Logics Circuit

Jagdish Lekhak,
Senior Engineer
Nepal Telecom Regional Directorate

Unit 1

Binary Systems

Introduction

We are in "Information age" since digital systems have such a prominent and growing role in modern society. They are involved in our business transactions, communications, transportation, medical treatment and entertainment. In industrial world they are heavily employed in design, manufacturing, distribution and sales.

Analog System

Analog systems process analog signals (continuous time signals) which can take any value within a range, for example the output from a speaker or a microphone.

An analog meter can display any value within the range available on its scale. However, the precision of readings is limited by our ability to read them. E.g. meter on the right shows 1.25V because the pointer is estimated to be half way between 1.2 and 1.3. The analogue meter can show any value between 1.2 and 1.3 but we are unable to read the scale more precisely than about half a division.

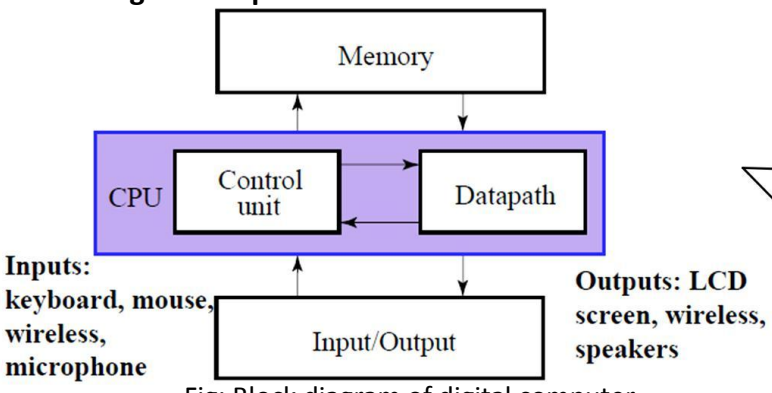
Digital System

- Digital systems process digital signals which can take only a limited number of values (discrete steps), usually just two values are used: the positive supply voltage (+Vs) and zero volts (0V).
- Digital systems contain devices such as logic gates, flip-flops, shift registers and counters.

A digital meter can display many values, but not every value within its range. For example the display on the right can show 6.25 and 6.26 but not a value between them. This is not a problem because digital meters normally have sufficient digits to show values more precisely than it is possible to read an analogue display.

The general purpose digital computer is a best known example of **digital system**.

Generic Digital computer structure



- Actually processor contains 4 functional modules: CPU, FPU, MMU and internal cache.
- Here only CPU is specified.

Fig: Block diagram of digital computer

Working principle of generic digital computer: Memory stores programs as well as input, output and intermediate data. The datapath performs arithmetic and other data-processing operations as specified by the program. The control unit supervises the flow of information between the various units. A datapath, when combined with the control unit, forms a component referred to as a *central processing unit,* or CPU. The program and data prepared by the user are transferred into memory by means of an input device such as a keyboard. An output device, such as a CRT (cathode-ray tube) monitor, displays the results of the computations and presents them to the user.

Advantages of digital system:

- Have made possible many scientific, industrial, and commercial advances that would have been unattainable otherwise.
- Less expensive
- More reliable
- Easy to manipulate
- Flexibility and Compatibility
- Information storage can be easier in digital computer systems than in analog ones. New features can often be added to a digital system more easily too.

Disadvantages of digital system:

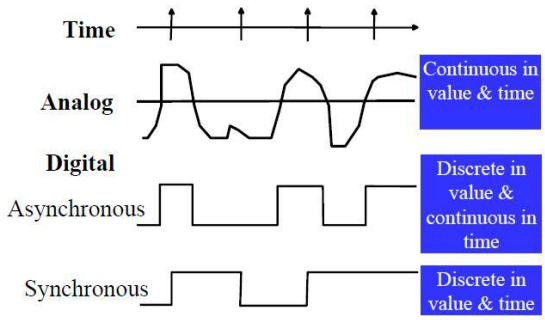
- Use more energy than analog circuits to accomplish the same tasks, thus producing more heat as well.
- Digital circuits are often fragile, in that if a single piece of digital data is lost or misinterpreted, the meaning of large blocks of related data can completely change.
- Digital computer manipulates discrete elements of information by means of a binary code.
- Quantization error during analog signal sampling.

Information Representation

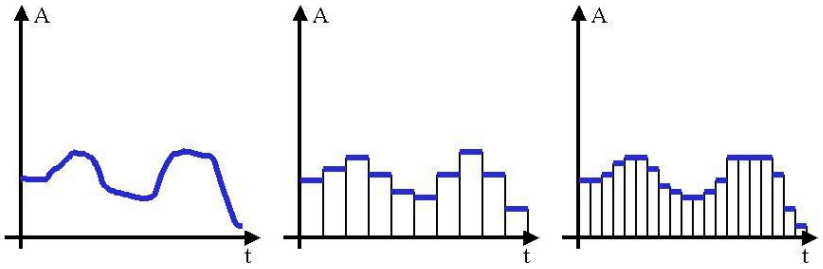
Signals

- Information variables represented by physical quantities.
- For digital systems, the variables take on discrete values.
- Two level or binary values are the most prevalent values in digital systems.
- Binary values are represented abstractly by:
  - digits 0 and 1
  - words (symbols) False (F) and True (T)
  - words (symbols) Low (L) and High (H)
  - and words On and Off.
- Binary values are represented by values or ranges of values of physical quantities

Signal Examples over time

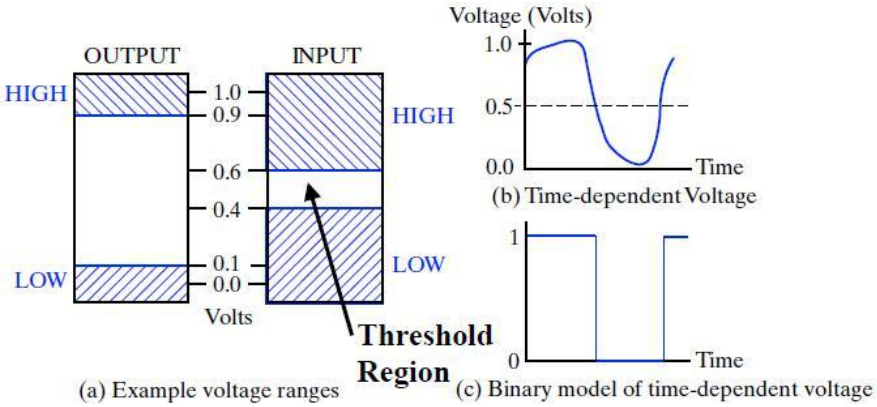


Here is an example waveform of a quantized signal. Notice how the magnitude of the wave can only take certain values, and that creates a step-like appearance. This image is discrete in magnitude, but is continuous in time (asynchronous):



Analog signal – continuously varying | Digital signal – large time divisions | Digital signal – small time divisions

Signal Example – physical quantity: Voltage

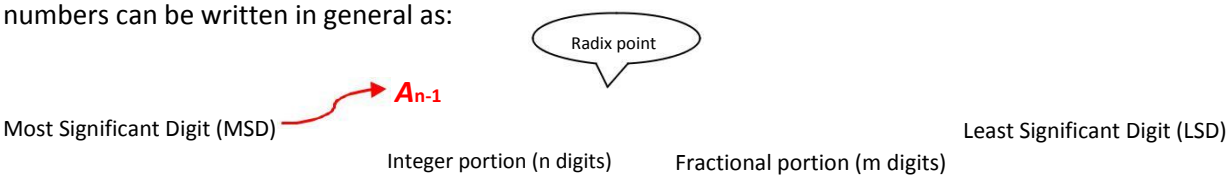


What are other physical quantities representing 0 and 1?

1. CPU: Voltage
- Disk: Magnetic Field Direction
2. CD: Surface Pits/Light
- Dynamic RAM: Electrical Charge

Number Systems

Here we discuss positional number systems with Positive radix (or base) ***r***. A number with radix ***r*** is represented by a string of digits as below i.e. wherever you guys see numbers of whatever bases, all numbers can be written in general as:



in which $\mathbf{0} \le A_i < r$ (since each being a symbol for particular base system viz. for r = 10 (decimal number system) A_i will be one of 0,1,2,…,8,9). Subscript i gives the position of the coefficient and, hence, the weight r^i by which the coefficient must be multiplied.

HEY! Confused? Don't worry! I will describe specific number systems (r=2, 8, 10 and 16) used in digital computers later one by one, then the concept will be quite clear.

In general, a number in base *r* contains *r* digits, 0, l, 2... *r*- 1, and is expressed as a power series in *r* with the general form:

$$(\text{Number})_r = A_{n-1}\, r^{n-1} + A_{n-2}\, r^{n-2} + \ldots + A_1\, r^1 + A_0\, r^0 + A_{-1}\, r^{-1} + A_{-2}\, r^{-2} + \ldots + A_{-m+1}\, r^{-m+1} + A_{-m}\, r^{-m}$$

$$(\text{Number})_r = \left(\sum_{i=0}^{i=n-1} A_i \cdot r^i\right) + \left(\sum_{j=-m}^{j=-1} A_j \cdot r^j\right)$$

$$(\text{Integer Portion}) + (\text{Fraction Portion})$$

Decimal Number System (Base-10 system)

Radix (r) = 10

Symbols = 0 through r-1 = 0 through 10-1 = {0, 1, 2... 8, 9}

I am starting from base-10 system since it is used vastly in everyday arithmetic besides computers to represent numbers by strings of digits or symbols defined above, possibly with a *decimal point*. Depending on its position in the string, each digit has an associated value of an integer raised to the power of 10.

Example: decimal number 724.5 is interpreted to represent 7 hundreds plus 2 tens plus 4 units plus *5* tenths.

$$724.5 = 7 \text{ X } 10^2 + 2 \text{ X } 10^1 + 4 \text{ X } 10^0 + 5 \text{ X } 10^{-1}$$

Binary Number System (Base-2 system)

Radix (r) = 2

Symbols = 0 through r-1 = 0 through 2-1 = {0, 1}

A binary numbers are expressed with a string of 1'sand 0's and, possibly, a *binary point* within it. The decimal equivalent of a binary number can be found by expanding the number into a power series with a base of 2.

Example: $(11010.01)_2$ can be interpreted using power series as:

$(11010.01)_2 = 1 \text{ X } 2^4 + 1 \text{ X } 2^3 + 0 \text{ X } 2^2 + 1 \text{ X } 2^1 + 0 \text{ X } 2^0 + 0 \text{ X } 2^{-1} + 1 \text{ X } 2^{-2} = (26.25)_{10}$

Digits in a binary number are called bits (**B**inary dig**IT**s). When a bit is equal to 0, it does not contribute to the sum during the conversion. Therefore, the conversion to decimal can be obtained by adding the numbers with powers of 2 corresponding to the bits that are equal to 1. Looking at above example,

$(11010.01)_2 = 16 + 8 + 2 + 0.25 = (26.25)_{10}$.

In computer work,

- 2^{10} is referred to as K (kilo),
- 2^{20} as M (mega),
- 2^{30} as G (giga),
- 2^{40} as T (tera) and so on.

n	2^n	n	2^n	n	2^n
0	1	8	256	16	65,536
1	2	9	512	17	131,072
2	4	10	1,024	18	262,144
3	8	11	2,048	19	524,288
4	16	12	4,096	20	1,048,576
5	32	13	8,192	21	2,097,152
6	64	14	16,384	22	4,194,304
7	128	15	32,768	23	8,388,608

Table: Numbers obtained from 2 to the power of n

Octal Number System (Base-8 system)

Radix (r) = 8

Symbols = 0 through r-1 = 0 through 8-1 = {0, 1, 2…6, 7}

An octal numbers are expressed with a strings of symbols defined above, possibly, an *octal point* within it. The decimal equivalent of a octal number can be found by expanding the number into a power series with a base of 8.
Example: $(40712.56)_8$ can be interpreted using power series as:
$(40712.56)_8 = 4 \text{ X } 8^4 + 0 \text{ X } 8^3 + 7 \text{ X } 8^2 + 1 \text{ X } 8^1 + 2 \text{ X } 8^0 + 5 \text{ X } 8^{-1} + 6 \text{ X } 8^{-2} = (16842.1)_{10}$

Hexadecimal Number System (Base-16 system)
Radix (r) = 16
Symbols = 0 through r-1 = 0 through 16-1 = {0, 1, 2…9, A, B, C, D, E, F}

A hexadecimal numbers are expressed with a strings of symbols defined above, possibly, a *hexadecimal point* with in it. The decimal equivalent of a hexadecimal number can be found by expanding the number into a power series with a base of 16.
Example: $(4D71B.C6)_{16}$ can be interpreted using power series as:

$$(4D71B.C6)_{16} = 4 \text{ X } 16^4 + D \text{ X } 16^3 + 7 \text{ X } 16^2 + 1 \text{ X } 16^1 + B \text{ X } 16^0 + C \text{ X } 16^{-1} + 6 \text{ X } 16^{-2}$$
$$= 4 \text{ X } 16^4 + 13 \text{ X } 16^3 + 7 \text{ X } 16^2 + 1 \text{ X } 16^1 + 11 \text{ X } 16^0 + 12 \text{ X } 16^{-1} + 6 \text{ X } 16^{-2}$$
$$= (317211.7734375)_{10}$$

Number Base Conversions

Case I: *Base-r system to Decimal*: Base-r system can be binary (r=2), octal (r=8), hexadecimal (r=16), base-60 system or any other. For decimal system as destination of conversion, we just use power series explained above with varying *r* and sum the result according to the arithmetic rules of base-10 system. I have already done examples for binary to decimal, octal to decimal and hexadecimal to decimal.

For refreshment lets assume base-1000 number $(458HQY)_{1000}$. Where n = 6 and m = 0.
$(458HQY)_{1000} = A_{n-1} r^{n-1} + A_{n-2} r^{n-2} + \ldots + A_1 r^1 + A_0 r^0 + A_{-1} r^{-1} + A_{-2} r^{-2} + \ldots + A_{-m+1} r^{-m+1} + A_{-m} r^{-m}$
$= 4 \text{ X } 1000^5 + 5 \text{ X } 1000^4 + 8 \text{ X } 1000^3 + H \text{ X } 1000^2 + Q \text{ X } 1000^1 + Y \text{ X } 1000^0$
=Resulting number will be in decimal. Here I have supposed various symbols for base-1000 system. Don't worry, if someone gives you base-1000 number for conversion, he should also define all 1000 symbols (0-999).

Case II: *Decimal to Base-r system*: Conversion follows following algorithm.

1. Separate the number into **integer** and **fraction** parts if radix point is given.
2. Divide "*Decimal Integer part*" by base *r* repeatedly until quotient becomes zero and storing remainders at each step.
3. Multiply "*Decimal Fraction part*" successively by r and accumulate the integer digits so obtained.
4. Combine both accumulated results and parenthesize the whole result with subscript *r*.

Example I: Decimal to binary

- $(41.6875)_{10} = (?)_2$
  Here Integer part = 41 and fractional part = 0.6875

Integer = 41		Fraction = 0.6875
41		0.6875
20	1	2
10	0	1.3750
5	0	x 2
2	1	0.7500
1	0	x 2
0	1	1.5000
		x 2
		1.0000

$(41)_{10} = (101001)_2$ $(0.6875)_{10} = (0.1011)_2$

$(41.6875)_{10} = (101001.1011)_2$

Example II: Decimal to octal

- $(153.45)_{10} = (?)_8$
  Here integer part = 153 and fractional part = 0.45

153	
19	1
2	3
0	2

This is simply division by 8, I am writing Quotients and remainders only.

$(153)_{10} = (231)_8$

0.45
X 8

3.60
X 8

4.80
X 8

6.40 (may not end, choice is upon you to end up)

Multiply always the portion after radix point.

$(0.45)_{10} = (346)_8$

$(153.45)_{10} = (231.346)_8$

Example III: Decimal to Hexadecimal

- $(1459.43)_{10} = (?)_{16}$
  Here integer part = 1459 and fractional part = 0.43

1459	
91	4
5	11 (=B)
0	5

$(1459)_{10} = (5B4)_{16}$

0.43
X16

6.80
X16

12.80
X16

12.80 (Never ending…)

$(0.43)_{10} = (6CC)_8$

$(1459.43)_{10} = (5B4.6CC)_{16}$

Case III: *Binary to octal & hexadecimal and vice-versa*: Conversion from and to binary, octal and hexadecimal representation plays an important part in digital computers. Since,

- 2^3 = 8, octal digit can be represented by at least 3 binary digits. (We have discussed this much better in class). So to convert given binary number into its equivalent octal, we divide it into groups of 3 bits, give each group an octal symbol and combine the result.
    - Integer part: Group bits from right to left of an octal point. 0's can be added to make it multiple of 3 (**not compulsory**).
    - Fractional part: Group bits from left to right of an octal point. 0's must be added to if bits are not multiple of 3 (**Note it**).
- 24 = 16, each hex digit corresponds to 4 bits. So to convert given binary number into its equivalent hex, we divide it into groups of 4 bits, give each group a hex digit and combine the result. If hex point is given, then process is similar as of octal.
- 15 numbers in 4 systems summarized below for easy reference.

Decimal (base 10)	Binary (base 2)	Octal (base 8)	Hexadecimal (base 16)
00	0000	00	0
01	0001	01	1
02	0010	02	2
03	0011	03	3
04	0100	04	4
05	0101	05	5
06	0110	06	6
07	0111	07	7
08	1000	10	8
09	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Example:

1. Binary to octal:
   $(10110001101011.11110000011)_2 = (010$ [redacted] $)_2 = (26153.7406)_8$
   2 6 1 5 3 7 4 0 6
2. Binary to hexadecimal:
   $(10110001101011.11110000011)_2 = (0010\ 1100\ 0110\ 1011.\ 1111\ 0000\ 0110)_2 = (2C6B.F06)_{16}$
   2 C 6 B F 0 6
3. From hex & octal to binary is quite easy, we just need to remember the binary of particular hex or octal digit.
   $(673.12)_8$ = 110 111 011. 001 010 = $(110111011.00101)_2$
   $(3A6.C)_{16}$ = 0011 1010 0110. 1100 = $(1110100110.11)_2$

Complements

Complements are used in digital computers for simplifying the subtraction operation and for logical manipulation. There are two types of complements for each base-r system: r's complement and the second as the (r - 1)'s complement. When the value of the base *r* is substituted, the two types are referred to as the 2's complement and 1's complement for binary numbers, the 10's complement and 9's complement for decimal numbers etc.

(r-1)'s Complement (diminished radix compl.)

(r-1)'s complement of a number N is defined as $(r^n - 1) - N$

Where **N** is the given number

r is the base of number system

n is the number of digits in the given number

To get the (r-1)'s complement fast, subtract each digit of a number from (r-1).

Example:

- 9's complement of 835_{10} is 164_{10} (Rule: $(10^n - 1) - N$)
- 1's complement of 1010_2 is 0101_2 (bit by bit complement operation)

r's Complement (radix complement)

r's complement of a number N is defined as $r^n - N$ Where **N** is the given number

r is the base of number system

n is the number of digits in the given number

To get the r's complement fast, add 1 to the low-order digit of its (r-1)'s complement.

Example:

- 10's complement of 835_{10} is $164_{10} + 1 = 165_{10}$
- 2's complement of 10102 is $0101_2 + 1 = 0110_2$

Subtraction with complements

The direct method of subtraction taught in elementary schools uses the borrow concept. When subtraction is implemented with digital hardware, this method is found to be less efficient than the method that uses complements.

The subtraction of two n-digit unsigned numbers *M - N* in base-*r* can be done as follows:

1. Add the minuend *M* to the *r's* complement of the subtrahend *N.* This performs $M + (r^n - N) = M - N + r''$.
2. If *M >= N,* the sum will produce an end carry, *r'',* which is discarded; what is left is the result *M - N.*
3. If *M < N,* the sum does not produce an end carry and is equal to $r^n - (N - M)$, which is the *r's* complement of *(N - M).* To obtain the answer in a familiar form, take the r's complement of the sum and place a negative sign in front.

Example I:

Using 10's complement, subtract 72532 − 3250.

M =	72532
10's complement of N =	+ 96750
Sum =	169282
Discard end carry 10^5 =	−100000
Answer =	69282

HEY! *M* has 5 digits and *N* has only 4 digits. Both numbers must have the same number of digits; so we can write *N* as 03250. Taking the 10's complement of *N* produces a 9 in the most significant position. The occurrence of the end carry signifies that *M >= N* and the result is positive.

Example II:

Using 10's complement, subtract 3250 − 72532.

$$\begin{aligned} M &= \quad 03250 \\ \text{10's complement of } N &= +\ 27468 \\ \text{Sum} &= \quad 30718 \end{aligned}$$

There is no end carry.

Answer: −(10's complement of 30718) = −69282

Example III:

Given the two binary numbers $X = 1010100$ and $Y = 1000011$, perform the subtraction (a) $X - Y$ and (b) $Y - X$ using 2's complements.

(a)

$$\begin{aligned} X &= \quad 1010100 \\ \text{2's complement of } Y &= +\ 0111101 \\ \text{Sum} &= \quad 10010001 \\ \text{Discard end carry } 2^7 &= -10000000 \\ \text{Answer: } X - Y &= \quad 0010001 \end{aligned}$$

(b)

$$\begin{aligned} Y &= \quad 1000011 \\ \text{2's complement of } X &= +\ 0101100 \\ \text{Sum} &= \quad 1101111 \end{aligned}$$

There is no end carry.

Answer: $Y - X$ = −(2's complement of 1101111) = −0010001 ■

Example IV: Repeating Example III using 1's complement.

(a) $X - Y = 1010100 - 1000011$

$$\begin{aligned} X &= \quad 1010100 \\ \text{1's complement of } Y &= +\ 0111100 \\ \text{Sum} &= \quad 10010000 \\ \text{End-around carry} &\longrightarrow +\ 1 \\ \text{Answer: } X - Y &= \quad 0010001 \end{aligned}$$

(b) $Y - X = 1000011 - 1010100$

$$\begin{aligned} Y &= \quad 1000011 \\ \text{1's complement of } X &= +\ 0101011 \\ \text{Sum} &= \quad 1101110 \end{aligned}$$

There is no end carry.

Answer: $Y - X$ = −(1's complement of 1101110) = −0010001

Binary Codes

Electronic digital systems use signals that have two distinct values and circuit elements that have two stable states. There is a direct analogy among binary signals, binary circuit elements, and binary digits. A binary number of *n* digits, for example, may be represented by *n* binary circuit elements, each having an output signal equivalent to a 0 or a 1. Digital systems represent and manipulate not only binary numbers, but also many other discrete elements of information. Any discrete element of information distinct among a group of quantities can be represented by a binary code. Binary codes play an important role in digital computers. The codes must be in binary because computers can only hold 1's and 0's.

1. Binary Coded Decimal (BCD)

The binary number system is the most natural system for a computer, but people are accustomed to the decimal system. So, to resolve this difference, computer uses decimals in coded form which the hardware understands. A binary code that distinguishes among 10 elements of decimal digits must contain at least four bits. Numerous different binary codes can be obtained by arranging four bits into 10 distinct combinations. The code most commonly used for the decimal digits is the straightforward binary assignment listed in the table below. This is called ***binary-coded decimal*** and is commonly referred to as **BCD**.

Decimal Symbol	BCD Digit
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Table: 4-bit BCD code for decimal digits

- A number with *n* decimal digits will require 4n bits in BCD. E.g. decimal 396 is represented in BCD with 12 bits as 0011 1001 0110.
- Numbers greater than 9 has a representation different from its equivalent binary number, even though both contain 1's and 0's.
- Binary combinations 1010 through 1111 are not used and have no meaning in the BCD code.
- Example :

$(185)_{10} = (0001\ 1000\ 0101)_{BCD} = (10111001)_2$

2. Error-Detection codes

Binary information can be transmitted from one location to another by electric wires or other communication medium. Any external noise introduced into the physical communication medium may change some of the bits from 0 to 1 or vice versa.

The purpose of an error-detection code is to detect such bit-reversal errors. One of the most common ways to achieve error detection is by means of a **parity bit**. A *parity bit* is the extra bit included to make the total number of 1's in the resulting code word either even or odd. A message of 4-bits and a parity bit P are shown in the table below:

Odd parity		Even parity	
Message	P	Message	P
0000	1	0000	0
0001	0	0001	1
0010	0	0010	1
0011	1	0011	0
0100	0	0100	1
0101	1	0101	0
0110	1	0110	0
0111	0	0111	1
1000	0	1000	1
1001	1	1001	0
1010	1	1010	0
1011	0	1011	1
1100	1	1100	0
1101	0	1101	1
1110	0	1110	1
1111	1	1111	0

Error Checking Mechanism:

→During the transmission of information from one location to another, an even parity bit is generated in the sending end for each message transmission. The message, together with the parity bit, is transmitted to its destination. The parity of
the received data is checked in the receiving end. If the parity of the received information is not even, it means that at least one bit has changed value during the transmission.
→This method detects one, three, or any odd combination of errors in each message that is transmitted. An even combination of errors is undetected. Additional error-detection schemes may be needed to take care of an even combination of errors.

3. Gray code (Reflected code)

It is a binary coding scheme used to represent digits generated from a mechanical sensor that may be prone to error. Used in telegraphy in the late 1800s, and also known as "reflected binary code".

Gray code was patented by Bell Labs researcher Frank Gray in 1947. In Gray code, there is **only one bit location different between two successive values**, which makes mechanical transitions from one digit to the next less error prone. The following chart shows normal binary representations from 0 to 15 and the corresponding Gray code.

Decimal digit	Binary code	Gray code
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

The Gray code is used in applications where the normal sequence of binary numbers may produce an error or ambiguity during the transition from one number to the next. If binary numbers are used, a change from 0111 to 1000 may produce an intermediate erroneous number 1001 if the rightmost bit takes more time to change than the other three bits. The Gray code eliminates this problem since only one bit changes in value during any transition between two numbers.

4. Alphanumeric codes

Alphanumeric character set is a set of elements that includes the 10 decimal digits, 26 letters of the alphabet and special characters such as $, %, + etc. It is necessary to formulate a binary code for this set to handle different data types. If only capital letters are included, we need a binary code of at least six bits, and if both uppercase letters and lowercase letters are included, we need a binary code of at least seven bits.

- **ASCII character code**

  The standard binary code for the alphanumeric characters is called ASCII (American Standard Code for Information Interchange). It uses seven bits to code 128 characters as shown in the table below. The seven bits of the code are designated by B_1 through B_7 with B_7 being the most significant bit.

American Standard Code for Information Interchange (ASCII)

	$B_7B_6B_5$							
$B_4B_3B_2B_1$	000	001	010	011	100	101	110	111
0000	NULL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

NOTE:
Decimal digits in ASCII can be converted to BCD by removing the three higher order bits, 011.

Various control character symbolic notation stands for:

NULL	NULL	DLE	Data link escape
SOH	Start of heading	DC1	Device control 1
STX	Start of text	DC2	Device control 2
ETX	End of text	DC3	Device control 3
EOT	End of transmission	DC4	Device control 4
ENQ	Enquiry	NAK	Negative acknowledge
ACK	Acknowledge	SYN	Synchronous idle
BEL	Bell	ETB	End of transmission block
BS	Backspace	CAN	Cancel
HT	Horizontal tab	EM	End of medium
LF	Line feed	SUB	Substitute
VT	Vertical tab	ESC	Escape
FF	Form feed	FS	File separator
CR	Carriage return	GS	Group separator
SO	Shift out	RS	Record separator
SI	Shift in	US	Unit separator
SP	Space	DEL	Delete

Example: ASCII for each symbol is ($B_7B_6B_5B_4B_3B_2B_1$)

G ← 100 0111, (← 010 1000, h ← 110 1000, > ← 011 1110 and so on.

- **EBCDIC character code**

  EBCDIC (Extended Binary Coded Decimal Interchange Code) is another alphanumeric code used in IBM equipment. It uses **eight bits** for each character. EBCDIC has the same character symbols as ASCII, but the bit assignment for characters is different. As the name implies, the binary code for the letters and numerals is an extension of the binary-coded decimal (BCD) code. This means that the last four bits of the code range from 0000 though 1001 as in BCD.

JL

Unit 2

Boolean algebra and Logic Gates

Before starting the discussion of Boolean algebra and complex logic gates (NAND, XOR etc), let me describe bit about the binary logic (which you guys have studied in discrete structure course) and how this logic is implemented in hardware using basic gates?

Binary logic

Binary logic consists of binary variables and logical operations. The variables are designated by letters of the alphabet such as *A, B,* C, *x, y,* Z, etc., with each variable having two and only two distinct possible values: 1 and 0. There are three basic logical operations: AND, OR, and NOT.

1. AND: This operation is represented by a dot or by the absence of an operator. For example, *x.y* = z or *xy* = z is read *"x* AND *y* is equal to *z."* The logical operation AND is interpreted to mean that z = 1 if and only if *x* = 1 *and y* = 1; otherwise z = 0. (Remember that *x, y,* and z are binary variables and can be equal either to 1 or 0, and nothing else.)
2. OR: This operation is represented by a plus sign. For example, *x* + *y* = z is read *"x* OR *y* is equal to *z,"* meaning that z = 0 if *x* = 0 *or* if *y* = 0 otherwise z = 1.
3. NOT: This operation is represented by a prime (sometimes by a bar). For example, *x'* = z is read "not *x* is equal to *z,"* meaning that z is what *x* is not. In other words, if *x* = 1, then z = 0; but if *x* = 0, then z = 1.

These definitions may be listed in a compact form using truth tables. A **truth table** is a table of all possible combinations of the variables showing the relation between the values that the variables may take and the result of the operation.

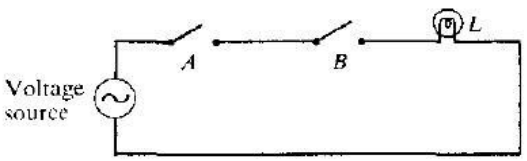
Truth Tables of Logical Operations

AND			OR			NOT	
x	*y*	*x y*	*x*	*y*	*x* + *y*	*x*	*x'*
0	0	0	0	0	0	0	1
0	1	0	0	1	1	1	0
1	0	0	1	0	1		
1	1	1	1	1	1		

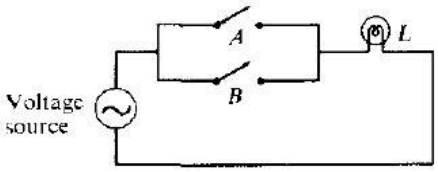
HEY! Binary logic should not be confused with binary arithmetic (However we use same symbols here). You should realize that an arithmetic variable designates a number that may consist of many digits. A logic variable is always either 1 or 0. For example, in binary arithmetic, 1 + 1 = 10 (read: "one plus one is equal to 2"), whereas in binary logic, we have 1 + 1 = 1 (read: "one OR one is equal to one").

Switching circuits and Binary Signals

The use of binary variables and the application of binary logic are demonstrated by the simple switching circuits shown below:



(a)Switches in series-Logical AND



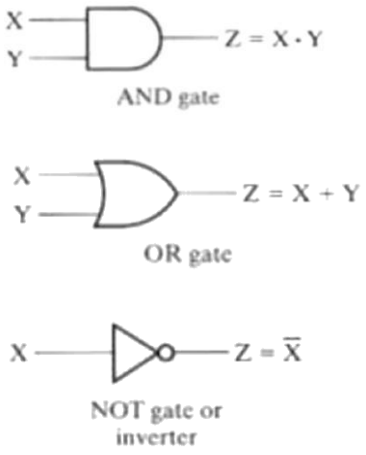
(b) Switches in parallel-Logical OR

Let the manual switches *A* and *B* represent two binary variables with values equal to 0 when the switch is open and 1 when the switch is closed. Similarly, let the lamp L represent a third binary variable equal to 1 when the light is on and 0 when off.

Electronic digital circuits are sometimes called **switching circuits** because they behave like a switch, with the active element such as a transistor either conducting (switch closed) or not conducting (switch open). Instead of changing the switch manually, an electronic switching circuit uses **binary signals** to control the conduction or non-conduction state of the transistor.

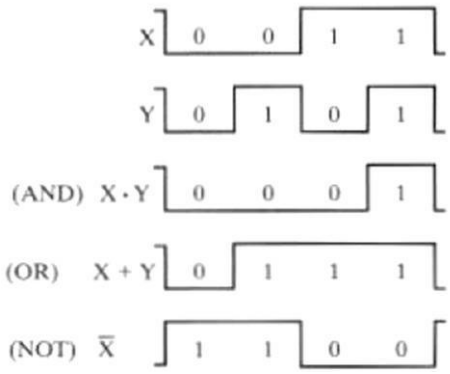
Basic Logic Gates (Digital logic gates will be covered in detail later)

Logic gates are electronic circuits that operate on one or more input signals to produce an output signal. Electrical signals such as voltages or currents exist throughout a digital system in either one of two recognizable values (bi-state 0 or 1). Voltage-operated circuits respond to two separate voltage ranges (Example of voltage ranges is discussed in unit 1) that represent a binary variable equal to logic 1 or logic 0. The graphics symbols used to designate the three types of gates AND, OR, and NOT are shown in Figure below:



(a) **Graphic symbols**

- These circuits, called *gates,* are blocks of hardware that produce a logic-1 or logic-0 output signal if input logic requirements are satisfied.
- Note that four different names have been used for the same type of circuits: digital circuits, switching circuits, logic circuits, and gates.
- AND and OR gates may have more than two inputs.
- NOT gate is single input circuit, it simply inverts the input.



(b) **Timing diagram**

The two input signals X and Y to the AND and OR gates take on one of four possible combinations: 00, 01, 10, or 11. These input signals are shown as timing diagrams, together with the timing diagrams for the corresponding output signal for each type of gate. The horizontal axis of a timing diagram represents time, and the vertical axis shows a signal as it changes between the two possible voltage levels. The low level represents logic 0 and the high level represents logic I. 1l1e AND gate responds with a logic-1 output signal when both input signals are logic-1. The OR gate responds with a logic-1 output signal if either input signal is logic-1.

Boolean algebra

In 1854 George Boole introduced a systematic treatment of logic and developed for this purpose an algebraic system now called *Boolean algebra.* In 1938 C. E. Shannon introduced a two-valued Boolean algebra called *switching algebra,* in which he demonstrated that the properties of bistable electrical switching circuits can be represented by this algebra.

Thus, the mathematical system of binary logic is known as **Boolean or switching algebra**. This algebra is conveniently used to describe the operation of complex networks of digital circuits. Designers of digital systems use Boolean algebra to transform circuit diagrams to algebraic expressions and vice versa. For any given algebra system, there are some initial assumptions, or postulates, that the system follows. We can deduce additional rules, theorems, and other properties of the system from this basic set of postulates. Boolean algebra systems often employ the postulates formulated by E. V. Huntington in 1904.

Postulates

Boolean algebra is an algebraic structure defined on a set of elements *B (Boolean system)* together with two binary operators + (OR) and • (AND) and unary operator ' (NOT), provided the following postulates are satisfied:

P1 → **Closure:** Boolean algebra is closed under the AND, OR, and NOT operations.

P2 → **Commutativity:** The • and + operators are commutative i.e. x + y = y + x and x • y = y • x, for all x, y ∈ B.

P3 → **Distribution:** • and + are distributive with respect to one another i.e.

$$x \bullet (y + z) = (x \bullet y) + (x \bullet z).$$

$x + (y \bullet z) = (x + y) \bullet (x + z)$, for all $x, y, z \in B$.

P4 → **Identity:** The identity element with respect to • is 1 and + is 0 i.e. x + 0 = 0 + x = x and x • 1=1• x = x. There is no identity element with respect to logical NOT.

P5 → **Inverse:** For every value x there exists a value x' such that x • x' = 0 and x + x' = 1. This value is the logical complement (or NOT) of x.

P6 → There exists at least two elements x, y ∈ B such that x ≠ y.

One can formulate many Boolean algebras (viz. set theory, n-bit vectors algebra), depending on the choice of elements of B and the rules of operation. Here, we deal only with a two-valued Boolean algebra, i.e., B = {0, 1}. Two-valued Boolean algebra has applications in set theory and in propositional logic. Our interest here is with the application of Boolean algebra to gate-type circuits.

Basic theorems and Properties of Boolean algebra

Duality

Postulates of Boolean algebra are found in pairs; one part may be obtained from the other if the binary operators and the identity elements are interchanged. This important property of Boolean algebra is called the **duality principle**. It states that "***Every algebraic expression deducible from the postulates of Boolean algebra remains valid if the operators and identity elements are interchanged***". In a two-valued Boolean algebra, the identity elements and the elements of the set *B* are the same: 1 and 0. If the ***dual*** of an algebraic expression is desired, we simply interchange OR and AND operators and replace 1's by 0's and 0's by 1's.

Basic theorems

The theorems, like the postulates, are listed in pairs; each relation is the dual of the one paired with it. The postulates are basic axioms of the algebraic structure and need no proof. The theorems must be proven from the postulates. six theorems of Boolean algebra are given below:

Theorem1:	Idempotence	(a) x + x = x	(b) x.x = x	One variable theorems
Theorem2:	Existence: 0&1	(a) x + 1 = 1	(b) x.0 = 0	
Theorem3:	Involution	(x')' = x		
Theorem4:	Associative	(a) x + (y + z) = (x + y) + z	(b) x(yz) = (xy)z	2 or 3 variable theorems
Theorem5:	Demorgan	(a) (x + y)' = x'y'	(b) (xy)' = x' + y'	
Theorem6:	Absorption	(a) x + xy = x	(b) x(x + y) = x	

Proofs:

(a) The proofs of the theorems with one variable are presented below:

THEOREM 1(a): x + x = x

x + x	= (x + x) . 1	(P4: Identity element)
	= (x + x)(x + x')	(P5: Existence of inverse)
	= x + xx'	(P3: Distribution)
	=x + 0	(P5: Existence of inverse)
	=x	(P4: Identity element)

THEOREM 1(b): x·x = x

x.x	= xx + 0	(P4: Identity element)
	=xx + xx'	(P5: Existence of inverse)
	=x(x + x')	(P3: Distribution)
	= x.1	(P5: Existence of inverse)
	=x	(P4: Identity element)

Hey! Each step in theorem 1(b) and 1(a) are dual of each other.

THEOREM 2(a): *x* + 1 = 1

x + 1	= 1· (x + 1)	(P4: Identity element)
	= (x + x')(x + 1)	(P5: Existence of inverse)
	=x + x'·1	(P3: Distribution)
	= x + x'	(P4: Identity element)
	= 1	(P5: Existence of inverse)

THEOREM 2(b): x.0 = 0 by duality.

THEOREM 3: *(x* ')' = *x.* From P5, we have *x* + *x'* = 1 and *x.x'* = 0, which defines the complement of *x.* The complement of *x'* is *x* and is also *(x')'.* Therefore, since the complement is unique, we have that *(x')'* = *x.*

(b) The theorems involving two or three variables may be proven algebraically from the postulates and the theorems that have already been proven. For example, lets prove Demorgan's theorem:

THEOREM 5(a): (x + y)' = x'y'

From postulate P5 (Existence of inverse), for every x in a Boolean algebra there is a unique x' such that

x + x' = 1 and x • x' = 0

So it is sufficient to show that x'y' is the complement of x + y. We'll do this by showing that (x + y) + (x'y') = 1 and (x + y) • (x'y') = 0.

(x + y) + (x'y')	= [(x + y) + x'] [(x + y) + y']	[OR distributes over AND (P3)]
	= [(y + x) + x'] [(x + y) + y']	[OR is commutative (P2)]
	= [y + (x + x')] [x + (y + y')]	[OR is associative (Theorem 3(a)), used twice]
	= (y + 1)(x + 1)	[Complement, x + x' = 1 (P5), twice]
	= 1 • 1	[x + 1 = 1, (Theorem 2), twice]
	= 1	[Idempotent, x • x = x (Theorem 1)]
Also,		
(x + y)(x'y')	= (x'y') (x + y)	[AND is commutative (P2)]
	= [(x'y') x] + [(x'y') y]	[AND distributes over OR (P3)]
	= [(y'x')x] + [(x'y')y]	[AND is commutative (P2)]
	= [y'(x'x)] + [x'(y'y)]	[AND is associative (Theorem 3(b)), twice]
	= [y'(xx')] + [x'(yy')]	[AND is commutative, twice]
	= *y' • 0+ + *x' • 0+	[Complement, x • x' = 0, twice]
	= 0 + 0	[x • 0 = 0, twice]
	= 0	[Idempotent, x + x = x]

THEOREM 5(a): (xy)' = x' + y' ← can be proved similarly as in Theorem 5(a). Each step in the proof of 5(b) is the dual of its 5(a) counterparts.

Hey! Theorems above can also be proved using truth tables (alternative to algebraic simplification). Viz. theorem 6(a) can be proved as:

			=
x	*y*	*xy*	*x* + *xy*
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	1

Operator Precedence

The operator precedence for evaluating Boolean expressions is

1. Parentheses → ()
2. NOT → '
3. AND → .
4. OR → +

In other words, the expression inside the parentheses must be evaluated before all other operations. The next operation that holds precedence is the complement, then follows the AND, and finally the OR.

Example: (a+b.c).d' → here we first evaluate 'b.c' and OR it with 'a' followed by ANDing with complement of 'd'.

Boolean Functions

A Boolean function is an expression formed with binary variables (variables that takes the value of 0 or 1), the two binary operators OR and AND, and unary operator NOT, parentheses, and an equal sign. For given value of the variables, the function can be either 0 or 1.

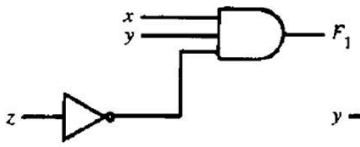
- **Boolean function represented as an algebraic expression:** Consider Boolean function $F_1 = xyz'$. Function F is equal to 1 if x=1, y=1 and z=0; otherwise $F_1 = 0$. Other examples are: $F_2 = x + y'z$, $F_3 = x'y'z + x'yz + xy'$, $F_4 = xy' + x'z$ etc.
- **Boolean function represented in a truth table:**
  The number of rows in the truth table is 2^n, where n is the number of binary variables in the function, The 1's and 0's combinations for each row is easily obtained from the binary numbers by counting from 0 to $2^n - 1$.

x	y	z	F_1	F_2	F_3	F_4
0	0	0	0	0	0	0
0	0	1	0	1	1	1
0	1	0	0	0	0	0
0	1	1	0	0	1	1
1	0	0	0	1	1	1
1	0	1	0	1	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	0

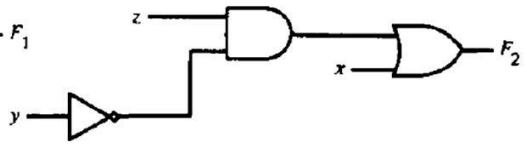
Million Dollar question ☺ : Is it possible to find two algebraic expressions that specify the same function? Answer is: **yes**. Being straightforward, the manipulation of Boolean algebra is applied mostly to the problem of finding simpler expressions for the same function.

Example: Functions F_3 and F_4 are same although they have different combinations of binary variables with in them.

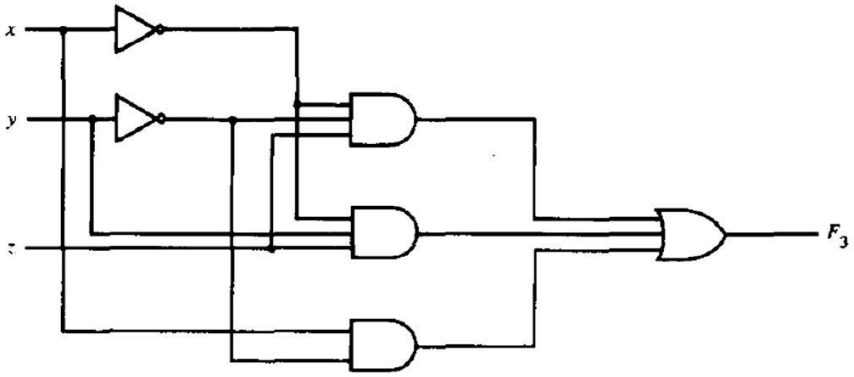
A Boolean function may be transformed from an algebraic expression into a logic diagram composed of AND, OR, and NOT gates. The implementation of the four functions introduced in the previous discussion is shown below:



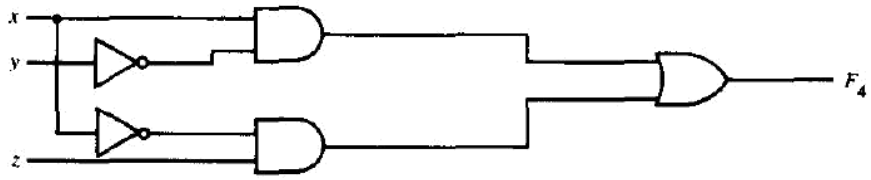
(a) $F_1 = xyz'$



(b) $F_2 = x + y'z$



(c) $F_3 = x'y'z + x'yz + xy'$



(d) $F_4 = xy' + x'z$

Fig: Implementation of Boolean functions with gates

Algebraic manipulation and simplification of Boolean function

- A ***literal*** is a primed or unprimed (i.e. complemented or un-complemented) variable. When a Boolean function is implemented with logic gates, each literal in the function designates an input to a gate, and each ***term*** is implemented with a gate.
- The **minimization** of the number of literals and the number of terms results in a circuit with less equipment. It is not always possible to minimize both simultaneously; usually, further criteria must be available. At the moment, we shall narrow the minimization criterion to literal minimization. We shall discuss other criteria in unit 3.
- The number of literals in a Boolean function can be minimized by algebraic manipulations. Unfortunately, there are no specific rules to follow that will guarantee the final answer. The only method available is a cut-and-try procedure employing the postulates, the basic theorems, and any other manipulation method that becomes familiar with use. The following examples illustrate this procedure.

→ Simplify the following Boolean functions to a minimum number of literals.

1. x + x 'y = (x + x ')(x + y) = 1.(x + y) = x + y
2. x(x' + y) = xx' + xy = 0 + xy = xy
3. x'y'z + x'yz + xy' = x'z(y' + y) + xy = x'z + xy
4. xy + x' z + yz = xy + x' z + yz (x + x')
   = xy + x'z + xyz + x'yz
   = xy(1 + z) + x'z(1 + y)
   = xy + x'z
5. (x + y)(x' + z)(y + z) = (x + y)(x' + z) by duality from function 4.

Complement of a function

The complement of a function *F* is *F'* and is obtained from an interchange of 0's for 1's and 1's for 0's in the value of *F.* The complement of a function may be derived algebraically through DeMorgan's theorem. DeMorgan's theorems can be extended to three or more variables. The three-variable form of the first DeMorgan's theorem is derived below.

(A + B + C)'	= (A + X)'	let B + C = X
	= A'X'	(DeMorgan)
	= A'· (B + C)'	substitute B + C = X
	= A'.(B'C')	(DeMorgan)
	= A'B'C'	(associative)

DeMorgan's theorems for any number of variables resemble in form the two variable case and can be derived by successive substitutions similar to the method used in the above derivation. These theorems can be generalized as follows:

(A + B + C + D + ... + F)' = A'B'C'D'… F'

(ABCD ... F)' = A' + B' + C' + D' + ... + F'

The generalized form of De Morgan's theorem states that the complement of a function is obtained by interchanging AND and OR operators and complementing each literal.

Two ways of getting complement of a Boolean function:

1. *Applying DeMorgan's theorem:*

   Question: Find the complement of the functions $F_1 = x'yz' + x'y'z$ *and* $F_2 = x(y'z' + yz)$.

   → By applying DeMorgan's theorem as many times as necessary, the complements are obtained as follows:

   $F_1' = (x'yz' + x'y'z)' = (x'yz')'(x'y'z)' = (x + y' + z)(x + y + z')$

   $F_2' = [x(y'z' + yz)]' = x' + (y'z' + yz)' = x' + (y'z')' \cdot (yz)' = x' + (y + z)(y' + z')$

2. *First finding dual of the algebraic expression and complementing each literal*

   Question: Find the complement of the functions F_1 and F_2 of example above by taking their duals and complementing each literal.

   - $F_1 = x'yz' + x'y'z$.
     The dual of F_1 is $(x' + y + z')(x' + y' + z)$.
     Complement each literal: $(x + y' + z)(x + y + z') = F_1'$.
   - $F_2 = x(y'z' + yz)$.
     The dual of F_1 is $x + (y' + z')(y + z)$.
     Complement each literal: $x' + (y + z)(y' + z') = F_2'$.

Other logic operations

When the binary operators AND and OR are placed between two variables, *x* and *y,* they form two Boolean functions, *x.y* and *x + y,* respectively. There are 2^2 functions for *n* binary variables. For two variables, n^2, and the number of possible Boolean functions is 16. Therefore, the AND and OR functions are only two of a total of 16 possible functions. It would be instructive to find the other 14 functions and investigate their properties.

Truth Tables for the 16 Functions of Two Binary Variables

x	y	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
Operator symbol			.	/		/		$\oplus$	+	$\downarrow$	$\odot$	′	$\subset$	′	$\supset$	$\uparrow$	

The 16 functions listed in truth table form above can be expressed algebraically by means of Boolean expressions as in following table. Each of the functions is listed with an accompanying name and a comment that explains the function in some way. The 16 functions listed can be subdivided into three categories:

1. Two functions that produce a constant 0 or 1.
2. Four functions with unary operations: complement and transfer.
3. Ten functions with binary operators that define eight different operations: AND, OR, NAND. NOR, exclusive OR, equivalence, inhibition, and implication.

Boolean functions	Operator symbol	Name	Comments
$F_0 = 0$		Null	Binary constant 0
$F_1 = xy$	$x \cdot y$	AND	x and y
$F_2 = xy'$	x/y	Inhibition	x but not y
$F_3 = x$		Transfer	x
$F_4 = x'y$	y/x	Inhibition	y but not x
$F_5 = y$		Transfer	y
$F_6 = xy' + x'y$	$x \oplus y$	Exclusive-OR	x or y but not both
$F_7 = x + y$	$x + y$	OR	x or y
$F_8 = (x + y)'$	$x \downarrow y$	NOR	Not-OR
$F_9 = xy + x'y'$	$x \odot y$	Equivalence	x equals y
$F_{10} = y'$	y'	Complement	Not y
$F_{11} = x + y'$	$x \subset y$	Implication	If y then x
$F_{12} = x'$	x'	Complement	Not x
$F_{13} = x' + y$	$x \supset y$	Implication	If x then y
$F_{14} = (xy)'$	$x \uparrow y$	NAND	Not-AND
$F_{15} = 1$		Identity	Binary constant 1

Table: **Boolean Expressions for the 16 Functions of Two Variables**

Digital Logic gates (In detail)

Boolean functions are expressed in terms of AND, OR, and NOT logic operations, and hence are easier to implement with these types of gates. The possibility of constructing gates for the other logic operations is of **practical interest**. Factors to be weighed when considering the construction of other types of logic gates are:

- The feasibility and economy of producing the gate with physical components
- The possibility of extending the gate to more than two inputs
- The basic properties of the binary operator such as commutativity and associativity, and
- The ability of the gate to implement Boolean functions alone or in conjunction with other gates.

Of the 16 functions defined in Table above, two are equal to a constant and four others are repeated twice. There are only ten functions left to be considered as candidates for logic gates. Two, inhibition and implication, are not commutative or associative and thus are impractical to use as standard logic gates. The other eight: **complement**, **transfer**, **AND**, **OR**, **NAND**, **NOR**, **exclusive-OR**, and **equivalence**, are used as standard gates in digital design.
The graphic symbols and truth tables of the eight gates are shown below:

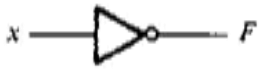
Name	Graphic symbol	Algebraic function	Truth table
AND		$F = xy$	$x\ y \mid F$ $0\ 0 \mid 0$ $0\ 1 \mid 0$ $1\ 0 \mid 0$ $1\ 1 \mid 1$
OR		$F = x + y$	$x\ y \mid F$ $0\ 0 \mid 0$ $0\ 1 \mid 1$ $1\ 0 \mid 1$ $1\ 1 \mid 1$
Inverter		$F = x'$	$x \mid F$ $0 \mid 1$ $1 \mid 0$
Buffer		$F = x$	$x \mid F$ $0 \mid 0$ $1 \mid 1$
NAND		$F = (xy)'$	$x\ y \mid F$ $0\ 0 \mid 1$ $0\ 1 \mid 1$ $1\ 0 \mid 1$ $1\ 1 \mid 0$
NOR		$F = (x + y)'$	$x\ y \mid F$ $0\ 0 \mid 1$ $0\ 1 \mid 0$ $1\ 0 \mid 0$ $1\ 1 \mid 0$
Exclusive-OR (XOR)		$F = xy' + x'y$ $= x \oplus y$	$x\ y \mid F$ $0\ 0 \mid 0$ $0\ 1 \mid 1$ $1\ 0 \mid 1$ $1\ 1 \mid 0$
Exclusive-NOR or equivalence		$F = xy + x'y'$ $= x \odot y$	$x\ y \mid F$ $0\ 0 \mid 1$ $0\ 1 \mid 0$ $1\ 0 \mid 0$ $1\ 1 \mid 1$

Fig: **listing of eight gates**

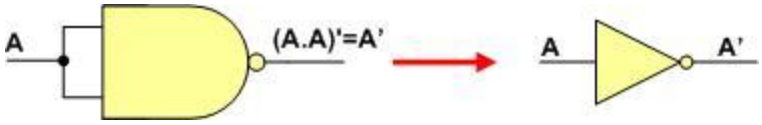
Universal gates

A universal gate is a gate which can implement any Boolean function without need to use any other gate type. The **NAND** and **NOR** gates are universal gates. In practice, this is advantageous since NAND and NOR gates are economical and easier to fabricate and are the basic gates used in all IC digital logic families.

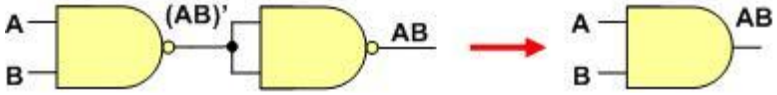
1. **NAND Gate is a Universal Gate**

   To prove that any Boolean function can be implemented using only NAND gates, we will show that the AND, OR, and NOT operations can be performed using only these gates.

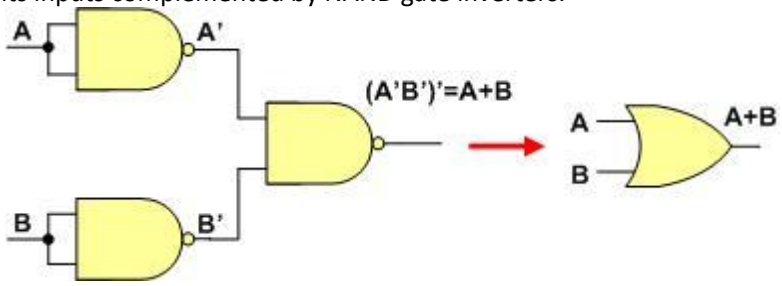
   - **Implementing an Inverter Using only NAND Gate:** All NAND input pins connected to the input signal A gives an output A'.



   - **Implementing AND Using only NAND Gates:** The AND is replaced by a NAND gate with its output complemented by a NAND gate inverter.



   - **Implementing OR Using only NAND Gates:** The OR gate is replaced by a NAND gate with all its inputs complemented by NAND gate inverters.

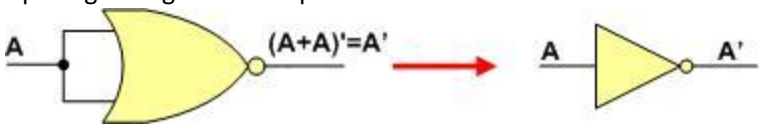


   Thus, the **NAND gate** is a universal gate since it can implement the AND, OR and NOT functions.

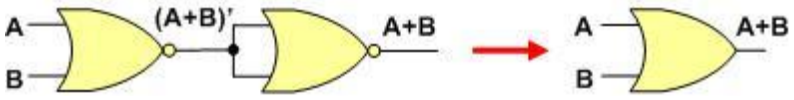
2. **NOR Gate is a Universal Gate:**

   To prove that any Boolean function can not be implemented using only NOR gates, we will show that the AND, OR, and NOT operations can be performed using only these gates.

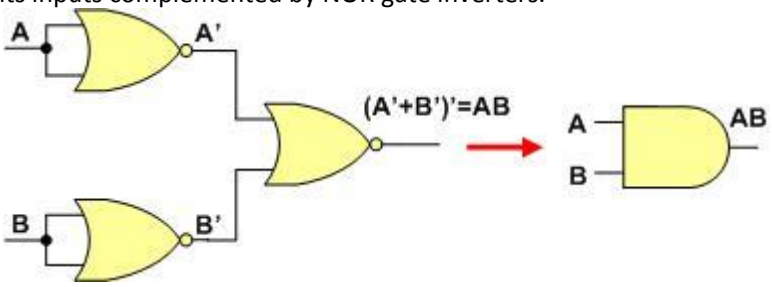
   - **Implementing neither an Inverter Using only NOR Gate:** All NOR input pins connect to the input signal **A** gives an output **A'**.



   - **Implementing OR Using only NOR Gates:** The OR is replaced by a NOR gate with its output complemented by a NOR gate inverter.



- **Implementing AND Using only NOR Gates:** The AND gate is replaced by a NOR gate with all its inputs complemented by NOR gate inverters.



Thus, the **NOR gate** is a universal gate since it can implement the AND, OR and NOT functions.

Extending gates to multiple inputs

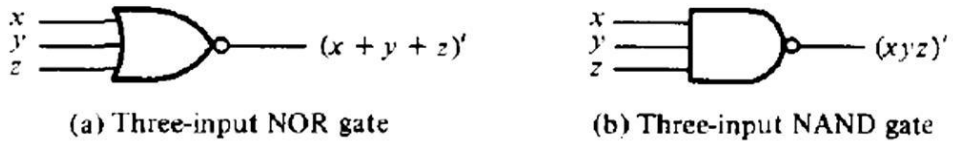
The gates shown in Fig above, except for the inverter and buffer, can be extended to have more than two inputs. A gate can be extended to have multiple inputs if the binary operation it represents is commutative and associative.

- The AND and OR operations, defined in Boolean algebra, possess these two properties. For the OR function, we have $x + y = y + x$ *commutative* and $(x + y) + z = x + (y + z) = x + y + z$ associative, which indicates that the gate inputs can be interchanged and that the OR function can be extended to three or more variables.
- The NAND and NOR functions are commutative and but not associative *x↓(y ↓ z) ≠ (x ↓ y) ↓z+

$$(x \downarrow y) \downarrow z = [(x + y)' + z]' = (x + y)z' = xz' + yz'$$

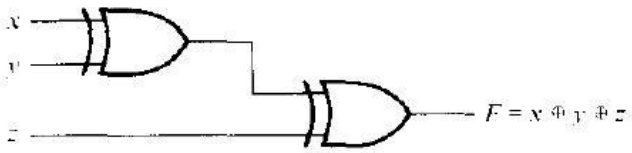
$$x \downarrow (y \downarrow z) = [x + (y + z)']' = x'(y + z) = x'y + x'z$$

Their gates can be extended to have more than two inputs, provided the definition of the operation is slightly modified. We define the multiple NOR (or NAND) gate as a complemented OR (or AND) gate i.e. x ↓ y ↓ z = (x + y + z)' and x ↑y ↑ z = (xyz)'.

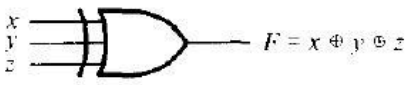


(a) Three-input NOR gate (b) Three-input NAND gate

- The exclusive-OR and equivalence gates are both commutative and associative and can be extended to more than two inputs



(a) Using 2-input gates



(b) 3-input gate

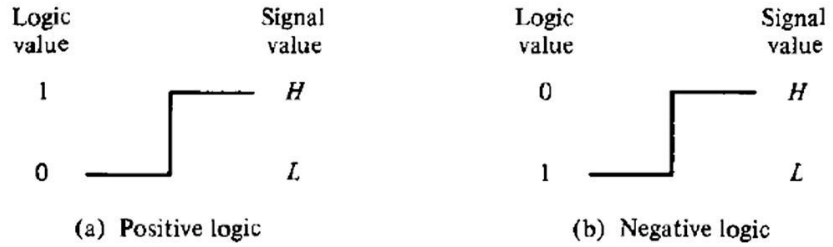
x	y	z	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

(c) Truth table

Fig: **3-input XOR gate**

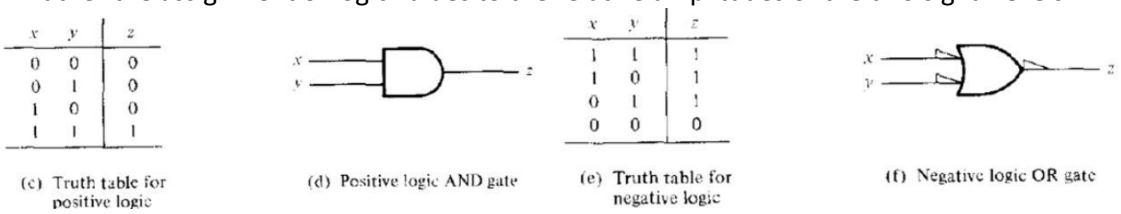
Positive and Negative Logic

The binary signal at the inputs and outputs of any gate has one of two values, except during transition. One signal value represents logic-1 and the other logic-0. So there is a possibility of two different assignments of signal level to logic value, as shown in Fig.



(a) Positive logic (b) Negative logic

- Choosing the high-level *H* to represent logic-1 defines a **positive logic system**.
- Choosing the low-level *L* to represent logic-1 defines a **negative logic system**.
- The terms positive and negative are somewhat **misleading** since both signals may be positive or both may be negative. It is not the actual signal values that determine the type of logic, but rather the assignment of logic values to the relative amplitudes of the two signal levels.



x	y	z
0	0	0
0	1	0
1	0	0
1	1	1

(c) Truth table for positive logic

(d) Positive logic AND gate

x	y	z
1	1	1
1	0	1
0	1	1
0	0	0

(e) Truth table for negative logic

(f) Negative logic OR gate

Fig: Demonstration of Positive and negative logic

Unit 3
Simplification of Boolean functions

Canonical and standard forms

We can write Boolean expressions in many ways, but some ways are more useful than others. We will look first at the "term" types, made up of "literals".

Minterms

- A **minterm** is a special product (ANDing of terms) of literals, in which each input variable appears exactly once.
- A function with n variables has 2^n minterms (since each variable can appear complemented or not)
- A three-variable function, such as f(x, y, z), has $2^3 = 8$ minterms:

  x'y'z' x'y'z x'yz' x'yz
  xy'z' xy'z xyz' xyz
- Each minterm is **true** for exactly one combination of inputs:

Maxterms

- A maxterm is a **sum** (or ORing of terms) of literals, in which each input variable appears exactly once.
- A function with n variables has 2^n maxterms
- The maxterms for a three-variable function f(x, y, z):

  x' + y' + z' x' + y' + z x' + y + z' x'+ y + z x + y' + z' x + y' + z
  x + y + z' x + y + z
- Each maxterm is **false** for exactly one combination of inputs:

			Minterms		Maxterms	
x	y	z	Term	Designation	Term	Designation
0	0	0	$x'y'z'$	m_0	$x + y + z$	M_0
0	0	1	$x'y'z$	m_1	$x + y + z'$	M_1
0	1	0	$x'yz'$	m_2	$x + y' + z$	M_2
0	1	1	$x'yz$	m_3	$x + y' + z'$	M_3
1	0	0	$xy'z'$	m_4	$x' + y + z$	M_4
1	0	1	$xy'z$	m_5	$x' + y + z'$	M_5
1	1	0	xyz'	m_6	$x' + y' + z$	M_6
1	1	1	xyz	m_7	$x' + y' + z'$	M_7

Table: Minterms and Maxterms for 3 Binary Variables with their symbolic shorthand

Hey! Each maxterm is the complement of its corresponding minterm and vice versa (viz. $m_0 = M_0'$, $M_4 = m_4'$ etc.).

JL

A Boolean function may be expressed algebraically (SOP or POS form) from a given truth table by:

- Forming a **minterm** for each combination of the variables that produces a 1 in the function, and then taking the OR of all those terms.
- Forming a **maxterm** for each combination of the variables that produces a 0 in the function, and then taking the AND of all those maxterms.

Canonical forms

Boolean functions expressed as a sum of min terms or product of maxterms are said to be in ***canonical form***. These complementary techniques are describes below. Canonical form is not efficient representation but sometimes useful in analysis and design. In an expression in canonical form, every variable appears in every term.

Sum of Minterms (Sum of Products or SOP)

We have seen, one can obtain 2" distinct minterms form n binary input variables and that any Boolean function can be expressed as a sum of minterms. The minterms whose sum defines the Boolean function are those that give the 1's of the function in a truth table. It is sometimes convenient to express the Boolean function in its **sum of minterms form**. If not in this form, it can be made so by first expanding the expression into a sum of AND terms. Each term is then inspected to see if it contains all the variables. If it misses one or more variables, it is ANDed with an expression such as $x + x'$, where x is one of the missing variables.

Question: Express the Boolean function in a sum of minterms.
Solution: The function has three variables *A, B,* and C.

- The first term *A* is missing two variables; therefore:

  A = A (B + B ') = AB + AB' [B is missing variable]

  This is still missing one variable C, so A = AB(C + C') + AB'(C + C') = ABC + ABC' + AB'C + AB'C'
- The second term B'C is missing one variable: $B'C = B'C(A + A') = AB'C + A'B'C$
- Combing all terms, we have F = A+B'C = ABC + ABC' + AB'C + AB'C' + AB'C + A'B'C
- But AB'C appears twice, and according to THEOREM 1 of Boolean algebra $x + x = x$, it is possible to remove one of them. Rearranging the minterms in ascending order, we finally obtain:

  $$F = A'B'C + AB'C' + AB'C + ABC' + ABC$$
  $$= m_1 + m_4 + m_5 + m_6 +$$

  m_7 Shorthand notation,

  $_{(\ ,\ ,C)=}$ $_{(1,4,5,6,7)}$

  The summation symbol stands for the ORing of terms: the numbers following it are the minterms of the function.

An **alternate procedure** for deriving the minterms of a Boolean function is to obtain the truth table of the function directly from the algebraic expression and then read the minterms from the truth table.

Truth Table for $F = A + B'C$

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Truth table for $F = A + B$ 'C, from the truth table, we can then read the five minterms of the function to be 1, 4, 5, 6, and 7.

Product of Maxterms (Product of Sums or POS)

Each of the 2^2 functions of *n* binary variables can be also expressed as a **product of maxterms**. To express the Boolean function as a product of maxterms, it must first be brought into a form of OR terms. This may be done by using the distributive law, *x + yz = (x + y)(x + z).* Then any missing variable *x* in each OR term is ORed with *xx'.* This procedure is clarified by the following example:

Question: Express the Boolean function *F = xy + x'z* in a product of maxterm form.

Solution:

- First, con vert the function into OR terms using the distributive law: F = xy + x' z = (xy + x ')(xy + z)
  = (x + x')(y + x')(x + z)(y + z)
  = (x' + y)(x + z)(y + z)
- The function has three variables: *x, y,* and z. Each OR term is missing one variable; therefore:
  x' + y = x' + y + zz' = (x' + y + z)(x' + y + z ')
  x + z = x + z + yy' = (x + y + z)(x + y' + z)
  y + z = y + z + xx' = (x + y + z)(x' + y + z)
- Combing all maxterms and removing repeated terms:

  $= M_0M_2M_4M_5$

  Shorthand notation:

  The product symbol denotes the ANDing of maxterms; the numbers are the maxterms of the function.

Conversion between canonical forms

The complement of a function expressed as the sum of minterms equals the sum of minterms missing from the original function.

For example: Consider the function, $(\ ,\ ,C)$ $=$ $(1, 4, 5, 6, 7)$

Its complement can be expressed as: $'\ ,\ ,C =$ $0, 2, 3$ $= 0 + 2 + 3$

Now, if we take the complement of *F'* by DeMorgan's theorem, we obtain *F* in a different form:

$\ ,\ ,C = +\ +$ $= \ ,\ ,\ = (0, 2, 3)$

3

The last conversion follows from the definition of min terms and maxterms that $m_j' = M_j$

General Procedure: To convert from one canonical form to another, interchange the symbols and and list those numbers missing from the original form. In order to find the missing terms, one must realize that the total number of minterms or maxterms is 2^n (numbered as 0 to 2^{n-1}), where *n* is the number of binary variables in the function.

Consider a function, *F = xy + x'z.* First, we derive the truth table of the function

x	y	z	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

- The minterms of the function are read from the truth table to be 1, 3, 6, and 7. The function expressed in sum of minterms is $(\ldots) = \sum(1,3,6,7)$
- Since there are a total of eight minterms or maxterms in a function of three variables, we determine the missing terms to be 0, 2, 4, and 5. The function expressed in product of maxterm is $(\ldots) = \prod(0,2,4,5)$

Standard Forms

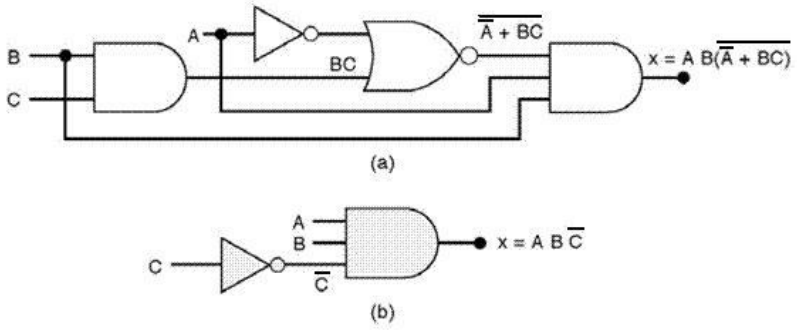
- This is another way to express Boolean functions. In this configuration, the terms that form the function may contain one, two, or any number of literals. There are two types of standard forms: the **sum of products** and **product of sums**.
- The ***sum of products*** is a Boolean expression containing AND terms, called *product terms,* of one or more literals each. The *sum* denotes the ORing of these terms.

  Example: F_1 = y' + xy + x'yz', the expression has three product terms of one, two, and three literals each, respectively. Their sum is in effect an OR operation.
- A ***product of sums*** is a Boolean expression containing OR terms, called *sum terms.* Each term may have any number of literals. The *product* denotes the ANDing of these terms. An example of a function expressed in product of sums is $F_1 = x(y' + z)(x' + y + z' + w)$, this expression has three sum terms of one, two, and four literals each, respectively. The product is an AND operation.
- Function can also be in **non-standard form**: $F_3 = (AB + CD)(A'B' + CD')$ is neither in SOP nor in POS forms. It can be changed to a standard form by using the distributive law as $F_3 = A'B'CD + ABC'D'$.

Simplifying Logic Circuits (Boolean functions): Two methods

First obtain one expression for the circuit, then try to simplify. Example: In diagram below, (a) can be simplified to (b) using one of following two methods:

1. Algebraic method (use Boolean algebra theorems)
2. Karnaugh mapping method (systematic, step-by-step approach)



METHOD 1: Minimization by Boolean algebra

- Make use of relationships and theorems to simplify Boolean Expressions
- Perform algebraic manipulation resulting in a complexity reduction.
- This method relies on your algebraic skill
- 3 things to try:
  - Grouping

    A + AB + BC

    A (1+ B) + BC

    A + BC [since 1+ B =1]
  - Multiplication by redundant variables
    - Multiplying by terms of the form A + A' does not alter the logic
    - Such multiplications by a variable missing from a term may enable minimization

      Example:

$$\begin{aligned} AB + A\overline{C} + BC &= AB(C + \overline{C}) + A\overline{C} + BC \\ &= ABC + AB\overline{C} + A\overline{C} + BC \\ &= BC(1 + A) + A\overline{C}(1 + B) \\ &= BC + A\overline{C} \end{aligned}$$

  - Application of DeMorgan's theorem
    - Expressions containing several inversions stacked one upon the other often are simplified by using DeMorgan's law which **unwraps** multiple inversions.
    - Example:

$$\begin{aligned} \overline{A\overline{B}\overline{C} + \overline{A}C\overline{D} + B\overline{C}} &= \overline{(\overline{A} + B + \overline{C}) + (\overline{A} + \overline{C} + \overline{D}) + B\overline{C}} \\ &= \overline{(\overline{A} + B + \overline{C} + \overline{D}) + B\overline{C}} \\ &= \overline{(\overline{A} + B + \overline{C} + \overline{D})} \\ &= A\overline{B}CD \end{aligned}$$

Question (Logic Design): Design a logic circuit having 3 inputs, A, B, C will have its output HIGH only when a majority of the inputs are HIGH.

Solution:

Step 1 Set up the truth table:
Step 2 Write minterm (AND term) for each case where the output is 1.

A	B	C	x	
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	1	$\rightarrow \overline{A}BC$
1	0	0	0	
1	0	1	1	$\rightarrow A\overline{B}C$
1	1	0	1	$\rightarrow AB\overline{C}$
1	1	1	1	$\rightarrow ABC$

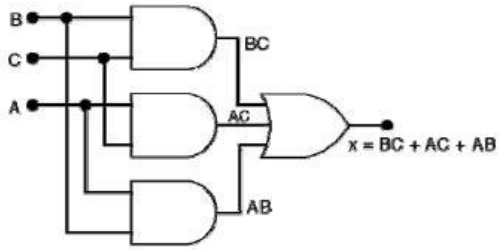
Step 3 Write the SOP from the output.

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

Step 4 Simplify the output expression

$$\begin{aligned} x &= \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC \\ x &= \overline{A}BC + ABC + A\overline{B}C + \boxed{ABC} + AB\overline{C} + \boxed{ABC} \\ &= BC(\overline{A} + A) + AC(\overline{B} + B) + AB(\overline{C} + C) \\ &= BC + AC + AB \end{aligned}$$

Step 5 Implement the circuit.



METHOD 2: Minimization by K-map (Karnaugh map)

Algebraic minimization of Boolean functions is rather awkward because it lacks specific rules to predict each succeeding step in the manipulative process. The map method provides a simple straightforward procedure for minimizing Boolean functions. This method may be regarded as a pictorial form of a truth table. The map method, first proposed by Veitch and modified by Karnaugh, is also known as the "Veitch diagram" or the "Karnaugh map."

- The k-map is a diagram made up of grid of squares.
- Each square represents one minterm.
- The minterms are ordered according to Gray code (only one variable changes between adjacent squares).
- Squares on edges are considered adjacent to squares on opposite edges.
- Karnaugh maps become clumsier to use with more than 4 variables.

In fact, the map presents a visual diagram of all possible ways a function may be expressed in a standard form. By recognizing various patterns, the user can derive alternative algebraic expressions for the same function, from which he can select the simplest one. We shall assume that the simplest algebraic expression is anyone in a sum of products or product of sums that has a minimum number of literals. (This expression is not necessarily unique)

Two variable maps

There are four minterms for a Boolean function with two variables. Hence, the two-variable map consists of four squares, one for each minterm, as shown in Figure:

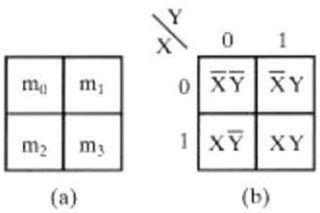


Fig: Two-variable map

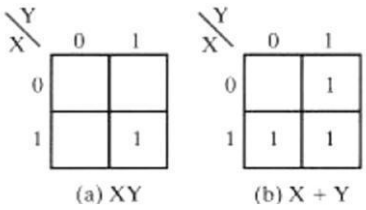


Fig: Representation of functions in the map

Three variable maps

There are eight minterms for three binary variables. Therefore, a three-variable map consists of eight squares, as shown in Figure. The map drawn in part (b) is marked with binary numbers for each row and each column to show the binary values of the minterms.

Hey! I will explain the process of simplification through examples.

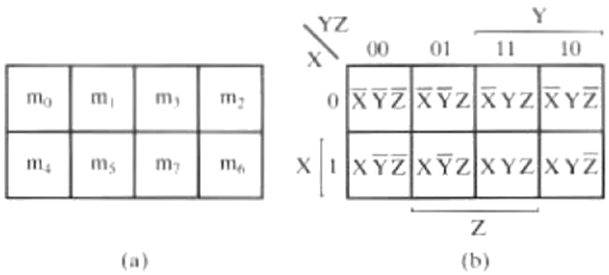
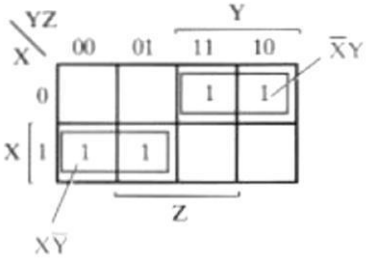


Fig: Three-variable map

Question: Simplify the Boolean function

Solution:

Step 1: First, a 1 is marked in each minterm that represents the function. This is shown in Figure, where the squares for minterms 010, 011, 100, and 101 are marked with 1's. For convenience, all of the remaining squares for which the function has value 0 are left blank rather than entering the 0's.



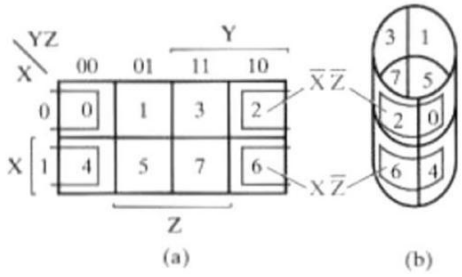
Step 2: Explore collections of squares on the map representing product terms to be considered for the simplified expression. We call such objects ***rectangles***. Rectangles that correspond to product terms are restricted to contain numbers of squares that are powers of 2, such as 1, 2(pair), 4(quad), 8(octet) ... **Goal** is to find the fewest such rectangles that include all of the minterms marked with 1's. This will give the fewest product terms.

Step 3: Sum up each rectangles (it may be pair, quad etc representing term) eliminating the variable that changes in value (or keeping intact the variables which have same value) throughout the rectangle.

From figure, logical sum of the corresponding two product terms gives the optimized expression for *F:*

$$F = X'Y+XY'$$

Point to understand



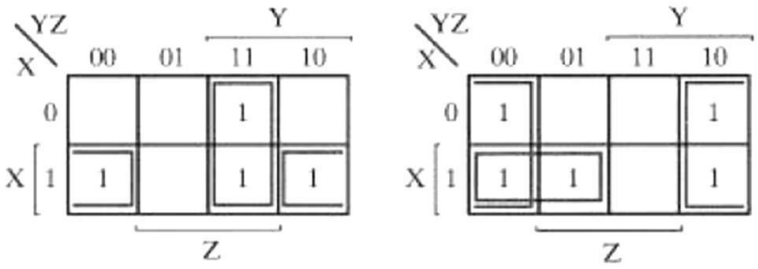
Minterm adjacencies are circular in nature. This figure shows Three-Variable Map in Flat and on a Cylinder to show adjacent squares.

Question: Simplify the following two Boolean functions:

$(x, y, z) = \sum(3, 4, 6, 7)$

$(x, y, z) = \sum(0, 2, 4, 5, 6)$

Solution: The map for F and G are given below:



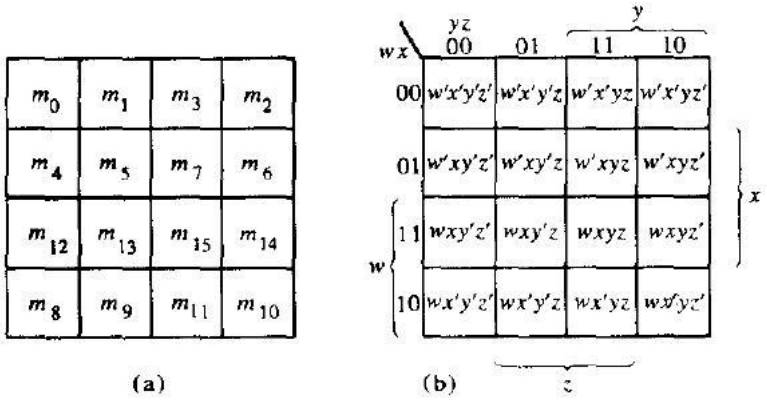
Writing the simplified expression for both functions:

F = YZ +XZ' and G = Z'+XY'

Hey! On occasion, there are alternative ways of combining squares to produce equally optimized expressions. It's upon your skill to use the easy and efficient strategy.

Four variable maps

The map for Boolean functions of four binary variables is shown in Fig below. In (a) are listed the 16 minterms and the squares assigned to each. In (b) the map is redrawn to show the relationship with the four variables.

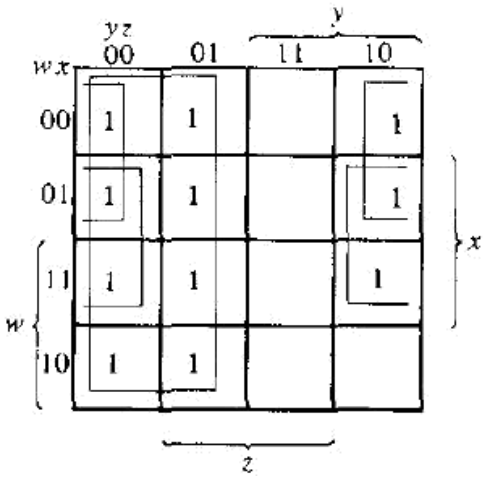


The map minimization of four-variable Boolean functions is similar to the method used to minimize three-variable functions. Adjacent squares are defined to be squares next to each other. In addition, the map is considered to lie on a surface with the top and bottom edges, as well as the right and left edges, touching each other to form adjacent squares. For example, m_0 and m_2 form adjacent squares, as do m_3 and m_{11}.

Question: Simplify the Boolean function $(\ldots) = \sum(0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14)$

Solution:

Since the function has four variables, a four-variable map must be used. Map representation is shown below:



The simplified function is: F = y' + w'z' + xz'

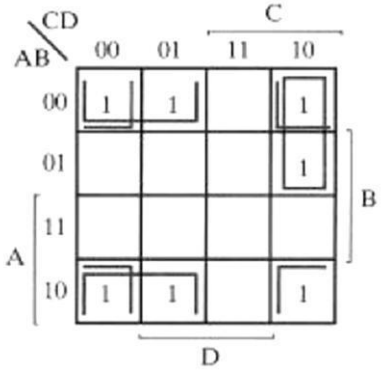
Question: Simplify the Boolean function
F = A'B'C' + B'CD' + A'BCD' + AB'C'

Solution:

First try just to reduce the standard form function into SOP form and then mark 1 for each minterm in the map.

F = A'B'C' + B'CD' + A'BCD' + AB'C'
=A'B'C'(D+D') + B'CD(A+A') + A'BCD' + AB'C'(D+D')
= A'B'C'D+ A'B'C'D' + AB'CD+A' B'CD + A'BCD' + AB'C'D+ AB'C'D'

This function also has 4 variables, so the area in the map covered by this function consists of the squares marked with 1's in following Fig.



Optimized function thus is: *F = B'D' + B'C' + A'CD'*

Don't care Conditions

The logical sum of the minterms associated with a Boolean function specifies the conditions under which the function is equal to 1. The function is equal to 0 for the rest of the min terms. This assumes that all the combinations of the values for the variables of the function are valid. In practice, there are some applications where the function is not specified for certain combinations of the variables.

Example: four-bit binary code for the decimal digits has six combinations that are not used and consequently are considered as unspecified.

In most applications, we simply **don't care what value is assumed by the function** for the unspecified minterms. For this reason, it is customary to call the unspecified minterms of a function **don't-care conditions**. These don't-care conditions can be used on a map to provide further simplification of the Boolean expression.

Don't-care minterm is a combination of variables whose logical value is not specified. To distinguish the don't-care condition from 1's and 0's, an X is used. Thus, an X inside a square in the map indicates that we don't care whether the value of 0 or 1 is assigned to *F* for the particular min term.

When choosing adjacent squares to simplify the function in a map, the don't-care minterms may be assumed to be either 0 or 1. When simplifying the function, we can choose to include each don't-care minterm with either the 1's or the 0's, depending on which combination gives the simplest expression.

Question: Simplify the Boolean function

$$F(\ldots) = \sum(1, 3, 7, 11, 15)$$

that has the don't-care conditions

$$d(\ldots) = (0, 2, 5)$$

Solution:

The map simplification is shown below. The minterms of *F* are marked by 1's, those of *d* are marked by X's, and the remaining squares are filled with 0's.

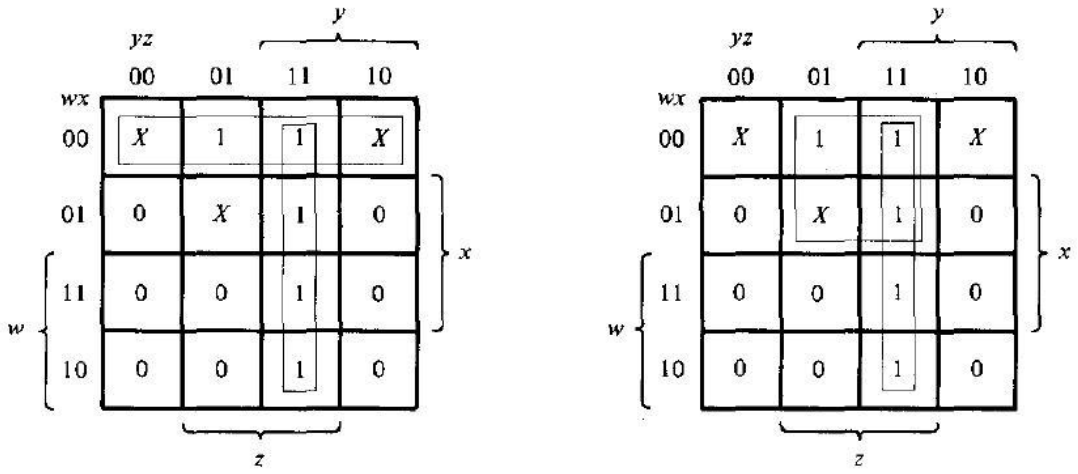


Fig: Map simplification with don't care conditions

In part (a) of the diagram, don't-care minterms 0 and 2 are included with the 1's, which results in the simplified function

F = yz + w'x'

In part (b), don't-care minterm 5 is included with the 1's and the simplified function now is

F = yz + w'z

Either one of the above expressions satisfies the conditions stated for this example.

Product of sum simplification

The optimized Boolean functions derived from the maps in all of the previous examples were expressed in sum-of-products (SOP) form. With only minor modification, the product-of-sums form can be obtained.

Procedure:

The 1's placed in the squares of the map represent the minterms of the function. The minterms not included in the function belong to the complement of the function. From this, we see that the complement of a function is represented in the map by the squares not marked by 1's. If we mark the empty squares with 0's and combine them into valid rectangles, we obtain an optimized expression of the complement of the function (F'). We then take the complement of F to obtain the function *F* as a product of sums.

Question: Simplify the following Boolean function (, , ,) = (0, 1, 2, 5, 8, 9, 10) in

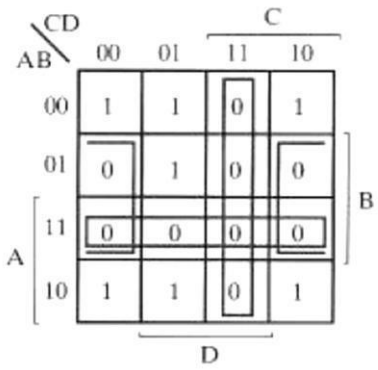
(a) Sum of products (SOP) and

(b) Product of sums (POS).

Solution:

The 1's marked in the map below represent all the minterms of the function. The squares marked with 0's represent the minterms not included in *F* and, therefore, denote *F'*.

(a) Combining the squares with 1's gives the simplified function in sum of products:
F = B'D' + B'C' + A'C'D



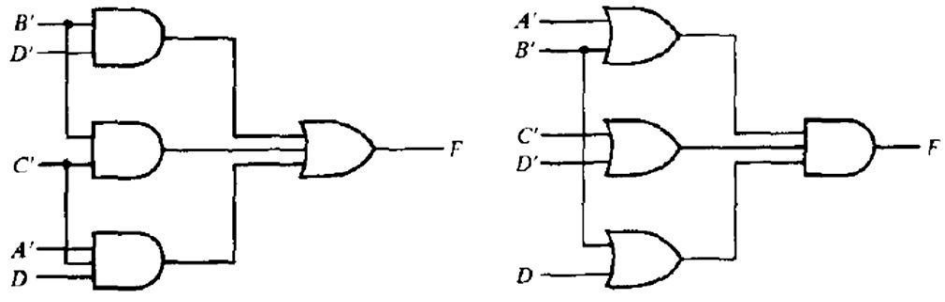
(b) If the squares marked with 0's are combined, as shown in the diagram, we obtain the simplified complemented function:

F' = AB + CD + BD'

Applying DeMorgan's theorem (by taking the dual and complementing each literal as described in unit2), we obtain the simplified function in product of sums:

F = (A' + B')(C' + D')(B' + D)

The Gate implementation of the simplified expressions obtained above in (a) and (b):



(a) $F = B'D' + B'C' + A'C'D$ (b) $F = (A' + B')(C' + D')(B' + D)$

NAND and NOR implementation

Digital circuits are more frequently constructed with NAND or NOR gates than with AND and OR gates. NAND and NOR gates are easier to fabricate with electronic components and are the basic gates used in all IC digital logic families. The procedure for **two-level implementation** is presented in this section.

NAND and NOR conversions (from AND, OR and NOT implemented Boolean functions)

Because of the prominence of NAND and NOR gates in the design of digital circuits, rules and procedures have been developed for the conversion from Boolean functions given in terms of AND, OR, and NOT into equivalent NAND and NOR logic diagrams.

To facilitate the conversion to NAND and NOR logic, there are two other graphic symbols for these gates.

(a) NAND gate

Two equivalent symbols for the NAND gate are shown in diagram below:

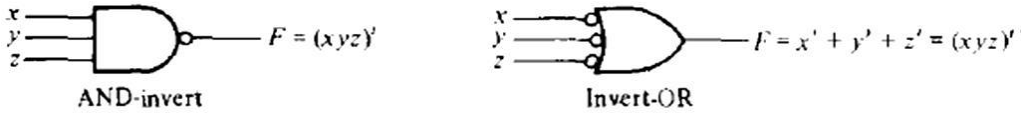


Fig: Two graphic symbols for NAND gate

(b) NOR gate

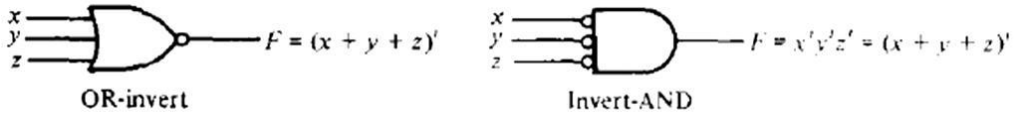


Fig: Two graphic symbols for NOR gate

(c) Inverter

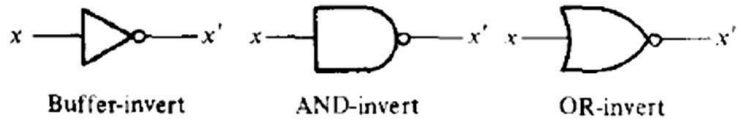
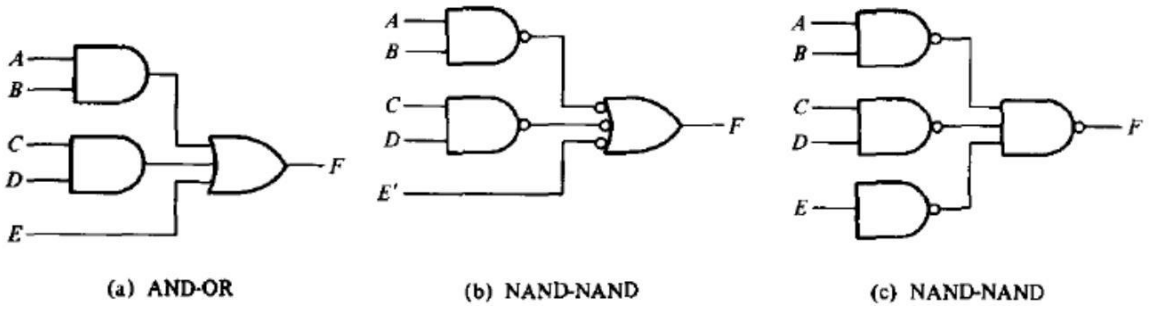


Fig: Three graphic symbols for NOT gate

NAND implementation

The implementation of a Boolean function with NAND gates requires that the function be simplified in the sum of products form. To see the relationship between a sum of products expression and its equivalent NAND implementation, consider the logic diagrams of Fig below. All three diagrams are equivalent and implement the function: *F=AB + CD + E*



The rule for obtaining the NAND logic diagram from a Boolean function is as follows:

First method:

(a) Simplify the function and express it in **sum of products**.

(b) Draw a NAND gate for each product term of the function that has at least two literals. The inputs to each NAND gate are the literals of the term. This constitutes a group of **first-level gates.**

(c) Draw a single NAND gate (using the AND-invert or invert-OR graphic symbol) in the second level, with inputs coming from outputs of first-level gates.

(d) A term with a single literal requires an inverter in the first level or may be complemented and applied as an input to the **second-level NAND gate**.

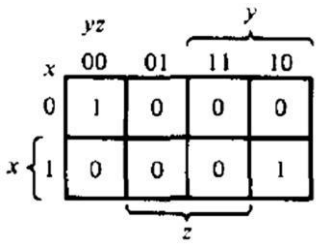
Second method:

If we combine the 0's in a map, we obtain the simplified expression of the *complement* of the function in sum of products. The complement of the function can then be implemented with two levels of NAND gates using the rules stated above. If the normal output is desired, it would be necessary to **insert a one-input NAND or inverter gate**. There are occasions where the designer may want to generate the complement of the function; so this second method may be preferable.

Question: Implement the following function with NAND gates:

Solution:

The first step is to simplify the function in sum of products form. This is attempted with the map. There are only two 1's in the map, and they can't be combined.



$F = x'y'z' + xyz'$

$F' = x'y + xy' + z$

Fig: Map simplification in SOP

METHOD1:

Two-level NAND implementation is shown below:

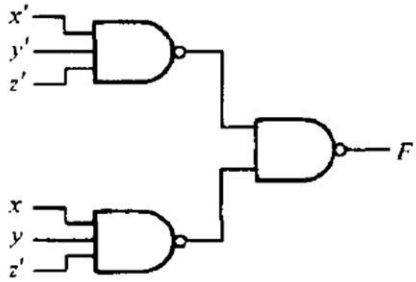


Fig: $F = x'y'z' + xyz'$

METHOD2:

Next we try to simplify the complement of the function in sum of products. This is done by combining the 0's in the map:

$F' = x'y + xy' + z$

The two-level NAND gate for generating F' is shown below:

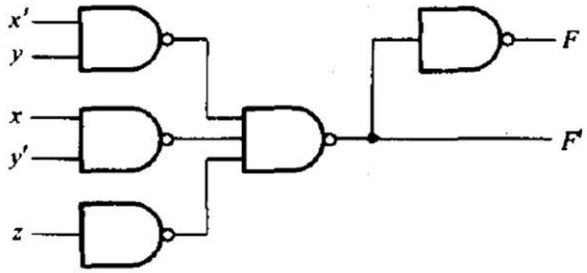


Fig: $F' = x'y + xy' + z$

If output F is required, it is necessary to add a one· input NAND gate to invert the function. This gives a three-level implementation.

NOR Implementation

The NOR function is the dual of the NAND function. For this reason, all procedures and rules for NOR logic are the duals of the corresponding procedures and rules developed for NAND logic. The implementation of a Boolean function with NOR gates requires that the function be simplified in product of sums form. A product of sums expression specifies a group of OR gates for the sum terms, followed by an AND gate to produce the product. The transformation from the OR-AND to the NOR-NOR diagram is depicted in Fig below. It is **similar to the NAND transformation discussed previously, except that now we use** the product of sums expression.

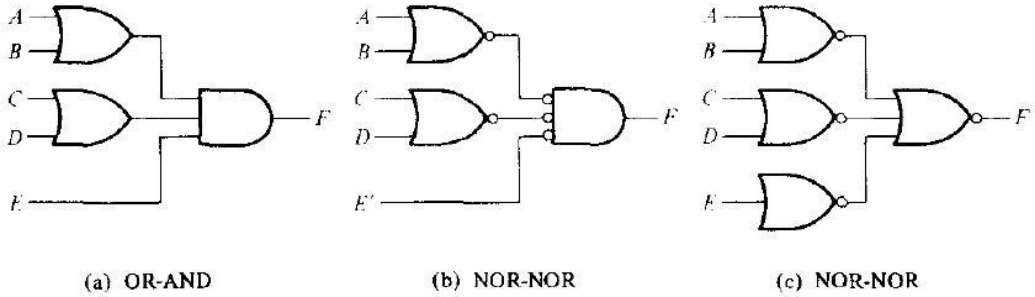


Fig: Three ways to implement $F = (A + B)(C + D)E$

All the rules for NOR implementation are similar to NAND except that these are duals, so I won't describe them here.

Question: Implement the following function with NOR gates:

Solution:

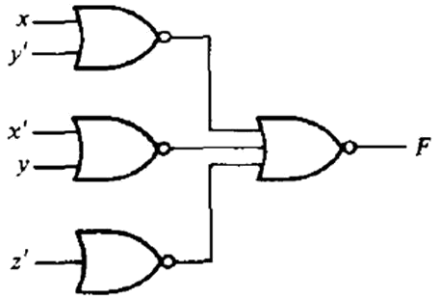
Map is drawn in previous question.

METHOD1

First, combine the 0's in the map to obtain

$F'=x'y+xy'+z$ this is the complement of the function in sum of products. Complement F' to obtain the simplified function in product of sums as required for NOR implementation:

$$F = (x + y') (x' + y) z'$$



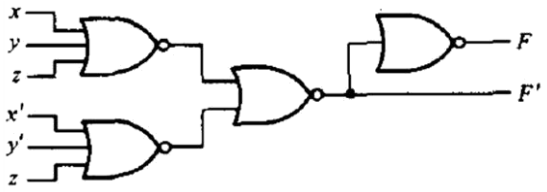
METHOD2

A second implementation is possible from the complement of the function in product of sums. For this case, first combine the 1's in the map to obtain

$$F = x'y'z' + xyz'$$

Complement this function to obtain the complement of the function in product of sums as required for NOR implementation:

$$F' = (x + y + z)(x' + y' + z)$$



Summary of NAND and NOR implementation

Case	Function to simplify	Standard form to use	How to derive	Implement with	Number of levels to F
(a)	F	Sum of products	Combine 1's in map	NAND	2
(b)	F'	Sum of products	Combine 0's in map	NAND	3
(c)	F	Product of sums	Complement F' in (b)	NOR	2
(d)	F'	Product of sums	Complement F in (a)	NOR	3

Unit 4
Combinational Logic

Introduction

In digital circuit theory, **combinational logic** is a *type of digital logic which is implemented by Boolean circuits, where the output is a pure function of the present input only*. This is in contrast to sequential logic, in which the output depends not only on the present input but also on the history of the input. In other words, sequential logic has *memory* while combinational logic does not.

Combinational Circuit

These are the circuit gates employing combinational logic.

- A combinational circuit consists of *n* input variables, logic gates, and *m* output variables. The logic gates accept signals from the inputs and generate signals to the outputs.
- For *n* input variables, there are 2^n possible combinations of binary input values. For each possible input combination, there is one and only one possible output combination. A combinational circuit can be described by *m* Boolean functions, one for each output variable. Each output function is expressed in terms of the *n* input variables.

Obviously, both input and output data are represented by binary signals, i.e., logic-1 and the other logic-0. The n input binary variables come from an external source; the m output variables go to an external destination. A block diagram of a combinational circuit is shown in Fig:

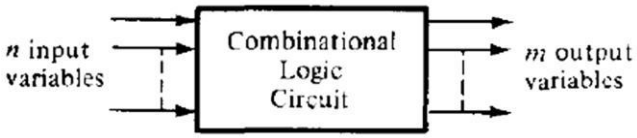


Fig: Block diagram of combinational circuit

Design procedure

The design of combinational circuits starts from the verbal outline of the problem and ends in a logic circuit diagram or a set of Boolean functions from which the logic diagram can be easily obtained. The procedure involves the following steps:

1. **Specification**
   - Write a specification for the circuit if one is not already available
2. **Formulation**
   - Derive a truth table or initial Boolean equations that define the required relationships between the inputs and outputs, if not in the specification.
   - Apply hierarchical design if appropriate
3. **Optimization**
   - Apply 2-level and multiple-level optimization
   - Draw a logic diagram for the resulting circuit using ANDs, ORs, and inverters
4. **Technology Mapping**
   - Map the logic diagram to the implementation technology selected
5. **Verification**
   - Verify the correctness of the final design manually or using simulation

In simple words, we can list out the design procedure of combinational circuits as:

1. The problem is stated.
2. The number of available input variables and required output variables is determined.
3. The input and output variables are assigned letter symbols.
4. The truth table that defines the required relationships between inputs and outputs is derived.
5. The simplified Boolean function for each output is obtained.
6. The logic diagram is drawn.

Adders

Digital computers perform a variety of information-processing tasks. Among the basic functions encountered are the various arithmetic operations. The most basic arithmetic operation, no doubt, is the addition of two binary digits.

Half-Adder

- A combinational circuit that performs the addition of two bits is called a *half-adder.*
- Circuit needs **two inputs** and **two outputs**. The input variables designate the augend (x) and addend (y) bits; the output variables produce the sum (S) and carry (C).
- Now we **formulate a Truth table** to exactly identify the function of half-adder.

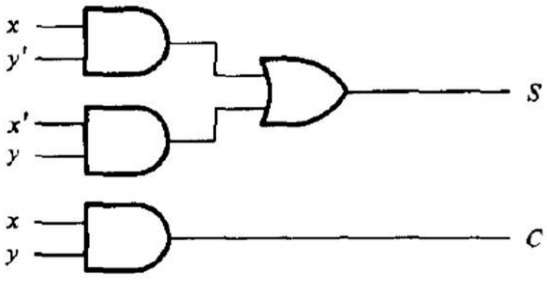
x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

The simplified Boolean functions for the two outputs can be obtained directly from the truth table. The simplified sum of products expressions are:

$$S = x'y + xy'$$

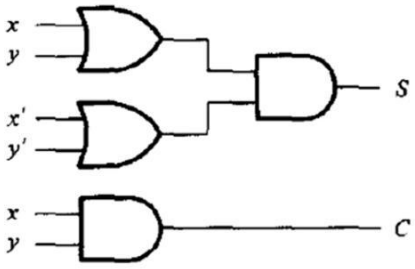
$$C = xy$$

- **Implementation:**

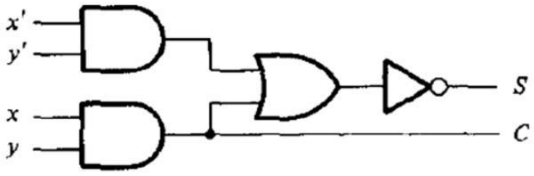


(a) $S = xy' + x'y$
$C = xy$

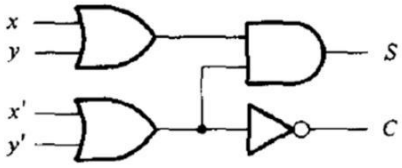
- Other realizations and implementations of Half-adders are:



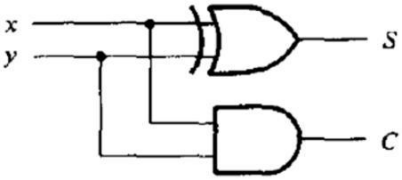
(b) $S = (x + y)(x' + y')$
$C = xy$



(c) $S = (C + x'y')'$
$C = xy$



(d) $S = (x + y)(x' + y')$
$C = (x' + y')'$



(e) $S = x \oplus y$
$C = xy$

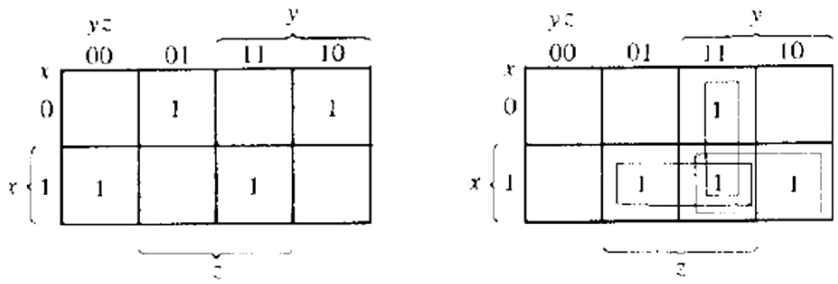
Full-Adder

- A *full-adder* is a combinational circuit that forms the arithmetic sum of three input bits.
- It consists of **three inputs** and **two outputs**. Two of the input variables, denoted by *x* and *y,* represent the **two significant bits** to be added. The third input, z, represents the **carry** from the previous lower significant position.
- **Truth table formulation:**

x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

The S output is equal to 1 when only one input is equal to 1 or when all three inputs are equal to 1. The C output has a carry of 1 if two or three inputs are equal to 1.

The input-output logical relationship of the full-adder circuit may be expressed in two Boolean functions, one for each output variable. Each output Boolean function requires a unique map for its simplification (maps are not necessary; you guys can use algebraic method for simplification). Simplified expression in sum of products can be obtained as:



S = x'y'z + x'yz' + xy'z' + xyz C = xy + xz + yz

- **Implementation:**

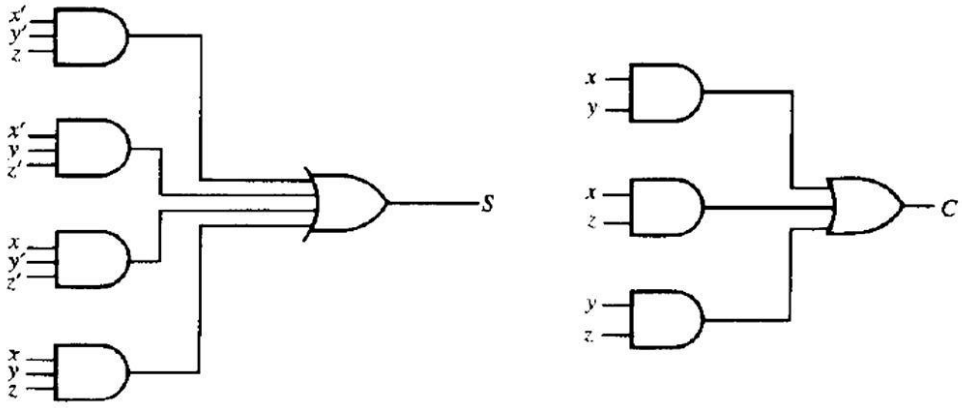


Fig: Implementation of a full-adder in sum of products.

- A full-adder can be implemented with **two half-adders** and one OR gate.

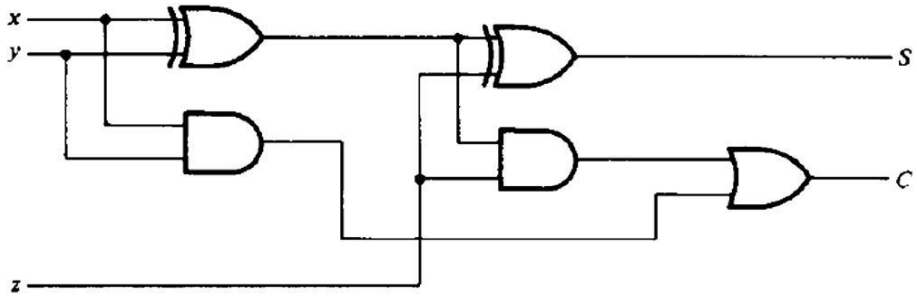


Fig: Implementation of a full-adder with two half-adders and an OR gate

Here, The S output from the second half-adder is the exclusive-OR of z and the output of the first half-adder, giving:

$S = z \oplus (x \oplus y)$

= z'(xy' + x'y) + z(xy' + x'y)'

= z'(xy' + x'y) + z(xy + x'y')

= xy'z' + x'yz' + xyz + x'y'z

$C = z(x \oplus y) + xy$

= z(xy' + x'y) + xy

= xy'z + x'yz + xy

Subtractors

The subtraction of two binary numbers may be accomplished by taking the complement of the subtrahend and adding it to the minuend. By this method, the subtraction operation becomes an

addition operation requiring full-adders for its machine implementation. It is possible to implement subtraction with logic circuits in a direct manner, as done with paper and pencil. By this method, each subtrahend bit of the number is subtracted from its corresponding **significant minuend bit** to form a **difference bit**. If the minuend bit is smaller than the subtrahend bit, a 1 is borrowed from the next significant position. Just as there are half- and full-adders, there are half- and full-subtractors.

Half-Subtractor

- A half-subtractor is a combinational circuit that subtracts two bits and produces their difference bit.
- Denoting minuend bit by x and the subtrahend bit by *y*. To perform *x - y,* we have to check the relative magnitudes of *x* and *y*:
  - If *x≥ y,* we have three possibilities: 0 - 0 = 0, 1 - 0 = 1, and 1 - 1 = 0.
  - If *x < y,* we have 0 - 1, and it is necessary to borrow a 1 from the next higher stage.
- The half-subtractor needs **two outputs**, difference (D) and borrow (B).
- The **truth table** for the input-output relationships of a half-subtractor can now be derived as follows:

x	*y*	*B*	*D*
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0

The output borrow *B* is a 0 as long as *x≥ y.* It is a 1 for *x* = 0 and *y* = 1. The *D* output is the result of the arithmetic operation *2B + x - y.*

The Boolean functions for the two outputs of the half-subtractor are derived directly from the truth table:

D = x'y + xy'

B = x'y

- **Implementation** for Half-subtractor is similar to Half-adder except the fact that x input of B is inverted. (Here, D is analogous to S and B is similar to C of half-adder circuit). Try it out, I don't like redundancy… ☺

Full-Subtractor

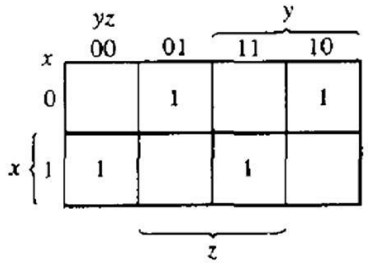
- A full-subtractor is a combinational circuit that performs a subtraction between two bits, taking into account that a 1 **may have been borrowed** by a lower significant stage.
- This circuit has **three inputs** and **two outputs**. The three inputs, *x, y,* and z, denote the minuend, subtrahend, and previous borrow, respectively. The two outputs, *D* and *B,* represent the difference and output-borrow, respectively.

- **Truth-table and output-function formulation**:

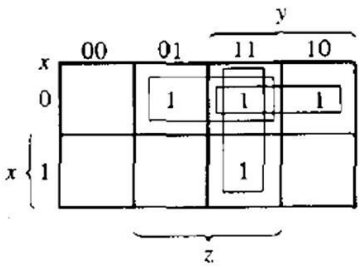
x	y	z	B	D
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

- The 1's and 0's for the output variables are determined from the subtraction of *x* - *y* - z.
- The combinations having input borrow *z* = 0 reduce to the same four conditions of the half-adder.
- For *x* = 0, *y* = 0, and *z* = 1, we have to borrow a 1 from the next stage, which makes *B* = 1 and adds 2 to *x.* Since 2 - 0 - 1 = 1, *D* = 1.
- For *x* = 0 and *yz* = 11, we need to borrow again, making *B* = 1 and *x* = 2. Since 2 - 1 - 1 = 0, *D* = 0.
- For *x* = 1 and *yz* = 01, we have *x* - *y* - *z* = 0, which makes *B* = 0 and *D* = 0.
- Finally, for *x* = 1, *y* = 1, *z* =1l, we have to borrow 1, making *B* = 1 and *x* = 3, and 3 - 1 - 1 = 1, making *D=1.*

The simplified Boolean functions for the two outputs of the full-subtractor are derived in the maps:



D = x'y'z + x'yz' + xy'z' + xyz



B = x'y + x'z + yz

- Circuit implementations are same as Full-adder except B output (analogous to C) is little different. (Don't worry! ladies and gentlemen, we will discuss it in class…)

Code Conversion

- The availability of a large variety of codes for the same discrete elements of information results in the use of different codes by different digital systems. It is sometimes necessary to use the output of one system as the input to another. A conversion circuit must be inserted between the two systems if each uses different codes for the same information. Thus, a **code converter** is a circuit that makes the two systems compatible even though each uses a different binary code.
- To convert from binary code A to binary code B, code converter has **input lines** supplying the bit combination of elements as specified by code A and the output lines of the converter generating the corresponding bit combination of code B. A Code converter (combinational circuit) performs this transformation by means of logic gates.
- The design procedure of code converters will be illustrated by means of a *specific example* of conversion from the BCD to the excess-3 code. I will describe 5-step design procedure of this code converter so that you guys will be able to understand how practical combinational circuits are designed.

Design example: BCD to Excess-3 code converter

1. **Specification**
   - Transforms BCD code for the decimal digits to Excess-3 code for the decimal digits
   - BCD code words for digits 0 through 9: 4-bit patterns 0000 to 1001, respectively.
   - Excess-3 code words for digits 0 through 9: 4-bit patterns consisting of 3 (binary 0011) added to each BCD code word
   - Implementation:
     - multiple-level circuit
2. **Formulation**
   - Conversion of 4-bit codes can be most easily formulated by a truth table
   - Variables- BCD: A, B, C, D
   - Variables- Excess-3: W, X, Y, Z
   - Don't Cares: BCD 1010 to 1111

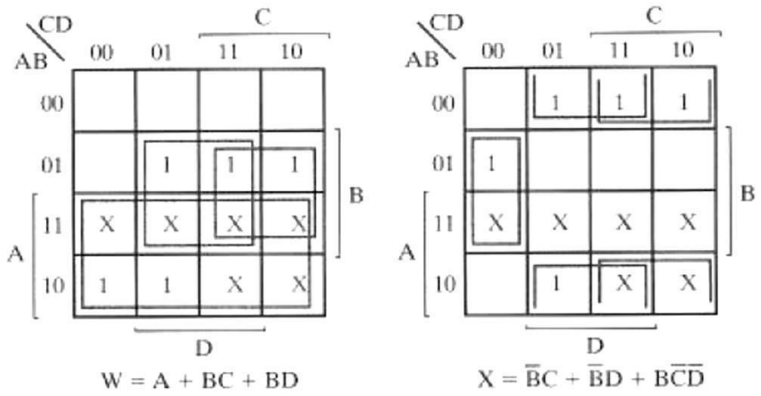
Decimal Digit	Input BCD				Output Excess-3			
	A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	0	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	1	0
8	1	0	0	0	1	0	1	1
9	1	0	0	1	1	1	0	0

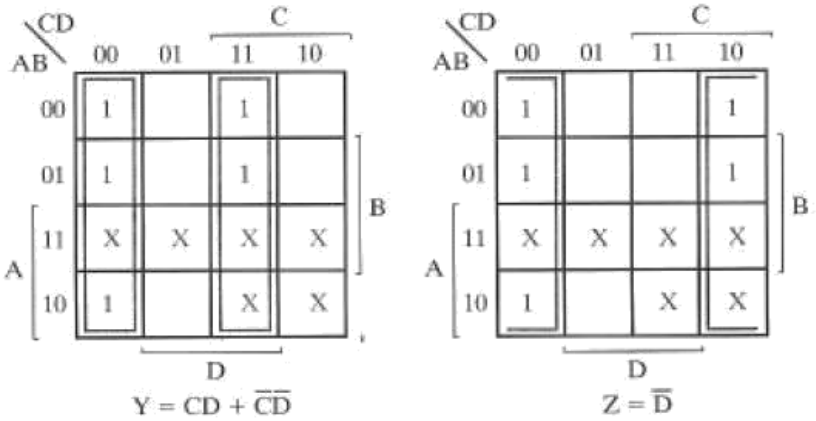
Note that the four BCD input variables may have 16 bit combinations, but only 10 are listed in the truth table. Others designate "don't care conditions".

Table: Truth table for code converter example

3. **Optimization**
   a. *2-level optimization*

   The k-maps are plotted to obtain simplified sum-of-products Boolean expressions for the outputs. Each of the four maps represents one of the outputs of the circuit as a function of the four inputs.





b. Multiple-level optimization

This second optimization step reduces the number of gate inputs (and hence the no. gates). The following manipulation illustrates optimization with multiple-output circuits implemented with three levels of gates:

$$T_1 = C + D$$

$$W = A + BC + BD = A + BT_1$$

$$X = \overline{B}C + \overline{B}D + B\overline{C}\,\overline{D} = \overline{B}T_1 + B\overline{C}\,\overline{D}$$

$$Y = CD + \overline{C}\,\overline{D}$$

$$Z = \overline{D}$$

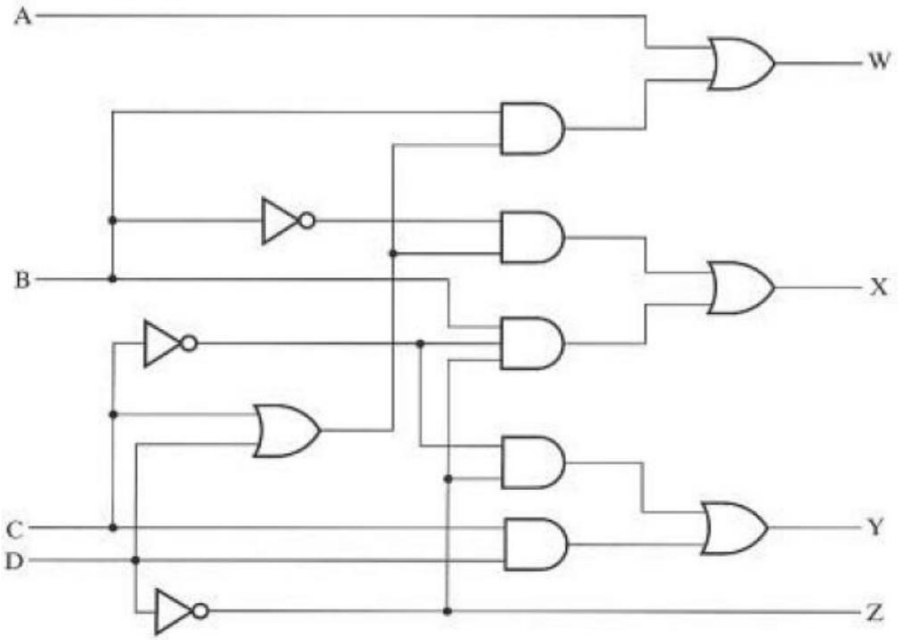


Fig: Logic Diagram of BCD- to-Excess-3 Code Converter

4. Technology mapping

This is concerned with the act of mapping of basic circuit (using AND, OR and NOT gates) to a specific circuit technology (such as NAND, NOR gate tech.).

HEY! This is advanced topic, and I won't discuss here. For exam point of view, if you are asked for BCD-to-Excess-3 code converter, you will finish up your answer by drawing basic circuit shown above.

5. **Verification**

   Here we need to test our designed circuit, whether it works correctly. You guys are so keen; hope you can do it… ☺.

Analysis Procedure

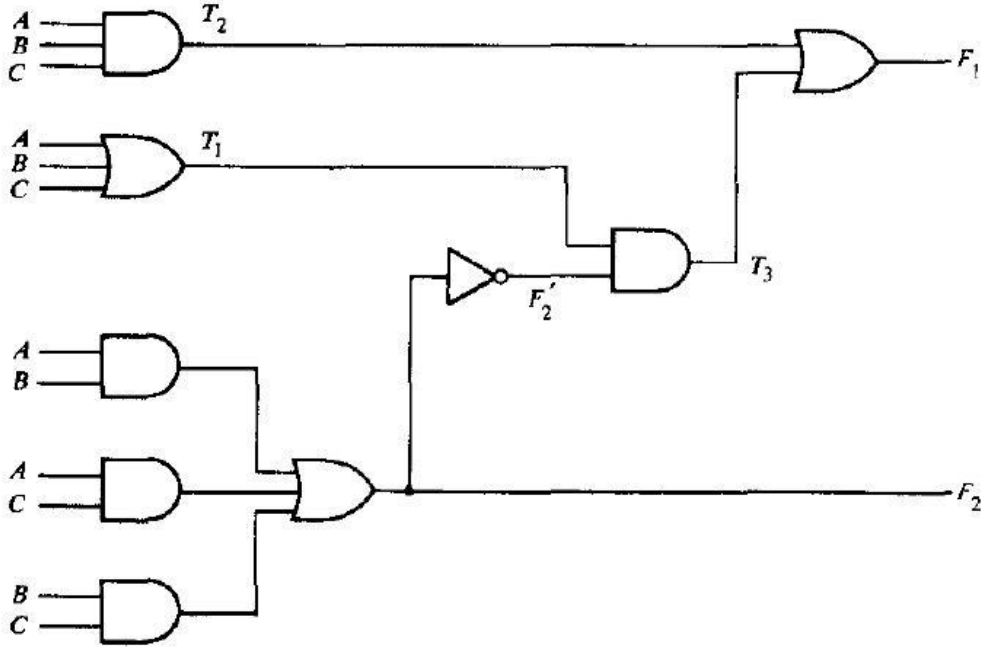
The design of a combinational circuit starts from the verbal specifications of a required function and ends with a set of output Boolean functions or a logic diagram. The **analysis of a combinational circuit** is somewhat the reverse process. It starts with a given logic diagram and culminates with a set of Boolean functions, a truth table, or a verbal explanation of the circuit operation.

Obtaining Boolean functions from logic diagram

Steps in analysis:

1. The first step in the analysis is to make sure that the given circuit is combinational and not sequential.
2. Assign symbols to all gate outputs that are a function of the input variables. Obtain the Boolean functions for each gate.
3. Label with other arbitrary symbols those gates that are a function of input variables and/or previously labeled gates. Find the Boolean functions for these gates.
4. Repeat step 3 until the outputs of the circuit are obtained.
5. By repeated substitution of previously defined functions, obtain the output Boolean functions in terms of input variables only.

Analysis of the combinational circuit below illustrates the proposed procedure:



We note that the circuit has three binary inputs, *A, B,* and C, and two binary outputs, F_1 and F_2. The outputs of various gates are labeled with intermediate symbols. The outputs of gates that are a function of input variables only are F_2, T_1 *and* T_2. The Boolean functions for these three outputs are

$$F_2 = AB + AC + BC$$
$$T_1 = A+B+C$$
$$T_2 = ABC$$

Next we consider outputs of gates that are a function of already defined symbols:

$$T_3 = F_2'T_1$$
$$F_1 = T_3 + T_2$$

The output Boolean function F_2 just expressed is already given as a function of the in puts only. To obtain F_1 as a function of *A, B,* and C, form a series of substitutions as follows:

$$\begin{aligned} F_1 = T_3 + T_2 &= F_2'T_1 + ABC = (AB + AC + BC)'(A + B + C) + ABC \\ &= (A' + B')(A' + C')(B' + C')(A + B + C) + ABC \\ &= (A' + B'C')(AB' + AC' + BC' + B'C) + ABC \\ &= A'BC' + A'B'C + AB'C' + ABC \end{aligned}$$

If you want to determine the information-transformation task achieved by this circuit, you can derive the truth table directly from the Boolean functions and try to recognize a familiar operation. For this example, we note that the circuit is a **full-adder**, with *F,* being the sum output and *F,* the carry output. *A, B,* and C are the three inputs added arithmetically.

Obtaining truth-table from logic diagram

The derivation of the truth table for the circuit is a straightforward process once the output Boolean functions are known. To obtain the truth table directly from the logic diagram without going through the derivations of the Boolean functions, proceed as follows:

Steps in analysis:

1. Determine the number of input variables to the circuit. For *n* inputs, form the 2^n possible input combinations of 1's and 0's by listing the binary numbers from 0 to $2^n - 1$.
2. Label the outputs of selected gates with arbitrary symbols.
3. Obtain the truth table for the outputs of those gates that are a function of the input variables only.
4. Proceed to obtain the truth table for the outputs of those gates that are a function of previously defined values until the columns for all outputs are determined.

This process can be illustrated using the circuit above:

We form the eight possible combinations for the three input variables. The truth table for F_2 is determined directly from the values of A, B, and C, with F_2 equal to 1 for any combination that has two or three inputs equal to 1. The truth table for F_2' is the complement of F_2. The truth tables for T_1 and T_2 are the OR and AND functions of the input variables, respectively. The values for T_3 are derived from T_1 and F_2'. T_3 is equal to 1 when both T_1 and F_2' are equal to 1, and to 0 otherwise. Finally, F_1 is equal to 1 for those combinations in which either T_2 or T_3 or both are equal to 1.

A	B	C	F_2	F_2'	T_1	T_2	T_3	F_1
0	0	0	0	1	0	0	0	0
0	0	1	0	1	1	0	1	1
0	1	0	0	1	1	0	1	1
0	1	1	1	0	1	0	0	0
1	0	0	0	1	1	0	1	1
1	0	1	1	0	1	0	0	0
1	1	0	1	0	1	0	0	0
1	1	1	1	0	1	1	0	1

Inspection of the truth-table combinations for *A, B,* C, F_1 and F_2 of table above shows that it is identical to the truth-table of the full-adder.

HEY! When a circuit with don't-care combinations is being analyzed, the situation is entirely different. We assume here that the don't-care input combinations will never occur.

NAND, NOR and Ex-OR circuits

In unit 3, SOP and POS form of Boolean functions are studied. Also we got to know, Such Boolean functions can be implemented with 2-level circuits using universal gates (look at NAND and NOR implementation of Boolean function, unit 3). Here we will look at the multiple level circuits employing universal gates i.e we will treat the functions which are in standard form.

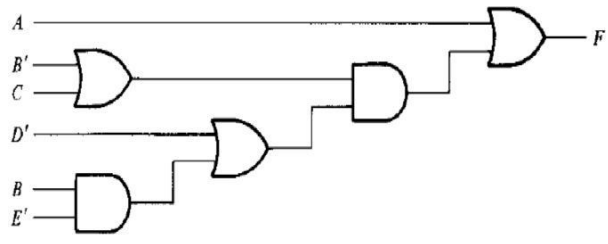
Multi-level NAND circuits

To implement a Boolean function with NAND gates we need to obtain the simplified Boolean function in terms of Boolean operators and then convert the function to NAND logic. The conversion of an algebraic expression from AND, OR, and complement to NAND can be done by simple circuit-manipulation techniques that change AND-OR diagrams to NAND diagrams.

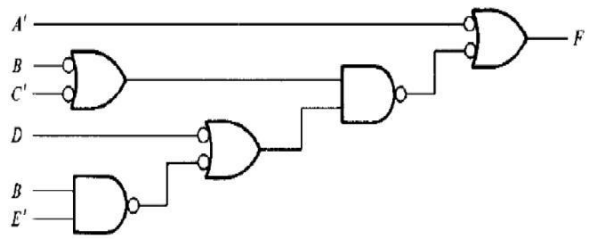
To obtain a multilevel NAND diagram from a Boolean expression, proceed as follows:

1. From the given Boolean expression, draw the logic diagram with AND, OR, and inverter gates. Assume that both the normal and complement inputs are available.
2. Convert all AND gates to NAND gates with AND-invert graphic symbols.
3. Convert all OR gates to NAND gates with invert-OR graphic symbols.
4. Check all small circles in the diagram. For every small circle that is not compensated by another small circle along the same line, insert an inverter (one-input NAND gate) or complement the input variable.

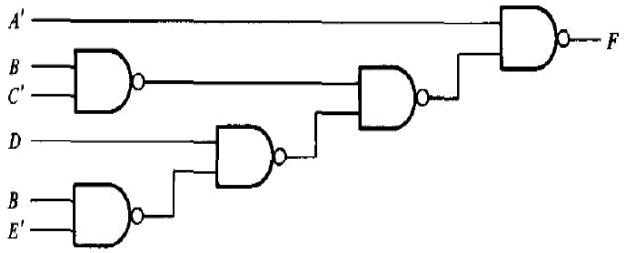
Example: F = A + (B' + C) (D' + BE ')



AND-OR diagram



NAND diagram using two graphic symbols



NAND diagram using one graphic symbol

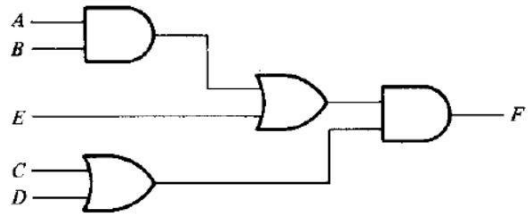
Multi-level NOR circuits

The NOR function is the dual of the NAND function. For this reason, all procedures and rules for NOR logic form a dual of the corresponding procedures and rules developed for NAND logic. Similar to NAND, NOR has also two graphic symbols: OR-invert and invert-AND symbol.

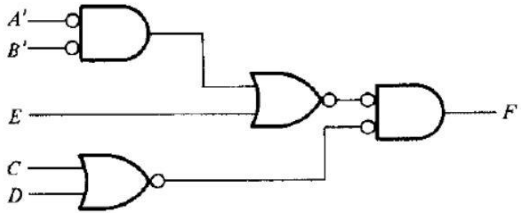
The procedure for implementing a Boolean function with NOR gates is similar to the procedure outlined in the previous section for NAND gates:

1. Draw the AND-OR logic diagram from the given algebraic expression. Assume that both the normal and complement inputs are available.
2. Convert all OR gates to NOR gates with OR-invert graphic symbols.
3. Convert all AND gates to NOR gates with invert-AND graphic symbols.
4. Any small circle that is not compensated by another small circle along the same line needs an inverter or the complementation of the input variable.

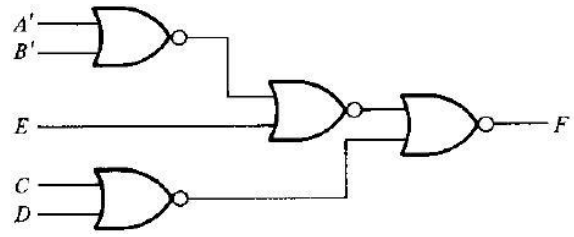
Example: F = (AB + E) (C + D)



AND-OR diagram



NOR diagram



Alternate NOR diagram

Ex-OR function

The exclusive-OR (XOR) denoted by the symbol ⊕, is a logical operation that performs the following Boolean operation:

$x \oplus y = xy' + x'y$

It is equal to 1 if only *x* is equal to 1 or if only y is equal to 1 but not when both are equal.

Realization of XOR using Basic gates and universal gates

A two-input exclusive-OR function is constructed with conventional gates using two inverters, two AND gates, and an OR gate and next firgure shows the implementation of the exclusive-OR with four NAND gates.

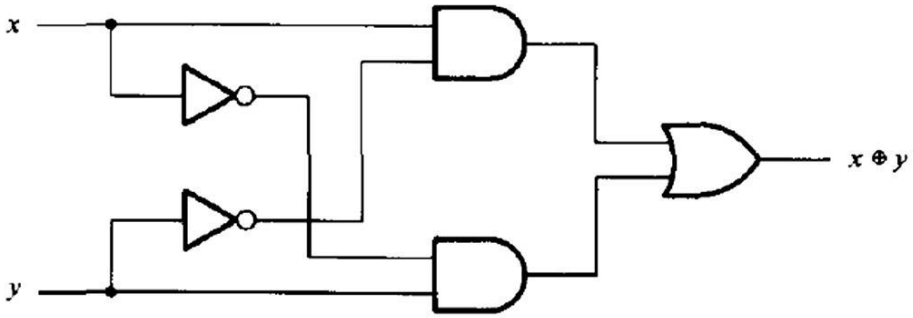


Fig: Implementation XOR with AND-OR-NOT gates

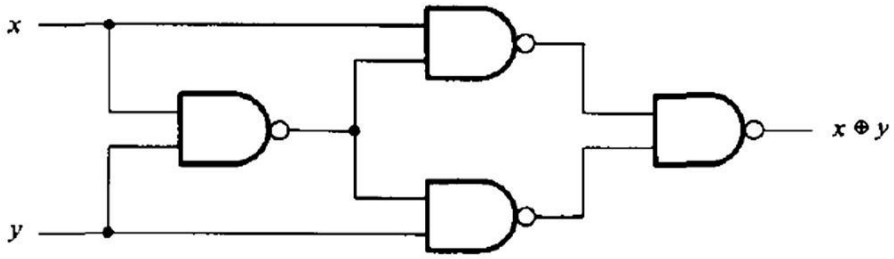


Fig: Realization of XOR with NAND gates

In second diagram, first NAND gate performs the operation (xy)' = (x' + y'). The other two-level NAND circuit produces the sum of products of its inputs:

$(x' + y')x + (x' + y')y = xy' + xy = x \oplus y$

Only a limited number of Boolean functions can be expressed in terms of exclusive-OR operations. Nevertheless, this function emerges quite often during the design of digital systems. It is particularly useful in arithmetic operations and error-detection and correction circuits.

Parity generator and Checker

Exclusive-OR functions are very useful in systems requiring error-detection and correction codes. As discussed before, a **parity bit** is used for the purpose of detecting errors during transmission of binary information. A parity bit is an extra bit included with a binary message to make the number of 1's either odd or even. The message, including the parity bit, is transmitted and then checked at the receiving end for errors. An error is detected if the checked parity does not correspond with the one transmitted.

- The circuit that generates the parity bit in the transmitter is called a ***parity generator***.

- The circuit that checks the parity in the receiver is called a ***parity checker***.

Example: Consider a 3-bit message to be transmitted together with an even parity bit.

The three bits, *x, y,* and z, constitute the message and are the inputs to the circuit. The parity bit *P* is the output. For even parity, the bit *P* must be generated to make the total number of 1's even (including *P).*

Three-Bit Message			Parity Bit
x	*y*	*z*	*P*
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

From the truth table, we see that *P* constitutes an odd function because it is equal to 1 for those minterms whose numerical values have an odd number of 1's. Therefore, *P* can be expressed as a three-variable exclusive-OR function: P = x ⊕ y ⊕ z.

Table: Even parity generator truth table

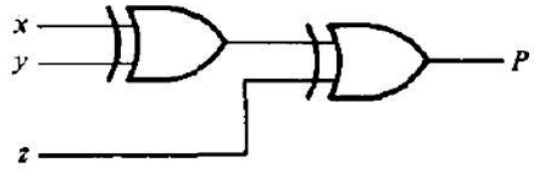
The three bits in the message together with the parity bit are transmitted to their destination, where they are applied to a parity-checker circuit to check for possible errors in the transmission.

Four Bits Received				Parity Error Check
x	*y*	*z*	*P*	*C*
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

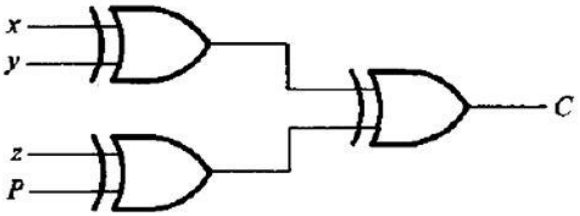
- Since the information was transmitted with even parity, the four bits received must have an even number of 1's. An error occurs during the transmission if the four bits received have an odd number of 1's, indicating that one bit has changed in value during transmission.
- The output of the parity checker, denoted by C, will be equal to 1 if an error occurs, that is, if the four bits received have an odd number of 1's.
- The parity checker can be implemented with exclusive-OR gates: C = x ⊕ y ⊕ z ⊕ P.

Table: Even parity checker truth table

Logic diagrams for parity generator and Parity checker are shown below:



(a) 3-bit even parity generator



(b) 4-bit even parity checker

Unit 5
Combinational Logic with MSI and LSI

(I have included **Point!** 's throughout the chapter to designate extra things you need to know about the concerned topic and are unimportant for the exam point of view.)

Introduction

The purpose of Boolean-algebra simplification is to obtain an algebraic expression that, when implemented, results in a low-cost circuit. Two circuits that perform the same function, the one that requires fewer gates is preferable because it will cost less. But this is not necessarily true when integrated circuits are used. With integrated circuits, it is not the count of gates that determines the cost, but the number and types of ICs employed and the number of interconnections needed to implement the digital circuits of varying complexities (I mean circuits with different level of integrations viz. SSI, MSI, LSI, VLSI, ULSI etc).

There are several combinational circuits that are employed extensively in the design of digital systems. These circuits are available in integrated circuits and are classified as **MSI components**. MSI components perform specific digital functions commonly needed in the design of digital systems.

Combinational circuit-type MSI components that are readily available in IC packages are binary adders, subtractors, comparators, decoders, encoders, and multiplexers. These components are also used as standard modules within more complex LSI and VLSI circuits and hence used extensively as basic building blocks in the design of digital computers and systems.

Binary Adder

This circuit sums up two binary numbers A and B of n-bits using full-adders to add each bit-pair & carry from previous bit position. The sum of *A* and *B* can be generated in two ways: either in a **serial fashion** or **in parallel**.

- The *serial addition method* uses only one full-adder circuit and a storage device to hold the generated output carry. The pair of bits in *A* and *B* are transferred serially, one at a time, through the single full-adder to produce a string of output bits for the sum. The stored output carry from one pair of bits is used as an input carry for the next pair of bits.
- The *parallel method* uses *n* full-adder circuits, and all bits of *A* and *B* are applied simultaneously. The outputs carry from one full-adder is connected to the input carry of the full-adder one position to its left. As soon as the carries are generated, the correct sum bits emerge from the sum outputs of all full-adders.

Binary Parallel adder

A *binary parallel adder* is a digital circuit that produces the arithmetic sum of two binary numbers in parallel. It consists of full-adders connected in a chain, with the output carry from each full-adder connected to the input carry of the next full-adder in the chain.

Diagram below shows the interconnection of four full-adder (FA) circuits to provide a 4-bit binary parallel adder. The **augend bits of *A*** and the **addend bits of *B*** are designated by subscript numbers from right to left. The carries are connected in a chain through the full-adders. The S outputs generate the required sum bits. The input carry to the adder is C_1 and the output carry is C_5.

When the 4-bit full-adder circuit is **enclosed within an IC package**, it has four terminals for the ***augend bits***, four terminals for the ***addend bits***, four terminals for the ***sum bits***, and two terminals for the ***input and output carries***.

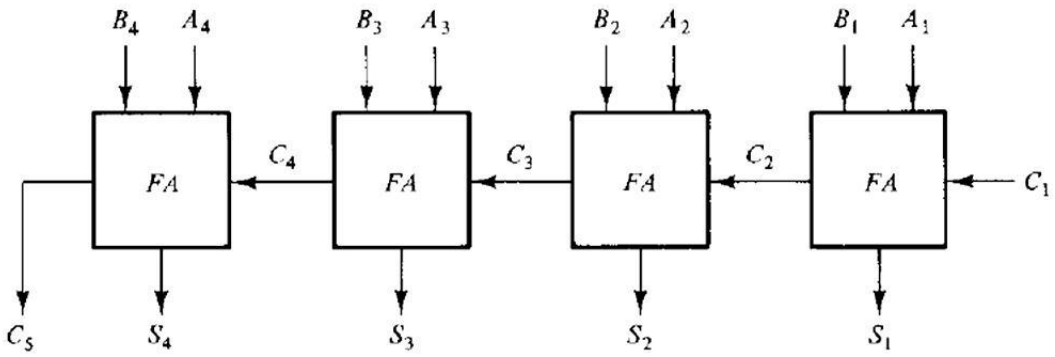


Fig: 4-bit parallel binary adder

Point-1! (To be noted my Boys… ☺)

The 4-bit binary-adder is a typical example of an MSI function. It can be used in many applications involving arithmetic operations. Observe that the design of this circuit by the **classical method** would require a truth table with 2^9 = 512 entries, since there are 9 inputs to the circuit. By using an iterative method of cascading an already known function, we were able to obtain a simple and well-organized implementation.

Point-2!

The **carry propagation time** is a limiting factor on the speed with which two numbers are added in parallel. Although a parallel adder, or any combinational circuit, will always have some value at its output terminals, the outputs will not be correct unless the signals are given enough time to propagate through the gates connected from the inputs to the outputs. Since all other arithmetic operations are implemented by successive additions, the time consumed during the addition process is very critical. An obvious solution for reducing the carry propagation delay time is to **employ faster gates** with reduced delays. But physical circuits have a limit to their capability. Another solution is to **increase the equipment complexity in such a way that the carry delay time is reduced**. There are several techniques for reducing the carry propagation time in a parallel adder. The most widely used technique employs the principle of *look-ahead* carry.

Decimal Adder

Computers or calculators that perform arithmetic operations directly in the decimal number system represent decimal numbers in binary-coded form.

- **Decimal adder** is a combinational circuit that sums up two decimal numbers adopting particular encoding technique.
- A decimal adder requires a *minimum of* nine inputs and five outputs, since four bits are required to code each decimal digit and the circuit must have an input carry and output carry.
- Of course, there is a wide variety of possible decimal adder circuits, dependent upon the code used to represent the decimal digits.

BCD Adder

This combinational circuit adds up two decimal numbers when they are encoded with binary-coded decimal (BCD) form.

- Adding two decimal digits in BCD, together with a possible carry, the output sum cannot be greater than 9 + 9 + 1 = 19.
- Applying two BCD digits to a 4-bit binary adder, the adder will form the sum in *binary* ranging from 0 to 19. These binary numbers are listed in Table below and are labeled by symbols K, Z_8, Z_4, Z_2, Z_1. K is the carry, and the subscripts under the letter z represent the weights 8, 4, 2, and 1 that can be assigned to the four bits in the BCD code.

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

- The first column in the table lists the binary sums as they appear in the outputs of a 4-bit *binary* adder.
- The output sum of two *decimal digits* must be represented in BCD and should appear in the form listed in the second column.
- **The problem** is to find a simple rule by which the binary number, in the first column can be converted to the correct BCD-digit representation of the number in the second column.

Looking at the table, we see that:

When (binary sum) <= 1001

Corresponding BCD number is identical, and therefore no conversion is needed.

When (binary sum) > 1001

Non-valid BCD representation is obtained. The addition of binary 6 (0110) to the binary sum converts it to the correct BCD representation and also produces an output carry as required.

The logic circuit that detects the necessary correction can be derived from the table entries.

- **Correction** is needed when
  - The binary sum has an output carry $K = 1$.
  - The other six combinations from 1010 to 1111 that have Z_8=1. To distinguish them from binary 1000 and 1001, which also have a 1 in position Z_8, we specify further that either Z_4 or Z_2 must have a 1. The condition for a correction and an output carry can be expressed by the Boolean function

$$C = K + Z_8Z_4 + Z_8Z_2$$

- When C = 1, it is necessary to add 0110 to the binary sum and provide an output carry for the next stage.

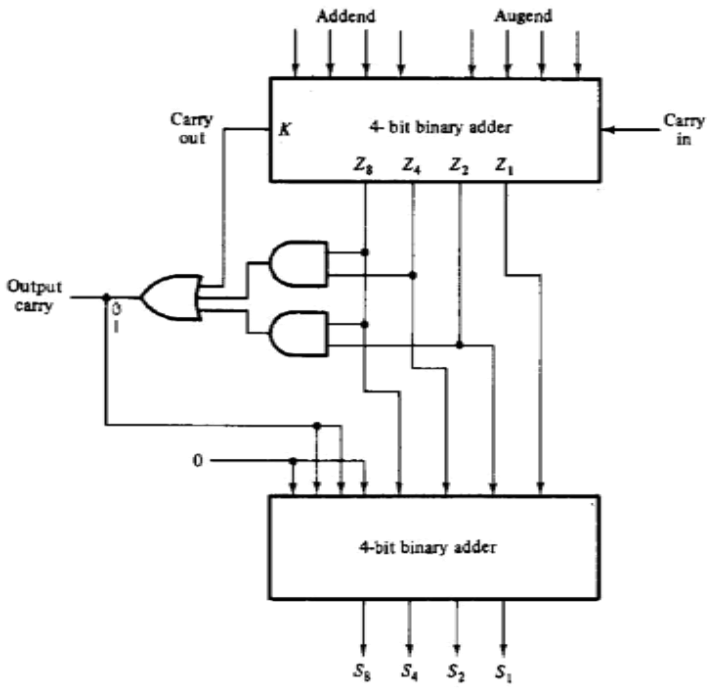


Fig: Block diagram of BCD adder

- A *BCD adder* is a circuit that adds two BCD digits in parallel and produces a sum digit also in BCD.
- BCD adder must include the **correction logic** in its internal construction.
- To add 0110 to the binary sum, we use a second 4-bit binary adder, as shown in diagram.
- The two decimal digits, together with the input carry, are first added in the top 4-bit binary adder to produce the binary sum.
- When the output carry = 0, nothing is added to the binary sum. When it is equal to 1, binary 0110 is added to the binary sum through the bottom 4-bit binary adder.
- The output carry generated from the bottom binary adder can be ignored, since it supplies information already available at the output-carry terminal.

A decimal parallel adder that adds *n* decimal digits needs *n* BCD adder stages. The output carry from one stage must be connected to the input carry of the next higher order stage.

Point!

The BCD adder can be constructed with three IC packages. Each of the 4-bit adders is an MST function and the three gates for the correction logic need one SST package. However, the BCD adder is available in one MSI circuit. To achieve shorter propagation delays, an MSI BCD adder includes the necessary circuits for look-ahead carries. The adder circuit for the correction does not need all four full-adders, and this circuit can be optimized within the IC package.

Magnitude Comparator

A *Magnitude comparator* is a combinational circuit that compares two numbers, *A* and *B,* and determines their relative magnitudes. The outcome of the comparison is specified by three binary variables that indicate whether $A > B$, $A = B$, or $A < B$.

Consider two numbers, *A* and *B,* with four digits each. Write the coefficients of the numbers with descending significance as follows:

$$A = A_3A_2A_1A_0$$
$$B = B_3B_2B_1B_0$$

Where each subscripted letter represents one of the digits in the number, the two numbers are equal if all pairs of significant digits are equal, i.e., if $A_3 = B_3$ and $A_2 = B_2$ and $A_1 = B_1$ and $A_0 = B_0$.

When the numbers are binary, the digits are either 1 or 0 and the equality relation of each pair of bits can be expressed logically with an equivalence function:

$$\mathbf{X_i = A_iB_i + A_i'B_i', \ i = 0, 1, 2, 3}$$

Where $X_i = 1$ only if the pair of bits in position i are equal, i.e., if both are 1's or both are 0's.

Algorithm

(A = B)

For the equality condition to exist, all X_i variables must be equal to 1. This dictates an AND operation of all variables:

$$(A = B) = X_3X_2X_1X_0$$

The *binary* variable ***(A = B)*** is equal to 1 only if all pairs of digits of the two numbers are equal.

(A<B) or (A>B)

To determine if *A* is greater than or less than *B,* we check the relative magnitudes of pairs of significant digits starting from the most significant position. If the two digits are equal, we compare the next lower significant pair of digits. This **comparison continues until a pair of unequal digits is reached**.

A > B: If the corresponding digit of A is 1 and that of B is 0.

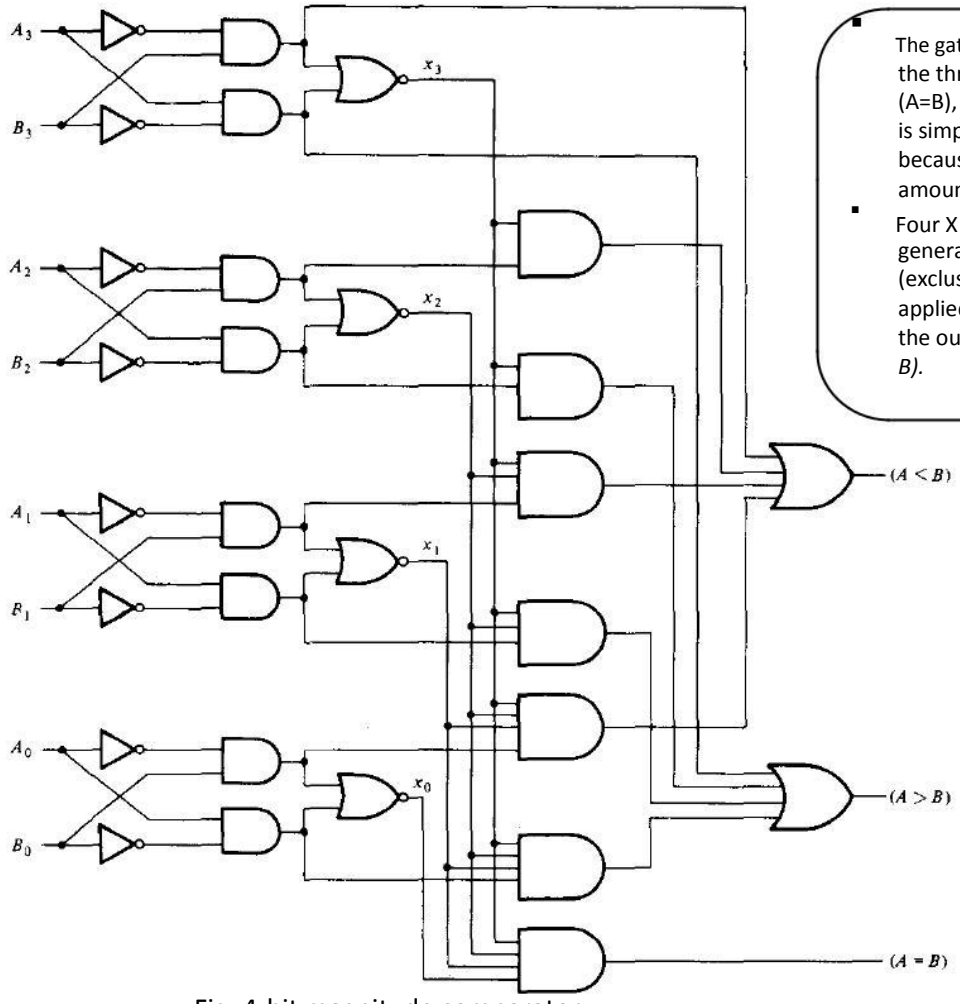
A < B: If the corresponding digit of *A* is 0 and that of *B* is 1.

The sequential comparison can be expressed logically by the following two Boolean functions:

$(A > B) = A_3B_3' + X_3A_2B_2' + x_3x_2A_1B_1' + X_3X_2X_1A_0B_0'$

$(A < B) = A_3'B_3 + X_3A_2'B_2 + x_3x_2A_1'B_1 + X_3X_2X_1A_0'B_0$

The symbols ***(A > B)*** and ***(A < B)*** are *binary* output variables that are equal to 1 when *A* > *B* or *A* < *B* respectively.



- The gate implementation of the three output variables (A=B), (A<B) and (A>B) derived is simpler than it seems because it involves a certain amount of repetition.
- Four X outputs can be generated. Equivalence (exclusive NOR) circuits and applied to an AND gate to give the output binary variable *(A = B).*

Fig: 4-bit magnitude comparator

Decoders and Encoders

Discrete quantities of information are represented in digital systems with binary codes. A binary code of *n* bits is capable of representing up to 2^n distinct elements of the coded information.

- **Decoder** is a combinational circuit that converts binary information from *n* input lines to a maximum of 2^n unique output lines.
  - If the *n-bit* decoded information has unused or don't-care combinations, the decoder output will have fewer than 2^n outputs.
  - *n-to-m-line* decoders have *m* <= 2^n.
- **Encoder** is a digital circuit that performs the inverse operation of a decoder. An encoder has 2^n (or fewer) input lines and *n* output lines. The output lines generate the binary code corresponding to the input value.
  - An example of an encoder is the octal-to-binary encoder which has eight inputs, one for each of the octal digits, and three outputs that generate the corresponding binary number.

Example: **3-to-8 line decoder**

The 3 inputs are decoded into 8 outputs, each output representing one of the minterms of the 3-input variables.

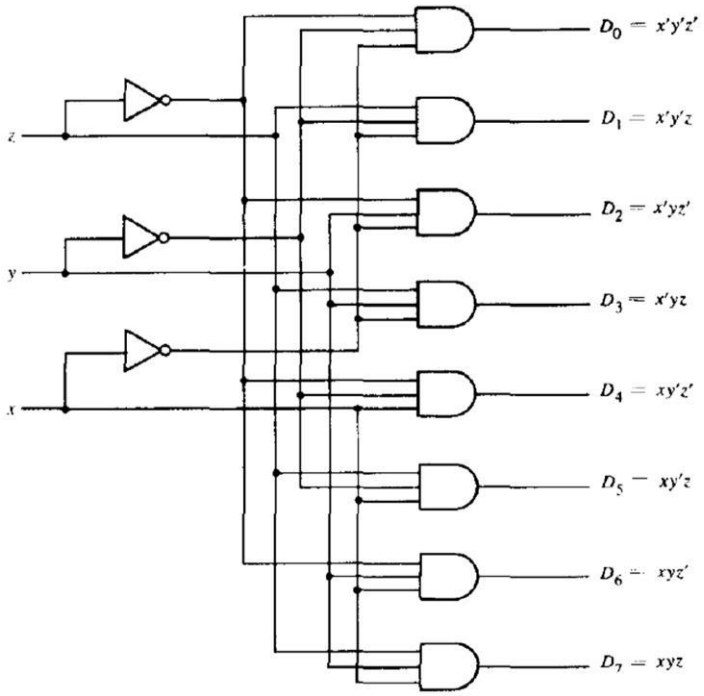


Fig: 3-to-8 line decoder

Inputs			Outputs							
x	y	z	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_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

Output variables are mutually exclusive because only one output can be equal to 1 at anyone time. The output line whose value is equal to 1 represents the minterm equivalent of the binary number presently available in the input lines.

Table: Truth-table for 3-to-8 line Decoder

Combinational logic Implementation

A decoder provides the 2^n minterm of *n* input variables. Since any Boolean function can be expressed in sum of minterms canonical form, one can use a decoder to generate the minterms and an external OR gate to form the sum.

- **Any combinational circuit with *n* inputs and *m* outputs can be implemented with an n-to-2^n-line decoder and *m* OR gates.**
- Boolean functions for the Decoder-implemented-circuit are expressed in sum of minterms. This form can be easily obtained from the truth table or by expanding the functions to their sum of minterms.

Example: *Implement a full-adder circuit with a decoder.*

Solution: From the truth table of the full-adder, we obtain the functions for this combinational circuit in sum of minterms as:

$$S(x, y, z) = \Sigma(1, 2, 4, 7)$$
$$C(x, y, z) = \Sigma(3, 5, 6, 7)$$

Since there are three inputs and a total of eight minterms, we need a 3-to-8-line decoder.

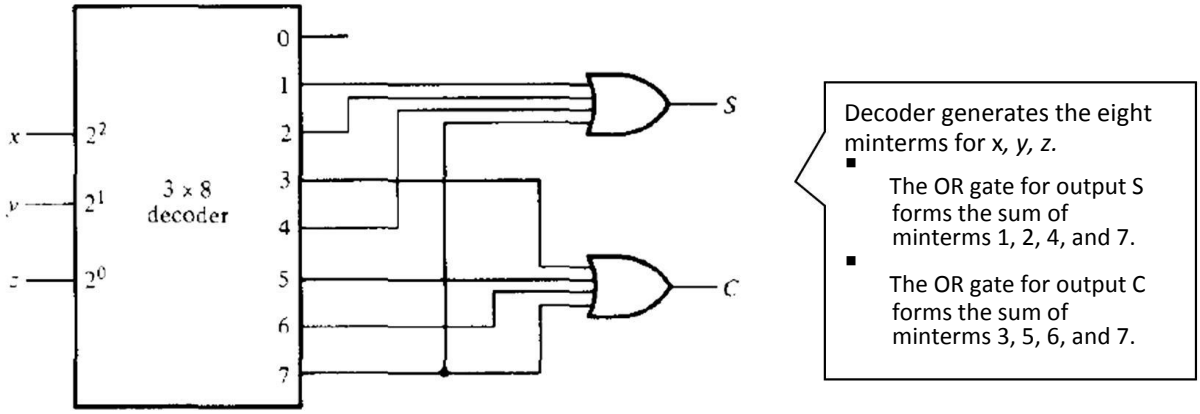


Fig: Implementation of Full-adder with a decoder circuit

Multiplexers

- A ***digital multiplexer*** is a combinational circuit that selects binary information from one of many input lines and directs it to a single output line.
  - The selection of a particular input line is controlled by a set of selection lines.
  - Normally, there are 2^n input lines and *n* selection lines whose bit combinations determine which input is selected.
- A ***demultiplexer*** is a circuit that receives information on a single line and transmits this information on one of 2^n possible output lines. The selection of a specific output line is controlled by the bit values of *n* selection lines.
  - A Decoder with an enable input can function as a demultiplexer.
  - Here, **enable input** and **input variables** for decoder is taken as **data input line** and **selection lines** for the demultiplexer respectively.

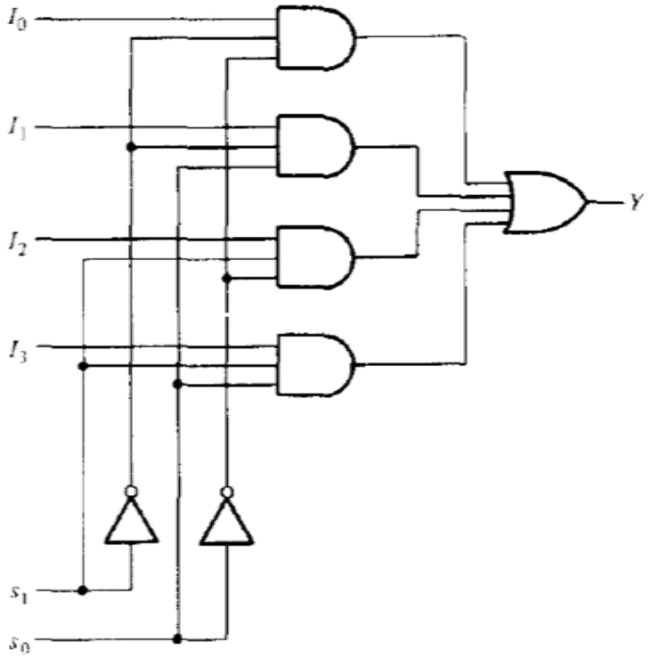


Fig: Logic Diagram: 4-to-1 line Multiplexer

s_1	s_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

Table: Function table

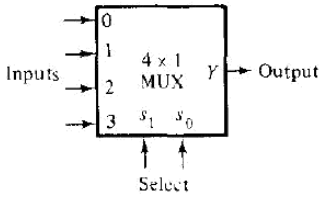


Fig: Block Diagram of Multiplexer

DEMO: consider the case when s_1s_2 = 10. The AND gate associated with input *J,* has two of its inputs equal to I and the third input connected to I_2. The other three AND gates have at least one input equal to 0, which makes their outputs equal to 0. The OR gate output is now equal to the value of I_2, thus providing a path from the selected input to the output. A multiplexer is also called a ***data selector,*** since it selects one of many inputs and steers the binary information to the output line.

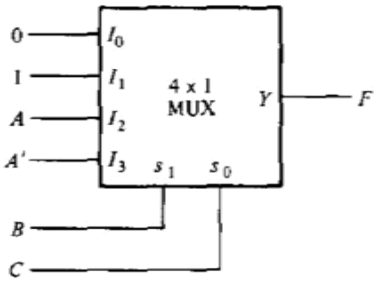
Boolean Function implementation

As decoder can be used to implement a Boolean function by employing an external OR gate, we can implement any Boolean function (in SOP) with multiplexer since multiplexer is essentially a decoder with the OR gate already available.

- If we have a Boolean function of *n* + 1 variables, we take *n* of these variables and connect them to the selection lines of a multiplexer. The remaining single variable of the function is used for the inputs of the multiplexer. If *A* is this single variable, the inputs of the multiplexer are chosen to be either *A* or *A*′ or 1 or 0. By judicious use of these four values for the inputs and by connecting the other variables to the selection lines, one can implement any Boolean function with a multiplexer.
- So, it is possible to generate any function of *n* + 1 variables with a 2^n-to-1 multiplexer.

Example: *Implement Boolean function* $F(A, B, C) = \Sigma(1, 3, 5, 6)$ *with multiplexer.*

Solution: The function can be implemented with a 4-to-1 multiplexer, as shown in Fig. below. Two of the variables, *B and* C, are applied to the selection lines in that order, i.e., *B* is connected to s_1 and C to s_0. The inputs of the multiplexer are 0, 1, *A* and *A*′.



(a) Multiplexer implementation

Minterm	A	B	C	F
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

(b) Truth table

	I_0	I_1	I_2	I_3
A'	0	①	2	③
A	4	⑤	⑥	7
	0	1	A	A'

(c) Implementation table

Most important thing during this implementation is the **implementation table** which is derived from following rules:

List the inputs of the multiplexer and under them list all the minterms in two rows. The first row lists all those minterms where *A* is complemented, and the second row all the minterms with *A* uncomplemented, as shown in above example. Circle all the minterms of the function and inspect each column separately.

- If the two minterms in a column are not circled, apply 0 to the corresponding multiplexer input.
- If the two minterms are circled, apply 1 to the corresponding multiplexer input.
- If the bottom minterm is circled and the top is not circled, apply *A* to the corresponding multiplexer input.
- If the top minterm is circled and the bottom is not circled, apply *A'* to the corresponding multiplexer input.

For clearer concept, do the following exercise:

Question: Implement the following function with a multiplexer:

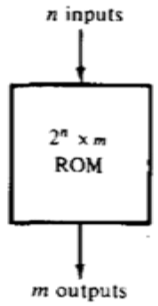
$$F(A, B, C, D) = \Sigma(0, 1, 3, 4, 8, 9, 15)$$

Solution: See text book, page no. 189.

Read Only Memory (ROM)

A read-only memory (ROM) is a device that includes both the decoder and the OR gates within a single IC package. The connections between the outputs of the decoder and the inputs of the OR gates can be specified for each particular configuration. The ROM is used to implement complex combinational circuits within one IC package or as permanent storage for binary information.

A ROM is essentially a memory (or storage) device in which permanent binary information is stored. The binary information must be specified by the designer and is then embedded in the unit to form the required interconnection pattern. ROMs come with special internal electronic fuses that can be "programmed" for a specific configuration. Once the pattern is established, it stays within the unit even when power is turned off and on again.



It consists of *n* input lines and *m* output lines. Each bit combination of the input variables is called an *address.* Each bit combination that comes out of the output lines is called a *word.* The number of bits per word is equal to the number of output lines, *m.* An address is essentially a binary number that denotes one of the minterms of *n* variables. The number of distinct addresses possible with *n* input variables is 2^n.

Example: **32 x 4 ROM** (unit consists of 32 words of 8 bits each)

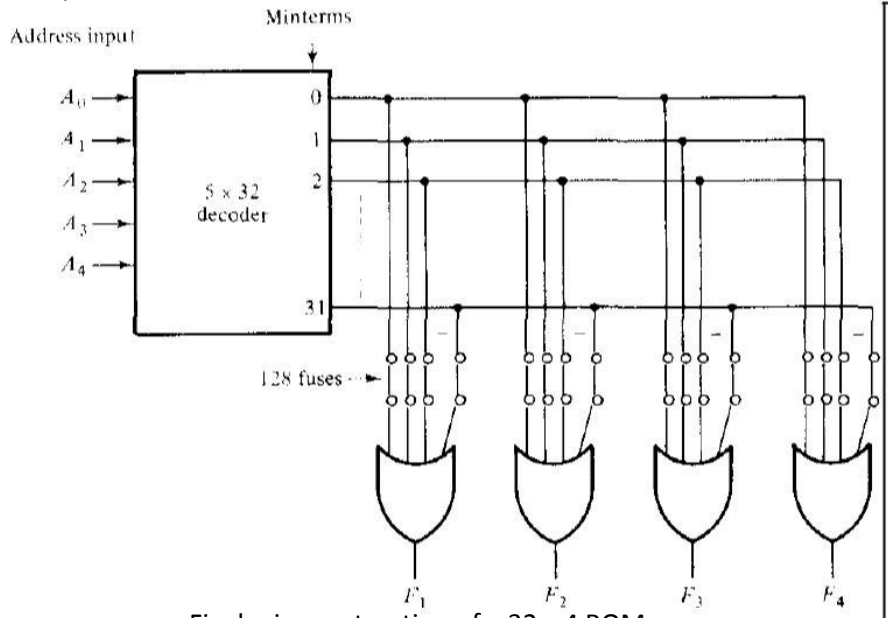


Fig: logic construction of a 32 x 4 ROM

The five input variables are decoded into 32 lines. Each output of the decoder represents one of the minterms of a function of five variables. Each one of the 32 addresses selects one and only one output from the decoder. The address is a 5-bit number applied to the inputs, and the selected minterm out of the decoder is the one marked with the equivalent decimal number. The 32 outputs of the decoder are connected through fuses to each OR gate. Only four of these fuses are shown in the diagram, but actually each OR gate has 32 inputs and each input goes through a fuse that can be blown as desired.

Combinational Logic Implementation

From the logic diagram of the ROM, it is clear that each output provides the sum of all the minterms of the *n* input variables. Remember that any Boolean function can be expressed in sum of minterms form. By breaking the links of those minterms not included in the function, each ROM output can be made to represent the Boolean function.

- For an n-input, *m-output* combinational circuit, we need a 2^n x *m* ROM.
- The blowing of the fuses is referred to as ***programming the ROM***.
- The designer need only specify a ROM program table that gives the information for the required paths in the ROM. The actual programming is a hardware procedure that follows the specifications listed in the program table.

Example: Consider a following truth table:

A_1	A_0	F_1	F_2
0	0	0	1
0	1	1	0
1	0	1	1
1	1	1	0

Truth table specifies a combinational circuit with 2 inputs and 2 outputs. The Boolean function can be represented in SOP:

$$F_1(A_1, A_0) = \Sigma(1, 2, 3)$$
$$F_2(A_1, A_0) = \Sigma(0, 2)$$

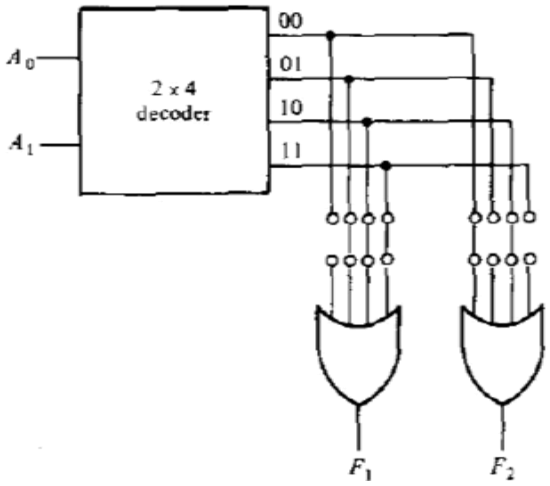


Diagram shows the internal construction of a 4X2 ROM. It is now necessary to determine which of the eight available fuses must be blown and which should be left intact. This can be easily done from the output functions listed in the truth table. Those minterms that specify an output of 0 should not have a path to the output through the OR gate. Thus, for this particular case, ti1e truth table shows three 0's, and their corresponding fuses to the OR gates must be blown.

Fig: Combinational-circuit implementation with a 4 x 2 ROM

This example demonstrates the general procedure for implementing any combinational circuit with a ROM. From the number of inputs and outputs in the combinational circuit, we first determine the size of ROM required. Then we must obtain the programming truth table of the ROM; no other manipulation or simplification is required. The 0's (or 1's) in the output functions of the truth table directly specify those fuses that must be blown to provide the required combinational circuit in sum of min terms form.

Question: Design a combinational circuit using a ROM. The circuit accepts a 3-bit number and generates an output binary number equal to the square of the input number.

Solution: See text book page no. 184, if any confusion, do ask me (even call me… ☺)

Types of ROM

ROMs: For simple ROMs, *mask programming* is done by the manufacturer during the fabrication process of the unit. The procedure for fabricating a ROM requires that the customer fill out the truth table the ROM is to satisfy. The truth table may be submitted on a special form provided by the manufacturer. The manufacturer makes the corresponding mask for the paths to produce the 1's and 0's according to the customer's truth table. This procedure is costly because the vendor charges the customer a special fee for custom masking a ROM. For this reason, mask programming is economical only if large quantities of the same ROM configuration are to be manufactured.

PROMs: *Programmable read-only memory* or PROM units contain all 0's (or all 1's) in every bit of the stored words. The approach called *field programming* is applied for fuses in the PROM which are blown by application of current pulses through the output terminals. This allows the user to program the unit in the laboratory to achieve the desired relationship between input addresses and stored words. Special units called *PROM programmers* are available commercially to facilitate this procedure. In any case, all procedures for programming ROMs are *hardware* procedures even though the word *programming* is used.

EPROMs: The hardware procedure for programming ROMs or PROMs is irreversible and, once programmed, the fixed pattern is permanent and cannot be altered. Once a bit pattern has been

established, the unit must be discarded if the bit pattern is to be changed. A third type of unit available is called ***erasable PROM,*** or **EPROM**. EPROMs can be restructured to the initial value (all 0's or all 1's) even though they have been changed previously. When an EPROM is placed under a special ultraviolet light for a given period of time, the shortwave radiation discharges the internal gates that serve as contacts**.** After erasure, the ROM returns to its initial state and can be reprogrammed.

EEPROMs: Certain ROMs can be erased with electrical signals instead of ultraviolet light, and these are called ***electrically erasable PROMs,*** or **EEPROMs**.

Point!

The function of a ROM can be interpreted in two different ways:

- The *first interpretation* is of a unit that implements any combinational circuit. From this point of view, each output terminal is considered separately as the output of a Boolean function expressed in sum of minterms.
- The *second interpretation* considers the ROM to be a storage unit having a fixed pattern of bit strings called *words.* From this point of view, the inputs specify an *address* to a specific stored word, which is then applied to the outputs. This is the reason why the unit is given the name *read-only memory. Memory* is commonly used to designate a storage unit. *Read* is commonly used to signify that the contents of a word specified by an address in a storage unit is placed at the output terminals. Thus, a ROM is a memory unit with a fixed word pattern that can be read out upon application of a given address.

Programmable Logic Array (PLA)

A combinational circuit may occasionally have don't-care conditions. When implemented with a ROM, a don't care condition becomes an address input that will never occur. The words at the don't-care addresses need not be programmed and may be left in their original state (all 0's or all 1's). The result is that not all the bit patterns available in the ROM are used, which may be considered a waste of available equipment.

Def^{n}*: Programmable Logic Array* or PLA is LSI component that can be used in economically as an alternative to ROM where number of don't-care conditions is excessive.

Difference between ROM and PLA

ROM	PLA
1. ROM generates all the minterms as an output of decoder.	1. PLA does not provide full decoding of the variables.
2. Uses decoder	2. Decoder is replaced by group of AND gates each of which can be programmed to generate a product term of the input variables.
3. The size of the PLA is specified by the number of inputs (n) and the number of outputs (m).	3. The size of the PLA is specified by the number of inputs (n), the number of product terms (k), and the number of outputs (m) (=number of sum terms)
4. No. of programmed fuses = $2^n * m$	4. No. of programmed fuses = $2n * k + k * m + m$

Block Diagram of PLA

A block diagram of the PLA is shown in Fig. below. It consists of *n* inputs, *m* outputs, *k* product terms, and *m* sum terms. The product terms constitute a group of *k* AND gates and the sum terms constitute a group of *m* OR gates. Fuses are inserted between all *n* inputs and their complement values to each of the AND gates. Fuses are also provided between the outputs of the AND gates and the inputs of the OR gates.

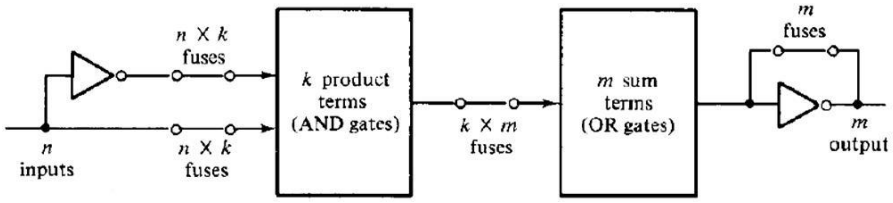


Fig: PLA block diagram

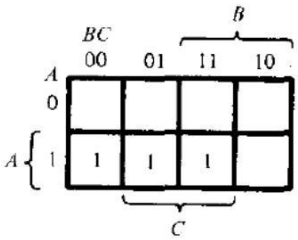
PLA program table and Boolean function Implementation

The use of a PLA must be considered for combinational circuits that have a large number of inputs and outputs. It is superior to a ROM for circuits that have a large number of don't-care conditions. Let me explain the example to demonstrate how PLA is programmed.

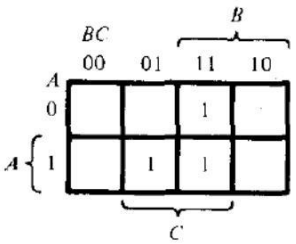
Consider a truth table of the combinational circuit:

A	B	C	F_1	F_2
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	0	1
1	0	0	1	0
1	0	1	1	1
1	1	0	0	0
1	1	1	1	1

PLA implements the functions in their sum of products form (standard form, not necessarily canonical as with ROM). Each product term in the expression requires an AND gate. It is necessary to simplify the function to a minimum number of product terms in order to minimize the number of AND gates used. The simplified functions in sum of products are obtained from the following maps:



$F_1 = AB' + AC$



$F_2 = AC + BC$

There are three distinct product terms in this combinational circuit: *AB', AC* and *BC.* The circuit has three inputs and two outputs; so the PLA can be drawn to implement this combinational circuit.

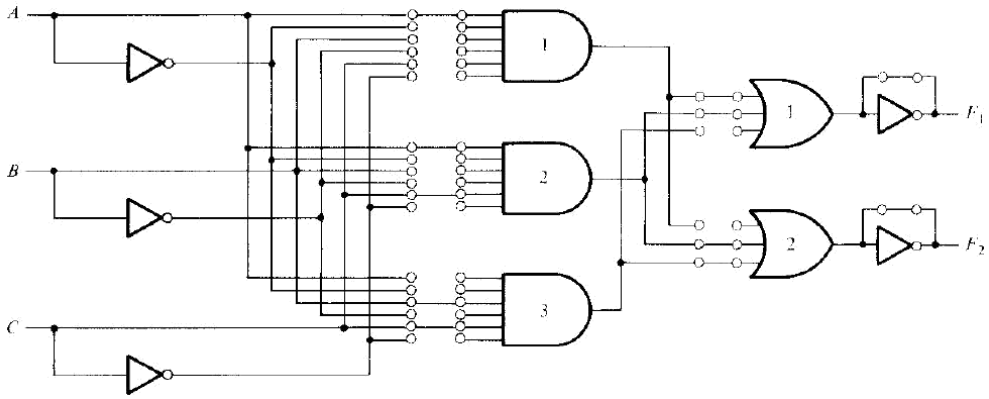


Fig: PLA with 3 inputs, 3 product terms, and 2 outputs

Programming the PLA means, we specify the paths in its AND-OR-NOT pattern. A typical **PLA program table** consists of three columns.

First column: lists the product terms numerically.

Second column: specifies the required paths between inputs and AND gates.

Third column: specifies the paths between the AND gates and the OR gates.

Under each output variable, we write a *T* (for true) if the output inverter is to be bypassed, and C (for complement) if the function is to be complemented with the output inverter.

	Product term	Inputs A	B	C	Outputs F_1	F_2
AB'	1	1	0	–	1	–
AC	2	1	–	1	1	1
BC	3	–	1	1	–	1
					T	T	T/C

Table: PLA program table

For each product term, the inputs are marked with 1, 0 or - (dash).

- If a variable in the product term appears in its normal form (unprimed), the corresponding input variable is marked with a 1.
- If it appears complemented (primed), the corresponding input variable is marked with a 0.
- If the variable is absent in the product term, it is marked with a dash.

Each product term is associated with an AND gate. The paths between the inputs and the AND gates are specified under the column heading *inputs.* A 1 in the input column specifies a path from the corresponding input to the input of the AND gate that forms the product term. A 0 in the input column specifies a path from the corresponding complemented input to the input of the AND gate. A dash specifies no connection.

The appropriate fuses are blown and the ones left intact form the desired paths. It is assumed that the open terminals in the AND gate behave like a 1 input. The paths between the AND and OR gates are specified under the column heading *outputs.* The output variables are marked with 1's for all those product terms that formulate the function. We have

$$F_1 = AB' + AC$$

So F_1 is marked with 1's for product terms 1 and 2 and with a dash for product term 3. Each product term that has a 1 in the output column requires a path from the corresponding AND gate to the output OR gate.

Unit 6
Sequential Logic

Introduction

Till now, we study combinational circuits in which the outputs at any instant of time are entirely dependent upon the inputs present at that time. Although every digital system is likely to have combinational circuits, most systems encountered in practice also include memory elements, which require that the system be described in terms of ***sequential logic***.

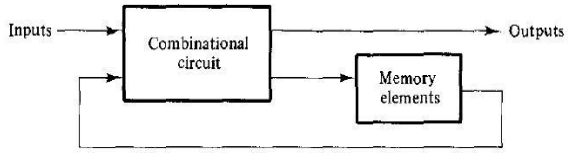


Fig: Block diagram of sequential circuit

- Memory elements are devices capable of storing binary information within them. The binary information stored in the memory elements at any given time defines the *state* of the sequential circuit.
- Block diagram shows **external outputs** in a sequential circuit are a function not only of **external inputs**, but also of the **present state** of the memory elements. Thus, a sequential circuit is specified by a time sequence of inputs, outputs, and internal states.

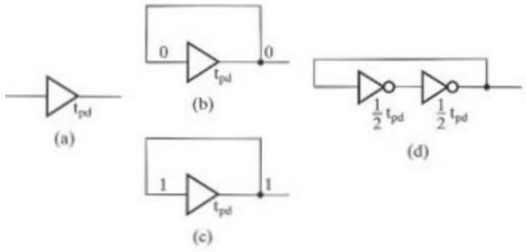
There are two main types of sequential circuits. Their classification depends on the timing of their signals.

- ***Synchronous sequential circuit***: whose behavior can be defined from the knowledge of its signals at discrete instants of time
  - A synchronous sequential logic system, by definition, must employ signals that affect the memory elements only at discrete instants of time. One way of achieving this goal is to use **pulses** of limited duration throughout the system so that one pulse-amplitude represents logic-1 and pulse amplitude (or the absence of a pulse) represents logic-0. The difficulty with a system of pulses is that any two pulses arriving from separate independent sources to the inputs of the same gate will exhibit unpredictable delays, will separate the pulses slightly, and will result in unreliable operation.
  - Practical synchronous sequential logic systems use fixed amplitudes such as voltage levels for the binary signals. Synchronization is achieved by a timing device called a ***master-clock generator,*** which generates a periodic train of ***clock pulses.*** The clock pulses are distributed throughout the system in such a way that memory elements are affected only with the arrival of the synchronization pulse. Synchronous sequential circuits that use clock pulses in the inputs of memory elements are called ***clocked sequential circuits.*** Clocked sequential circuits are the type encountered most frequently. They do not manifest instability problems and their timing is easily divided into independent discrete steps, each of which is considered separately. The sequential circuits discussed in this chapter are exclusively of the clocked type.

- ***Asynchronous sequential circuit***: Behavior depends upon the order in which its input signals change and can be affected at any instant of time. The memory elements commonly used in asynchronous sequential circuits are time-delay devices.

Information storage in digital system

- Fig (a) shows a buffer which has a propagation delay t_{pd} and can store information for time t_{pd} since buffer input at time t reaches to its output at time t_{pd}. But, in general, we wish to store information for an indefinite time that is typically much longer than the time delay of one or even many gates. This stored value is to be changed at arbitrary times based on the inputs applied to the circuit and should not depend on the specific time delay of a gate.
- In Fig (b) we have output of buffer connected to its input making a feedback path. This time input to buffer has been 0 for at least time t_{pd}. Then the output produced by the buffer will be 0 at time $t + t_{pd}$. This output is applied to the input so that the output will also be 0 at time $1 + 2t_{pd}$. This relationship between input and output holds for all t, so the 0 will be stored indefinitely.
- A buffer is usually implemented by using two inverters, as shown in Fig (d). The signal is inverted twice, i.e. (X')' = X, giving no net inversion of the signal around the loop.



- In fact, this example is an illustration of one of the most popular methods of implementing storage in computer memories.
- With inverters there is no way to change the information stored. By replacing the inverters with NOR or NAND gates, the information can be changed. Asynchronous storage circuits called latches are made in this manner.

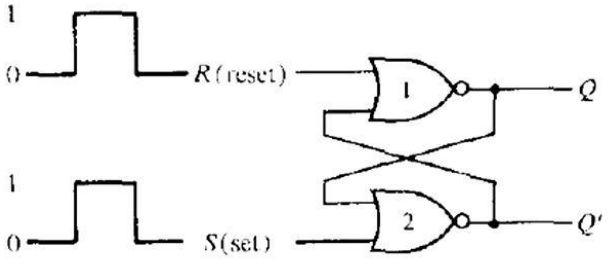
Flip-Flops

The memory elements used in clocked sequential circuits are called *flip-flops.* These circuits are binary cells capable of storing one bit of information. A flip-flop circuit has two outputs, one for the normal value and one for the complement value of the bit stored in it. Binary information can enter a flip-flop in a variety of ways, a fact that gives rise to different types of flip-flops.

- A flip-flop circuit can maintain a binary state indefinitely (as long as power is delivered to the circuit) until directed by an input signal to switch states.
- The major differences among various types of flip-flops are in the number of inputs they possess and in the manner in which the inputs affect the binary state.

Basic flip-flop circuit (*direct-coupled RS* flip-flop or *SR latch*)

A flip-flop circuit can be constructed from two NAND gates or two NOR gates. These constructions are shown in the logic diagrams below. Each circuit forms a basic flip-flop upon which other more complicated types can be built. The cross-coupled connection from the output of one gate to the input of the other gate constitutes a feedback path. For this reason, the circuits are classified as asynchronous sequential circuits. Each flip-flop has **two outputs**, Q and Q', and **two inputs**, *set* and *reset.*



S	R	Q	Q'	
1	0	1	0	
0	0	1	0	(after S = 1, R = 0)
0	1	0	1	
0	0	0	1	(after S = 0, R = 1)
1	1	0	0	

(a) Logic diagram (b) Truth table

Fig: basic flip-flop circuit with NOR gates

- Output of a NOR gate is 0 if any input is 1, and that the output is 1 only when all inputs are 0.
- First, assume that the set input is 1 and the reset input is 0. Since gate-2 has an input of 1, its output Q' must be 0, which puts both inputs of gate-1 at 0, so that output Q is 1. When the set input is returned to 0, the outputs remain the same i.e. output Q' stay at 0, which leaves both inputs of gate-1 at 0, so that output Q is 1.
- Similarly, 1 in the reset input changes output Q to 0 and Q' to 1. When the reset input returns to 0, the outputs do not change.
- When a 1 is applied to both the set and the reset inputs, both Q and Q' outputs go to 0. This condition violates the fact that outputs Q and Q' are the complements of each other. In normal operation, this condition must be avoided by making sure that 1's are not applied to both inputs simultaneously.

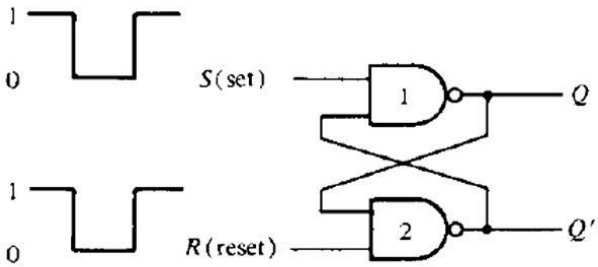
A flip-flop has two useful states.

Set state: When Q = 1 and Q' = 0, (or 1-state),

Reset state: When Q = 0 and Q' = 1, (or 0-state)

The outputs Q and Q' are complements of each other and are referred to as the normal and complement outputs, respectively. The binary state of the flip-flop is taken to be the value of the normal output.

Under normal operation, both inputs remain at 0 unless the state of the flip-flop has to be changed. The application of a momentary 1 to the set input causes the flip-flop to go to the set state. The set input must go back to 0 before a 1 is applied to the reset input. A momentary 1 applied to the reset input causes the flip-flop to go the clear state. When both inputs are initially 0, a 1 applied to the set input while the flip-flop is in the set state or a 1 applied to the reset input while the flip-flop is in the clear state, leaves the outputs unchanged.



S	R	Q	Q'	
1	0	0	1	
1	1	0	1	(after S = 1, R = 0)
0	1	1	0	
1	1	1	0	(after S = 0, R = 1)
0	0	1	1	

(a) Logic diagram (b) Truth table

Fig: Basic flip-flop circuit with NAND gates

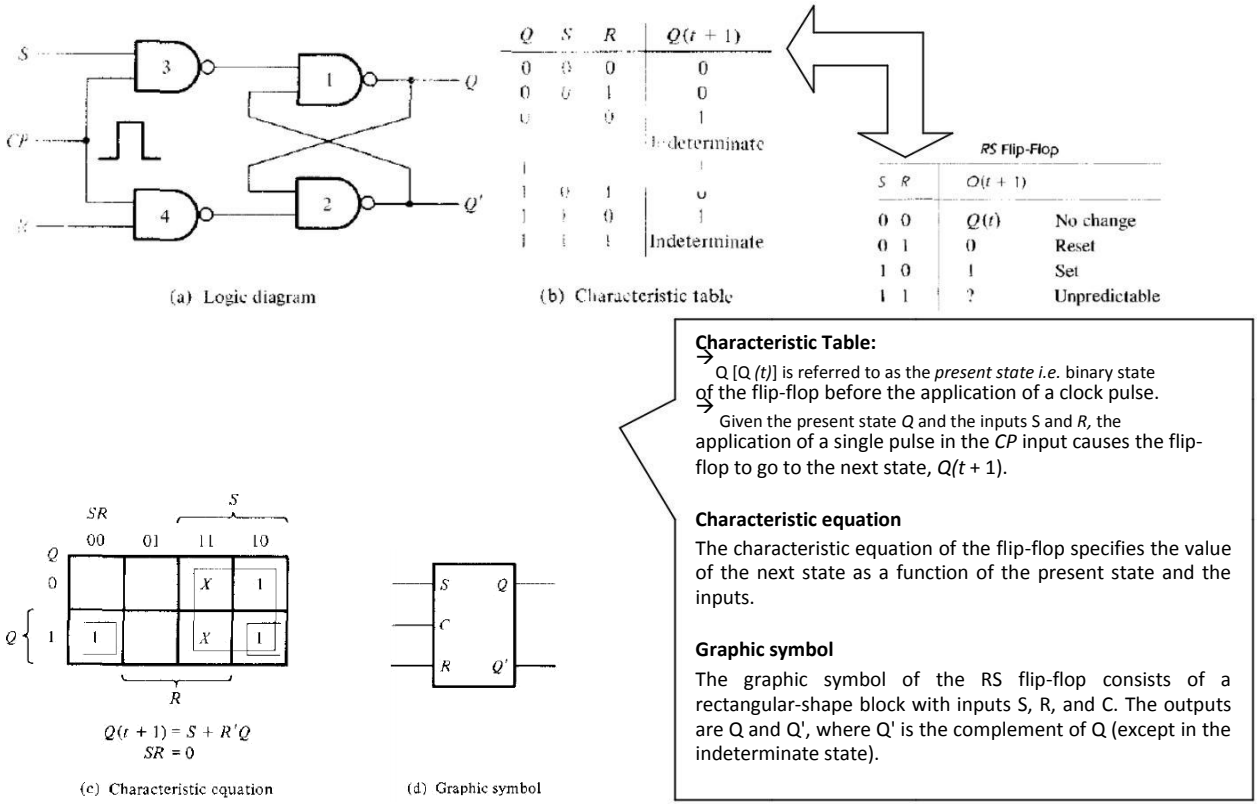
- The NAND basic flip-flop circuit operates with both inputs normally at 1 unless the state of the flip-flop has to be changed.
- The application of a momentary 0 to the set input causes output Q to go to 1 and Q' to go to 0, thus putting the flip-flop into the set state
- After the set input returns to 1, a momentary 0 to the reset input causes a transition to the clear state.
- When both inputs go to 0, both outputs go to 1- a condition avoided in normal flip-flop operation.

The operation of the basic flip-flop can be modified by providing an additional control input that determines when the state of the circuit is to be changed. This fact arises 4 common types of flip-flops and are discussed in what follows:

1. RS Flip-Flop

It consists of a basic flip-flop circuit and two additional NAND gates along with clock pulse (CP) input. The pulse input acts as an enable signal for the other two inputs.

- When the pulse input goes to 1, information from the S or *R* input is allowed to reach the output.
- Set state: S = 1, *R* = 0, and *CP* = 1.
- Reset state: S = 0, *R* = 1, and *CP* = 1.
- In either case, when *CP* returns to 0, the circuit remains in its previous state. When *CP* = 1 and both the S and *R* inputs are equal to 0, the state of the circuit does not change.



Characteristic Table:

→ Q [Q *(t)*] is referred to as the *present state i.e.* binary state of the flip-flop before the application of a clock pulse.

→ Given the present state *Q* and the inputs S and *R,* the application of a single pulse in the *CP* input causes the flip-flop to go to the next state, *Q(t* + 1).

Characteristic equation

The characteristic equation of the flip-flop specifies the value of the next state as a function of the present state and the inputs.

Graphic symbol

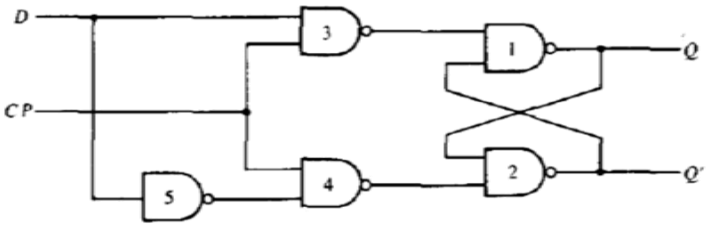
The graphic symbol of the RS flip-flop consists of a rectangular-shape block with inputs S, R, and C. The outputs are Q and Q', where Q' is the complement of Q (except in the indeterminate state).

Indeterminate condition makes the circuit of Fig. above difficult to manage and it is seldom used in practice. Nevertheless, it is an important circuit because all other flip-flops are constructed from it.

D Flip-Flop

One way to eliminate the undesirable condition of the indeterminate state in the *RS* flip-flop is to ensure that inputs S and *R* are never equal to 1 at the same time. This is done in the *D* flip-flop shown in Fig. below. The *D* flip-flop has only two inputs: *D* and *CP.* The *D* input goes directly to the S input and its complement is applied to the *R* input.

- As long as CP is 0, the outputs of gates 3 and 4 are at the 1 level and the circuit cannot change state regardless of the value of *D.*
- The *D* input is sampled when *CP* = 1.
  - If *D* is 1, the *Q* output goes to 1, placing the circuit in the **set state**.
  - If *D* is 0, output *Q* goes to 0 and the circuit switches to the **clear state**.



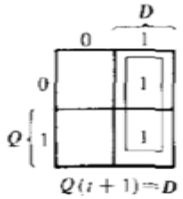
(a) Logic diagram

D Flip-Flop

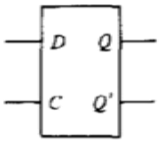
D	Q(t + 1)	
0	0	Reset
1	1	Set

Q	D	Q(t + 1)
0	0	0
0	1	1
1	0	0
1	1	1

(b) Characteristic table



(c) Characteristic equation

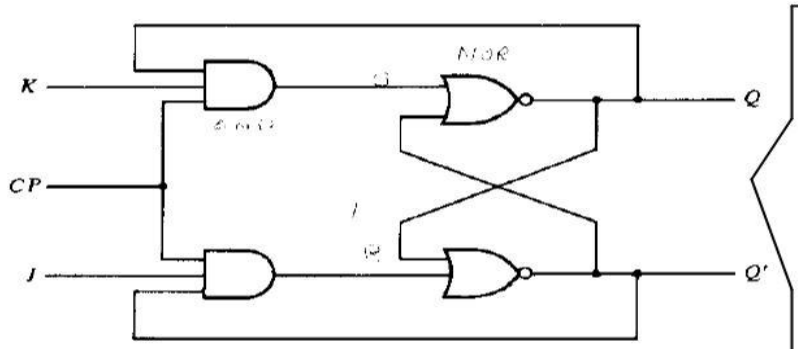


(d) Graphic symbol

JK Flip-Flop

A *JK* flip-flop is a refinement of the *RS* flip-flop in that the indeterminate state of the *RS* type is defined in the *JK* type. Inputs *J* and *K* behave like inputs *S* and *R* to set and clear the flip-flop, respectively. The input marked *J* is for *set* and the input marked *K* is for *reset.* When both inputs *J* and *K* are equal to 1, the flip-flop switches to its complement state, that is, if *Q* = 1, it switches to *Q* = 0, and vice versa.

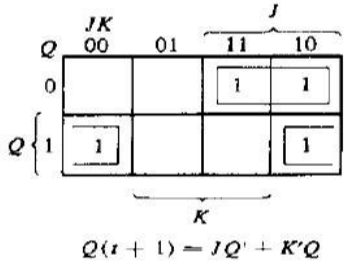
A *JK* flip-flop constructed with two cross-coupled NOR gates and two AND gates is shown in Fig. below:



(a) Logic diagram

Q	J	K	Q (t + 1)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

(b) Characteristic table



$Q(t+1) = JQ' + K'Q$

(c) Characteristic equation

→ A *JK* flip-flop constructed with two cross-coupled NOR gates and two AND gates.

→ Output *Q* is ANDed with *K* and *CP* inputs so that the flip-flop is cleared during a clock pulse only if *Q* was previously 1.

→ Similarly, output *Q*′ is ANDed with J and *CP* inputs so that the flop-flop is set with a clock pulse only when *Q*′ was previously 1.

→ Because of the feedback connection in the *JK* flipflop, a *CP* pulse that remains in the 1 state while both *J* and *K* are equal to 1 will cause the output to complement again and repeat complementing until the pulse goes back to 0.

→ To avoid this undesirable operation, the clock pulse must have a time duration that is shorter than the propagation delay time of the flip-flop. This is a restrictive requirement of JK which is eliminated with a master-slave or edge-triggered construction.

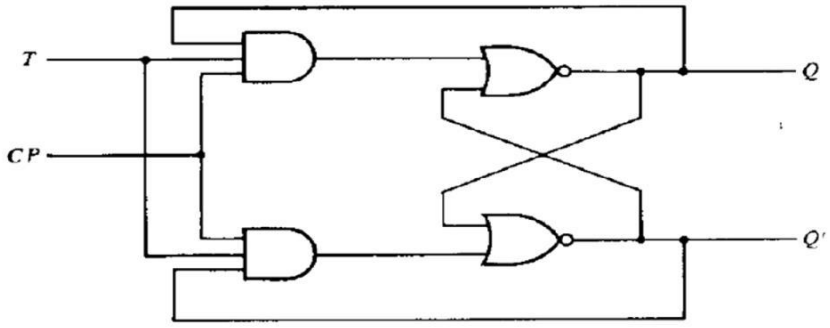
JK Flip-Flop

J	K	Q(t + 1)	
0	0	Q(t)	No change
0	1	0	Reset
1	0	1	Set
1	1	Q′(t)	Complement

T Flip-Flop

The *T* flip-flop is a single-input version of the *JK* flip-flop and is obtained from the *JK* flip-flop when both inputs are tied together. The designation *T* comes from the ability of the flip-flop to "toggle," or complement, its state. Regardless of the present state, the flip-flop complements its output when the clock pulse occurs while input *T* is 1. The characteristic table and characteristic equation show that:

- When $T = 0$, $Q(t + 1) = Q$, that is, the next state is the same as the present state and no change occurs.
- When $T = 1$, then $Q(t + 1) = Q'$, and the state of the flip-flop is complemented.



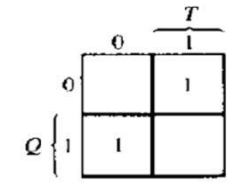
(a) Logic diagram

T Flip-Flop

T	$Q(t+1)$	
0	$Q(t)$	No change
1	$Q'(t)$	Complement

Q	T	$Q(t+1)$
0	0	0
0	1	1
1	0	1
1	1	0

(b) Characteristic table



$Q(t + 1) = TQ' + T'Q$

(c) Characteristic equation

Triggering of Flip-Flops

The state of a flip-flop is switched by a momentary change in the input signal. This momentary change is called a ***trigger*** and the transition it causes is said to trigger the flip-flop. Clocked flip-flops are triggered by ***pulses***. A pulse starts from an initial value of 0, goes momentarily to 1, and after a short time, returns to its initial 0 value.

A clock pulse may be either positive or negative.

- A positive clock source remains at o during the interval between pulses and goes to 1 during the occurrence of a pulse. The pulse goes through two signal transitions: from 0 to 1 and the return from 1 to 0. As shown in Fig. below, the positive transition is defined as the *positive edge* and the negative transition as the *negative edge.*
- This definition applies also to negative pulses.

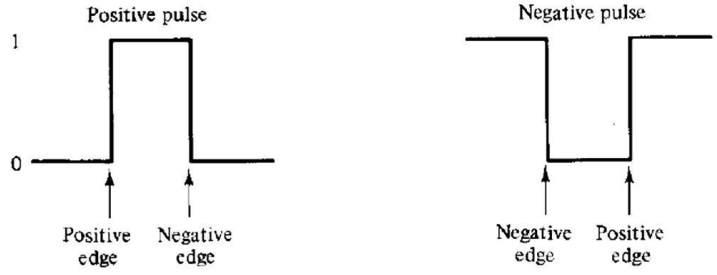


Fig: Definition of clock pulse transition

The clocked flip-flops introduced earlier are triggered during the positive edge of the pulse, and the state transition starts as soon as the pulse reaches the logic-1 level. The new state of the flip-flop may appear at the output terminals while the input pulse is still 1. If the other inputs of the flip-flop change while the clock is still 1, the flip-flop will start responding to these new values and a new output state may occur.

Edge triggering is achieved by using a master -slave or edge triggered flip-flop as discussed in what follows.

1. Master-slave Flip-Flop

A master-slave flip-flop is constructed from two separate flip-flops. One circuit serves as a **master** and the other as a **slave**, and the overall circuit is referred to as a *master slave flip-flop.*

- ***RS master-slave flip-flop***

  It consists of a master flip-flop, a slave flip-flop, and an inverter. When clock pulse CP is 0, the output of the inverter is 1. Since the clock input of the slave is 1, the flip-flop is enabled and output Q is equal to Y, while Q' is equal to Y'. The master flip-flop is disabled because CP = 0. When the pulse becomes 1, the information then at the external R and S inputs is transmitted to the master flip-flop. The slave flip-flop, however, is isolated as long as the pulse is at its 1 level, because the output of the inverter is 0. When the pulse returns to 0, the master flip-flop is isolated; this prevents the external inputs from affecting it. The slave flip-flop then goes to the same state as the master flip-flop.

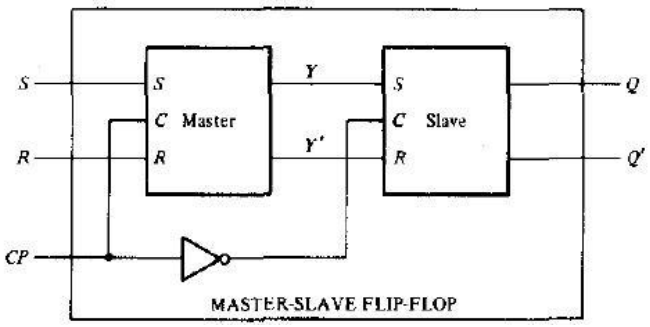


Fig: Logic diagram of RS master slave flip-flop

- ***JK Master-slave Flip-Flop***

Master-slave *JK* flip-flop constructed with NAND gates is shown in Fig. below. It consists of two flip-flops; gates 1 through 4 form the **master flip-flop**, and gates 5 through 8 form the **slave flip-flop**. The information present at the *J* and *K* inputs is transmitted to the master flip-flop on the positive edge of a clock pulse and is held there until the negative edge of the clock pulse occurs, after which it is allowed to pass through to the slave flip-flop.

Operation:

- The clock input is normally 0, which prevents the *J* and *K* inputs from affecting the master flip-flop.
- The slave flip-flop is a clocked *RS* type, with the master flip-flop supplying the inputs and the clock input being inverted by gate 9.
- When the clock is 0, $Q = Y$, and $Q' = Y'$.
- When the positive edge of a clock pulse occurs, the master flip-flop is affected and may switch states.
- The slave flip-flop is isolated as long as the clock is at the 1 level
- When the clock input returns to 0, the master flip-flop is isolated from the *J* and *K* inputs and the slave flip-flop goes to the same state as the master flip-flop.

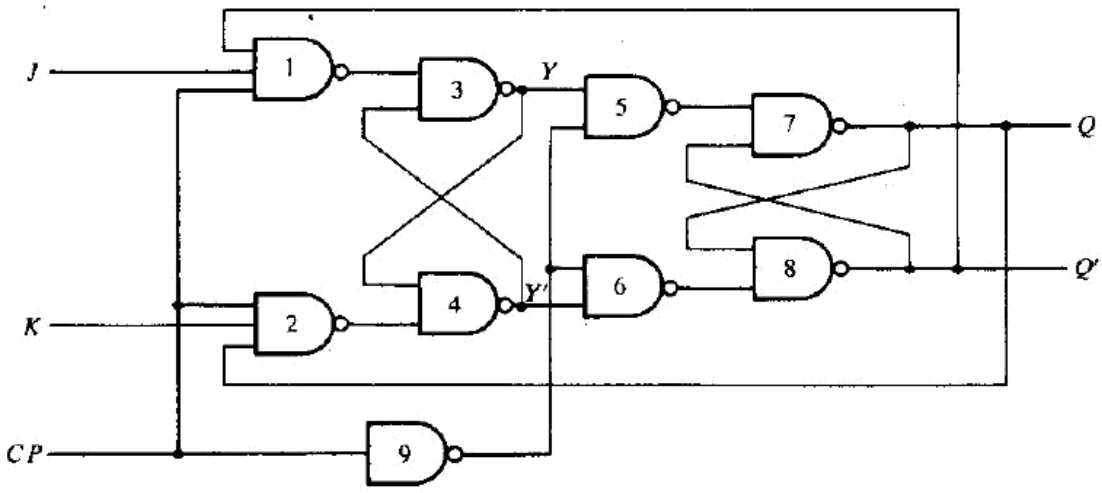


Fig: Clocked master-slave JK flip-flop

Point!

Consider a digital system containing many master-slave flip-flops, with the outputs of some flip-flops going to the inputs of other flip-flops. Assume that clock-pulse inputs to all flip-flops are synchronized (occur at the same time). At the beginning of each clock pulse, some of the master elements change state, but all flip-flop outputs remain at their previous values. After the clock pulse returns to 0, some of the outputs change state, but none of these new states have an effect on any of the master elements until the next clock pulse. Thus, **the states of flip-flops in the system can be changed simultaneously during the same clock pulse**, even though outputs of flip-flops are connected to inputs of flip-flops. This is possible because the new state appears at the output terminals only after the clock pulse has returned to O. Therefore, the binary content of one flip-flop can be transferred to a second flip-flop and the content of the second transferred to the first, and both transfers can occur during the same clock pulse.

2. Edge-Triggered Flip-Flop

Edge-triggered flip-flop (alternative to master-slave) synchronizes the state changes during clock-pulse transitions. In this type of flip-flop, output transitions occur at a specific level of the clock pulse. When the pulse input level exceeds this threshold level, the inputs are locked out and the flip-flop is therefore unresponsive to further changes in inputs until the clock pulse returns to 0 and another pulse occurs. Some edge-triggered flip-flops cause a transition on the positive edge of the pulse, and others cause a transition on the negative edge of the pulse.

The logic diagram of a D-type positive-edge-triggered flip-flop is shown below. It consists of three basic flip-flops. NAND gates 1 and 2 make up one basic flip-flop and gates 3 and 4 another. The third basic flip-flop comprising gates 5 and 6 provides the outputs to the circuit. Inputs S and *R* of the third basic flip-flop must be maintained at logic-1 for the outputs to remain in their steady state values.

- When S = 0 and *R* = 1, the output goes to the set state with Q = 1.
- When S = 1 and *R* = 0, the output goes to the clear state with Q = 0.

Inputs S and R are determined from the states of the other two basic flip-flops. These two basic flip-flops respond to the external inputs D (data) and CP (clock pulse).

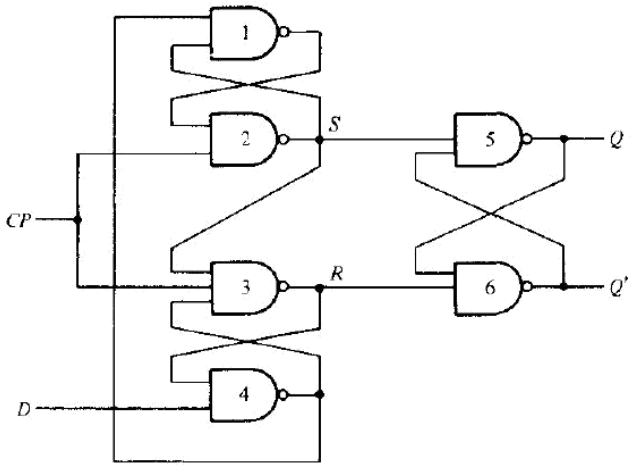
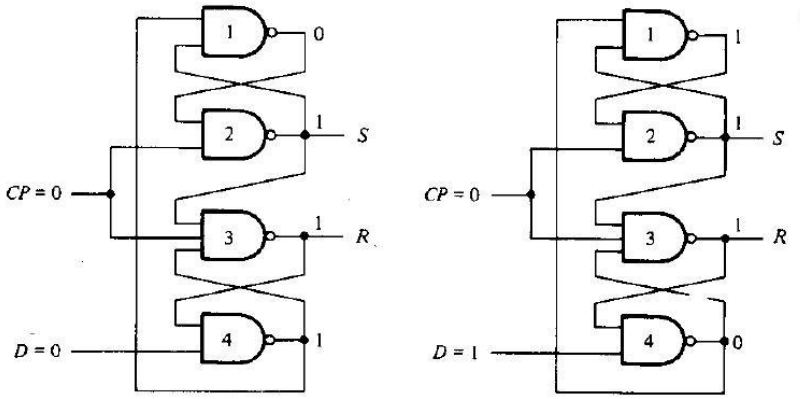
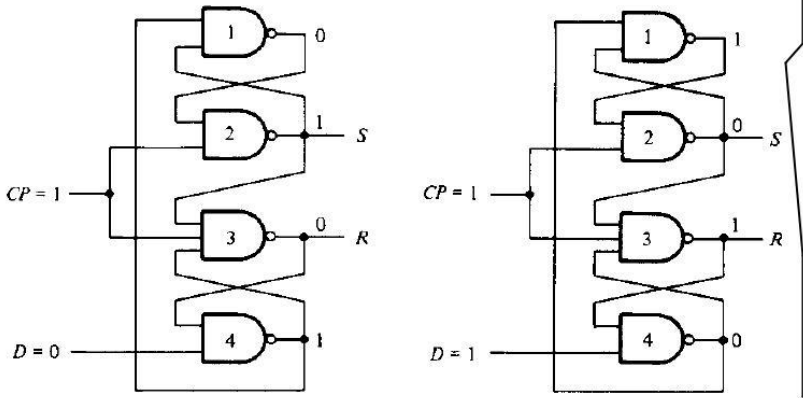


Fig: D-type positive-edge-triggered flip-flop

Operation:



(a) With $CP = 0$



(b) With $CP = 1$

- Gates 1 to 4 are redrawn to show all possible transitions. Outputs S and *R* from gates 2 and 3 go to gates 5 and 6
- Fig (a) shows the binary values at the outputs of the four gates when *CP* = 0. Input *D* may be equal to 0 or 1. In either case, a *CP* of 0 causes the outputs of gates 2 and 3 to go to 1, thus making S = *R* = 1, which is the condition for a steady state output.
- When *CP* = 1
  - If *D* = 1 then S changes to 0, but *R* remains at 1, which causes the output of the flip-flop *Q* to go to 1 (set state).
  - If D = 0 then S = 1 and R = 0. Flip-flop goes to clear state (Q = 0).

Direct Inputs

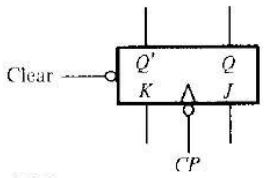
Flip-flops available in IC packages sometimes provide special inputs for setting or clearing the flip-flop asynchronously. These inputs are usually called ***direct preset*** and ***direct clear****.* They affect the flip-flop on a positive (or negative) value of the input signal without the need for a clock pulse. These inputs are useful for bringing all flip-flops to an initial state prior to their clocked operation.

Example: After power is turned on in a digital system, the states of its flip-flops are indeterminate. A *clear* switch clears all the flip-flops to an initial cleared state and a *start* switch begins the system's clocked operation. The clear switch must clear all flip-flops asynchronously without the need for a pulse.

→ The direct-clear input has a small circle to indicate that, normally, this input must be maintained at 1. If the clear input is maintained at 0, the flip-flop remains cleared, regardless of the other inputs or the clock pulse.

→ The function table specifies the circuit operation. The X's are don't-care conditions, which indicate that a 0 in the direct-clear input disables all other inputs. Only when the clear input is I would a negative transition of the clock have an effect on the outputs.

→ The outputs do not change if J = *K* = O. The flip-flop toggles, or complements, when J = *K* = I. Some flip-flops may also have a direct-preset input, which sets the output *Q* to I (and *Q*′ to 0) asynchronously.



Function table

Inputs				Outputs	
Clear	Clock	*J*	*K*	*Q*	*Q*′
0	*X*	*X*	*X*	0	1
1	↓	0	0	No change	
1	↓	0	1	0	1
1	↓	1	0	1	0
1	↓	1	1	Toggle	

Fig: JK flip-flop with direct clear

Analysis of Clocked Sequential Circuits

The behavior of a sequential circuit is determined from the inputs, the outputs, and the state of its flip-flops. The outputs and the next state are both a function of the inputs and the present state. The **analysis of a sequential circuit** consists of obtaining a table or a diagram for the time sequence of inputs, outputs, and internal states. It is also possible to write Boolean expressions that describe the behavior of the sequential circuit.

A logic diagram is recognized as a clocked sequential circuit if it includes flip-flops. The flip-flops may be of any type and the logic diagram may or may not include combinational circuit gates.

I will discuss analysis process, along with different terms, with the help of specific example.

Example

An example of a clocked sequential circuit is shown in Fig. below. The circuit consists of two *D* flip-flops *A* and *B,* an input *x,* and an output *y.*

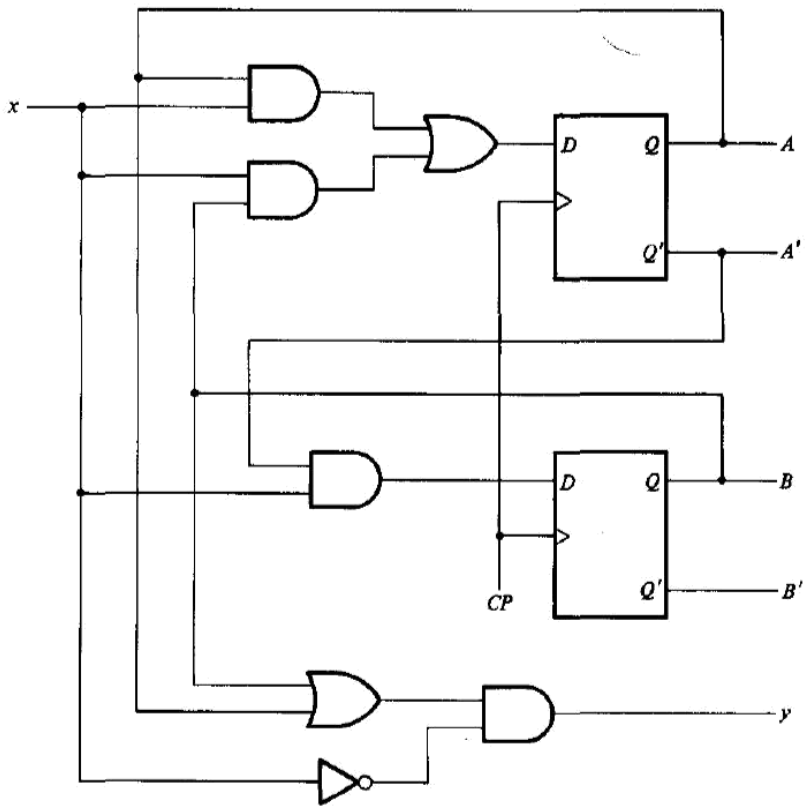


Fig: Example of sequential circuit

State Equations

A state equation is an algebraic expression that specifies the condition for a flip-flop state transition. The left side of the equation denotes the next state of the flip-flop and the right side of the equation is a Boolean expression that specifies the present state and input conditions that make the next state equal to 1.

In above example, *D* inputs determine the flip-flop's next state, so it is possible to write a set of next-state equations for the circuit:

$$A(t + 1) = A(t)x(t) + B(t)x(t)$$
$$B(t + 1) = A'(t)x(t)$$

Can be written conveniently as:

$$A(t + 1) = Ax + Bx$$
$$B(t + 1) = A'x$$

Similarly, the present-state value of the output *y* can be expressed algebraically as follows:

$$y(t) = [A(t) + B(t)]x'(t)$$

Removing the symbol *(t)* for the present state, we obtain the output Boolean function:

$$y = (A + B)x'$$

State table

The time sequence of inputs, outputs, and flip-flop states can be enumerated in a *state table.* The state table for the example circuit above is shown in the table below.

Present State		Input	Next State		Output
A	*B*	*x*	*A*	*B*	*y*
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0

→ The table consists of four sections:

Present state: shows the states of flip-flops *A* and *B* at any given time *t*

Input: gives a value of *x* for each possible present state

Next state: shows the states of the flip-flops one clock period later at time *t* + 1.

Output: gives the value of *y* for each present state.

→ The derivation of a state table consists of first listing all possible binary combinations of present state and inputs.

→ Next state and output column is derived from the **state equations**.

This table can alternatively be represented as:

	Next State		Output	
Present State	$x = 0$	$x = 1$	$x = 0$	$x = 1$
AB	*AB*	*AB*	*y*	*y*
00	00	01	0	0
01	00	11	1	0
10	00	10	1	0
11	00	10	1	0

State Diagram

The information available in a state table can be represented graphically in a **state diagram**. In this type of diagram, a state is represented by a circle, and the transition between states is indicated by directed lines connecting the circles.

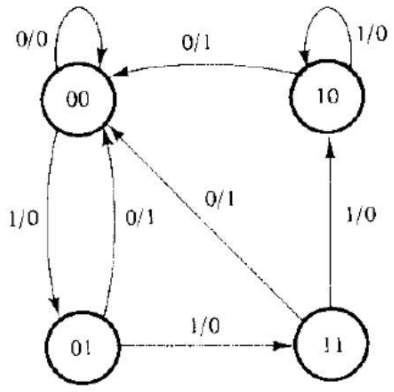


Fig: State diagram for our example

→ The binary number inside each circle identifies the state of the flip-flops.

→ The directed lines are labeled with two binary numbers separated by a slash viz. (input value/output value) during the present state. E.g. directed line from state 00 to 01 is labeled 1/0, meaning that when the sequential circuit is in the present state 00 and the input is 1, the output is 0. After a clock transition, the circuit goes to the next state 01.

→ A directed line connecting a circle with itself indicates that no change of state occurs.

→ There is no difference between a state table and a state diagram except in the manner of representation. The state table is easier to derive from a given logic diagram and the state diagram follows directly from the state table. The state diagram gives a pictorial view of state transitions and is the form suitable for human interpretation of the circuit operation.

State Reduction and assignment

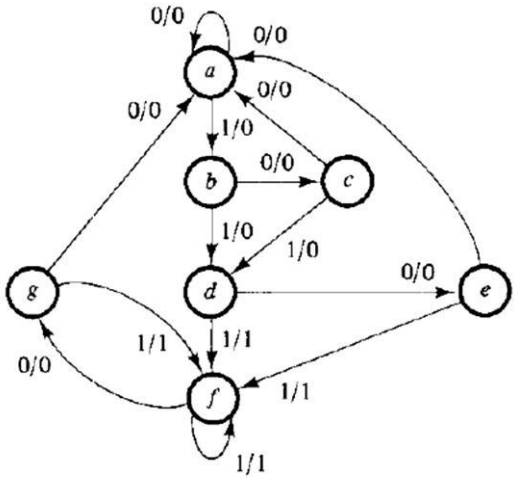
The analysis of sequential circuits starts from a circuit diagram and culminates in a state table or diagram. The design of a sequential circuit starts from a set of specifications and culminates in a logic diagram. Any design process must consider the problem of minimizing the cost of the final circuit (reduce the number of gates and flip-flops during the design).

State Reduction

The reduction of the number of flip-flops in a sequential circuit is referred to as the ***state-reduction*** problem. State-reduction algorithms are concerned with procedures for reducing the number of states in a state-table while keeping the external input-output requirements unchanged.

Example

Consider a sequential circuit with following specification. States marked inside the circles are denoted by letter symbols instead of by their binary values.



Consider the input sequence 01010110100 starting from the initial state *a.* Each input of 0 or 1 produces an output of 0 or 1 and causes the circuit to go to the next state. From the state diagram, we obtain the output and state sequence for the given input sequence as follows:

state	*a*	*a*	*b*	*c*	*d*	*e*	*f*	*f*	*g*	*f*	*g*	*a*
input	0	1	0	1	0	1	1	0	1	0	0	
output	0	0	0	0	0	1	1	0	1	0	0	

In each column, we have the present state, input value, and output value. The next state is written on top of the next column.

Algorithm: "Two states are said to be equivalent if, for each member of the set of inputs, they give exactly the same output and send the circuit either to the same state or to an equivalent state. When two states are equivalent, one of them can be removed without altering the input-output relationships."

- First, we need the state table (from state diagram above)

	Next State		Output	
Present State	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	*a*	*b*	0	0
b	*c*	*d*	0	0
c	*a*	*d*	0	0
d	*e*	*f*	0	1
e	*a*	*f*	0	1
f	*g*	*f*	0	1
g	*a*	*f*	0	1

- Look for two present states that go to the same next state and have the same output for both input combinations. States *g* and *e* are two such states: they both go to states *a* and *f* and have

outputs of 0 and 1 for $x = 0$ and $x = 1$, respectively. Therefore, states g and *e* are equivalent; one can be removed.

Present State	Next State $x = 0$	Next State $x = 1$	Output $x = 0$	Output $x = 1$
a	*a*	*b*	0	0
b	*c*	*d*	0	0
c	*a*	*d*	0	0
d	*e*	*f*	0	1
e	*a*	*f*	0	1
~~*f*~~	~~*e*~~	*f*	0	1
~~*g*~~	*a*	*f*	0	1

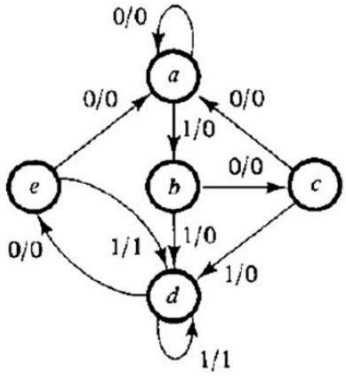
→ The row with present state g is crossed out and state *g* is replaced by state *e* each time it occurs in the next-state columns.

→ Present state f now has next states *e* and f and outputs 0 and 1 for $x = 0$ and $x = 1$, respectively.

→ The same next states and outputs appear in the row with present state *d*. Therefore, states f and d are equivalent; state f can be removed and replaced by *d*

- Final reduced table and state diagram for the reduced table consists of only five states.

Present State	Next state $x = 0$	Next state $x = 1$	Output $x = 0$	Output $x = 1$
a	*a*	*b*	0	0
b	*c*	*d*	0	0
c	*a*	*d*	0	0
d	*e*	*d*	0	1
e	*a*	*d*	0	1



State Assignment

The cost of the combinational-circuit part of a sequential circuit can be reduced by using the known simplification methods for combinational circuits. However, there is another factor, known as the ***state-assignment*** problem that comes into play in minimizing the combinational gates. State-assignment procedures are concerned with methods for assigning binary values to states in such a way as to reduce the cost of the combinational circuit that drives the flip-flops.

Consider a example shown in state reduction, 3 examples of possible binary assignments are shown in Table below for the five states of the reduced table. Assignment 1 is a straight binary assignment for the sequence of states from *a* through *e*. The other two assignments are chosen arbitrarily. In fact, there are 140 different distinct assignments for this circuit.

The binary form of the state table is used to derive the combinational-circuit part of the sequential circuit. The complexity of the combinational circuit obtained depends on the binary state assignment chosen.

State	Assignment 1	Assignment 2	Assignment 3
a	001	000	000
b	010	010	100
c	011	011	010
d	100	101	101
e	101	111	011

Fig: 3 possible binary sate assignments

Present state	Next State $x = 0$	Next State $x = 1$	Output $x = 0$	Output $x = 1$
001	001	010	0	0
010	011	100	0	0
011	001	100	0	0
100	101	100	0	1
101	001	100	0	1

Fig: Reduced state table with binary assignment 1

Excitation Tables

A table that lists required inputs for a given change of state (Present to next-state) is called an *excitation table.*

Flip-Flop Excitation Tables

$Q(t)$	$Q(t+1)$	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

(a) *RS*

$Q(t)$	$Q(t+1)$	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

(b) *JK*

$Q(t)$	$Q(t+1)$	D
0	0	0
0	1	1
1	0	0
1	1	1

(c) *D*

$Q(t)$	$Q(t+1)$	T
0	0	0
0	1	1
1	0	1
1	1	0

(b) *T*

The required input conditions for each of the four transitions are derived from the information available in the characteristic table. The symbol X in the tables represents a don't-care condition, i.e., it does not matter whether the input is 1 or 0.

Design Procedure

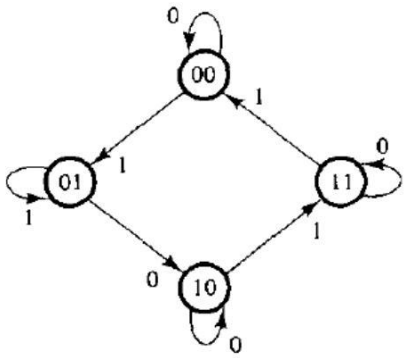
The design of a clocked sequential circuit starts from a set of specifications (state table) and ends in a logic diagram or a list of Boolean functions from which the logic diagram can be obtained.

Procedure:

The procedure can be summarized by a list of consecutive recommended steps:

(1) State the **word description of the circuit behavior**. It may be a state diagram, a timing diagram, or other pertinent information.
(2) From the given information about the circuit, obtain the **state table**.
(3) Apply **state-reduction** methods if the sequential circuit can be characterized by input-output relationships independent of the number of states.
(4) **Assign binary values** to each state if the state table obtained in step 2 or 3 contains letter symbols.
(5) Determine the **number of flip-flops** needed and assign a letter symbol to each.
(6) Choose the **type of flip-flop** to be used.
(7) From the state table, derive the **circuit excitation and output tables**.
(8) Using the map or any other simplification method, derive **the circuit output functions and the flip-flop input functions**.
(9) Draw the **logic diagram**.

Example: Design Procedure



Procedure step: (1) and (2)

- The state diagram consists of four states with binary values already assigned.
- Directed lines contain single binary digit without a slash, we conclude that there is one input variable and no output variables. (The state of the flip-flops may be considered the outputs of the circuit).
- The two flip-flops needed to represent the four states are designated *A* and *B.*
- The input variable is designated *x.*

Fig: State-diagram for design example

Present State		Next State			
		$x = 0$		$x = 1$	
A	*B*	*A*	*B*	*A*	*B*
0	0	0	0	0	1
0	1	1	0	0	1
1	0	1	0	1	1
1	1	1	1	0	0

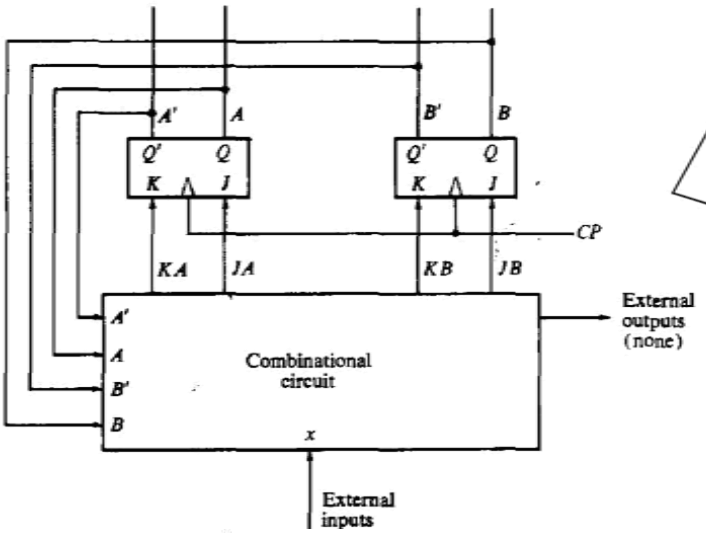
Procedure step: (3)
The state table for this circuit, derived from the state diagram. Note that there is no output section for this circuit.

Fig: State Table

Inputs of Combinational Circuit			Next State		Outputs of Combinational Circuit			
Present State		Input			Flip-Flop Inputs			
A	*B*	*x*	*A*	*B*	*JA*	*KA*	*JB*	*KB*
0	0	0	0	0	0	*X*	0	*X*
0	0	1	0	1	0	*X*	1	*X*
0	1	0	1	0	1	*X*	*X*	1
0	1	1	0	1	0	*X*	*X*	0
1	0	0	1	0	*X*	0	0	*X*
1	0	1	1	1	*X*	0	1	*X*
1	1	0	1	1	*X*	0	*X*	0
1	1	1	0	0	*X*	1	*X*	1

In the derivation of the excitation table, present state and input variables are arranged in the form of a truth table.

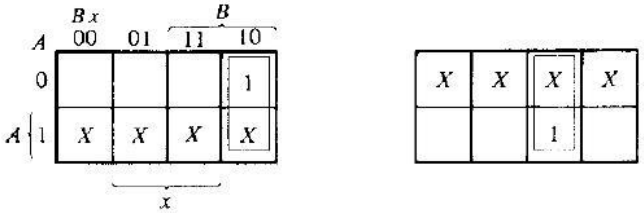
- JK type is used here. (PS (6))
- Since *JK* flip-flops are used we need columns for the J and *K* inputs of flip-flops *A* (denoted by *JA* and *KA)* and *B* (denoted by *JB* and *KB).*



→ Shows the two *JK* flip-flops needed for the circuit and a box to represent the combinational circuit.

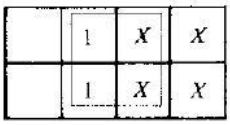
→ From the block diagram, it is clear that the outputs of the combinational circuit go to flip-flop inputs and external outputs (if specified).

→ The inputs to the combinational circuit are the external inputs and the present state values of the flip-flops.

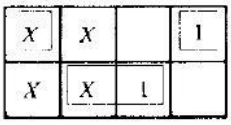


$JA = Bx'$

$KA = Bx$

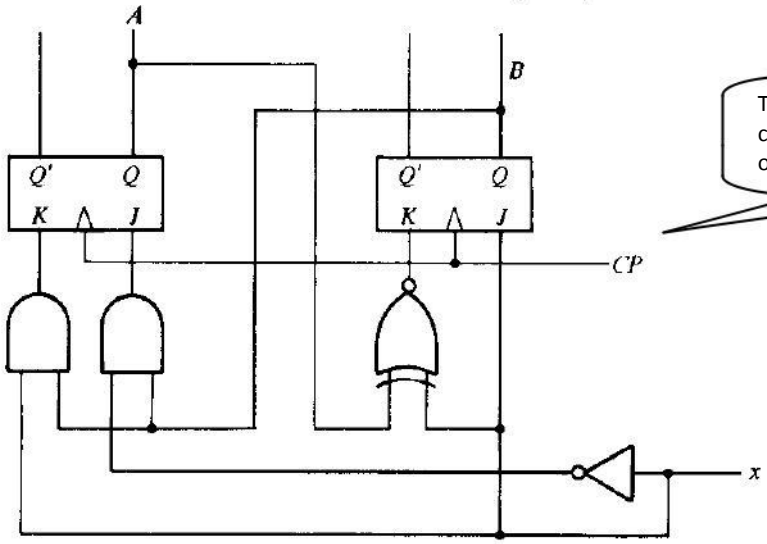


$JB = x$



$KB = (A \oplus x)'$

- Derivation of simplified Boolean functions for the combinational circuit.
- The information from the truth table is transferred into the maps.
- The inputs are the variables *A, B,* and *x;* the outputs are the variables *JA, KA, JB,* and KB.



The logic diagram is drawn in by side and consists of two flip-flops, two AND gates, one exclusive-NOR gate, and one inverter.

Unit 7
Registers, Counters and Memory units

A circuit with only flip-flops is considered a sequential circuit even in the absence of combinational gates. Certain MSI circuits that include flip-flops are classified by the operation that they perform rather than the name sequential circuit. Two such MSI components are **registers** and **counters**.

Registers

- A register is a group of binary cells suitable for holding binary information. A group of flip-flops constitutes a register.
- An *n-bit* register has a group of *n* flip-flops and is capable of storing any binary information containing *n* bits.
- In addition to the flip-flops, a register may have combinational gates that perform certain data-processing tasks.

Various types of registers are available in MSI circuits. The simplest possible register is one that consists of only flip-flops without any external gates. Following fig. shows such a register constructed with four D-type flip-flops and a common clock-pulse input.

- The clock pulse input, CP, enables all flip-flops, so that the information presently available at the four inputs can be transferred into the 4-bit register.
- The four outputs can be sampled to obtain the information presently stored in the register.

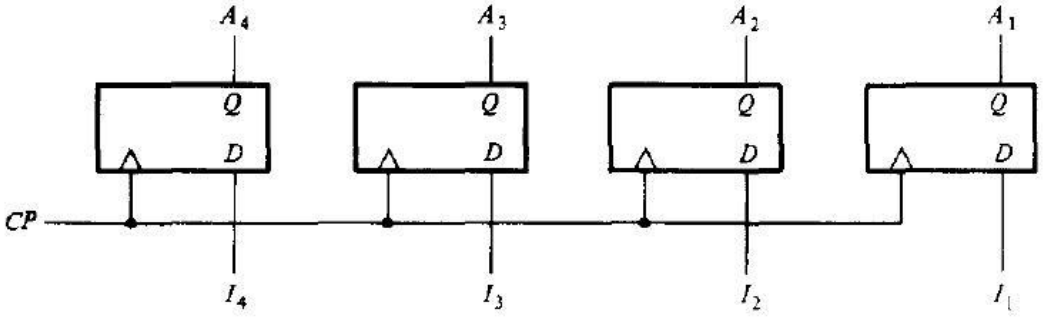


Fig: 4-bit register

Register with parallel load

The transfer of new information into a register is referred to as ***loading*** the register. If all the bits of the register are loaded simultaneously with a single clock pulse, we say that the loading is done in parallel. A pulse applied to the *CP* input of the register of Fig. above will load all four inputs in parallel. When *CP* goes to 1, the input information is loaded into the register. If *CP* remains at 0, the content of the register is not changed. Note that the change of state in the outputs occurs at the positive edge of the pulse.

Shift Registers

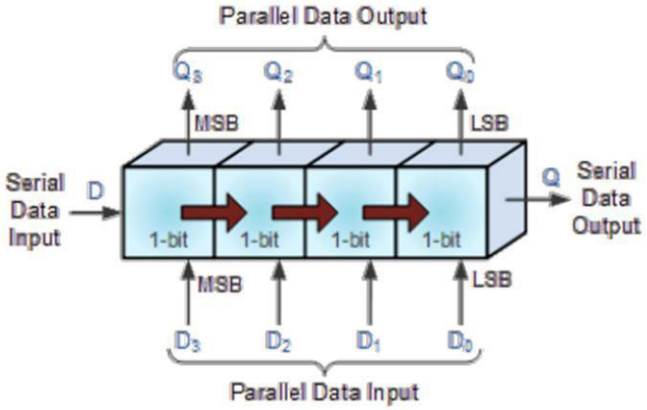
- A register capable of shifting its binary information either to the right or to the left is called a ***shift register***. The logical configuration of a shift register consists of a chain of flip-flops connected in cascade, with the output of one flip-flop connected to the input of the next flip-flop. All flip-flops receive a common clock pulse that causes the shift from one stage to the next.
- The Shift Register is used for data storage or data movement and are used in calculators or computers to store data such as two binary numbers before they are added together, or to convert the data from either a serial to parallel or parallel to serial format. The individual data

latches that make up a single shift register are all driven by a common clock (Clk) signal making them synchronous devices. Shift register IC's are generally provided with a ***clear*** or ***reset*** connection so that they can be "SET" or "RESET" as required.

Generally, shift registers operate in one of **four different modes** with the basic movement of data through a shift register being:

- **Serial-in to Parallel-out (SIPO)** - the register is loaded with serial data, one bit at a time, with the stored data being available in parallel form.
- **Serial-in to Serial-out (SISO)** - the data is shifted serially "IN" and "OUT" of the register, one bit at a time in either a left or right direction under clock control.
- **Parallel-in to Serial-out (PISO)** - the parallel data is loaded into the register simultaneously and is shifted out of the register serially one bit at a time under clock control.
- **Parallel-in to parallel-out (PIPO)** - the parallel data is loaded simultaneously into the register, and transferred together to their respective outputs by the same clock pulse.

The effect of data movement from left to right through a shift register can be presented graphically as:



Also, the directional movement of the data through a shift register can be either to the left, (left shifting) to the right, (right shifting) left-in but right-out, (rotation) or both left and right shifting within the same register thereby making it *bidirectional*.

Serial-in to Parallel-out (SIPO)

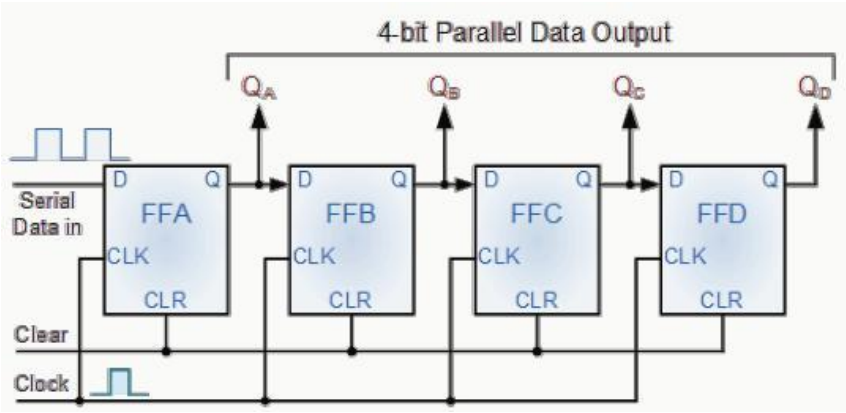
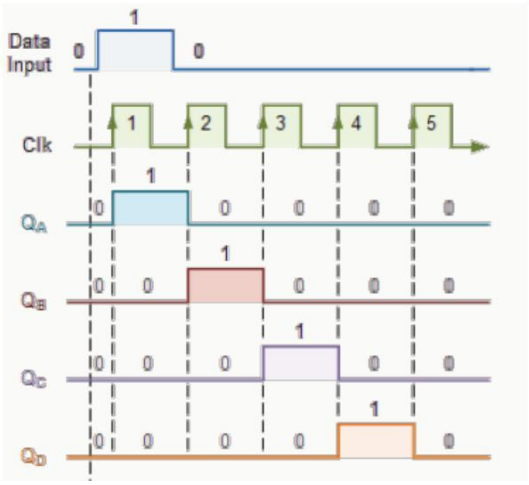


Fig: 4-bit Serial-in to Parallel-out Shift Register

Operation

- Let's assume that all the flip-flops (FFA to FFD) have just been RESET (CLEAR input) and that all the outputs Q_A to Q_D are at logic level "0" i.e., no parallel data output.
- If a logic "1" is connected to the DATA input pin of FFA then on the first clock pulse the output of FFA and therefore the resulting Q_A will be set HIGH to logic "1" with all the other outputs still remaining LOW at logic "0".
- Assume now that the DATA input pin of FFA has returned LOW again to logic "0" giving us one data pulse or 0-1-0.
- The second clock pulse will change the output of FFA to logic "0" and the output of FFB and Q_B HIGH to logic "1" as its input D has the logic "1" level on it from Q_A. The logic "1" has now moved or been "shifted" one place along the register to the right as it is now at Q_A. When the third clock pulse arrives this logic "1" value moves to the output of FFC (Q) and so on until the arrival of the fifth clock pulse which sets all the outputs Q_A to Q_D back again to logic level "0" because the input to FFA has remained constant at logic level "0".
- The effect of each clock pulse is to shift the data contents of each stage one place to the right, and this is shown in the following table until the complete data value of 0-0-0-1 is stored in the register.

Clock Pulse No	QA	QB	QC	QD
0	0	0	0	0
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1
5	0	0	0	0



→ Commonly available SIPO IC's include the standard 8-bit 74LS164 or the 74LS594.

Serial-in to Serial-out (SISO)

This shift register is very similar to the SIPO above, except were before the data was read directly in a parallel form from the outputs Q_A to Q_D, this time the data is allowed to flow straight through the register and out of the other end. Since there is only one output, the DATA leaves the shift register one bit at a time in a serial pattern, hence the name Serial-in to Serial-Out Shift Register or SISO. The SISO

shift register is one of the simplest of the four configurations as it has only three connections, the serial input (SI) which determines what enters the left hand flip-flop, the serial output (SO) which is taken from the output of the right hand flip-flop and the sequencing clock signal (Clk). The logic circuit diagram below shows a generalized serial-in serial-out shift register.

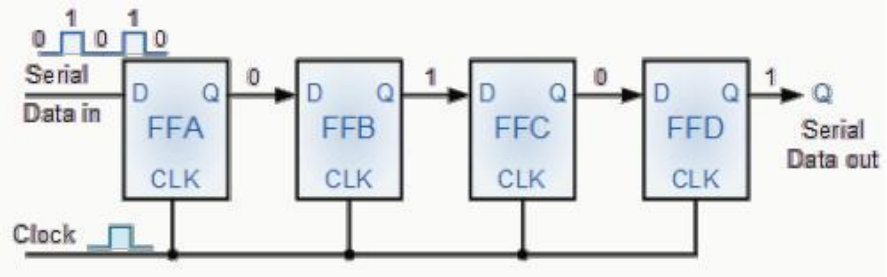


Fig: 4-bit Serial-in to Serial-out Shift Register

What's the point of a SISO shift register if the output data is exactly the same as the input data?

→ Well this type of Shift Register also acts as a temporary storage device or as a time delay device for the data, with the amount of time delay being controlled by the number of stages in the register, 4, 8, 16 etc or by varying the application of the clock pulses.

→ Commonly available IC's include the 74HC595 8-bit Serial-in/Serial-out Shift Register all with 3-state outputs.

Parallel-in to Serial-out (PISO)

The Parallel-in to Serial-out shift register acts in the opposite way to the serial-in to parallel-out one above. The data is loaded into the register in a parallel format i.e. all the data bits enter their inputs simultaneously, to the parallel input pins P_A to P_D of the register. The data is then read out sequentially in the normal shift-right mode from the register at Q representing the data present at P_A to P_D. This data is outputted one bit at a time on each clock cycle in a serial format. It is important to note that with this system a clock pulse is not required to parallel load the register as it is already present, but four clock pulses are required to unload the data.

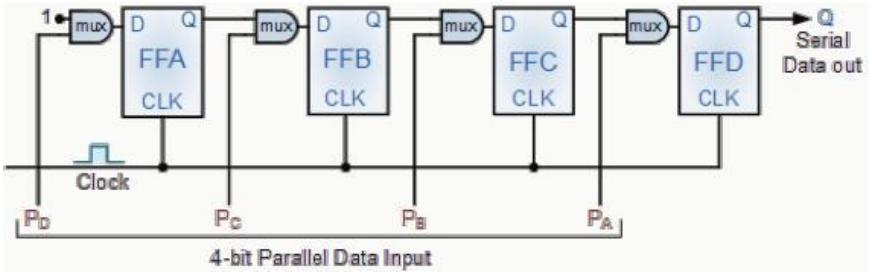


Fig: 4-bit Parallel-in to Serial-out Shift Register

→ **Advantage**: As this type of shift register converts parallel data, such as an 8-bit data word into serial format, it can be used to multiplex many different input lines into a single serial DATA stream which can be sent directly to a computer or transmitted over a communications line.

→ Commonly available IC's include the 74HC166 8-bit Parallel-in/Serial-out Shift Registers.

Parallel-in to Parallel-out (PIPO)

The final mode of operation is the Parallel-in to Parallel-out Shift Register. This type of register also acts as a temporary storage device or as a time delay device similar to the SISO configuration above. The data is presented in a parallel format to the parallel input pins P_A to P_D and then transferred together directly to their respective output pins Q_A to Q_D by the same clock pulse. Then one clock pulse loads and unloads the register. This arrangement for parallel loading and unloading is shown below.

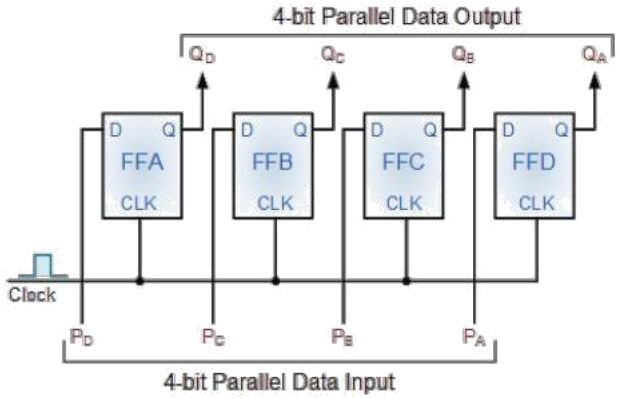


Fig: 4-bit Parallel-in to Parallel-out Shift Register

The PIPO shift register is the simplest of the four configurations as it has only three connections, the parallel input (PI) which determines what enters the flip-flop, the parallel output (PO) and the sequencing clock signal (Clk). Similar to the Serial-in to Serial-out shift register, this type of register also acts as a temporary storage device or as a time delay device, with the amount of time delay being varied by the frequency of the clock pulses. Also, in this type of register there are no interconnections between the individual flip-flops since no serial shifting of the data is required.

Ripple Counters (Asynchronous Counters)

- MSI counters come in two categories: ripple counters and synchronous counters.
- In a **ripple counter (Asynchronous Counter)**; flip-flop output transition serves as a source for triggering other flip-flops. In other words, the *CP* inputs of all flip-flops (except the first) are triggered not by the incoming pulses, but rather by the transition that occurs in other flip-flops.
- **Synchronous counter**, the input pulses are applied to all *CP* inputs of all flip-flops. The change of state of a particular flip-flop is dependent on the present state of other flip-flops.

Binary Ripple Counter

A binary ripple counter consists of a series connection of complementing flip-flops *(T* or *JK* type), with the output of each flip-flop connected to the *CP* input of the next higher-order flip-flop. The flip-flop holding the least significant bit receives the incoming count pulses. The diagram of a 4-bit binary ripple counter is shown in Fig. below. All *J* and *K* inputs are equal to 1.

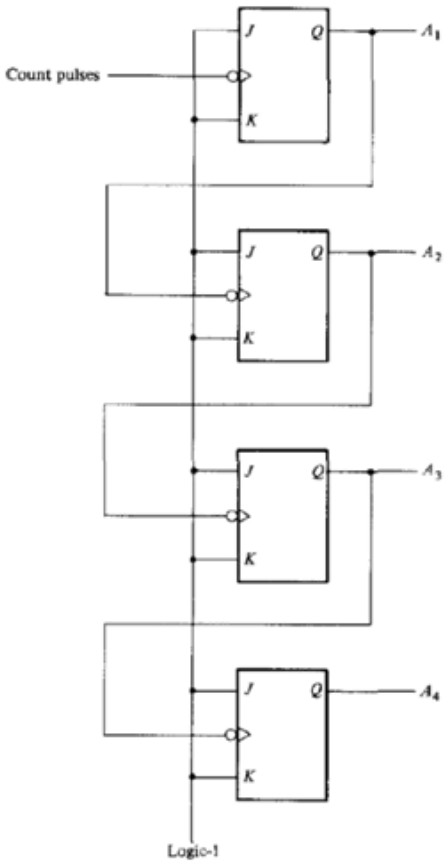


Fig: 4-bit binary ripple counter

All *J* and *K* inputs are equal to 1. The small circle in the *CP* input indicates that the flip-flop complements during a negative-going transition or when the output to which it is connected goes from 1 to 0.

To understand the operation of the binary counter, refer to its count sequence given in Table.

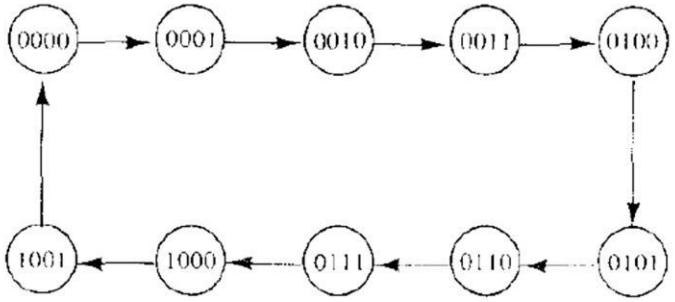
- It is obvious that the lowest-order bit A_1 must be complemented with each count pulse. Every time A_1 goes from 1 to 0, it complements A_2. Every time A_2 goes from 1 to 0, it complements A_3, and so on.
- For example: take the transition from count 0111 to 1000. The arrows in the table emphasize the transitions in this case. A_1 is complemented with the count pulse. Since A_1 goes from 1 to 0, it triggers A_2 and complements it. As a result, A_2 goes from 1 to 0, which in turn complements A_3. A_3 now goes from 1 to 0, which complements A_4. The output transition of A_4, if connected to a next stage, will not trigger the next flip-flop since it goes from 0 to 1. The flip-flops change one at a time in rapid succession, and the signal propagates through the counter in a *ripple* fashion.

Count Sequence

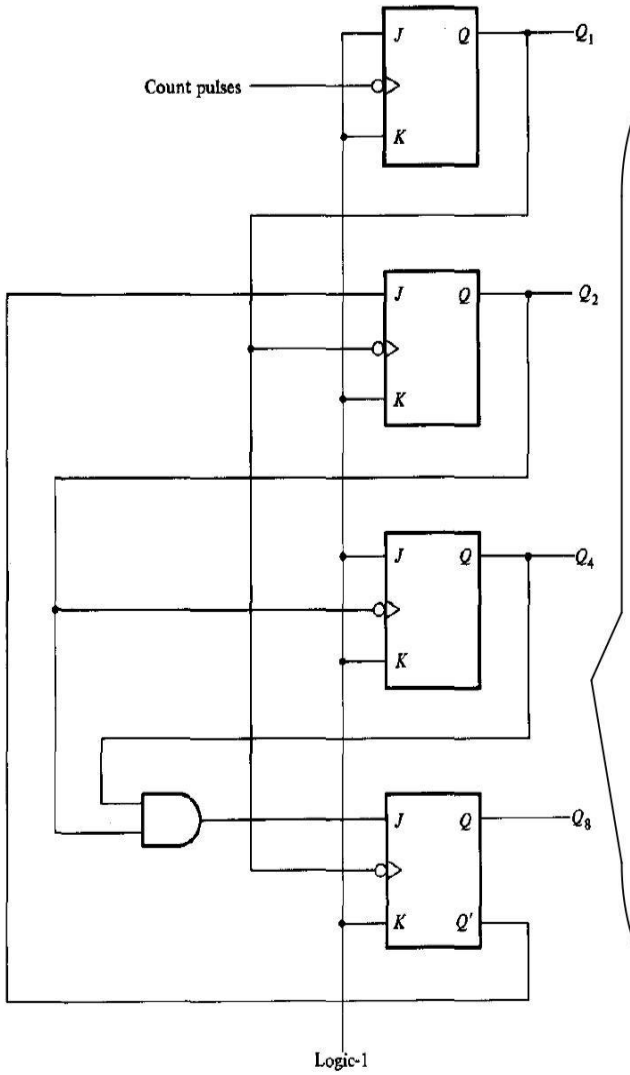
A_4	A_3	A_2	A_1	Conditions for Complementing Flip-Flops	
0	0	0	0	Complement A_1	
0	0	0	1	Complement A_1	A_1 will go from 1 to 0 and complement A_2
0	0	1	0	Complement A_1	
0	0	1	1	Complement A_1	A_1 will go from 1 to 0 and complement A_2; A_2 will go from 1 to 0 and complement A_3
0	1	0	0	Complement A_1	
0	1	0	1	Complement A_1	A_1 will go from 1 to 0 and complement A_2
0	1	1	0	Complement A_1	
0	1	1	1	Complement A_1	A_1 will go from 1 to 0 and complement A_2; A_2 will go from 1 to 0 and complement A_3; A_3 will go from 1 to 0 and complement A_4
1	0	0	0	and so on . . .	

BCD Ripple Counter (Decade Counter)

A decimal counter follows a sequence of ten states and returns to 0 after the count of 9. Such a counter must have at least four flip-flops to represent each decimal digit, since a decimal digit is represented by a binary code with at least four bits. The sequence of states in a decimal counter is dictated by the binary code used to represent a decimal digit.



If BCD is used, the sequence of states is as shown in the state diagram. This is similar to a binary counter, except that the state after 1001 (code for decimal digit 9) is 0000 (code for decimal digit 0).



- The four outputs are designated by the letter symbol Q with a numeric subscript equal to the binary weight of the corresponding bit in the BCD code.

• The flip-flops trigger on the negative edge. • A ripple counter is an asynchronous sequential circuit and cannot be described by Boolean equations developed for describing clocked sequential circuits. Signals that affect the flip-flop transition depend on the order in which they change from 1 to 0.

- **Operation**:

When CP input goes from 1 to 0, the flip-flop is set if J = 1, is cleared if K = 1, is complemented if J = K = 1, and is left unchanged if J = K = 0. The following are the conditions for each flip-flop state transition:

1. Q_1 is complemented on the negative edge of every count pulse.
2. Q_2 is complemented if Q_8 = 0 and Q_1 goes from 1 to 0. Q_2 is cleared if Q_8 = *1* and Q_1 goes from 1 to 0.
3. Q_4 is complemented when Q_2 goes from 1 to 0.
4. Q_8 is complemented when Q_4 Q_2 = 11 and Q_1 goes from 1 to 0. Q_8 is cleared if either Q_4 or Q_2 is 0 and Q_1 goes from 1 to 0.

Fig: BCD ripple counter

BCD Counter above counts from 0 to 9. To count in decimal from 0 to 99, we need a two-decade counter. To count from 0 to 999, we need a three-decade counter.

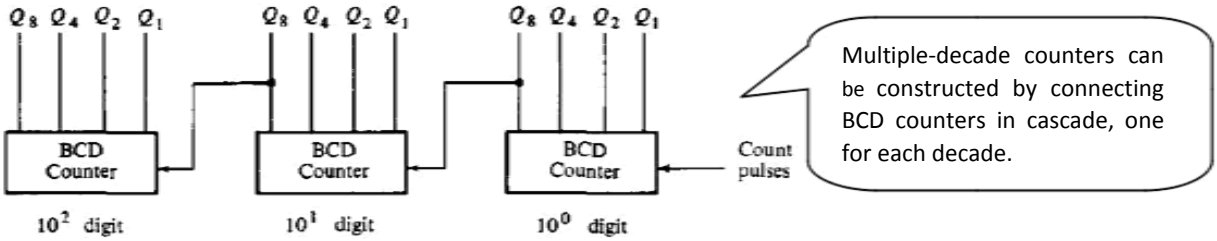
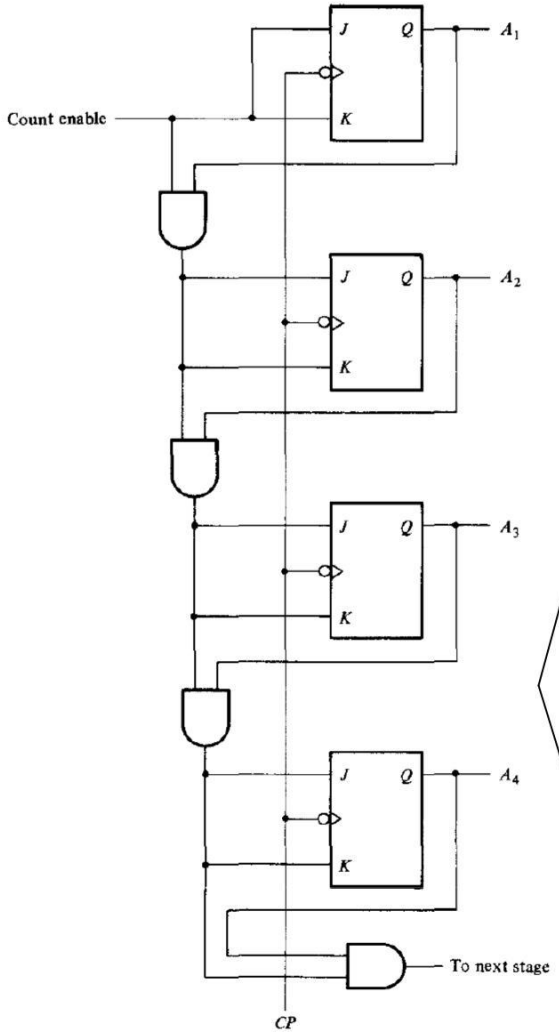


Fig: Block diagram of a three-decade decimal BCD counter

Synchronous Counters

Synchronous counters are distinguished from ripple counters in that clock pulses are applied to the *CP* inputs of *all* flip-flops. The common pulse triggers all the flip-flops simultaneously, rather than one at a time in succession as in a ripple counter. The decision whether a flip-flop is to be complemented or not is determined from the values of the *J* and *K* inputs at the time of the pulse. If *J* = *K* = 0, the flip-flop remains unchanged. If *J* = *K* = 1, the flip-flop complements.

Binary Counter



- The design of synchronous binary counters is so simple that there is no need to go through a rigorous sequential-logic design process. In a synchronous binary counter, the flip-flop in the lowest-order position is complemented with every pulse. This means that it's *J* and *K* inputs must be maintained at logic-1. A flip-flop in any other position is complemented with a pulse provided all the bits in the lower-order positions are equal to 1, because the lower-order bits (when all 1's) will change to 0's on the next count pulse.
- Synchronous binary counters have a regular pattern and can easily be constructed with complementing flip-flops and gates. The regular pattern can be clearly seen from the 4-bit counter depicted in Fig by side.
- The *CP* terminals of all flip-flops are connected to a common clock-pulse source. The first stage A_1 has its *J* and *K* equal to 1 if the counter is enabled. The other *J* and *K* inputs are equal to 1 if all previous low-order bits are equal to 1 and the count is enabled. The chain of AND gates generates the required logic for the *J* and *K* inputs in each stage. The counter can be extended to any number of stages, with each stage having an additional flip-flop and an AND gate that gives an output of 1 if all previous flip-flop outputs are 1's.

Fig: 4-bit Synchronous Binary Counter

Binary Up-Down Counter

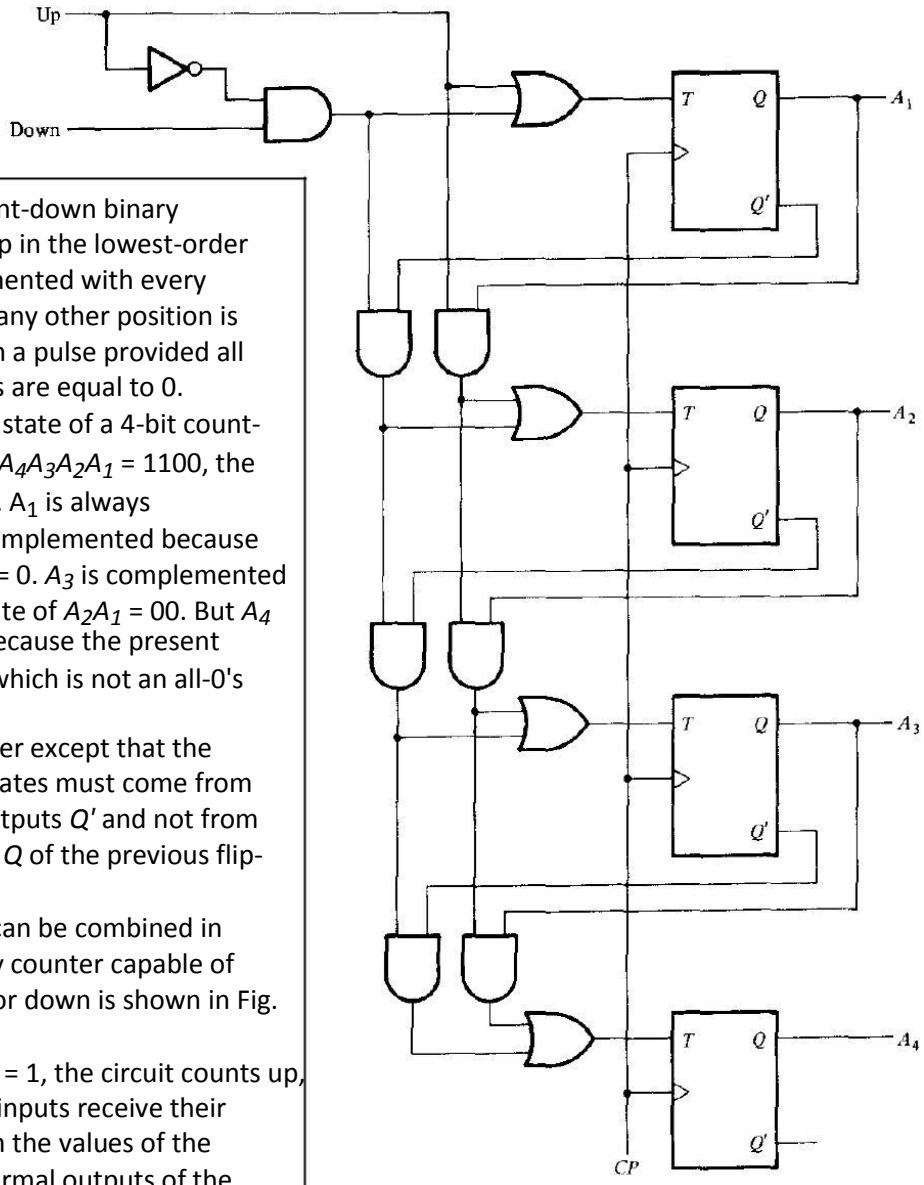


Fig: 4-bit up-down counter

- In a synchronous count-down binary counter, the flip-flop in the lowest-order position is complemented with every pulse. A flip-flop in any other position is complemented with a pulse provided all the lower-order bits are equal to 0.

Example: if the present state of a 4-bit count-down binary counter is $A_4A_3A_2A_1$ = 1100, the next count will be 1011. A_1 is always complemented. A_2 is complemented because the present state of A_1 = 0. A_3 is complemented because the present state of A_2A_1 = 00. But A_4 is not complemented because the present state of $A_3A_2A_1$ = 100, which is not an all-0's condition.

- Same as Binary counter except that the inputs to the AND gates must come from the complement outputs *Q'* and not from the normal outputs *Q* of the previous flip-flops.
- The two operations can be combined in one circuit. A binary counter capable of counting either up or down is shown in Fig. by side.
    - When **up** = 1, the circuit counts up, since the *T* inputs receive their signals from the values of the previous normal outputs of the flip-flops.
    - When **down** = 1 and **up** = 0, the circuit counts down

BCD Counter

A BCD counter counts in binary-coded decimal from 0000 to 1001 and back to 0000. Because of the return to 0 after a count of 9, a BCD counter does not have a regular pattern as in a straight binary count. To derive the circuit of a BCD synchronous counter, it is necessary to go through a design procedure discussed earlier.

The excitation for the *T* flip-flops is obtained from the present and next state conditions. An output *y* is also shown in the table. This output is equal to 1 when the counter present state is 1001. In this way, *y* can enable the count of the next-higher-order decade while the same pulse switches the present decade

from 1001 to 0000. The flip-flop input functions from the excitation table can be simplified by means of maps. The unused states for minterms 10 to 15 are taken as don't-care terms.

Present State				Next State				Output	Flip-Flop Inputs			
Q_8	Q_4	Q_2	Q_1	Q_8	Q_4	Q_2	Q_1	y	TQ_8	TQ_4	TQ_2	TQ_1
0	0	0	0	0	0	0	1	0	0	0	0	1
0	0	0	1	0	0	1	0	0	0	0	1	1
0	0	1	0	0	0	1	1	0	0	0	0	1
0	0	1	1	0	1	0	0	0	0	1	1	1
0	1	0	0	0	1	0	1	0	0	0	0	1
0	1	0	1	0	1	1	0	0	0	0	1	1
0	1	1	0	0	1	1	1	0	0	0	0	1
0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	0	0	1	0	0	0	0	1
1	0	0	1	0	0	0	0	1	1	0	0	1

The simplified functions are:

$$TQ_1 = 1$$
$$TQ_2 = Q_8'Q_1$$
$$TQ_4 = Q_2Q_1$$
$$TQ_8 = Q_8Q_1 + Q_4Q_2Q_1$$
$$y = Q_8Q_1$$

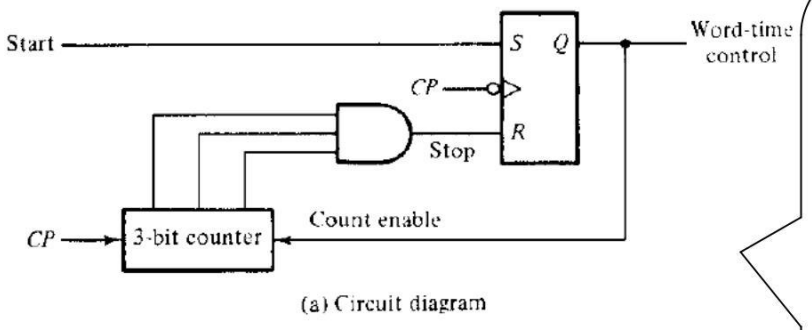
The circuit can be easily drawn with four *T* flip-flops, five AND gates, and one OR gate. Synchronous BCD counters can be cascaded to form a counter for decimal numbers of any length.

Timing Sequences

The sequences of operations in a digital system are specified by a control unit. The control unit that supervises the operations in a digital system would normally consist of timing signals that determine the time sequence in which the operations are executed. The timing sequences in the control unit can be easily generated by means of counters or shift registers.

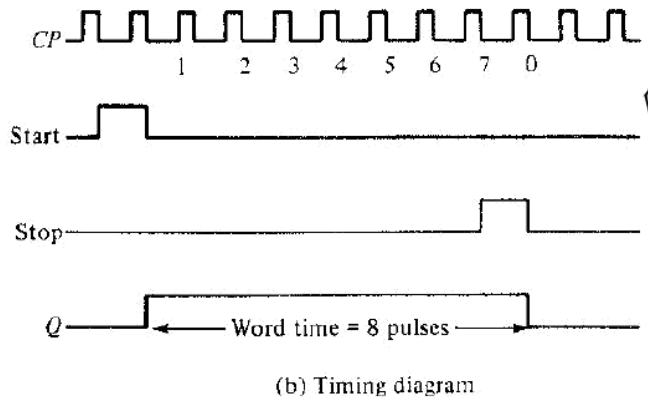
Word-Time Generation

The control unit in a serial computer must generate a *word-time* signal that stays on for a number of pulses equal to the number of bits in the shift registers. The word-time signal can be generated by means of a counter that counts the required number of pulses. Example:



(a) Circuit diagram

- Assume that the word-time signal to be generated must stay on for a period of eight clock pulses.
- Fig. shows a counter circuit that accomplishes this task.
- Initially, the 3-bit counter is cleared to 0. A start signal will set flip-flop Q. The output of this flip-flop supplies the word-time control and also enables the counter. After the count of eight pulses, the flip-flop is reset and Q goes to 0.



(b) Timing diagram

The timing diagram demonstrates the operation of the circuit. The start signal is synchronized with the clock and stays on for one clock-pulse period. After Q is set to 1, the counter starts counting the clock pulses. When the counter reaches the count of 7 (binary 111), it sends a stop signal to the reset input of the flip-flop. The stop signal becomes a 1 after the negative-edge transition of pulse 7. The next clock pulse switches the counter to the 000 state and also clears *Q*. Now the counter is disabled and the word-time signal stays at 0.

Timing Signals

The control unit in a digital system that operates in the parallel mode must generate timing signals that stay on for only one clock pulse period. Timing signals that control the sequence of operations in a digital system can be generated with a shift register or a counter with a decoder. A *ring counter* is a circular shift register with only one flip-flop being set at any particular time; all others are cleared. The single bit is shifted from one flip-flop to the other to produce the sequence of timing signals.

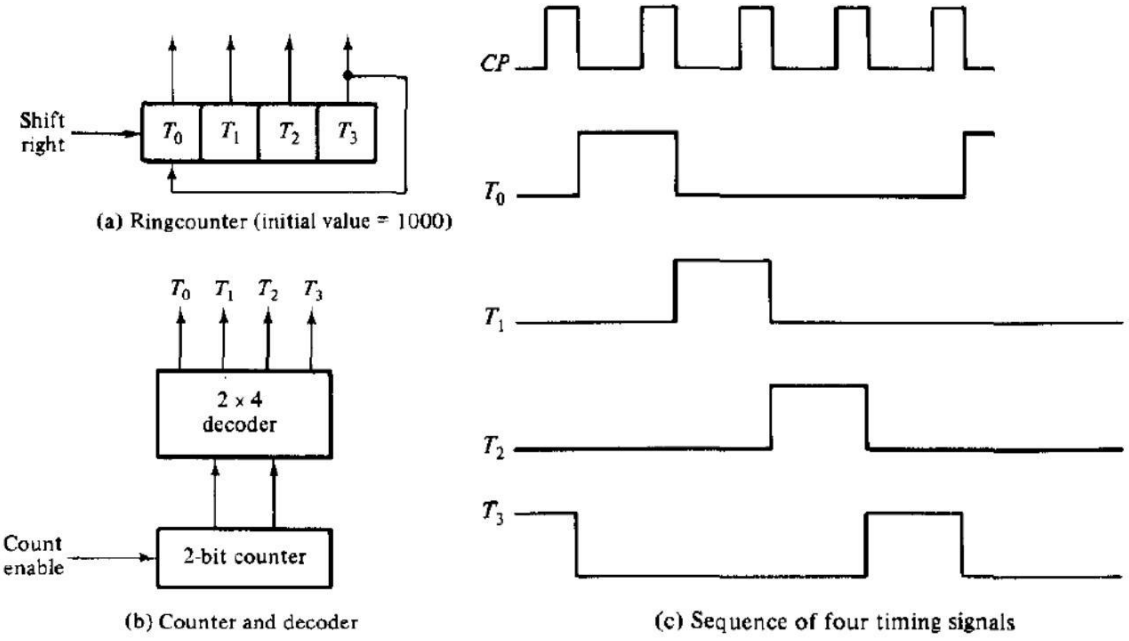


Fig: Generation of Timing Signals

- Figure (a) shows a 4-bit shift register connected as a ring counter. The initial value of the register is 1000, which produces the variable T_0. The single bit is shifted right with every clock pulse and circulates back from T_3 to T_0. Each flip-flop is in the 1 state once every four clock pulses and produces one of the four timing signals shown in Fig (c). Each output becomes a 1 after the negative-edge transition of a clock pulse and remains 1 during the next clock pulse.
- The timing signals can be generated also by continuously enabling a 2-bit counter that goes through four distinct states. The decoder shown in Fig. (b) decodes the four states of the counter and generates the required sequence of timing signals.

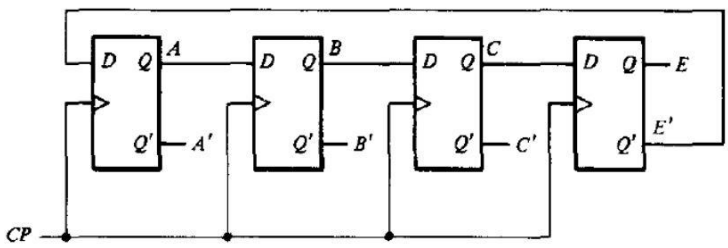
Point!

- To generate 2^n timing signals, we need either a shift register with 2^n flip-flops or an n-bit counter together with an n-to-2^n-line decoder. For example, 16 timing signals can be generated with a 16-bit shift register connected as a ring counter or with a 4-bit counter and a 4-to-16-line decoder.
- It is also possible to generate the timing signals with a combination of a shift register and a decoder. In this way, the number of flip-flops is less than a ring counter, and the decoder requires only 2-input gates. This combination is sometimes called a *Johnson counter. (Oh ya, we r studying it in what follows…* ☺ *)*

Johnson Counter

A **Johnson counter** is a k-bit **switch-tail ring counter** with *2k* decoding gates to provide outputs for *2k* timing signals.

- A **switch-tail ring counter** is a circular shift register with the complement output of the last flip-flop connected to the input of the first flip-flop.
- A k-bit **ring counter** circulates a single bit among the flip-flops to provide *k distinguishable* states.
- The number of states can be doubled if the shift register is connected as a *switch-tail* ring counter.



(a) Four-stage switch-tail ring counter

Sequence number	Flip-flop outputs A	B	C	E	AND gate required for output
1	0	0	0	0	$A'E'$
2	1	0	0	0	AB'
3	1	1	0	0	BC'
4	1	1	1	0	CE'
5	1	1	1	1	AE
6	0	1	1	1	$A'B$
7	0	0	1	1	$B'C$
8	0	0	0	1	$C'E$

(b) Count sequence and required decoding.

Fig: Construction of a Johnson counter

The eight AND gates listed in the table, when connected to the circuit will complete the construction of the Johnson counter. Since each gate is enabled during one particular state sequence, the outputs of the gates generate eight timing sequences in succession.

Operation:

The decoding of a k-bit switch-tail ring counter to obtain *2k* timing sequences follows a regular pattern. The all-0's state is decoded by taking the complement of the two extreme flip-flop outputs. The all-1's state is decoded by taking the normal outputs of the two extreme flip-flops. All other states are decoded from an adjacent 1, 0 or 0, 1 pattern in the sequence. For example, sequence 7 has an adjacent 0, 1 pattern in flip-flops *B and* C. The decoded output is then obtained by taking the complement of *B and* the normal output of C, or *B'* C.

- Johnson counters can be constructed for any number of timing sequences. The number of flip-flops needed is one-half the number of timing signals. The number of decoding gates is equal to the number of timing signals and only 2-input gates are employed.

Memory unit (Random Access Memory-RAM)

- A memory unit is a collection of storage cells together with associated circuits needed to transfer information in and out of the device. Memory cells can be accessed for information transfer to or from any desired random location and hence the name *random access memory,* abbreviated RAM.
- A memory unit stores binary information in groups of bits called ***words***. A word in memory is an entity of bits that move in and out of storage as a unit. A memory word is a group of 1's and 0's and may represent a number, an instruction, one or more alphanumeric characters, or any other binary-coded information.

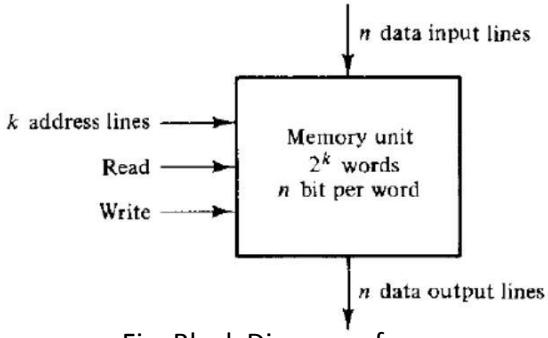


Fig: Block Diagram of a memory unit

- The communication between a memory and its environment is achieved through:
  - ***n* data input lines** : provide information to be stored in memory
  - ***n* data output lines**: supply the information coming out of memory.
  - ***k* address lines**: specify particular word chosen among the many available.
  - **two control inputs**: specify the direction of transfer desired
- Each word in memory is assigned an identification number, called an **address**, starting from 0 and continuing with 1, 2, 3, up to 2^k - I, where k is the number of address lines.
- Computer memories may range from 1024 words, requiring an address of 10 bits, to 2^{32} words, requiring 32 address bits.

Conventions for Memory storage:

K (kilo) = 2^{10}
M (mega) = 2^{20}
G (giga) = 2^{30}
Thus, 64K = 2^{16}, 2M = 2^{21}, and 4G = 2^{32}

Example: Memory unit with a capacity of 1K words of 16 bits each. Since 1K = 1024 = 210 and 16 bits constitute two bytes, we can say that the memory can accommodate 2048 = 2K bytes.

Memory address		Memory content
Binary	decimal	
0000000000	0	1011010101011101
0000000001	1	1010101110001001
0000000010	2	0000110101000110
⋮	⋮	⋮
1111111101	1021	1001110100010100
1111111110	1022	0000110100011110
1111111111	1023	1101111000100101

Fig: Possible content of 1024 x 16 memory

Each word contains 16 bits, which can be divided into two bytes. The words are recognized by their decimal address from 0 to 1023. The equivalent binary address consists of 10 bits. The first address is specified with ten 0's, and the last address is specified with ten 1's. A word in memory is elected by its binary address. When a word is read or written, the memory operates on all 16 bits as a single unit.

Memory address register (MAR): It is CPU register which contains the address of the memory words. If memory has k address lines, then MAR is of k-bits.

Memory Buffer Register (MBR): It contains the word-data pointed by the MAR.

Write and Read Operations

The two operations that a random-access memory can perform are the write and read operations. The write signal specifies a transfer-in operation and the read signal specifies a transfer-out operation. On accepting one of these control signals, the internal circuits inside the memory provide the desired function.

Write Operation: transferring a new word to be stored into memory

1. Transfer the binary address of the desired word to the address lines.
2. Transfer the data bits that must be stored in memory to the data input lines.
3. Activate the *write* input.

Read Operation: transferring a stored word out of memory

1. Transfer the binary address of the desired word to the address lines.
2. Activate the read input.

Commercial memory components available in IC chips sometimes provide the two control inputs for reading and writing in a somewhat different configuration. The memory operations that result from these control inputs are specified in Table below.

Control Inputs to Memory Chip

Memory Enable	Read/Write	Memory Operation
0	*X*	None
1	0	Write to selected word
1	1	Read from selected word

The memory enable (sometimes called the **chip select**) is used to enable the particular memory chip in a multichip implementation of a large memory. When the memory enable is inactive, memory chip is not selected and no operation is performed. When the memory enable input is active, the read/write input determines the operation to be performed.

IC memory (Binary Cell- BC)

The internal construction of a random-access memory of *m* words with *n* bits per word consists of *m* x *n* binary storage cells and associated decoding circuits for selecting individual words. The binary storage cell is the basic building block of a memory unit.

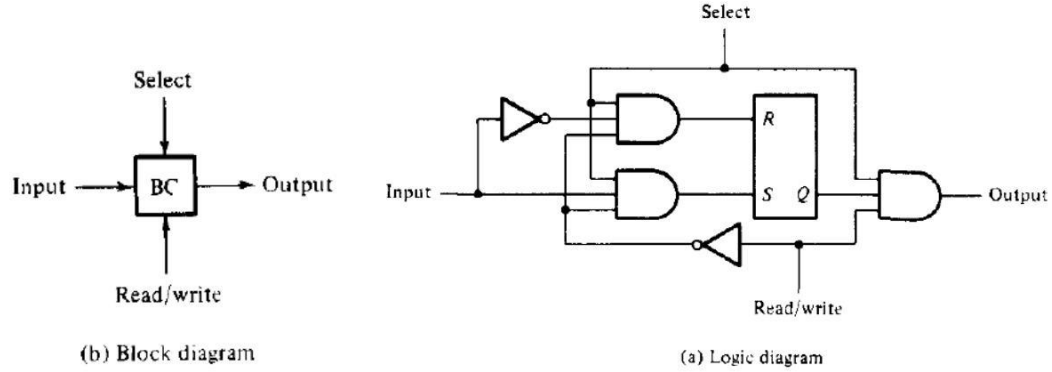


Fig: Memory Cell