COMPUTER ORGANIZATION AND ARCHITECTURE

Introduction

The course is about the structure and functions of the computer. It presents the nature and characteristics of modern day computers. This is a challenging task because:

1. There are so many products from single chip microcomputers to supercomputers.
2. The rapid pace of change that has always characterized computer technology continues.

However certain fundamental concepts apply consistently.

Although computers have varying sizes, capabilities, and cost, they share many characteristics and operating principles. The course describes general principles of computer architecture that apply to computers of any category.

ORGANISATION AND ARCHITECURE

Computer architecture is concerned with the structure and behaviour of the various functional modules of the computer and how they interact to provide the processing needs of the user.

Architectural attributes include the instruction set, the number of bits used, I/O mechanisms, techniques for addressing etc.

Computer organisation is concerned with the way the hardware components are connected together to form a computer system. Organisational attributes include hardware details visible to the user e.g. the interfaces, memory technology etc.

N.B. A number of manufacturers offer many different computer models (organizations) but all having the same architecture and thus differing in costs.

Computer design is concerned with the development of the hardware for the computer taking into consideration a given set of specifications.

WHY STUDY COMPUTER ORGANIZATION AND ARCHITECTURE

To be a professional in any field of computing one should not regard a computer as a black box that executes programs by magic. Students need to understand the computer's functional components, their characteristics, their performance and their interactions.

Students need to understand computer architecture in order to structure their programs so that they run more efficiently on a real machine e.g. in selecting a system to use they should be able to understand the tradeoffs among various components such as CPU clock speed vs memory size.

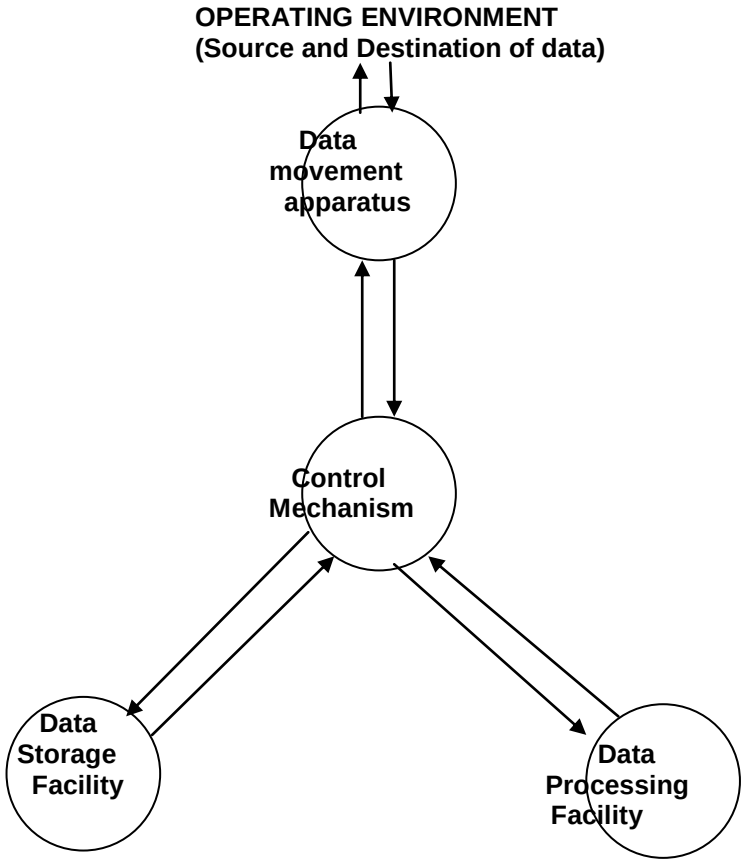
A graduate may be required to select the most cost effective computer for use in an organization.

Computer architecture concepts are needed in other courses e.g. how a computer provides architectural support for programming languages and how operating system facilities reinforce concepts in those areas.

Computer Functions

The basic requirements of the computer are to:

- Process data
- Store data



- Move the data between the different computer components and the external world; i.e, I/O functions (data received from or delivered to a device) and data communication functions (when data is moved longer distances)
- To control all the above operations

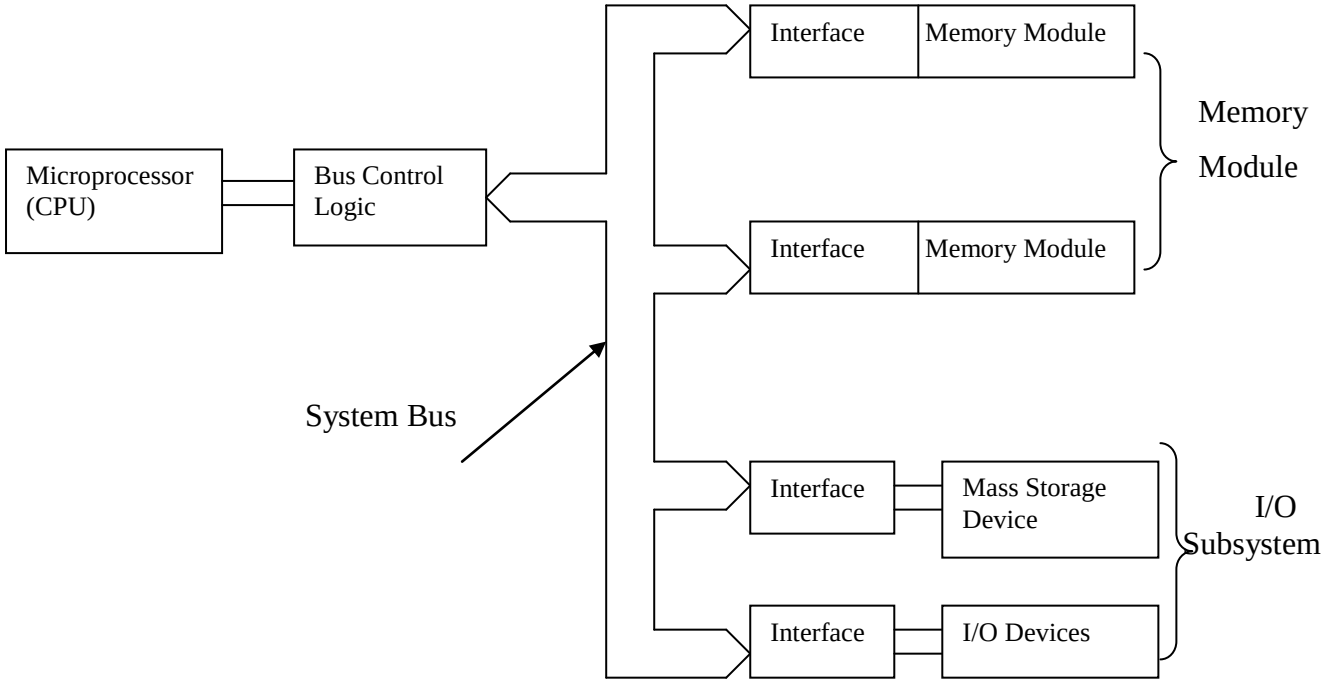
Computer structure

There are mainly four structural components

1. *Central Processing Unit*:
    - Decode the instructions and use them to control the activities within the system
    - It also performs the arithmetic(+ , -, /, *) and logical (>,>=,<,<=, =, =!) computations. ➔ (Data processing)

2. *Main Memory*:

Stores data and instructions that are currently being used.



3. I/O Subsystem:

Moves data between the computer and external environment. It consists of a variety of devices for communicating with the external world and for storing large quantities of information.

4. *System Interconnection:*

Mechanism to provide communication between the CPU, memory and the I/O sub system. It consists of the System Bus and the Interfaces

System Bus.

A set of conductors that connect the CPU to its memory and I/O devices. The bus conductors are normally separated into 3 groups:

- *The Data Lines:* for transmitting information

- *Address Lines:* Indicate where information is to come from or where it is to be placed.
- *Control Lines:* To regulate the activities on the bus.

Interfaces

Circuitry needed to connect the bus to a device. Memory interfaces consist of logic

- Needed to decode the address of the memory location being accessed.
- Buffer data onto/off the bus.

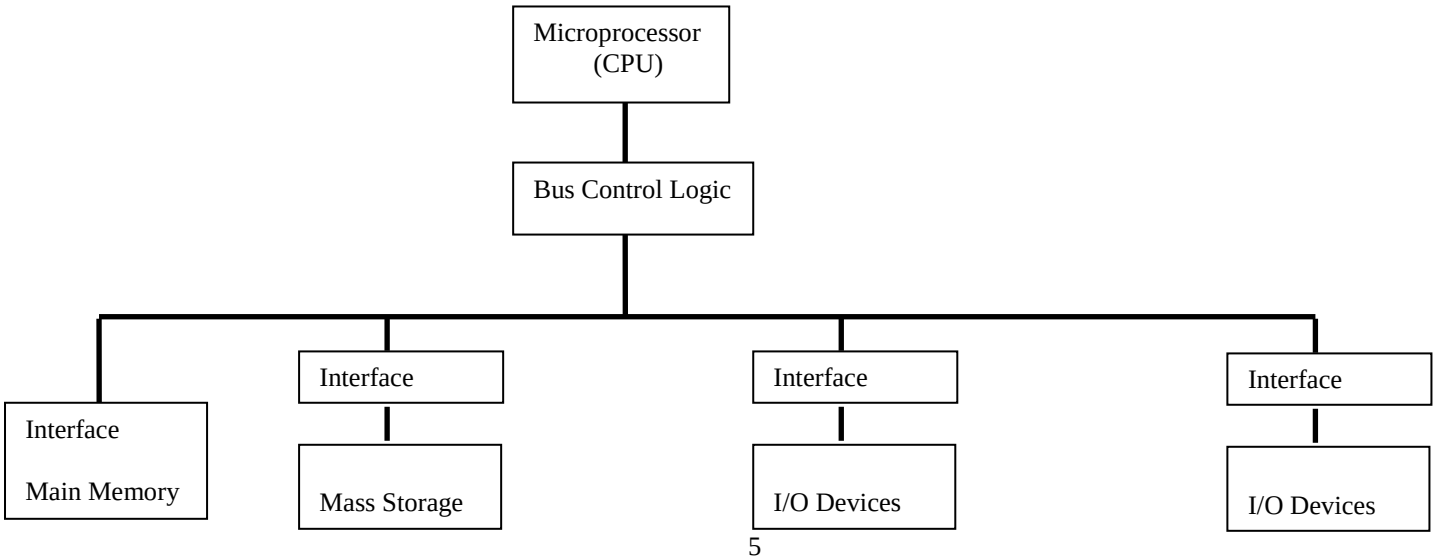
  Contain circuitry to perform memory reads or write.

I/O interfaces must

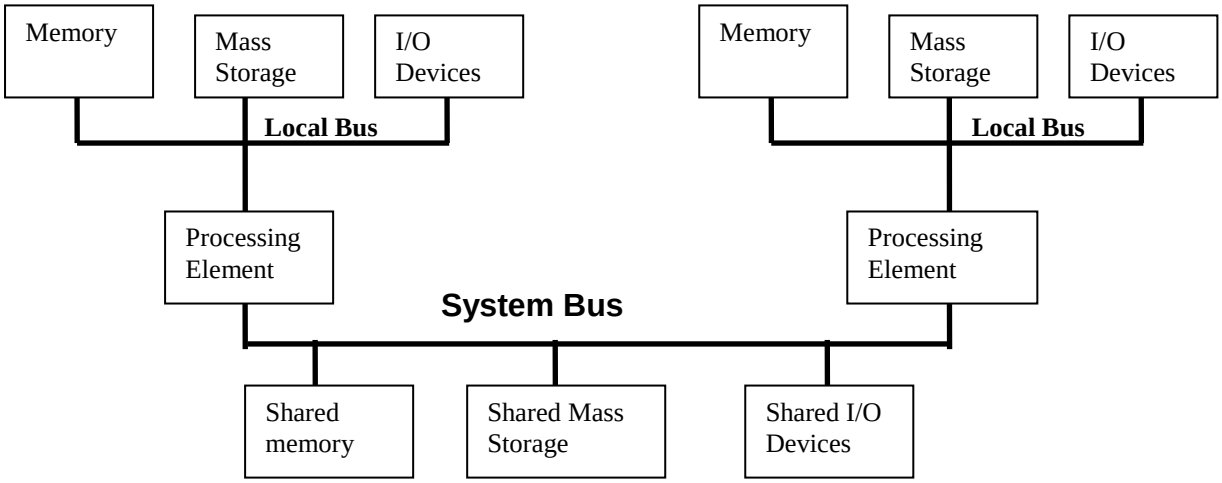
- Buffer data onto/off the system bus
- Receive commands from the CPU
- Transmit information from their devices to the CPU.

There may be one or more of each of the aforementioned components where we have the Single Bus Architecture or Multiprocessing.

(i) The **single bus / processor architecture** which has only one processing element and all the other components are connected to a single link (the **System Bus)**



(ii) The **Multiprocessing System** which has several processing elements surrounded by different subsystems and a central link (the **system bus**) connecting the different subsystems together.



The links in the subsystems are called **local buses.** Each subsystem can operate as an independent computer but can take advantage of the shared resources. The shared main memory can be used for passing information between subsystems and the shared mass storage can be used to store large programs and large quantities of data that are needed by more than one subsystem.

The competition for the shared resources by the different elements is called **contention**.

DATA REPRESENTATIONS

- Several formats are used to store data.
- Because a computer uses binary numbers all these formats are patterns of 1's and 0's.
- These binary digits are called BITS.
- In Computer circuits, 0's and 1's are voltage levels where a 0 is low voltage / OFF and a 1 is high Voltage /ON

These formats fall in two main categories

1. **Number Formats**: They store only numbers; There are three number formats:
   - Integer or Fixed point Formats.
   - Floating Point Formats
   - Binary Coded decimal (BCD)
2. **Alphanumeric Codes**: These store both numbers and characters including the alphabetic characters.

NUMBER FORMATS

Integer Formats

Human beings are trained to understand decimal system.

e.g. $5437 = 5000 + 400 + 30 + 7 = (5 * 10^3) + (4 * 10^2) + (3 * 10^1) + (7 * 10^0)$

In Binary

$11011 = (1 * 2^4) + (1 * 2^3) + (0 * 2^2) + (1 * 2^1) + (1 * 2^0) = 16 + 8 + 0 + 2 + 1 = 27_{10}$

Horner's Rule: Used to convert binary numbers to decimal.

e.g. $101101 = (1 * 2^5) + (0 * 2^4) + (1 * 2^3) + (1 * 2^2) + (0 * 2^1) + (1 * 2^0)$

$= 32 + 0 + 8 + 4 + 0 + 1 = 45_{10}$

Using Horner's Rule: $((((1 * 2 + 0)2 + 1)2 + 1)2 + 0)2 + 1 = 45_{10}$

- To convert from decimal to binary you perform successive divisions.
- e.g. $45_{10} = 101101_2$; $137_{10} = 1001001_2$

Hexadecimal

It has 15 digits 0 – 15

Decimal	Hexadecimal	Binary	Octal
0	0	0000	0
1	1	0001	1
2	2	0010	2
3	3	0011	3
4	4	0100	4
5	5	0101	5
6	6	0110	6
7	7	0111	7
8	8	1000	10
9	9	1001	11
10	A	1010	12
11	B	1011	13
12	C	1100	14
13	D	1101	15
14	E	1110	16
15	F	1111	17
16	10	10000	20

It's easier to represent numbers in hexadecimal than in binary.

e.g. To convert 110011011_2 to Hexadecimal

$(1 * 2^8)+ (1 * 2^7) + (1 * 2^4) + (1 * 2^3)+ (1 * 2^1)+ (1 * 2^0)$
$256 + 128 + 16 +8 + 2 + 1 = 411_{10}$

411 / 16 = 25 rem 11 B

25/16 = 1 rem 9 9 => $411_{10} = 19B_{16}$

- Each Hexadecimal digit can be represented by a unique combination of 4 binary bits
- => 1 1001 1011 = 0001 1001 1011
  1 9 B
- To convert $1C2E_{16}$ to decimal you expand.

- To convert 15797_{10} to hexadecimal you perform successive divisions.

Octal System

Each octal digit can be represented by a unique combination of three bits.

e.g. to convert 110011011_2 to base 8 first covert to decimal then perform successive divisions of 8 on the decimal number.

$110011011_2 = 411_{10}$
411/8 = 51 rem 3
51/8 = 6 rem 3 => $110011011_2 = 411_{10} = 633_8$

$101011000110_2 = 101\ 011\ 000\ 110 = 5306_8$
$= 1010\ 1100\ 0110 = AC6_{16}$
$1573_8 = 001\ 101\ 111\ 011$
$= 0011\ 0111\ 1011 = 37B_{16}$
$A748_{16} = 1010\ 011\ 0100\ 1000 =$
$= 001\ 010\ 011\ 101\ 001\ 000 = 123510_8$

In decimal you can expand numbers with fractions. E.g.

$$631.25 = (6 * 10^2) + (3 * 10^1) + (1 * 10^0) + (2 * 10^{-1}) + (5 * 10^{-2})$$

Similarly in binary;

$101.11 = (1 * 2^2) + (0 * 2^1) + (1 * 2^0) + (1 * 2^{-1}) + (1 * 2^{-2}) = 4 + 0 + 1 + ½ + ¼ = 5.75_{10}$

- If a decimal number is given the binary equivalent of the integer portion is obtained as usual but the fraction part is obtained by successively multiplying by 2.
- e.g. 13.6875;

13 = 1101 → 1 0 1 1
.6875 * 2 = 1.375
.375 * 2 = 0.75
.75 * 2 = 1.5
.5 * 2 = 1.0
=> 13.6875 = 1101.1011

Similarly $150.3125_{10} = 10010110.0101_2$

- If a binary number contains digits to the right of the decimal point we convert them by starting at the binary point and move to the right.

  e.g. $11.1010011011_2 = 011\ \ 101\ \ 001\ \ 101\ \ 100 = 3.5154_8$
  $= 0011\ \ 1010\ \ 0110\ \ 1100 = 3.A6C_{16}$

Similarly

$5.145_8 = 101\ 001\ 100\ 101 = 0101\ 0011\ 0010\ 1000 = 5.328_{16}$

- Just like in Base 10 the decimal point can be moved by multiplying by the appropriate power of the base.

  e.g. $101.11 = 1011 * 2^{-2} = 0.1011 * 2^3$

- Horner's Rule can also be applied to numbers with fractions

  e.g. $0.0101 = 0 + (0 * 2^{-1}) + (1 * 2^{-2}) + (0 * 2^{-3}) + (1 * 2^{-4}) = ¼ + 1/16 = 5/16 = 0.3125$

  Using Horner's rule

  $0.5(0 + 0.5(1+0.5(0+0.5 * 1))) = 0.3125$

Binary Arithmetic

```
  110101          101101          10110          11111101 / 1011
+  10010         -100110        * 1011
 1000111            111

   BA41            BA41
 + 14AF          - 14AF
   CEF0            A592
```

REPRESENTATION OF NUMBERS IN COMPUTERS

Because the storage capacity of a computer's memory and control circuitry is finite, it is necessary to group the bits it operates on into finite sequences. The size of bit groupings is an important factor in designing a computer.

Once a group of bits is decided upon there are only finite numbers of integers that can be represented by the group.

If there are n bits in a group the number of possible combinations of 0's and 1's is 2^n.

If the bits are used to represent non–negative integers, integers 0 through $2^n - 1$ can be represented.
With 8 bits integers 0 – 255 can be represented.

We usually estimate high powers when the groups are large where **n** may be 24, 32, 64 etc.
$2^{10} = 1024 = 10^3$
e.g. $2^{36} = 2^6.2^{30} = 2^6(2^{10})^3 = 2^6(10^3)^3 = 64 * 10^9$

If the result of any operation does not fit into the number of bits reserved for it an **overflow** is said to occur.
All the 4 arithmetic operations can cause an overflow.

$360 + 720 – 300 = 360 + (720 – 300)$ and $(360 + 720) – 300$

SIGNED INTEGERS

Normally a negative number is written by writing its magnitude and then placing a negative sign to the left of the magnitude of the number.

- A computer element can take only a 0 or 1; so a minus sign must be represented by a 0 or a 1.
- If a number is to be stored in n bits, the magnitude is placed in the $n - 1$ right most bits and the MSB represents the sign.
- A negative number is represented by a 1 and a positive number by a 0.
- Such a format is called the **Sign Magnitude Format**.

In this format there is a difference between – 0 and + 0 but they both have the same magnitude.

- The range of integers that can be expressed in a group of 8 bits is from $-(2^7 - 1) = -127$ to $(2^7 - 1) = +127$
- In general a *d* bit binary sign magnitude representation in which the first bit represents the sign has a range of
  $-(2^{d-1} - 1)$ to $+(2^{d-1} - 1)$.

In an 8 bit signed magnitude format, numbers

$00101011_2 = +43_{10}$	$10101011 = -43_{10}$
$01111111_2 = +127_{10}$	$11111111 = -127_{10}$
$00000000_2 = +0$	$10000000 = -0$

To add two sign magnitude numbers, we follow the usual addition rules.

- If the sign differs we subtract the smaller number from the larger number and give the result the sign of the larger number.
- If the signs are the same we add them and we give the result the same sign.

- +5 + -7 = 10000111 − 00000101 = 10000010 (-2)
  -5 + -7 = 10000101 − 10000111 = 10001100 (-12)

COMPLEMENTS

They are used to simplify subtraction & logical operations.

Consider numbers –1000 to 999 and let X be any number in that range.

The 4 digit 10's complement of X is defined as $10^4 - X$

If X = 0572, the 4 digit 10's complement of X is 10000 – 0572 = 9428

Consider the addition 0557 + - 328 = 229

The 10's complement of 328 = $10^4 - 328 = 9672$

Add 0557 *The 10's complement is used to represent its negative value*
+ 9672
1 0229 ← *Only 4 of the significant digits are saved.*
This one is lost

0557 + - 725 => $10^4 - 725 = 9275$ => 0557 +9275 = 9832

(i) A 9 in the most significant Digit indicates that the sum is negative. If the magnitude is wanted the 10's complement of the sum is taken.

i.e. $10^4 - 9832 = 0162$

(ii) N.B. The most significant digit is reserved to represent the sign leaving 3 digits for the magnitude.

2's Complement

The d digit 2's complement of a d bit binary integer N is equal to

$2^d - N$ where the subtraction is done in binary.

=> The eight bit 2's complement of an 8 bit binary number 000000101 is $100000000 - 00000101 = 11111011$

⇨ Subtraction of $01101010 - 01011011 = 00001111$

⇨ Using 2's complement, the complement of $01011011 = 100000000 - 01011011 = 10100101$

⇨ 01101010
+ 10100101
1 00001111

1 is lost

The 2's complement may also be computed by applying the following rules.

(1) Invert the bit values; the result is called **one's complement**.

(2) Add 1

e.g. for N = 00000101 —*Invert*→ 11111010 + 1 = 11111011

17_{10} = 00010001 —*Invert*→ 11101110 + 1 = 11101111 = -17

119_{10} = 00010001 —*Invert*→ 10001000 + 1 =10001001 = -119

N.B. Note the difference between the sign magnitude representation and the 2's complement representation.

Rules to convert to decimal:

(i) If a number is positive (beginning with a 0), convert it to base 10 directly as usual.

(ii) If it is negative (begins with 1) get its 2's complement and convert it to base 10.

e.g. to convert a 2's complement number 11111001 to decimal:

- ⇨ It is a 2's complement number; it is negative because it begins with a 1
- ⇨ Get its complement ; i.e. 11111001 *Invert* → 00000110 + 1 = 00000111 = - 7

Addition of 2's Complement Binary Integers:

Note that the left most bits are not treated separately as in signed magnitude numbers.

(a)	7	00000111	(b)	-7	11111001
	+ 5	00000101		+5	00000101
	12	00001100		-2	11111110
(c)	-7	11111001	(d)	7	00000111
	+- 5	11111011		+ -5	11111011
	-12	1 11110100		2	1 00000010

The carry is discarded

Overflow in 2's complement:

An overflow in 2's complement occurs when:

(i) The sign of the result differs from the sign of the common numbers being added OR

(ii) There is a carry into but not out of the MSB OR

(iii) There is a carry out of but not into the MSB.

e.g.				
126	01111110	-126	10000010
+ 5	+ 00000101	+ - 5	11111011
131	10000011	-131	01111101

FLOATING POINT FORMATS

It is a format for storing numbers given in scientific notation inside the computer. This helps to handle very large and very small numbers. It also helps to reduce leading and trailing zeros. It is written in the form

Fraction * base^{exponent}

e.g in base 10 $0.000000357 = 0.357 * 10^{-6}$, $625000000 = 0.625 * 10^{9}$

The fraction part is sometimes called the **significand** and the exponent the **characteristic.**

A floating point format is designated by:

(i) The base

(ii) The number of bits reserved for the exponent

(iii) The number of bits reserved for the fraction

(iv) The method for storing the sign and magnitude of the exponent

(v) The method for storing the sign and magnitude of the fraction.

(vi) The order in which the two signs and the two magnitudes are to occur.

The combination of the above factors for a given computer depends upon the designer.

The Typical floating Point Format:

Sign of Exponent	Magnitude of Exponent	Sign of fraction	Magnitude of fraction

←1 bit--→←----N bits---------- >< 1 bit > ←----------M bits--------→

- Although a base is chosen, it never appears in the format; once chosen it is fixed. Bases are always powers of 2, 8, and 16.
- 1 bit is needed for each sign.

- The total number of bits therefore in the FPF is N + M + 2 where N = bits reserved for the magnitude of the exponent and M = bits reserved for the magnitude of the fraction.

Designers usually determine

i. The resolution *(number of bits reserved for the magnitude of the fraction)* that the system must be able to accommodate and
ii. The largest and smallest non-zero magnitude that the system must be able to handle.

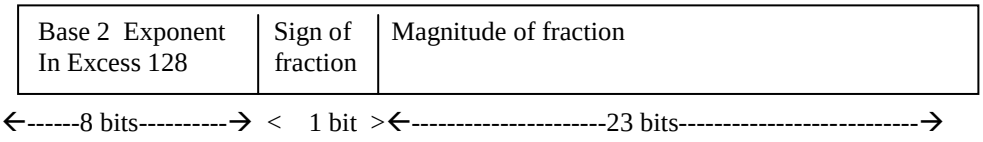
If base 2 is assumed the largest number that can be stored using a floating point format is approximately $2^{2^{N}-1}$ where N = number of bits reserved for the exponent.

If N = 7 the largest number is approximately
$2^{127} = 2^{7}(2^{10})^{12} = 128 * 10^{36} = 10^{38}$ and the smallest is $2^{-126} = 10^{-38}$

- If an operation results in a number that is so large that the maximum size of the exponent is exceeded an **exponent overflow** is said to occur.
- Similarly if the exponent is negative and the magnitude becomes too large, then an **exponent underflow** occurs.
- If N + 1 bits are reserved for the exponent and the sign, the offset or bias chosen = 2^{N} and the format is called the **excess 2^{N} format**
- If N = 7 the offset = 2^{7} = 128 = 10000000_{2} and the format is called the **excess 128 format**.
- When using the excess format the exponent is obtained from the quantity by subtracting the offset.
- In excess 128 format, the number 01111110 = 126 implies that the exponent = -2.

- In Excess 2^N format a 1 in the MSB represents a positive exponent and a 0 in the MSB represents a negative exponent.

The Typical 32 bit floating Point Format:



Consider a number $-13.6875 = -1101.1011_2 = -0.11011011 * 2^4$

⇨ The Exponent in excess 128 = $128 + 4 = 132 = 10000100_2$

⇨ Sign of the fraction is negative => 1

10000100 1 1101101100000000……
Exponent sign fraction = **84ED80000**

- To reverse consider a number 7E5C0000
- => Exponent = $7E = 01111110 = 126$ => $126 - 128 = -2$
  $5C = 0\ .101\ 1110$ => $0.101111 * 2^{-2} = 0.0010111_2 = 0.1796875_{10}$

Sign of fraction

Steps in Arithmetic

1. Pre-normalisation if the operands are not normalized.
2. Alignment
3. Post normalization of the result.

Examples

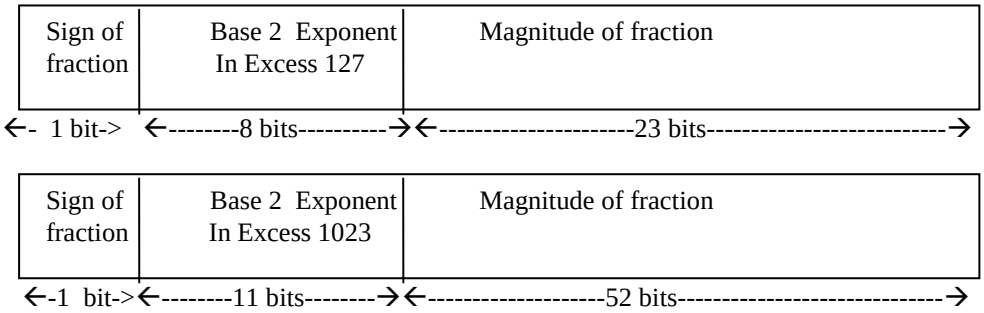
(1) 826013AC *Alignment* 826013AC
+8040AB04 → 82102AC1
82703E6D

(2) 83600000 *No Alignment*
+83C00000
83200000 *normalisation* → 82400000

(3) 82600000 *No Alignment*
x 85D00000
87BC0000 *normalisation* → 86F80000

The IEEE/INTEL 32 bit floating Point Format:

It has 2 forms; the single precision and the double precision format where N = 127 and 1023 respectively.



e.g. –5.375 = -101.011 = $-1.01011 * 2^2$
Exponent = 2 + 127 = 129 = 10000001
1 10000001 01011 ….. = **C0AC0000**

Conversely 3E600000 = 0011 1110 0110 0000 00000 …
=>Exponent = 01111100 = 124 – 127 = -3
=>Fraction = 0.11
Significant = $1.11 * 2^{-3} = 0.0011_2 = 0.21875_{10}$

Examples

(1) 40700000
+C0580000
40180000 ⟶ 3EC00000

(2) 40C00000 *No Alignment*
+ C0800000
40C00000 ⟶ **40000000**

(3) 41380000
*3F400000
410A0000

(4) C0A80000
/ 40E00000
BF400000

BINARY CODED DECIMAL (BCD)

In BCD each figure of the number to be coded is represented by its 4 bit binary equivalent e.g.

8159 = 1000 0001 0101 1001

There are two BCD formats.

1. **Packed (Condensed) BCD:** In this format each figure occupies half a byte; e.g. 341 = 0011 0100 0001

2. **Extended (unpacked) BCD:** Each decimal figure is represented by a byte. In this case the first 4 bits of a byte can be filled with zeros or ones depending on the manufacturer, e.g.

341 = 00000011 00000100 00000001

- Hardware designed for BCD is more complex than that for binary formats. E.g. 16 bits are used to write 8159 in BCD while only 13 bits (1111111011111) would be required in binary.

The advantage of BCD format is that it is closer to the alphanumeric codes used for I/Os.

Numbers in text data formats must be converted from text form to binary form. Conversion is usually done by converting text input data to BCD, converting BCD to binary, do calculations then convert the result to BCD, then BCD to text output.

To convert a BCD number to binary, multiply consecutively by 10 = 1010_2

e.g. 825_{10} = 1000 0010 0101

((1000 * 1010) + 0010)1010 +0101 = 1100111001.

The reverse conversion is done by successive divisions by 10 (1010_2) then using the 4 bit remainder as the BCD digit, e.g. Using the above example:

1100111001/1010 = 1010010 rem 0101; 1010010/1010 = 1000 rem 0010

⇨ 1000 0010 0101

Because only 10 of the 16 bit combinations are needed to represent the decimal digits, any 2 of the remaining combinations can be used to represent positive and negative signs.

The four bit combinations representing the sign can appear either to the left or to the right of the combinations.

The signs often used are 1100 for positive and 1101 for negative.

e.g. –34 = 0011 0100 **1101;** +159 = 0001 0101 1001 **1100**

BCD addition can be done by successively adding the binary representation of the digits and adjusting the results.

The adjustment rule is**: If the sum of 2 digits is > 9, add 6 to the sum.**

	1748	0001 0111 0100 1000
+	2925	0010 1001 0010 0101
	4673	0100 0000 0111 1101
		0110 0110
		0100 0110 0111 0011

ALPHANUMERIC CODES

It is the assignment of bit combinations to the letters of the alphabet, decimal digits 0 – 9, punctuation marks and several special characters.

The two most prominent Alphanumeric Codes are:

1. **EBCDIC** (*Extended Binary Coded Decimal Interchange Code*); this is mostly used by IBM.
2. **ASCII:** (*American Standard Code for Information Interchange*); used by other manufacturers.

ASCII represents each character with a 7 bit string

⇨ The total number of characters that can be represented is $2^7 = 128$

e.g. J O H N = 4A 6F 68 6E = 1001010 110111 1101000 1101110

Since most computers manipulate an 8 bit quantity, the extra bit when 7 bit ASCII is used depends on the designer. It can be set to a particular value or ignored.

EXPRESSION EVALUATION

- When writing an algebraic expression we use parenthesis to indicate the order in which the elementary operations are to be performed.

- If the ordering is specified by using parenthesis the expression is said to be in an **infix notation** because the operators are placed between their operands.

- The **postfix** or **reverse Polish Notation** places operators after the operands.

  e.g.: A/(B + C) = ABC+/

  (A + B) * [C * (D + E) + F] = AB+CDE+*F+*

  $$\frac{a + b * c(d + e)}{f * (g + h)} = abc * de + *+ fgh - * /$$

Evaluating Postfix Expressions

Uses LIFO Structure (Stack)

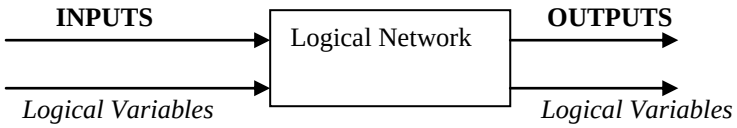
AB*CDE/-+ = (A * B) + (C – D/E)

ASCII CODE

ASCII Char	HEX Code	Control Character	ASCII Char	HEX Code	Control Character	ASCII Char	HEX Code	Control Character
NUL	00	Null	+	2B		V	56	
SOH	01	Start heading	,	2C		W	57	
STX	02	Start text	-	2D		X	58	
ETX	03	End text	.	2E		Y	59	
EOT	04	End transmission	/	2F		Z	5A	
ENQ	05	Inquiry	0	30		[	5B	
ACK	06	Acknowledgment	1	31		\	5C	
BEL	07	Bell	2	32		]	5D	
BS	08	Backspace	3	33		^	5E	
HT	09	Horizontal tab	4	34		_	5F	
LF	0A	Line feed	5	35		‘	60	
VT	0B	Vertical tab	6	36		a	61	
FF	0C	Form feed	7	37		b	62	
CR	0D	Carriage return	8	38		c	63	
SO	0E	Shift out	9	39		d	64	
SI	0F	Shift in	:	3A		e	65	
DLE	10	Data link escape	;	3B		f	66	
DC1	11	Device control 1	<	3C		g	67	
DC21	12	Device control 2	=	3D		h	68	
DC3	13	Device control 3	>	3E		i	69	
DC4	14	Device control 4	?	3F		j	6A	
NAK	15	Neg. acknowledge	@	40		k	6B	
SYN	16	Synchronous/Idle	A	41		l	6C	
ETB	17	End trans. Block	B	42		m	6D	
CAN	18	Cancel data	C	43		n	6E	
EM	19	End of medium	D	44		o	6F	
SUB	1A	Start special seq.	E	45		p	70	
ESC	1B	Escape	F	46		q	71	
FS	1C	File separator	G	47		r	72	
GS	1D	Group Separator	H	48		s	73	
RS	1E	Record separator	I	49		t	74	
US	1F	Unit separator	J	4A		u	75	
SP	20	Space	K	4B		v	76	
!	21		L	4C		w	77	
“	22		M	4D		x	78	
#	23		N	4E		y	79	
$	24		0	4F		z	7A	
%	25		P	50		{	7B	
&	26		Q	51		|	7C	
‘	27		R	52		}	7D	
(	28		S	53		~	7E	
)	29		T	54		DEL	7F	Delete rubout
*	2A		U	55				

LOGIC CIRCUITS

- Digital computers are based upon electronic components whose inputs and outputs are at anyone point in one of two possible states; the states are mostly voltage levels;
- One voltage level can be denoted by a 1 and another by a 0.
- If the higher voltage is associated with 1, the circuit is said to be based upon **positive logic.** If the lower voltage is associated with a 1, the circuit is said to be based on **negative logic**.
- A variable that can take on two states e.g. (0, 1, True, false; on/off) is called **a logical variable**.
- A circuit whose inputs and outputs are described by logical variables is called a **logical network**.



There are two types of logical networks:

1. **Combinatorial networks:** Their outputs depend on the current inputs.
2. **Sequential Networks:** Their outputs depend on both the current state of the network as well as the inputs.

Logic Gates

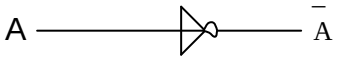
A combinatorial circuit with only one output is called a logic gate. They accept logical values at their inputs and they produce corresponding logical values at their outputs.

- A table listing all the outputs for the various inputs is called a **truth table** (derived from the True/ False logic in mathematics.)
- All combinatorial circuits can be constructed from the elementary logic gates.

The most common 7 logic gates are:

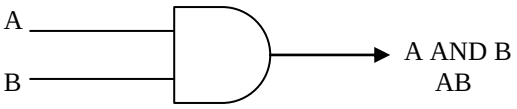
1. **The inverter(NOT) Gate**:

When the input is 1 the output is 0 and vice versa.



INPUT	OUTPUT
A	**$\bar{A}$**
0	1
1	0

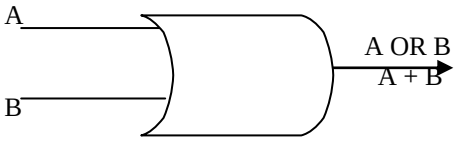
2. **The AND Gate**:



Input		Output
A	**B**	**AB**
0	0	0
0	1	0
1	0	0
1	1	1

The output is 1 if all the inputs are 1's

3. **The OR Gate**:

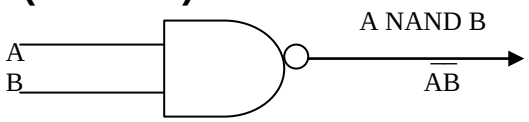


Input		Output
A	**B**	**A + B**
0	0	0
0	1	1
1	0	1
1	1	1

The output is 0 if all the inputs are 0's.

If an inverter is combined with another logic gate, the presence of the inverter is indicated by placing a small circle at the affected input or output.

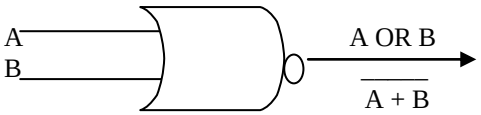
4. The NAND (NOT AND) Gate:



Input		Output
A	B	$\overline{AB}$
0	0	1
0	1	1
1	0	1
1	1	0

The output is 0 if all the inputs are 1's.

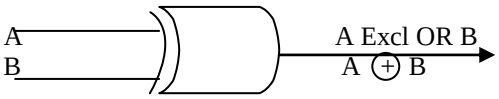
5. The NOR (Not OR) Gate:



Input		Output
A	B	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

The output is 1 if all the inputs are 0's.

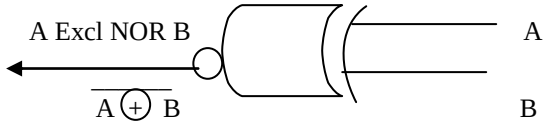
6. The EXCLUSIVE OR Gate:



Input		Output
A	B	A ⊕ B
0	0	0
0	1	1
1	0	1
1	1	0

The output is 0 if all the inputs are the same. OR
It outputs a 1 if the number of 1's in the inputs is odd.

7. **The EXCLUSIVE NOR Gate**:



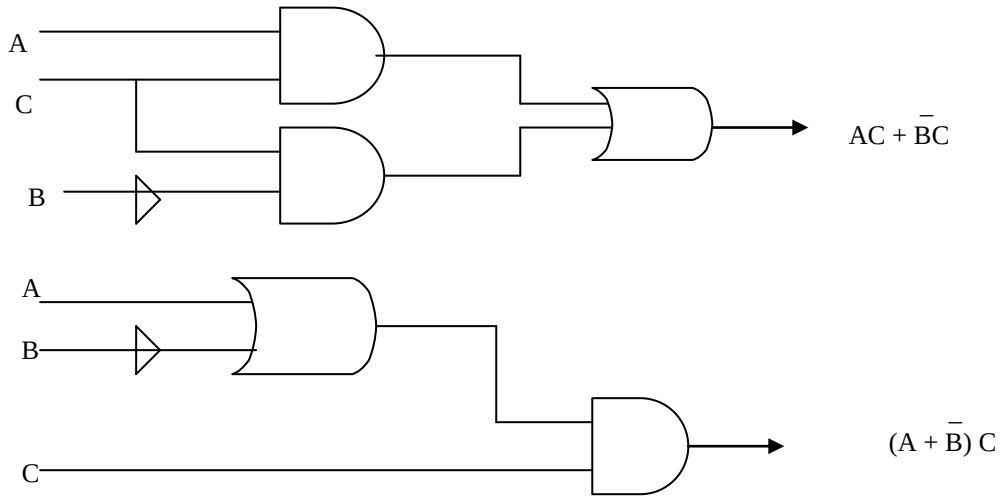
Input		Output
A	**B**	$\overline{\mathbf{A \oplus B}}$
0	0	1
0	1	0
1	0	0
1	1	1

The output is 1 if all the inputs are the same.

Complex Logic gates

Complex logic circuits can be built by combining several of the elementary logic gates.

A graphical illustration of a logic circuit is called a **logical diagram.**



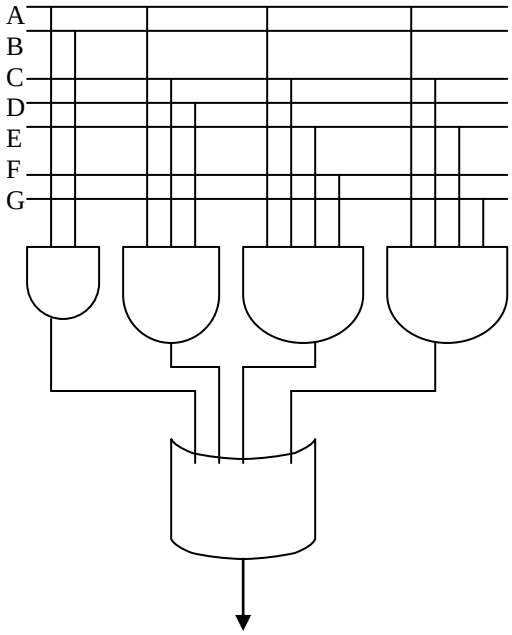
Two logic circuits are said to be equivalent if they have exactly the same input /output relationship.

One way to prove equivalence is to compare the **truth table** outputs of the two circuits.

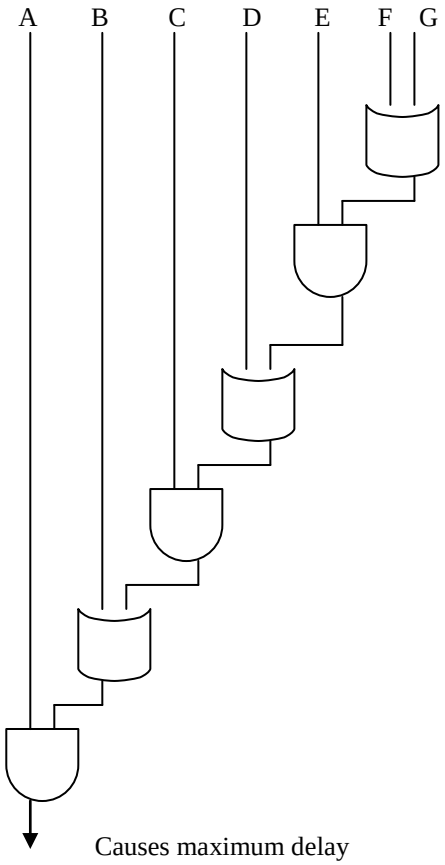
A	**B**	**C**	$\overline{\mathbf{B}}$	**AC**	$\overline{\mathbf{B}}$**C**	**AC +** $\overline{\mathbf{B}}$**C**	**A +** $\overline{\mathbf{B}}$	**(A +** $\overline{\mathbf{B}}$**)C**
0	0	0	1	0	0	**0**	1	**0**
0	0	1	1	0	1	**1**	1	**1**
0	1	0	0	0	0	**0**	0	**0**
0	1	1	0	0	0	**0**	0	**0**
1	0	0	1	0	0	**0**	1	**0**
1	0	1	1	1	1	**1**	1	**1**
1	1	0	0	0	0	**0**	1	**0**
1	1	1	0	1	0	**1**	1	**1**

To try Circuits for AB(C+D) and (ABC + ABD) and their Truth Tables.

X = AB + ACD + ACEF + ACEG



(Causes minimum delay)



Causes maximum delay
minimum number of inputs

BOOLEAN ALGEBRA

It is a mathematical structure that consists of a set containing only a 0 and 1, the unary operator (complementation) and the binary operation of addition and multiplication. Subtraction and Division are not defined in Boolean algebra. This branch of mathematics was developed by George Boole in 1847.

Some of the equivalencies mostly frequently used in reducing Boolean Expressions

A	=	$\overline{\overline{A}}$
AA	=	A
$A + A$	=	A
$A \cdot 0$	=	0
$A + 0$	=	A
$A \cdot 1$	=	A
$A + 1$	=	1
$A \cdot \overline{A}$	=	0
$A + \overline{A}$	=	1
AB	=	BA
$A + B$	=	$B + A$
$(AB)C$	=	$A(BC)$
$A + (B + C)$	=	$(A + B) + C$
$A(B + C)$	=	$AB + AC$
$(B + C)A$	=	$BA + CA$
$\overline{A + B}$	=	$\overline{A}\,\overline{B}$
$\overline{AB}$	=	$\overline{A} + \overline{B}$
$AB + A\overline{B}$	=	A
$A + AB$	=	A
$(A + \overline{B})B$	=	AB
$(A + B)(A + \overline{B})$	=	A
$(A + B)(A + C)$	=	$A + BC$
$A(A + B)$	=	A
$A\overline{B} + B$	=	$A + B$
$\overline{A}B + A\overline{B}$	=	$A \oplus B$
$\overline{AB}(A + B)$	=	$A \oplus B$

e.g.

$\bar{A}\bar{B}C + BC + AB$

⇨ $C(\bar{A}\bar{B} + B) + AB$

⇨

⇨ $C(\bar{A} + B) + AB$ (*third equivalence from the bottom*)

⇨

⇨ $\bar{A}C + BC + AB$

⇨

⇨ $\bar{A}C + (\bar{A} + A)BC + AB$

⇨

⇨ $\bar{A}C + \bar{A}BC + ABC + AB$

⇨

⇨ $\bar{A}C(1 + B) + AB(C + 1)$

⇨

⇨ $A\bar{C} + AB$

$\bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$

⇨ $\bar{A}BC + ABC + A\bar{B}C + ABC + AB\bar{C} + ABC$

⇨

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

⇨ $BC + AC + AB$

Digital Design Process

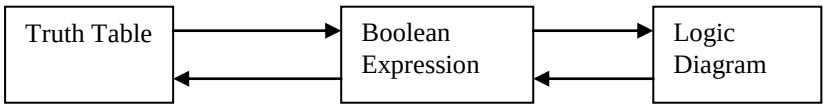
1. Determine all the input/output relationships that must be true for the network being designed and put them in convenient tabular form.
2. Use the drawn up table to find Boolean expressions for each output.
3. Simplify the expressions from 2 above
4. Use the expressions resulting from step 3 to develop the desired logical diagram.

Three design tools have been introduced:

1. Truth Table (*To define a logical network*)
2. Boolean Expression *(for minimization)*

3. Logic diagram *(For the actual design)*

A designer must be able to convert from one form of a network's description to another.



Example

Suppose that a three input network is needed that will output a 1 if the majority of the inputs are 1's otherwise the output is zero.

Step 1 (Draw a truth Table)

A	**B**	**C**	**X**	
0	0	0	**0**	X_0
0	0	1	**0**	X_1
0	1	0	**0**	X_2
0	1	1	**1**	X_3
1	0	0	**0**	X_4
1	0	1	**1**	X_5
1	1	0	**1**	X_6
1	1	1	**1**	X_7

Step 2 (Find the Boolean expression.)

There are two types of Boolean Expressions

(i) SUM OF PRODUCTS (SOP's)

A product is 1 if and only if all of its factors are 1

A sum is 1 if at least one of its terms is a 1.

$X_3 = \bar{A}BC$; $X_5 = A\bar{B}C$; $X_6 = AB\bar{C}$; $X_7 = ABC$

$X = X_3 + X_5 + X_6 + X_7$

$\Rightarrow \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$

(ii) PRODUCT OF SUMS (POS)

A product is 0 if at least one of its factors is 0

A sum is 0 if all its terms are 0's.

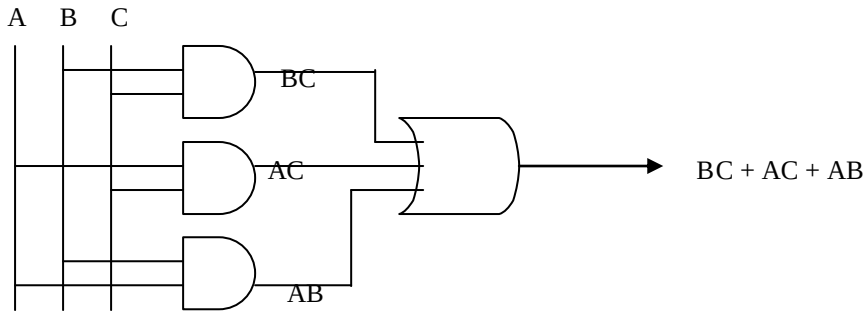
$X_0 = A + B + C$ $X_1 = A + B + \bar{C}$ $X_2 = A + \bar{B} + C$ $X_4 = \bar{A} + B + C$

$X = X_0X_1X_2X_4$ $=>(A + B + C)(A + B + \overline{C})(A + \overline{B} + C)(\overline{A} + B + C)$

Step 3 Simplify the expression

$=> \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC = BC + AC + AB$

Step 4 Draw the logic diagram.



Minimisation of expressions using Karnaugh Maps

- The occurrence of a variable or its complement in an expression is called a **literal**.
- A term in the SUM OF PRODUCTS that includes a literal for every input is called a **miniterm**.
- A term in the PRODUCT OF SUMS that includes a literal for every input is called a **maxiterm**.

e.g. in $A\overline{B}C + \overline{A}BC + A\overline{C}$; $A\overline{B}C$ and $\overline{A}BC$ are miniterms, $A\overline{C}$ is not a miniterm.

Similarly in $(\overline{A} + B + \overline{C})(A + \overline{B} + \overline{C})(\overline{A} + C)$, $(\overline{A} + B + \overline{C})$ and $(A + \overline{B} + \overline{C})$ are maxiterms, $(A + \overline{C})$ is not.

A Karnaugh map is a truth table for a single output consisting of arrays of squares where each square corresponds to a row of a truth table.

The symbols at the top represent the variables associated with the columns and the symbols on the left represent the variables associated with the rows.

The value of each output for each input is put in the corresponding square.

For each 1 in the Karnaugh map there is a corresponding miniterm in the output's Sum of product expression and each 0 represents a maxiterm in the Product of Sums expression.

Two inputs

B \ A	0	1
0	0	2
1	1	3

Three inputs

C \ AB	00	01	11	10
0	0	2	6	4
1	1	3	7	5

Four inputs

CD \ AB	00	01	11	10
00	0	4	12	8
01	1	5	13	9
11	3	7	15	11
10	2	6	14	10

Example 1

A	B	C	X
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Using Boolean simplification:

$$ABC + AB\bar{C} + \bar{A}BC$$

$$ABC + AB\bar{C} + ABC + \bar{A}BC$$

$$= \mathbf{AB + BC}$$

AB

C \ AB	00	01	11	10
0	0	0	1	0
1	0	1	1	0

Look for adjacent groups that include 2^n miniterms where n is an integer. The larger the group the greater is the reduction.

Example 2

A	B	C	X
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

Using Boolean simplification:

$$\bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$$

$$\bar{A}BC + ABC + A\bar{B}C + ABC + AB\bar{C} + ABC$$

$$= BC + AC + AB$$

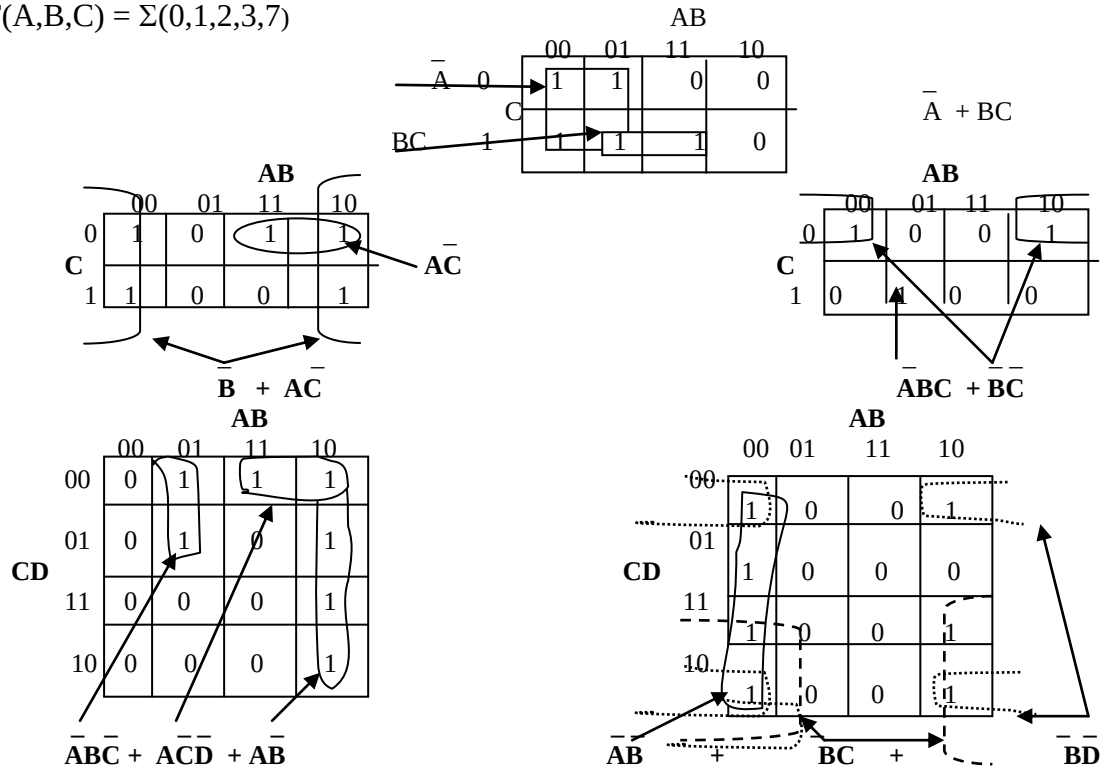
AB

C \ AB	00	01	11	10
0	0	0	1	0
1	0	1	1	1

A function may be used to state where the output is 1 instead of drawing a Karnaugh map. e.g. for the above example F(A,B,C) = Σ(3,5,6,7)

Example 3

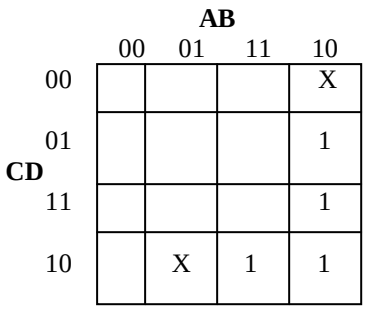
F(A,B,C) = Σ(0,1,2,3,7)



For some designs some input combinations cannot occur. Outputs corresponding to these combinations are optional. These combinations may be chosen to be either 0 or 1 as is convenient in the maximization process.

These outputs are represented by X's in the Karnaugh Map and they may or may not be included in the prime implicants.

They are called **Don't Care Case**s denoted by the function **d(A,B,C) = Σ(….)**



$\bar{A}B$ + AC

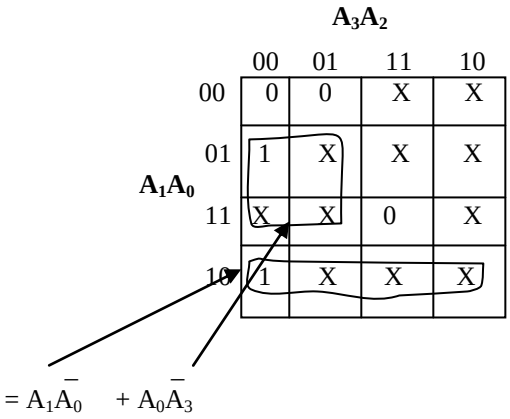
It is important to include the miniterm $A\bar{B}\bar{C}\bar{D}$ in C but not $\bar{A}BC\bar{D}$

Example

A network is needed that will output a 1 if the binary $A_3A_2A_1A_0$ is greater than 0 and less than 4.

Assume also the inputs are controlled by a rotary switch such that only one input can be a 1 at any given time except when the switch position will also allow all inputs to be 1.

A_3	A_2	A_1	A_0	X
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	X
0	1	0	0	0
0	1	0	1	X
0	1	1	0	X
0	1	1	1	X
1	0	0	0	0
1	0	0	1	X
1	0	1	0	X
1	0	1	1	X
1	1	0	0	X
1	1	0	1	X
1	1	1	0	X
1	1	1	1	0



BINARY ADDER

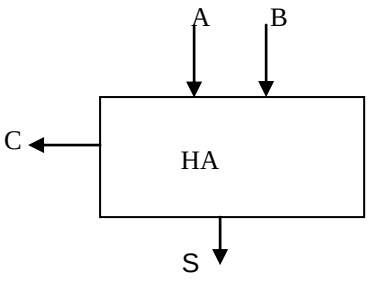
A circuit for adding 2 1 bit quantities is called a half adder.

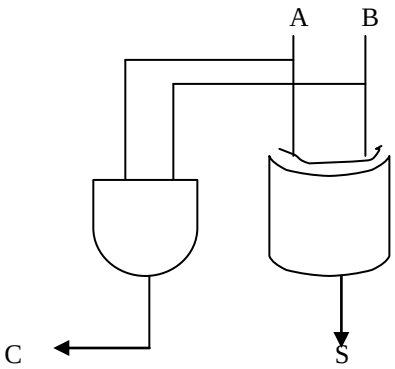
```
    A
+   B
  Carry  sum
```

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$C = AB$

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

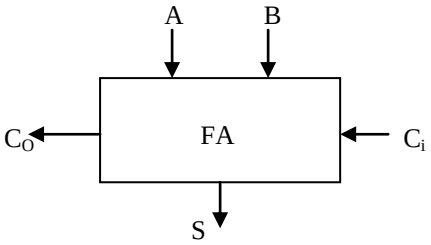




Full Adder

For a 2 bit addition, addition of a higher order bit takes into account a possible carry from the low order sum.

The adder that includes a carry input from a lower order sum is called a Full Adder.

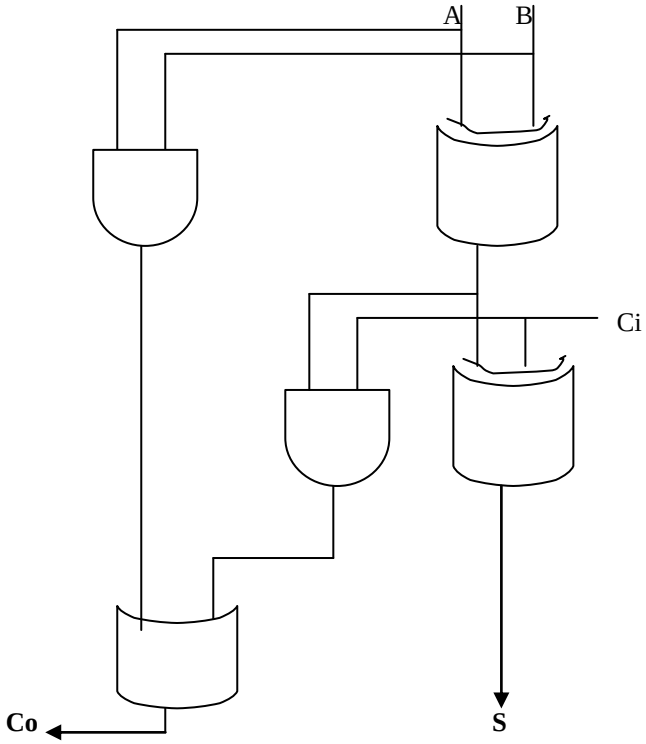


A	**B**	**C_i**	**S**	**C_o**
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$\mathbf{C_o} = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$

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

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



$\mathbf{S} = \bar{A}\bar{B}C + \bar{A}B\bar{C} + A\bar{B}\bar{C} + ABC = \bar{A}(\bar{B}C + B\bar{C}) + A(\bar{B}\bar{C} + BC)$

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

To build an adder of more bits, you just duplicate the full adders the required number of times.

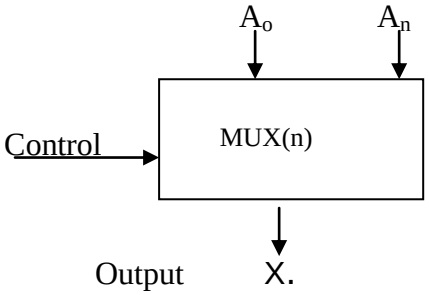
Such a circuit is called a **Ripple Carry Adder**. The carry out bit is used as a carry in into its left neighbour. The carry into the right most bit is set to 0.

MULTIPLEXER (*Data Selector***)**

It is a logical network capable of selecting a single set of data inputs from a number of sets of inputs and it passes the selected inputs to the outputs.

A multiplexer has 2 kinds of inputs, the *control inputs* and the *data inputs.*

The control inputs serve to select which of the inputs in the data is to be passed through to the outputs.



The simplest multiplexer is the 2 to 1 multiplexer. It has 2 data inputs, A and B, one control input and one output

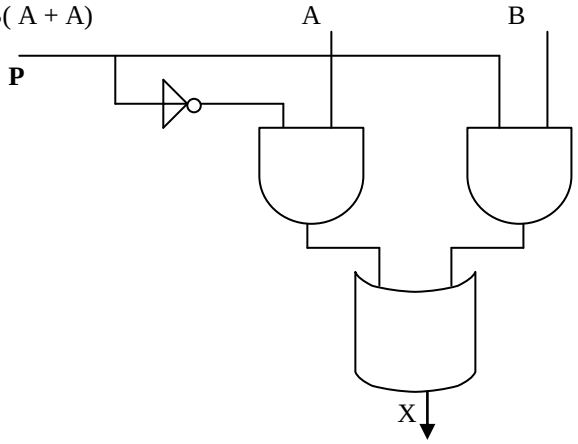
When the control input P = 0, the output X is A and when P = 1, X = B.

P	**A**	**B**	**X**
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

$X = \bar{P}\bar{A}B + \bar{P}AB + P\bar{A}B + PAB$

$= \bar{P}A(\bar{B} + B) + PB(\bar{A} + A)$

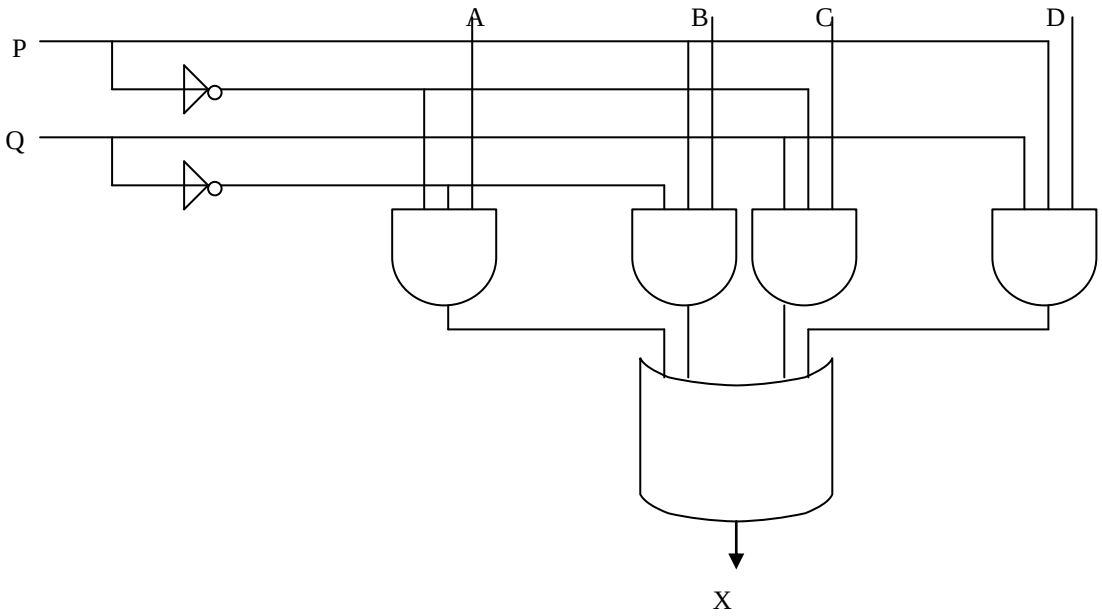
$= \bar{P}A + PB$



A 4 to 1 multiplexer

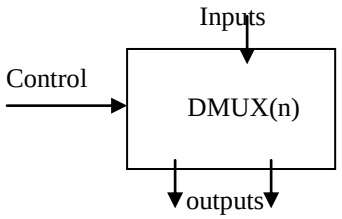
It selects only one of the 4 inputs. It has 2 control lines to choose one of the 4 possible inputs.

In general for an n to 1 multiplexer, the inequality $2^K >= n$ must be satisfied where k = number of control lines.



A Demultiplexer

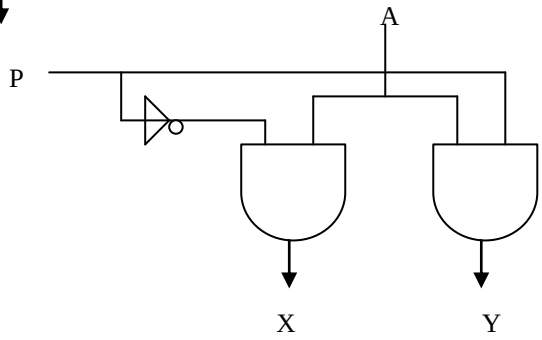
It has 1 set of data inputs and two or more sets of outputs and a set of control inputs whose purpose is to select the set of outputs to transmit. The other outputs are 0.



P	A	X	Y
0	0	0	0
0	1	1	0
1	0	0	0
1	1	0	1

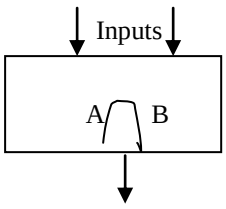
$X = \bar{P}A$

$Y = PA$



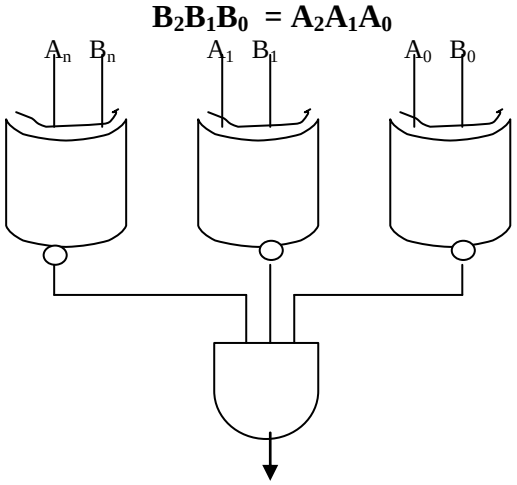
A comparator

Compares two sets of inputs and outputs a 1 if the comparison is satisfied. The comparisons are =,!=, >, >=, <, <=



The equality comparator

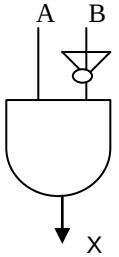
The design is based upon the XNOR gate which outputs a 1 if its inputs are the same and a 0 if they are not equal.



1 bit > comparator

X = 1 if A > B

A	B	X	=> $X = A\bar{B}$
0	0	0	
0	1	0	
1	0	1	
0	0	0	



2 bit > comparator X = 1 if $A_1A_0 > B_1B_0$

A_1	A_0	B_1	B_0	X_o :
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

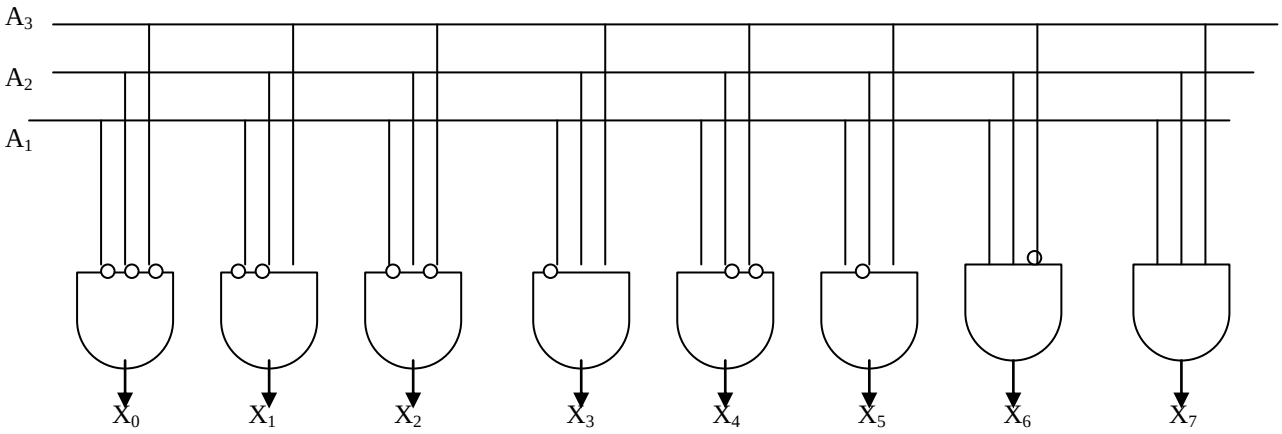
Decoder

Circuit whose outputs are miniterms of the inputs. Exactly only one output is a 1 at any given time.

If **n** is the number of inputs and **m** the number of outputs then $2^n >= m$

e.g. if the binary number on the input lines is **k** then output line k will be 1 and all the others will be 0's.

A_1	A_2	A_3	X_0	X_1	X_2	X_3	X_4	X_5	X_6	X_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1



The Encoder

The opposite of a decoder. Only one input can be a 1 (activated) at a time. Its number in binary is presented as the output.

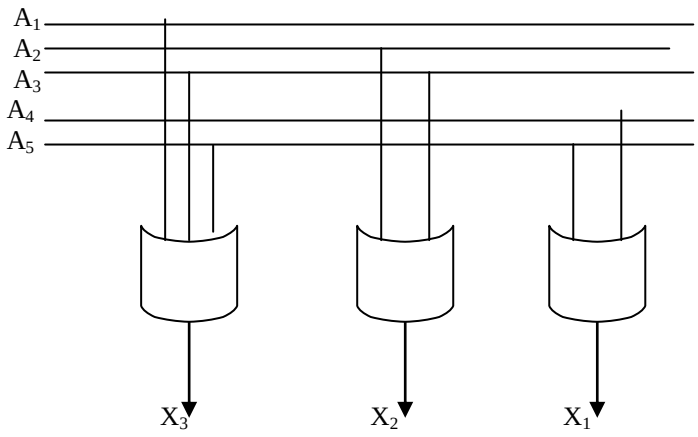
It has 2^n inputs and n outputs.

A_0	A_1	A_2	A_3	X_1	X_2
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	1	1	1	1

Inputs

A_1	A_2	A_3	A_4	A_5	X_1	X_2	X_3
1	0	0	0	0	0	0	1
0	1	0	0	0	0	1	0
0	0	1	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	0	0	1	1	0	1

$\mathbf{X_3 = A_1 + A_3 + A_5}$ $\mathbf{X_2 = A_2 + A_3}$ $\mathbf{X_1 = A_4 + A_5}$



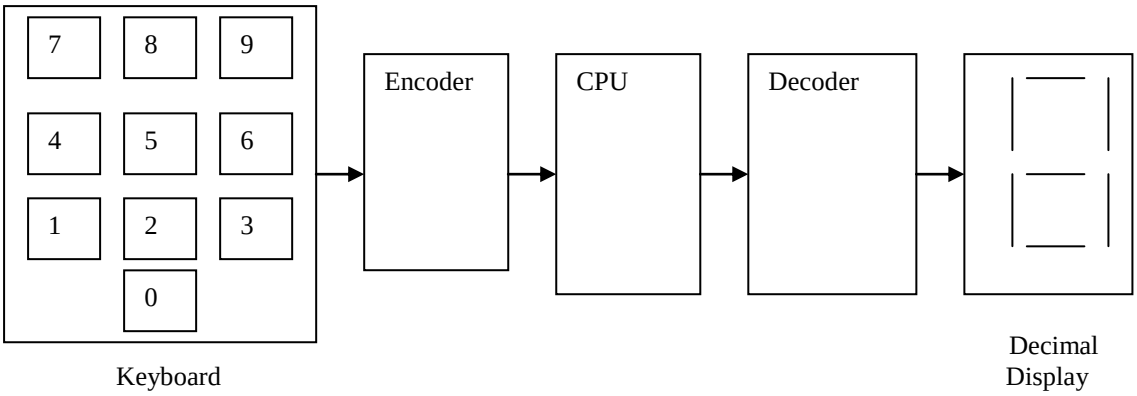
Code Converters

They are electronic circuits whose purpose is to convert data from one format to another.

Data in a computer system may take on several different forms as it changes from one format to another.

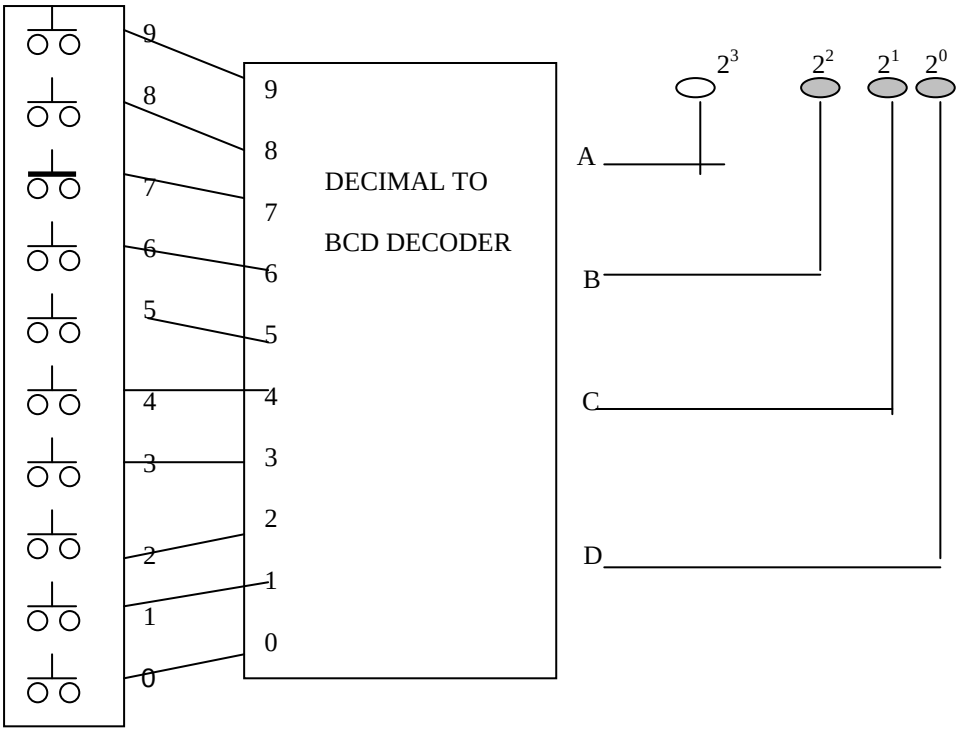
e.g. the decimal input from a keyboard calculator must be converted into BCD using an encoder.

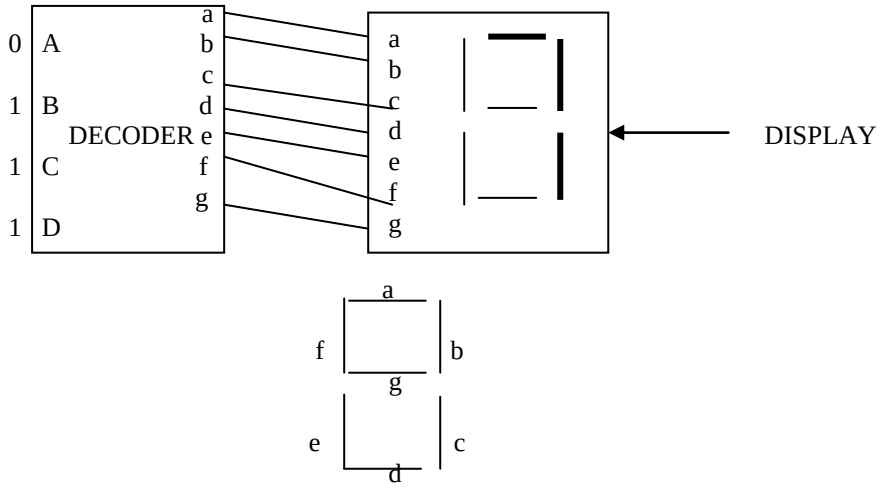
The CPU's output is in BCD and the decoder translates the BCD to a special 7 segment display code by a decoder.



The encoder circuit

The encoder has 10 active inputs and 4 outputs connected to input lamps. The input 7 causes a BCD output of 0111.





A	**B**	**C**	**D**	**a**	**b**	**c**	**d**	**e**	**f**	**g**
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	0	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	0	0	1	1

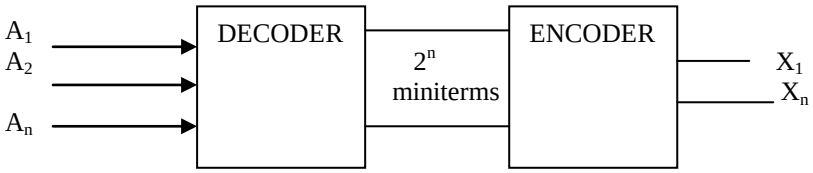
ROMS AND PLA'S

ROM

Read Only Memory. The circuit is equivalent to a decoder that outputs all possible miniterms of the inputs followed by an encoder.

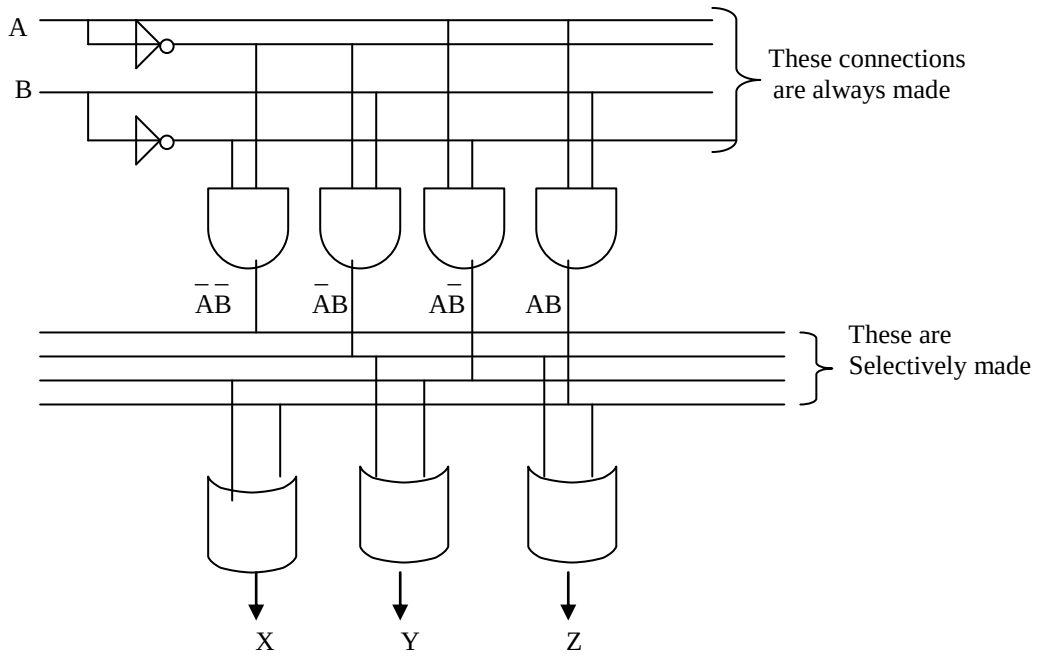
The output combinations are permanently embedded in its circuitry and the inputs serve to select one of these combinations. Each output is obtained by disconnecting the OR inputs from the AND gates whose miniterms are not to be included in the output.

Because a ROM must produce all the possible miniterms its decoder portion is fixed by n. Its encoder portion however depends on both the outputs and the way in which all the outputs of a decoder are used to generate the final ROM outputs.



Once the disconnections are made they cannot be changed. (ROM nature). e.g to construct a 2 input (A, B) three output (X, Y, Z) ROM such that

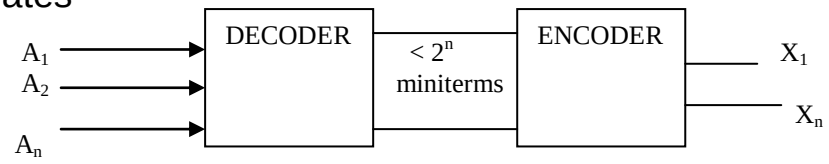
X = $A\bar{B}$ + AB; **Y = $\bar{A}B$ + $A\bar{B}$** **Z = $\bar{A}B$ + AB**



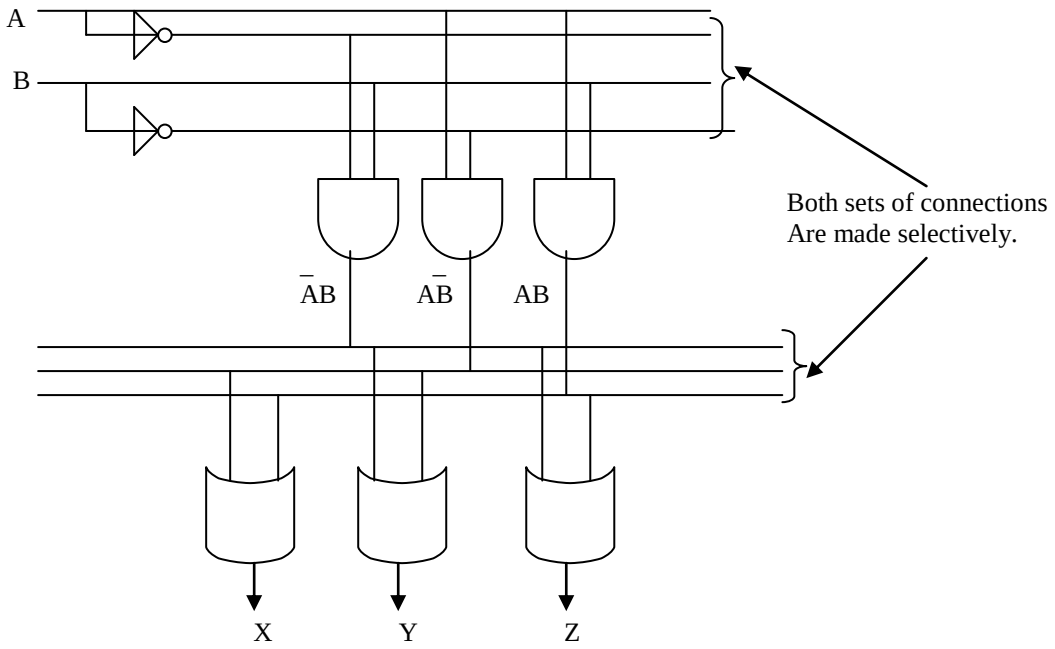
When a ROM is used as memory the inputs are the memory's address and the outputs are the contents of the address.

PLA (Programmed Logic Array)

Similar to a ROM but does not output all the miniterms that will not be needed in any of the outputs. i.e. the decoder does not necessarily produce all the miniterms. For an n input network, we have <= 2^n AND gates



e.g. a three AND gate PLA implementation of the ROM in the previous example.



SEQUENTIAL LOGIC & COMPUTER CIRCUITS

Combinatorial circuits have no memory, do not contain feedbacks and they are time independent. They cannot therefore be used for storage. Their outputs solely depend on their inputs. They have no intrinsic timing control.

Sequential circuits have internal states that can be used to store information and modify their inputs.

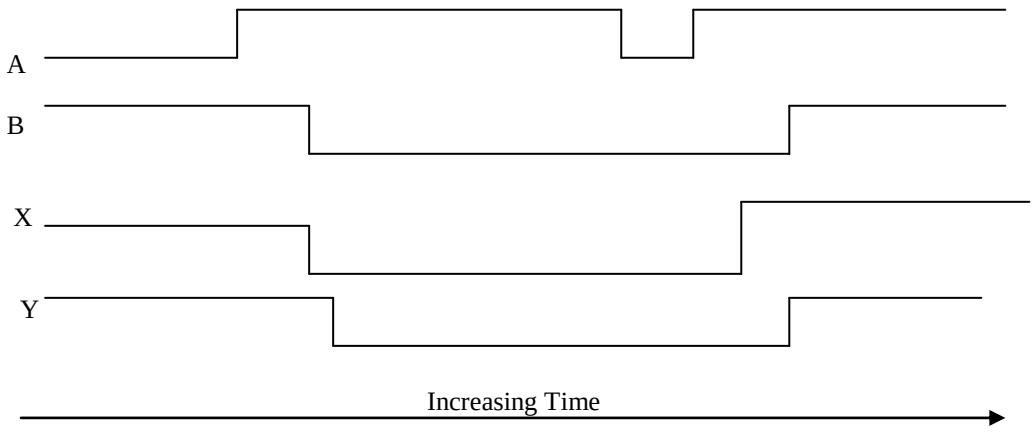
E.g.

(i) **A simple counting device**; its output depends on the input signal that causes the output to increment and also on the current count that was previously determined.

(ii) **A memory circuit**; the input causes the contents of memory to be applied to the outputs.

Time in a sequential circuit takes on a significant role.

A sequential network is defined according to its inputs and outputs over a period of time. The aid used in examining the time dependent aspect of a sequential network is called a **timing diagram.**



A begins in state 0, B and X begin in state 1.

Transition of A to 1 does not change X.

Transition of B from 1 to 0 causes X to change to 0.

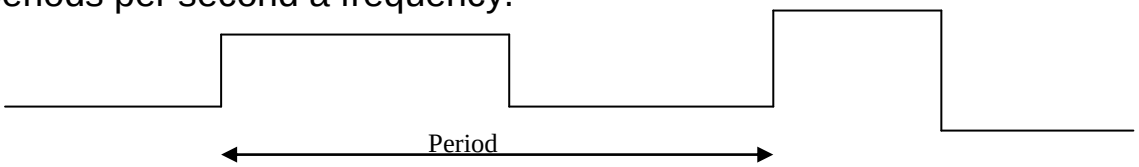
Transition of A from 1 to 0 does not affect X but the following transition of A causes X to change to 1.

Output Y changes state 25 μs after a change in X.

The S shaped curves indicate that the state of Y at a particular time is not known but a change in state will occur. If it was a 0 it will change to 1 and vece versa.

Sequential networks are synchronised by a time standard called a clock. A clock generates an evenly spaced train of pulses.

Time between consecutive pulses is called a period. The number of periods per second a frequency.



Elementary sequential circuits fall into a class of binary electronic circuits known as multvibrators which may be astable, monostable or stable.

Astable multivibrators

The cannot maintain a fixed state but the keep on switching back and forth between its states.

Monostable multivibrators

They can take on two states but are stable in only one of them. They can only temporarily stay in the unstable state.

Bistable

They are stable in either of the 2 states and can therefore maintain either state indefinetely.

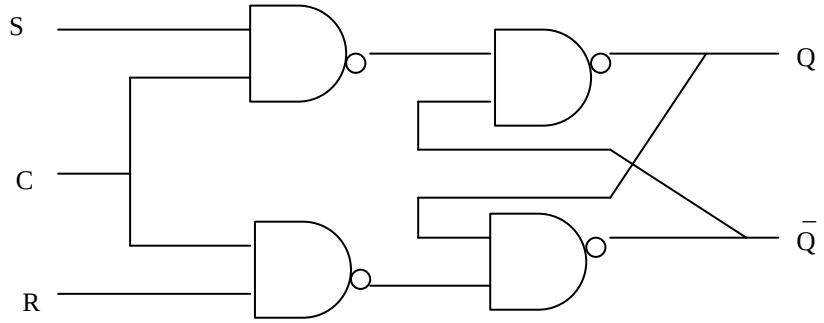
Flip Flops

They are bistable devices i.e. they are 2 state devices in nature. They assume one of the 2 possible states and they will maintain that current state until an external excitation causes it to change state.

They are devices used in sequential networks.

The most common flip-flops are the R-S, J-K, T, and the D flip flop.

The R-S flip flop (Reset –Set)



R	**S**	$\mathbf{Q^+}$	$\mathbf{\bar{Q}^+}$
0	0	Q^-	Q^-
0	1	1	0
1	0	0	1
1	1	-	-

It has 3 inputs and 2 outputs.

A clock input synchronises the action of the flip-flop with its surrounding network. S and R (Set / Reset Inputs). The two outputs (Q and $\bar{Q}$) are always in opposite states from each other. Because both R and S inputs are ANDED with the clock, they have no effect on the state of the flip-flop when the clock is in 0 state. The network is then stable. In this state,

if Q = 1, $\bar{Q}$ is = 0 and Q is maintained at 1. If $\bar{Q}$ is 1 Q = 0 and $\bar{Q}$ is maintained at 1. If the clock is raised to 1 the network will not change if R = S = 0

If R = 1, S = 0 => Q = 0 and $\bar{Q}$ = 1

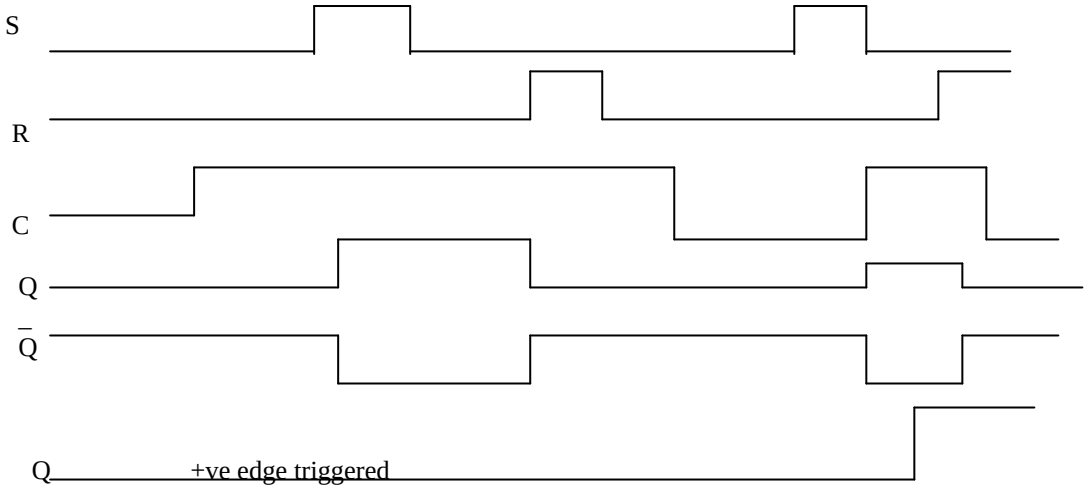
R = 0, S = 1 => Q = 1 and Q = $\bar{0}$

Most significant changes occur when there is a clock transition.

If R = S = 1 => Q = $\overline{Q}$ = 1 is not a useful combination of inputs; this is not defined in the table.

The subscripts Q^- and $\overline{Q}^-$ indicate outputs just before the clock becomes 1 and the subscripts Q^+ and $\overline{Q}^+$ show outputs just after the clock becomes 1. When the clock is high (i.e. = 1) the output follows the changes in the inputs.

R and S = 0 => no changes in outputs.If S is pulsed, Q will be set (becomes 1) regardless of its previous state and Q $\overline{\text{w}}$ill be cleared. => (becomes 0). If R is pulsed, Q is cleared (or reset) and $\overline{Q}$ will become 1.



If the clock input is constant, the outputs will follow the changes in the inputs at all times.

All clocked flip flops that react to their inputs anytime C = 1 are called **latches**.

The R-S flip-flop is called a latch because it uses the clock inputs to determine whether or not the inputs will be recognised.

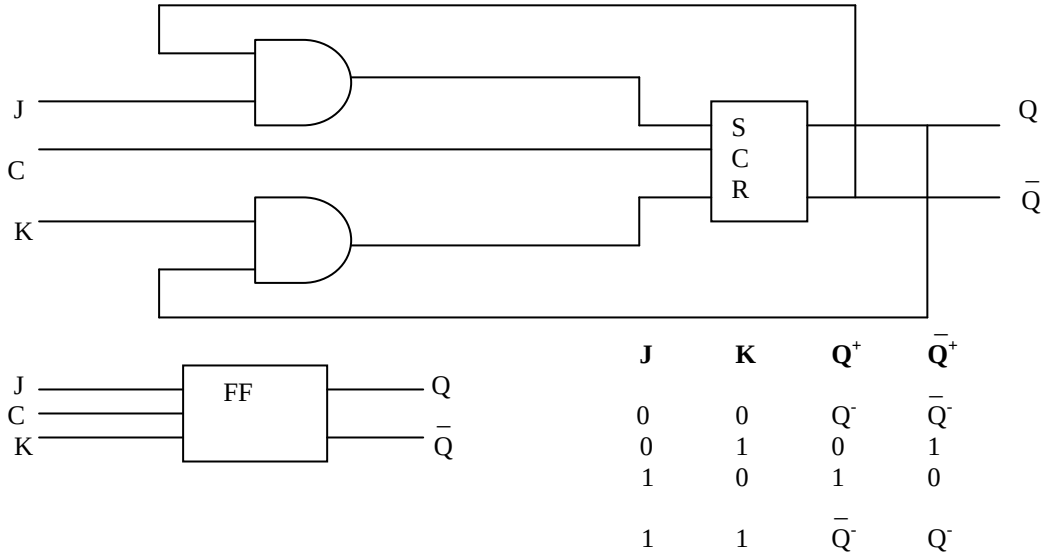
If a flip-flop changes only at the very beginning (or the very end) of a clock pulse it is called an **edge triggered** flipflop. They change state only when there is a 0 to 1 transition at C (+ve edge triggered) or a 1 to 0 transition at C (-ve edge triggered)

A change from 0 to 1 is a +ve transition and the +ve transition of a clock pulse is called the **leading edge**.

A change from 1 to 0 is a negative transition and the negative transition of a clock is called a **trailing edge**.

The J-K Filp Flop

It is an R-S flip flop that has been modified by feeding the outputs back and ANDYING them with the inputs. It has the same behaviour like the R-S flip flop except that the C = J = K = 1 combination is meaningful and the results in the output states is reversed.

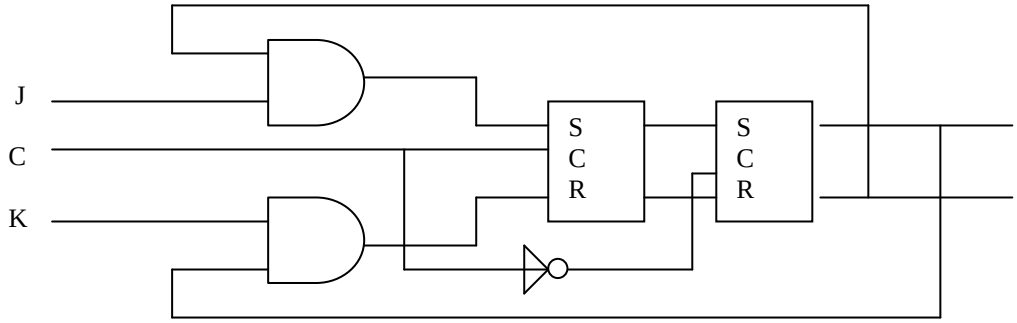


J	K	Q^+	$\bar{Q}^+$
0	0	Q^-	$\bar{Q}^-$
0	1	0	1
1	0	1	0
1	1	$\bar{Q}^-$	Q^-

The J-K flip –flop is constructed from an edge triggered R-S flip-flop otherwise the C=J=K=1 state would be unstable.

A modification of the J-K flipflop is the master-slave J-K flip-flop. The master responds to the +ve level of the clock ; the slave responds to the –ve level of the clock.

By properly adjusting the trigger levels on the master and slave the slave can be disconnected from the master while the master is being set and the master can be disconnected from the inputs while the slave is being reset.



With a simple J-K flip-flop an input transition while it is being triggered can cause the output to be momentarily unstable.

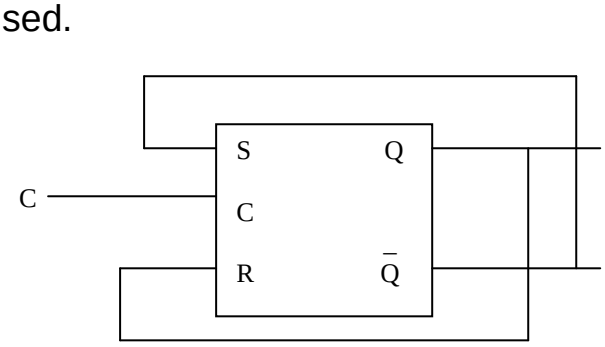
The isolation of the master from the slave while the master is being set eliminates this problem and guarantees a clear switching action at the output.

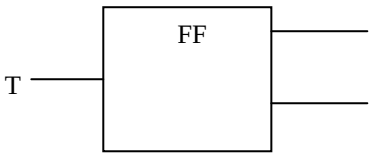
Toggling (change of state) occurs when the clock pulse changes from high to low.

The J-K flip-flop is –ve edge triggered flip-flop. i.e. it takes a low to activate the clock.

The T Flip –Flop (Toggle)

It has only one input. Its output states are reversed each time the input is pulsed.





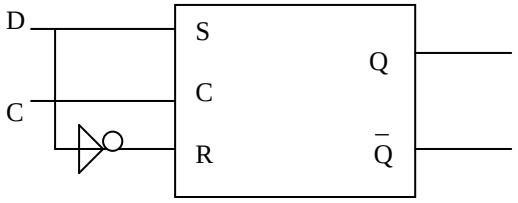
T	$\mathbf{Q^+}$	$\mathbf{\overline{Q}^+}$
0	Q^-	$\overline{Q}^-$
1	$\overline{Q}^-$	Q^-

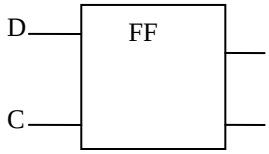
It is used in the design of counters.

A T flip flop can be obtained from a J-K flip flop by permanently applying 1’s to the J and K inputs. The R-S flip flop must be edge triggered.

The D flip flop (Data)

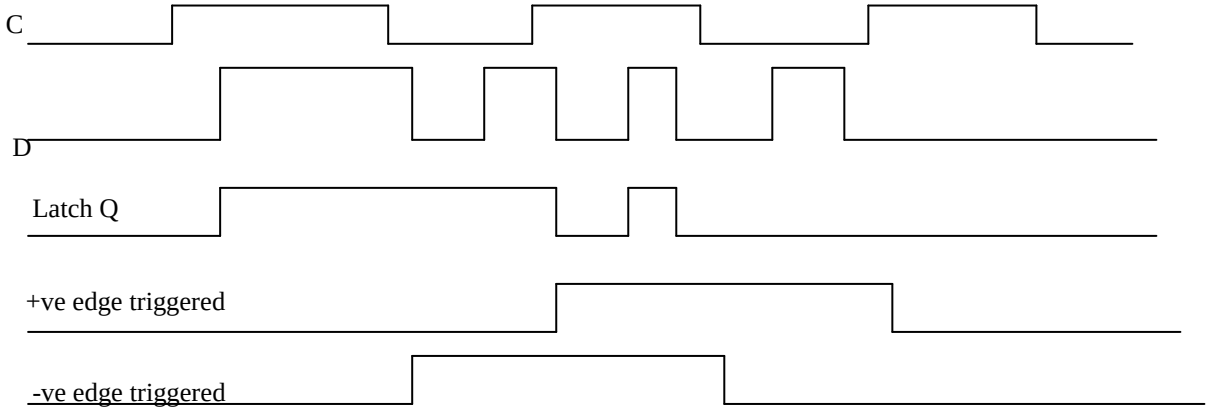
It has 2 inputs, a clock input and an input labelled D such that the Q output is equal to the D input whenever the clock input is set to 1; otherwise it is not affected by the D input.





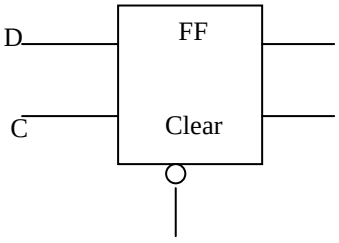
D	$\mathbf{Q^+}$	$\mathbf{\overline{Q}^+}$
0	0	1
1	1	0

It is used in sequential networks like in registers . It is easily constructed from the R-S flip flop by letting the D input be S input and connecting R to D through an inverter.

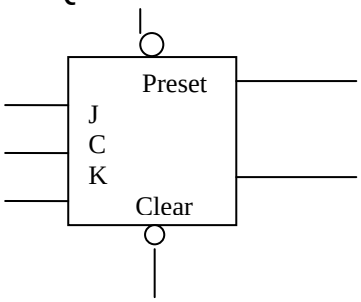


Clear and Preset Inputs

Flip flops can clear or set the Q output irrespective of the state of the other inputs. The clear input clears Q and the Preset input sets Q



A D flipflop with a **Clear** feature



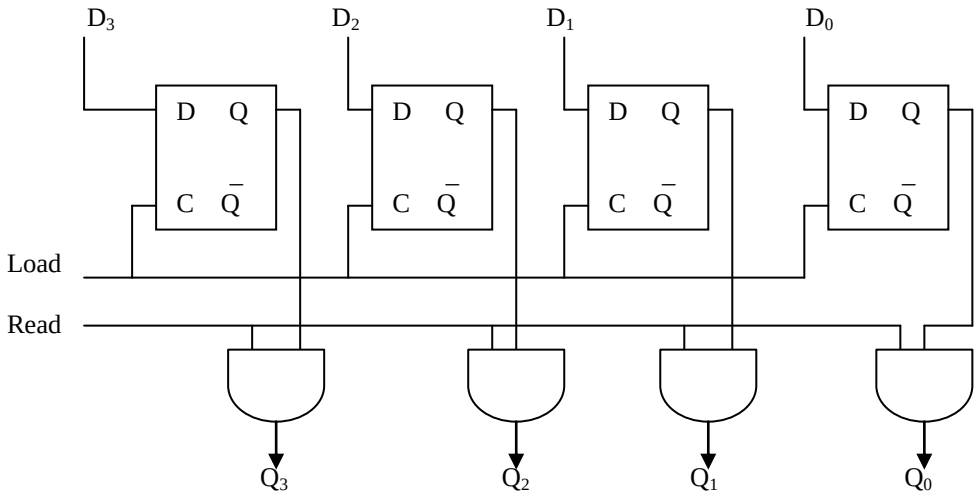
A J-K flipflop with **clear** and **preset** inputs

The ———○ indicates a negative logic nature whereby a 0 input initiates the action instead of a 1 input.

REGISTERS

A number of flipflops placed in parallel to form several bits of storage. Each flip flop is capable of storing 1 bit of information. Registers are used anywhere in the computer where it is necessary to store a number of bits.

e.g. a 4 bit register constructed from four D flipflops



The common clock load permits new information to be loaded.

The common Read line and associated AND gates provide a controlled Read out mechanism.

If n flip flops are used, the register is said to have a length of width n.

The currently stored data are the current states of the flip flops and can be monitored at the Q outputs.

In general a register consists of a group of flipflops and gates. The flip flops hold the binary information and the gates control when and how new information is transferred into the register.

Transfer of new information into the register is known as loading the register.

If all bits of the register are loaded simultaneously with a common clock pulse transition we say that the loading is done in parallel.

Shift registers

Frequently it is necessary to shift the bits in a register either to the left or to the right. A register that is capable of performing this function is called a shift register.

Shifting bits is necessary because some registers must be able to re-arrange their contents. e.g. if the flip flops in an 8 bit register contain 0 1 1 0 0 1 0 1 and a 1 bit left shift operation is performed, the new contents of the register will be 1 1 0 0 1 0 1 0.

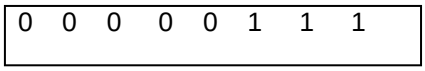
After a 1 bit right shift the new contents would be 0 0 1 1 0 0 1 0

For the left shift the left most bit is lost and 0 is inserted on the right.

If the left most bit is brought around and put in the right bit during a left shift, or the right most bit is put in the left bit during a right shift the operation is called a **rotation**.

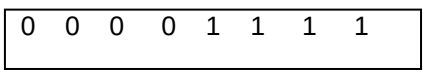
RLC (Rotate Left)

The content of the accumulator is rotated left one position. The low order bit and the carry flag are both set to the value shifted out of the high order bit position.



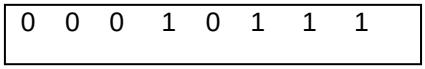
RRC (Rotate Right)

The content of the accumulator is rotated right one position. The high order bit and the carry flag are both set to the value shifted out of the low order bit position.



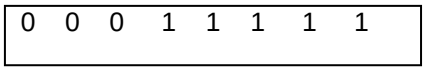
RAL (Rotate Left through carry)

The content of the accumulator is rotated left one position through the carry flag. The low order bit is set equal to the carry flag and the carry flag is set to the value shifted out of the high order bit position.



RAR (Rotate Right through carry)

The content of the accumulator is rotated right one position through the carry flag. The high order bit is set equal to the carry flag and the carry flag is set to the value shifted out of the low order bit position.



E.G

Consider initially A = 01101001 and the carry flag (c) = 1

RLC 11010010 RRC 10110100
RAL 11010011 RAR 10110100

Multiplication by 2 has the same effect as shifting a binary number left by 1 bit and another way of performing a left shift is to add the A register to itself.

One way of programming the multiplication of two single precision integers is to zero the pair of registers that will hold the result and successively examine the bits of the multiplier starting with the least significant bit. If the bit is 1 the multiplicand is added to the product register pair otherwise no addition is done; then the multiplicand is shifted one to the left and the next bit in the multiplier is tested. The process is continued until all the multiplier bits have been examined.

Shift registers are classified according to their inputs and outputs. Both the input and output may be classified as serial or parallel.

Serial input is one for which the input arrives 1 bit at a time, and each time a bit arrives, the register is shifted by 1 to accommodate the new bit.

A parallel input is one for which the inputs are all loaded at the same time.

Serial and parallel outputs are similarly defined.

Shift registers are used in

- Interfaces for changing the form of data that are to be transmitted or are being received.
- Processing elements (microprocessors) for performing packing, unpacking, bit searching and arithmentic operations.

Data Transmission

Shift registers are used most importantly in converting different types of data communications.

Most computers include 2 types of binary data transmission.

If n bits are to be transmitted they could be sent simultaneously over n signal paths. This is called **parallel data transmission**.

Alternatively they could be sent one after the other over 1 signal path. This is called **Serial Data transmission**.

In parallel transmission, the number of lines employed is greater than the number of bits. The extra lines are the **control lines** that are used by the transmitting device to signal to the receiving device when data is ready to be read and the receiving device to signal to the transmitting device that the data has been read.

The passing back and forth of signals on the control lines during transmission is called **handshaking**.

Sometimes one of the control lines transmits clock pulses and the timing of all the other signals is controlled by these pulses. Data transmission in this case is said to be **synchronous.**

A transmission that is not controlled by a common clock signal is said to be **asynchoronous**.

In serial transmission data is sent over a single line in sequential fashion.

For each character the transmitter and receiver divide the period of time used to transmit data into sub intervals. One subinterval is a bit.

Control lines may again be used for handshaking or synchronisation.

Serial transmission is often made over a single pair of lines and the beginning and end of transmission are marked by special bits called a start bit, a stop bit and parity bits.

A character transmitted in the asynchronous serial mode consists of the following 4 parts:

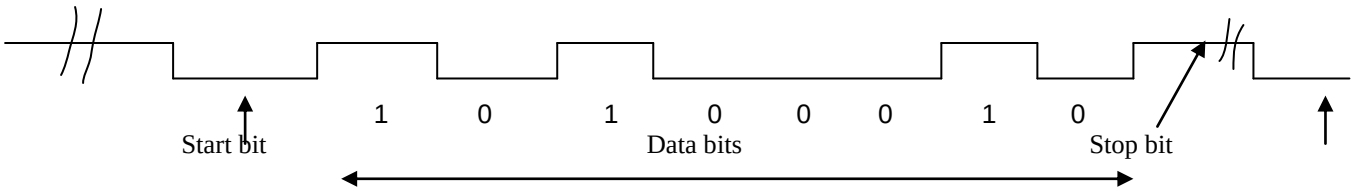
1 A start bit

2 Five to 8 data bits

3 An optional even / odd parity bit

4 1 or 2 stop bits.

A timing diagram to transmit an ASCII character E = 45 with 1 start bit and 1 stop bit



At the end of each character the signal always goes to a logical 1 for the stop bit. It remains 1 until the start of the next character which begins with a start bit at logical 0.

The logical 1 and logical 0 are respectively knows as the **mark** and **space**.

Advantages of Parallel transmission

Higher information transfer rate can be attained.

Disadvantages

More wires (or communication channels) are needed.

Whenever distance is a factor serial transmission is chosen.

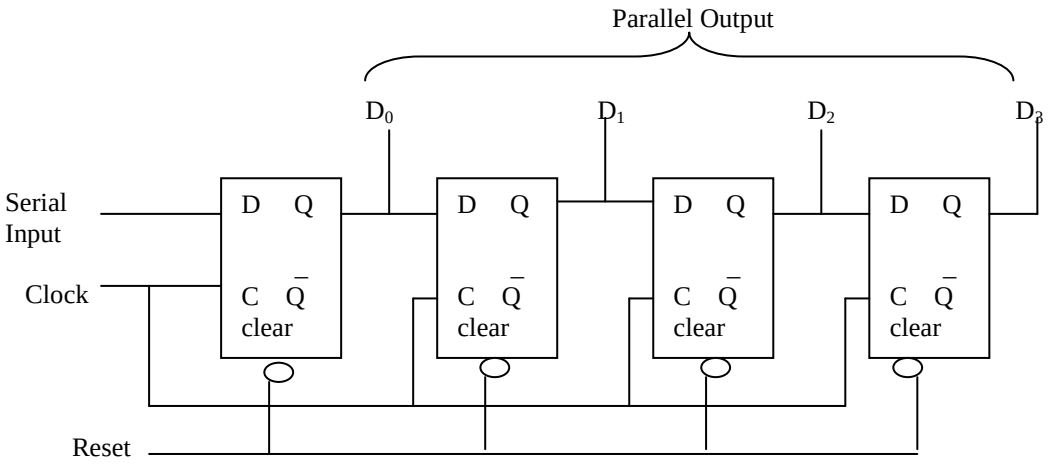
If the transfer rate must be high parallel communication may be required.

Because a computer system includes both types of data communications, it must also include means of converting from one type to another.

Serial to parallel converter

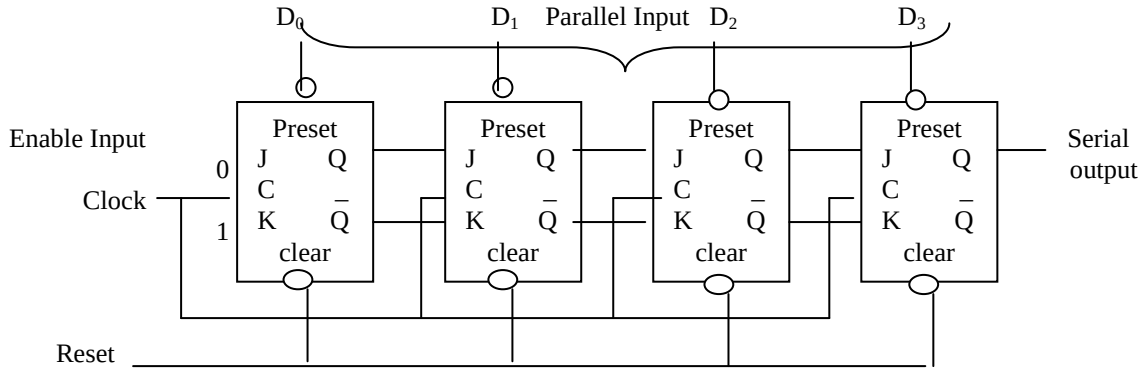
A negative logic reset precedes each data character thus clearing the shift register.

Each clock pulse loads a data bit until the register is full. After the 4 bit character is loaded it can be read from the parallel output lines $D_0 – D_3$



Parallel to Serial converter

Once data is made available at $D_3 – D_0$ it can be loaded into the shift register by load signal. The 4 clock pulses are used to cause the 4 bit character to appear sequentially at the output



Binary Counters

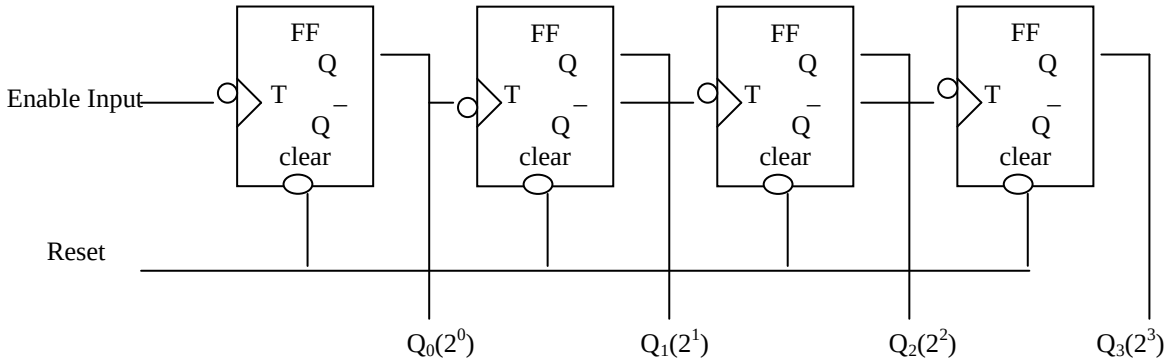
They are circuits used to count and store the number of pulses arriving at its input.

Counters are found in almost all equipment containing digital logic.

They count the number of occurrences of an event and they are useful for generating timing signals to control the sequence of operations in digital computers.

A counter that follows the binary number sequence is called a binary counter. An n bit binary counter is a register of n flip flops and associated gates that follows a sequence of states according to the binary count of n bits from 0 to $2^n – 1$

e.g. a 4 bit counter capable of counting from 0000 through 1111.



The T flip flops are negatively edge triggered.

Incrementing takes place on the trailing edge of the input pulses.

The enable input provides a means of turning the counting process on and off without removing the clock signal from the flip flop and the reset input clears the counter.

Going through a sequence of binary numbers e.g. 0000 0001, 00010 etc, the lower order bit is complemented after every count and every other bit is complemented from one count to the next if all its lower bits are equal to 1.

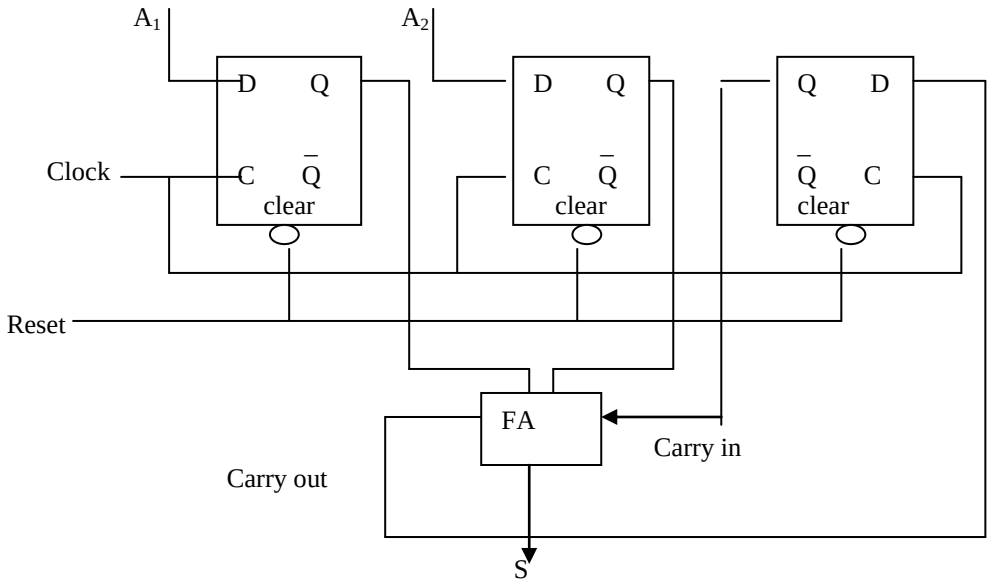
A sixteenth pulse has the same effect as the Reset input for it clears all the 4 bits.

A counter circuit employs flip flops with complementing capabilities like a J-K flip flop when C = J = K = 1 or a T flip flop.

Serial Adders and subtractors

The combinatorial adders and subtractors are parallel devices because they receive all the bits in the operands in parallel.

It is possible to design serial arithmetic circuits that receive their operands one bit at a time.



Because carries and borrows must be saved until the next bit arrives, these circuits must have memories. They must be sequential.

A serial adder is a FA with D flip flops in three input lines.

The carry out output is fed back into the carry in flipflop so that it will provide the carry in for the next bit.

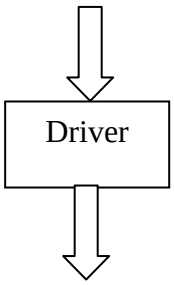
The flipflops must be reset to 0 and the lower order bits must be received first.

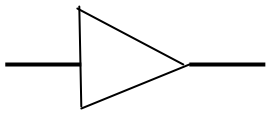
Link Connections

When joining distinct logic circuits together with a link you need to:

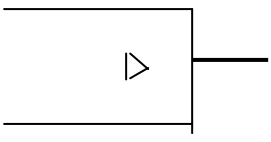
- Convert from voltage signals to current signals or viceversa.
- Increase the power of the signal
- Electrically disconnect a logic circuit from the link
- Connect several logic circuits to the same set of conductors.

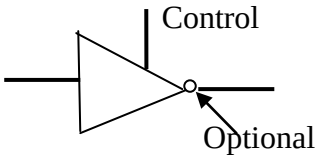
The first two problems are resolved by using circuits referred to as **drivers**. A driver that is on the receiving end of a transmission is called a **receiver**; combinations that both transmit and receive are called **transceivers**.





The third problem is solved by a **tristate driver** or **tristate gate** whose output may be 0, 1 or a high impedence state.



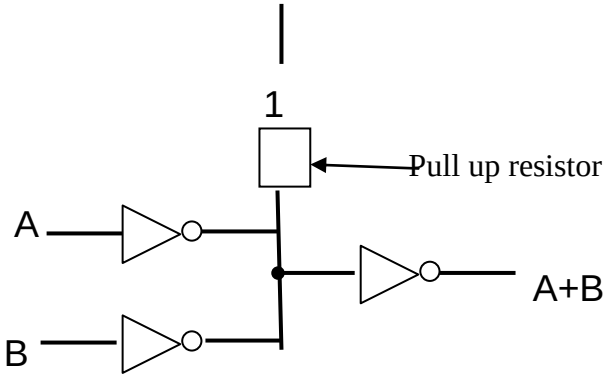


IEEE standard tristate
Output symbol

A tristate driver has two inputs, a data input and a control input. When the control input is 1, the output is the same as the input (or the complement of the input if the driver is also the inverter); when the control input is 0 the input is disconnected from the output (the high impedence state).

Control	Input	Output
0	0	Disconnected
0	1	Disconnected
1	0	0
1	1	1

The fourth problem can be solved by using a wire-ORed gate (open gate collector) whose output can be directly tied to the outputs of other wire-ORed gates without damaging any gate.The state at the common point is 1 if all the gate outputs would normally be 1; otherwise it is a zero. A resisitor called a ***pull up resisitor*** is placed between the common output and the state 1 voltage.



Sequential Network Design

The design of a sequential network is primarily based upon the states of the network. The behaviour of a sequential circuit is determined from the inputs, the outputs and the state of the flipflops.

A state is determined by a 0 & 1 combination of the outputs of the flipflops of the network.

If a network contains **n** flip flops, the possible number of states would be 2^n. The actual number of states that the network can be in may be less

because the construction of the network may not allow some input combinations to occur.

If **m** is the number of states that can occur then $2^n >= m$.

Each state is assigned a unique 0 – 1 combination of the outputs of the flipflop. Both the outputs and the next state are a function of the inputs and the present state.

A sequential circuit is specified by:

(i) **A State Table** that relates the next state as a function of the inputs and the present state. In clocked sequential circuits, the transition from the current to the next state is activated by a clock signal.

(ii) **Output Table**: gives the outputs as a function of the current state.

Example: assume 2 inputs A and B, five states S0 –S4 and three outputs X, Y, Z.

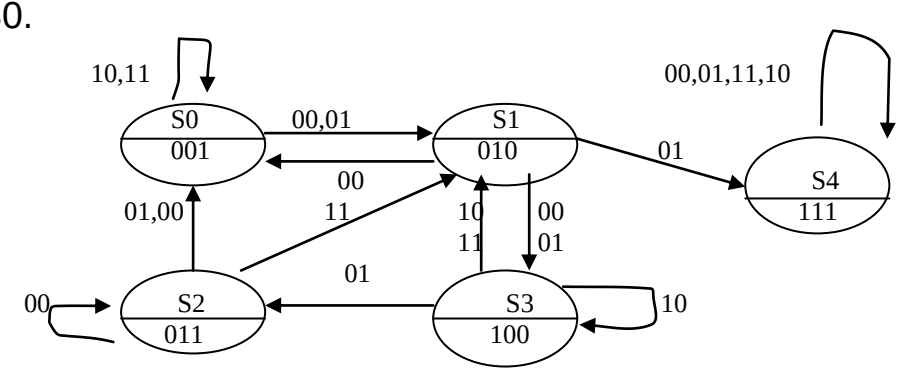
State table

	Inputs			
	00	**01**	**11**	**10**
S_0	S_1	S_1	S_0	S_0
S_1	S_0	S_4	S_3	S_3
S_2	S_2	S_0	S_1	S_0
S_3	S_1	S_2	S_1	S_3
S_4	S_4	S_4	S_4	S_4

Output Table

		Outputs		
		X	**Y**	**Z**
	S_0	0	0	1
	S_1	0	1	0
new state	**S_2**	0	1	1
	S_3	1	0	0
	S_4	1	1	1

If the current state is S2 and the input is 01, then the new state will be S0.



Information in the state table and output tables can be combined into a state diagram where circles represent the states and outputs and the arrows represent the transition between states.

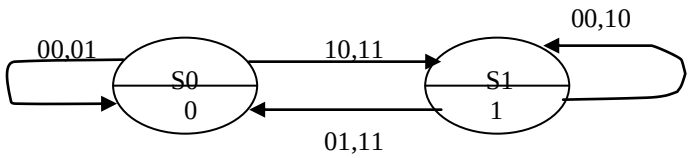
Example:

The state table, output table and state diagram of a network consisting of only one J-K flip flop

State table

	JK				Output Table	
	00	**01**	**11**	**10**		**X**
S_0	S_0	S_0	S_1	S_1	**S_0**	0
S_1	S_1	S_0	S_0	S_1	**S_1**	1

S_0 corresponds to Q = 0 S_1 to Q = 1



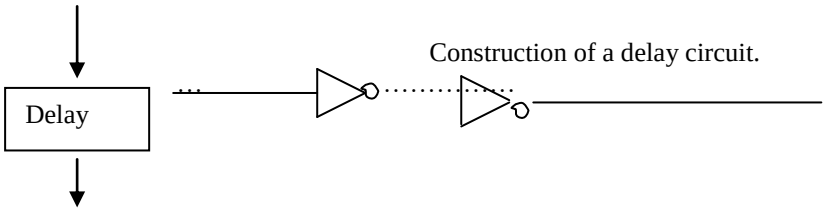
Delays

No electronic circuits react instataneously whether they are combinatorial or not. The length of the delay depends upon the resistances and capacitances built within the circuits.

Sometimes delays are undesirable, other times they are desirable such that if natural delays are not enough special circuits called **delay devices** are included in a design to create the required delay.

For a delay device the output is the same as the input except that the output occurs at a later time.

Amount of delay is referred to as the **delay time.**



Construction of a delay circuit.

The amount of time it takes for a logic gate's or flipflop's output to reflect its inputs is called its **switching time** and the **propagation delay** is the time **i**t takes electromagnetic signals to travel through the circuit and links.

If 2n inverters are used to build a delay device then the delay time = 2n * switching time of the inverter.

(Read Timing considerations and output devices)

INTEGRATED CIRCUITS AND TECHNOLOGIES.

Modern digital logic circuits are constructed onto the surfaces of thin slices of a base material e.g. silicon. The resulting products are Integrated devices (IC's) which are normally 0.1cm^3 to 1cm^3 in volume but may contain hundreds of thousands of logic gates.

Advantages of fabricating circuits containing large numbers of gates into a small area include:

1. The small transistors used to make the gates use very small power.
2. Capacitances are small. So the swiching time is very little.
3. Distances are short so that propagation delays are short.
4. The number of soldered connections is limited to the relatively low number of connections to the circuitry outside the IC.

Disadvantages:

2. Because of the small size of the IC's transistors, the amount of power that an IC can output may be small.
3. Because of the small surface area, special means may be needed to dissipate the heat generated by the IC.

Limitations in putting all the computer's circuitry into a single IC include:

1. Fabrication techniques and heat dissipation requirements limit the density of the transistors in a circuit.
2. An IC's phyisical size is limited.

Reliability of an IC depends on its temperature and the materials used to make the IC.

For an IC to be functional, its temperature must be kept below a limit that is characteristic of the materials. Special cooling may be used to keep the temperature down thereby allowing the density or speed of the IC's to be increased.

For some fast computers like the supercomputers, the processing elements are submerged in liquid nitrogen.

The other restriction is due to the fact that flaws occurring during IC manufacturing cannot be corrected and therefore any flaw would cause the entire IC worthless.

The probability of a manufacturing flaw increases as the area of the IC's surface increases. The percentage of the rejects is proportional to the size of the IC.

The ratio of the good IC's to the total produced is called the YIELD.

A method of constructing an IC is called a technology. It is determined by both the geometry of a transistor and the materials used.

As the technology of IC's has improved the number of gates that can be put on a chip has also increased.

SSI Small Scale Integration (SSI) devices contain several independent gates in a single package. The inputs and outputs of the gates are connected directly to the pins in the package. The number of gates is usually less than 10 and it is limited to the number of pins available to the IC.

Medium Scale Integration (MSI): between 10 – 500 gates on a single chip. They are used in elementary digital functions e.g. in adders, decoders, registers etc.

Large Scale Integration (LSI): 500 to a few thousand gates in a sigle package. They include digital systems like memory chips, processors e.t.c.

Very Large Scale Intrgration (VLSI): contain thousands of gates on a chip. E.g. large memory arrays and other complex microcomputer chips.
The most common technologies are the following:

1. **Transistor-Transistor Logic (TTL**):

   Power dissipation = 10 mW

   Switching time = 9ns

   It has been operational for many years and it is sometimes considered as a standard.

   It was an evolution of the previous technologies that used diodes and transistors. It was first called DTL (Diode Transistor Logic). The diodes were later replaced by transistors to improve on circuit operation.

   Several variations of the TTL include

   - High Speed TTL
   - Low Power TTL
   - Schottky TTL
   - Low Power Schottky TTL
   - Advanced Schottky TTL.

Emitter Coupled Logic: (ECL)

Power 25 mW

Switching Time 2ns

Used in systems requiring high speed operations.e.g. in super computers and signal processors where high speed is essential.

THE COMPUTER STRUCTURE

THE CPU

Factors that must be considered when learning about any CPU are

1. *Microprocessor Architecture*:The arrangement of registers in the CPU, number of bits in the address and data buses etc.
2. *Instruction Set*: Listing of operations the microprocessor can perform:
    - Transfering data
    - Arithmetic and logical operations
    - Data testing
    - Branching instructions
    - I/O operations
3. *Control Signals:* Outputs that direct other IC's e.g. ROMS and I/O ports when to operate.
4. *Pin Functions:* gives details about special inputs and outputs of the microprocessor.
5. *Minimal System:* how other devices are connected to the microprocessor.

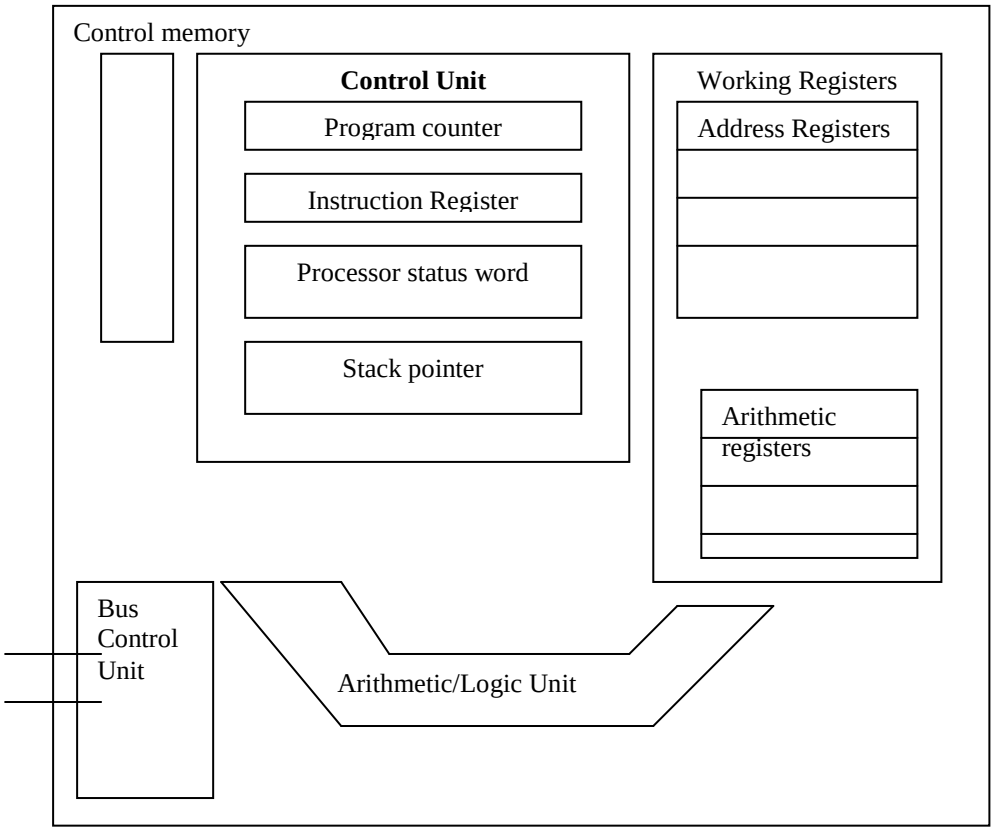
The main structural components of the CPU are

- The Control Unit
- The working Registers
- The Artihmetic and Control Logic.
- The CPU interconnections

The control unit contains the following registers:

1. *The Program Counter (PC)* :It holds the address of the main memory location from which the next instruction is to be fetched.

2. *Instruction Register (IR)* Receives the instruction when it is brought from memory and holds it while it gets decoded and executed.
3. *Processor Status Word (PSW)* contains condition flags which indicate the current status of the CPU and the important characteristics of the result of the previous instruction.
4. *Stack Pointer (SP):* Accesses a special part of memory called a stack. It is used to temporarily store important information while sub routines are being executed. It hold the address at the top of the stack.



Working Registers

They are Arithmetic registers or accumulators and address registers.

(i) ***Arithmetic Registers***: They temporarily hold the operands and the result of the arithmetic operations.

Accessing a register is faster than accessing memory. If several operations are tobe performed it is better to have the operands in registers than in memory and return only the result to memory. The more arithmetic registers a CPU has the faster it can execute computations.

(ii) ***Address Registers***: for addressing data and instructions in main memory.

If a register can be used for both arithmetic operations and addressing it is then called a general purpose register.

Arithmetic/Logic Unit

It performs arithmetic and logical operations on the contents of the working registers, the PC, memory locations etc. It also sets and clears the appropriate flags.

MEMORY

A byte: a group of 8 bits

A nibble: a group of 4 bits

A word: a group of 2,3, or 4 bytes depending on the computer and its system bus structure.

Each byte has an identifying address associated with it.

When a byte is to be accessed its address is transmitted to the appropriate interface via address lines.

Addresses are composed of bit combinations and the set of all bit combinations for a given situation is called an **address space**.

Some of the high order bits in a memory address are used to select the module and the remaining lower order bits identify the bytes or word within the module.

Similarly an interface is identified by the high order bits of an I/O address and the register within the interface is selected by the 2 or 3 low order bits.

The act of putting information into or taking information from a memory location is called *memory access*.

The number of bits in an address determine the size of an address space. If an address is n bits wide then there are 2^n possible addresses ($0 – 2^n–1$).

The number of address lines in the system bus dictates the size of memory or memory and the I/O space. A total of n address lines would imply a maximum memory (or overall memory and I/O) capacity of 2^n bytes.

16 address lines imply $2^{16} = 2^6 (2^{10}) = 64K$

Byte Ordering

Bytes in a word can be numbered from left to right or from right to left.

Big Endian: numbering from left to right.

Little Endian: numbering from right to left: This is the numbering adopted by Intel.

Classifications of memory

Memory can be classified as to whether it can retain its contents when power is turned off. In this class we have:

(i) **Volatile**: (MOS (Metal Oxide Semiconductor))

(ii) **Non Volatile**: (Magnetic Core).

It can also be classified according t o its Read/Write capabilities.

(a) **ROM**: (Read Only Memory): It cannot be written to and its contents cannot therefore be changed accidentally. Once its contents are set they can only be changed by special equipment.

There are mainly 4 types of ROM depending on the way in which their contents are set. Setting the contents of memory is called programming.

(i) **MASKED ROM**: contents are set by a masking operation performed while the chip is being manufactured. They cannot be altered by the user.

(ii) **PROM (Programmable ROM)**: contents can be set by the user using special equipment. Once programmed its contents can never be changed.

(iii) **EPROM(Erasable Programmable ROM)** Programmed by charge injection and once programmed the charge distribution is maintained until it is disturbed by some external energy source like Ultra Violet light.

It is primarily used during the development / testing stages and are replaced with ROMS or PROMS once the design is complete.

Eproms are non volatile but they can lose their contents with age. A device that cancels the contents of Eprom is called an *Eprom Eraser*.

(iv) **EAPROM(Electrically Alterable Programmable ROM)** Programmed and erased electrically instead of ultra violet light.

(b) **RAM**: (Random Access Memory) : It can be read and written to. It is called Random because all locations can be accessed with equal ease.

Ram is of two types:

(i) **Static Ram**: keep its contents so long as power is on.

(ii) **Dynamic Ram**: made of capacitors that can be charged or discharged. It must be refreshed often because of charge leakage.

I/O INTERFACES

All data transfers except that within the CPU itself is done over one or more buses. All I/O devices and main memory must somehow be connected to these buses. If there is more than one bus one bus is for memory and the other is for the other peripherals.

If there is only one bus (Single bus architecture) the same bus is used for both memory and I/O transfers. *Most micro computers have single buses.*

Memory and peripherals are connected to these buses through *interfaces and controllers*.

A controller is circuitry needed to initiate commands given to a device and to sense the status of the device.

An interface is circuitry needed to connect the peripheral and its control circuitry to the bus.

The interface must perform some combination of the following functions.

(i) Make the status of the peripheral available to the computer.

(ii) Provide buffer storage for input data.

(iii) Relay commands from the computer to the peripheral.

(iv) Provide buffer storage for output data.

(v) Signal to the CPU when the operation is complete.

(vi) Signal to the computer when an error occurs.

(vii) Pack bits into bytes or words for input and unpack them for output.

Data transfers between I/O or mass storage devices and the CPU or memory is categorised according to the amount of data transferred.

1. **Byte/Word Transfer**: one byte or word is mobed by one command. e.g. a terminal.
2. **Block Transfer:** A whole block of information is moved by a single command e.g. Direct memory Access transfers which are between memory and the peripheral.

In block transfers a device's interface must be used in conjunction with a DMA controller that can access memory directly without intervention by the CPU. e.g. a disk uses DMA.

Most devices that require high transfer rates are DMA devices.

When DMA capability is available it has higher priority over all other bus activity. Many interfaces are designed to perform both types of transfers.

SYSTEM BUS

Data Lines: transfer information. When communicating with memory the information is data or instructions. With I/O or mass storage devices information may be data, device status or commands.

The number of data lines determine the number of bits that can be transferred simultaneously; they have a direct bearing on speed.

The number of data lines are used to classify a microcomputer as 8 bit, 16 bit or 32 bit.

Address Lines: carry bit combinations that are decoded as addresses by the interfaces connected to the bus.

In a memory module each byte has its own address. The memory interface is designed to recognise all addresses of the bytes in its memory. No two bytes can have the same address.

The top few bits of an address select the module while the remaining bits find the particular memory location within the module.

If I/O devices share a bus with memory some addresses are reserved for their interface registers. Each register must have its own address and each interface must recognise the addresses of all its registers.

Once a memory or device interface recognises one of its addresses it either inputs the information on the data lines and transmits it to the proper place or it retrieves the necessary data and puts it on the data lines.

Control Lines: control signals must be passed back and forth among the CPU, the memory modules and the device interfaces. This information includes:

(i) **Request for Bus Usage**:made by the DMA controller.

(ii) **Grant for Bus usage**: Given by the CPU according to the pre-determined priority scheme.

(iii) **Interrupt signals:** External events require attention of the CPU.

(iv) **Timing Signals:** coordinating data and address transfers on the bus.

(v) **Parity signals:** Indicating data transfer errors.

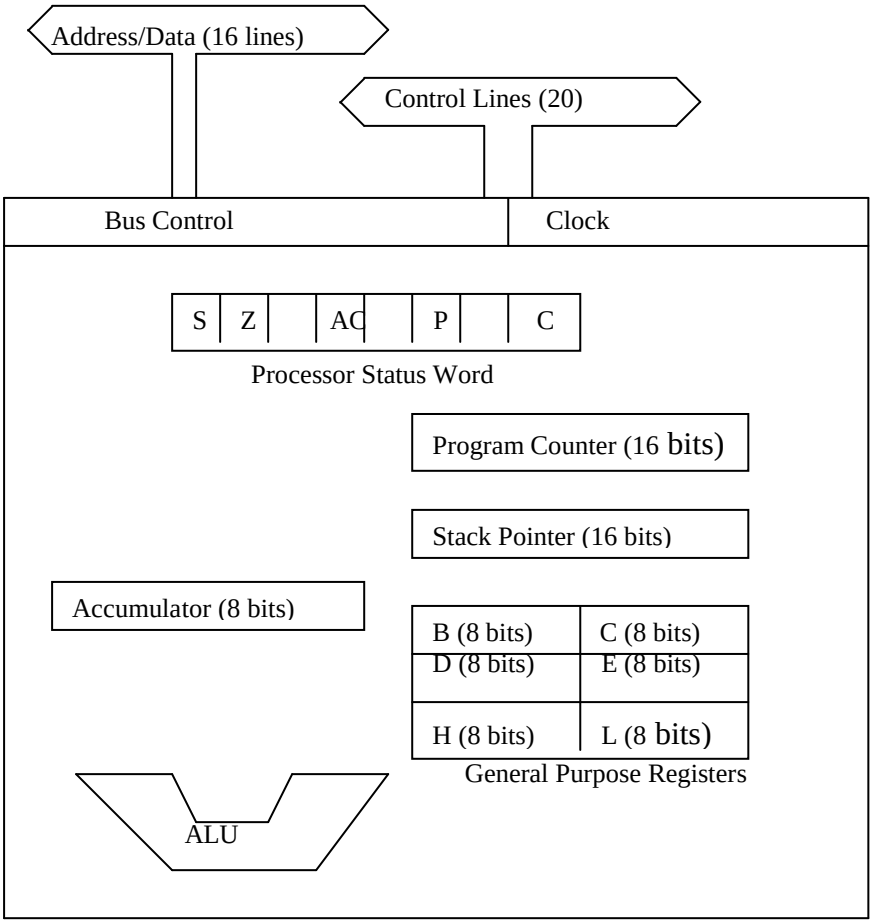
(vi) Signals for indicating malfunctions or power loss**.**

EXAMPLES OF CPU'S

The Intel 8085

It is an 8 bit processor i.e. has 8 data lines (1 byte of data can be transmitted at a time.

Has 6 general purpose registers namely B, C, D, E, H, L with 8 bits each and associated in pairs.



1 8 bit accumulator.

1 16 bit stack pointer

1 16 bit program counter

1 PSW with 5 flags.

Zero (Z): set when the result of the operation is zero.

Sign (S): set when sign of the result is negative.

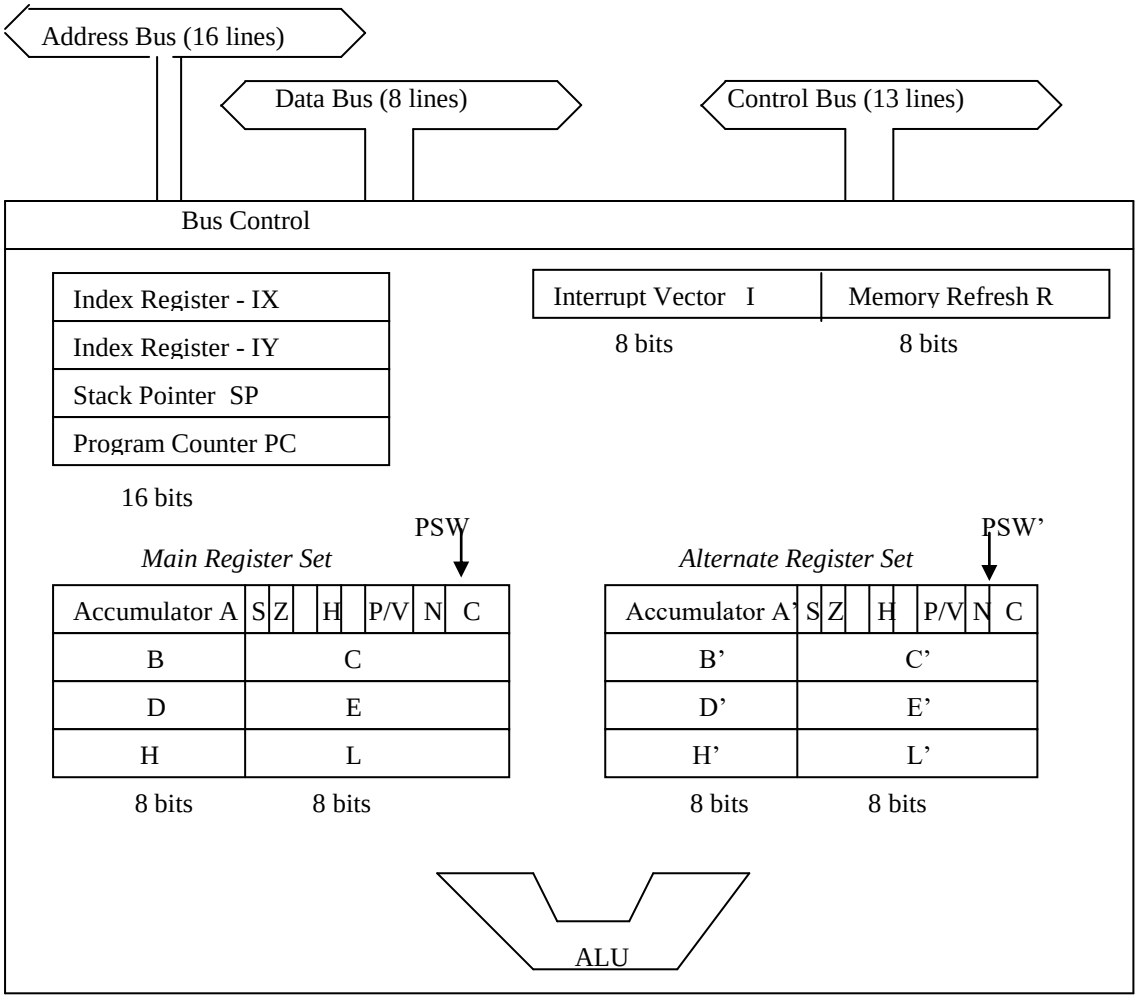
Parity (P): When the parity of the bits in the result is even.

Carry (C): Addition resulted into a carry or subtraction or comparison resulted into a borrow.

Auxiliary Carry (AC) Carry in BCD arithmetic.

The address and data share the same bus lines and they must take turns to use them. (They are time multiplexed). The address must be sent first and then data is sent or received.

The Zilog (Z80) Microprocessor



It is an 8 bit processor i.e. has 8 data lines (1 byte of data can be transmitted at a time.

2 16 bit index registers for base addressing

identical sets of registers each containing an 8 bit accumulator

A PSW with 6 flags and 6 general purpose registers.

2 8 bit special purpose registers

1 16 bit stack pointer

1 16 bit program counter

PSW flags are similar to Intel 8085 flags except

Parity (P)/Overflow : for input and logical operations i.e even parity like in 8085; for arithmetic operations it is an overflow.

N = 1 during subtraction

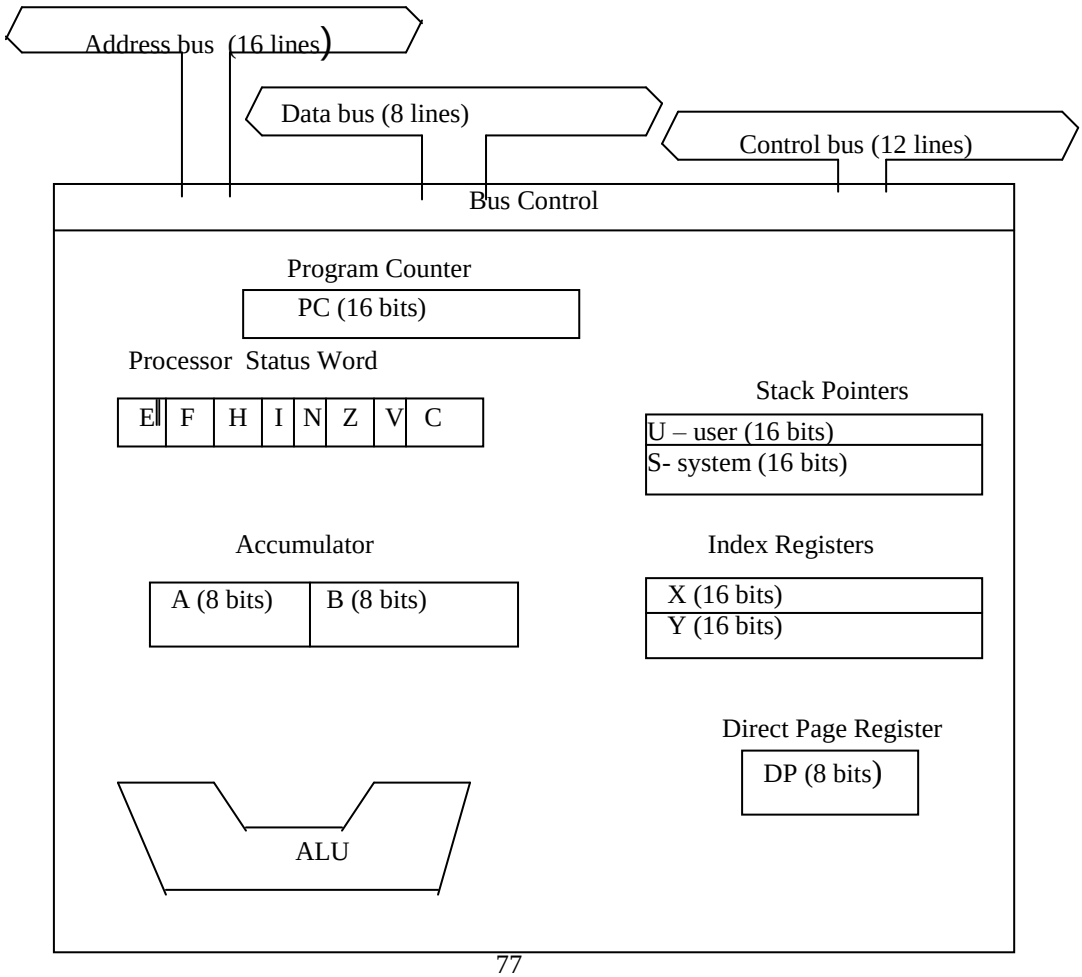
H = Half Carry: Like AC in Intel 8085

Address and data use separate lines.

Address space = $0 – 2^{16} – 1$

13 control signals.

The Motorola MC 6809



It is an 8 bit processor i.e. has 8 data lines (1 byte of data can be transmitted at a time.

2 16 bit index registers for base addressing

An 8 bit page register for addressing.

1 16 bit program counter

2 16 bit stack pointer, (one for a user stack and one for a system stack).

2 8 bit accumulators that can be used as a pair to form one 16 bit accumulator.

Z and C are the same as on the 8085.

N and H correspond to S and AC of the 8085 respectively.

V is overflow flag; set when a 2's complement signed arithmetic operation produces an overflow.

It is a 40 pin chip with 8 data pins 16 address pins, 12 control pins 2 oscillator pins a +5 V supply pin and a ground.

Addresses and data use separate lines

MACHINE LANGUAGE INSTRUCTIONS

At the time of execution all instructions are made up of a sequence of bytes (a combination of zeros and ones).

Because instructions in their 0's and 1's form can be directly understood by the computer they are therefore called **machine language instructions.**

All other forms of programs, assembler, high level etc must be reduced to their machine level form.

INSTRUCTION FORMATS

The arrangement of an instruction with respect to assigning meaning to its various groups of bits is called its format.

The portion of the instruction that specifies what the instruction does is called the **operation code** (opcode).

Any *address or piece of data* that is required by the instruction to complete its execution is called an **operand**

An instruction therefore consists of an operation code and a number of operands.

Most computers are designed so that not more than 2 operands are needed by a single instruction.

Some instructions require only one operand and they are called **single operand instructions**; others are double **operand instructions**.

If a quantity is taken from a location, it is called the *source operand.*

The location that is changed or where the source operand is taken is called the *destination*.

All instruction formats reserve the first bits of the instruction for at least part of the opcode but beyond this the formats vary considerably from

one computer to the next. The remaining bits designate the operands or their locations.

Instructions vary in length from 1 byte to 3 or 6 bytes.

The working registers for the Intel 8085 are A, B, C, D, E, H and L.

Register Addresses are:

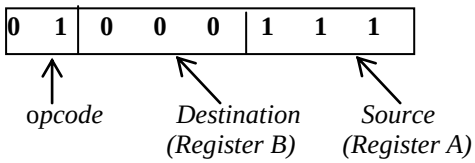
Register	Address	Register pair	
B	000	BC	00
C	001		
D	010	DE	01
E	011		
H	100	HL	10
L	101		
A	111		

The registers are sometimes considered in pairs of BC, DE and HL.

Both registers in the pair have the same higher 2 bits in their registers.

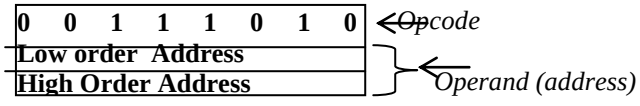
e.g.

Register to register transfer



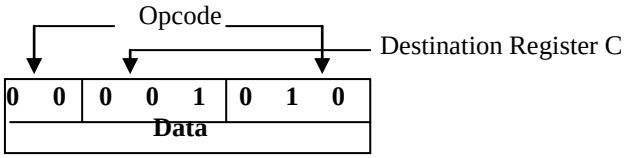
The instruction copies contents of register A (111) to register B (000); register A remains with its contents.

Load accumulator from memory



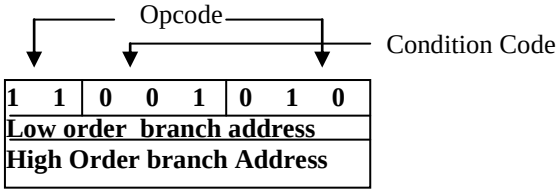
The instruction moves contents of a memory location whose address is specified in the two bytes to the accumulator

Transfer of immediate Data



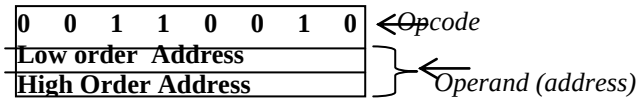
The instruction puts contents of the second byte (data) into register C (001)

Conditional branch to Zero Result



*The instruction makes the program to branch to the given address if the given condition (001, if zero) is satisfied***.**

Store Contents of Accumulator to Memory

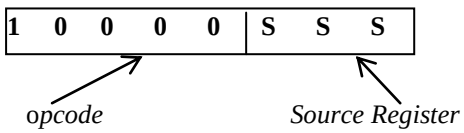


The instruction stores contents of the accumulator to a memory location whose address is specified in the given address.

Binary arithmetic

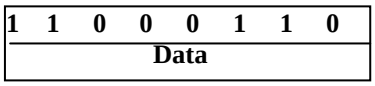
For all addition and subtraction instructions, the addend or minuend must be in register A and it is replaced by the result that is put in register A. The augend or subtrahend may come from any working register, a memory location or from the instruction.

Add Contents of a register to the Accumulator



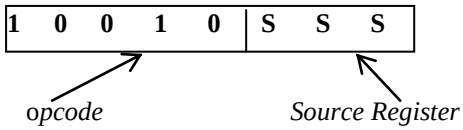
The instruction adds what is in a register described by the three bits SSS to contents of the accumulator; the answer stays in the accumulator.

Add Immediate Data to Register A



The instruction adds the number in the second byte (data) to what is in the accumulator. The answer remains in the accumulator.

Subtract contents of a register from the Accumulator



The instruction subtracts contents of the register whose address are the three bits SSS from the accumulator. The answer stays in the accumulator.

ADDRESSING MODES

They are the methods used to locate and fetch an operand from an internal CPU register or from a memory location.

Each processor has its own addressing modes.

1. *Immediate Addressing*: Information is part of the instruction. No addressing is needed to get the information.

   It is mostly used for quantities that are constants.

   They are 2 byte instructions where the operand is the second byte.
2. *Direct addressing*: The address is part of the instruction.
3. *Register addressing*: The operand is in the register and the register's address is part of the instruction.
4. *Indirect Addressing*: The address is in the location whose address is specified as part of the instruction. This location may be a register (register indirect addressing) or it may be a memory location.

   e.g., add contents of register R1 to the memory location whose address is in register R2.
5. *Base addressing*: The address is formed by adding the contents of a memory location or register to a number called a displacement which is part of the instruction. It is used primarily to reference arrays or in relocating a program in memory.
6. *Indexing*: It is a process of incrementing or decrementing an address as the computer sequences through a set of consecutive or evenly spaced addresses. This is done by successively changing an address that is stored in a register called an index register that can be incremented or decremented.
7. *Auto incrementing / decrementing*: The index is automatically incremented by an instruction.

Instruction Execution Time

The combination of actions taken during the execution of an instruction is called an **instruction cycle**. Instruction cycles are subdivided into **machine cycles** and the machine cycles are in turn subdivided into **states** (**clock cycles**). Each machine cycle is equivalent to a memory or I/O access. The retrieval of the first byte of the instruction from memory is called the **instruction fetch**. The first machine cycle of an instruction cycle is called a **fetch cycle**. The fetch cycle normally consists of 4 to 6 states depending on what actions are taken in addition to the instruction fetch. All the other machine cycles normally consist of three states. E.g. the instruction **LDA NUM** has 4 machine cycles which are:

(i) the fetch cycle

(ii) 2 cycles to read the address

(iii) 1 cycle to read the byte at NUM.

The fetch cycle has 4 states and each of the other three cycles has 3 states which makes a total of 13 states.

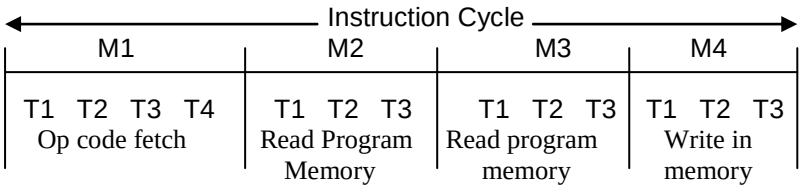
Accessing memory ordinarily takes 3 states:

(i) To send the address to memory

(ii) Memory looks up for the required information

(iii) Memory transmits the information back to the CPU.

A write operation is broken down similarly.



In each case one of the states is used by memory. If the memory cycle time is more than the clock cycle time memory then needs more than

one state to perform its function and a **wait state** is thereby introduced to enable the memory to complete its operation.

E.g. If a clock period is 300ns and the memory access time is 700ns the CPU will need 5 states to get 1 byte of information from memory i.e. the usual three states plus two wait states and the total required would be 1500ns.

The LXI instruction requires three memory accesses and the number of states required is 10. If the clock cycle time is 300ns and the memory access time is 700ns then it will take 10 + (3 memory accesses * 2 wait states) = 16 states and 4800ns to complete the LXI instruction.

Memory modules can be of different speeds. The instruction may be in a module that has a different speed from that of the operands. If the LHLD instruction is in the 200ns access time module and the operand is in the 700ns access time module the total number of states needed to execute the instruction assuming a 300 ns clock cycle would be

16 + (0 * 3) + (2 * 2) = 20 states and the total time needed would be 6000ns.

In general, if:

B Basic number of states in the table
W_m Wait states for instruction memory
B_i Bytes in instruction
W_o Wait states for operand memory
B_o Bytes in operand.

Then the total number of states is determined by:

$$\mathbf{B + (W_m * B_i) + (W_o * B_o)}$$

The instruction execution time is obtained by multiplying the total number of states with clock's period.

Summary of the 8085 instruction set execution times. *Two possible times (e.g. 7/10) indicates time depends on condition flag settings. First figure corresponds to no branch.*

Instruction	**Machine Cycles**	**States**		**Instruction**	**Machine Cycles**	**Sates**
MOV r1, r2	1	4		ANA r	1	4
MOV M, r	2	7		XRA r	1	4
MOV r, M	2	7		ORA r	1	4
MVI r	2	7		ANA M	2	7
MVI M	3	10		XRA M	2	7
LXI rp	3	10		ORA M	2	7
XCHG	1	4		ANI	2	7
STA	4	13		XRI	2	7
LDA	4	13		ORI	2	7
STAX B	2	7		CMA	1	4
STAX D	2	7		JMP	3	10
LDAX B	2	7		J(condition)	2/3	7/10
LDAX D		7		PCHL	1	6
SHLD	5	16		CALL	5	18
LHLD	5	16		C(Condition)	2/5	9/18
INX rp	1	6		RET	3	10
DCX rp	1	6		R (condition)	1/3	6/12
INR r	1	4		RLC	1	4
DCR	1	4		RRC	1	4
INR M	3	10		RAL	1	4
DCR M	3	10		RAR	1	4
ADD r	1	4		DAD rp	3	10
ADC r	1	4		CMC	1	4
SUB r	1	4		STC	1	4
SBB r	1	4		NOP	1	4
CMP r	1	4		HLT	1	5
ADD M	2	7		PUSH rp	3	12
ADC M	2	7		PUSH PSW	3	12
SUB M	2	7		POP rp	3	10
SBB M	2	7		POP PSW	3	10
CMP M	2	7		SPHL	1	6
ADI	2	7		XTHL	5	16
ACI	2	7		IN	3	10
SUI	2	7		OUT	3	10
SBI	2	7		RST	3	12
CPI	2	7		EI	1	4
DAA	1	4		DI	1	4
				RIM	1	4
				SIM	1	4

ASSEMBLER LANGUAGE

It is a type of language that is closer to machine language instructions.

There is an assembler language instruction for each machine language instruction.

An assembler converts Assembler Language into machine instructions.

There are 2 types of statements in assembler:

(i) Instructions: These are translated into machine code by the assembler.

(ii) Directive: Gives directions to the assembler during the assembly process but they are not translated into machine code.

Acronyms called **mnemonics** indicate the type of instruction.

Character strings called symbols or identifiers represent addresses and perhaps numbers.

A typical assembler instruction would be

MOV A , M

A typical assembler directive would be

COST : DS 1

This directive causes the assembler to reserve a byte and associate a symbol COST to it.

Example:

For a problem ANS: = X + Y; it can be solved as follows in the 8085 microprocessor.

```
LDA     X
MOV     B, A
LDA     Y
ADD     B
STA     ANS
```

3 points are worth noting.

(i) Most instructions involve movement of information from one part of the computer to another.

(ii) Computers do not work on entities with a flexible manner. e.g. for the ADD instruction, the second operand must be in the accumulator.

(iii) All programs whatever the language involve inputting, processing and outputting.

Assembler Instruction Format

The General format is

Label: Mnemonic Operand, Operand, ; remarks

Label: It is a symbol assigned to the address of the first byte of the instruction in which it appears.

Its presence is optional; if present it provides a symbolic name that can be used in branch instructions to branch to an instruction.

If there is no label then there is no colon.

All instruction must contain a **mnemonic**.

The presence of **operands** depends on the instruction. If there is more than one operand they are separated by a comma.

Remarks are for documenting the program; they are optional.

e.g. BRADDR: MOV A, M

Label *mnemonic* *destination Operand* *Source Operand*

Register – Register Transfer

e.g. NOW: MOV B, D

Label *mnemonic* *destination Operand* *Source Operand*

Load accumulator from memory

e.g. LDA NUM

Mnemonic *address of operand to be loaded into memory*

Transfer of immediate operand to a register

e.g. MVI E, 6

Mnemonic *Destination register* *Immediate Data*

Conditional branch on non-zero branch

e.g. JNZ HERE

Mnemonic *address*

Branch Condition

(i) All instructions are 1, 2, or 3 bytes long.

(ii) Instructions that involve only register or register indirect addressing are 1 byte long;

(iii) Those that involve I/O or immediate operands are 2 or 3 bytes long.

The working registers are A, B, C, D, E, H and L.

Register Addresses are:

Register	**Address**	**Register pair**	
B	000	BC	00
C	001		
D	010	DE	01
E	011		
H	100	HL	10
L	101		
A	111		

The registers are sometimes considered in pairs of BC, DE and HL.

Both registers in the pair have the same higher 2 bits in their registers.

Assembler Directives.

The directives direct the assembler during the assembly process.

The **ASM 85** has 3 directives.

They have the format

Label: Mnemonic Operand, Operand

The label is optional

The directives are DS, DB, and DW

DS (*Define Storage*)

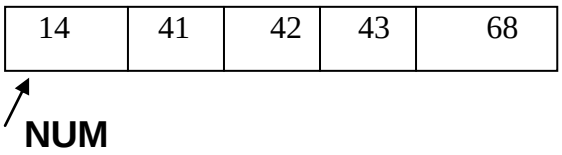
It is used to reserve memory and perhaps to assign a label to the first byte of the reserved area. e.g. ARRAY: DS 20 reserves 20 bytes and assigns the label ARRAY to the byte with the lowest address.

DB (*Define Byte*)

Used to put values into or pre-assign values to memory locations as well as reserve space and assign labels.

It serves as the DATA statement in Fortran. It can include up to 8 operands where each operand is a string constant with no more than 128 characters or constant expressions that evaluate to a 2's complement number from -128 to 127.

e.g. NUM: DB 14H, 'ABC',011101000B
reserves 5 bytes associated with a label NUM with the first byte.



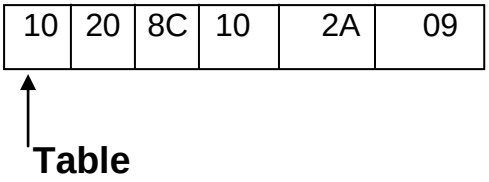
DW (*Define Word*)

Similar to DB except that it reserves words instead of bytes. Each of its possible 8 operands should evaluate to a 16 bit number or a single string of one or two characters.

The lower order byte of the word is stored in the lower byte address and the high order byte in the higher byte address e.g.

TABLE: DW TASK1, TASK2, 092AH

TASK1 and TASK2 are labels. Assuming that TASK1 and TASK2 have been assigned memory locations 2010 and 108C respectively



Transfer Instructions:

It is possible to move data between registers, between memory and registers or between memory locations.

The 8085 assembler has 13 transfer instructions where 4 of them can move an entire word.

LHLD aa **Load H & L from address aa** *(Addressing: Direct)*

0 0 1 0 1 0 1 0
Low Order Address
High order Address

The contents of a memory location whose address is specified in byte 2 and byte 3 of the instruction is moved to register L. The contents of the memory location at the succeeding address is moved to register H.

SHLD aa **Store H & L at memory location aa.** *(Addressing : Direct)*

0 0 1 0 0 0 1 0
Low Order Address
High order Address

The contents of register L is moved to a memory location whose address is specified in byte 2 and byte 3. The contents of register H is moved to the succeeding memory location.

LXI rp, data **Load register pair with immediate data**

0	0	**rp**	0	0	0	1	
Low Order Address							
High order Address							

Byte 3 of the instruction is moved into the high order register of the register pair rp. Byte 2 is moved into the lower order register of the pair.

XCHG **Exchange D & E with H & L**

1	1	1	0	1	0	1	1

The contents of registers H & L are exchanged with the contents of registers D & E.

The other nine instructions move one byte at a time.

MOV r1, r2 **Move Register** (*Addressing = Register*)

0	1	D	D	D	S	S	S

The contents of register r2 are moved to register r1.

MOV r, m **Move from memory** (*Addressing = indirect*)

0	1	D	D	D	1	1	0

The contents of a memory location whose address is in register pair HL is moved to register r.

MOV m, r **Move from register to memory** (*Addressing = indirect*)

0	1	1	1	0	S	S	S

The contents of register r are moved to the memory location whose address is in HL.

MVI r, data **Move Immediate** (*Addressing = immediate*)

0	0	D	D	D	1	1	0
Data							

The content of byte 2 of the instruction is moved to register r.

MVI M, data **Move to memory** (*Addressing = Immediate/register Indirect*)

0	0	1	1	0	1	1	0
Data							

The content of byte 2 of the instruction is moved to a memory location whose address is in registers H and L.

LDAX rp **Load Accumulator Indirect** (*Addressing =register indirect*)

0	0	rp		1	0	1	0

The contents of a memory location whose address is in the register pair rp is moved to register A. Only register pairs BC and DE can be used

STAX rp **Store Accumulator Indirect** *(Addressing =register indirect)*

0	0	rp		0	0	1	0

The contents of the accumulator are moved to a memory location whose address is in the register pair rp.

LDA aa **Load accumulator direct** *(Addressing : Direct)*

0	0	1	1	1	0	1	0
Low Order Address							
High order Address							

The contents of a memory location whose address is specified in byte 2 and byte 3 of the instruction is moved to register A.

STA aa **Store accumulator direct** *(Addressing : Direct)*

0	0	1	1	0	0	1	0
Low Order Address							
High order Address							

The contents of the accumulator are moved to the memory location whose address is specified in byte 2 and byte 3 of the instruction.

Assume three methods of putting a number 28_{10} = 1C in register B.

Using Direct and register addressing.

Address	Machine Code	Assembler Code
0210	3A	LDA X
0211	A0	
0212	02	
0213	47	MOV B,A
02A0	1C	X: DB 1CH

Using Indirect addressing.

Address	Machine Code	Assembler Code
0210	21	LXI H, X
0211	A0	
0212	02	
0213	46	MOV B,M

.

02A0	1C	X: DB 1CH

Using Immediate addressing.

Address	Machine Code	Assembler Code
0210	06	MVI B, 1CH
0211	1C	

Two ways of moving a word from one memory location to another memory location.

Address	Machine Code	Assembler Code
0120	2A	LHLD NUMS
0121	E2	
0122	01	
0123	22	SHLD NUMD
0124	F6	
0125	01	
01E2		NUMS DS 2
01E3		
01F6		NUMD DS 2
01F7		

Address	Machine Code	Assembler Code
0120	3A	LDA NUMS
0121	E2	
0122	01	
0123	32	STA NUMD
0124	F6	
0125	01	
0126	3A	LDA NUMS + 1
0127	E3	
0128	01	
0129	32	STA NUMD + 1
012A	F7	
012B	01	
01E20		NUMS DS 2
01E3		

01F6		NUMD DS 2
01F7		

Incrementing and Decrementing Instructions

These are needed in high level languages for loops and indexing through arrays.

In Assembler we have instructions for adding 1 to the quantity in any register, adding 1 to the memory byte pointed to by the HL pair and adding 1 to the 16 bit quantity in a register pair.

There are also corresponding instructions for subtracting 1 from these quantities.

INR r **Increment Register** *(Addressing: Register)*

0	0	D	D	D	1	0	0

The contents of register r. is incremented by 1

INR m **Increment memory** *(Addressing: Register Indirect)*

0	0	1	1	0	1	0	0

The contents of a memory location whose address is contained in the HL register pair is incremented by 1

INX rp **Increment Register pair***(Addressing: Register***)**

0	0	r	p	0	0	1	1

The contents of register pair rp is incremented by 1

Similarly

DCR r **decrement Register** *(Addressing: Register***)**

0	0	D	D	D	1	0	1

The contents of register r is decremented by 1

DCR m decrement memory *(Addressing: Register Indirect***)**

0	0	1	1	0	1	0	1

The contents of a memory location whose address is contained in the HL register pair is decremented by 1

DCX rp **decrement Register pair***(Addressing: Register***)**

0	0	rp		1	0	1	1

The contents of register pair rp are decremented by 1
e.g. To move 3 consecutive bytes beginning at SRC to 3 consecutive bytes beginning at DEST

Address	**Machine Code**	**Assembler Code**
0050	21	LXI H, SRC
0051	80	
0052	00	
0053	46	MOV B,M
0054	23	INX H
0055	4E	MOV C,M
0056	23	INX H
0057	56	MOV D,M
0058	21	LXI H, DES
0059	20	
005A	01	
005B	70	MOV M,B
005C	23	INX H
005D	71	MOV M,C

005E	23	INX H
005F	72	MOV M,D

0080		SRC: DS 3
0081		
0082		

0120		DES: DS 3
0121		
0122		

Binary Arithmetic

For all addition and subtraction instructions, the addend or minuend must be in register A and it is replaced by the result that is put in register A.

The augend or subtrahend may come from any working register, memory location whose address is in the register pair HL or from the instruction.

ADD r **Add Register** *(Addressing: Register*)

1	0	0	0	0	S	S	S

The contents of register r is added to the content of the accumulator; the result is put in the accumulator.

ADD m **Add memory** (*Addressing: Register Indirect*)

1	0	0	0	0	1	1	0

The contents of a memory location whose address is contained in the HL register pair is added to the contents of the accumulator.

ADI data **Add Immediate data** (*Addressing: Immediate)*

1	1	0	0	0	1	1	0
Data							

The contents of the second byte of the instruction are added to the contents of the accumulator.

Similarly we have

1. SUB r

1	0	0	1	0	S	S	S

2. SUB M

1	0	0	1	0	1	1	0

3. SUI data

1	1	0	1	0	1	1	0
Data							

E.g. ANS = TOTL – (Num1 + Num2) + 6

(Assume that num1 and num2 are consecutive memory locations.

```
LXI     H, Num1
MOV     A, M
INX     H
ADD     M
MOV     B, A
LDA     TOTL
SUB     B
ADI     6
STA     ANS

ANS     DS 1
NUM     DS 1
TOTL    DS 1
```

This code is limited to 8 bits i.e. 0 – 255 or –128 + 127

This is called single precision arithmetic.

For larger numbers which are 2 or more bytes, arithmetic operations that accommodate carries and borrows are used.

DAD rp (Add register pair to H and L)

0	0	r	p	1	0	0	1

The content of the register pair rp is added to the content of the register pair H and L. The result is placed in the register pair H and L.

Branching

There are conditional and unconditional branches.

(i) JMP addr (Jump)

Control is transferred to the instruction whose address is specified.

1 1 0 0 0 0 1 1
Low Order
High Order

(ii) Jcond addr (conditional jump)

If the specified condition is true control is transferred to the instruction whose address is specified.

1 1 C C C 0 1 0
Low Order
High Order

Conditions			CCC
NZ	Not Zero	(z = 0)	000
Z	Zero	(z = 1)	001
NC	No Carry	(c = 0)	010
C	Carry	(c = 1)	011
PO	Parity Odd	(p =0)	100
PE	Parity even	(p =1)	101
P	Plus	(s = 0)	110
M	Minus	(s=1)	111

Conditional branches are often used in conjunction with the **compare** instructions.

CMP r (Compare register)

The content of register r is subtracted from the contents of the accumulator. The accumulator remains unchanged.

CMP M (Compare memory)

Contents of a memory location whose address is in the H and L register pair is subtracted from the accumulator.

CPI data (Compare immediate)

The contents of the 2nd byte of the instruction is subtracted from the accumulator.

e.g.

```
If   X > Y
        X = X + 1
Else
        X = X – 1
End if
Ans = X
```

```
        LDA     Y
        MOV     B, A
        LDA     X
        CMP     B
        JP      GREATER
        DCR     A
        JMP     OUT
Greater INR     A
OUT     STA     ANS
```

```
If (nn >= 0 AND   nn >= mm)
        GO TO EXIT
ENDIF
```

```
        LDA     NN
        CPI     0
        JM      CONT
        LXI     H, MM
        CMP     M
        JP      EXIT

CONT
EXIT
```

e.g.

```
If   NUM < 0
        NEG = NEG + 1
Else
        IF NUM = 0 THEN
                ZER0 = ZERO + 1
        ELSE
                POS = POS + 1
ENDIF
```

```
                LDA     NUM
                CPI     0
                JP      NOT NEGAT
                LXI     H, NEG
                INR     M
                JUMP    OUT
NOT NEGAT       JNZ     POSIT
                LXI     H, ZER
                INR M
                JMP     OUT
POSIT           LXI     H, POS
                INR     M

OUT

                NUM     DS      1
                NEG     DB      0
                ZER     DB      0
                POS     DB      0
```

Looping

The same instructions are executed several times with only minor adjustments. This involves executing a set of code until a counting process fails a test or passes it.

For i = 1 to 10
x = x + I
End for
Sum = x

```
                MVI     D, 0
                LXI     H, X
                MOV     C, M
        LOOP    MOV     A, C
                INR     D
                ADD     D
                MOV     C, A
                MOV     A, D
                CPI     10
                JM      LOOP
                MOV     A, C
                STA     SUM
```

50 bytes of data are to be moved from locations beginning at 0200 to locations at 0400

```
                MVI     L, 50
                LXI     D, SRC
                LXI     B, DES
        LOOP    LDAX    D
                STAX    B
                INX     D
```

```
                    INX         B
                    DCR         L
                    JNZ         LOOP

0200    SRC:        DS      50
0400    DES:        DS      50
```