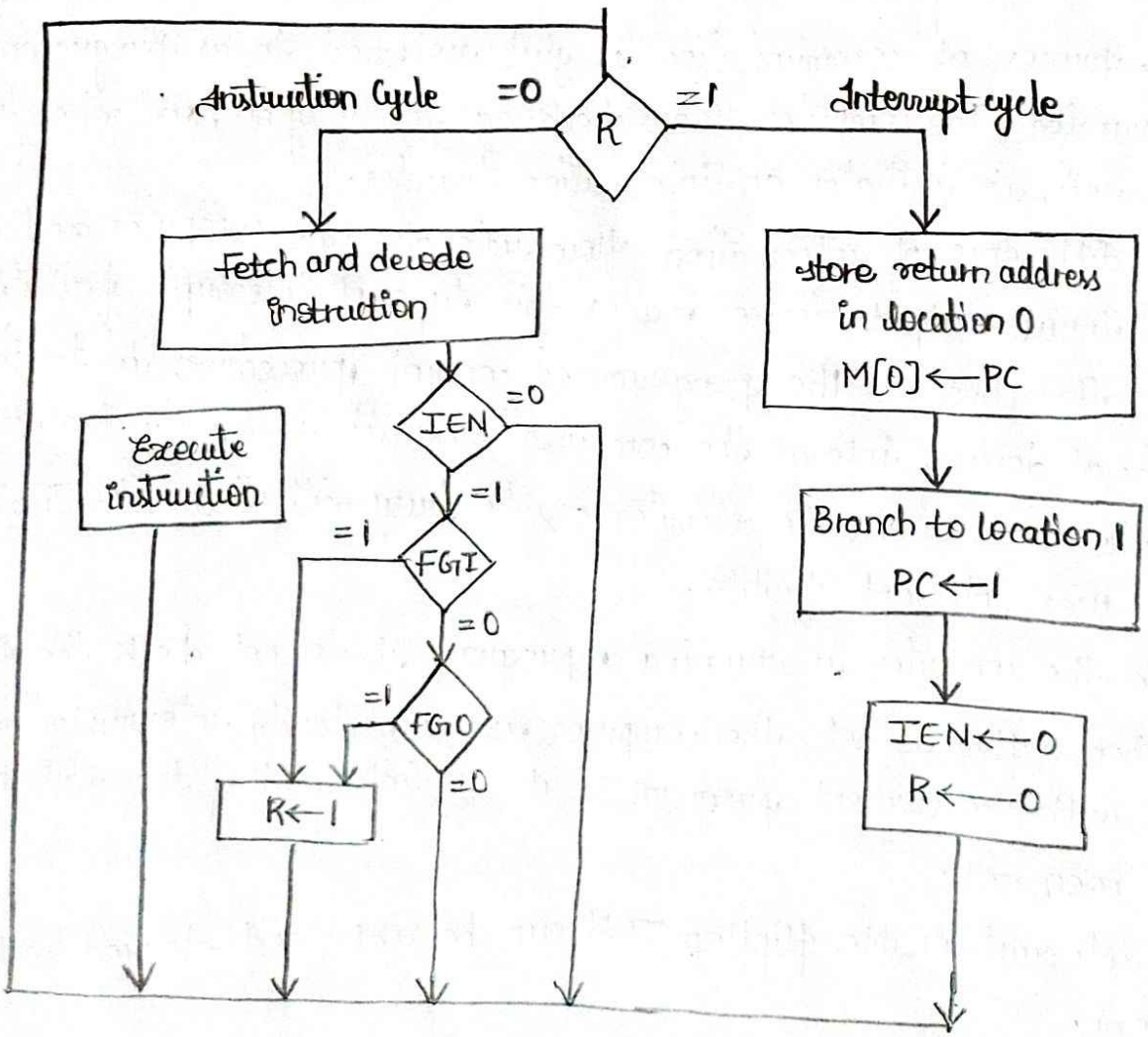
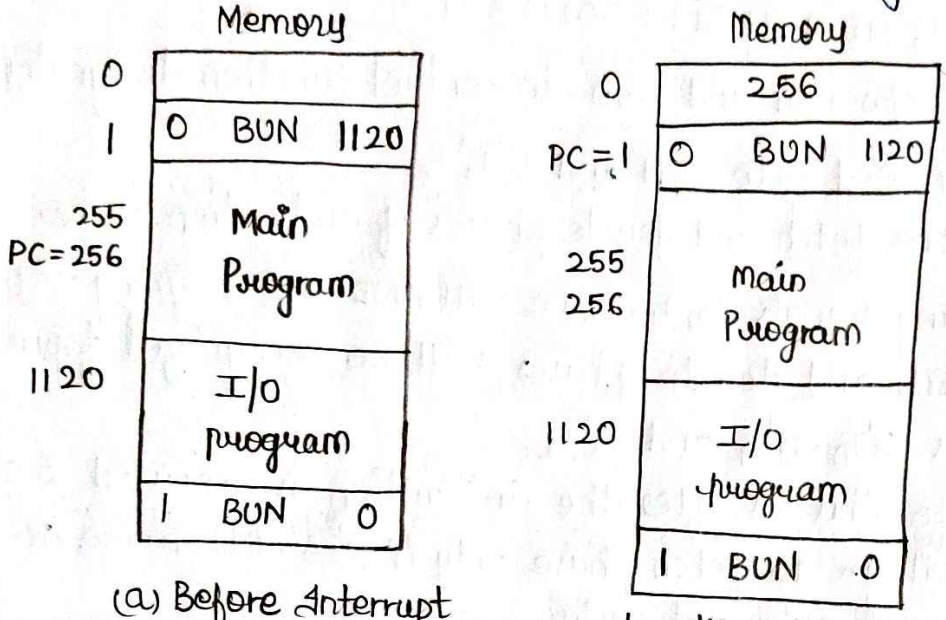



fig:- flowchart for interrupt cycle.

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.




(a) Before Interrupt (b) After 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 FG1 or FG0 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

$$T_0'T_1'T_2'(IEN)(FG1 + FG0) : R \leftarrow 1$$

→ The symbol "+" b/w FG1 and FG0 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-IV

# Micro programmed Control

# Central Processing Unit

# Computer Arithmetic

# 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:-

In 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 as of following fig.

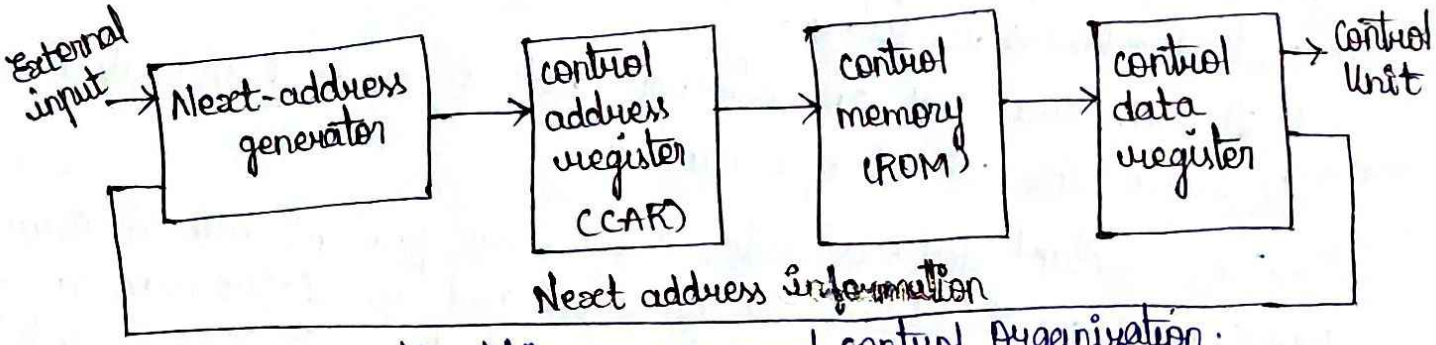



fig: Microprogrammed control organization.

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

→ The control 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 stard 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 returns.

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

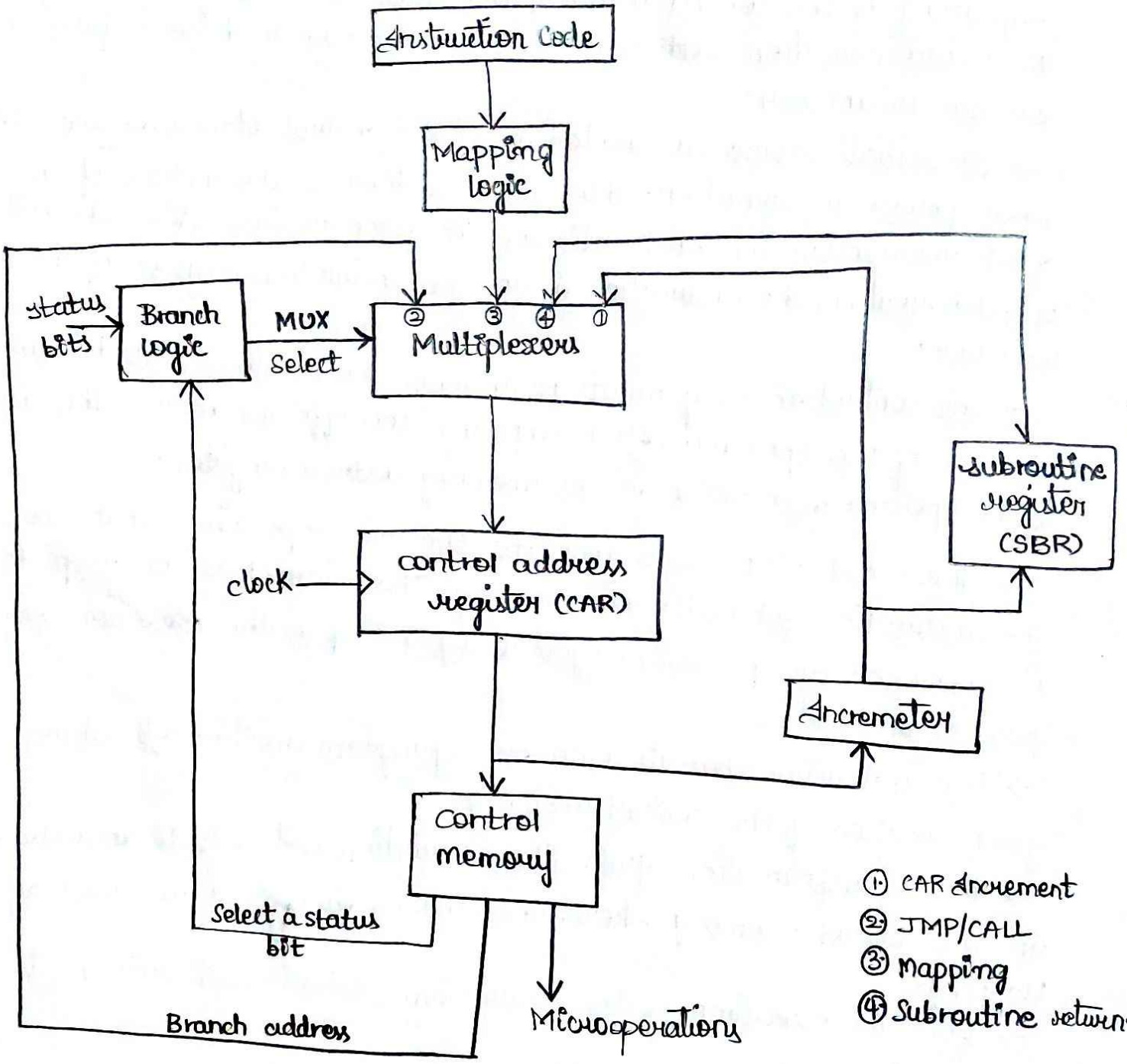



fig: Selection of address for control memory.

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 a 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.

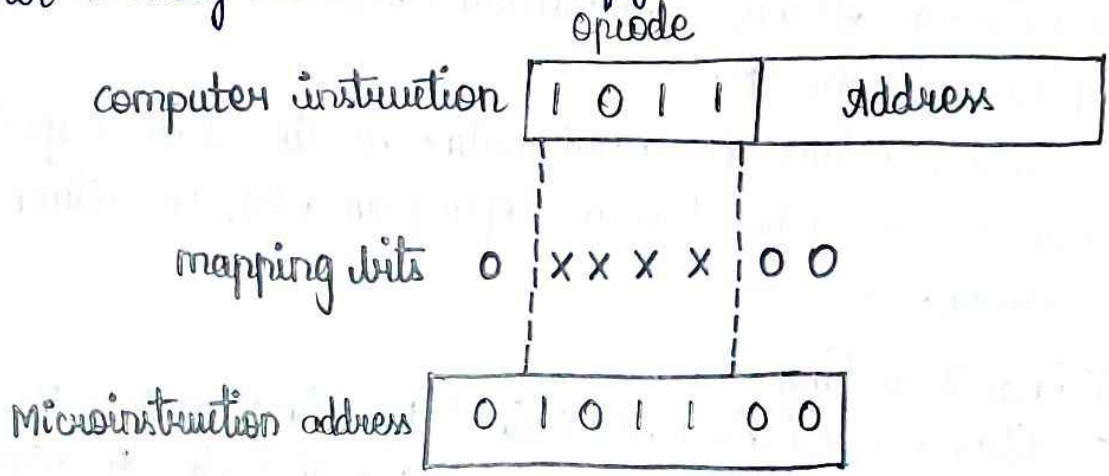
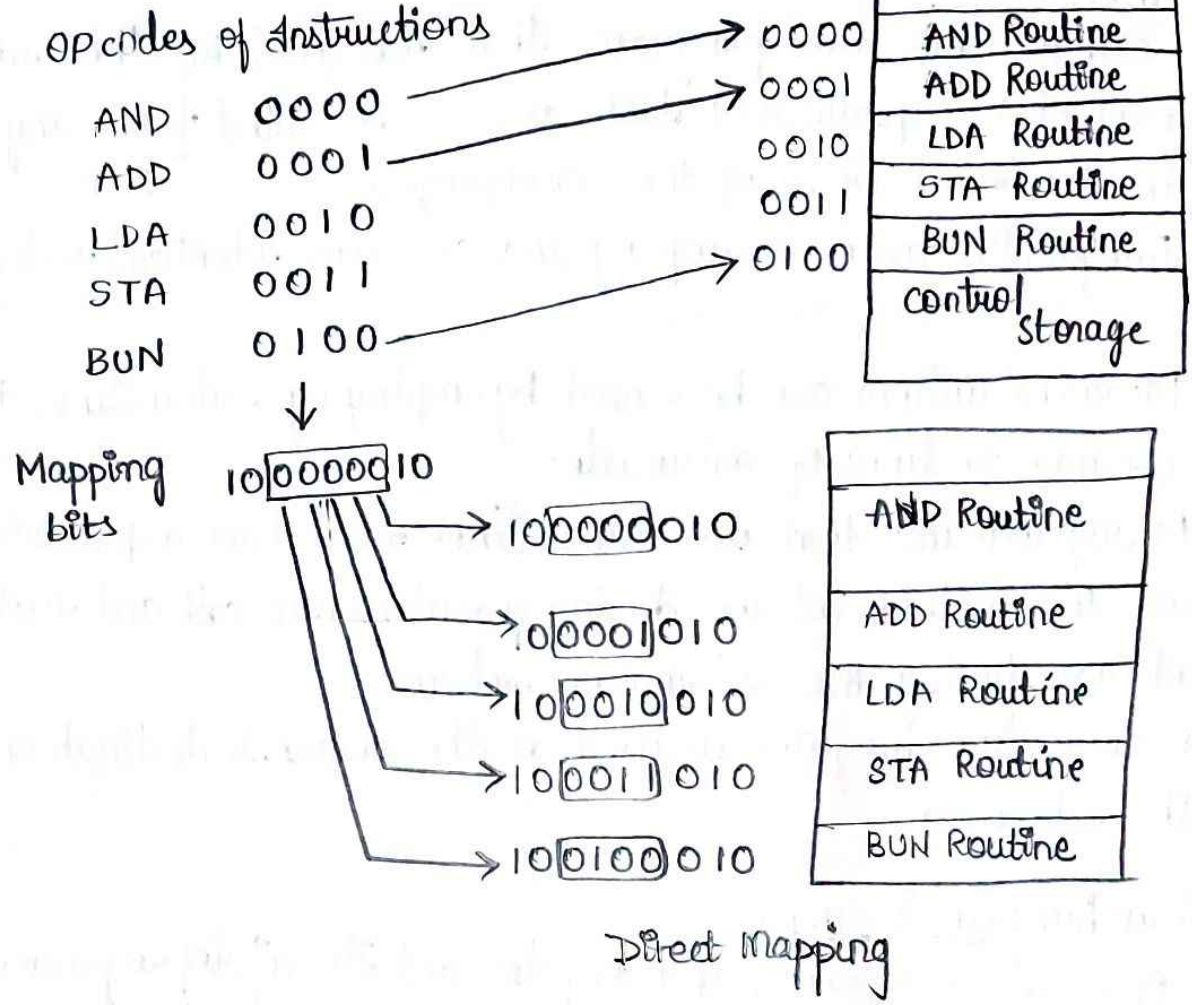



fig: Mapping from instruction code to microinstruction address.

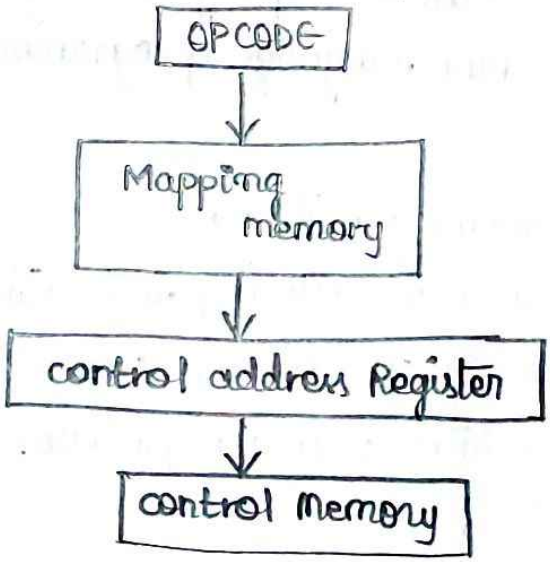
→ Mapping consists of placing a 0 in the most significant bit of the address, transferring the four operation code bits, and clearing the 2 least significant bits of the control address register.

→ 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.

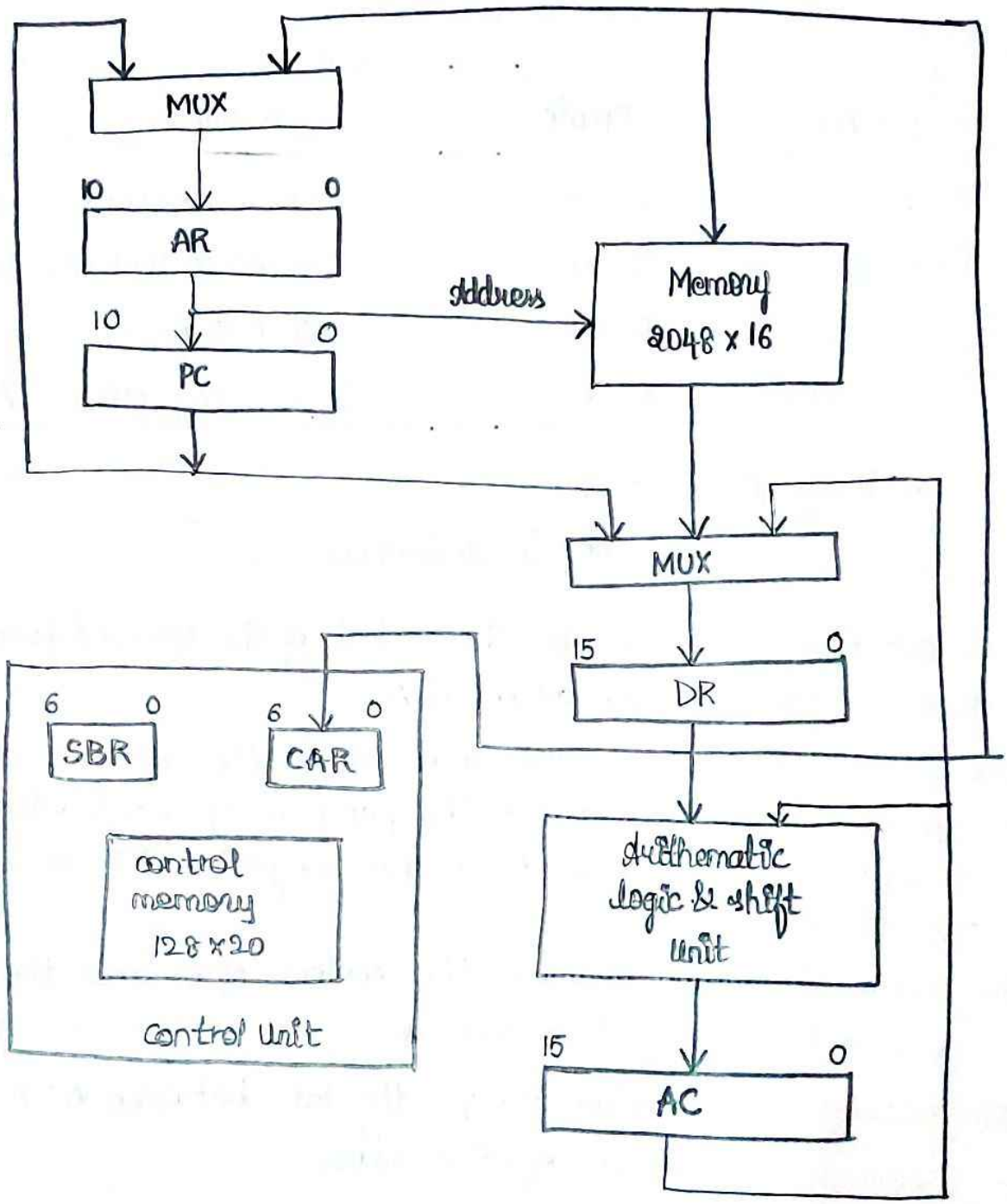



fig: Computer Hardware Configuration

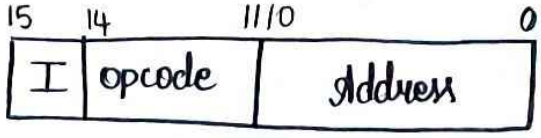
Micro instruction format :-

→ 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

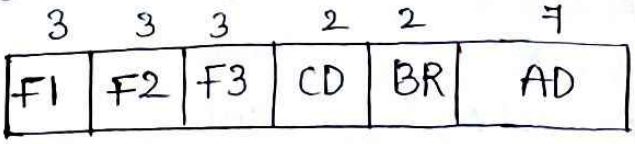
→ The ~~Add~~ ADD instruction adds the content of the operand found in the effective address to the content of AC.

→ 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 ← AC ⊕ DR | XOR |
| 010 | AC ← $\overline{AC}$ | 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<br>CAR ← CAR + 1 if condition = 0 |
| 01 | CALL | CAR ← AD, SBR ← CAR + 1 if condition = 1<br>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 [illegible] 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 0xxxxx00, 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 | | | |
| ADD: | NOP | I | CALL | INDRCT |
| | 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~~ 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 & 1's 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 signal |

| | |
|---|---|
| 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.

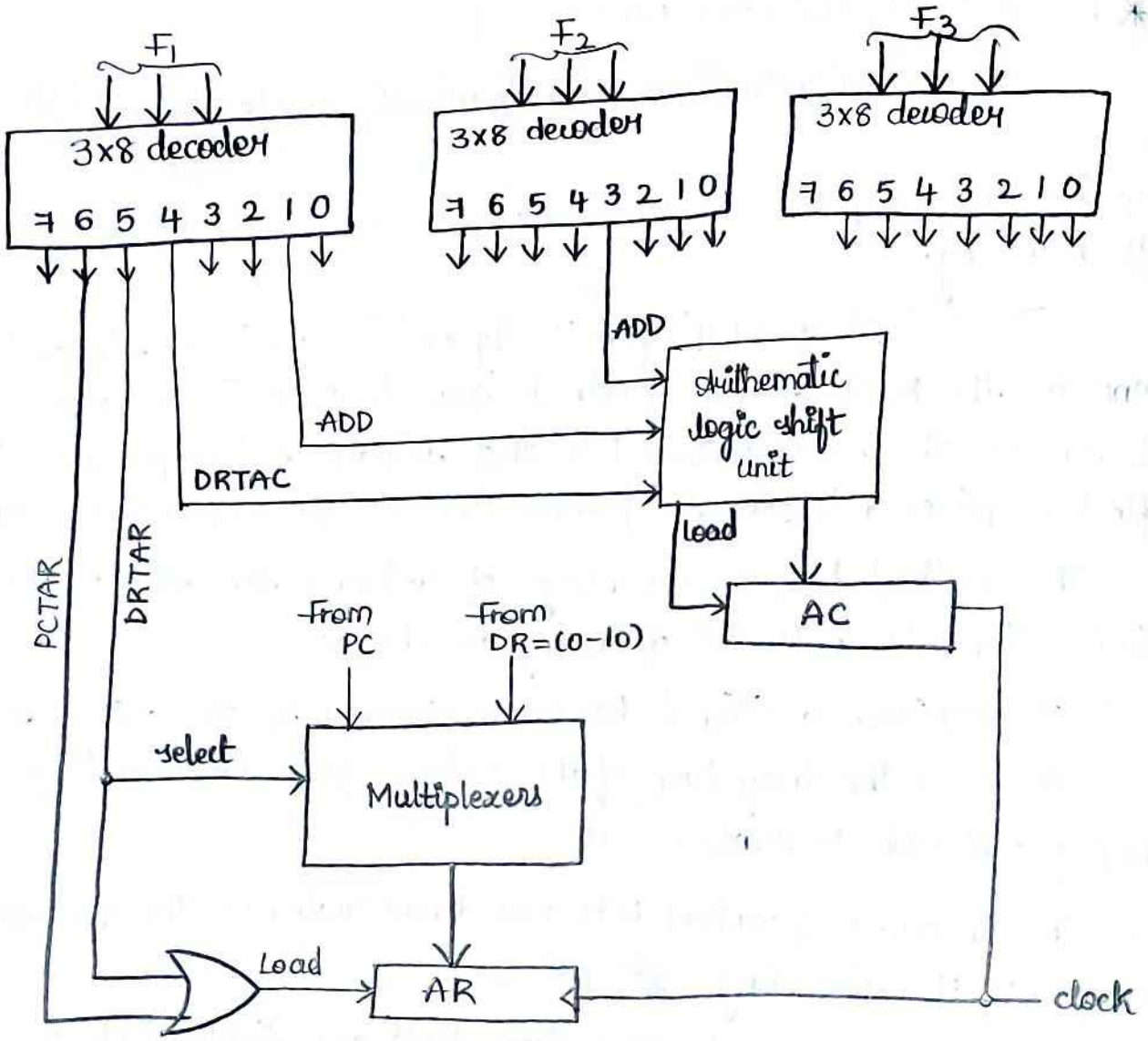



fig: Decoding of microoperation fields

→ 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.

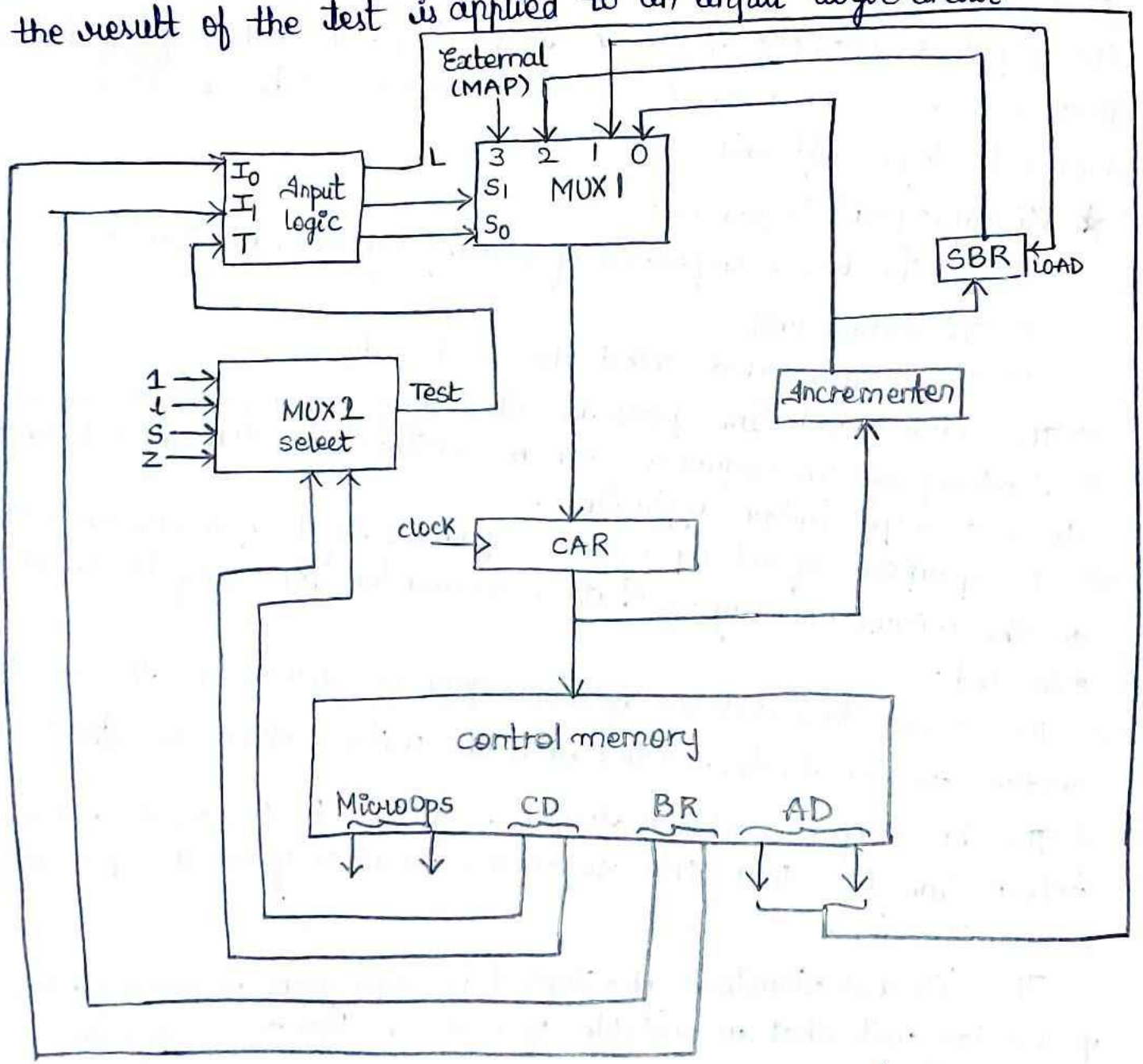



fig:- Microprogram Sequencer for a control memory

→ 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:

$$S_1 = I_1$$

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

$$L = I_1' T I_0.$$

# 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.

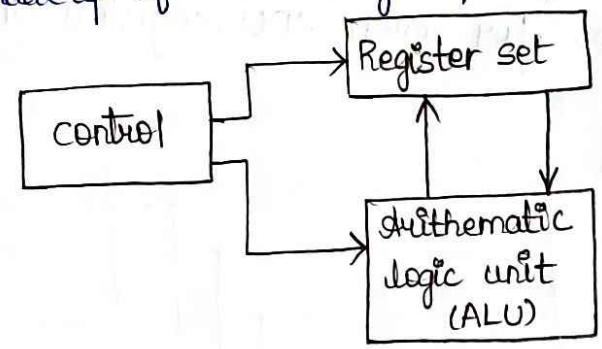



fig: Major Components of CPU

(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.

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.

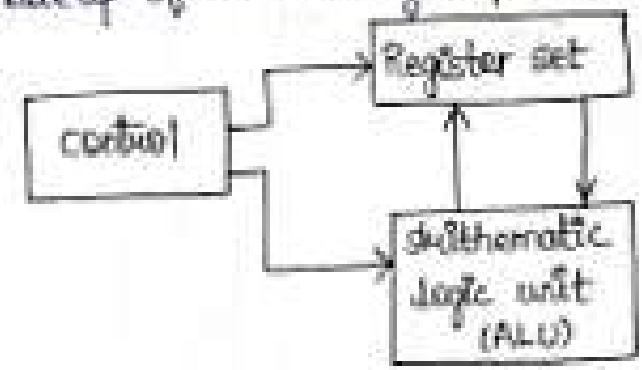



fig: Major Components of CPU

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

(ii) The arithmetic 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 format, 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.

→ 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 7 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 $AC \leftarrow AC + M[AC]$

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 & 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 performe -ed, the addressing mode of the instruction & the location of operands. The computer then executes the instruction and returns to step 1 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 (registers 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

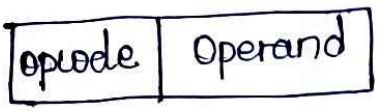
CMA : Take complement of content of AC

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.

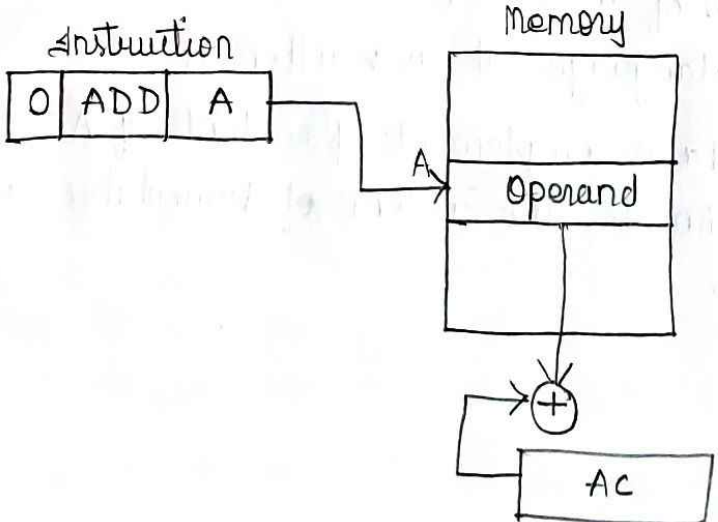
This mode is very useful for initialising the register to a constant value.

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

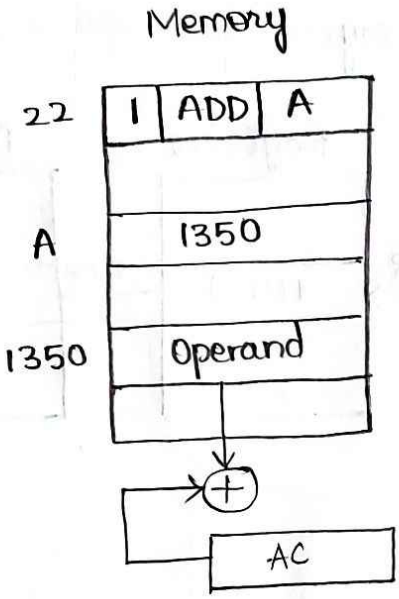
For example, ADD A → means add contents of cell A to accum -ulator.



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



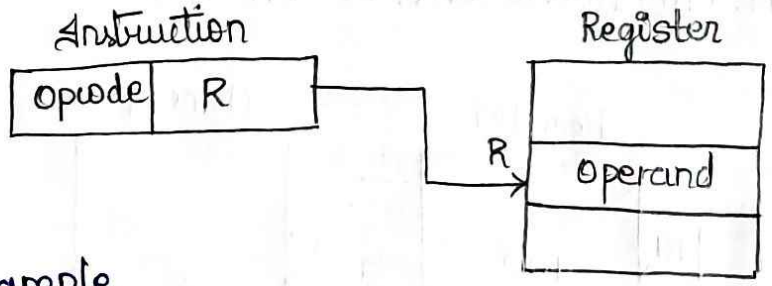

Thus AC ← M[M[A]]

(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

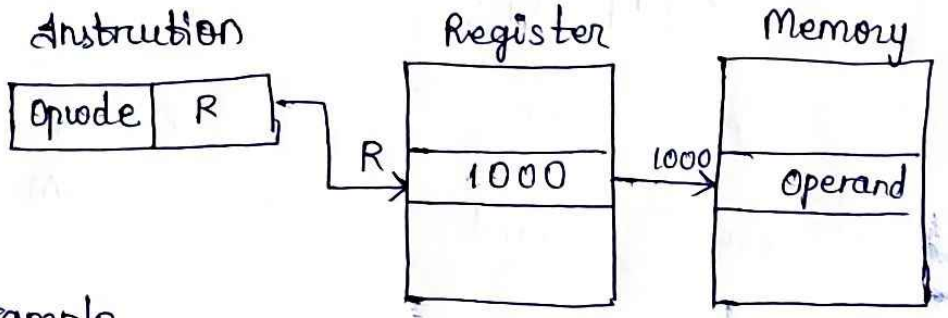
Here, AX, BX are used as register names of each of 16-bit register.

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

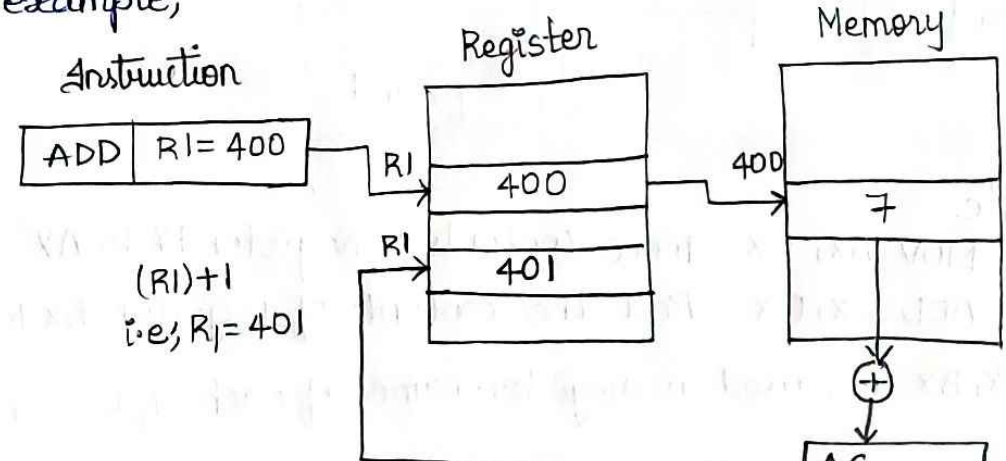
MOV 1000H, operand

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,



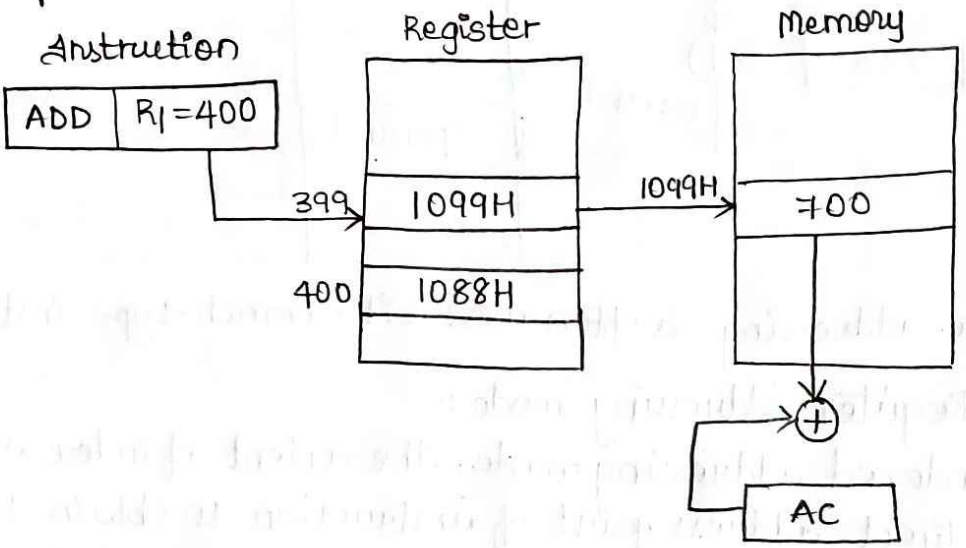
Here, the effective address is (R) = 400 and Operand in AC = 7. After loading $R_1$ is incremented by 1. It becomes 401.

In the autoinvrement mode the $R_1$ is invremente to 401 after execution of instruction

Auto-devrement addressing mode:-

Auto devrement addressing mode is reverse of auto invrement, 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-devrement mode, the register $R_1$ is devremented 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 instru-ction is added to implicitly reference register program counter to obtain effective address

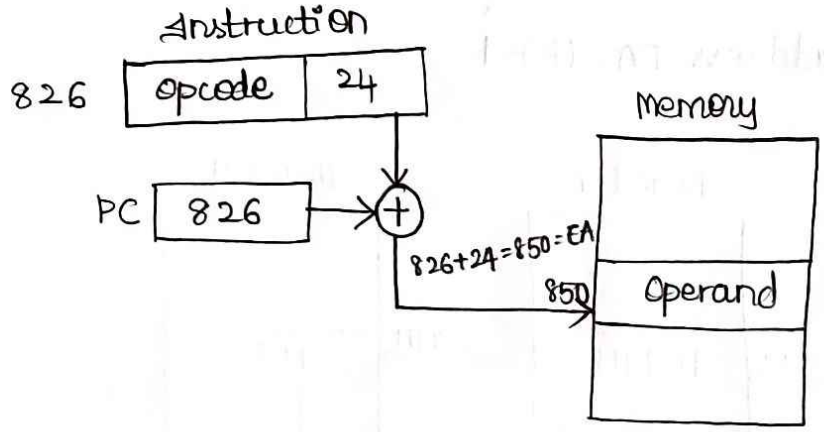
EA = A + PC

for example,

Assume that PC contains a no: 825 and the address part 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



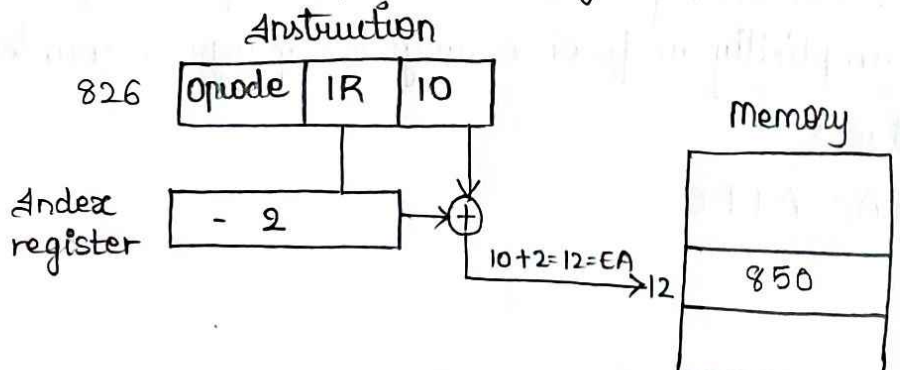

Relative addressing is often used with branch type instruction.

⑨ Index Register Addressing mode :-

In 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.

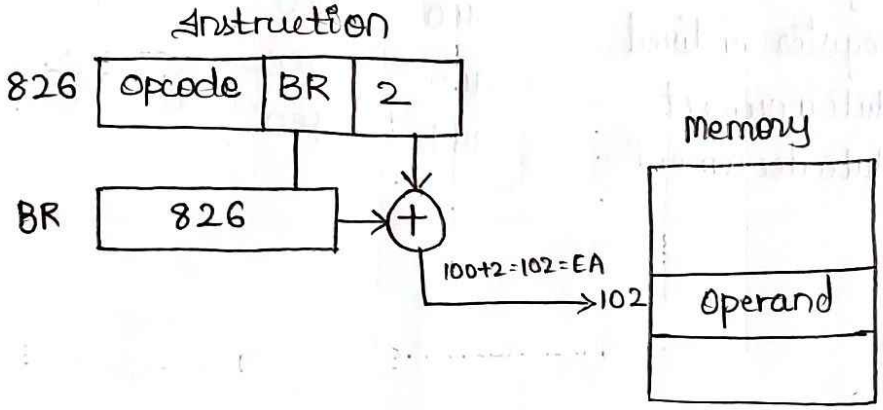
Thus in index addressing mode EA = A + Index



(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

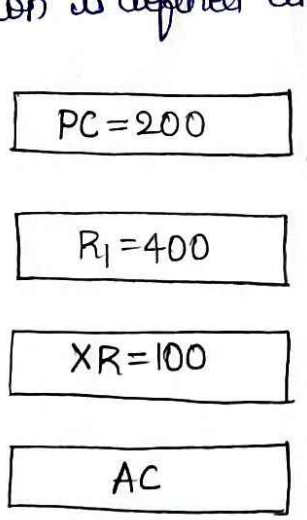
EA = A + Base.




:-

Numerical Example

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



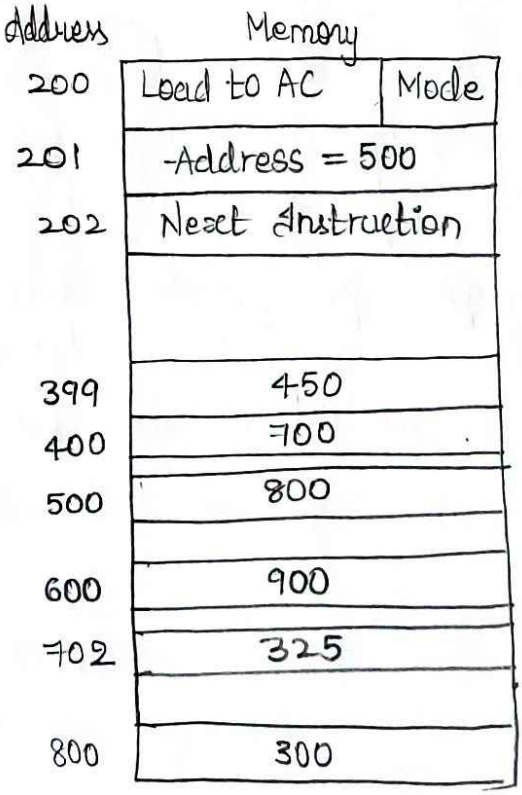




fig: Numerical

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 mnemonic symbol. It must be realized that different computers use different mnemonics for the same instruction name.

→ The load instruction is used to transfer for 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. Arithematic instructions
2. Logical and bit manipulation instructions
3. shift instructions.

1. Arithematic Instructions :-

→ The four basic arithematic 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 |

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 increm-ented 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.

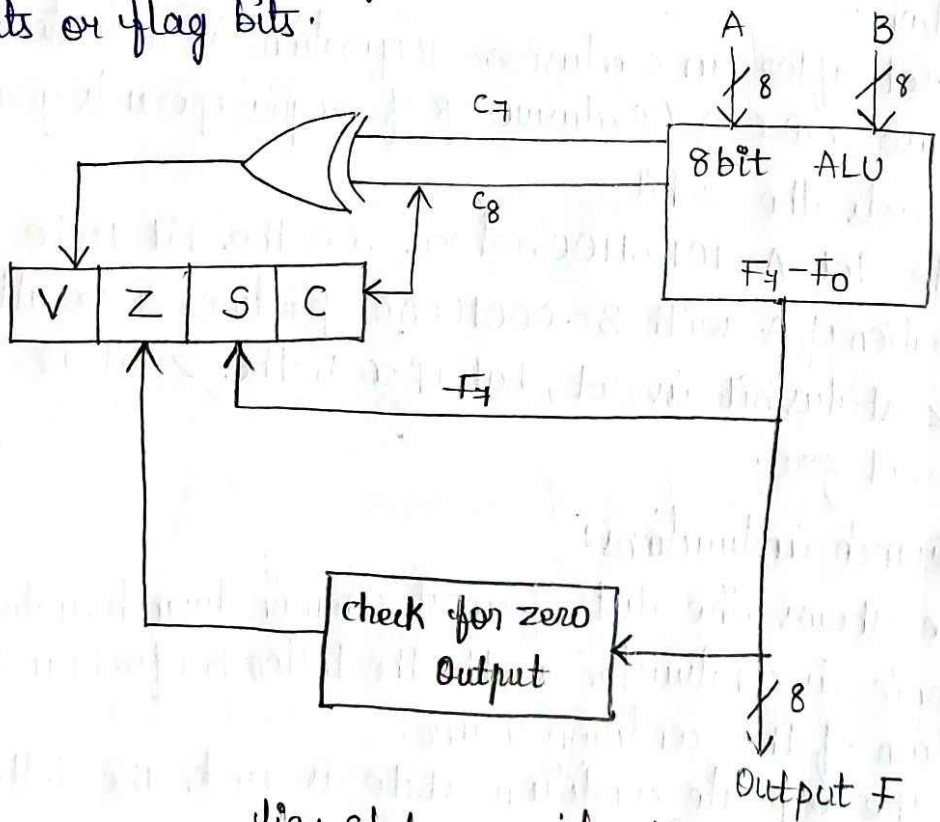



fig: Status register bits.

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=1011x1100, where x is the bit to be checked. The AND operation of A with B=00010000 produces a result 000x0000 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 $A \geq B$ is complement of $A < B$ and $A \leq B$ 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.

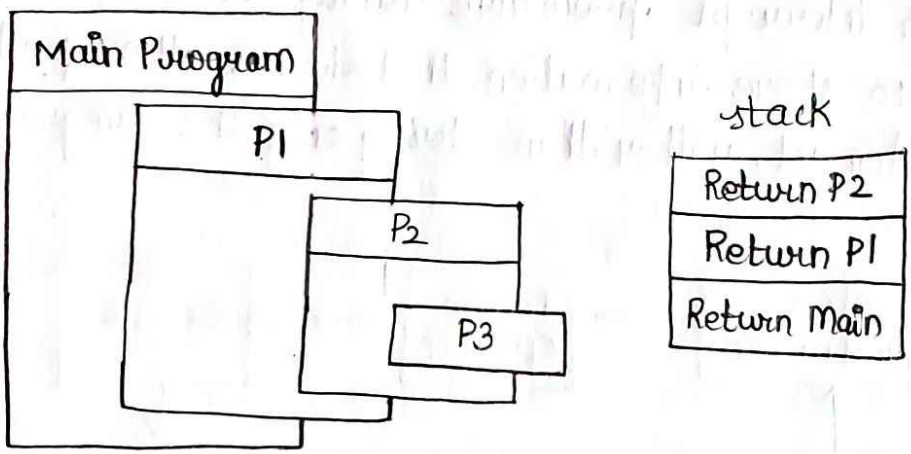
★ Subroutine Call and Return:-

→ 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.

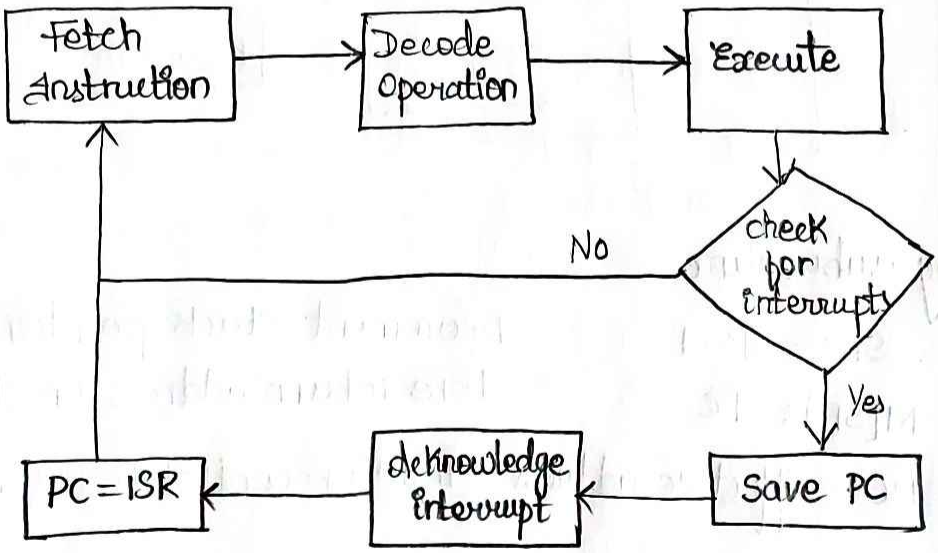
→ The procedure for servicing the interrupt in this case is called the interrupt service routine (ISR).

→ 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.



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 every day 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 parentheses 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 occurred.

→ ~~The result of an overflow on the sum of two signed 2's complement number~~

→ 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.

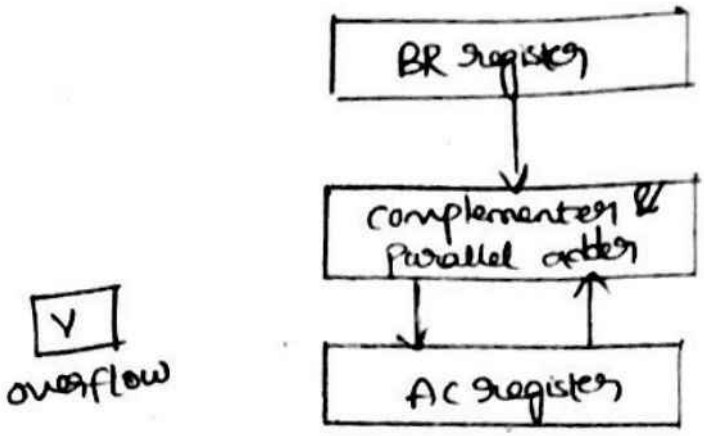



Fig 10.3 : Hardware for signed 2's complement addition & subtraction

→ 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 order.

→ 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.

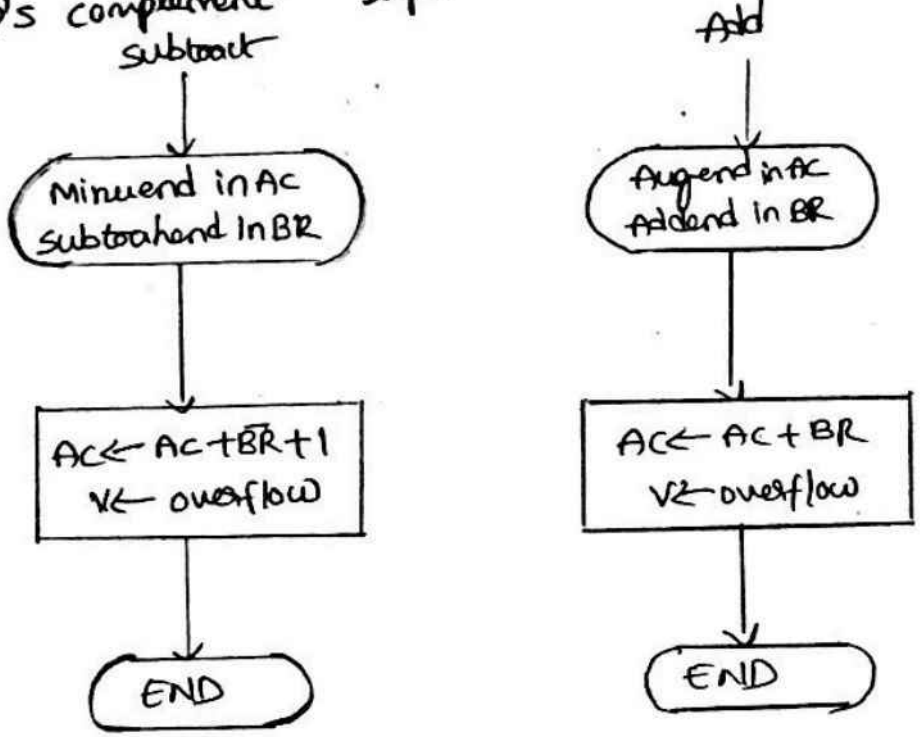



fig 10.4 Algorithm for adding & subtracting numbers in signed 2's complement representation.

→ 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:

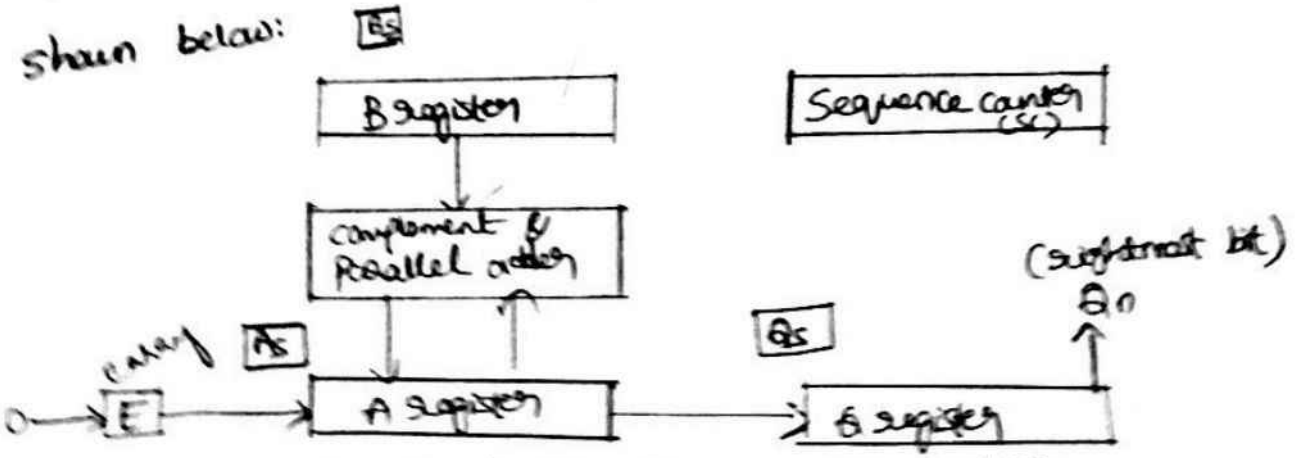


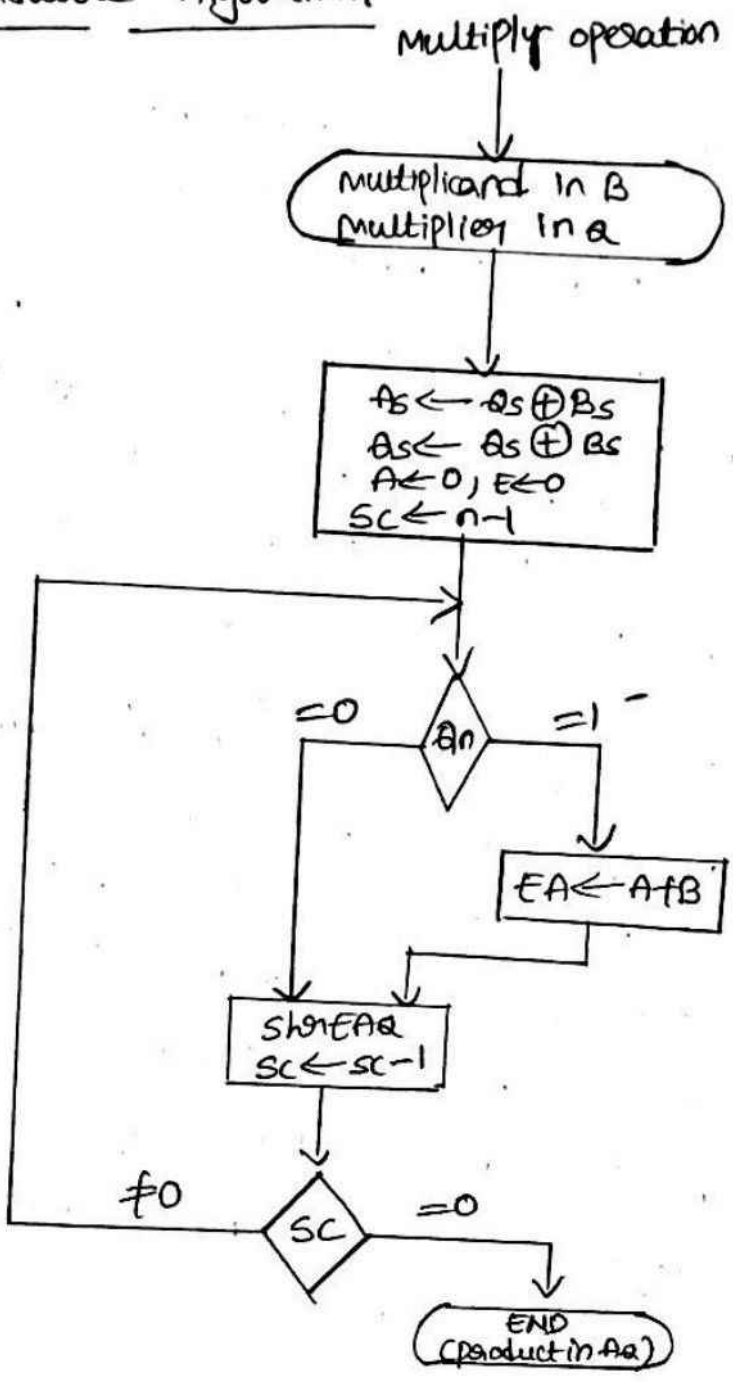

Fig 10.5 Hardware for multiply operation.

-> 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 $A_s$

$B_s$ 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 digital 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 ~~prev~~ 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 $Q_{n+1}$). 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.

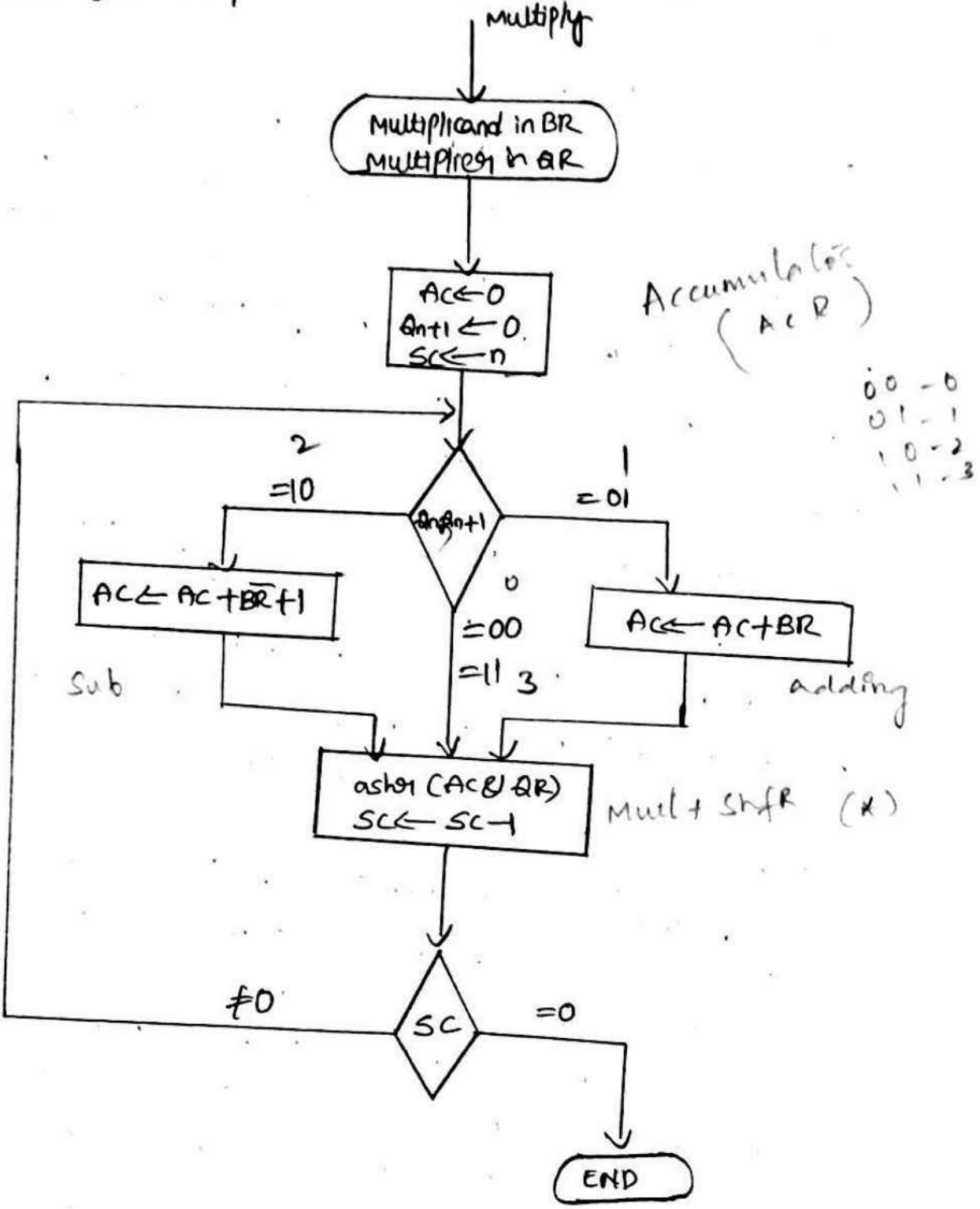



Fig 10-8 Booth algorithm for multiplication of signed-2's complement numbers.

**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 :-**

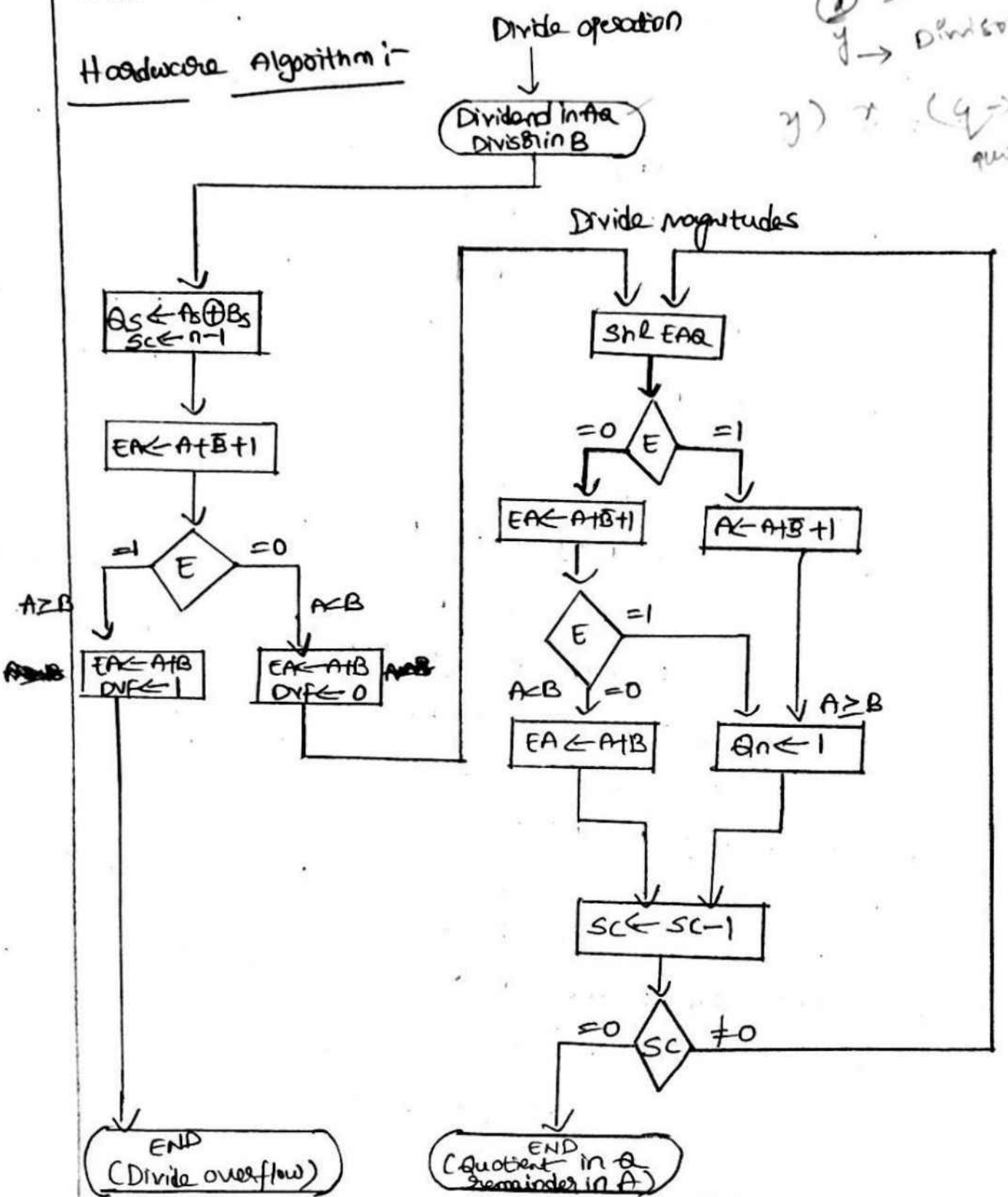



Fig 10.13 Flowchart for divide operation

→ 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 an 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 results in

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

→ The carry from this addition is not transferred to E if we want E to remain a '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, An 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 An.

→ This process is repeated again with register A holding the partial remainder. After n-1 times, the quotient magnitude is formed in register A & the remainder is found in register.

## Floating Point Arithmetic operations :-

**Basic considerations:-** A floating point number in computer registers consists of two parts: a mantissa m & an exponent e.

$$m \times r^e$$

→ 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.

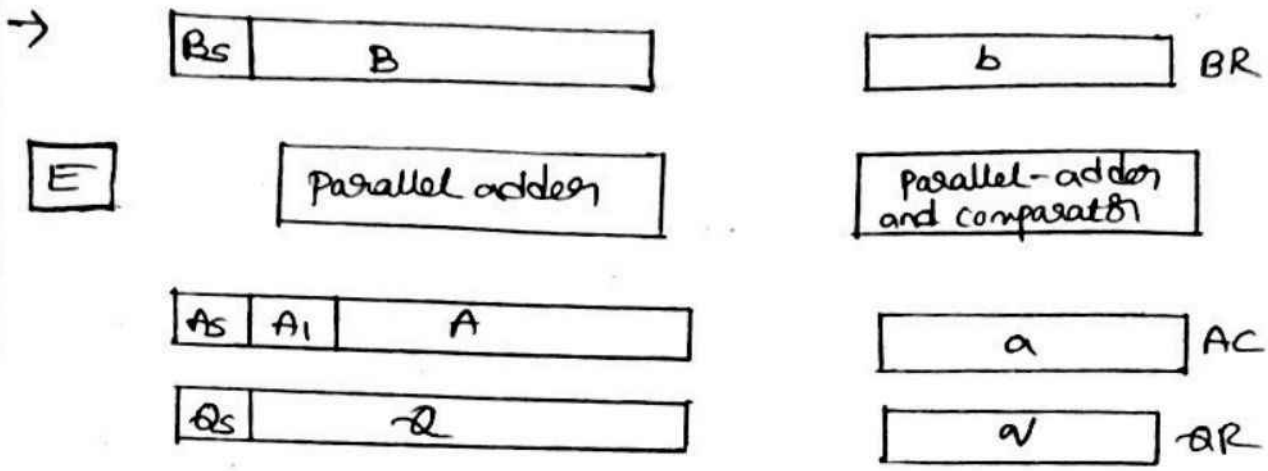



Fig 10.14 Registers for floating-point arithmetic operations.

→ 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 baised 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 the 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.

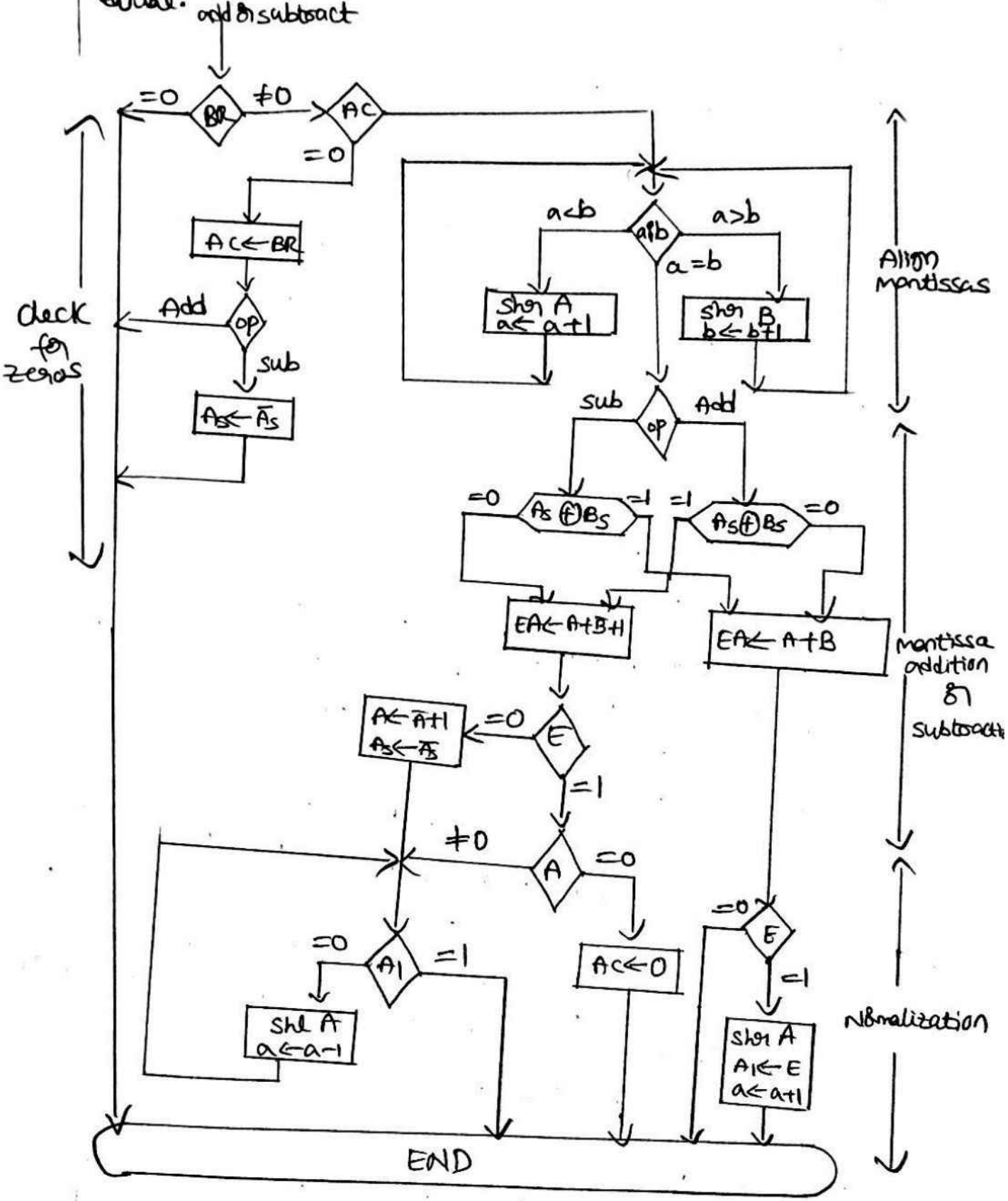



Fig 10.15 Addition & subtraction of floating-point numbers.

→ 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 comparision 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.

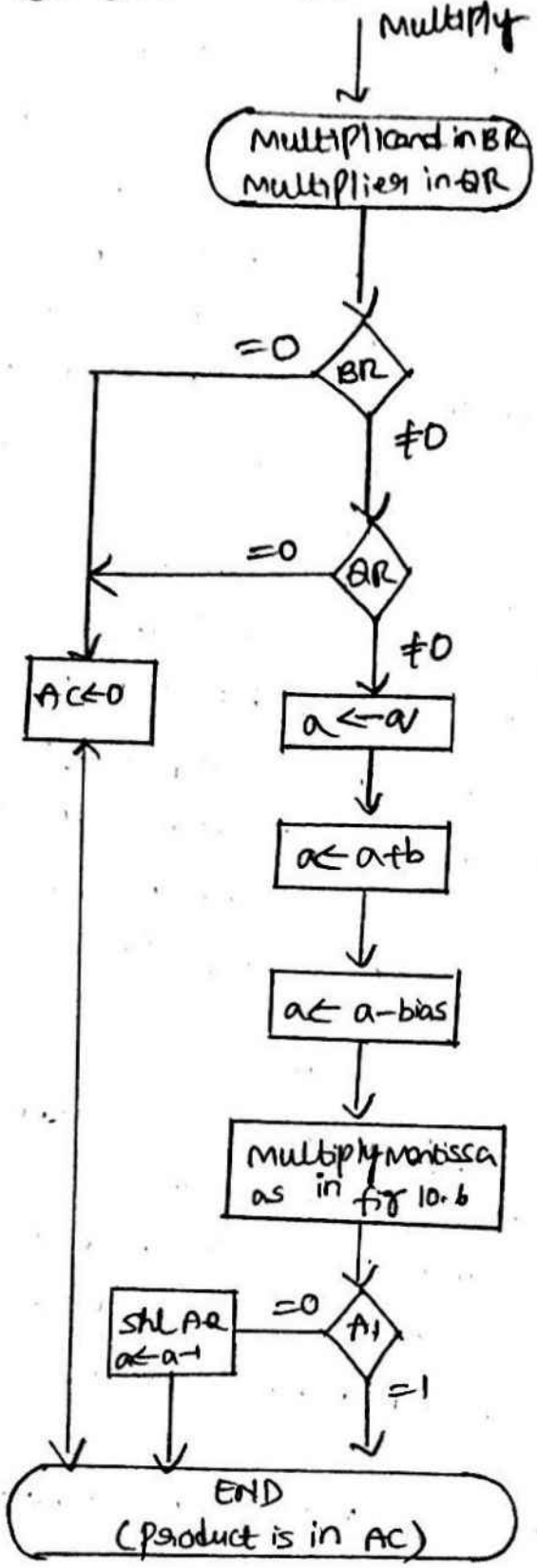



Fig 10.16 Multiplication of floating-point numbers.

→ 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 a & the adder is between exponents a & b. It is necessary to transfer the exponents from a 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 Q & 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.

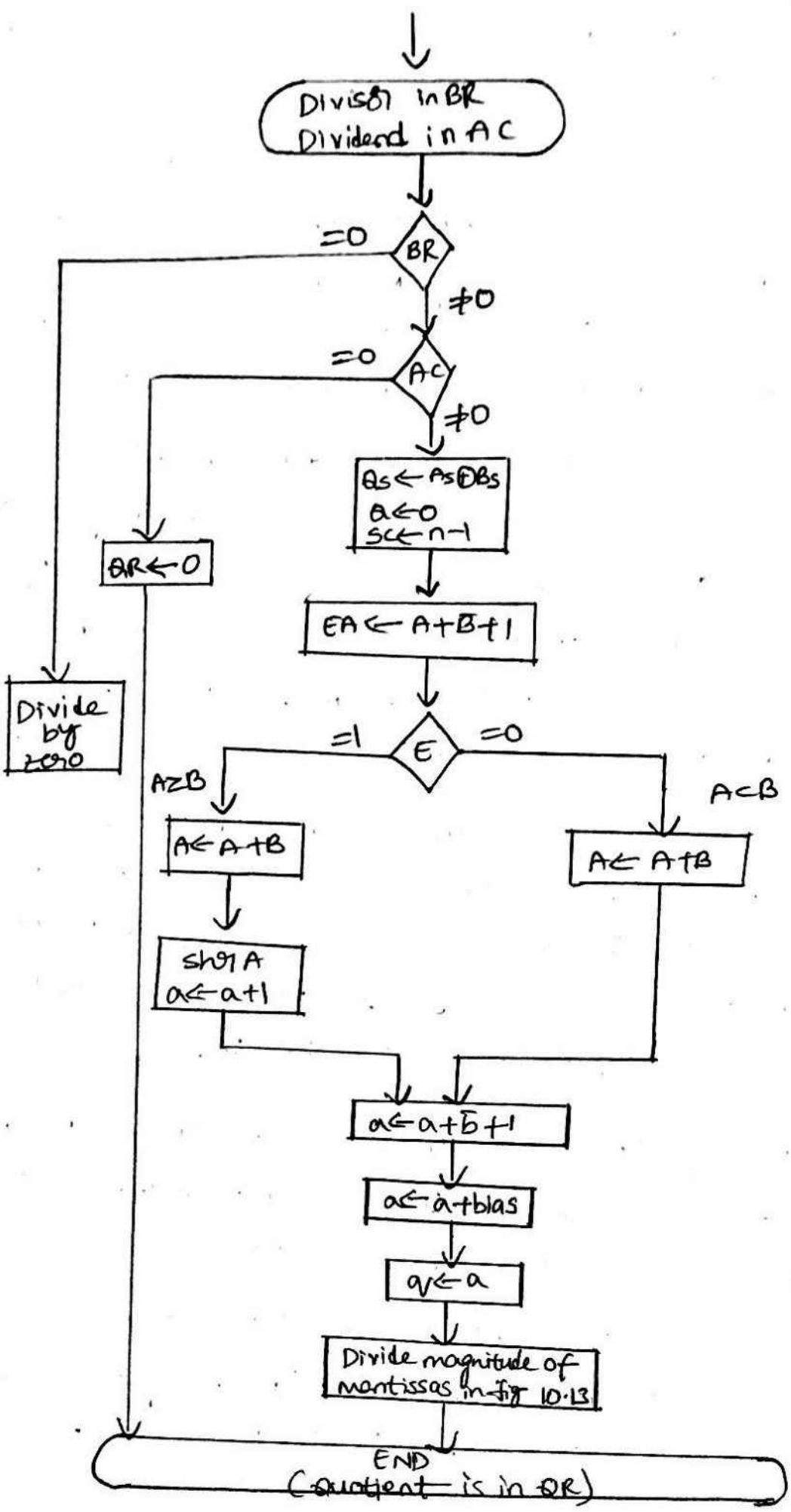



Fig 10.17 Division of floating point numbers.

→ 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 dividend 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 micooperations 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 micooperations. It can add or subtract decimal numbers, usually by formatting 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.

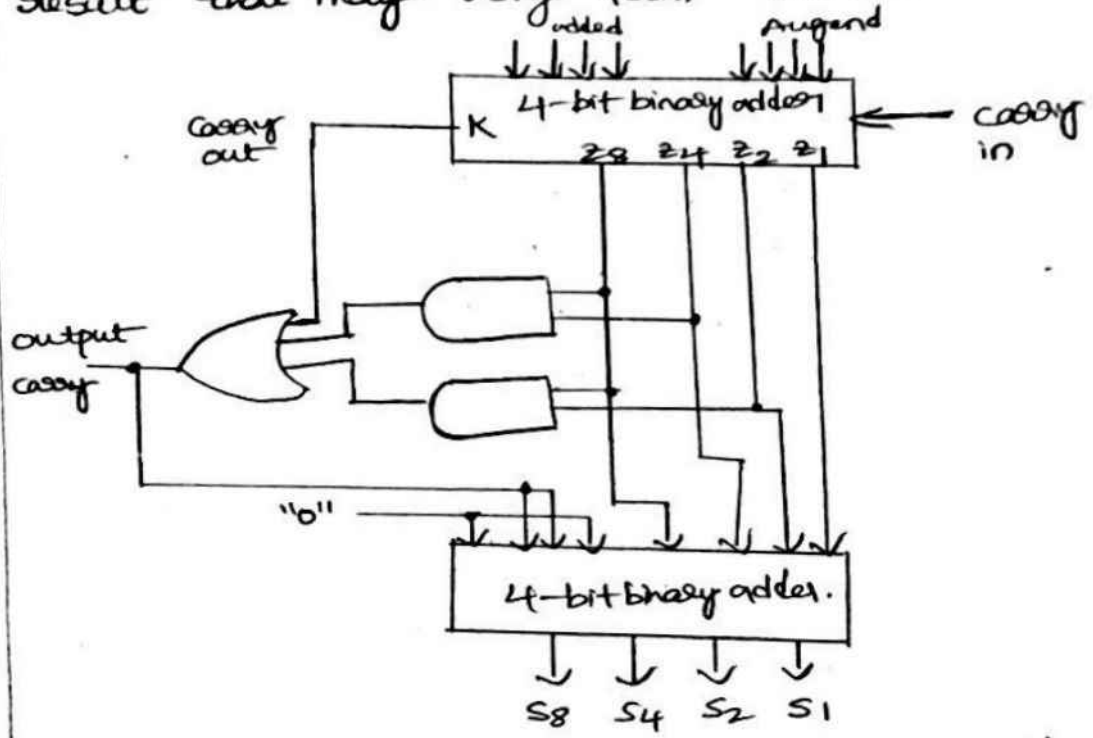



Fig 10.8 Block diagram of BCD adder.

→ 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 conversion is needed.

→ when the binary number 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.

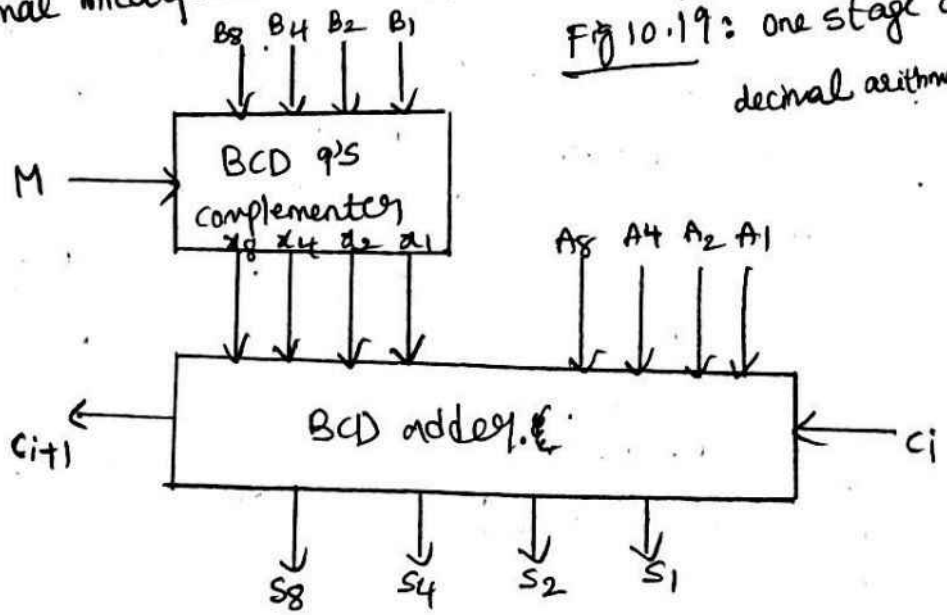



Fig 10.19: One stage of a decimal arithmetic unit

→ 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 $A \leftarrow A+B+1$ 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.

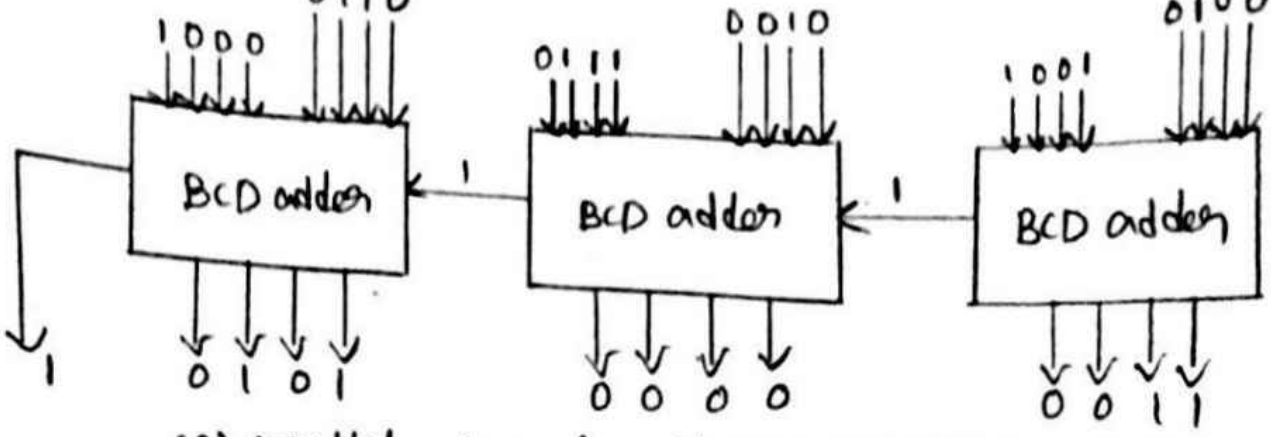
→ The binary data must employ a binary adder and a complementer. The decimal data must employ a decimal arithmetic unit capable of adding two BCD numbers & forming the 9's complement of the subtrahend, as shown in figure one stage of a decimal arithmetic unit.

→ 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.

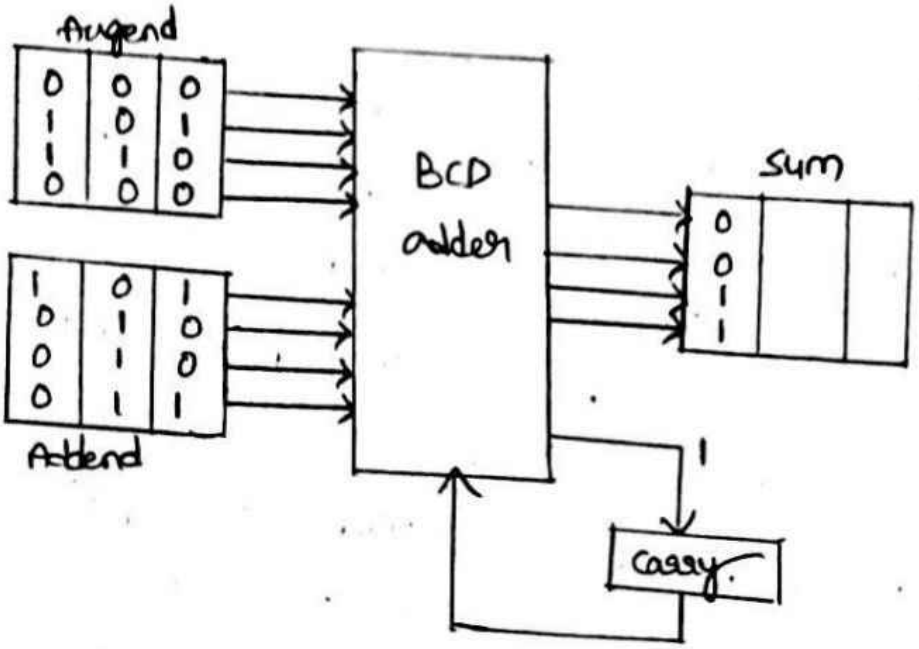
→ In digit-serial bit-parallel method, the digits are applied to a single BCD adder serially, while the bits of each coded digit are transferred in parallel. The sum is formed by shifting the decimal numbers through the BCD adder one at a time.

→ For K decimal digits, this configuration requires K microoperations, one for each decimal shift. In all the serial adder, the bits are shifted one at a time through a full-adder.

→ The binary sum formed after four shifts must be corrected into a valid BCD digit.




(a) parallel decimal addition: 624+879 = 1503.




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

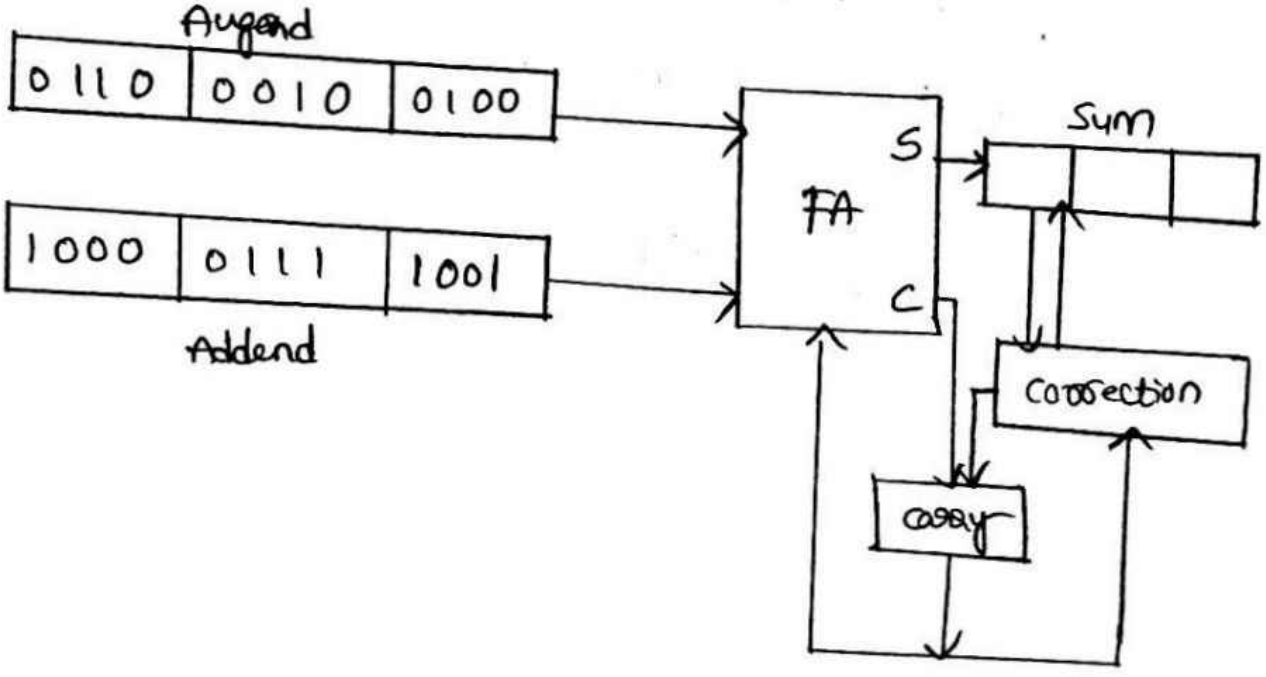



Figure 10.20 Three ways of adding decimal numbers.

→ 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.

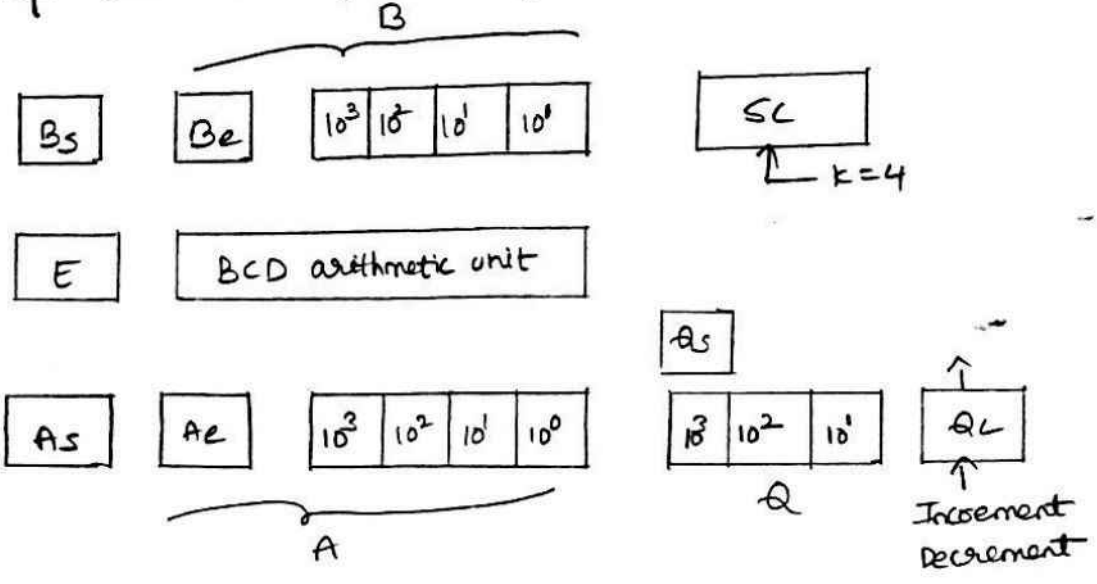



Fig 10.21 Registers for decimal arithmetic multiplication & division.

-) 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.

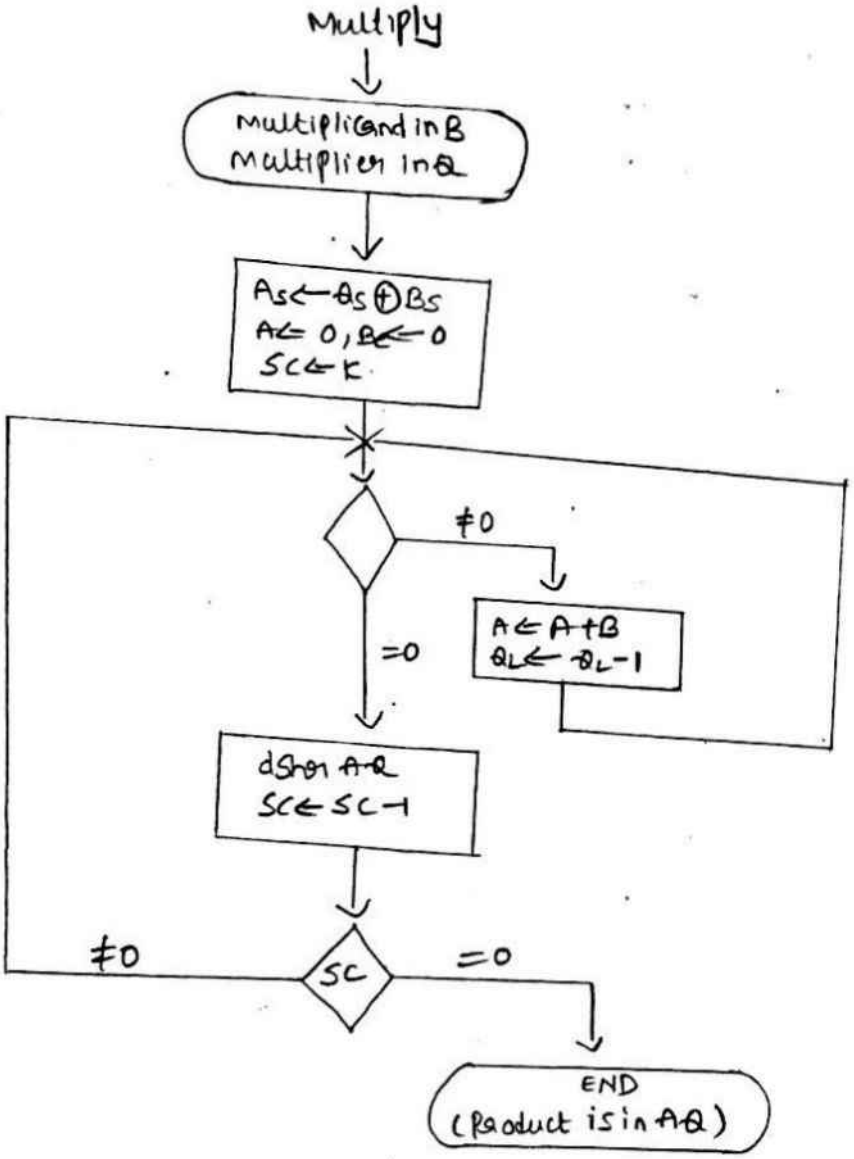



Fig 10.22 Flowchart for decimal multiplication.

→ 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 QL is checked. If it is not equal to 0, the multiplicand in B is added to the partial product in A once & QL is decremented.

→ QL 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 QL. 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.

→

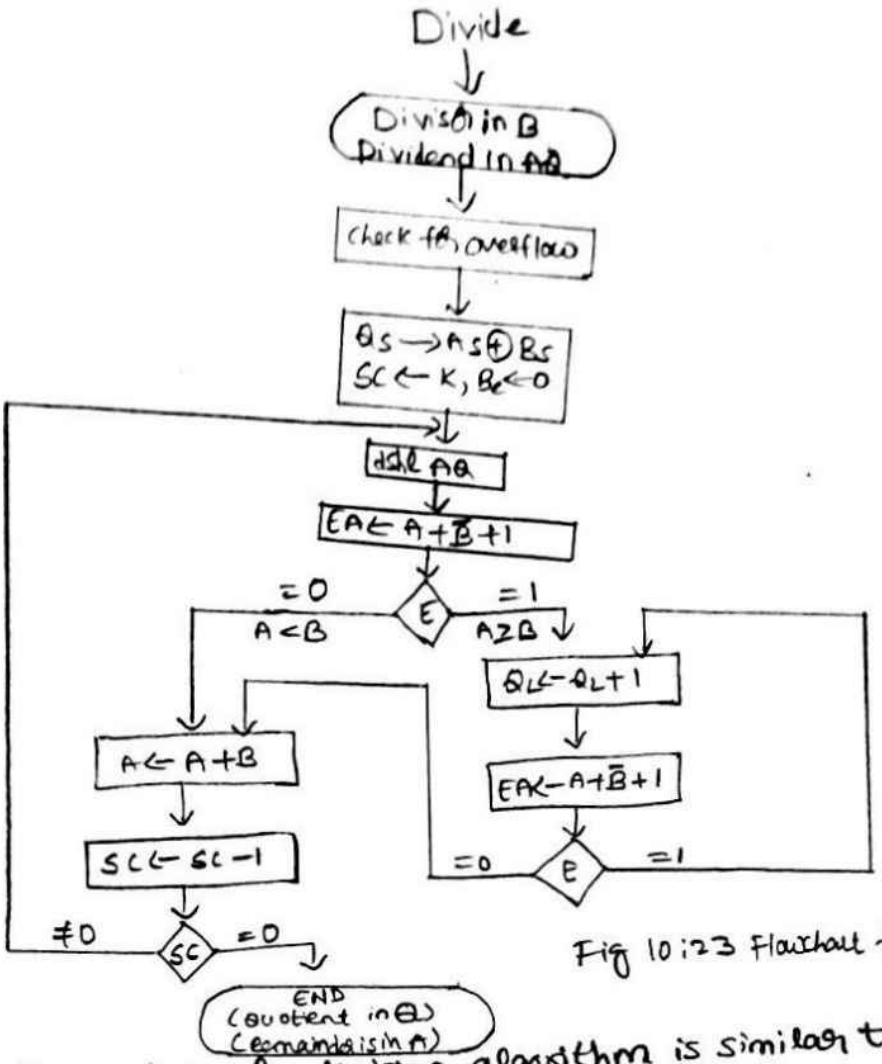



Fig 10.23 Flowchart for decimal division.

→ 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 recognized by E being 0.

→ when this occurs, the quotient digit is not incremented but the divisor is added to ~~this~~ 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.

# UNIT - V

## Input-Output Organization

## Memory Organization

The I/0 bus consists of data lines, address lines, and control lines. ***Each peripheral device has associated with it an interface unit.***

Each ***interface decodes the address and control received from the I/O bus***, interprets them for the peripheral, and ***provides signals for the peripheral controller***. It also ***synchronizes the data flow and supervises the transfer*** between peripheral and processor.

Each ***peripheral has its own controller that*** operates the particular electromechanical device. For example, the ***printer controller controls the paper motion, the print timing, and the selection of printing characters***. A controller may be housed separately or may be physically integrated with the peripheral.

The I/O bus from the processor is attached to all peripheral interfaces. ***To communicate with a particular device***, the processor places a device address on the address lines. ***Each interface attached to the I/0 bus contains an address decoder that monitors the address lines***. ***When the interface detects its own address, it activates the path between the bus lines and the device that it controls***. All peripherals whose address does not correspond to the address in the bus are disabled by their interface.

At the same time that the address is made available in the address lines, the ***processor provides a function code in the control lines. The interface selected responds to the function code and proceeds to execute*** it. The function code is referred to as an I/O command and is in essence an instruction that is executed in the interface and its attached peripheral unit. There are **four types of commands that an interface may receive**. They are classified as control, status, data output, and data input.

- A ***control command*** is issued to ***activate the peripheral*** and to inform it what to do.
- A ***status command*** is used to test various status conditions in the interface and the peripheral. For Example, the computer may wish to check the status of the peripheral before a transfer is initiated.
- A ***data output command*** causes the interface to respond by transferring ***data from the bus into one of its registers.***
- The ***data input command*** is the opposite of the data output. In this case the ***interface receives an item of data from the peripheral and places it in its buffer register.*** The processor checks if data are available by means of a status command and then issues a data input command. ***The interface places the data on the data lines, where they are accepted by the processor.***

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

1. Use two separate buses, one for memory and the other for I/O. (This method uses a separate I/O Processor alongside CPU to provide an independent pathway for the transfer of information between external devices and internal memory.)

2. Use one common bus for both memory and I/O but have separate control lines for each.

3. Use one common bus for memory and I/O with common control lines.

**Isolated I/O:** Many computers use one common bus to transfer information between memory or I/0 and the CPU. The distinction between a memory transfer and I/0 transfer is made through ***separate read and write lines***.

The ***CPU specifies whether the address on the address lines*** is for a memory word or for an interface register by ***enabling one of two possible read or write lines***. ***The I/0 read and I/0 write control lines are enabled during an I/0 transfer. The memory read and memory write control lines are enabled during a memory transfer***. This configuration isolates all I/0 interface addresses from the addresses assigned to memory and is referred to as the isolated I/0 method for assigning addresses in a common bus. In the isolated I/0 configuration, the CPU has distinct ***input and output instructions***, and each of these instructions is associated with the address of an interface register. When the CPU fetches and decodes the operation code of an input or output instruction, it places the address associated with the instruction into the common address lines. At the same time, it enables the I/0 read (for input) or I/0 write (for output) control line. This informs the external components that are attached to the common bus that the address in the address lines is for an interface register and not for a memory word.

**Memory-mapped I/O:** **Memory mapped I/O** uses the same address space for both memory and I/O. This is the case in computers that ***employ only one set of read and write signals and do not distinguish between memory and I/O addresses***. This configuration is referred to as memory-mapped I/O. In a memory-mapped l/O organization ***there are no specific input or output instructions.*** The ***CPU can manipulate l/O data residing in interface registers with the same instructions that are used to manipulate memory words***. Each interface is organized as a set of registers that respond to read and write requests in the normal address space.

Computers with memory-mapped l/O can ***use memory-type instructions to access l/0 data***. It allows the computer to use the same instructions for either input-output transfers or for memory transfers. The ***advantage is that the load and store instructions used for reading and writing from memory can be used to input and output data from l/O registers.*** In a typical computer, there are more memory-reference instructions than l/O instructions. With memory-mapped l/O all instructions that refer to memory are also available for l/O.

## DATA TRANSFER MODES:

***The transfer of data between two units may be done in parallel or serial***. In ***parallel data transmission, each bit of the message has its own path*** and the total message is transmitted at the same time. This means that an n-bit message must be transmitted through n separate conductor paths. In serial data transmission, each bit in the message is sent in sequence one at a time. ***This method requires the use of one pair of conductors or one conductor and a common ground.*** Parallel transmission is faster but requires many wires. It is **used for short distances and where speed is important**. Serial transmission is slower but is less expensive since it requires only one pair of conductors.

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

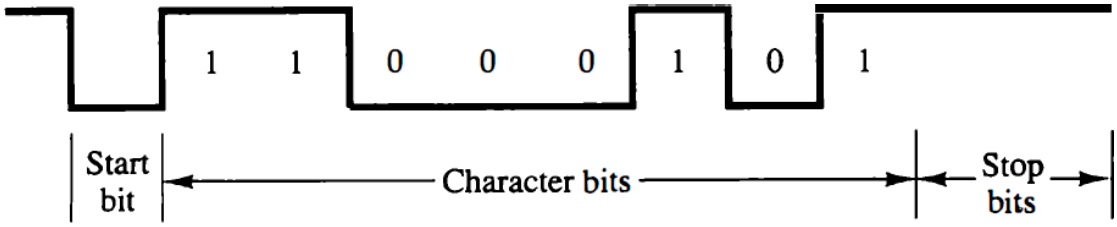
**Synchronous Data Transfer:** In synchronous transmission, the two units (***Sending Unit and Receiving unit) share a common clock frequency*** and bits are transmitted continuously at the rate dictated by the clock pulses. In long distant serial transmission, each unit is driven by a separate clock of the same frequency. *Synchronization signals are transmitted periodically between the two units to keep their clocks in step with each other*.

In asynchronous transmission, binary information is sent only when it is available and the line remains idle when there is no information to be transmitted. This is in contrast to synchronous transmission, ***where bits must be transmitted continuously to keep the clock frequency in both units synchronized with each other.***

***Eg:Any two units of a digital system are designed independently, such as CPU and I/O interface. If the registers in the I/O interface share a common clock with CPU registers, then transfer between the two units is said to be synchronous.***

**Asynchronous Serial Transfer:** A serial asynchronous data transmission technique used in many interactive terminals ***employs special bits that are inserted at both ends of the character code***. With this technique, each character consists of three parts: a start bit, the character bits, and stop bits. The convention is that the transmitter rests at the 1-state when no characters are transmitted. The first bit, called the ***start bit, is always a 0 and is used to indicate the beginning of a character.*** The last bit called the stop bit is always a 1.

An example of this format is shown below:




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

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

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

3. The character bits always follow the start bit.

4. After the last bit of the character is transmitted, a stop bit is detected when the line returns to the 1-state for at least one bit time.

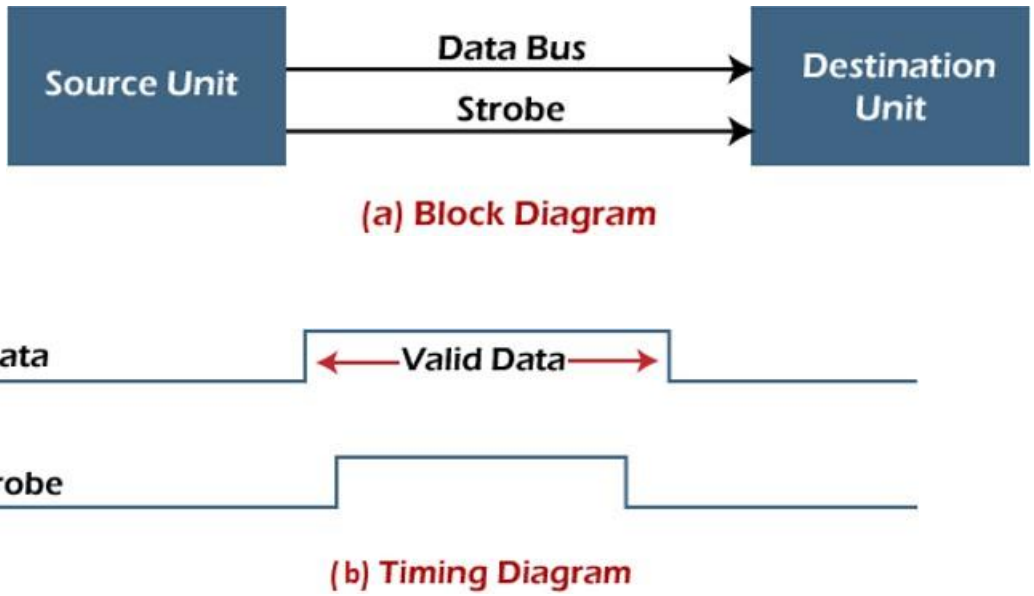
Asynchronous way of data transfer can be achieved using two methods:

1) Strobe control

2) Handshaking

**Strobe Control Method**: The Strobe Control method of asynchronous data transfer employs a single control line to time each transfer. This control line is also known as a strobe, and it may be achieved either by source or destination, depending on which initiate the transfer.

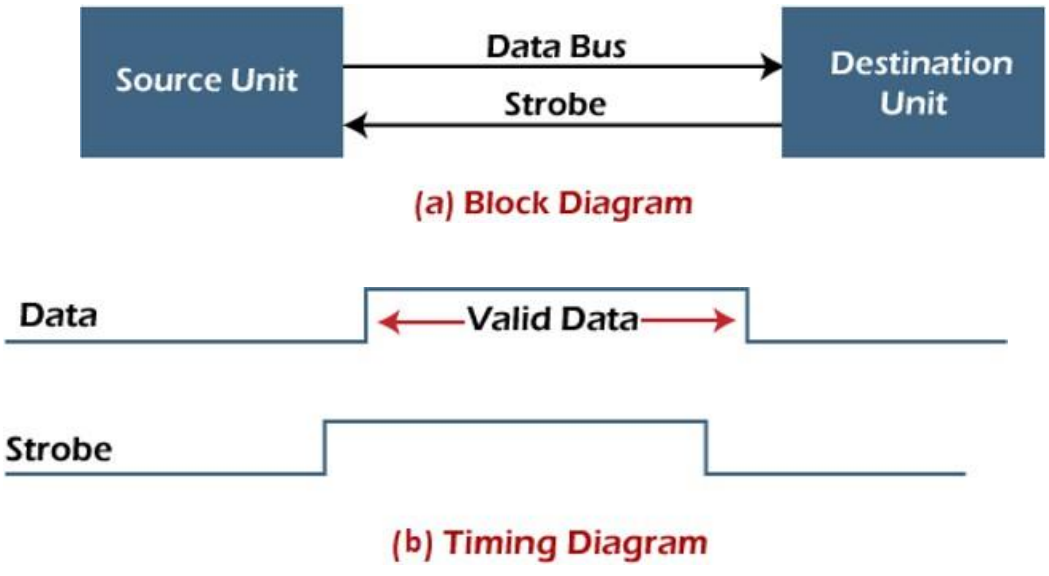
**Source initiated strobe:** In the below block diagram, strobe is initiated by source, and as shown in the timing diagram, the source unit first places the data on the data bus.




After a brief delay, the source activates a strobe pulse. The information on the data bus and strobe control signal remains in the active state for a sufficient time to allow the destination unit to receive the data. **The destination unit uses a falling edge of strobe control to transfer the contents of a data bus to one of its internal registers.** The source removes the data from the data bus after it disables its strobe pulse. Thus, new valid data will be available only after the strobe is enabled again.

Example: The strobe may be a memory-write control signal from the CPU to a memory unit.

**Destination initiated strobe:** In the below block diagram, the strobe initiated by destination, and in the timing diagram, the destination unit first activates the strobe pulse, informing the source to provide the data.



The falling edge of the strobe pulse can use again to trigger a destination register. The destination unit then disables the strobe. Finally, and source removes the data from the data bus after a determined time interval.

Example: the strobe may be a memory read control from the memory unit to CPU.

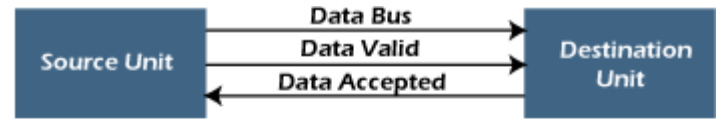
**Handshaking Method:** The strobe method has the disadvantage that the source unit that initiates the transfer has no way of knowing whether the destination has received the data that was placed in the bus. Similarly, a destination unit that initiates the transfer has no way of knowing whether the source unit has placed data on the bus.

So this problem is solved by the handshaking method. The handshaking method introduces a second control signal line.

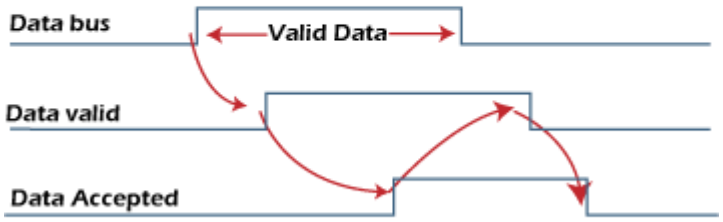
In this method, one control line is in the same direction as the data flow in the bus from the source to the destination. The source unit uses it to inform the destination unit whether there are valid data in the bus.

The other control line is in the other direction from the destination to the source. This is because the destination unit uses it to inform the source whether it can accept data. And in it also, the sequence of control depends on the unit that initiates the transfer. So it means the sequence of control depends on whether the transfer is initiated by source and destination.

**Source initiated handshaking:** In the below block diagram, two handshaking lines are "data valid", which is generated by the source unit, and "data accepted", generated by the destination unit.




(a) Block Diagram



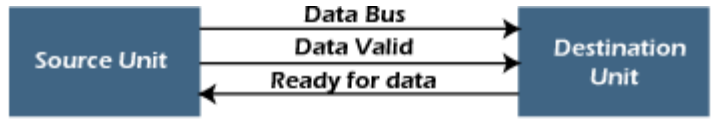

(b) Timing Diagram

The timing diagram shows the timing relationship of the exchange of signals between the two units. The source initiates a transfer by placing data on the bus and enabling its data valid signal. The destination unit then activates the data accepted signal after it accepts the data from the bus.

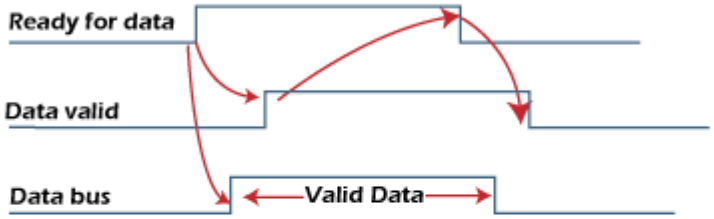
The source unit then disables its valid data signal, which invalidates the data on the bus. After this, the destination unit disables its data accepted signal, and the system goes into its initial state. The source unit does not send the next data item until after the destination unit shows

readiness to accept new data by disabling the data accepted signal. This sequence of events described in its sequence diagram, which shows the above sequence in which the system is present at any given time.

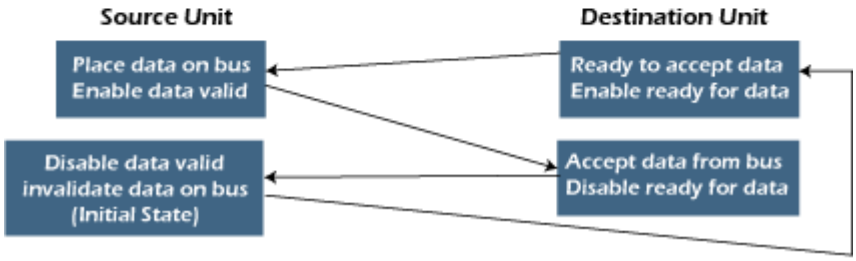
**Destination initiated handshaking:** In the below block diagram, the two handshaking lines are "**data valid**", generated by the source unit, and "**ready for data**" generated by the destination unit.




(a) Block Diagram




(b) Timing Diagram




(c) Sequence Diagram (Sequence of events)

## I/O Transfers:

Binary information received from an external device is usually stored in memory for later processing. Information transferred from the central computer into an external device originates in the memory unit. The CPU merely executes the I/0 instructions and may accept the data temporarily, but the ultimate source or destination is the memory unit. Data transfer between the central computer and I/0 devices may be handled in a variety of modes. Some modes use the 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:

1) **Program Controlled I/O**
2) **Interrupt-initiated I/0**
3) **Direct memory access (DMA)**

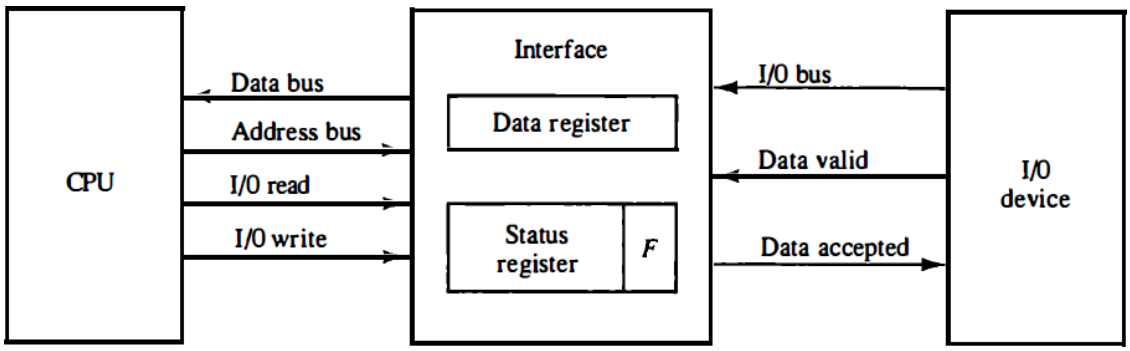
1. ***Program Controlled I/O:*** Programmed I/0 operations are the result of I/0 instructions written in the computer program. ***CPU executes a program that transfers data between I/O device and memory.*** Each data item transfer is initiated by an instruction in the program. **Usually, the transfer is to and from a CPU register and peripheral.** ***Other***

***instructions are needed to transfer the data to and from CPU and memory***. Transferring data under program control requires constant monitoring of the peripheral by the CPU. Once a data transfer is initiated, the CPU is required to monitor the interface to see when a transfer can again be made. In the programmed I/0 method, the CPU stays in a program loop until the I/0 unit indicates that it is ready for data transfer. This is a time-consuming process since it keeps the processor busy needlessly.

**Example of Programmed I/0:**

In the programmed I/0 method, the I/0 device does not have direct access to memory. A transfer from an I/0 device to memory requires the execution of several instructions by the CPU, including an input instruction to transfer the data from the device to the CPU and a store instruction to transfer the data from the CPU to memory. Other instructions may be needed to verify that the data are available from the device and to count the numbers of words transferred.

An example of data transfer from an I/O device through an interface into the CPU is shown in Fig below:




*F* = Flag bit

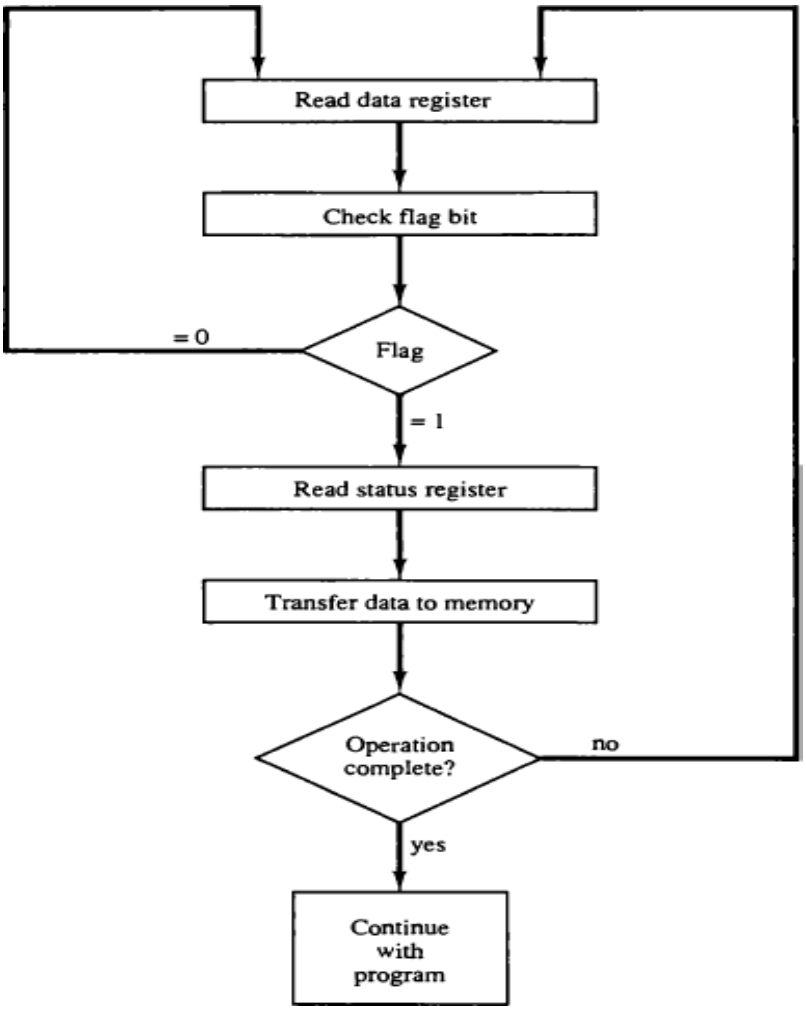
The device transfers bytes of data one at a time as they are available. When a byte of data is available, the device places it in the I/O bus and enables its data valid line. The interface accepts the byte into its data register and enables the data accepted line. The interface sets a bit in the status register that we will refer to as an F or "flag" bit. The device can now disable the data valid line, but it will not transfer another byte until the data accepted line is disabled by the interface.

A program is written for the computer to check the flag in the status register to determine if a byte has been placed in the data register by the VO device. This is done by reading the status register into a CPU register and checking the value of the flag bit. If the flag is equal to 1, the CPU reads the data from the data register. The flag bit is then cleared to 0 by either the CPU or the interface, depending on how the interface circuits are designed. Once the flag is cleared, the interface disables the data accepted line and the device can then transfer the next data byte. A flowchart of the program that must be written for the CPU is shown in Fig below. It is assumed that the device is sending a sequence of bytes that must be stored in memory. The transfer of each byte requires three instructions:

1 Read the status register.

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

3. Read the data register.




Each byte is read into a CPU register and then transferred to memory with a store instruction. A common I/O programming task is to transfer a block of words from an I/O device and store them in a memory buffer. The programmed VO method is particularly useful in small low-speed computers or in systems that are dedicated to monitor a device continuously.

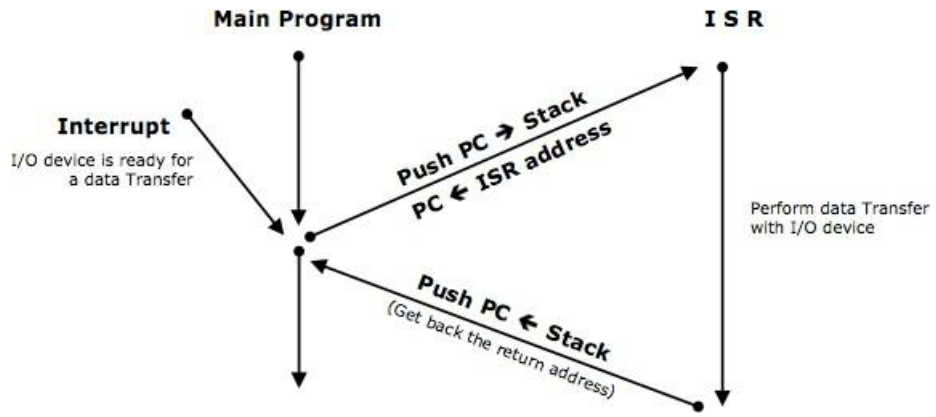
***2) INTERRUPT DRIVEN I/O:***

1) In interrupt driven I/O, the transfer is not initiated by the processor.
2) Instead, **an I/O device which wants to perform a data transfer with the processor, must give an interrupt to the processor.**
3) An interrupt is a condition that makes the processor execute an ISR (Interrupt Service Routine).
4) In the ISR, processor will perform data Transfer with the I/O device.
5) This relieves the processor from periodically checking the status of every I/O device thereby saves as
lot of time of the processor.
6) **The processor is free to carry on its own operations.**
7) Whenever a device wants to transfer data, it will interrupt the processor.
8) This is how many I/O devices Transfer data with the processor.

9) E.g.:: **Keyboard.** Instead of the processor checking all the time, whether a key is pressed, the
keyboard interrupts the processor as an when we press a key. In the ISR of the keyboard, which is a
part of the keyboard driver software, the processor will read the data from the keyboard.
10) **Hence interrupt driven I/O is much better than Polled I/O (Programmed I/O).**

**INTERRUPT HANDLING MECHANISM**

1) When an interrupt occurs, processor, firstly, finishes the current instruction.
2) It then suspends the current program and executes an ISR.
3) To do so, it Pushes the value of PC (address of next instruction), into the stack.
4) Now it loads the ISR address into PC and proceeds to execute the ISR.
5) At the end of the ISR, it POPs the return address from the stack and loads it back into PC.
6) This is how the processor return to the very next instruction in the program.




| | INTERRUPT DRIVEN I/O | POLLING (PROGRAMMED I/O) |
|---|---|---|
| 1 | **I/O device interrupts the processor whenever it wants to perform a data Transfer.** | **Processor periodically checks (polls) the status of every I/O device to know if it wants to perform a data Transfer.** |
| 2 | **Processor is free** to carry on its own operations, hence **saves system time**. | **Processor is busy** in constantly checking all I/O devices, hence **system time wasted**. |
| 3 | **Additional hardware required** to handle interrupts. E.g.:: 8259 Programmable interrupt controller. | Additional hardware **not required**. |
| 4 | Increases **cost and complexity** of the system. | System is **cheaper and less complex**. |
| 5 | Interrupt **priority** has to be managed through software or through hardware. | No such issue. |
| 6 | Interrupt **vector addresses** (ISR Addresses) need to be stored in an Interrupt Vector table - **IVT**. | No such issue. |

### *3) DMA BASED I/O*

DMA means **transferring data directly between memory and I/O.**
DMA transfers are **very fast** as compared to Processor based transfers due to two reasons.
1. They are **hardware based** so no time is wasted in fetching and decoding instructions.
2. Transfers are **directly between memory and I/O** without data going via the Processor.
To Perform a DMA transfer we need a **DMA Controller like 8237/ 8257.**
It is capable of taking control of the buses from the Processor.
The process is performed as follows.
1) By Default **Processor is the bus master**.
2) The DMA transfer parameters first initialized by the processor.
3) **Processor programs two registers inside the DMAC called CAR and CWCR** giving the starting
address and the number of bytes to be transferred.
4) DMAC now ensures that the I/O device is ready for the transfer by checking the DREQ signal.
5) **If DREQ=1, then DMAC gives HOLD signal** to the Processor requesting control of the system bus.
6) **Processor releases control of the bus** after finishing the current machine (bus) cycle.
7) Processor **gives HLDA** informing DMAC that it is now the bus master.
8) **DMAC issues DACK#** (by default active low, but can be changed) to I/O device indicating that the
transfer is about to begin.
9) Now DMAC **transfers one byte in one cycle**.
10) After every byte is transferred the **Address register and Count register are decremented by 1**.
11) This repeats till Count reaches **"0"** also called **Terminal Count**.
12) Now the **transfer is complete**.
13) DMAC **returns the system bus to Processor by making HOLD = 0**.
14) Processor once again **becomes bus master**.

**Advantage of DMA**

DMA transfers are very fast.

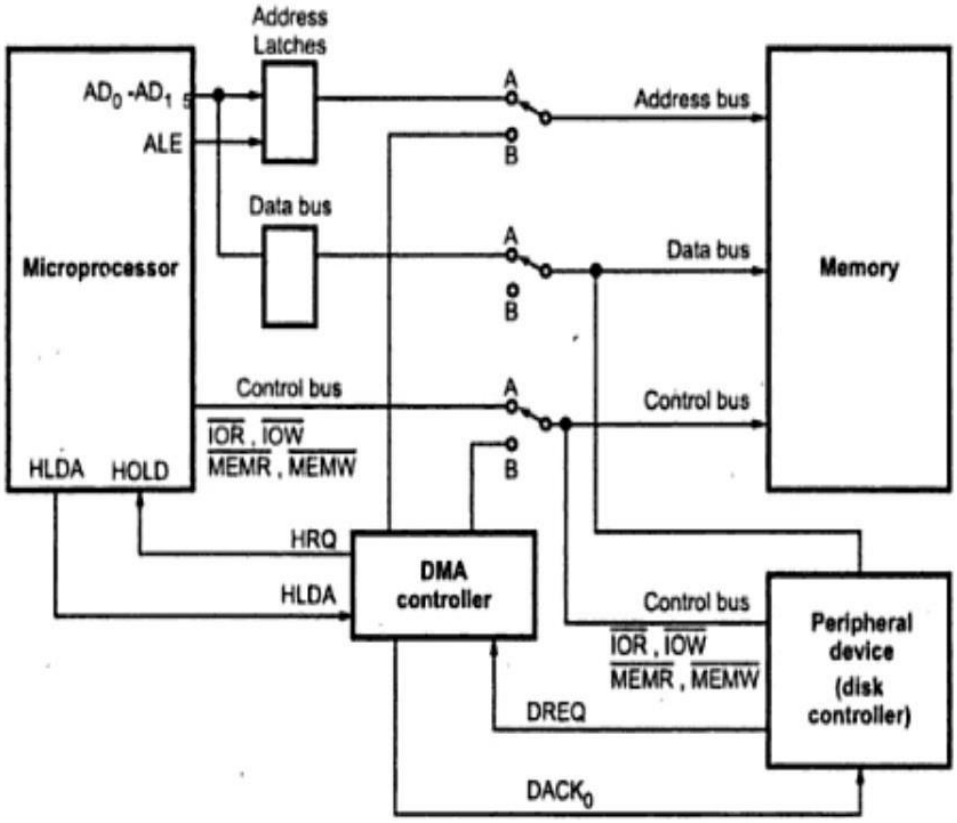
**Drawback of DMA**

DMAC becomes the bus master. Hence during DMA cycles, the processor cannot perform any operations
as the bus is already being used for DMA. The processor remains in HOLD state.

**Difference between Interrupt Request and DMA request**

When an interrupt occurs, the processor has to suspend the current program, execute the ISR and then
**return to the next instruction** of the main program. Hence it is necessary that the **processor completes the current instruction** before servicing an interrupt request.
When a DMA request occurs, the processor has to simply relinquish (give away) control of the system bus
and enter hold state. When ever it gets back the bus it can **resume from where it had left**. Hence the processor **need not finish the current instruction** before servicing a DMA request. It simply has to **finish the current machine cycle**. **Hence Instruction cycles are Interrupt Breakpoints and Machine cycles are DMA breakpoints.**

## DMA Operation




**TYPES / METHODS / TECHNIQUES OF DMA TRANSFERS**

There are **four modes of data transfer**:

**1) BLOCK TRANSFER MODE / BURST MODE.**

In this mode, the DMAC is programmed to **transfer ALL THE BYTES** in one complete DMA operation. After a byte is transferred, the **CAR and CWCR are adjusted** accordingly. The **system bus** is **returned** to the **processor, ONLY after all the bytes are transferred.** It is the **fastest** form of DMA but keeps the **processor inactive** for a long time.

**2) SINGLE BYTE TRANSFER MODE/ CYCLE STEALING.**

Once the DMAC becomes the bus master, it will transfer only **ONE BYTE** and return the bus to the processor. As soon as the processor performs one bus cycle, DMAC will once again take the bus back from the processor. Hence both **DMAC and processor** are **constantly stealing bus cycles** from each other. It is the **most popular** method of DMA, because it keeps the **processor active in the background**. After a byte is transferred, the **CAR** and **CWCR** are **adjusted** accordingly.

**3) DEMAND TRANSFER MODE.**

It is very **similar to Block Transfer**, **except** that the **DREQ must remain active throughout the DMA operation**. **If** during the operation **DREQ goes low**, the **DMA operation is stopped** and the **busses** are **returned** to the **processor.**

In the meantime, the **processor** can **continue** with its own operations. **Once DREQ goes high again**, the **DMA operation continues** from where it had stopped. This means, the transfer happens on demand from the I/O device, hence the name.

**4) HIDDEN MODE / TRANSPARENT MODE.**

In this mode, **once the processor programs all parameters inside the DMAC, the DMAC** does not request the processor for the control of the bus. Instead, it observes the processor. **It waits for the processor to enter idle state**. Once the processor is idle, the DMAC will take control of the bus and perform the Transfer. So, the Transfer is **totally transparent to the processor** or hidden from the processor. Hence the name.

**Interrupts and Exceptions:**

**Interrupt**

The term Interrupt is usually reserved for hardware interrupts. They are program control interruptions caused by external hardware events. Here, external means external to the CPU. Hardware interrupts usually come from many different sources such as timer chip, peripheral devices (keyboards, mouse, etc.), I/O ports (serial, parallel, etc.), disk drives, CMOS clock, expansion cards (sound card, video card, etc). That means hardware interrupts almost never occur due to some event related to the executing program.

**Example –**

An event like a key press on the keyboard by the user, or an internal hardware timer timing out can raise this kind of interrupt and can inform the CPU that a certain device needs some attention. In a situation like that the CPU will stop whatever it was doing (i.e. pauses the current program), provides the service required by the device and will get back to the normal program. When hardware interrupts occur and the CPU starts the ISR, other hardware interrupts are disabled (e.g. in 80×86 machines). If you need other hardware interrupts to occur while the ISR is running, you need to do that explicitly by clearing the interrupt flag (with sti instruction). In 80×86 machines, clearing the interrupt flag will only affect hardware interrupts.

**Exception**

Exception is a software interrupt, which can be identified as a special handler routine. An exception occurs due to an "exceptional" condition that occurs during program execution.

**Example –**

Division by zero, execution of an illegal opcode or memory related fault could cause exceptions. Whenever an exception is raised, the CPU temporarily suspends the program it was executing and starts the ISR. ISR will contain what to do with the exception. It may correct the problem or if it is not possible it may abort the program gracefully by printing a suitable error message. Although a specific instruction does not cause an exception, an exception will always be caused by an instruction. For example, the division by zero error can only occur during the execution of the division instruction.

Exceptions and interrupts are unexpected events which will disrupt the normal flow of execution of instruction (that is currently executing by processor). An exception is an unexpected event from within the processor. Interrupt is an unexpected event from outside the processor. Whenever an exception or interrupt occurs, the hardware starts executing the code that performs an action in response to the exception. This action may involve killing a process, outputting an error message, communicating with an external device, or horribly crashing the entire computer system by initiating a "Blue Screen of Death" and halting the CPU. The instructions responsible for this action reside in the operating system kernel, and the code that performs this action is called the interrupt handler code. handler code is an operating system subroutine. Then, After the handler code is executed, it may be possible to continue execution after the instruction where the execution or interrupt occurred.