Number System:

What is Number?

A *number* is a mathematical quantity, usually correlated in electronics to a physical quantity such as voltage, current, or resistance. There are many different types of numbers. Here are just a few types, for example:

- WHOLE NUMBERS: 1, 2, 3, 4, 5, 6, 7, 8, 9 . . .
- INTEGERS: -4, -3, -2, -1, 0, 1, 2, 3, 4 . . .
- IRRATIONAL NUMBERS: π (approx. 3.1415927), *e* (approx. 2.718281828), square root of any prime
- REAL NUMBERS: (All one-dimensional numerical values, negative and positive, including zero, whole, integer, and irrational numbers)
- COMPLEX NUMBERS: 3 - j4 , $34.5 \angle 20^{o}$

Different types of numbers find different application in the physical world. Whole numbers work well for counting discrete objects, such as the number of resistors in a circuit. Integers are needed when negative equivalents of whole numbers are required. Irrational numbers are numbers that cannot be exactly expressed as the ratio of two integers, and the ratio of a perfect circle's circumference to its diameter (π) is a good physical example of this. The non-integer quantities of voltage, current, and resistance that we're used to dealing with in DC circuits can be expressed as real numbers, in either fractional or decimal form. For AC circuit analysis, however, real numbers fail to capture the dual essence of magnitude and phase angle, and so we turn to the use of complex numbers in either rectangular or polar form.

If we are to use numbers to understand processes in the physical world, make scientific predictions, or balance our checkbooks, we must have a way of symbolically denoting them. In other words, we may know how much money we have in our checking account, but to keep record of it we need to have some system worked out to symbolize that quantity on paper, or in some other kind of form for record-keeping and tracking. There are two basic ways we can do this: analog and digital. With analog representation, the quantity is symbolized in a way that is infinitely divisible. With digital representation, the quantity is symbolized in a way that is discretely packaged.

Digital and Analog Numbers

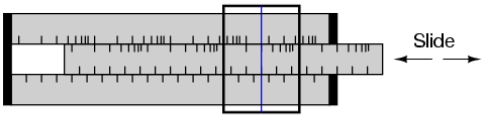
Thermometer is an example of an analog representation of a number. There is no real limit to how finely divided the height of that column can be made to symbolize the amount of money in the account. Changing the height of that column is something that can be done without changing the essential nature of what it is. Length is a physical quantity that can be

divided as small as you would like, with no practical limit. The slide rule is a mechanical device that uses the very same physical quantity -- length -- to represent numbers, and to help perform arithmetical operations with two or more numbers at a time. It, too, is an analog device. On the other hand, a *digital* representation of that same monetary figure, written with standard symbols (sometimes called ciphers), looks like this:

```
$35,955.38
```

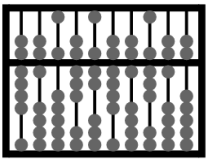
Unlike the "thermometer" poster with its red column, those symbolic characters above cannot be finely divided: that particular combination of ciphers stand for one quantity and one quantity only. If more money is added to the account (+ $40.12), different symbols must be used to represent the new balance ($35,995.50), or at least the same symbols arranged in different patterns. This is an example of digital representation. The counterpart to the slide rule (analog) is also a digital device: the abacus, with beads that are moved back and forth on rods to symbolize numerical quantities:

Slide rule *(an analog device)*



Numerical quantities are represented by the positioning of the slide.

Abacus *(a digital device)*



Numerical quantities are represented by the discrete positions of the beads.

Differences between Analog and Digital

ANALOG	DIGITAL
Intuitively understood	Requires training to interpret
Infinitely divisible	Discrete
Prone to errors of precision	Absolute precision

The digital representation is composed of individual, discrete symbols (decimal digits and abacus beads) necessarily means that it will be able to symbolize quantities in precise steps. On the other hand, an analog representation (such as a slide rule's length) is not composed of individual steps, but rather a continuous range of motion. The ability for a slide rule to characterize a numerical quantity to infinite resolution is a trade-off for imprecision. If a slide rule is bumped, an error will be introduced into the representation of the number that was "entered" into it. However, an abacus must be bumped much harder before its beads are completely dislodged from their places (sufficient to represent a different number).

Numeration System?

The Romans devised a system that was a substantial improvement over hash marks, because it used a variety of symbols (or ciphers) to represent increasingly large quantities. The notation for 1 is the capital letter I. The notation for 5 is the capital letter V. Other ciphers possess increasing values:

X = 10 L = 50 C = 100 D = 500 M = 1000

If a cipher is accompanied by another cipher of equal or lesser value to the immediate right of it, with no ciphers greater than that other cipher to the right of that other cipher, that other cipher's value is added to the total quantity. Thus, VIII symbolizes the number 8, and CLVIIsymbolizes the number 157. On the other hand, if a cipher is accompanied by another cipher of lesser value to the immediate left, that other cipher's value is subtracted from the first. Therefore, IV symbolizes the number 4 (V minus I), and CM symbolizes the number 900 (M minus C). You might have noticed that ending credit sequences for most motion pictures contain a notice for the date of production, in Roman numerals. For the year 1987, it would read: MCMLXXXVII. Let's break this numeral down into its constituent parts, from left to right:

M = 1000 + CM = 900 + L = 50 + XXX = 30 + V = 5 + II = 2

Large numbers are very difficult to denote this way, and the left vs. right / subtraction vs. addition of values can be very confusing, too. Another major problem with this system is that there is no provision for representing the number zero or negative numbers, both very important concepts in mathematics. Roman culture, however, was more pragmatic with respect to mathematics than most, choosing only to develop their numeration system as far as it was necessary for use in daily life.

We owe one of the most important ideas in numeration to the ancient Babylonians, who were the first (as far as we know) to develop the concept of cipher position, or place value, in representing larger numbers. Instead of inventing new ciphers to represent larger numbers, as the Romans did, they re-used the same ciphers, placing them in different positions from right to left. Our own decimal numeration system uses this concept, with only ten ciphers (0, 1, 2, 3, 4, 5, 6, 7, 8, and 9) used in "weighted" positions to represent very large and very small numbers.

Each cipher represents an integer quantity, and each place from right to left in the notation represents a multiplying constant, or weight, for each integer quantity. For example, if we see the decimal notation "1206", we known that this may be broken down into its constituent weight-products as such:

$1206 = 1000 + 200 + 6$

$1206 = (1 \times 1000) + (2 \times 100) + (0 \times 10) + (6 \times 1)$

Each cipher is called a digit in the decimal numeration system, and each weight, or place value, is ten times that of the one to the immediate right. So, we have a ones place, a tens place, a hundreds place, a thousands place, and so on, working from right to left.

Right about now, you're probably wondering why I'm laboring to describe the obvious. Who needs to be told how decimal numeration works, after you've studied math as advanced as algebra and trigonometry? The reason is to better understand other numeration systems, by first knowing the how's and why's of the one you're already used to.

The decimal numeration system uses ten ciphers, and place-weights that are multiples of ten. What if we made a numeration system with the same strategy of weighted places, except with fewer or more ciphers?

The binary numeration system is such a system. Instead of ten different cipher symbols, with each weight constant being ten times the one before it, we only have two cipher symbols, and each weight constant is twice as much as the one before it. The two allowable cipher symbols for the binary system of numeration are "1" and "0," and these ciphers are arranged right-to-left in doubling values of weight. The rightmost place is the ones place, just as with decimal notation. Proceeding to the left, we have the twos place, the fours place, the eights place, thesixteens place, and so on. For example, the following binary number can be expressed, just like the decimal number 1206, as a sum of each cipher value times its respective weight constant:

11010 = 2 + 8 + 16 = 26

11010 = (1 x 16) + (1 x 8) + (0 x 4) + (1 x 2) + (0 x 1)

This can get quite confusing, as I've written a number with binary numeration (11010), and then shown its place values and total in standard, decimal numeration form (16 + 8 + 2 = 26). In the above example, we're mixing two different kinds of numerical notation. To avoid unnecessary confusion, we have to denote which form of numeration we're using when we write (or type!). Typically, this is done in subscript form, with a "2" for binary and a "10" for decimal, so the binary number 11010_2 is equal to the decimal number 26_{10}.

The subscripts are not mathematical operation symbols like superscripts (exponents) are. All they do is indicate what system of numeration we're using when we write these symbols for other people to read. If you see "310", all this means is the number three written using decimalnumeration. However, if you see "3^{10}", this means something completely different: three to the tenth power (59,049). As usual, if no subscript is shown, the cipher(s) are assumed to be representing a decimal number.

Commonly, the number of cipher types (and therefore, the place-value multiplier) used in a numeration system is called that system's base. Binary is referred to as "base two" numeration, and decimal as "base ten." Additionally, we refer to each cipher position in binary as a bit rather than the familiar word digit used in the decimal system.

Now, why would anyone use binary numeration? The decimal system, with its ten ciphers, makes a lot of sense, being that we have ten fingers on which to count between our two hands. (It is interesting that some ancient central American cultures used numeration systems with a base of twenty. Presumably, they used both fingers and toes to count!!). But the primary reason that the binary numeration system is used in modern electronic computers is because of the ease of representing two cipher states (0 and 1) electronically.

With relatively simple circuitry, we can perform mathematical operations on binary numbers by representing each bit of the numbers by a circuit which is either on (current) or off (no current). Just like the abacus with each rod representing another decimal digit, we simply add more circuits to give us more bits to symbolize larger numbers. Binary numeration also lends itself well to the storage and retrieval of numerical information: on magnetic tape (spots of iron oxide on the tape either being magnetized for a binary "1" or demagnetized for a binary "0"), optical disks (a laser-burned pit in the aluminum foil representing a binary "1" and an unburned spot representing a binary "0"), or a variety of other media types.

What is Number System?

Number system is a set of numerals for representing numbers. It is the system of naming or representing numbers, as the decimal system or the binary system. It is also called numeral system. A number is a symbol or group of symbols that represents a number. Numerals differ from numbers just as words differ from the things they refer to. The symbols "11", "eleven" and "XI" are different numerals, all representing the same number.

Different Number Systems are:

Number System	Base	Numerals
binary	2	0,1
ternary	3	0,1,2
quaternary	4	0,1,2,3
quinary	5	0,1,2,3,4
senary	6	0,1,2,3,4,5
septenary	7	0,1,2,3,4,5,6
octonary	8	0,1,2,3,4,5,6,7
nonary	9	0,1,2,3,4,5,6,7,8
decimal	10	0,1,2,3,4,5,6,7,8,9
undenary	11	0,1,2,3,4,5,6,7,8,9,A
duodenary	12	0,1,2,3,4,5,6,7,8,9,A,B

Hexadecimal	16	0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

Number and Numeration

It is imperative to understand that the type of numeration system used to represent numbers has no impact upon the outcome of any arithmetical function (addition, subtraction, multiplication, division, roots, powers, or logarithms). A number is a number is a number; one plus one will always equal two (so long as we're dealing with real numbers), no matter how you symbolize one, one, and two. A prime number in decimal form is still prime if its shown in binary form, or octal, or hexadecimal. π is still the ratio between the circumference and diameter of a circle, no matter what symbol(s) you use to denote its value. The essential functions and interrelations of mathematics are unaffected by the particular system of symbols we might choose to represent quantities. This distinction between numbers and systems of numeration is critical to understand. The essential distinction between the two is much like that between an object and the spoken word(s) we associate with it. A house is still a house regardless of whether we call it by its English name house or its Spanish name casa. The first is the actual thing, while the second is merely the symbol for the thing. That being said, performing a simple arithmetic operation such as addition (longhand) in binary form can be confusing to a person accustomed to working with decimal numeration only.

Decimal Number

Decimal, or denary, notation is the most common way of writing the base 10 numeral system, which uses various symbols for ten distinct quantities (0, 1, 2, 3, 4, 5, 6, 7, 8 and 9, called digits) together with the decimal point and the sign symbols + (plus) and − (minus) to represent numbers.
For an n-digit number, the value that each digit represents depends on its weight or position. The weights are based on powers of 10.

8^{th}	7^{th}	6^{th}	5^{th}	4^{th}	3^{rd}	2^{nd}	1^{st}	**POSITION**
10^7=10000000	**10^6=1000000**	**10^5=100000**	**10^4=10000**	**10^3=1000**	**10^2=100**	**10^1=10**	**10^0=1**	**WEIG**

here's the decimal number system as an example:

- digits (or symbols) allowed: 0-9
- base (or radix): 10
- the order of the digits is significant
- 123 is really

$1 \times 100 + 2 \times 10 + 3 \times 1$

$1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$

  - 3 is the most significant digit [MSD] (it carries the most weight)
  - 5 is the least significant digit [LSD] (it carries the least weight)

Binary Number

A binary number is a base two number system, i.e., it has only two values: 0 and 1. The binary system is a base-2 system means there are 2 distinct digits (0 and 1) to represent any quantity. All numbers larger than 1 require more digits. The binary number is represented by subscripted b or B or 2, like, 100010_b or 100010_B or 10010_2.

To express any number in base 2 we use powers much like our own decimal system.

8TH	7TH	6TH	5TH	4TH	3RD	2ND	1ST	POSITION
2^7=128	2^6=64	2^5=32	2^4=16	2^3=8	2^2=4	2^1=2	2^0=1	WEIGHT

When spoken, binary numerals are usually pronounced by pronouncing each individual digit, in order to distinguish them from decimal numbers. For example, the binary numeral "100" is pronounced "one zero zero", rather than "one hundred", to make its binary nature explicit, and for purposes of correctness. Since the binary numeral "100" is equal to the decimal value four, it would be confusing, and numerically incorrect, to refer to the numeral as "one hundred."

here's the characters of binary number system:

- digits (symbols) allowed: 0, 1
- Base (radix): 2

- Each binary digit is called a BIT
- The order of the digits is significant
- Numbering of the digits

Most Significant Bit [MSB] Least Significant Bit [LSB]

n-1 0

where n is the number of digits in the number

- 1001 (base 2) is really

$$1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

= 9 (in base 10)

- 11000 (base 2) is really

$$1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$$

$$= 24_{10} \text{ (base 10)}$$

Binary arithmetic

Arithmetic in binary is much like arithmetic in other numeral systems. Addition, subtraction, multiplication, and division can be performed on binary numerals.

Addition

The simplest arithmetic operation in binary is addition. Adding two single-digit binary numbers is relatively simple:

0 + 0 = 0

0 + 1 = 1

1 + 0 = 1

1 + 1 = 10 (the 1 is carried)

Adding two "1" values produces the value "10", equivalent to the decimal value 2. This is similar to what happens in decimal when certain single-digit numbers are added together; if the result exceeds the value of the radix (10), the digit to the left is incremented:

5 + 5 = 10

7 + 9 = 16

This is known as *carrying* in most numeral systems. When the result of an addition exceeds the value of the radix, the procedure is to "carry the one" to the left, adding it to the next positional value. Carrying works the same way in binary:

```
  1 1 1 1    (carry)

  0 1 1 0 1

+ 1 0 1 1 1

-------------

= 1 0 0 1 0 0
```

In this example, two numerals are being added together: 01101 (13 decimal) and 10111 (23 decimal). The top row shows the carry bits used. Starting in the rightmost column, 1 + 1 = 10. The 1 is carried to the left, and the 0 is written at the bottom of the rightmost column. The second column from the right is added: 1 + 0 + 1 = 10 again; the 1 is carried, and 0 is written at the bottom. The third column: 1 + 1 + 1 = 11. This time, a 1 is carried, and a 1 is written in the bottom row. Proceeding like this gives the final answer 100100.

Subtraction

Subtraction works in much the same way:

0 - 0 = 0

0 - 1 = 1 (with borrow 1)

1 - 0 = 1

1 - 1 = 0

One binary numeral can be subtracted from another as follows:

```
      *  ***  (starred columns are borrowed from)
  1 1 0 1 1 10
-     1 0 1 1
-----------------
= 1 0 1 0 1 1 1
```

Subtracting a positive number is equivalent to *adding* a negative number of equal absolute value; computers typically use the two's complement notation to represent negative values. This notation eliminates the need for a separate "subtract" operation. For further details, see two's complement.

Multiplication

Multiplication in binary is similar to its decimal counterpart. Two numbers *A* and *B* can be multiplied by partial products: for each digit in*B*, the product of that digit in *A* is calculated and written on a new line, shifted leftward so that its rightmost digit lines up with the digit in*B* that was used. The sum of all these partial products gives the final result.

Since there are only two digits in binary, there are only two possible outcomes of each partial multiplication:

- If the digit in *B* is 0, the partial product is also 0
- If the digit in *B* is 1, the partial product is equal to *A*

For example, the binary numbers 1011 and 1010 are multiplied as follows:

```
   1 0 1 1   (A)
 × 1 0 1 0   (B)
 ----------
   0 0 0 0   ← Corresponds to a zero in B
 1 0 1 1     ← Corresponds to a one in B
 0 0 0 0
+1 0 1 1
----------
=1 1 0 1 1 1 0
```

Octal Number

The octal numeral system is the base-8 number system, and uses the digits 0–7, i.e., 0, 1, 2, 3, 4, 5, 6, and 7. It was invented by King Charles XII of Sweden. All numbers larger than 7 require more digits.

Here's some characters of octal number system:

- digits (symbols) allowed: 0-7
- base (radix): 8

- the order of the digits is significant
- An example – $\mathbf{123_8}$ (base 8) is really

$\mathbf{1 \times 8^2 + 2 \times 8^1 + 3 \times 8^0}$

$= \mathbf{64 + 16 + 3}$

$= \mathbf{229_{10}}$ (base 10)

- Another Example: $\mathbf{1161_8}$ (base 8) is really

$\mathbf{1 \times 8^3 + 1 \times 8^2 + 6 \times 8^1 + 1 \times 8^0}$

$\mathbf{= 512 + 64 + 48 + 1}$

$\mathbf{= 625_{10}}$ **(base 10)**

Hexadecimal Number

Hexadecimal or simply hex is a numeral system with a radix or base of 16 usually written using the symbols 0–9 and A–F or a–f, i.e., 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E and F. Here A to F represent 10 to 15. After F, more digits are required.

Here's some characters of hexadecimal number system:

- digits (symbols) allowed: 0-9, A-F
- base (radix): 16
- the order of the digits is significant
- **An Example : $\mathbf{B3_{16}}$** (base 16) is really

$\mathbf{= B \times 16^1 + 3 \times 16^0}$

$\mathbf{= 176 + 3}$

$= 163_{10}$ **(base 10)**

Conversion among Different Number Systems

Converting from other number bases to decimal

Other number systems use different bases. The **binary** number system uses base 2, so the place values of the digits of a binary number correspond to powers of 2. For example, the value of the binary number 11010 is determined by computing the place value of each of the digits of the number:

1	1	0	1	0	The Binary number
2^4	2^3	2^2	2^1	2^0	Place values

So the binary number 11010 represents the value

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

$$= 16+8+0+2+0$$

$$= 26$$

So, $11010_2 = 26_{10}$

Likewise, to convert octal into decimal equivalent, each digit of octal number should be multiplied by the power of 8 ranging from 0 from right to left, and for hex, by 16.

Example:

i. Convert 756_8 into decimal equivalent

Solution: Here the octal number, 756

The equivalent decimal is:

$$7 \times 8^2 + 5 \times 8^1 + 6 \times 8^0$$

$$= 448 + 40 + 6$$

$$= 494_{10}$$

So, $756_8 = 494_{10}$

ii. Convert $2AF_{16}$ into decimal equivalent

Solution: *Here the hexadecimal number, 2AF.*

The equivalent decimal equivalent of A is 10 and F is 15, so, first, we have to convert these digits into decimal.

2	A	F
2	10	15

Now, the decimal equivalent

$$2 \times 16^2 + 10 \times 16^1 + 15 \times 16^0$$

$$= 512 + 160 + 15$$

$$= 687_{10}$$

So, $2AF_{16} = 687_{10}$

Converting from decimal base to other number system

The way to convert decimal into other number system equivalent is to repeatedly divide the decimal number by the base in which it is to be converted, until the quotient becomes zero. As the number is divided, the remainders - in reverse order - form the digits of the number in the other base.

Some Examples:

1. Convert 26 Decimal to Binary Equivalent

Base Value	Number	Remainder
2	26	0
2	13	1
2	6	0
2	3	1
	1	

So, $26_{10} = 11010_2$

2. Convert Decimal 494 into Octal

Base Value	Number	Remainder
8	494	6
8	61	5
	7	

So, $494_{10} = 756_8$

3. Convert Decimal 687 into Hexadecimal

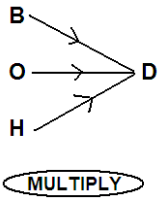
Base Value	Number	Remainder
16	687	15
16	42	10
	2	

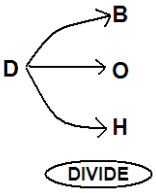
Here from buttom = 2 10 15
Hexadecimal Equivalent = 2 A F

So, $687_{10} = 2AF_{16}$

Things to be remember:

To convert other number System to Decimal, the each digit should be multiplied by the incrementing power from zero from right and added each, and to convert Decimal to other number system, the number should be divided by the base value.

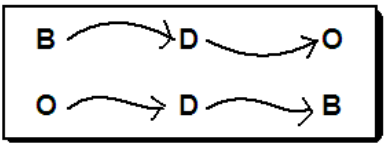




D = DECIMAL
B = BINARY
O = OCTAL
H = HEXADECIMAL (HEX)

Converting from binary to octal and octal to binary using indirect method

To convert binary to octal using Indirect Method, first binary number is converted into decimal and decimal will be into Octal and vice versa for converting Octal to Binary.



Converting from binary to octal and octal to binary using direct method

To convert binary to octal and vice versa using direct method, we need a table:

Octal	0	1	2	3	4	5	6	7
Binary	000	001	010	011	100	101	110	111

Each 3-binary digits are replaced by one octal digit, and vice versa, using the above table.

Example 1: Convert 10110_2 into Octal Number.

Solution: Here the binary number: 10110.

Grouping into three from right, we have: 010 110

From the table, Octal equivalent: 2 6

So, $10110_2 = 26_8$

Example 2: Convert 276_8 into Binary equivalent.

Solution: Here the octal number: 276.

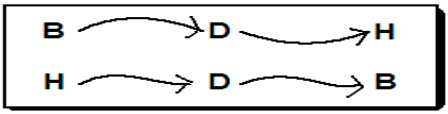
Octal Number: 2 7 6

From the table, Binary Equivalent: 010 111 110

So, $276_8 = 10111110_2$

Converting from binary to hexadecimal and hexadecimal to binary using indirect method

To convert binary to hex using indirect Method, first binary number is converted into decimal and decimal will be into Hexadecimal and vice versa for converting Hexadecimal to Binary.



Converting from binary to hexadecimal and hexadecimal to binary using indirect method

To convert binary to hexadecimal and vice versa using direct method, we need a table:

Hexadecimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Binary	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

Each 4-binary digits are replaced by one octal digit, and vice versa, using the above table.

Example 1: Convert 111010_2 into Hexadecimal Number.

Solution: Here the binary number: 111010.

Grouping into four from right, we have: 0011 1010

From the table, Octal equivalent: 3 A

So, $111010_2 = 3A_{16}$

Example 2: Convert $3BD_{16}$ into Binary equivalent.

Solution: Here the hexadecimal number: 3BD.

Hexadecimal Number: 3 B D

From the table, Binary Equivalent: 0011 1011 1101

So, $3BD_{16} = 1110111101_2$

Things to be remember:

Students must create the Conversion Table performing the conversion.

Conversion of binary fraction to decimal fraction

In a binary fraction, the position of each digit(bit) indicates its relative weight as was the case with the integer part, except the weights to in the reverse direction. Thus after the decimal point, the first digit (bit) has a weight of ½ , the next one has a weight of ¼, followed by 1/8 and so on.

20 . 2-1 2-2 2-3 2-4 … …. …

. 1 0 1 1 0 0 0

The decimal equivalent of this binary number 0.1011 can be worked out by considering

the weight of each bit. Thus in this case it turns out to be (1/2) x 1 + (1/4) x 0 + (1/8) x 1 + (1/16) x 1.

Conversion of decimal fraction to binary fraction

To convert a decimal fraction to its binary fraction, multiplication by 2 is carried out repetitively and the integer part of the result is saved and placed after the decimal point.

The fractional part is taken and multiplied by 2. The process can be stopped any time after the desired accuracy has been achieved.

Example: convert (0.68)10 to binary fraction.

0.68 * 2 = 1.36 integer part is 1

Take the fractional part and continue the process

0.36 * 2 = 0.72 integer part is 0

0.72 * 2 = 1.44 integer part is 1

0.44 * 2 = 0.88 integer part is 0

The digits are placed in the order in which they are generated, and not in the reverse order. Let us say we need the accuracy up to 4 decimal places. Here is the result.

Answer = 0. 1 0 1 0…..

Example: convert (70.68)10 to binary equivalent.

First convert 70 into its binary form which is 1000110. Then convert 0.68 into binary form upto 4 decimal places to get 0.1010. Now put the two parts together.

Answer = 1 0 0 0 1 1 0 . 1 0 1 0....

Complement Method

Complement arithmetic offers a simple and elegant method for performing addition and subtraction on a computer. Complements are used in digital computers for simplifying the subtraction operations and for logical manipulation. There are two types of complements for each base-r system.

(1) the r's complement and

(2) the (r-1)'s complement.

When the value of the base is substituted, the two types receives the names 2's and 1's complement for binary numbers or 10's and 9's complement for decimal numbers.

The r's Complement

- Given a positive number N in base r with an integer part of n digits, the r's complement of N is defined as r^n-N for N ≠ 0 and 0 for N=0. The following numerical example will help clarify the definition.
- The 10's complement of $(52520)_{10}$ is 10^5-52520 =47480
- The number of digits in the number is n=5
- The 2's complement of $(101100)_2$ is $(2^6)_{10}$ –$(101100)_2$=(1000000 - 101100)=010100 (or we can obtain 2's complement by complementing each digit and adding 1).

The (r-1)'s Complement

Given a positive number N in base r with an integer part of n digits, the (r-1)'s complement of N is defined r^n-r^{-m}-N. Some numerical examples follow (n is the number of digit in front of decimal point and m is the number of digit after (behind) decimal point):

- The 9's complement of $(52520)_{10}$ is $(10^5-1-52520)=99999-52520=47479$.
- The 9's complement of $(25.639)_{10}$ is $(10^2-10^{-3}-25.639)=99.999-25.639=74.360$
- The 1's complement of $(101100)_2$ is $(2^6-1) – (101100)=(111111-101100)=010011$.

From the examples, we see that the 9's complement of a decimal number is formed simply by subtracting every digit from 9. The 1's complement of a binary number is even simpler to form. The 1's are changed to 0's and the 0's to 1's. Since the (r-1)'s complement is very easily obtained.

It is sometimes convenient to use it when the r's complement is desired. From the definitions and a comparison of the results obtained in the examples, it follows that the r's complement can be obtained from the (r-1)'s complement after the addition of r^{-m} to the least significant digit. For example, the 2's complement of 10110100 is obtained from the 1's complement 01001011 by adding 1 to give 01001100.

Subtraction with r's complement

The direct method of subtraction taught in elementary schools uses the borrow concept. In this method, we borrow a 1 from a higher significant position when the minuend digit is smaller than the corresponding subtrahend digit.

This seems to be easiest when people perform subtraction with paper and pencil. When subtraction is implemented by means of digital components, this method is found to be less efficient than the method that uses complements and addition as stated below. The subtraction of two positive numbers (M-N), both of base r.

Subtraction with (r-1)'s complement

The procedure for subtraction with the (r-1)'s complement is exactly the same as oe variation, called, "end-around carry" as shown below. The subtraction of M-N, both positive numbers in base r, may be calculated in the following manner.

- Add the minuend M to the (r-1)'s complement of the subtrahend N.
- inspect the result obtained in step 1 for an end carry
  - if an end carry occurs, add 1 to the least significant digit (end-around carry)
  - if an end does not occur, take the (r-1)'s complement of the number obtained in step 1 and place a negative sign infront.

Decimal Subtraction using 9's and 10's Complement Method

Decimal Subtraction by 9's Complement is done by the following steps:

Step I: Making Same No of Digit,

Step II: Making complement (9's complement) of the subtrahend number of step I.

Step III: Adding Minuend No. and the result of step II.

Step IV: Adding 1 to the value of step III

Step V: Removing the overflow digit (if exists), and we get the answer.

Example: Subtract the decimal number using 9's complement: 38765-9899

Solution:

Here, Minuend Number = 38765 and Subtrahend (the number to be subtracted) = 9899

Step I: Making Same no. of digit, we get:

Minuend No. = 38765

Subtrahend = 09899

Step II: Making Complement (9's complement) of subtrahend no (subtracting each digit by 9).,

99999-09899 = 90100

Step III: Adding minuend and complemented subtrahend of step II,

38765 + 90100 = 128865

Step IV: Adding 1 to the final value, i.e., 128865,

128865 + 1 = 128866

Step V: Removing the overflow digit, i.e, Minuend number has 5 digits and step IV result has 6 digits, so leftmost digit is overflow.

Step IV value : 128866

Removing leftmost digit : 28866 *which is the required answer.*

Decimal Subtraction by 10's Complement is done by the following steps:

Step I: Making Same No of Digit,

Step II: Making complement (9's complement) of the subtrahend number of step I.

Step III: Adding 1 to the value of step II.

Step IV: Adding Minuend No. and the result of step III.

Step V: Removing the overflow digit (if exists), and we get the answer.

Example: Subtract the decimal number using 10's complement: 38765-9899

Solution:

Here, Minuend Number = 38765 and Subtrahend (the number to be subtracted) = 9899

Step I: Making Same no. of digit, we get:

Minuend No. = 38765

Subtrahend = 09899

Step II: Making Complement (9's complement) of subtrahend no (subtracting each digit by 9).,

99999-09899 = 90100

Step III: Adding 1 to the last value, i.e., 90100,

90100 + 1 = 90101

Step IV: Adding minuend and value of step III,

38765 + 90101 = 128866

Step V: Removing the overflow digit, i.e, Minuend number has 5 digits and step IV result has 6 digits, so leftmost digit is overflow.

Step IV value : 128866

Removing leftmost digit : 28866 *which is the required answer.*

Binary Subtraction using 1's and 2's Complement Method

Binary Subtraction by 1's Complement is done by the following steps:

Step I: Making Same No of Digit,

Step II: Making complement (1's complement) of the subtrahend number of step I. (Swapping 1 and 0).

Step III: Adding Minuend No. and the result of step II.

Step IV: Adding 1 to the value of step III

Step V: Removing the overflow digit (if exists), and we get the answer.

Example: Subtract the binary number using 1's complement: 10010-1001

Solution:

Here, Minuend Number = 10010 and Subtrahend (the number to be subtracted) = 1001

Step I: Making Same no. of digit, we get:

Minuend No. = 10010

Subtrahend = 01001

Step II: Making Complement (1's complement) of subtrahend no (replacing 1 for 0 and 0 for 1),

10110

Step III: Adding minuend and complemented subtrahend of step II,

10010 + 10110 = 101000

Step IV: Adding 1 to the final value, i.e., 101000,

101000 + 1 = 101001

Step V: Removing the overflow digit, i.e, Minuend number has 5 digits and step IV result has 6 digits, so leftmost digit is overflow.

Step IV value : 101001

Removing leftmost digit : 01001 *which is the required answer.*

Binary Subtraction by 2's Complement is done by the following steps:

Step I: Making Same No of Digit,

Step II: Making complement (1's complement) of the subtrahend number of step I (replacing 1 for 0 and 0 for 1).

Step III: Adding 1 to the value of step II.

Step IV: Adding Minuend No. and the result of step III.

Step V: Removing the overflow digit (if exists), and we get the answer.

Example: Subtract the binary number using 2's complement: 10010-1001

Solution:

Here, Minuend Number = 10010 and Subtrahend (the number to be subtracted) = 1001

Step I: Making Same no. of digit, we get:

Minuend No. = 10010

Subtrahend = 01001

Step II: Making Complement (1's complement) of subtrahend no (replacing 1 for 0 and 0 for 1),

10110

Step III: Adding 1 to the final value, i.e., 10110,

10110 + 1 = 10111

Step IV: Adding minuend and complemented subtrahend of step III,

10110 + 10111 = 101001

Step V: Removing the overflow digit, i.e, Minuend number has 5 digits and step IV result has 6 digits, so leftmost digit is overflow.

Step IV value : 101001

Removing leftmost digit : 01001 *which is the required answer.*

BCD Numbers:

BCD stands for Binary Coded Decimal. BCD represents each of the digits of an unsigned decimal as the 4-bit binary equivalents. **BCD** is also known as **packet decimal** and is numbers 0 through 9 converted to four-digit binary. Below is a list of the decimal numbers 0 through 9 and the binary conversion.

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

Decimal	Binary	Gray
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

It is a method that uses binary digits 0 which represent “off” and 1 which represent “on”. BCD has been in use since the first UNIVAC computer. Each digit is called a bit. Four bits are called a nibble and is used to represent each decimal digit (0 through 9). The first binary number system was documented by Gottfried Leibniz in the 17th century. In 1854 mathematician George Boole came up with a system of logic that is know today as Boolean algebra (based on two elements 0's and 1's)

Gray Code

This is a variable weighted code and is cyclic. This means that it is arranged so that every transition from one value to the next value involves only one bit change. The gray code is sometimes referred to as reflected binary, because the first eight values compare with those of the last 8 values, but in reverse order.

The gray code is often used in mechanical applications such as shaft encoders.

Conversion between Binary and Gray Code:

Modulo 2 Arithmetic

this is binary addition with the carry ignored.

Converting Gray Code to Binary

1. write down the number in gray code
2. the most significant bit of the binary number is the most significant bit of the gray code
3. add (using modulo 2) the next significant bit of the binary number to the next significant bit of the gray coded number to obtain the next binary bit
4. repeat step C till all bits of the gray coded number have been added modulo 2 the resultant number is the binary equivalent of the gray number
   Example, convert 1101101 in gray code to binary

	Gray	Binary	
1.	1101101		
2.	1101101	1	copy down the msb
3.	1101101	10	1 modulo2 1 = 0
4.	1101101	100	0 modulo2 0 = 0
3/4	1101101		1001 0 modulo2 1 = 1
3/4	1101101		10010 1 modulo2 1 = 0
3/4	1101101		100100 0 modulo2 0 = 0
3/4	1101101		1001001 0 modulo2 1 = 1

the answer is 1001001

Converting Binary to Gray

A. write down the number in binary code

B. the most significant bit of the gray number is the most significant bit of the binary code

C. add (using modulo 2) the next significant bit of the binary number to the next significant bit of the binary number to obtain the next gray coded bit

D. repeat step C till all bits of the binary coded number have been added modulo 2 the resultant number is the gray coded equivalent of the binary number

Example, convert 1001001 in binary code to gray code

	Binary	Gray	
1.	1001001		
2.	1001001	1	copy down the msb
3.	1001001	11	1 modulo2 0 = 1
4.	1001001	110	0 modulo2 0 = 0
3/4	1001001		1101 0 modulo2 1 = 1
3/4	1001001		11011 1 modulo2 0 = 1
3/4	1001001		110110 0 modulo2 0 = 0
3/4	1001001		1101101 0 modulo2 1 = 1

The answer is 1101101

ASCII Character Code

Most application of the digital computer require the manipulation of data that consist not only of numbers but also of the letters and some other special characters. This collection is called the *alphanumeric characters*. The *alphanumeric character set* is a set of elements that includes the 10 decimal digits, the 26 letters of the alphabet and a number of special characters, like !@#$%^&*. Such a set contains between 32 and 64 elements if only uppercase letters are included. And the set consists between 64 and 128 if both lowercase and uppercase letters are included. In the first case the binary code will require 6 bits and in the second case the binary code needs 7 bits.

00	NUL	10	DLE	20	SP	30	0	40	@	50	P	60	`	70	p
01	SOH	11	DC1	21	!	31	1	41	A	51	Q	61	a	71	q
02	STX	12	DC2	22	"	32	2	42	B	52	R	62	b	72	r
03	ETX	13	DC3	23	#	33	3	43	C	53	S	63	c	73	s
04	EOT	14	DC4	24	$	34	4	44	D	54	T	64	d	74	t
05	ENQ	15	NAK	25	%	35	5	45	E	55	U	65	e	75	u
06	ACK	16	SYN	26	&	36	6	46	F	56	V	66	f	76	v
07	BEL	17	ETB	27	'	37	7	47	G	57	W	67	g	77	w
08	BS	18	CAN	28	(	38	8	48	H	58	X	68	h	78	x
09	HT	19	EM	29	)	39	9	49	I	59	Y	69	i	79	y
0A	LF	1A	SUB	2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
0B	VT	1B	ESC	2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
0C	FF	1C	FS	2C	´	3C	<	4C	L	5C	\	6C	l	7C	|
0D	CR	1D	GS	2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
0E	SO	1E	RS	2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
0F	SI	1F	US	2F	/	3F	?	4F	O	5F	_	6F	o	7F	DEL

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

The standard alphanumeric binary code is the ASCII (American Standard Code for Information Interchange), which uses 7-bits to code for 128 characters. Now ASCII is extended up to 256 characters by using 8-bits. ASCII is divided into two sets: 128 characters (standard ASCII) and additional 128 characters (extended ASCII)

ASCII was developed in 1968 to standardize data transmission among disparate hardware and software systems and is built into most minicomputers and all PCs. Most PC-based systems use an 8-bit extended ASCII code, with an extra 128 characters used to represent special symbols, foreign language characters, and graphic symbols.

The hexadecimal equivalent for ASCII is listed in the adjacent table. ASCII is a 7-bit code, commonly stored in 8-bit bytes.

- "A" is at 41_{16}. To convert upper case letters to lower case letters, add 20_{16}. Thus "a" is at $41_{16} + 20_{16} = 61_{16}$.
- The character "5" at position 3516 is different than the number 5. To convert character-numbers into number-numbers, subtract 30_{16}: $35_{16} - 30_{16} = 5$.

Why binary numbers are important for computing

The binary numbers play an important role in digital computer operation. The entire code inside the computer must be in binary numbers because the register can only hold the information in binary forms. Binary codes often changed into the symbols. The operation specified for digital computers must take into consideration of meaning of the bits stored in registers so that operations are performed on the operands of the same type. Binary codes can be formulated for any set of discrete elements like musical notes, locations, chess position on board etc. Binary codes are also used to formulate instructions that specify control information for the computer. Computer generally use two discrete variables like ON/OFF, TRUE/FALSE, HIGH/LOW etc. These two just opposite values are represented in binary form. All the data and information are formed by the combination of these two values. Not only computers but all the digital devices are using the binary digits for representing the data.

EBCDIC Character Code

EBCDIC stands for Extended Binary Coded Decimal Interchange Code. It is an IBM code that uses 8-bits to represent 256 possible characters, including text, numbers, punctuation marks, and transmission control characters. This code is primarily in IBM mainframes and minicomputers.

EBCDIC Character Code

- **EBCDIC is an 8-bit code.**

00	NUL	20	DS	40	SP	60	–	80		A0		C0	{	E0	\
01	SOH	21	SOS	41		61	/	81	a	A1	~	C1	A	E1	
02	STX	22	FS	42		62		82	b	A2	s	C2	B	E2	S
03	ETX	23		43		63		83	c	A3	t	C3	C	E3	T
04	PF	24	BYP	44		64		84	d	A4	u	C4	D	E4	U
05	HT	25	LF	45		65		85	e	A5	v	C5	E	E5	V
06	LC	26	ETB	46		66		86	f	A6	w	C6	F	E6	W
07	DEL	27	ESC	47		67		87	g	A7	x	C7	G	E7	X
08		28		48		68		88	h	A8	y	C8	H	E8	Y
09		29		49		69		89	i	A9	z	C9	I	E9	Z
0A	SMM	2A	SM	4A	¢	6A	‘	8A		AA		CA		EA	
0B	VT	2B	CU2	4B		6B	,	8B		AB		CB		EB	
0C	FF	2C		4C	<	6C	%	8C		AC		CC		EC	
0D	CR	2D	ENQ	4D	(	6D	_	8D		AD		CD		ED	
0E	SO	2E	ACK	4E	+	6E	>	8E		AE		CE		EE	
0F	SI	2F	BEL	4F	|	6F	?	8F		AF		CF		EF	
10	DLE	30		50	&	70		90		B0		D0	}	F0	0
11	DC1	31		51		71		91	j	B1		D1	J	F1	1
12	DC2	32	SYN	52		72		92	k	B2		D2	K	F2	2
13	TM	33		53		73		93	l	B3		D3	L	F3	3
14	RES	34	PN	54		74		94	m	B4		D4	M	F4	4
15	NL	35	RS	55		75		95	n	B5		D5	N	F5	5
16	BS	36	UC	56		76		96	o	B6		D6	O	F6	6
17	IL	37	EOT	57		77		97	p	B7		D7	P	F7	7
18	CAN	38		58		78		98	q	B8		D8	Q	F8	8
19	EM	39		59		79		99	r	B9		D9	R	F9	9
1A	CC	3A		5A	!	7A	:	9A		BA		DA		FA	|
1B	CU1	3B	CU3	5B	$	7B	#	9B		BB		DB		FB	
1C	IFS	3C	DC4	5C	·	7C	@	9C		BC		DC		FC	
1D	IGS	3D	NAK	5D	)	7D	'	9D		BD		DD		FD	
1E	IRS	3E		5E	;	7E	=	9E		BE		DE		FE	
1F	IUS	3F	SUB	5F	¬	7F	"	9F		BF		DF		FF	

STX	Start of text	RS	Reader Stop	DC
DLE	Data Link Escape	PF	Punch Off	DC
BS	Backspace	DS	Digit Select	DC
ACK	Acknowledge	PN	Punch On	CU
SOH	Start of Heading	SM	Set Mode	CU
ENQ	Enquiry	LC	Lower Case	CU
ESC	Escape	CC	Cursor Control	SY
BYP	Bypass	CR	Carriage Return	IFS
CAN	Cancel	EM	End of Medium	EC
RES	Restore	FF	Form Feed	ET
SI	Shift In	TM	Tape Mark	NA
SO	Shift Out	UC	Upper Case	SM
DEL	Delete	FS	Field Separator	SC
SUB	Substitute	HT	Horizontal Tab	IG
NL	New Line	VT	Vertical Tab	IR
LF	Line Feed	UC	Upper Case	IU

Unicode

Unicode is a 16-bit character encoding standard developed by the Unicode Consortium between 1988 and 1991. It uses 2 bytes (16 bits) to represent each character. It enables almost all the written language of the world to be represented using a single character set. The 8-bit ASCII is not capable of representing all the combinations o letters and diacritical marks that are used just with the Roman Alphabets. Approximately 39,000 of the 65536 possible Unicode character codes have been assigned to data, 21,000 of them being used for Chinese ideographs. The remaining combinations are open for expansion.

0000	NUL	0020	SP	0040	@	0060	`	0080	Ctrl	00A0	NBS	00C0	À	00E0	à
0001	SOH	0021	!	0041	A	0061	a	0081	Ctrl	00A1	¡	00C1	Á	00E1	á
0002	STX	0022	"	0042	B	0062	b	0082	Ctrl	00A2	¢	00C2	Â	00E2	â
0003	ETX	0023	#	0043	C	0063	c	0083	Ctrl	00A3	£	00C3	Ã	00E3	ã
0004	EOT	0024	$	0044	D	0064	d	0084	Ctrl	00A4	¤	00C4	Ä	00E4	ä
0005	ENQ	0025	%	0045	E	0065	e	0085	Ctrl	00A5	¥	00C5	Å	00E5	å
0006	ACK	0026	&	0046	F	0066	f	0086	Ctrl	00A6	¦	00C6	Æ	00E6	æ
0007	BEL	0027	'	0047	G	0067	g	0087	Ctrl	00A7	§	00C7	Ç	00E7	ç
0008	BS	0028	(	0048	H	0068	h	0088	Ctrl	00A8	¨	00C8	È	00E8	è
0009	HT	0029	)	0049	I	0069	i	0089	Ctrl	00A9	©	00C9	É	00E9	é
000A	LF	002A	*	004A	J	006A	j	008A	Ctrl	00AA	ª	00CA	Ê	00EA	ê
000B	VT	002B	+	004B	K	006B	k	008B	Ctrl	00AB	«	00CB	Ë	00EB	ë
000C	FF	002C	´	004C	L	006C	l	008C	Ctrl	00AC	¬	00CC	Ì	00EC	ì
000D	CR	002D	-	004D	M	006D	m	008D	Ctrl	00AD	–	00CD	Í	00ED	í
000E	SO	002E	.	004E	N	006E	n	008E	Ctrl	00AE	®	00CE	Î	00EE	î
000F	SI	002F	/	004F	O	006F	o	008F	Ctrl	00AF	¯	00CF	Ï	00EF	ï
0010	DLE	0030	0	0050	P	0070	p	0090	Ctrl	00B0	°	00D0	Ð	00F0	¶
0011	DC1	0031	1	0051	Q	0071	q	0091	Ctrl	00B1	±	00D1	Ñ	00F1	ñ
0012	DC2	0032	2	0052	R	0072	r	0092	Ctrl	00B2	²	00D2	Ò	00F2	ò
0013	DC3	0033	3	0053	S	0073	s	0093	Ctrl	00B3	³	00D3	Ó	00F3	ó
0014	DC4	0034	4	0054	T	0074	t	0094	Ctrl	00B4	´	00D4	Ô	00F4	ô
0015	NAK	0035	5	0055	U	0075	u	0095	Ctrl	00B5	µ	00D5	Õ	00F5	õ
0016	SYN	0036	6	0056	V	0076	v	0096	Ctrl	00B6	¶	00D6	Ö	00F6	ö
0017	ETB	0037	7	0057	W	0077	w	0097	Ctrl	00B7	·	00D7	×	00F7	÷
0018	CAN	0038	8	0058	X	0078	x	0098	Ctrl	00B8	¸	00D8	Ø	00F8	ø
0019	EM	0039	9	0059	Y	0079	y	0099	Ctrl	00B9	¹	00D9	Ù	00F9	ù
001A	SUB	003A	:	005A	Z	007A	z	009A	Ctrl	00BA	º	00DA	Ú	00FA	ú
001B	ESC	003B	;	005B	[	007B	{	009B	Ctrl	00BB	»	00DB	Û	00FB	û
001C	FS	003C	<	005C	\	007C	|	009C	Ctrl	00BC	1/4	00DC	Ü	00FC	ü
001D	GS	003D	=	005D	]	007D	}	009D	Ctrl	00BD	1/2	00DD	Ý	00FD	Þ
001E	RS	003E	>	005E	^	007E	~	009E	Ctrl	00BE	3/4	00DE	ý	00FE	þ
001F	US	003F	?	005F	_	007F	DEL	009F	Ctrl	00BF	¿	00DF	ß	00FF	ÿ

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

Boolean Algebra

George Boole: Father of Boolean Algebra and Digital Logic

Boolean algebra is named for its inventor, English mathematician George Boole, born in 1815. His father, a tradesman, began teaching him mathematics at an early age. George Boole was born in Lincoln, England, on November 2, 1815 and invented Boolean algebra. In 1832, binary algebra, now known as Boolean algebra, necessary for creating a binary computer, was developed by seventeen-year-old George Boole. In 1847, George Boole proposed a system of algebra that could be used to manipulate propositions, that is, assertions that could be either true or false. In 1847, the relationships among Boolean algebra, set algebra, logic, and binary arithmetic have given Boolean algebras a central role in the development of electronic digital computers. In 1849, Boolean logic is a symbolic logic system invented by George Boole. George Boole first set up a logic algebra of this type in 1854, is a algebra structure with very good properties. It is used in many areas of information processing and switch theory. George Boole, the father of Boolean algebra, published A Treatise on the Calculus of Finite Differences in 1860. It was an instant classic, and later became the basis for our contemporary digital computer systems. In 1864, George Boole published many other papers regarding other subjects, but he is most known for the boolean algebra. Unfortunately, he died unexpectedly on the 8th of December 1864, at the age of 49. He had walked in the rain for a long time to arrive to his class on time. He then lectured in his wet clothes and he died of a harsh cold. Boolean algebra is still used today in electronic circuits.

In 1903, John V. Atanasoff and Clifford Berry tries make electrical computer that apply algebra boolean at circuit electrical. This approach is relied on George Boole (1815-1864) work result as the shape of binary system algebra, which state that every mathematic equation can be expressed as correctness or wrong. In 1937, Claude Shannon, a graduate student at MIT, discovered the connection between electronic circuits and Boolean algebra. This connection is essential to the operation of computers and modern electronics circuits. Computers and circuits utilizeBoolean algebra for all their decision making calculations, and without it they would be quite useless. George Boole was well ahead of his time with his mathematical theories and the combination of mathematics and logic. In July 1948, Claude Shannon, an American mathematician, wrote an article in the Bell Systems Technical Journal: “A Mathematical theory of communication.” It drew on the 19th century-work of George Boole (after whom Boolean algebra is named) and suggested that all computations can be reduced to ones and zeroes — and if these were achieved by switches in an electrical circuit, a digital computer could be built. Boolean Data Types are the most primitive type that exists on the computer.

The Boolean type consists of 2 states stored in a bit representing true and false. The type was initially used to represent truth values of logic in boolean algebra, which was designed by George Boole in the 19th century.

Introduction

The circuits in computers have inputs (0 or 1) and produce outputs (0 or 1). Circuits can be constructed using any basic unit that has two different states, one for the 0 input/output, and one for the 1 input/output. Typically, such units are in the form of switches that can be either on or off. In 1938, Claude Shannon showed how the rules of propositional logic could be used to design circuits (in his Master's thesis at MIT). These rules form the basis for **Boolean algebra**. A Boolean algebra is just the operations and rules used for working with the set {0,1} where 0 and 1 can take on different meanings. Propositional Logic (which he have been studying) is a Boolean algebra.

Boolean Operators

The main Boolean operators we will use are **complement**, **sum** and **product**. The complement of an element (which always has a value of 0 or 1) is denoted with a single quote: 0' = 1 and 1' = 0. The Boolean sum denoted by + or OR has the following values:

1 + 1 = 1

1 + 0 = 1

0 + 1 = 1

0 + 0 = 0

The Boolean product denoted by • or by AND has the following values:

$1 \bullet 1 = 1$

$1 \bullet 0 = 0$

$0 \bullet 1 = 0$

$0 \bullet 0 = 0$

Typically, the • is dropped so $x \bullet y = xy$. The rules of precedence for these operators are: complement, product, sum.

Therefore, the value of

$$1 \bullet 0 + (0 + 1)' = 1 \bullet 0 + 1'$$

$$= 1 \bullet 0 + 0$$

$$= 0 + 0$$

$$= 0$$

Boolean Expressions and Functions

x is a **Boolean variable** if it can only assume the value of either 0 or 1. A Boolean function is a function whose domain is a set of n-tuples of 0's and 1's, and whose range is an element of the basic Boolean set {0,1}. We always display the values of a Boolean function in a truth table. A Boolean expression on the Boolean variables $\{x_1, x_2, ..., x_n\}$ is an expression using those variables and the operations of a boolean algebra. Every Boolean expression defines a Boolean function. The values of this function are obtained by substituting 0 and 1 for the variables in the expression. For example, we can define a Boolean expression xy + xy' by a Boolean function F(x,y) = xy + xy'. The values of this function are displayed in the table below - all we did was substitute all possible values for the variables.

x	y	xy	xy'	F(x,y)
0	0	0	0	0
0	1	0	0	0
1	0	0	1	1
1	1	1	0	1

The domain in this function is the 2-tuple which represents the values of x and y. The range is {0,1} in the last column. The n-tuple of a Boolean function is just the possible values of the variables. Two Boolean expressions are equivalent if they represent the same function (i.e., have the same truth table).

Boolean Identities

Identity Name

$(x')' = x$	Involution Law
$x + x' = 1$	Complement
$x \bullet x' = 0$	
$x + x = x$	Idempotent Laws
$x \bullet x = x$	
$x + 0 = x$	Identity Laws
$x \bullet 1 = x$	
$x + 1 = 1$	Dominance Laws
$x \bullet 0 = 0$	
$x + y = y + x$	Commutative Laws

$xy = yx$

$x + (y + z) = (x + y) + z$ Associative Laws

$x(yz) = (xy)z$

$x + yz = (x + y)(x + z)$ Distributive Laws

$x(y + z) = xy + xz$

$(xy)' = x' + y'$ DeMorgans Laws

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

$x + (xy) = x$ Absorption Laws

$x(x + y) = x$

$x + x'y = x + y$ Redundancy Laws

$x(x' + y) = xy$

$xy + x'z + yz = xy + x'z$ Consensus Laws

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

Standard Forms of Boolean algebras

All Boolean expressions, regardless of their form can be converted into either of two standard forms:

1) Sum-Of-Product forms (SOP)

2) Product-Of-Sums forms (POS)

Standardization makes the evaluation, simplification and implementation of Boolean expressions much more systematic and easier. When two or more products of variables summed by then it is called sum of product. When two or more sums of variables producted by then it is called product of sum.

Duality Principle

There is a duality principle which applies to all Boolean algebras. In the definition of the identities above, we always include two parts that represents the dual identities. The only different is we interchange • and +, and 0 and 1. Any proven theorem is automatically true for the dual of the theorem.

Proof using truth table:

In Boolean algebra, we can prove the identities using truth tables or using other identities. So, for example the distributive law is proven true by the last two columns of the following table:

x	y	z	y+z	xy	xz	x(y+z)	xy+xz
1	1	1	1	1	1	1	1
1	1	0	1	1	0	1	1

1	0	1	1	0	1	1	1
1	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0
0	1	0	1	0	0	0	0
0	0	1	1	0	0	0	0
0	0	0	0	0	0	0	0

Proof using identity:

A proof of the absorption law $x(x + y) = x$ using identities:

$x(x + y) = (x + 0)(x + y)$ identity law

$= x + 0 \bullet y$ distributive law

$= x + y \bullet 0$ commutative law

$= x + 0$ dominance law

$= x$ identity law

More on Boolean Functions

In preparation for using all the above information to build circuits, we need to be able to solve two basic problems:

1) Given a table of values for a Boolean function, how do we derive the corresponding Boolean expression?

2) Is there a smaller set of operators that can be used to represent a Boolean function?

First, question 1. Given the following table, how do we figure out the corresponding Boolean expression?

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

Notice that the only time that F is 1 is when x = z = 1 and y = 0. This translates to xy'z meaning that the expression xy'z has the value 1 if and only if x = y' = z = 1. G is 1 in two cases: x = y = 1 and z = 0, and x = z = 0 and y = 1. When we have to deal with two or more cases, we represent the sum of the two products: xyz' + x'yz'. This is the Boolean expression for G.

This illustrates a basic method for constructing an expression from the values of a Boolean function. A **minterm** of the Boolean variables $x_1, x_2, ..., x_n$ is a Boolean product $y_1y_2...y_n$ where $y_i = x_i$ or $y_i = x_i'$. A minterm has a value of 1 if and only if all the values of its variables are 1. So, if x = y = 1 and z = 0 then xyz' is the minterm that equals 1. By taking Boolean sums of minterms, we can build up a Boolean expression that represents the values of a function represented by a table. The minterms in this sum correspond to those combinations of the values for which the function has a value of 1. This Boolean sum is sometimes called a **sum of products expansion** or **disjunctive normal form**.

There is a dual entity called a **maxterm** which is a **product of sums expansion** (**conjunctive normal form**). The table below shows the duality:

x	y	z	minterm	maxterm
0	0	0	x'y'z'	x+y+z
0	0	1	x'y'z	x+y+z'
0	1	0	x'yz'	x+y'+z
0	1	1	x'yz	x+y'+z'
1	0	0	xy'z'	x'+y+z
1	0	1	xy'z	x'+y+z'
1	1	0	xyz'	x'+y'+z
1	1	1	xyz	x'+y'+z'

For the following Boolean function F(x,y,z) = (x+z) y

x	y	z	x+z	(x+z) y
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	1	1
1	0	0	1	0
1	0	1	1	0
1	1	0	1	1
1	1	1	1	1

xyz + xyz' + x'yz is the minterm expression;
(x+y+z)(x+y+z')(x+y'+z)(x'+y+z)(x'+y+z') is the maxterm expression. Every Boolean function has both a minterm and maxterm expansion.

Functional Completeness

The other question posed in the previous section was: Do we need all three operators to express Boolean functions? As shown above, it's easy with {' + •} - since every Boolean function can be expressed with these three operators, we say the set is **functionally complete**. Is there a smaller set of functionally complete operators? There is a smaller set if one of the three operators can be expressed in terms of the other two. For example:

x + y = (x'y')'

which means we can get rid of the + operator (this is an application of DeMorgan's Law). So, the set of {' •} is functionally complete. In a similar way we can eliminate Boolean products:

xy = (x' + y')'

Is the set {+ •} functionally complete? There is no way to represent the Boolean function F(x) = x' using these two operators, so this set is not functionally complete.

Finally, is there a set of one operator that is functionally complete? Only if we define a new operator. We could define a NAND operator "|" as follows:

x	y	x| y =[x.y]'
0	0	1
0	1	1
1	0	1
1	1	0

This set is functionally complete. To prove this, we know that the set {• '} is functionally complete so all we have to do is show that these two operators can be expressed using |:

x' = x | x x' = (x.x)'

xy = (x | y) | (x | y) xy=((x.y)'(x.y))'

A similar operator that is functionally complete is the NOR operator: 0

x	y	x0y (x+y)'
0	0	1
0	1	0
1	0	0
1	1	0

Simplification of Boolean Functions

We know that any Boolean function can be built from ANDs, ORs, and NOTs using minterm expansion. However, a practicing computer engineer will very rarely be satisfied with a minterm expansion, because as a rule, it requires more gates than necessary. The laws and identities of Boolean algebra will almost always allow us to simplify a minterm expansion. For example, the minterm expansion for a Boolean function f of three variables might be represented as

follows (taking the values directly from the truth table):

$$f = x'y'z' + x'y'z + x'yz' + x'yz + xyz' + xyz$$

This would require a circuit with maximum gates: 12 ANDs, 5 ORs and 9 NOTs. Using the identities of Boolean algebra, this minterm expansion can be simplified considerably:

$f = x'y'z' + x'y'z + x'yz' + x'yz + xyz' + xyz$

$= x'y'(z' + z) + x'y(z' + z) + xy(z' + z)$ *distributive law*

$= x'y' + x'y + xy$ *complementarity & identity*

$= x'(y' + y) + xy$ *distributive law*

$= x' + xy$ *complementarity & identity*

$= x' + y$ *redundancy*

So, that big long minterm reduces down to x' + y which can be built with 1 OR and 1 NOT. This is an important point, but reducing minterms can require a lot of luck knowing which identities to apply when. Therefore, we will look at a very simple technique that usually leads to a significant simplification of minterms. It won't always produce the simplest form, but it's close enough for most engineers considering the difficulty of the alternative method.

Karnaugh Maps

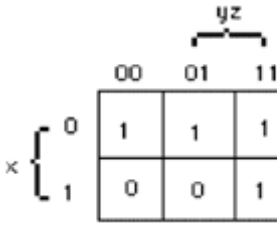
A Karnaugh map is a pictorial representation of a truth table. Karnaugh maps were invented by Maurice Karnaugh, a telecommunications engineer. He developed them at Bell Labs in 1953 while studying the application of digital logic to the design of telephone circuits. This method is typically used on Boolean functions of two, three or four variables - past that, it gets quite cumbersome and other techniques are frequently used. The purposeof Karnaugh Map is to make it easier to transform a logic function to a minimal sum of products. Karnaugh maps are a graphical way of using the relationship AB + AB = A to simplify a Boolean expression and thus simplify the resulting circuit. A Karnaugh map is a completely mechanical method of performing this simplification, and so has an advantage over manipulation of expressions using Boolean algebra. Karnaugh maps are effective for expressions of up to about six variables. The Karnaugh map uses a rectangle divided into rows and columns in such a way that any product term in the expression to be simplified can be represented as the intersection of a row and a column. The

rows and columns are labeled with each term in the expression and its complement. The labels must be arranged so that each horizontal or vertical move changes the state of one and only one variable.

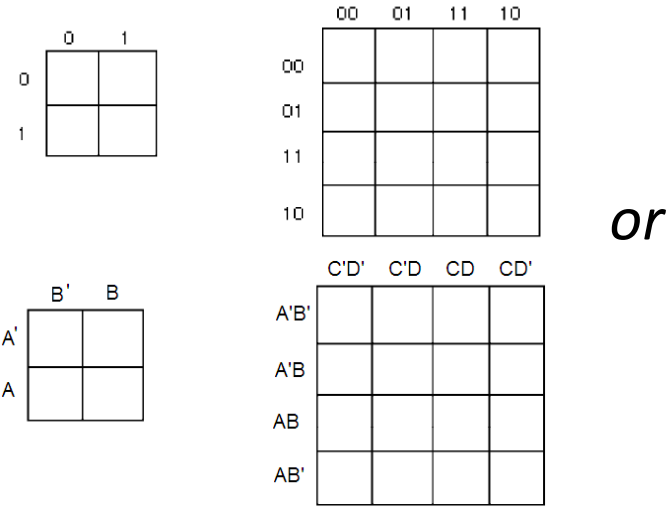
A Karnaugh Map is also called K-map.

A Karnaugh map is a 2-dim representation of the truth table for a Boolean function. For example:

x	y	z	f(x,y,z)
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



Each of the eight cells in the map corresponds to one of the eight possible combinations of x, y, & z. The templates for 2 and 4 variable Karnaugh maps are given below.

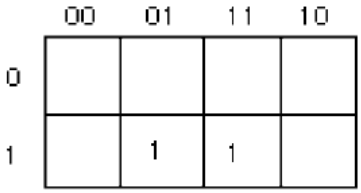


In Karnaugh map, the sequence of variables written in Gray form, not in binary order.

Once we have placed the 1's in the map, there is a simple procedure that we apply. Before doing that though, it is necessary to understand the basis for the procedure. The reason Karnaugh maps are useful is because certain simple Boolean functions have simple Karnaugh maps. These simple functions are called **product functions**; they are products of some or all of the variables *and* their complements. For example, x_1, $x_2'x_4$ and xyz are all product functions but

$x_1 + x_2'$ and xy + zw are not.

Take the product function f(x,y,z) = xz. This function accepts two inputs 101 and 111. Its Karnaugh map:



Notice that the 1's lie in a 1x2 rectangular block. Another example is g(x,y,z) = y'. This function accepts 000,001,100,101 (since for all the other possibilities y = 1 and therefore y' = 0). Its Karnaugh map:

	00	01	11	10
0	1	1		
1	1	1		

The minterms for this one lie in a 2x2 rectangular block. It turns out that every product function has a Karnaugh map whose minterms are confined to a block whose sides are 1, 2 or 4 cells long. Thus, it becomes important to recognize what product function is represented by a particular Karnaugh map.

To do this, first write down the truth set (the set of combinations with a value of 1 from the map). For example:

	00	01	11	10
0			1	
1			1	

The truth set is {011, 111}. Next, we analyze the values of the truth set. If the variables are x, y, and z, we notice that x can be 0 or 1; while y and z can be only 1. We characterize this analysis as follows: {*11} where the "*" is a wildcard. If the unknown product function involved x, it would only accept inputs for which x assumed some particular value. Since x can be either 0 or 1, we conclude that x is not involved. y, on the other hand, must be involved since y = 1 for all inputs. So y must be a term (not y'); the same is true for z. The product function is yz.

Rules for creating Karnaugh Map

To use a Karnaugh map to simplify an expression:

1. Draw a "map" with one box for each possible product term in the expression. The boxes must be arranged so that a one-box movement either horizontal or vertical changes one and only one variable.
2. For each product term in the expression to be simplified, place a checkmark in the box whose labels are the product's variables and their complements.
3. Draw loops around adjacent pairs of checkmarks. Blocks are "adjacent" horizontally and vertically only, not diagonally. A block may be "in" more than

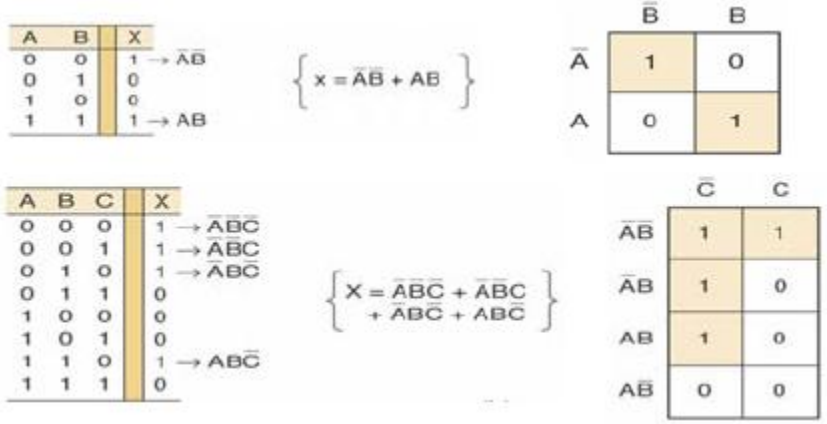
1. one loop, and a single loop may span multiple rows, multiple columns, or both, so long as the number of checkmarks enclosed is a power of two.

4. For each loop, write an unduplicated list of the terms which appear; i.e. no matter how many times A appears, write down only one A.
5. If a term and its complement both appear in the list, e.g. both A and $\bar{A}$, delete both from the list.
6. For each list, write the Boolean product of the remaining terms.
7. Write the Boolean sum of the products from Step 5; this is the simplified expression.

Karnaugh Map Method:

- The truth table values are placed in the K map.
- Adjacent K map square differ in only one variable both horizontally and vertically.
- The pattern from top to bottom and left to right must be in the form
- A SOP (Sum of Product) expression can be obtained by ORing all squares that contain a 1.
- Looping adjacent groups of 2, 4, or 8 1s will result in further simplification. Looping of 2, 4 and 8 is called the Pair, Quad and Octet.
- When the largest possible groups have been looped, only the common terms are placed in the final expression.
- Looping may also be wrapped between top, bottom, and sides.
- For 2 and 3 variables, looping adjacent groups of 2, 4, or 8 1s will result in further simplification.

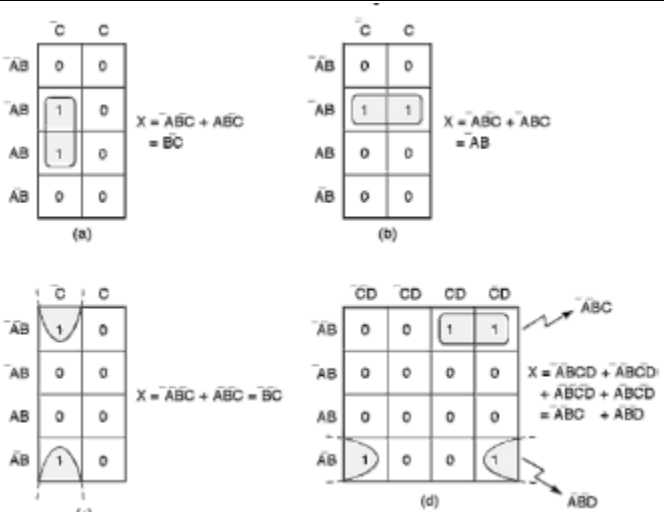
Some Examples:



Looping on Karnaugh Maps

Specific rules apply to looping on a Karnaugh Map, they are summarised here

- Loops must contain 2n cells set to 1.
- A single cell (loop of 20) cannot be simplified.
- A loop of 2 (21) is independant of 1 variable, a loop of 4 (22) is independant of 2 variables. In general a loop of 2n cells is independant of n variables.
- Using the largest loops possible will give the simplest functions.
- All cells in the K-map set to 1 must be included in at least one loop when developing the minterm or maxterm form.
- Loops may overlap if they contain at least one other unlooped cell in the K-map.
- Any loop that has all of its cells included in other loops is redundant.
- Loops must be square or rectangular. Diagonal or L-shaped loops are invalid.
- The edges of a K-map are considered to be adjacent. Therefore a loop can leave at the opt of a K-map and re-enter at the bottom, similarly for the two sides.
- There may be different ways of looping a K-map since for any given circuit there may not be a unique minimal form.



Don't Care Condition

Consider the table for BCD (8421) numbers given in Fig.7-8. Note that the binary numbers 0000 to 1001 on the table are used to specify decimal numbers from 0 to 9. For convenience, the table is completed in the shaded section, which shows other possible combinations of the variables D, C, B, and A. These six combinations (1010, 1011, 1100, 1101, 1110, and 1111) are not used by the BCD code. These combinations are called don't cares when plotted on a Karnaugh map. The don't cares may have some effect on simplifying any logic diagram that might be constructed.

Suppose a problem specifying that a warning light would come ON when the BCD count reached 1001 (decimal 9); see the truth table in Fig.7-9. See 1 is placed in the output column (Y) of the truth table after the input 1001. The Boolean expression for this table (above the shaded section) is D.C'.B'.A = Y. This is shown to the right of

the table. The "not used" combinations in the shaded section of the truth table might have some effect on this problem. A Karnaugh map is drawn in Fig. 7-10b. The 1 for the D.C'.B'.A term is plotted on the map. The six *don't cares* (X's from the truth table) are plotted as X's on the map. An X on the map means that square can be either a 1 or a 0. A loop is drawn around adjacent 1s. The X's on the map can be considered 1s, so the single loop is drawn around the 1 and three X's. Remember that only groups of two, four, or eight adjacent 1s and X's are looped together. The loop contains four squares, which will eliminate two variables. The B and C variables are eliminated, leaving the simplified Boolean expression D.A = Y in Fig. 7-10c. As was said earlier, unused combinations from a truth table are called don't cares. They are shown as X's on a Karnaugh map. Including don't cares (X's) in loops on a map helps to further simplify Boolean expressions.

BCD (8421) number				Decimal equivalent
D	*C*	*B*	*A*	
8s	4s	2s	1s	
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	not used
1	0	1	1	not used
1	1	0	0	not used
1	1	0	1	not used
1	1	1	0	not used
1	1	1	1	not used

Inputs				Output
D	*C*	*B*	*A*	
8s	4s	2s	1s	*Y*
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	X
1	0	1	1	X
1	1	0	0	X
1	1	0	1	X
1	1	1	0	X
1	1	1	1	X

$D \cdot \bar{C} \cdot \bar{B} \cdot A$

Fig. Table of BCD numbers

$D \cdot \bar{C} \cdot \bar{B} \cdot A = Y$

(a) Unsimplified Boolean expression

$D \cdot A = Y$

(c) Simplified Boolean expression

	$\bar{C} \cdot \bar{D}$	$\bar{C} \cdot D$	$C \cdot D$	$C \cdot \bar{D}$
$\bar{A} \cdot \bar{B}$			X	
$\bar{A} \cdot B$		X	X	
$A \cdot B$		X	X	
$A \cdot \bar{B}$		1	X	

Fig. Using K-map in don't care condition.

Number System:

What is Number System?

Number system is a set of numerals for representing numbers. It is the system of naming or representing numbers, as the decimal system or the binary system. It is also called numeral system. A number is a symbol or group of symbols that represents a number. Numerals differ from numbers just as words differ from the things they refer to. The symbols "11", "eleven" and "XI" are different numerals, all representing the same number.
Different Number Systems are:

Number System	Base	Numerals
binary	2	0,1
ternary	3	0,1,2
quaternary	4	0,1,2,3
quinary	5	0,1,2,3,4
senary	6	0,1,2,3,4,5
septenary	7	0,1,2,3,4,5,6
octonary	8	0,1,2,3,4,5,6,7
nonary	9	0,1,2,3,4,5,6,7,8
decimal	10	0,1,2,3,4,5,6,7,8,9
undenary	11	0,1,2,3,4,5,6,7,8,9,A
duodenary	12	0,1,2,3,4,5,6,7,8,9,A,B
Hexadecimal	16	0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

Decimal Number

Decimal, or denary, notation is the most common way of writing the base 10 numeral system, which uses various symbols for ten distinct quantities (0, 1, 2, 3, 4, 5, 6, 7, 8 and 9, called digits) together with the decimal point and the sign symbols + (plus) and − (minus) to represent numbers.
For an n-digit number, the value that each digit represents depends on its weight or position. The weights are based on powers of 10.

8^{th}	7^{th}	6^{th}	5^{th}	4^{th}	3^{rd}	2^{nd}	1^{st}	POSITION
10^7=10000000	10^6=1000000	10^5=100000	10^4=10000	10^3=1000	10^2=100	10^1=10	10^0=1	WEIGHT

here's the decimal number system as an example:

- digits (or symbols) allowed: 0-9
- base (or radix): 10
- the order of the digits is significant
- 123 is really
  1 x 100 + 2 x 10 + 3 x 1
  1 x 10^2 + 2 x 10^1 + 3 x 10^0
  - 3 is the most significant digit [MSD] (it carries the most weight)

- 5 is the least significant digit [LSD] (it carries the least weight)

Binary Number

A binary number is a base two number system, i.e., it has only two values: 0 and 1. The binary system is a base-2 system means there are 2 distinct digits (0 and 1) to represent any quantity. All numbers larger than 1 require more digits. The binary number is represented by subscripted b or B or 2, like, 100010_b or 100010_B or 10010_2.

To express any number in base 2 we use powers much like our own decimal system.

8TH	7TH	6TH	5TH	4TH	3RD	2ND	1ST	POSITION
2^7=128	2^6=64	2^5=32	2^4=16	2^3=8	2^2=4	2^1=2	2^0=1	WEIGHT

When spoken, binary numerals are usually pronounced by pronouncing each individual digit, in order to distinguish them from decimal numbers. For example, the binary numeral "100" is pronounced "one zero zero", rather than "one hundred", to make its binary nature explicit, and for purposes of correctness. Since the binary numeral "100" is equal to the decimal value four, it would be confusing, and numerically incorrect, to refer to the numeral as "one hundred."

here's the characters of binary number system:

- digits (symbols) allowed: 0, 1
- Base (radix): 2
- Each binary digit is called a BIT
- The order of the digits is significant
- Numbering of the digits

Most Significant Bit [MSB] Least Significant Bit [LSB]

n-1 0

where n is the number of digits in the number

- 1001 (base 2) is really
  $1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$
  = 9 (in base 10)
- 11000 (base 2) is really
  $1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
  = 24_{10} (base 10)

Binary arithmetic

Arithmetic in binary is much like arithmetic in other numeral systems. Addition, subtraction,

multiplication, and division can be performed on binary numerals.

Addition

The simplest arithmetic operation in binary is addition. Adding two single-digit binary numbers is relatively simple:

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 10 (the 1 is carried)

Adding two "1" values produces the value "10", equivalent to the decimal value 2. This is similar to what happens in decimal when certain single-digit numbers are added together; if the result exceeds the value of the radix (10), the digit to the left is incremented:

5 + 5 = 10
7 + 9 = 16

This is known as *carrying* in most numeral systems. When the result of an addition exceeds the value of the radix, the procedure is to "carry the one" to the left, adding it to the next positional value. Carrying works the same way in binary:

```
  1 1 1 1   (carry)
  0 1 1 0 1
+ 1 0 1 1 1
-------------
= 1 0 0 1 0 0
```

In this example, two numerals are being added together: 01101 (13 decimal) and 10111 (23 decimal). The top row shows the carry bits used. Starting in the rightmost column, 1 + 1 = 10. The 1 is carried to the left, and the 0 is written at the bottom of the rightmost column. The second column from the right is added: 1 + 0 + 1 = 10 again; the 1 is carried, and 0 is written at the bottom. The third column: 1 + 1 + 1 = 11. This time, a 1 is carried, and a 1 is written in the bottom row. Proceeding like this gives the final answer 100100.

Subtraction

Subtraction works in much the same way:

0 - 0 = 0
0 - 1 = 1 (with borrow 1)
1 - 0 = 1
1 - 1 = 0

One binary numeral can be subtracted from another as follows:

```
    *  ***  (starred columns are borrowed from)
 1 1 0 1 1 1 0
-    1 0 1 1 1
--------------------
= 1 0 1 0 1 1 1
```

Subtracting a positive number is equivalent to *adding* a negative number of equal absolute value; computers typically use the two's complement notation to represent negative values. This notation eliminates the need for a separate "subtract" operation. For further details, see two's complement.

Multiplication

Multiplication in binary is similar to its decimal counterpart. Two numbers *A* and *B* can be multiplied by partial products: for each digit in*B*, the product of that digit in *A* is calculated and written on a new line, shifted leftward so that its rightmost digit lines up with the digit in*B* that was used. The sum of all these partial products gives the final result.

Since there are only two digits in binary, there are only two possible outcomes of each partial multiplication:

- If the digit in *B* is 0, the partial product is also 0
- If the digit in *B* is 1, the partial product is equal to *A*

For example, the binary numbers 1011 and 1010 are multiplied as follows:

```
   1 0 1 1   (A)
× 1 0 1 0   (B)
-----------------
   0 0 0 0  ← Corresponds to a zero in B
 1 0 1 1    ← Corresponds to a one in B
```

```
  0 0 0 0
+ 1 0 1 1
----------------
= 1 1 0 1 1 1 0
```

Octal Number

The octal numeral system is the base-8 number system, and uses the digits 0–7, i.e., 0, 1, 2, 3, 4, 5, 6, and 7. It was invented by King Charles XII of Sweden. All numbers larger than 7 require more digits.

Here's some characters of octal number system:

- digits (symbols) allowed: 0-7
- base (radix): 8
- the order of the digits is significant
- An example – $\mathbf{123_8}$ (base 8) is really
  $\mathbf{1 \times 8^2 + 2 \times 8^1 + 3 \times 8^0}$
  $\mathbf{= 64 + 16 + 3}$
  $= \mathbf{229_{10}}$ (base 10)

- Another Example: $\mathbf{1161_8}$ (base 8) is really
  $\mathbf{1 \times 8^3 + 1 \times 8^2 + 6 \times 8^1 + 1 \times 8^0}$
  $\mathbf{= 512 + 64 + 48 + 1}$
  $\mathbf{= 625_{10}}$ **(base 10)**

Hexadecimal Number

Hexadecimal or simply hex is a numeral system with a radix or base of 16 usually written using the symbols 0–9 and A–F or a–f, i.e., 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E and F. Here A to F represent 10 to 15. After F, more digits are required.

Here's some characters of hexadecimal number system:

- digits (symbols) allowed: 0-9, A-F
- base (radix): 16
- the order of the digits is significant
- **An Example : $\mathbf{B3_{16}}$** (base 16) is really
  $\mathbf{= B \times 16^1 + 3 \times 16^0}$
  $\mathbf{= 176 + 3}$
  $\mathbf{= 163_{10}}$ **(base 10)**

Conversion among Different Number Systems

Converting from other number bases to decimal

Other number systems use different bases. The **binary** number system uses base 2, so the place values of the digits of a binary number correspond to powers of 2. For example, the value of the binary number 11010 is determined by computing the place value of each of the digits of the number:

1	1	0	1	0	The Binary number
2^4	2^3	2^2	2^1	2^0	Place values

So the binary number 11010 represents the value

$$= 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$
$$= 16+8+0+2+0$$
$$= 26$$

So, $11010_2 = 26_{10}$

Likewise, to convert octal into decimal equivalent, each digit of octal number should be multiplied by the power of 8 ranging from 0 from right to left, and for hex, by 16.
Example:

i. Convert 756_8 into decimal equivalent

Solution: Here the octal number, 756

The equivalent decimal is:

$$7 \times 8^2 + 5 \times 8^1 + 6 \times 8^0$$
$$= 448 + 40 + 6$$
$$= 494_{10}$$

So, $756_8 = 494_{10}$

ii. Convert $2AF_{16}$ into decimal equivalent

Solution: *Here the hexadecimal number, 2AF.*

The equivalent decimal equivalent of A is 10 and F is 15, so, first, we have to convert these digits into decimal.

2	A	F
2	10	15

Now, the decimal equivalent

$$2 \times 16^2 + 10 \times 16^1 + 15 \times 16^0$$
$$= 512 + 160 + 15$$
$$= 687_{10}$$

So, $2AF_{16} = 687_{10}$

Converting from decimal base to other number system

The way to convert decimal into other number system equivalent is to repeatedly divide the decimal number by the base in which it is to be converted, until the quotient becomes zero. As the number is divided, the remainders - in reverse order - form the digits of the number in the other base.

Some Examples:

1. Convert 26 Decimal to Binary Equivalent

Base Value	Number	Remainder
2	26	0
2	13	1
2	6	0
2	3	1
	1	

So, $26_{10} = 11010_2$

2. Convert Decimal 494 into Octal

Base Value	Number	Remainder
8	494	6
8	61	5
	7	

So, $494_{10} = 756_8$

3. Convert Decimal 687 into Hexadecimal

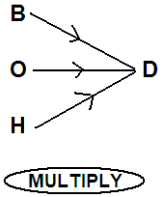
Base Value	Number	Remainder
16	687	15
16	42	10
	2	

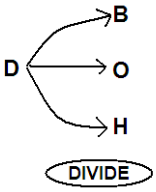
Here from buttom = 2 10 15
Hexadecimal Equivalent = 2 A F

So, $687_{10} = 2AF_{16}$

Things to be remember:

To convert other number System to Decimal, the each digit should be multiplied by the incrementing power from zero from right and added each, and to convert Decimal to other number system, the number should be divided by the base value.

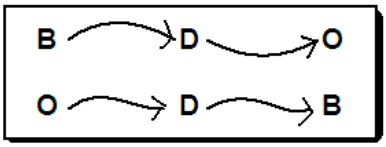




D = DECIMAL
B = BINARY
O = OCTAL
H = HEXADECIMAL (HEX)

Converting from binary to octal and octal to binary using indirect method

To convert binary to octal using Indirect Method, first binary number is converted into decimal and decimal will be into Octal and vice versa for converting Octal to Binary.



Converting from binary to octal and octal to binary using direct method

To convert binary to octal and vice versa using direct method, we need a table:

Octal	0	1	2	3	4	5	6	7
Binary	000	001	010	011	100	101	110	111

Each 3-binary digits are replaced by one octal digit, and vice versa, using the above table.

Example 1: Convert 10110_2 into Octal Number.

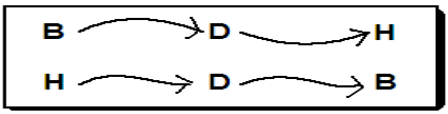
Solution: Here the binary number: 10110.
Grouping into three from right, we have: 010 110
From the table, Octal equivalent: 2 6
So, $10110_2 = 26_8$

Example 2: Convert 276_8 into Binary equivalent.

Solution: Here the octal number: 276.
Octal Number: 2 7 6
From the table, Binary Equivalent: 010 111 110
So, $276_8 = 10111110_2$

Converting from binary to hexadecimal and hexadecimal to binary using indirect method

To convert binary to hex using indirect Method, first binary number is converted into decimal and decimal will be into Hexadecimal and vice versa for converting Hexadecimal to Binary.



Converting from binary to hexadecimal and hexadecimal to binary using indirect method

To convert binary to hexadecimal and vice versa using direct method, we need a table:

Hexadecimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Binary	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Each 4-binary digits are replaced by one octal digit, and vice versa, using the above table.

Example 1: Convert 111010_2 into Hexadecimal Number.

Solution: Here the binary number: 111010.
Grouping into four from right, we have: 0011 1010
From the table, Octal equivalent: 3 A
So, $111010_2 = 3A_{16}$

Example 2: Convert $3BD_{16}$ into Binary equivalent.

Solution: Here the hexadecimal number: 3BD.
Hexadecimal Number: 3 B D
From the table, Binary Equivalent: 0011 1011 1101
So, $3BD_{16} = 1110111101_2$

Things to be remember:

Students must create the Conversion Table performing the conversion.

Complement Method

Complement arithmetic offers a simple and elegant method for performing addition and subtraction on a computer. Complements are used in digital computers for simplifying the subtraction operations and for logical manipulation. There are two types of complements for each base-r system.

(1) the r's complement and
(2) the (r-1)'s complement.

When the value of the base is substituted, the two types receives the names 2's and 1's complement for binary numbers or 10's and 9's complement for decimal numbers.

The r's Complement

- Given a positive number N in base r with an integer part of n digits, the r's complement of N is defined as r^n-N for N ≠ 0 and 0 for N=0. The following numerical example will help clarify the definition.
- The 10's complement of $(52520)_{10}$ is 10^5-52520 =47480
- The number of digits in the number is n=5
- The 2's complement of $(101100)_2$ is $(2^6)_{10}$ –$(101100)_2$=(1000000 - 101100)=010100 (or we can obtain 2's complement by complementing each digit and adding 1).

The (r-1)'s Complement

Given a positive number N in base r with an integer part of n digits, the (r-1)'s complement of N is defined r^n-r^{-m}-N. Some numerical examples follow (n is the number of digit in front of decimal point and m is the number of digit after (behind) decimal point):

- The 9's complement of $(52520)_{10}$ is (10^5-1-52520)=99999-52520=47479.
- The 9's complement of $(25.639)_{10}$ is (10^2-10^{-3}-25.639)=99.999-25.639=74.360
- The 1's complement of $(101100)_2$ is (2^6-1) – (101100)=(111111-101100)=010011.

From the examples, we see that the 9's complement of a decimal number is formed simply by subtracting every digit from 9. The 1's complement of a binary number is even simpler to form. The 1's are changed to 0's and the 0's to 1's. Since the (r-1)'s complement is very easily obtained.

It is sometimes convenient to use it when the r's complement is desired. From the definitions and a comparison of the results obtained in the examples, it follows that the r's complement can be obtained from the (r-1)'s complement after the addition of r^{-m} to the least significant digit. For example, the 2's complement of 10110100 is obtained from the 1's complement 01001011 by adding 1 to give 01001100.

Subtraction with r's complement

The direct method of subtraction taught in elementary schools uses the borrow concept. In this method, we borrow a 1 from a higher significant position when the minuend digit is smaller than the corresponding subtrahend digit.

This seems to be easiest when people perform subtraction with paper and pencil. When subtraction is implemented by means of digital components, this method is found to be less efficient than the method that uses complements and addition as stated below. The subtraction of two positive numbers (M-N), both of base r.

Subtraction with (r-1)'s complement

The procedure for subtraction with the (r-1)'s complement is exactly the same as oe variation, called, "end-around carry" as shown below. The subtraction of M-N, both positive numbers in base r, may be calculated in the following manner.

- Add the minuend M to the (r-1)'s complement of the subtrahend N.
- inspect the result obtained in step 1 for an end carry
  - if an end carry occurs, add 1 to the least significant digit (end-around carry)
  - if an end does not occur, take the (r-1)'s complement of the number obtained in step 1 and place a negative sign infront.

Decimal Subtraction using 9's and 10's Complement Method

Decimal Subtraction by 9's Complement is done by the following steps:

Step I: Making Same No of Digit,
Step II: Making complement (9's complement) of the subtrahend number of step I.
Step III: Adding Minuend No. and the result of step II.
Step IV: Adding 1 to the value of step III
Step V: Removing the overflow digit (if exists), and we get the answer.

Example: Subtract the decimal number using 9's complement: 38765-9899
Solution:

Here, Minuend Number = 38765 and Subtrahend (the number to be subtracted) = 9899
Step I: Making Same no. of digit, we get:
Minuend No. = 38765
Subtrahend = 09899
Step II: Making Complement (9's complement) of subtrahend no (subtracting each digit by 9).,
99999-09899 = 90100
Step III: Adding minuend and complemented subtrahend of step II,
38765 + 90100 = 128865
Step IV: Adding 1 to the final value, i.e., 128865,
128865 + 1 = 128866
Step V: Removing the overflow digit, i.e, Minuend number has 5 digits and step IV result has 6 digits, so leftmost digit is overflow.
Step IV value : 128866
Removing leftmost digit : 28866 *which is the required answer.*

Decimal Subtraction by 10's Complement is done by the following steps:

Step I: Making Same No of Digit,
Step II: Making complement (9's complement) of the subtrahend number of step I.
Step III: Adding 1 to the value of step II.
Step IV: Adding Minuend No. and the result of step III.
Step V: Removing the overflow digit (if exists), and we get the answer.

Example: Subtract the decimal number using 10's complement: 38765-9899
Solution:

Here, Minuend Number = 38765 and Subtrahend (the number to be subtracted) = 9899
Step I: Making Same no. of digit, we get:

Minuend No. = 38765
Subtrahend = 09899

Step II: Making Complement (9's complement) of subtrahend no (subtracting each digit by 9).,
99999-09899 = 90100

Step III: Adding 1 to the last value, i.e., 90100,
90100 + 1 = 90101

Step IV: Adding minuend and value of step III,
38765 + 90101 = 128866

Step V: Removing the overflow digit, i.e, Minuend number has 5 digits and step IV result has 6 digits, so leftmost digit is overflow.
Step IV value : 128866
Removing leftmost digit : 28866 *which is the required answer.*

Binary Subtraction using 1's and 2's Complement Method

Binary Subtraction by 1's Complement is done by the following steps:

Step I: Making Same No of Digit,
Step II: Making complement (1's complement) of the subtrahend number of step I. (Swapping 1 and 0).
Step III: Adding Minuend No. and the result of step II.
Step IV: Adding 1 to the value of step III
Step V: Removing the overflow digit (if exists), and we get the answer.

Example: Subtract the binary number using 1's complement: 10010-1001
Solution:

Here, Minuend Number = 10010 and Subtrahend (the number to be subtracted) = 1001

Step I: Making Same no. of digit, we get:
Minuend No. = 10010
Subtrahend = 01001

Step II: Making Complement (1's complement) of subtrahend no (replacing 1 for 0 and 0 for 1),
10110

Step III: Adding minuend and complemented subtrahend of step II,
10010 + 10110 = 101000

Step IV: Adding 1 to the final value, i.e., 101000,
101000 + 1 = 101001

Step V: Removing the overflow digit, i.e, Minuend number has 5 digits and step IV result has 6 digits, so leftmost digit is overflow.
Step IV value : 101001
Removing leftmost digit : 01001 *which is the required answer.*

Binary Subtraction by 2's Complement is done by the following steps:

Step I: Making Same No of Digit,
Step II: Making complement (1's complement) of the subtrahend number of step I (replacing 1 for 0 and 0 for 1).

Step III: Adding 1 to the value of step II.
Step IV: Adding Minuend No. and the result of step III.
Step V: Removing the overflow digit (if exists), and we get the answer.

Example: Subtract the binary number using 2's complement: 10010-1001
Solution:

Here, Minuend Number = 10010 and Subtrahend (the number to be subtracted) = 1001
Step I: Making Same no. of digit, we get:
Minuend No. = 10010
Subtrahend = 01001
Step II: Making Complement (1's complement) of subtrahend no (replacing 1 for 0 and 0 for 1),
10110
Step III: Adding 1 to the final value, i.e., 10110,
10110 + 1 = 10111
Step IV: Adding minuend and complemented subtrahend of step III,
10110 + 10111 = 101001
Step V: Removing the overflow digit, i.e, Minuend number has 5 digits and step IV result has 6 digits, so leftmost digit is overflow.
Step IV value : 101001
Removing leftmost digit : 01001 *which is the required answer.*

BCD Numbers:

BCD stands for Binary Coded Decimal. BCD represents each of the digits of an unsigned decimal as the 4-bit binary equivalents. **BCD** is also known as **packet decimal** and is numbers 0 through 9 converted to four-digit binary. Below is a list of the decimal numbers 0 through 9 and the binary conversion.

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

It is a method that uses binary digits 0 which represent “off” and 1 which represent “on”. BCD has been in use since the first UNIVAC computer. Each digit is called a bit. Four bits are called a nibble and is used to represent each decimal digit (0 through 9). The first binary number system was documented by Gottfried Leibniz in the 17th century. In 1854 mathematician George Boole came up with a system of logic that is know today as Boolean algebra (based on two elements 0's and 1's)

Logic Function and Boolean Algebra

Boolean algebra is a set of rules, laws and theorems by which logical expressions can be expressed symbolically in equation form and manipulated mathematically. There are only to constants in the Boolean algebra – 0 and 1. No negative (- ve) or fractional numbers are allowed in Boolean algebra. The architecture and operational method of modern digital computer is completely based on Boolean algebra. So, it is one of the most important study of the computer science.

Logic Function (Boolean Function)

Different functions performed in Boolean algebra are called logic functions. Logical facts such as AND (product), OR (sum) or NOT (negation) and their combinations can be performed within Boolean algebra. Basic logic functions are AND, OR and NOT.

AND Function

The result of AND function is only TRUE if all the elements are TRUE otherwise if any element is FALSE, the result will also be FALSE. For example:

f(x)= a AND b AND c.

This function *f(x)* will only be TRUE if *a, b* and *c* all are TRUE otherwise *f(x)* will be false. The AND function just acts like multiplication, so, it is performed using the product algebraic rules and denoted by dot (.), like:

f(x) = a.b.c

OR Function

The result of OR function is only FALSE where all the elements are FALSE, otherwise if any one element is TRUE then the result will also be TRUE. For example:

f(x) = a OR b OR c.

In this example, the function *f(x) will only be* FALSE *if a, b and c* all are FALSE, otherwise *f(x)* will be TRUE.

The OR function just acts like addition, so it is performed using the sum algebraic rules and denoted by the *plus* sign (+), like:

f(x) = a + b + c

NOT Function

The NOT function always negates the element. If the element is TRUE, the result of NOT is always FALSE and vice versa. For example:

f(x) = NOT a

The NOT function just acts like negation, so it is performed using the negative algebraic rules and denoted by a overhead bar ($\overline{a}$) or a single quote (a').

f(x) = $\overline{a}$ or *f(x) = a'*

What is truth table?

Truth table is a table which shows the truth and falsity of output of any given combination of the inputs. It contains all possible combinations of inputs, their manipulation and output. Truth table is an essential for Boolean algebra. Boolean algebra only contains two values – o and 1. So, the truth table contains only the Os and 1s. O is the representation of FALSE or OFF or LOW or NO and the 1 is the representation for TRUE or ON or HIGH or YES. A truth table specifies the value of Boolean expression for every possible combination of values of the variable in the expression.

Boolean Expression

A Boolean expression is the operation among different Boolean variables and the Boolean constants (either 0 or 1). Boolean variable is a variable if it takes values either 0 or 1. The example of Boolean expression is given below:

$AC + (A + B) + \overline{C}\ \overline{A}$

Following rules are to be followed when evaluating a Boolean expression:

1. Perform all individual complements, like $\overline{AB}$.
2. Perform all operations within parenthesis, like (A+B).

3. Perform an AND operation before OR operation.
4. If the whole expression has complemented then perform the expressions and then complement the result.

Duality Principle

The dual in Boolean expression is formed by replacing AND with OR, OR with AND, 0 with 1, 1 with 0. The complement remain unchanged.

Example, if $f = (x+1).(x+ \overline{y})$ then the dual expression,

$f^D = (x .0)+(x. \overline{y})$

Logic Gate

A logic gate is an electronic device that is operated on one or more input signal to produce an output signal. Gates are digital devices and they are also called logic circuit.

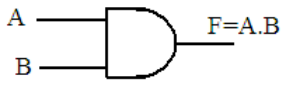
Operators

Operators are signs or symbols that are used to manipulate values or variables. Any operators that are used for manipulating Boolean

values are called Boolean operators. In Boolean algebra, AND, OR, NOT, XOR Gates are represented by **.**, **+**, $\bar{\ }$, and $\oplus$ operators, respectively.

AND Gate

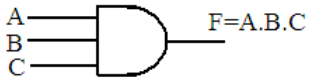
An AND gate is an electronic element that has two or more input signals and produces only a single output signal. The output of AND gate is TRUE only when all inputs are TRUE, otherwise the output is FALSE. The Logic Diagrams and the Truth Tables for two-input and three-input AND gates are shown below:



Fig(1): Symbol for Two-input AND Gate

Inputs		Output
A	B	F=A.B
0	0	0
0	1	0
1	0	0
1	1	1

Table (1) : Truth Table for 2-input AND Gate



Fig(2): Symbol for Three-input AND Gate

Inputs			Output
A	B	C	F=A.B.C
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

(2) : Truth Table for 3-input AND Gate

OR Gate

The OR Gate is a digital electronic element that accepts two or more input signals and produces a single output equivalent to the sum of the inputs. The output of OR gate is FALSE only when all input signals are FALSE, otherwise the output will be TRUE. The Logic Diagrams

and the Truth Tables for two-input and three-input OR gates are shown below:

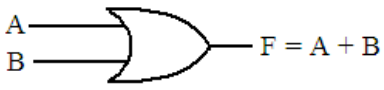


Fig. Symbol for 2-input OR Gate

Input A	Input B	Output F=A+B
0	0	0
0	1	1
1	0	1
1	1	1

Table: Truth Table for 2- input OR Gate

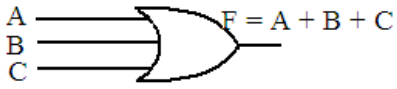


Fig. Symbol for 3-input OR Gate

Inputs A	Inputs B	Inputs C	Output F=A+B+C
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Table: Truth Table for 3- input OR Gate

NOT Gate

A NOT gate is a digital electronic element that accepts only a single input and produces the output equivalent to the complement of the input signal. The output of the NOT gate is FALSE if the input is TRUE and vice versa. NOT gate is a unary form of the logic gate that produce the just opposite value of the input value, i.e., if the input signal carries the value 1, the output of the NOT gate is 0 and if the input is 0 then the output be 1. It always produces the complement (reverse) form of the input signal. So, it is also called an inverter. The

Boolean expression for the NOT gate is Y = $\overline{A}$ or A'.

The Symbol and the Truth Table for a NOT gate is given below:

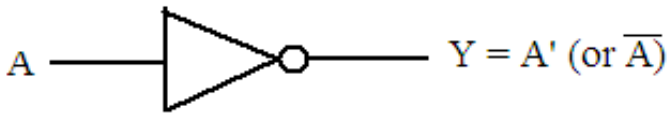


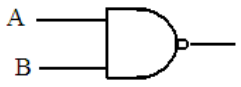
Fig. Logic Symbol for a NOT Gate

Input A	Output Y=A' [or $\overline{A}$]
0	0
1	1

Table: Truth Table for a NOT Gate

NAND Gate

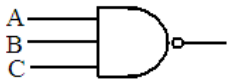
NAND gate is an electronic digital element that combines both the features of NOT and AND gate. It accepts two or more input signals and produces a output equivalent to the complement of the product of the input signals. The output of the NAND gate is FALSE only is the all input signals have the value TRUE, otherwise it gives always the TRUE value. The logic expression for a NAND gate is F= (A.B)'. The Symbols and Truth Tables for 2-input and three-input NOR gate is given below:



Fig(1): Symbol for Two-input NAND Gate

Input A	Input B	Output F=(A.B)
0	0	1
0	1	1
1	0	1
1	1	0

Table (1) : Truth Table for 2-input NAND Gate



Fig(2): Symbol for Three-input NAND Gate

Inputs A	Inputs B	Inputs C	Output F=(A.B.C)'
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

(2) : Truth Table for 3-input NAND Gate

NOR Gate

NOR gate is an electronic digital element that combines both the features of NOT and OR gate. It accepts two or more input signals and produces a output equivalent to the complement of the sum of the input signals. The output of the NOR gate is TRUE only is the all input signals have the value FALSE, otherwise it gives always the FALSE value. The logic expression for a NOR gate is F= (A+B)'. The

Symbols and Truth Tables for 2-input and three-input NOR gate is given below:

Fig. Symbol for 2-input NORGate

Input		Output
A	B	F=(A+B)'
0	0	1
0	1	0
1	0	0
1	1	0

Table: Truth Table for 2- input NORGate

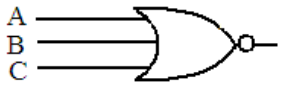


Fig. Symbol for 3-input NOR Gate

Inputs			Output
A	B	C	F=(A+B+C)'
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Table: Truth Table for 3- input NOR Gate

XOR Gate

XOR (also referred to as Ex-OR) stands for Exclusive OR. The XOR gate has two or more input signals and only a single output. It is a digital logic element that accepts two or more values and produces the output in exclusive or forms, i.e., either but not both. The output of an XOR gate is TRUE only when both inputs are different, otherwise if both inputs are same (1s or 0s), the output will be FALSE. It is a combination of AND, OR and NOT gates. It is represented by a XOR (⊕) operator. The Boolean expression for an XOR Gate is, Y= A⊕B, in fact it has the expression: Y = A'.B+A.B'

The Logic Symbol and the Truth Table for an XOR Gate is given below:

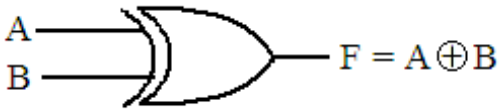


Fig: Logic Symbol for an XOR Gate

Inputs		Output F=A⊕B (F = A'.B+AB')
A	B	
0	0	0
0	1	1
1	0	1
1	1	0

Table: Truth Table for an XOR Gate

XNOR Gate

XOR (also referred to as Ex-NOR) stands for Exclusive NOR. The XNOR gate has two or more input signals and only a single output. It is a digital logic element that accepts two or more values and produces the output in complement of exclusive or forms, i.e., both but not either. The output of an XNOR gate is TRUE only when both inputs are same, otherwise if both inputs are different, the output will be FALSE. It is a combination of AND, OR and NOT gates. It is represented by a whole complement and XOR (⊕) operators. The Boolean expression for an XNOR Gate is, Y= (A⊕B)', actually, it has the expression: Y = A'.B'+A.B

The Logic Symbol and the Truth Table for an XNOR Gate is given below:

Fig: Logic Symbol for an XNOR Gate

Inputs		Output F=(A⊕B)' (F = A'.B'+A.B)
A	B	
0	0	1
0	1	0
1	0	0
1	1	1

Table: Truth Table for an XNOR Gate

Boolean Functions

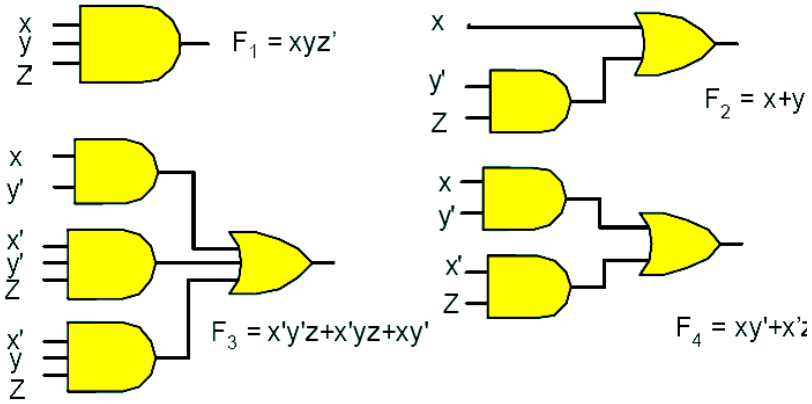
The following Boolean functions are represented in the truth table.

$F_1 = xyz'$ $F_2 = x+y'z$

$F_3 = x'y'z+x'yz+xy'$ $F_4 = xy' + x'z$

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

The functions can be implemented, using the basic logic gates, as shown in the following logic diagrams.



Algebraic Manipulation

Boolean functions are made up of terms. Each term consists of a number of literals. A literal is a variable or the complement of a variable. Each term is represented by a logic gate and each literal represents an input to a logic gate. By reducing the number of terms, the number of literals, or both, a simpler logic

circuit can be used to implement the Boolean function. Reduction of the number of terms and/or number of literals is done by algebraic manipulation.

Examples

1. $x(x'+y)=xx'+xy=0+xy=xy$
2. The dual of (1) is □

$x+x'y=(x+x')(x+y)=1.(x+y)=x+y$

Complement of a Function

The complement of a Boolean function may be obtained by either one of two methods:

1. Repetitive application of DeMorgan's theorem.
2. Taking the dual of the function and complementing each literal.

Example:

Find the complement of $F = x'yz' + x'y'z$

First method

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

Second method

dual of F → $F^{dual} = (x' + y + z')(x' + y' + z)$

then complement each literal

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

Canonical and Standard Forms

A function is said to be in Canonical Form if it is expressed either in Sum of Minterms form or Product of Maxterms form.

Minterms and Maxterms

			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

A term that is a Minterm or a Maxterm must contain all the input variables.

Minterms and Maxterms for three binary values

			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

Functions of Three Variables:

x	y	z	F_1	F_2	F_1'
0	0	0	0	0	1
0	0	1	1	0	0
0	1	0	0	0	1
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	0	1	1
1	1	0	0	1	1
1	1	1	1	1	0

$$F_1 = x'y'z + xy'z' + xyz$$

$$F_2 = x'yz + xy'z + xyz' + xyz$$

$$F_1 = m_1 + m_4 + m_7$$

$$= \sum m(1,4,7) = \sum(1,4,7)$$

$$F_2 = m_3 + m_5 + m_6 + m_7$$

$$= \sum(3,5,6,7)$$

$$F_1' = m_0 + m_2 + m_3 + m_5 + m_6$$

$$= \sum(0,2,3,5,6)$$

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

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

$$= M_0.M_2.M_3.M_5.M_6$$

$$= \prod(0,2,3,5,6)$$

$$Similarly, F_2 = \prod(0,1,2,4)$$

Example 1: Express F = A + B'C in Sum Of Minterms or SOP form.

A	B	C	A+B'C
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

$$F = A(B + B') + B'C = AB + AB'+B'C$$
$$= AB(C + C') + AB'(C + C') + (A + A')B'C$$
$$= ABC + ABC'+AB'C + AB'C'+AB'C + A'B'C$$
$$= ABC + ABC'+AB'C + AB'C'+A'B'C$$
$$= m7 + m6 + m5 + m4 + m1$$
$$= \Sigma(1,4,5,6,7)$$

Example 2: Express F = $xy + x'z$ in Product of Maxterms or POS form.

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

$$F = (xy + x')(xy + z)$$
$$= (x + x')(y + x')(x + z)(y + z)$$
$$= (x'+y)(x + z)(y + z)$$
$$= (x'+y + zz')(x + yy'+z)(xx'+y + z)$$
$$= (x'+y + z)(x'+y + z')(x + y + z)(x + y'+z)(x + y + z)(x'+y + z)$$
$$= (x + y + z)(x + y'+z)(x'+y + z)(x'+y + z')$$
$$= M_0.M_2.M_4.M_5$$
$$= \Pi(0,2,4,5)$$

Conversion between Canonical Forms

Example:

$$F(A,B,C) = \Sigma(1,4,5,6,7)$$
$$F'(A,B,C) = \Sigma(0,2,3)$$
$$\therefore F(A,B,C) = (m_0 + m_2 + m_3)'$$
$$= M_0.M_2.M_3 = \Pi(0,2,3)$$

Standard Forms

A Standard form can be obtained upon simplifying a Sum of Minterms or Product of Maxterms expression. A term in a SOP or POS form need not contain all the variables of a function.

S O P → Sum Of Products

$F_1 = y'+xy + x'yz'$

P O S → Product Of Sums

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

A non-standard form can be changed to SOP form.

$F3 = (AB + CD)(A'B'+C'D')$

$= A'B'CD + ABC'D'$

Truth Tables of 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			.	/		/		$\oplus$	+	$\downarrow$	Θ	'	$\subset$	'	$\supset$	$\uparrow$	

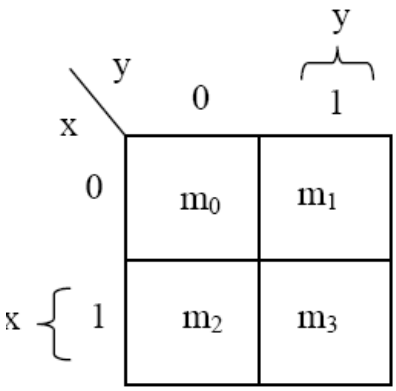
Boolean Expressions for the 16 Functions of Two Variables

Boolean Functions	Operator Symbols	Name	Comments
$F_0 = 0$		Null	Binary Constant 0
$F_1 = xy$	$x.y$	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
$F_6 = xy' + xy'$	$x \oplus y$	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
$F_9 = xy + x'y'$	$x \Theta y$	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

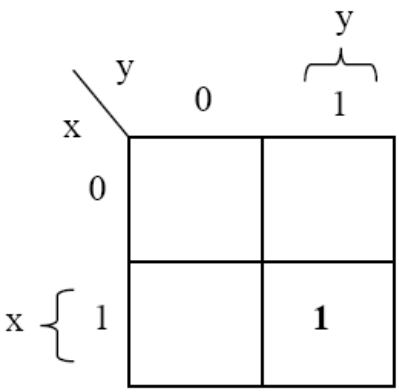
Karnaugh Map

The map is a diagram mad up of squares, each small square represents a minterm. Any function is made up of minterms and can be represented on the map. Adjacents minterms can be combined to form simpler terms.

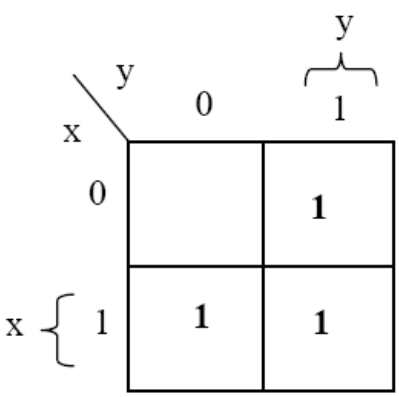
Two Variable Map



Two variable map

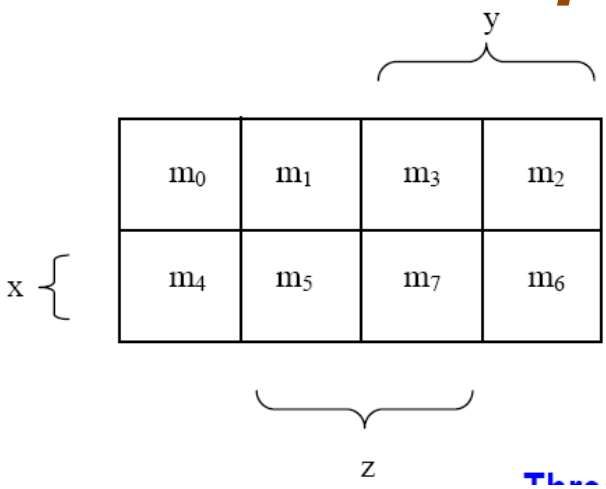


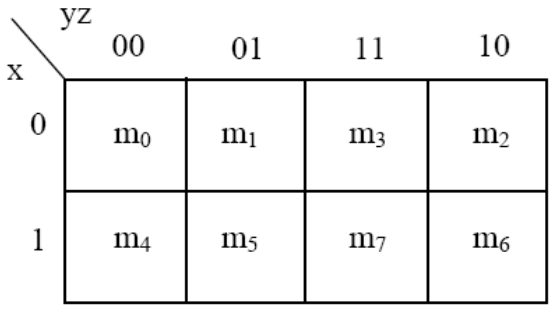
F=x.y



F=x+y

Three Variable Map





Three Variable Map

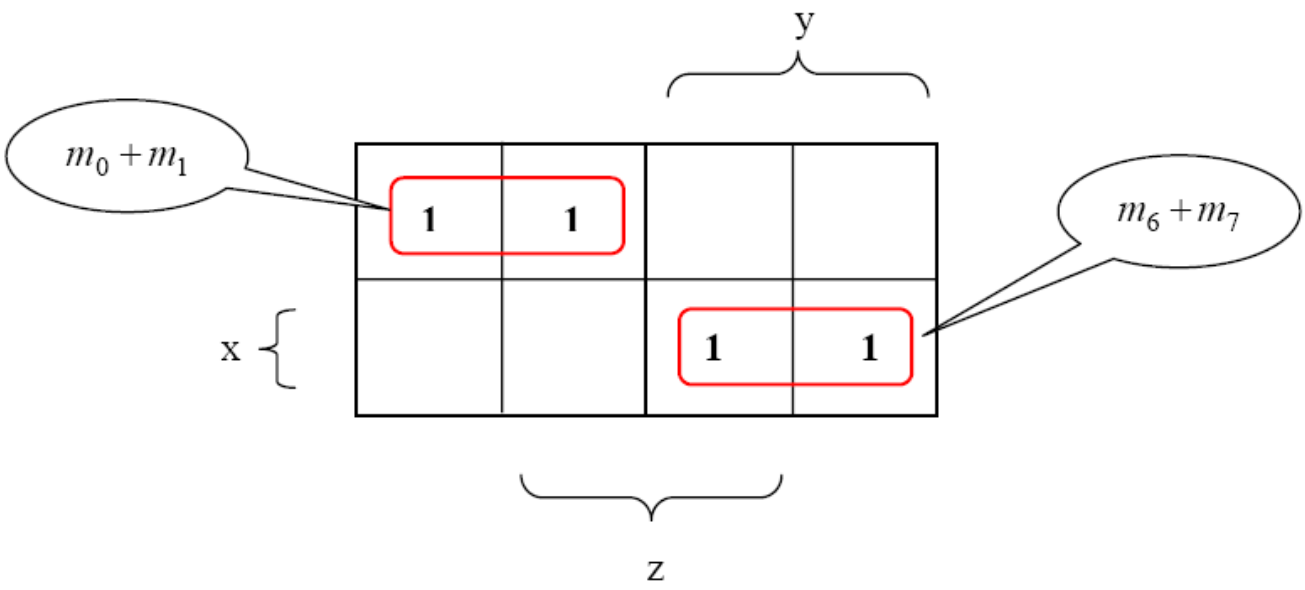
Simplification

Suppose a function of three variables x, y, z is given by:
F = Σ (0,1,6,7).
It is clear from the map that m0 and m1 are neighbours. Also m6 and m7.
Similarly:

$$m_0 + m_1 = x'y'z' + x'y'z = x'y'$$

$$m_6 + m_7 = xyz' + xyz = xy$$

This is shown in the map by the two groups of two minterms each.



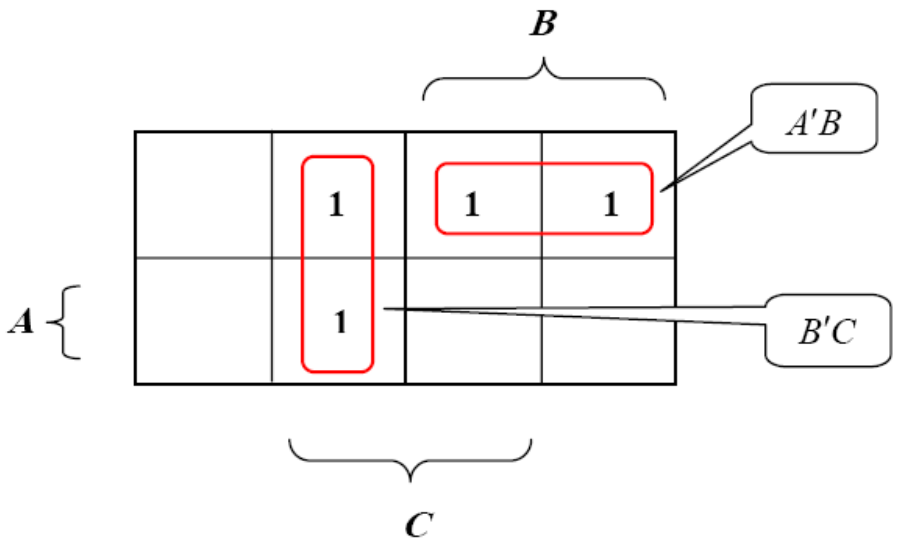
Example:

Simplify the Boolean function $F = A'C + A'B + AB'C$

Solution: To simplify the expression we have to put it in a sum of minterms form.

$$F = A'C + A'B + AB'C$$
$$= A'B'C + A'BC + A'BC' + A'BC + AB'C$$
$$= \sum(1,3,2,3,5)$$
$$= \sum(1,2,3,5)$$

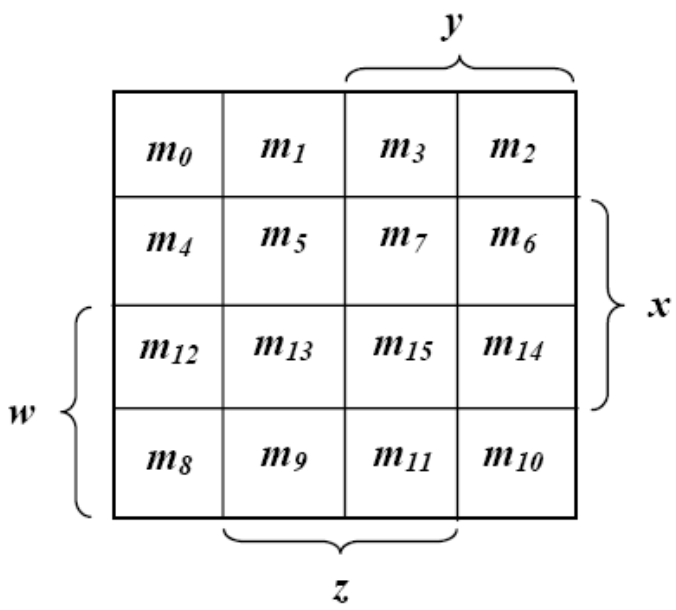
From the map:

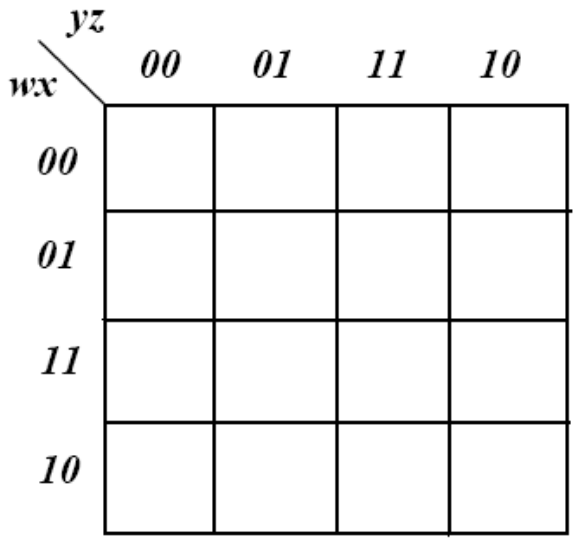


$$F = A'B + B'C$$

Four Variable Map

The map of four variables is shown below. It consists of sixteen small squares, arrange as a 4 x 4 small squares. Each small square represents one minterm. Each minterm has four neibouring (or adjacent) minterms.





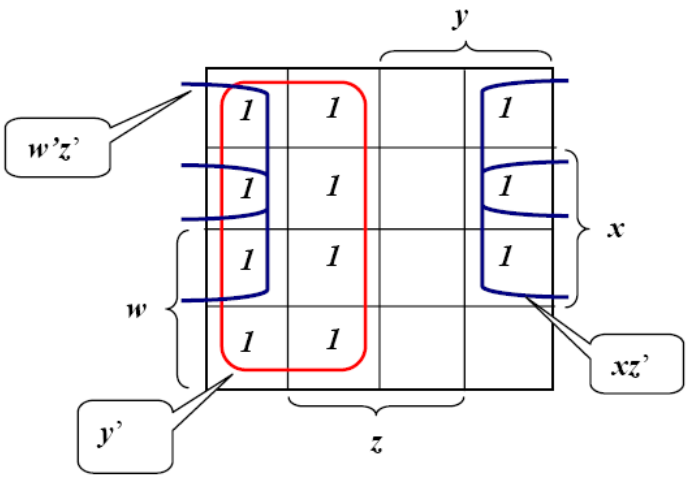
The map minimization of four variable functions depends on combining adjacent squares. The combination of adjacent squares help in the simplification of the functions.

- One square represents minterm (4 literals).
- Two adjacent squares represent a term with three literals.
- Four adjacent squares represent a term of two literals.
- Eight adjacent squares represent a term of one literal.
- Sixteen adjacent squares represent the function (F = 1).

Example:

Simplify the Boolean function

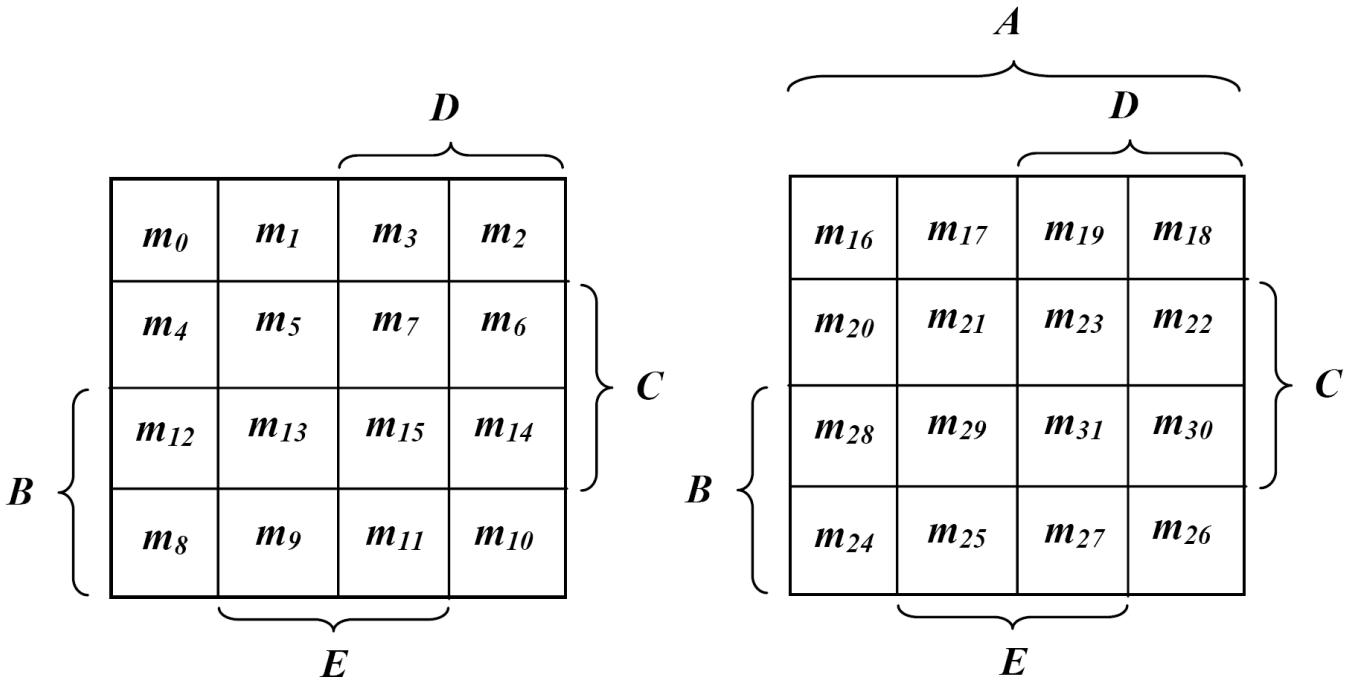
$F(w,x,y,z) = \Sigma(0,1,2,4,5,6,8,9,12,13,14)$ **using Karnaugh map.**



$$F = y' + w'z' + xz'$$

Five Variable Map

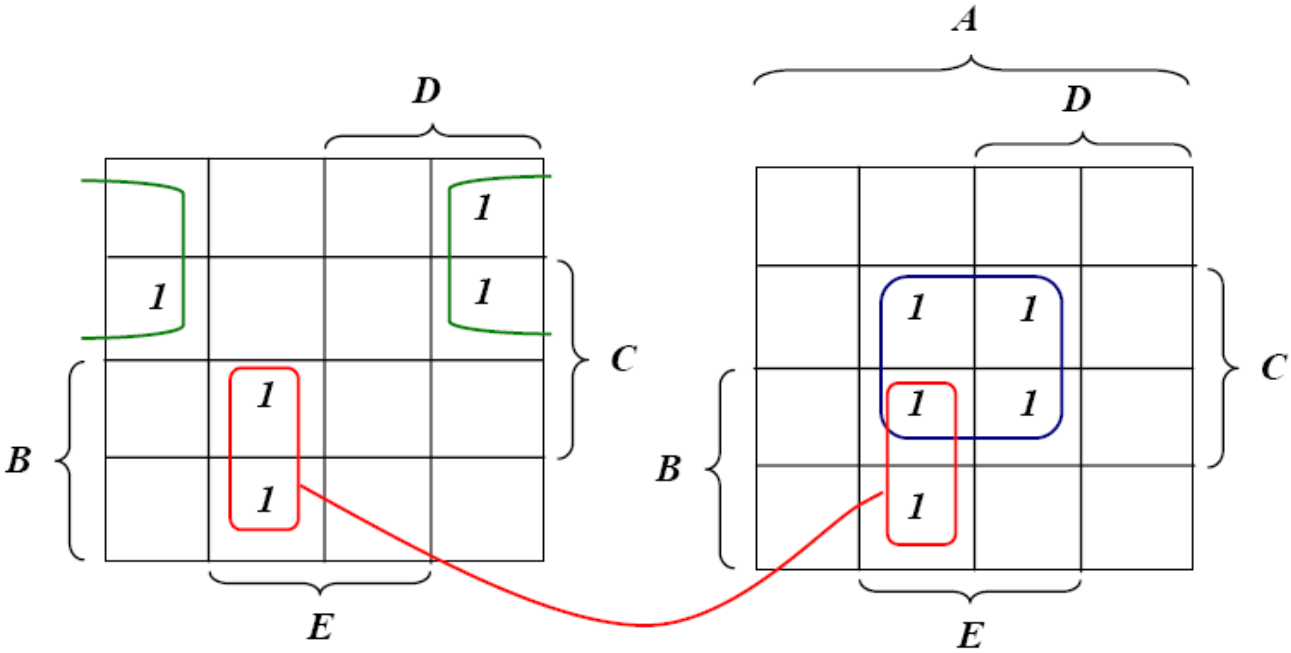
The map of five variables is shown below. It consists of two four variable maps.



Example:

Simplify the Boolean function

$$F(A,B,C,D,E) = \sum(0,2,4,6,9,13,21,23,25,29,31)$$



$$F(A,B,C,D,E) = A'B'E' + BD'E+ACE$$

Product of Sums (POS) Simplification

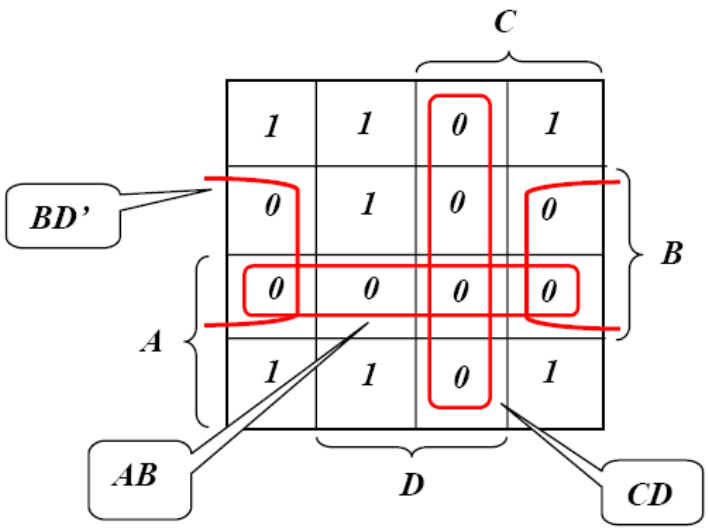
We combine the zeros of the function to obtain a simplified expression in a POS form. This follows from the fact that the zeros of the function are the ones of the complement of the function.

Example:

Simplify the Boolean function

$F(A,B,C,D) = \Sigma(0,1,2,5,8,9,10)$

a. In SOP form **b. In POS form.**



Combining the ones of the function, we get:

$$F = B'C' + B'D' + A'C'D$$

The complement of the function can be expressed in SOP by combing the zeros of the function:

F′ = AB+CD+BD′

We complement the last espression to get the function F in POS form,

$$F = \left(AB + CD + BD'\right)' = (A' + B')(C' + D')(B' + D)$$

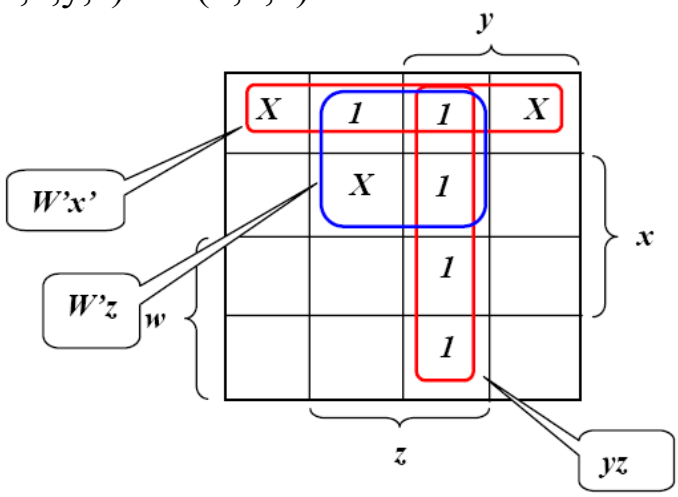
Don’t care conditions

When we design combinational logic circuits, we sometimes encounter situations where combinations of the input variables will never occur. In such cases, we can assume that these conditions can take on the value 1 or 0,

whichever going to give us a simpler expression. These conditions are indicated by letter X or D.

Example

Simplify the function F(w,x,y,z) = Σ(1,3,7,11,15), which has the don't care conditions d(w,x,y,z) = Σ(0,2,5)



$$F = w'x' + w'z + yz$$

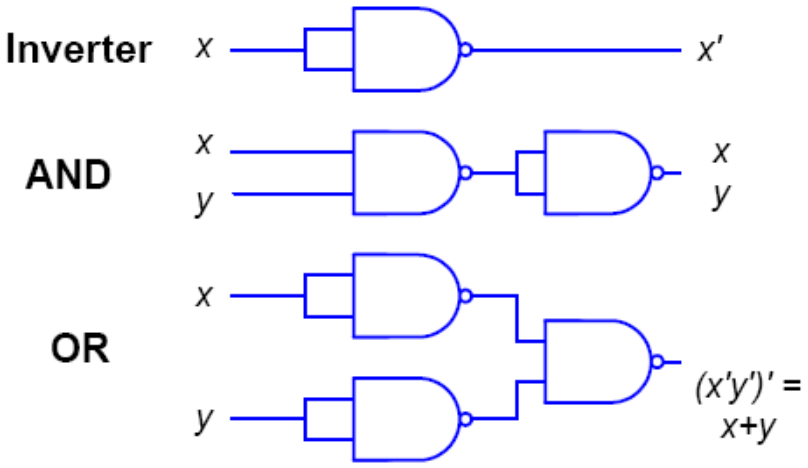
The Universal Gates:

The NAND and NOR gates are called the universal gates because they can be used for all types of gates. Digital circuits are frequently constructed with NAND or NOR gates rather

than with AND or 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.

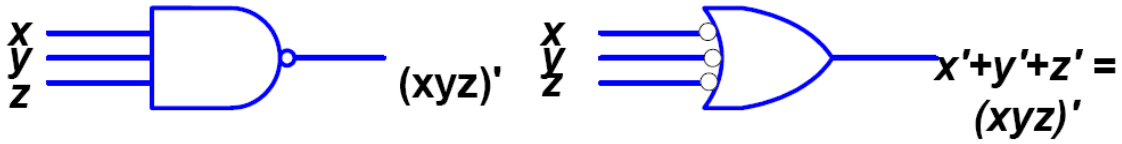
NAND Gate is a Universal Gate

The NAND gate is called the universal gate because it can be used for all types of gates. The basic AND, OR, and NOT gates can be implemented using NAND gates only.

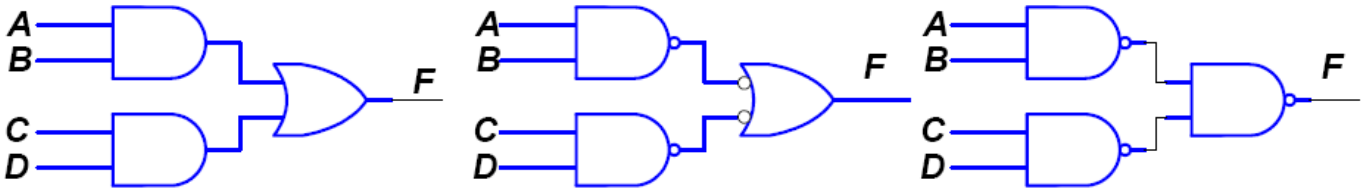


Logic Operations with NAND gates

Two equivalent graphic symbols for NAND gate are shown below:

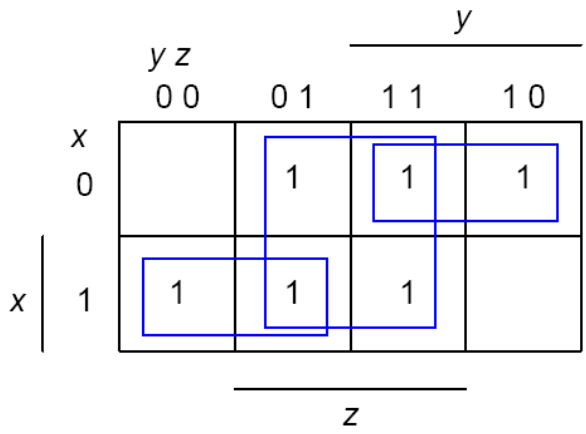


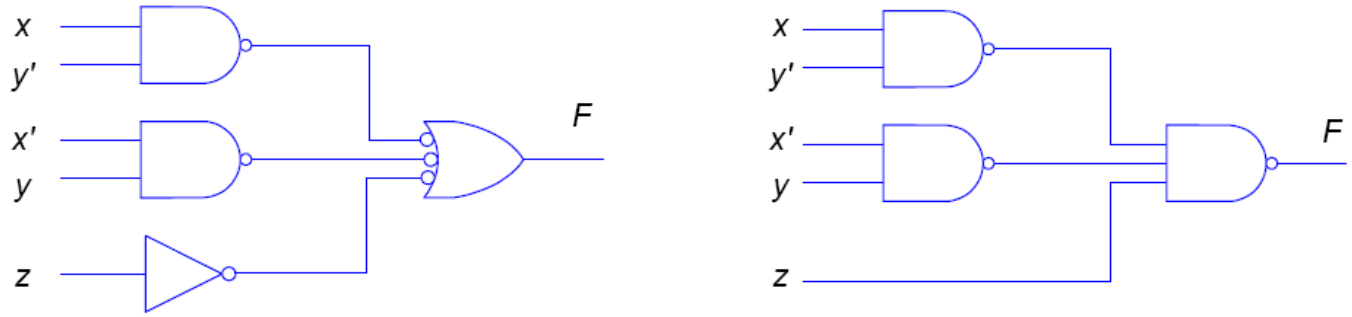
Example 1: Implement F = AB + CD using NAND gates



Example 2: Implement F = Σ(1,2,3,4,5,7) using NAND gates

Simplification of the function gives F = *xy'+ x'y + z*

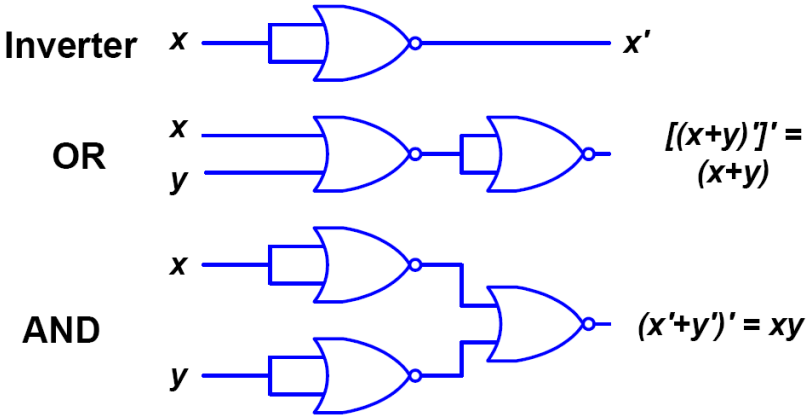




Two-level NAND implementation for F = xy'+ x'y + z

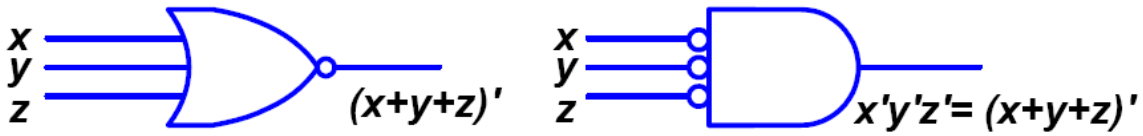
NOR Gate is a Universal Gate:

The NOR gate is called the universal gate because it can be used for all types of gates. The basic AND, OR, and NOT gates can be implemented using NOR gates only.

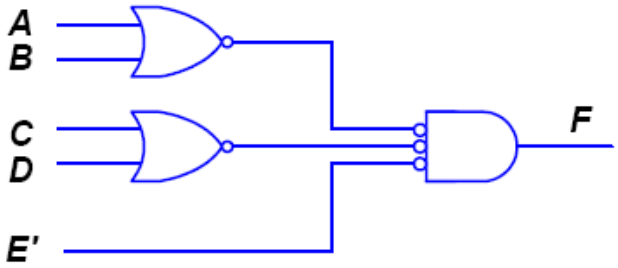


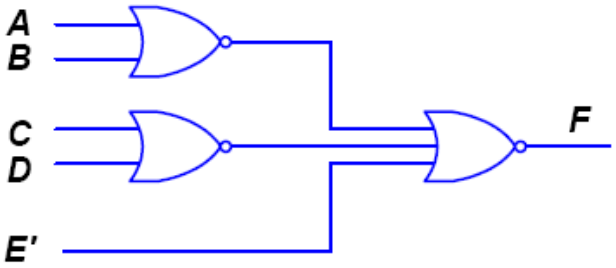
Logic Operations with NOR gates

Two equivalent graphic symbols for the NOR gate, are shown below:



Example: Implement F = (A + B)(C + D)E using NOR gates.

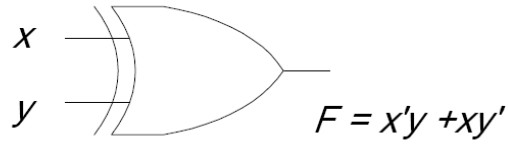




EXCLUSIVE-OR FUNCTION

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

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



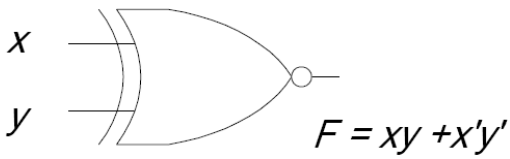
x	y	x ⊕ y
0	0	0
0	1	1
1	0	1
1	1	0

XOR Truth Table

The output of the XOR gate is 1 only when either input is 1, otherwise it gives 0s.

The exclusive-NOR, also known as equivalence function, performs the following Boolean operation:

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



x	y	x Θ y
0	0	1
0	1	0
1	0	0
1	1	1

XNOR Truth Table

The output of the XNOR gate is 1 only when both inputs are 1, otherwise it gives 0s.

The following identities apply to the XOR operation:

$x \oplus 0 = x$

$x \oplus 1 = x'$

$x \oplus x = 0$

$x \oplus x' = 1$

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

$x \oplus y = y \oplus x$

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

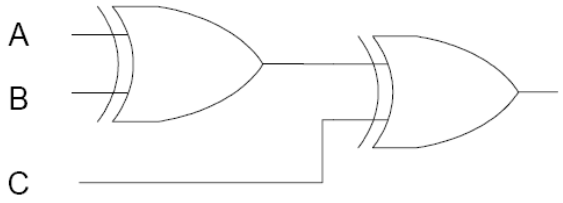
The exclusive-OR operation with three and four variables can be expressed as:

$A \oplus B \oplus C = \sum(1,2,4,7)$

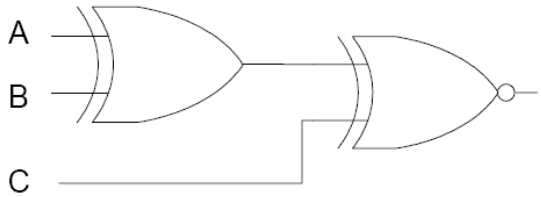
$A \oplus B \oplus C \oplus D = \sum(1,2,4,7,8,11,13,14)$

We can see from the above expressions that the XOR function is 1 only when an odd number of variables are equal to1. Hence, in general, the multi-variable XOR operation is defined as the odd function.

The 3-input odd function is implemented by means of 2-input XOR gates as shown below. The complement of an odd function is obtained by replacing the output gate with an XNOR gate.



3-input Odd function



3-input even function

Exclusive-OR gates are useful for generating and checking a parity bit that is used for detecting/correcting errors during transmission of binary data over communication channels.

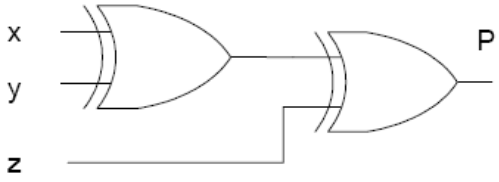
Example

Transmitting a 3-bit message with 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, which is an odd function and can be expressed as:

$$P = x \oplus y \oplus z$$

The truth table and the logic diagram for the parity generator is shown below.

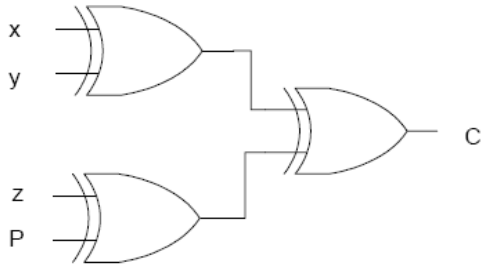
Message			***Parity***
x	*y*	*z*	
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1



3-bit even parity generator

The three bits in the message together with the even parity pit P are transmitted. The receiver at the destination checks for even number of 1's in the 4-bit message and generates an error C equal to 1 if the number of 1's in the message is odd. Here, again, we can use the odd function property of the XOR gate that produces an output of 1 if odd number of inputs is equal to 1. The truth table and the logic diagram for the parity checker is shown below.

4-bit Received Message				***Error***
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



4-bit even parity checker

Combinational Circuits

Logic circuits are either combinational or sequential. Combinational logic circuit consists of logic gates whose outputs at any time are determined from the present combination of the inputs. Sequential circuits consist of memory elements and logic gates.

Boolean Algebra Logic Analysis Procedure

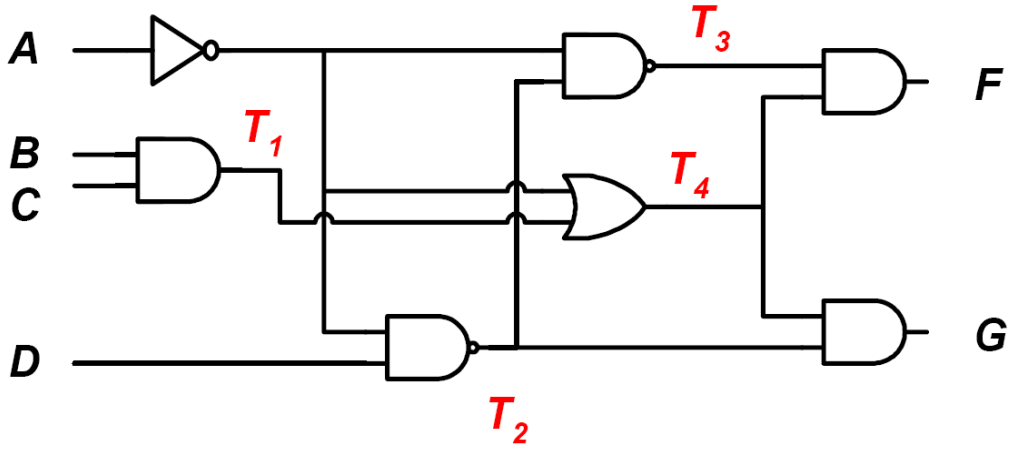
1. Obtain Boolean expression from logic diagram

1. Label all gate outputs that are a function of input variables. Obtain Boolean function for each gate.
2. Label all gate outputs that are a function of input variables and previously labeled gates. Obtain Boolean function for each of these gates.
3. Repeat step (b) until the outputs of the circuit are obtained.

2. Obtain the truth table from the logic diagram

1. Prepare the truth table for n input variables and 2n input combinations.
2. Label all gate outputs that are a function of input variables. Fill in the truth table for these outputs.
3. Label all gate outputs that are functions of input variables and previously labeled gates. Fill in the truth table columns for these outputs.
4. Repeat step (c) until the columns for all the outputs are obtained.

Example 1: Determine the Boolean functions for the outputs F and G as a function of the four inputs A, B, C, and D.



$$T_1 = BC$$

$$T_2 = (A'D)' = A + D'$$

$$T_3 = (A'T_2)' = A + T_2' = A + A'D = A + D$$

$$T_4 = A' + T_1 = A' + BC$$

$$F = T_3T_4 = (A + D)(A' + BC) = AA' + ABC + A'D + BCD$$
$$= ABC + A'D + BCD = ABC + A'D$$

$$G = T_2T_4 = (A + D')(A' + BC) = AA' + ABC + A'D' + BCD'$$
$$= ABC + A'D' + BCD' = ABC + A'D'$$

Example 2: Analyze the previous logic circuit by establishing the truth table for F and G.

Inputs								Outputs	
A	*B*	C	*D*	T_1	T_2	*T3*	T_4	*F*	*G*
0	0	0	0	0	1	0	1	0	1
0	0	0	1	0	0	1	1	1	0
0	0	1	0	0	1	0	1	0	1
0	0	1	1	0	0	1	1	1	0
0	1	0	0	0	1	0	1	0	1
0	1	0	1	0	0	1	1	1	0
0	1	1	0	1	1	0	1	0	1
0	1	1	1	1	0	1	1	1	0
1	0	0	0	0	1	1	0	0	0
1	0	0	1	0	1	1	0	0	0
1	0	1	0	0	1	1	0	0	0
1	0	1	1	0	1	1	0	0	0
1	1	0	0	0	1	1	0	0	0
1	1	0	1	0	1	1	0	0	0
1	1	1	0	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1

Design Procedure

1. Describe the problem (i.e., the problem statement).
2. Determine the available number of input variables and required output variables.
3. Assign letter symbols to the input and output variables.
4. Derive the truth table that defines the required relationships between inputs and outputs.
5. Obtain the simplified Boolean function for each output.
6. Draw the logic diagram.

Design of a Code Converter

Design a combinational logic circuit that will convert from a BCD code to Excess-3 code.

Design → 4 inputs → A, B, C, and C

4 outputs → w, x, y, and z

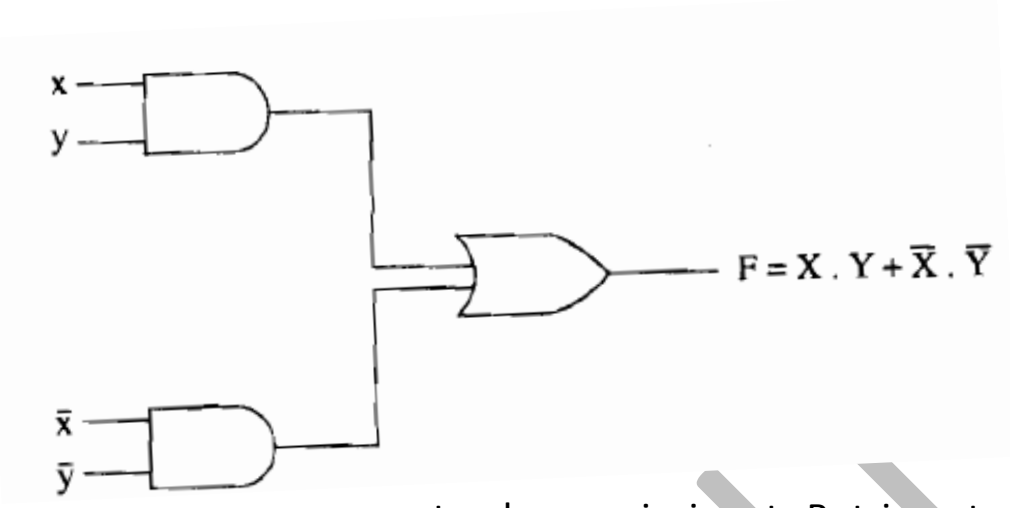
Truth Table:

Inputs (BCD Code)				Outputs(Excess-3)			
A	*B*	C	*D*	*w*	*x*	*y*	*z*
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

The six unused combinations are considered don't care conditions. These correspond to 10, 11, 12, 13, 14, and 15. Simplification of the output functions is made using Karnaugh maps and making use of the don't care conditions.

What is the Combinational Circuit?

Combinational circuits are interconnected circuits of gates according to certain rule to produce an output depending on its input value. A well-formed combinational circuit should not have feedback loops. A combinational circuit call be represented as a network of gates and, therefore, can be expressed by a truth table or a Boolean expression.



The output of a combinational circuit is related to its input by a combinational function, which is independent of time. Therefore, for an ideal combinational circuit the output should change instantaneously according to changes in input. But in actual cases there is slight delay. This delay is normally proportional to the depth or numbers of levels i.e. the maximum number of gates lying on ally path from input to output. For example, the depth of the combinational circuit in figure is two.

The basic design issue related to combinational circuits is: the Minimization of number of gates. The normal circuit constraints for combinational circuit design are:

- The depth of the circuit should not exceed a specific level.
- Number of input lines to a gate (fan in) and to how many gates its output can be fed (fan out) are constraint by the circuit power constraints.

Minimization of gates:

The simplification of Boolean expression is very useful for combinational circuit design. The following two methods are used for this- Algebraic Simplification and Karnaugh Maps.

Binary Addition:

We know that, in binary addition, to add two numbers, following rules have to be followed:

Input	Output

First Digit (A)	Second Digit (B)	Carry (C)	Sum (S)
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

From the above table, we know that, the Sum (S) follows the XOR function of Boolean algebra and the carry (C) follows the AND function. So a simple adder can be constructed, as the above table, is given below:

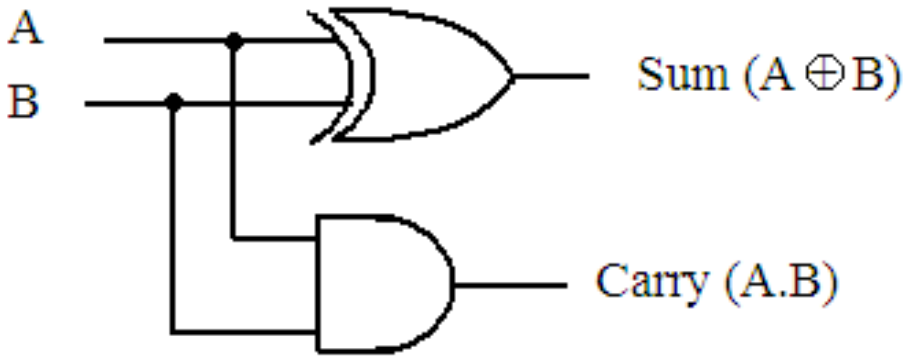


Fig. The half adder

The above figure adds two bits, so it is called the *Half Adder*.

Adder

Adder is a logical element that adds different bits which are given by users. User gives the input signals. The input is always either 0 or 1. So adder adds such 0s and 1s. digital computers use these logical circuits to perform binary addition of two or more bits.

Half Adder

Half adder is a logical circuit that adds two bits. Here the following truth table shows the binary addition of two bits.

Input		Output	
First Digit (A)	Second Digit (B)	Carry (C)	Sum (S)
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Table shows that the carry output is simply an AND function and Sum output is an XOR function, When we use these two gates from the inputs, the resulting circuit is like below:

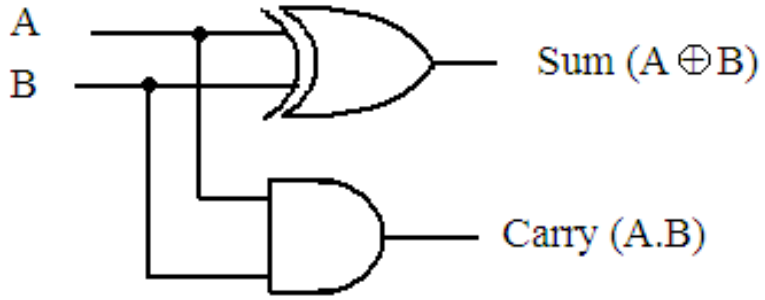
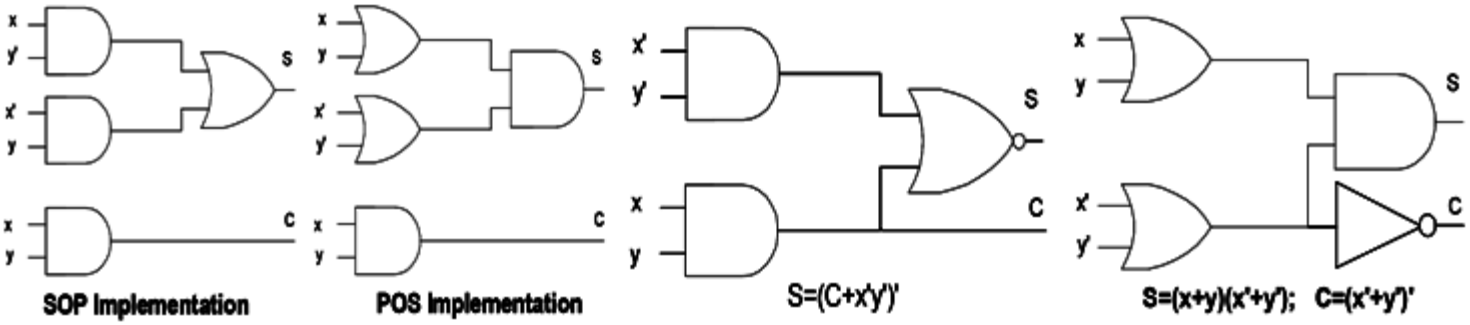


Fig. The half adder

A half adder can be mad by using different gate topologies as shown below:



Full Adder

Full adder is a logic circuit that adds three binary digits. Full adder can be obtained by adding plus one Carry input. Like half adder, a full adder also has the output a Sum and a Carry. So, we will have both an input carry and output carry that are denoted by C_{in} and C_{out}. The sum is represented by S. The truth table for a full adder (3-input adder) is given below:

Input Signals		Outputs		
A	**B**	**C_{in}**	**C_{out}**	**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

By this table, we can observe that, when two or more than two inputs are 1, the C_{out} will be 1, otherwose it will be 0, i.e., A AND B OR B AND C OR C AND A must be 1.

$\Rightarrow C_{out} = A.B + B.C + C.A$

This expression is the result of C_{out}.

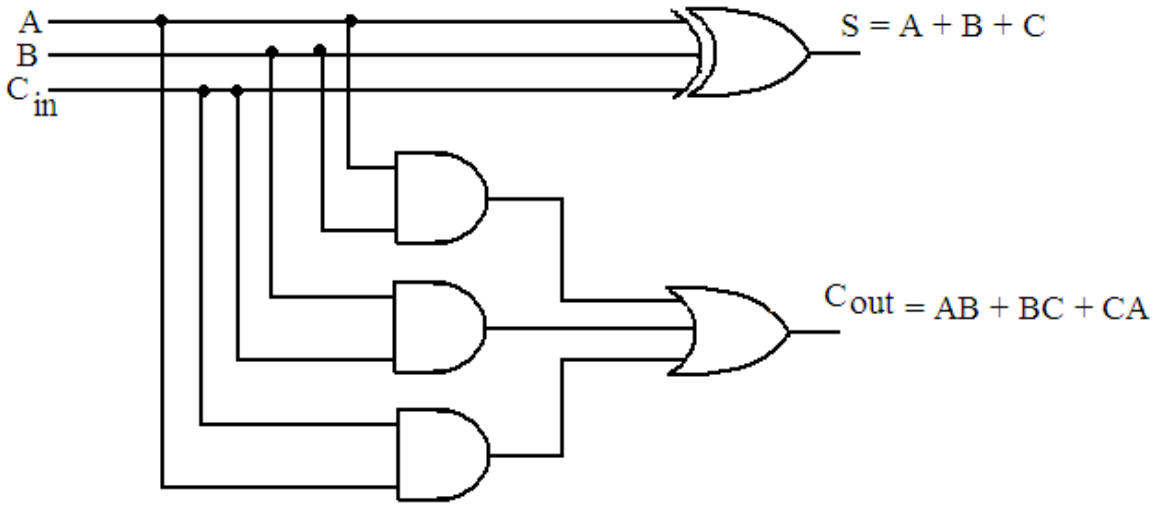


Fig. Logic Diagram of a Full Adder

By the table, it is clear that, the S will be 1 if odd inputs are 1. So it is XOR operation among these 3 inputs.

$\Rightarrow S = A \oplus B \oplus C$

This expression is the result of the Sum (S) output.

So, we have,

$\Rightarrow C_{out} = A.B + B.C + C.A$ (i)

$\Rightarrow S = A \oplus B \oplus C$ (ii)

From these two expressions, we can create the circuit of full adder.

So, SUM is 1, when the number of 1s input is odd and $CARRY_{out}$ will be 1 when two or more inputs are 1s. The circuit above is too complicated to design when it will be used for further other circuits. So, to simplify the circuit, a symbol is used, which is shown below:

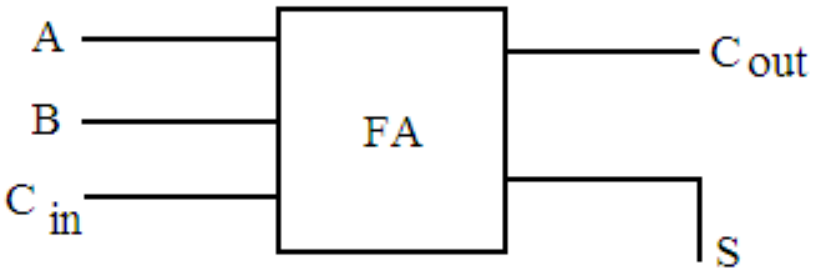


Fig. Block diagram of a Full Adder (FA)

Binary Adder

The binary adder is a logic circuit that can add two binary numbers. It is also called as Ripple Adder. Two binary numbers, each of n bits, can be added using a binary adder, a cascade of n full adders; each full adder handles one bit. Each C_{out} of a full adder is connected to the C_{in} of the higher full adder. The C_{in} of the least significant full adder is set to 0.

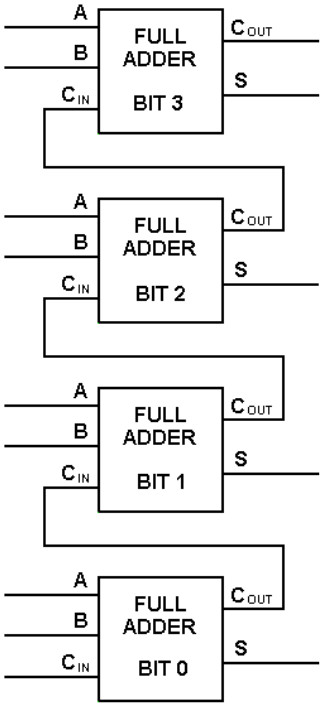
Let us consider, two binary numbers are, $A_3A_2A_1A_0$ and $B_3B_2B_1B_0$, where A's and B's are the bits of numbers. By adding these binary numbers, we get,

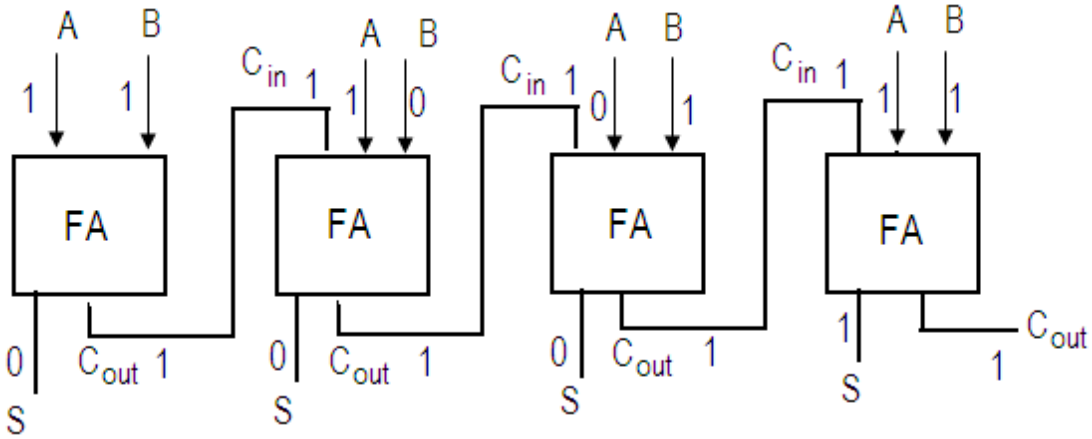
Carry: $C_3C_2C_1$
First Number: $A_3A_2A_1A_0$
Second Number: $B_3B_2B_1B_0$
Result: $C_4S_3S_2S_1S_0$

Lets take an example, A = 1011 and B = 1101, then

Carry: 1 1 1
First Number: 1 0 1 1
Second Number: 1 1 0 1
Result: 1 1 0 0 0

Representing this problem by logic circuit, using full adder, we have:

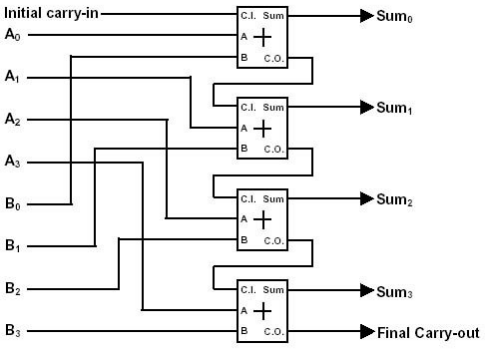


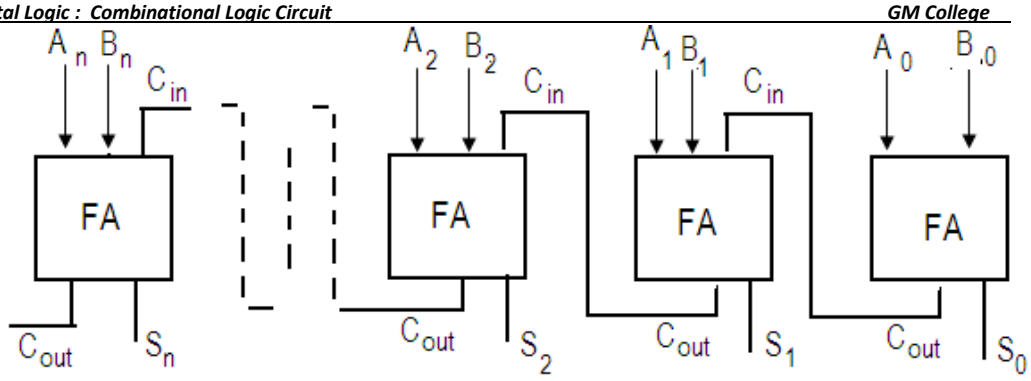


4-bit binary parallel adder can be implemented in integrated circuit form by cascading 4 full adders as shown below. The disadvantage of this adder is the possible slowing down of the addition due to the carry propagation time.

So, the logic circuit of a binary adder is:

Or it looks like when n-bit two numbers are added:

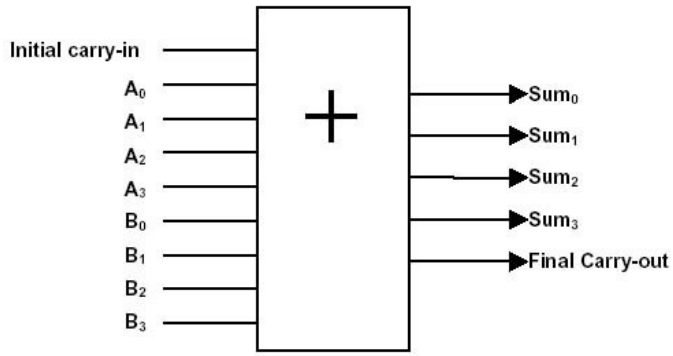




Sometimes it can be drawn as:

So, to perform multi-bit addition, the full adder must be allocated for each bits to be added simultaneously. For example, to add two 5-bit numbers, five full adder with carry is needed. The C_{out} of lower order of magnitude will become C_{in} for higher order of magnitude, which is shown in the figures.

This is will add two 4-bit binary numbers. The block diagram of a full adder is shown below. The below figure is a 4-bit binary adder.



Full Subtractor

Subtractor is a logic circuit that subtracts two binary numbers. It is generally used by ALU and other devices of computer to perform the subtraction operation. We know that the full adder is:

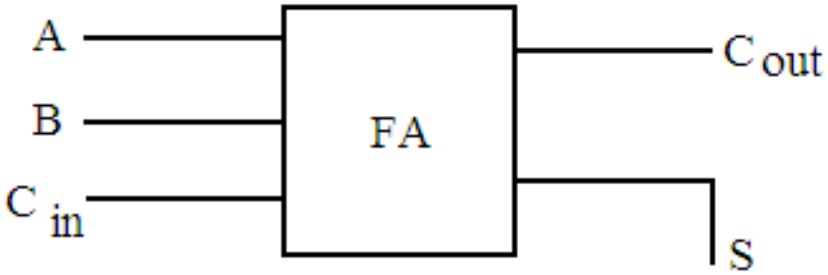


Fig. Block diagram of a Full Adder (FA)

Suppose two binary numbers $A_3A_2A_1A_0$ and $B_3B_2B_1B_0$ are added, so the expression will be,

$A_3A_2A_1A_0 + B_3B_2B_1B_0$

In digital system, for subtraction, the complement methods are generally used. So the exprssion - $A_3A_2A_1A_0 - B_3B_2B_1B_0$ is represented as:

$A_3\,A_2\,A_1\,A_0 + (\ \overline{B_3}\ \ \overline{B_2}\ \ \overline{B_1}\ \ \overline{B_0}\ )$ for 1's complement, and

$A_3\,A_2\,A_1\,A_0 + (\ \overline{B_3}\ \ \overline{B_2}\ \ \overline{B_1}\ \ \overline{B_0}\ ) + 1$ for 2's complement.

The exact result will be produced by the 2's complement method. Hence, each bit of 'B' is required to be complemented and 1 is added once, i.e., the LSB of A, $\overline{B}$ and 1 is to be added. So the first C_{in} is to be 1, then the figure will be:

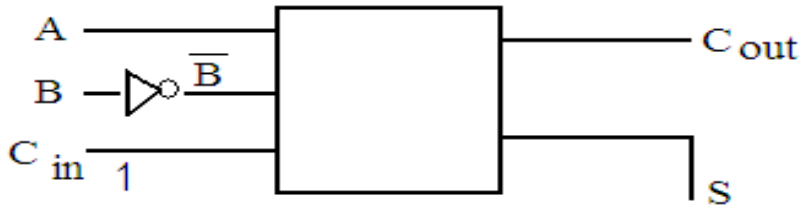


Fig. Full Subtractor

Binary Subtractor

We get the result of A – B, if we invert B and 1 with the lowest order (LSB) C_{in}. So the binary subtractor will be:

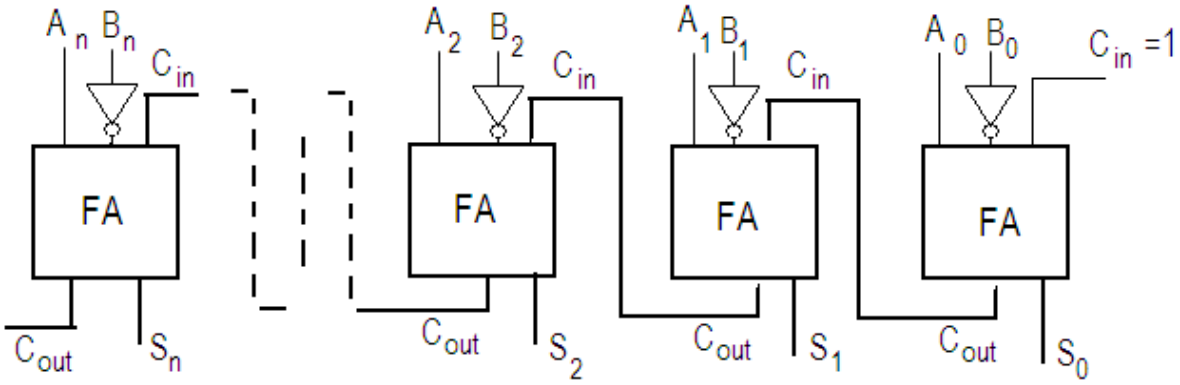


Fig. Logic diagram of an n-bit subtractor

Binary subtractor is a series of full adder which performs the subtraction operations.

Binary Adder-Subtractor

A single circuit can be used for addition and subtraction by using XOR gates and control input to control whether we will add or subtract on any given occasion. The XOR gate will leave the B input number unchanged with the control input 0. In this condition, it acts as binary adder and with the control input 1, the XOR gates will invert the B input number to form its 1's complement and will also add 1 through initial carry. This changes B to its 2's complement addition. So the output result will actually be A.B.

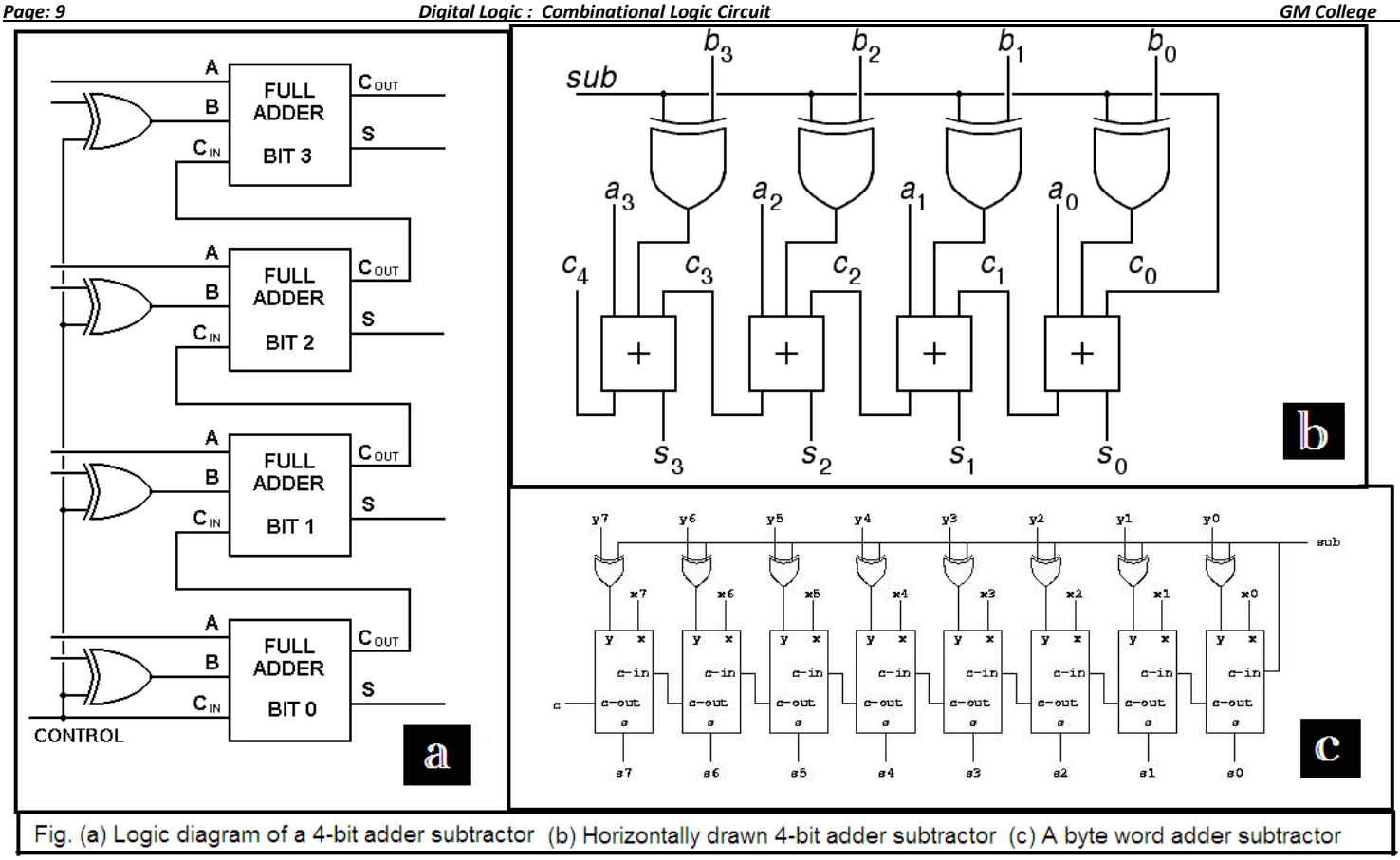


Fig. (a) Logic diagram of a 4-bit adder subtractor (b) Horizontally drawn 4-bit adder subtractor (c) A byte word adder subtractor

So, if we apply a logic 0 to the control input, the adder adds and acts as a binary adder. If we use a logic 1, the adder subtracts and acts as a binary subtractor for unsigned numbers.

Multiplexer (MUX)

A multiplexer is a logic circuit that converts several input signals into one signal. A multiplexer interleaves many transmissions onto a single circuit or channel. The digital multiplexer circuit is used for combining two or more digital signals into a single output signal by placing inputs into a single output at different time. So, it is also called time division multiplexing, but in electronics it is popularly known as MUX. It is also called data selector. It selects only one input signal among many inputs for producing output. At any given time, a mux accepts several digital data inputs and selects one of them to pass on to the output. It is a device for combining several digital signals into an aggregate bit stream.

MUX selects only one input at a time with the help of SELECT input also referred to as CONTROL or ADDRESS. The functional diagram of a multiplexer is given below:

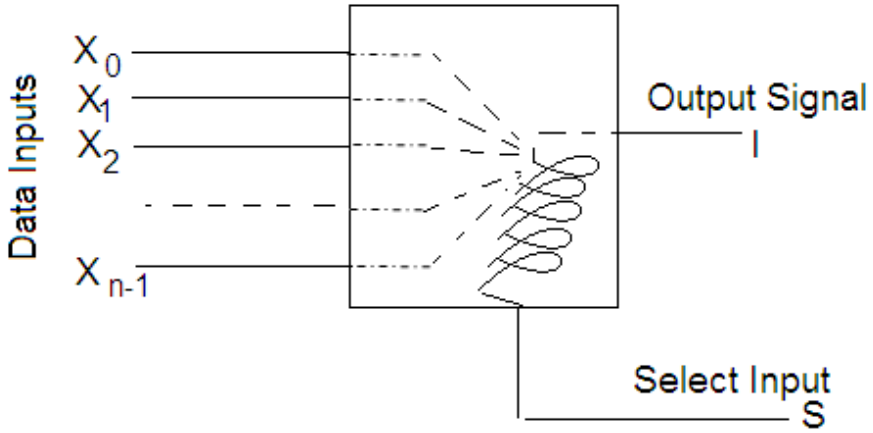


Fig. Functional Diagram of a digital multiplexer

The select input acts as like a controller that controls which data input will be switched to be the output. For particular SELECT input, output will have equal data of input x_0 and for another particular SELECT input MUX has output Z equal to another input x and so on. MUX selects one output out of 'n' numbers of inputs and transmits selected data to a single output channel. This operation is called multiplexing.

We can create two-input, four-input, 8-input, 16-input and more inputs multiplexer by using 'n' number of SELECT inputs, where 'n' is the exponential power of 2, i.e., 2^n. For 2-input multiplexer, $2= 2^1$; so, n=1. That means, it requires one select input. Likewise, for 4-bit multiplexer, $4 = 2^2$, n=2; so, 2 select inputs are required. Similarly, For 8-input MUX, 3 select inputs are required and for 16-bit mux, 4 select inputs are required.

Multiplexers are created by sum-of product method. By possible combination of SELECT inputs, we can easily create digital multiplexers.

Basic 2-input MUX

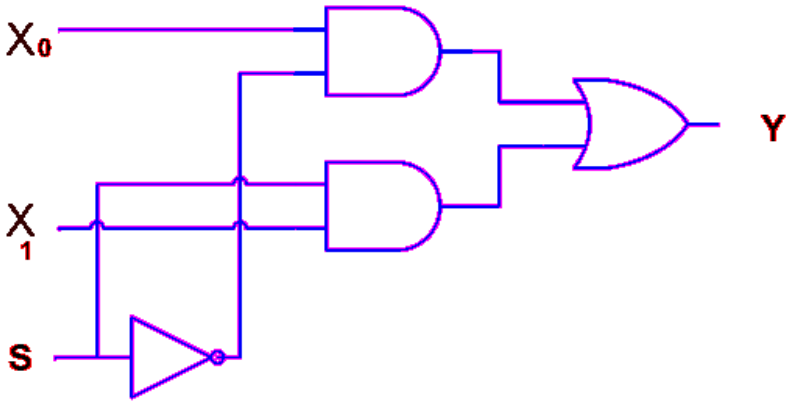
A 2-input multiplexer is a digital circuit that accepts two data inputs and produces only one at a time. The SELECT input selects which one of the input signals will be selected for output. For 2-input MUX, one select input is required. The possible value for a SELECT input is either 1 or 0. So, the function for 2-input MUX is:

$Z = X \overline{S} + X S$ (using complement for 0 and non-complement for 1).

Converting the first X by X_0 and the second X by X_1, the output will be,

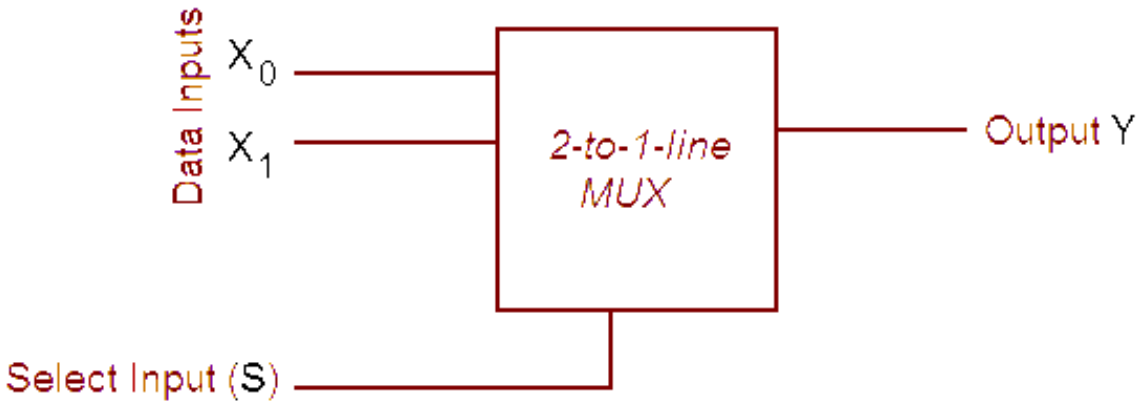
$Z = X_0 \overline{S} + X_1 S$

The Logic diagram for the above expression is,

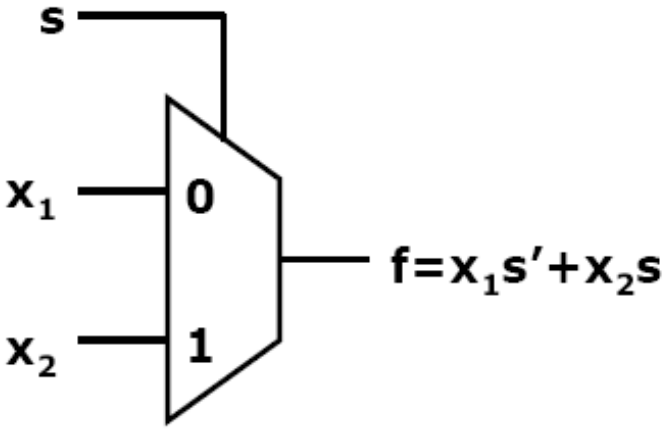


The 2-input MUX is also called 2-line-to-1-line multiplexer, or simply 2-line multiplexer.

The block diagram for a 2-line-to-1-line multiplexer is like:



Sometimes the Multiplexer is represented as:



The 4-input MUX

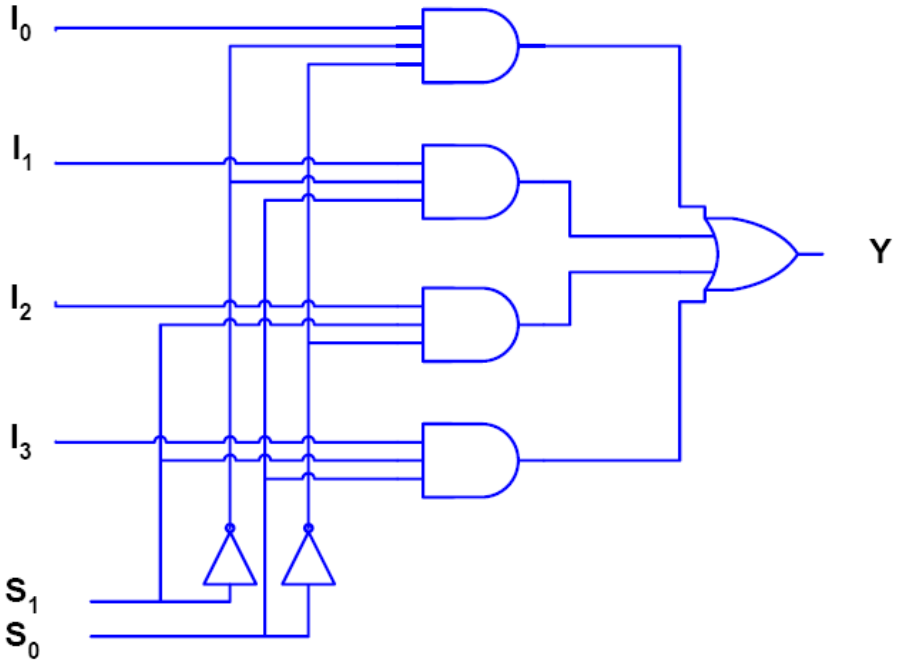
A *4-input* (or *4-to-1 line multiplexer*) consists of four AND gates. Each input is connected to one AND gate. Selection lines S1 and S0 are decoded to select a particular AND gate. The outputs of the AND gates are applied to a single OR gate that provides the output of the multiplexer Y. The output of the multiplexer is then given by:

$$Y = S_1' S_0' I_0 + S_1' S_0 I_1 + S_1 S_0' I_2 + S_1 S_0 I_3$$

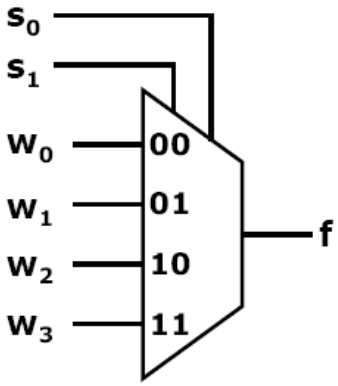
The function table of the multiplexer is shown below:

Select Inputs		Output
S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

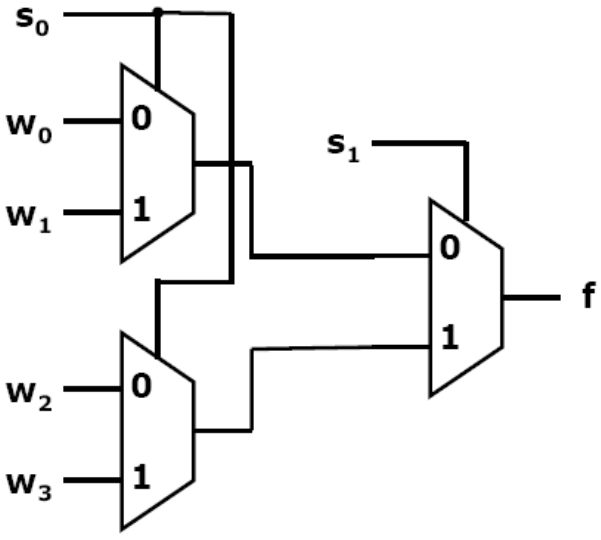
The logic circuit for a 4-line-to-1-line MUX is given below:



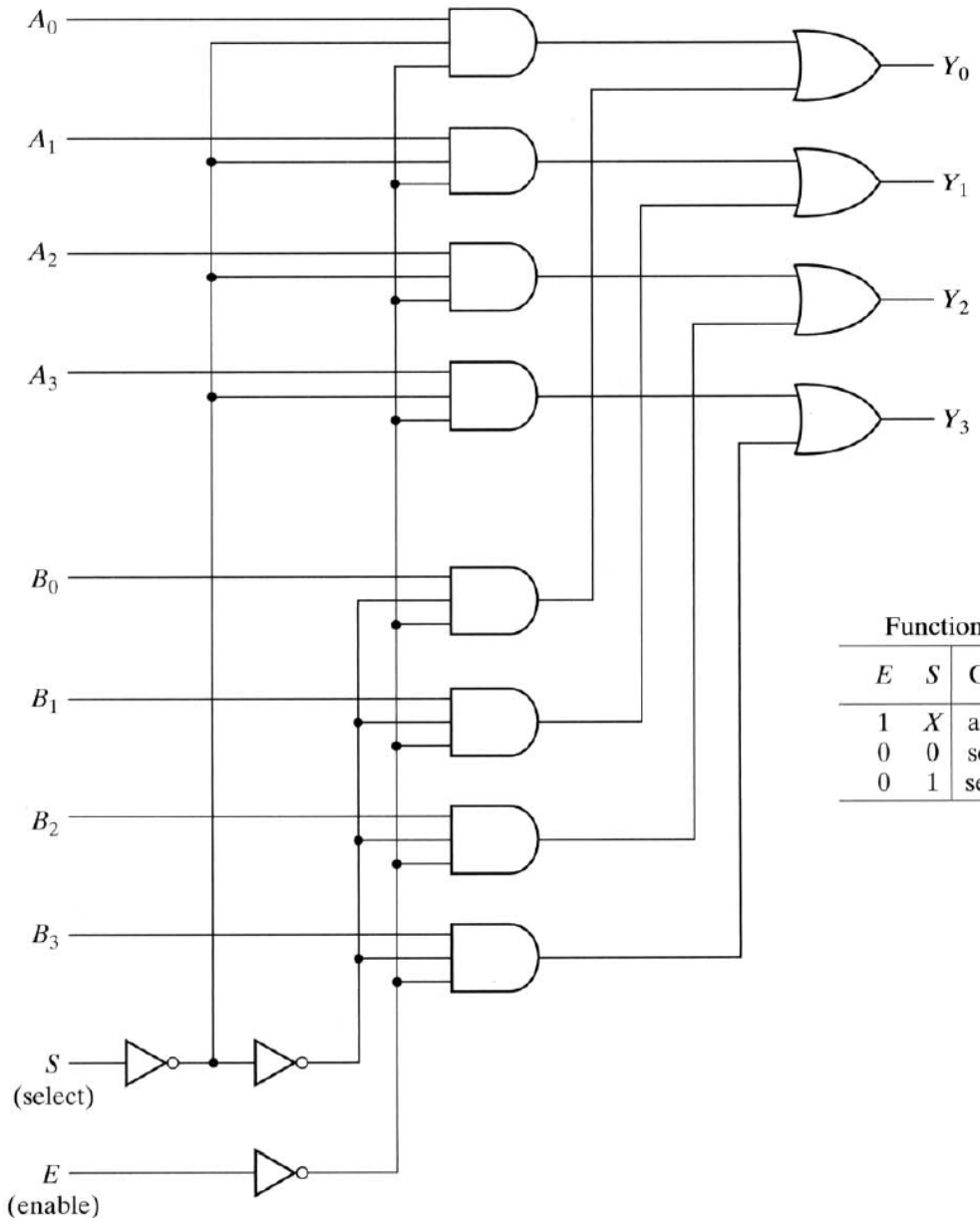
The block diagram of 4-input MUX is sometimes represented as:



A 4-input MUX can also be created using two 2-input MUX.



The 8-input Multiplexer (8-line-to-1-line MUX, Quad-to-one MUX)



Function table

E	S	Output Y
1	X	all 0's
0	0	select A
0	1	select B

Boolean Function Implementation

A multiplexer is a decoder and an OR gate that provides the output. The multiplexer can be used to implement Boolean functions of n variables. This can be achieved using either 2^n-to-1 multiplexer or $2^{(n-1)}$-to-1 multiplexer.

Using 2^n-to-1 multiplexer

The n variables are connected to the n selection lines. Each input of the multiplexer is set to 0 or 1, depending on which minterm of the function is present.
Example: Implement F(x,y,z) = Σ(1,2,6,7) using 8-to-1 multiplexer.

Solution: Connect the variables x, y, z to the selection inputs S_2, S_1, and S_0. Then set $I_0 = I_3 = I_4 = I_5 = 0$ and $I_1 = I_2 = I_6 = I_7 = 1$.

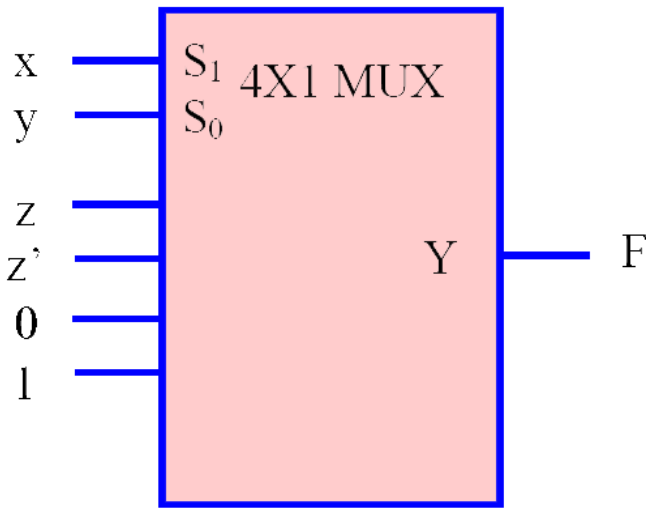
2. Using $2^{(n-1)}$-to-1 multiplexer

We connect (n-1) variables to the selection lines. The multiplexer inputs are going to be either 0 or 1 or the remaining variable or the complement of the remaining variable.

Example: Implement F(x,y,z) = Σ(1,2,6,7) using 4-to-1 multiplexer.

Solution: Connect the variables x and y to the selection inputs S_1, and S_0. The inputs of the multiplexers can be obtained from the truth table as shown below.

x	y	z	F	
0	0	0	0	F=z
0	0	1	1	
0	1	0	1	F=z'
0	1	1	0	
1	0	0	0	F=0
1	0	1	0	
1	1	0	1	F=1
1	1	1	1	



Application of MUX

- A MUX can be used as 2 X 2 cross bar. It is useful in applications where it is necessary to connect one set of wires to another set of wires, where the connection pattern changes from time to time. Telephone switching networks is an example.
- In programmable devices (PLDs, CPLDs and FPGAs) programmable switches connect wires inside the device. These can be implemented with multiplexers.
- MUXs can be used to synthesize logic functions.

- Can do the job for other special functions. For example, a 3-input XOR can be implemented with two 2-input MUXs.

Demultiplexer (DEMUX)

A demultiplexer is a logic circuit that converts single input signals into several signals. A demultiplexer interleaves single transmission onto many circuits or channels. The digital demultiplexer circuit is used for producing single signal into two or more digital signals. So, it is also called time division demultiplexing, but in electronics it is popularly known as DEMUX. It is also called data distributor. It combines the possible outputs for a single input for producing output. At any given time, a demux accepts single digital data input and distributes into many signals to pass on to the output. It does the just reverse operation of multiplexer. A circuit that performs the opposite, placing the value of a single input onto one for multiple outputs is a demultiplexer.

DEMUX distributes many signals from a single data input signal with the help of SELECT input also referred to as CONTROL or ADDRESS. The functional diagram of a multiplexer is given below:

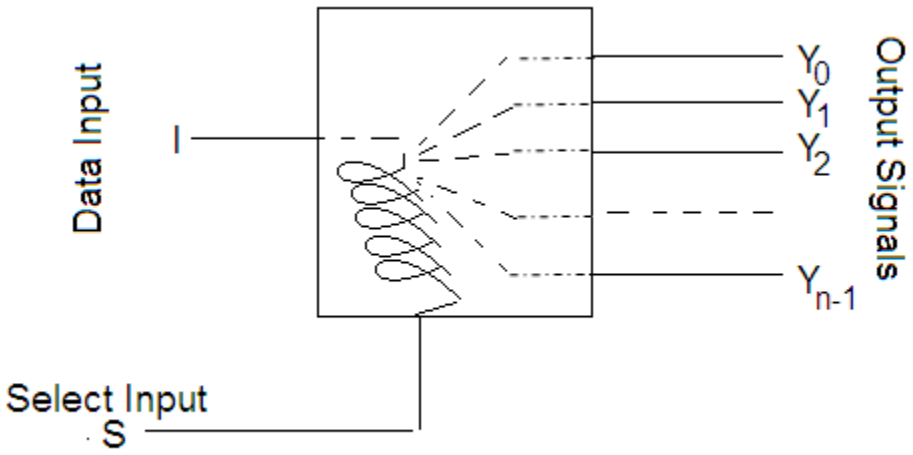
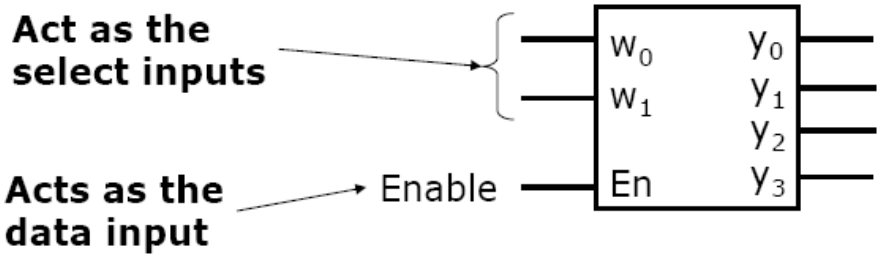


Fig. Functional Diagram of a digital demultiplexer

A decoder with enable input can function as a demultiplexer. A demultiplexer is a combinational circuit that has one input and up to 2^n outputs and it directs the input to an output depending on the values of n selection lines.
The Block diagram for a MUX can be created as:



Basic 1-line-to-2-line Demultiplexer

The basic 1-to-2-line demux contains a single data input and two outputs. The outputs are generated by using the select inputs.

Here, the required number of output is 2. So, select input 'n' (from 2^n is no. of output) is 1. The logic circuit of demux is:

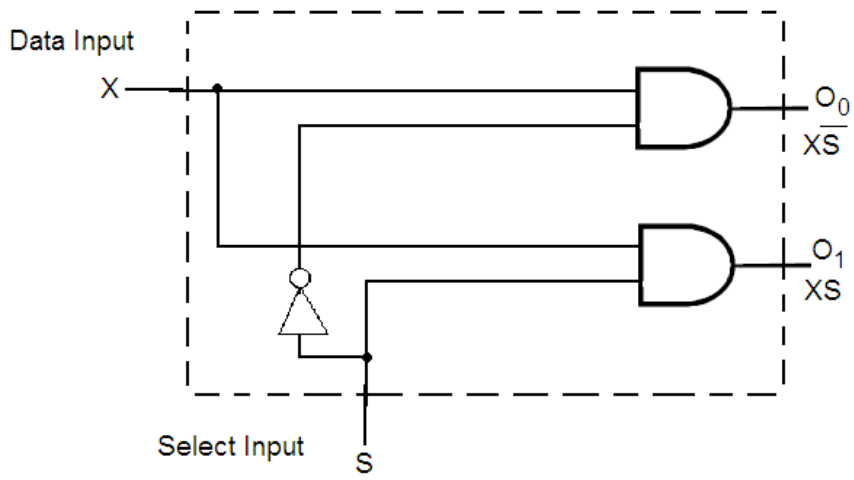


Fig. Logic Circuit for a Basic 1-to-2-line Demultiplexer

The function table of the 1-to-2-line demultiplexer is shown below:

Select Input S	Outputs	
	O_1	O_0
0	0	X
1	X	0

Here, X is the data input.

1-line-to-4-line Demultiplexer

It contains 1 data input and 4 output. It requires two select inputs. So, it is also called 2-line-to-4-line DEMUX.

Here the outputs are:

$x=XS_0'S_1'$, $XS_0'S_1$, XS_0S_1', XS_0S_1

The functional table for the 1:4 DEMUX is:

Select Inputs	Outputs

S_0	S_1	O_3	O_2	O_1	O_0
0	**0**	0	0	0	X
0	**1**	0	0	X	0
1	**0**	0	X	0	0
1	**1**	X	0	0	0

Decoder

Digital Decoder is a logic circuit that decodes encoded information. A decoder decodes (recodes) binary number by using demultiplexer. A decoder is a combinational circuit that converts binary information from *n* input lines to a 2*n* unique output lines. It accepts the set of binary number inputs and activates only the output that concerns to these input numbers. It determines which input is present and activates only one output correspondent to that number, and all other outputs remain inactivate. A binary decoder has ***n*** data inputs and ***2n*** outputs. Only one output is asserted at any time (***one-hot encoded***) and each output corresponds to one valuation of the inputs.
The block diagram can be created as:

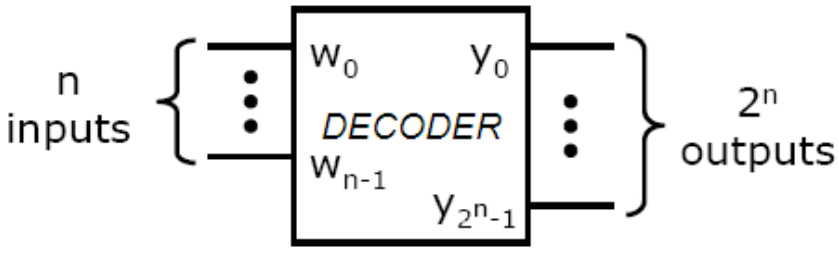
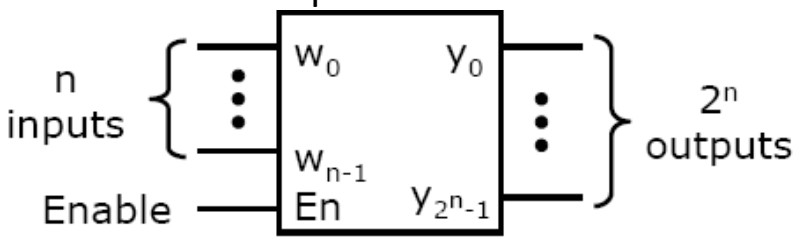


Fig. Block Diagram of a Decoder

Sometimes an ENABLE input is used that enables/disables the decoding job of the decoder. This is represented as:



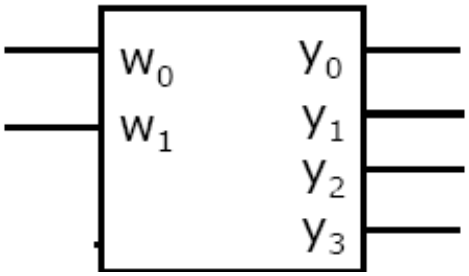
If En=0, none of the decoder outputs is asserted, and If En=1, one of the outputs is asserted according to the valuation of the inputs.

Applications of Decoder

- Microprocessor memory system: selecting different banks of memory.
- Microprocessor I/O: Selecting different devices.
- Memory: Decoding memory addresses (e.g. in ROM).
- Displaying information into human readable form (like seven segment decoder).

2-to-4 decoder circuit

It accepts two binary values and produces one of the four value depending up on the combinations of the input signals. The block diagram of the 2-to-4-line decode is:



The truth table for 2-to-4 decoder is:

w_1	w_0	y_0	y_1	y_2	y_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

The logic diagram for a 2-to-4 decoder is:

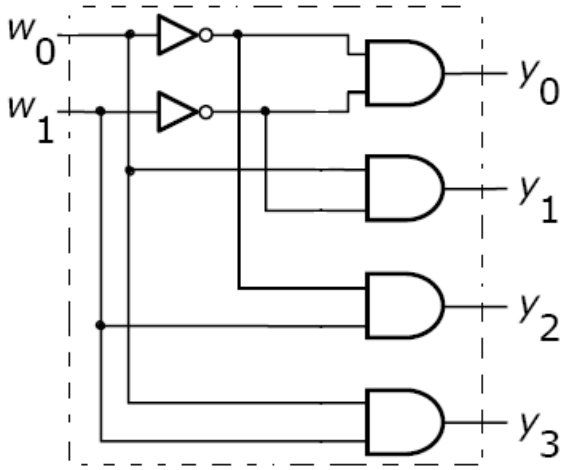
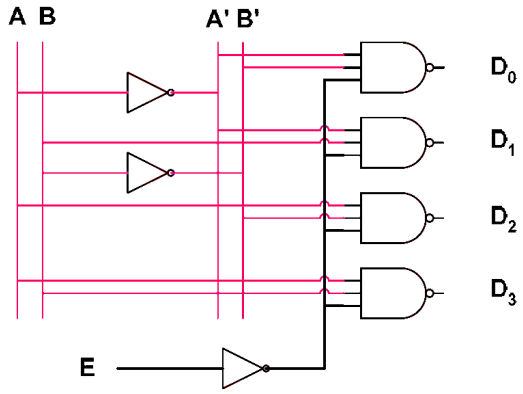


Fig. Logic diagram of 2-to-4-line Decoder

2-to-4 Line Decoder with Enable Input Using NAND gates

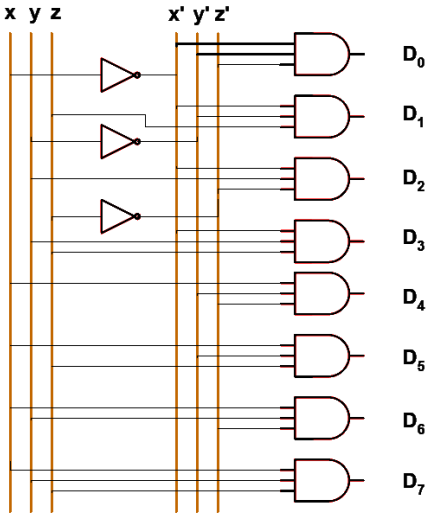
If we use NAND gates to construct the decoder then the outputs are inverted. Decoders are also constructed with one or more enable inputs. An example is shown for the 2-to4 line decoder.

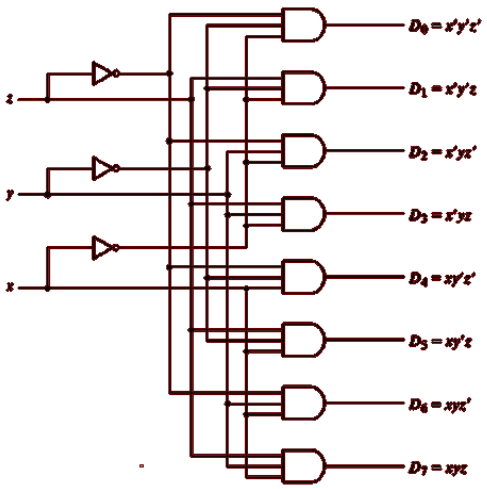


E	A	B	D_0	D_1	D_2	D_3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

3-to-8 decoder circuit (Binary-to-Octal Decoder)

A 3-to-8 line decoder has three elements and eight outputs. The decoder decodes the input binary code represented by the three bits and generates all eight minterms of the inputs. Only one output is one while the other seven are zeros. This decoder can be implemented using three inverters and eight AND gates as shown.

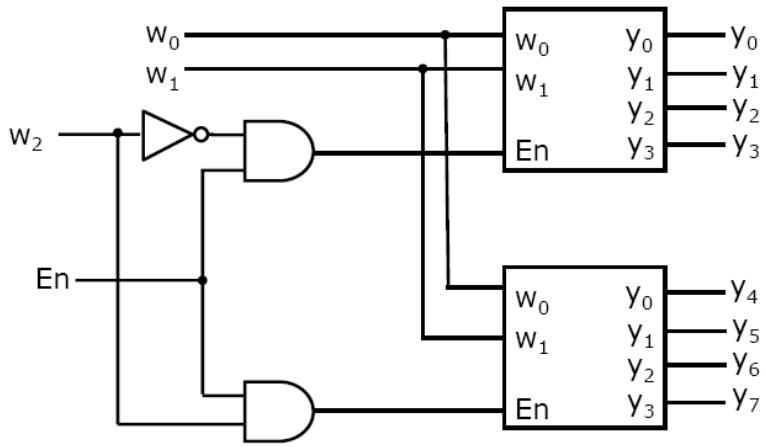




The following truth table is for the 3-to-8 decoder.

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

A 3-to-8 line Decoder can also be created by cascading two 2-to-4-line decoders, which is shown below:



The m-to-n Decoder

We ca create any number of decoder. General uniform decoder uses n inputs and 2_n numbers of outputs. Input represents the binary number ad output is the unique output line flag.

3-to-16 line decoder circuit (Binary-to-Hex Decoder)

This converts binary number into hex value. It can be created by cascading two 3-to-8-line decoders, which is shown below:

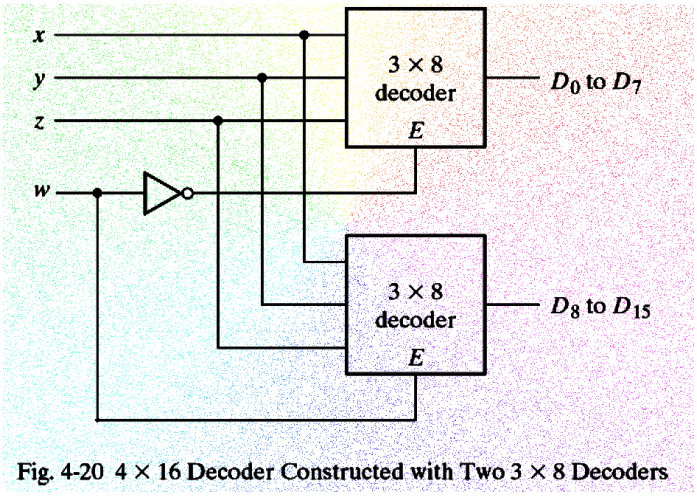
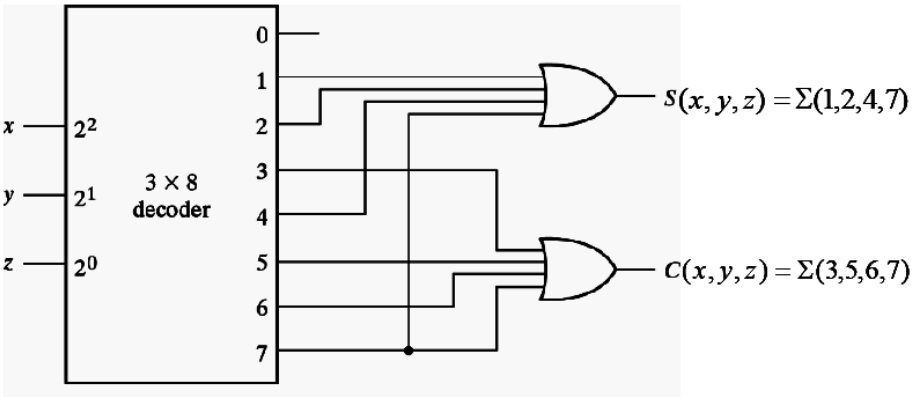


Fig. 4-20 4 × 16 Decoder Constructed with Two 3 × 8 Decoders

When *w* = 0, the top decoder is enabled and the bottom is disabled. Top decoder generates 8 minterms 0000 to 0111, while the bottom decoder outputs are 0's. When *w* = 1, the top decoder is disabled and the bottom is enabled. Bottom decoder generates 8 minterms 1000 to 1111, while the top decoder outputs are 0's.

Full-Adder using Decoder



Implementation of Full adder using Decoder.

BCD-to-seven-segment decoder

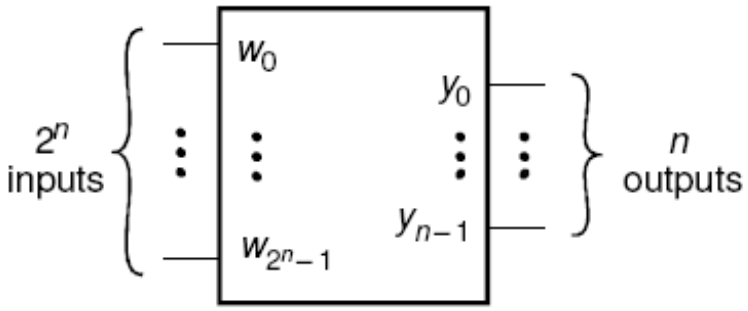
A BCD to Seven Segment Decoder decodes the binary value into seven segment display.

It produces the following display as the input combinations:

BCD inputs				*segment outputs*							*display*
D	C	B	A	a	b	c	d	e	f	g	
0	0	0	0	1	1	1	1	1	1	0	0
0	0	0	1	0	1	1	0	0	0	0	1
0	0	1	0	1	1	0	1	1	0	1	2
0	0	1	1	1	1	1	1	0	0	1	3
0	1	0	0	0	1	1	0	0	1	1	4
0	1	0	1	1	0	1	1	0	1	1	5
0	1	1	0	0	0	1	1	1	1	1	6
0	1	1	1	1	1	1	0	0	0	0	7
1	0	0	0	1	1	1	1	1	1	1	8
1	0	0	1	1	1	1	0	0	1	1	9

Encoder

An encoder is a device that outputs a binary code representing which one of many inputs is active. The outputs are usually active low. The block diagram is shown below:



What is a Flip-Flop?

The storage elements in clocked sequential circuits are called flip-flops. A flip-flop is defined as a circuit that alternates between two possible states when a pulse is received at the input. A flip-flop is a binary storage element capable of storing one bit of information. A flip-flop has two outputs, one for the normal value (Q) and one for the complement value of the bit stored in it (Q'). A flip-flop maintains a binary state until directed by a clock pulse to switch the states. The flip-flop is also called bistable multivibrator, since it has alternative one of the possible two sates. There are several flip-flops which have different number of inputs they possess and different manner in which the inputs affect the binary state.

The most common types of flip-flops are:

(i) SR Flip-Flop (ii) D Flip-Flop (iii) JK Flip-Flop (iv) T Flip-Flop

SR Flip flop

The SR Flip Flop can be divided into two types:

(i) Basic SR Flip-Flop (asynchronous) (ii) Clocked SR Flip-Flop (Synchronous)

Basic SR Flip Flop

The basic flip-flops can be designed by using two NOR gates or two NAND gates as shown in below figure.

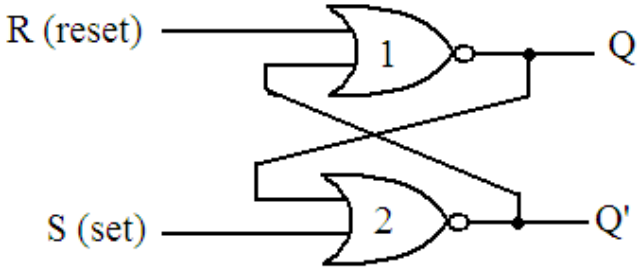


Fig (a) Basic Flip-Flop Circuit with NOR Gates

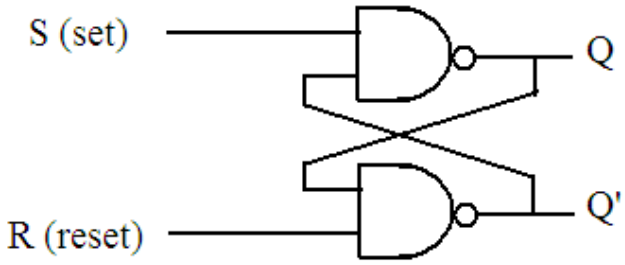


Fig (a) Basic Flip-Flop Circuit with NAND Gates

The cross-coupled connection from the output of one gate to the input of the other gate constitutes a feedback path. Each flip-plop has two inputs S (set) and R (reset) and two outputs Q and Q'. This type of simple flip flop is sometimes called Direct-coupled RS flip-flop or RS latch.

Clocked RS Flip-Flop

The clocked RS flip-flop is the standard RS flip-flop. It is a storage element that has three inputs – S (set), R (reset) and CLK (clock signal). It has a output and a complemented output Q'. The logic diagram, graphic symbol and characteristic table of RS Flip-flop is shown below:

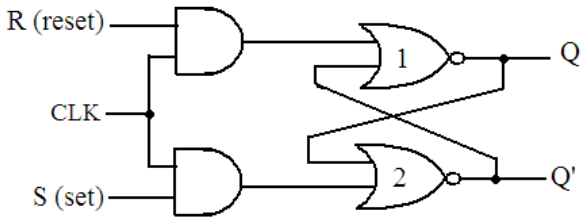


Fig (a) The Logic Diagram of Clocked RS Flip-Flop

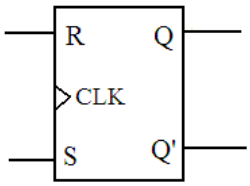


Fig (b) Graphic Symbol

S	R	Q (t+1)	Effect
0	0	Q(t)	No Change
0	1	0	Clear to 0
1	0	1	Set to 1
1	1	?	Indeterminate (Race Condition)

Fig (c) The Characteristic Table

The RS flip-flop shown in above figure consists of a basic NOR flip-flop and two AND gates.

If there is no signal at the clock input CLK, the output of the circuit cannot change irrespective of the values at inputs S & R. Only when the clock signal changes from 0 to 1 can the output be affected according to the values in inputs S and R. If S=1 and R=0 when CLK changes from 0 to 1, output Q is cleared to 0. If both S and R are equal to 1, the output is unpredictable and may go to either 0 or 1, depending on internal

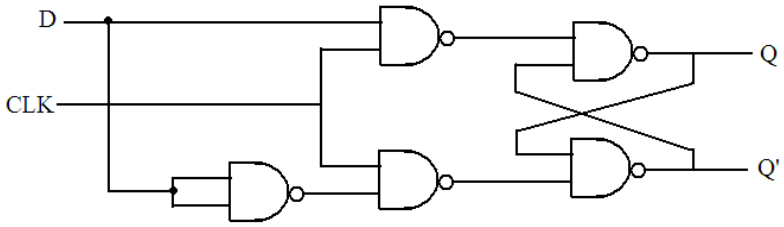
timing delays that occur within the circuit. The characteristic table shown above figure summarizes the operation of the RS flip-flop. The S and R columns give the binary values of the two inputs. Q(t) is the binary state of the Q output at a given time. The Q(t) is referred to as the *present state*. The Q(t+1) is the binary state of the Q output after the occurrence of a clock transition. This state is known as *next state*.

If R=S=0, a clock transition produces no change of state, i.e., Q(t+1) = Q(t). If S=0 and R = 1, the flip-flop produces the 0 (clear) state and if S=1 and R=0, the flip-flop goes to 1 (set) state. The flip-flop should not be pulsed when S=R=1 since it produces an indeterminate next state. This state is called the *Race Condition*.

D Flip-Flop

The 'D' of the D Flip-Flop stands for the *data*. It is the slight modification of the RS flip-flop. An RS flip-flop is converted to a D flip-flop by inserting an inverter between S and R and assigning the symbol D to the

single input. This flip-flop, its graphic symbol and the characteristic table is given below:



Fig(a) Logic Diagram of D Flip-Flop with NAND Gate

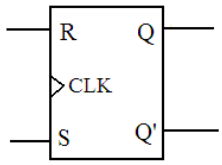


Fig (b) Graphic Symbol

D	Q (t+1)	Effect
0	0	Clear to 0
1	1	Set to 1

Fig (c) The Characteristic Table

The D input is sampled during the occurrence of a clock transition from 0 to 1. If D=1, the output of the flip-flop goes to the 1 state, but if D=0, the output of the flip-flop goes to the 0 state. From the above characteristic table, it is clear that the next state Q(t+1) is determined from the D input. The relationship can be expressed by the characteristic equation: *Q (t+1) = D.*

JK Flip-flop

The JK flip-flop is the advancement over the RS flip-flop that eliminates the race condition occurs in the RS flip-flop. In JK flip-flop, the inputs, J and K behave like the S and R to set and clear the flip-flop. When inputs J and K are both equal to 1, a clock transition

switches the outputs of the flip-flop to their complement state. The logic diagram, graphic Symbol and the characteristic table of JK flip-flop is given below:

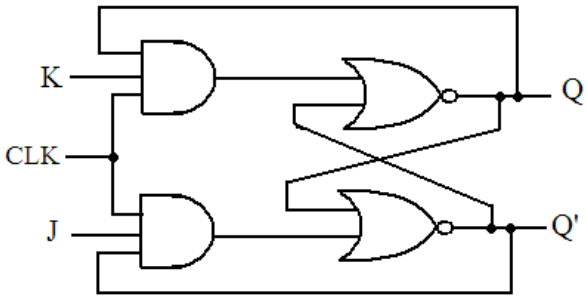


Fig (a) The Logic Diagram of JK Flip-Flop

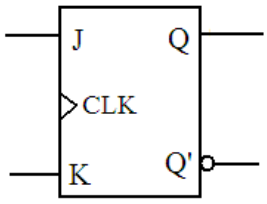


Fig (b) Graphic Symbol

J	K	Q (t+1)	Effect
0	0	Q (t)	No Change
0	1	0	Clear to 0
1	0	1	Set to 1
1	1	Q' (t)	Complement

Fig (c) The Characteristic Table

The J input of the JK flip-flop is equivalent to the S (set) input of the SR flip-flop and K is equivalent to the R of SR flip-flop. Instead of the indeterminate (race) condition, the JK flip-flop has a complement condition Q (t+1) = Q'(t) when both J and K are equal to 1.

T Flip-flop

The 'T' of the T flip-flop stands for the toggle. It is the single input version of the JK flip-flop. It is obtained by the JK type when inputs J and K are connected to

provide a single input designated by T. The logic diagram, graphic symbol and the characteristic symbol for T flip-flop is shown below:

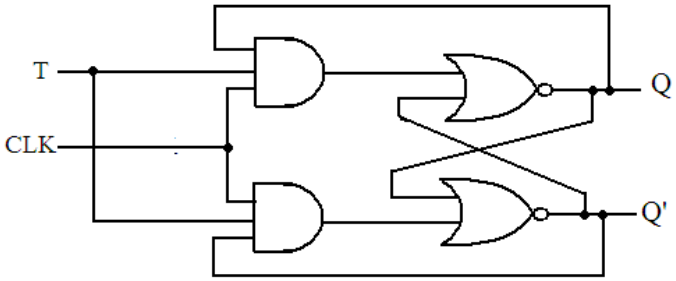


Fig (a) The Logic Diagram of T Flip-Flop

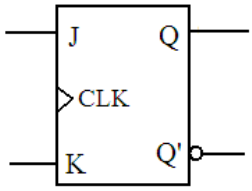


Fig (b) Graphic Symbol

T	Q (t+1)	Effect
0	Q (t)	No Change
1	Q' (t)	Complement

Fig (c) The Characteristic Table

The T flip-flop has only two conditions. When T = 0 [J=K=0], a clock transition does not change the state of the flip-flop. When T = 1 [J = K= 1], a clock transition complements the state of the flip-flop. These conditions can be expressed as :

$$Q(t+1) = Q(t) \oplus T.$$

Edge Trigger Flip-Flop

Edge Triggered flip-flop is the most common type of flip-flop used to synchronize the state change during a clock pulse transition. The term Trigger means the momentary change in the input signal of the flip-flop. The transition caused by a trigger is called to trigger

the flip-flop. Clocked flip-flops are triggered by a clock pulse. A clock pulse starts from an initial value 0, then after a fixed period, it changes its value to 1, and after this declared time span, it returns to its original value 0, and then again to 1, and so. The time of input signals and the output state time may have a time delay. In this case the operation on a flip-flop cannot be performed. To prevent such the problem, the flip-flop must be joined with a clock pulse, in which the signal 0 gives the waiting (no change) for the flip-flop. To ensure the delay time, a flip-flop must have a signal propagation delay from input to output in excess of the clock pulse. To find the delay time is a difficult way for a flip-flop. So, a better way to solve the feedback timing problem is by implementing the pulse transition rather than the pulse duration.

A clock pulse may have two values (like other values), either 0 or 1. The 1 value here is the presence of clock signal and the 0 means the absence. A clock pulse goes through two signal transitions- from 0 to 1 and from 1 to 0. The clock transition from 0 to 1 is

called the positive transition of clock pulse and the edge is called the positive edge. The clock transition from 1 to 0 is called the negative transition and the edge is called the negative edge of the clock pulse. This is shown in the following figure:

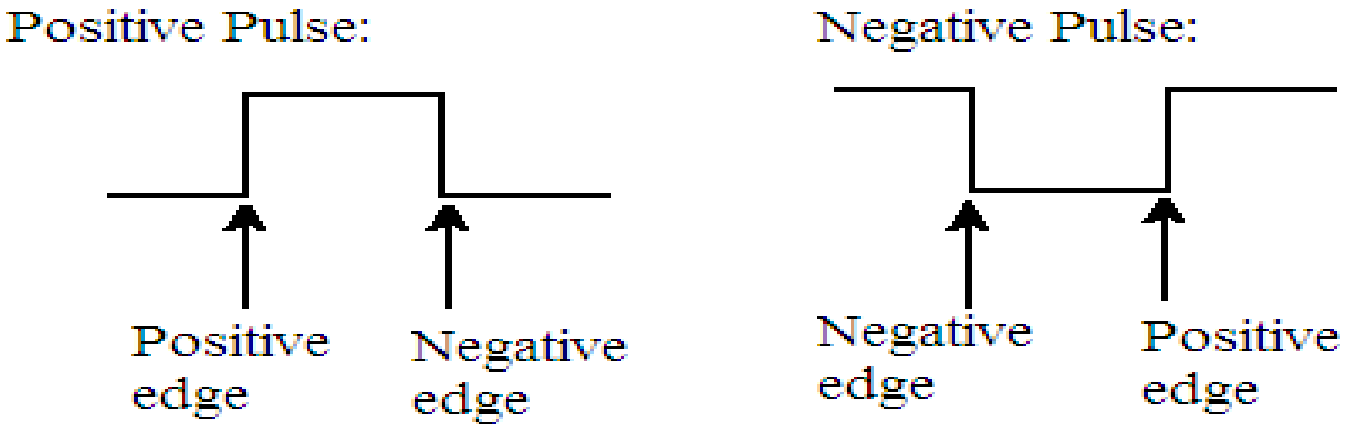
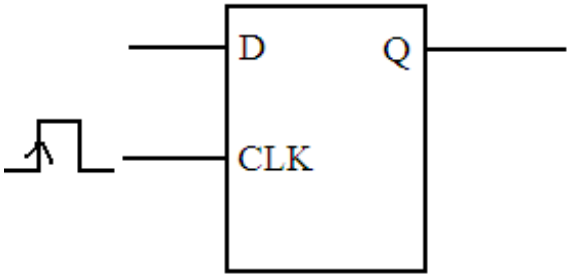
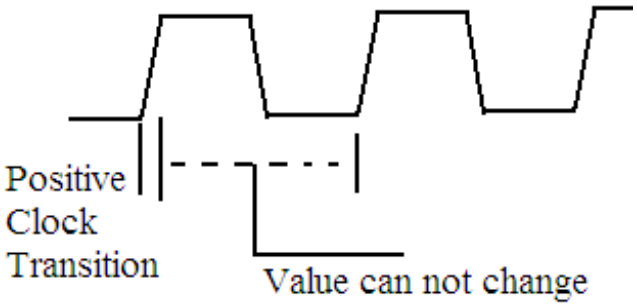


Fig.: Positive and Negative edges of a clock pulse

In the flip-flop symbol, we generally use the first figure for the positive edge triggered flip-flops and the second figure for the negative edge triggered flip-flops. Some edge-triggered flip-flops cause a transition on the rising edge of the clock signal, and others cause a falling edge of the clock transition.

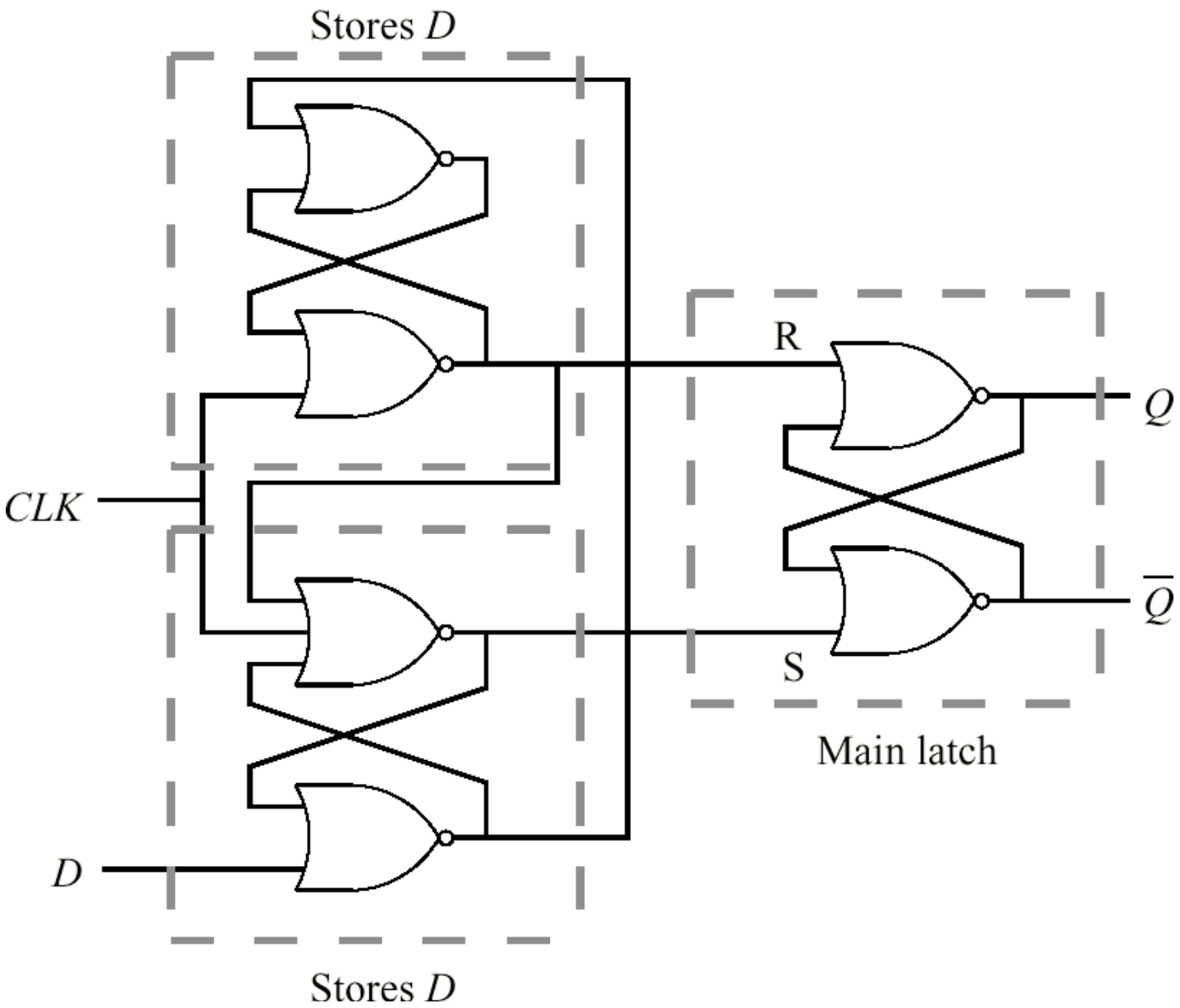
The block diagram of a positive triggered D flip-flop is shown below.





The minimum time to set up the edge transition effectively is called the *setup time.* The time in which the input holds to wait for a succession of clock transition is called a *hold time.*

The logic diagram a a negative edge triggered flip-flop is shown below:



It has the following effects:

- When the clock is high, the two input latches output 0, so the Main latch remains in its previous state, regardless of changes in D.
- When the clock goes high-to-low, values in the two input latches will affect the state of the Main latch.

- While the clock is low, D cannot affect the Main latch.

Master Slave Flip-Flop

A master-slave flip-flop is constructed by using two similar type (sometimes different) types of flip-flops, in which one flip-flop serves as the Master and another one Slave. An example of D master slave flip-flop is shown below. It consists a master D flip-flop, a slave D flip-flop and an inverter on clock signal before the slave flip-flop. When the clock signal CLK is 0, the output of the inverter is 1. Since the clock input of the slave is 1, the flip-flop is enabled and the output Q is equal to Y. At this state, the master flip-flop is disabled, since the clock pulse has the value 0. When the pulse becomes 1, the information at the external D is transmitted to the master flip-flop. So, when the clock pulse has the value 0, the slave is active and master is isolated. Similarly, if the clock pulse goes to 1, the master will be activated and the slave becomes isolated until the clock value goes to 0.

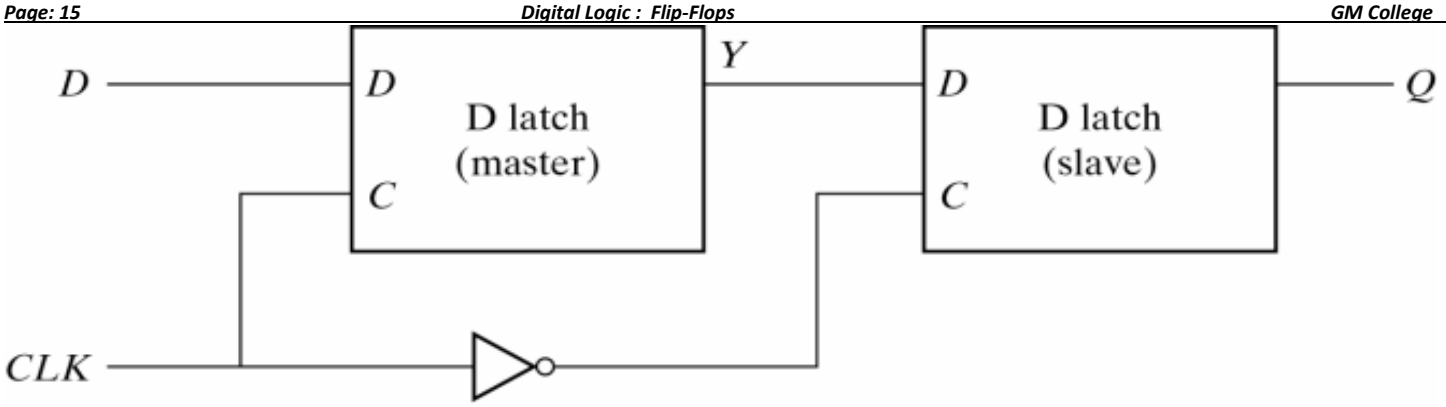


Fig. 5-9 Master-Slave *D* Flip-Flop

The circuit samples the D input and changes its output at the negative edge of the clock, CLK. When the clock is 0, the output of the inverter is 1. The slave latch is enabled and its output Q is equal to the master output Y. The master latch is disabled (CLK = 0). When the CLK changes to high, D input is transferred to the master latch. The slave remains disabled as long as C is low. Any change in the input changes Y, but not Q. The output of the flip-flop can change when CLK makes a transition 1→ 0.

The master slave combination of flip-flop can be constructed from any type of flip-flop by adding a clocked RS flip-flop with an inverted clock signal to form the slave. An example is shown below, which is the clocked master-slave JK flip-flop.

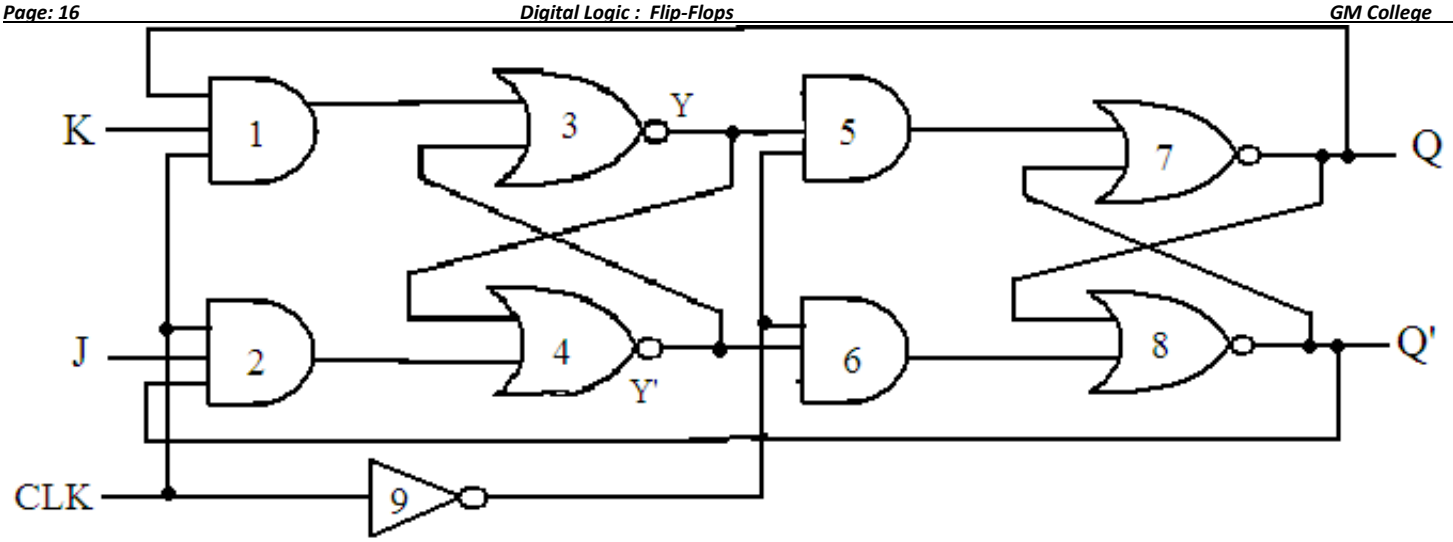
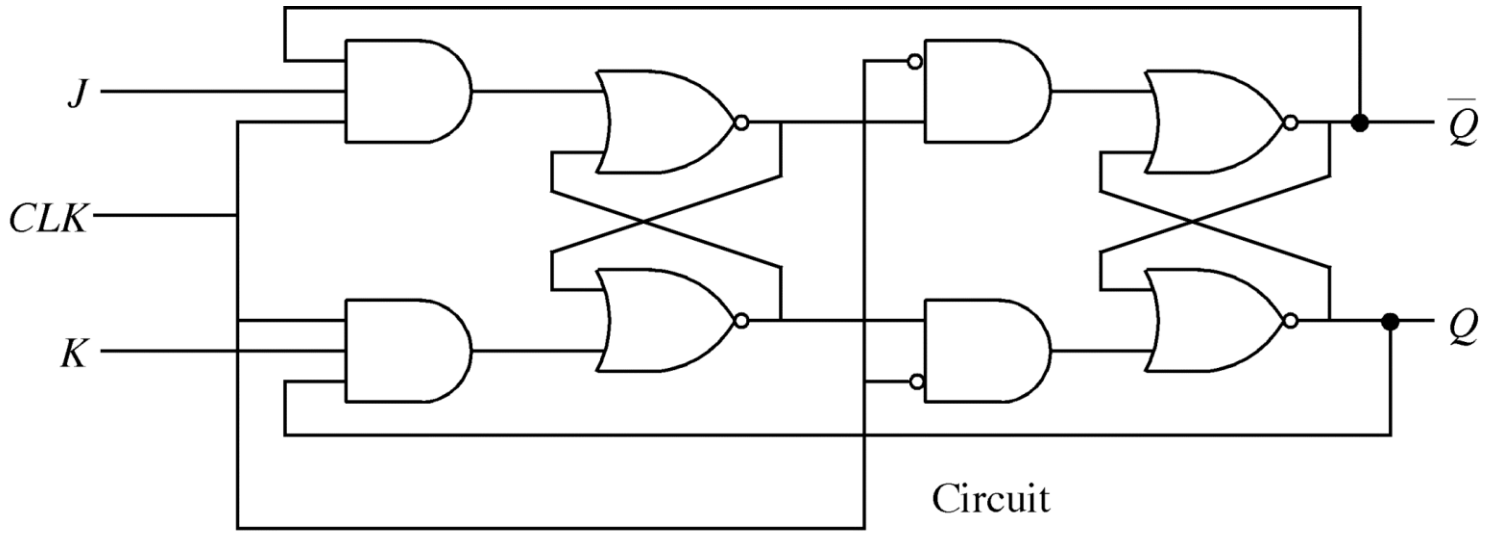


Fig: Clocked master-slave JK Flip-Flop

It consists of two flip-flops. Gates 1 through 4 forms the master flip-flop and gates 5 through 8 forms the slave flip-flop. The information present at the J and K input is transmitted to the master flip-flop on the positive edge of the clock pulse and is held there until the negative edge of the clock pulse occurs. Then this allows passing through to the slave flip-flop. The clock input now is normally 0 that keeps the outputs of gates 1 and 2 at the 1 level. This prevents the J and K inputs from affecting the master flip-flop. The slave flip-flop is a clocked RS flip-flop with the master flip-flop supplying the inputs and the clock input being inverted by gate 9. When the clock signal is 1, the output of gate 9 is 1, so that output Q is

equal to Y and Q' is equal to 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 level 1 and when the clock input returns to 0, the master flip-flop is isolated from J and K inputs and the slave flip-flop goes to the same as the master flip-flop.

In general, the rising edge of the clock loads new data into the master, while the slave continues to hold previous data. The falling edge of the clock loads the new master data into the slave.
The master-slave JK flip-flop is shown in the following diagram.



Unit VI: Sequential Circuits: State Table, State Diagram, Simple Sequential Circuits

---

What is a sequential circuit?

A sequential circuit is an interconnection of flip-flops and logic gates. The combinations of gates create a combinational logic circuit, but when they are interconnected with the flip-flop, the overall circuit is called the sequential circuit. The sequential circuit creates the memory elements that are used to hold the information by using the flip-flops. The binary information stored in the memory elements at any given time defines the state of the sequential circuit at that time. The sequential circuit receives binary information from the external inputs. These inputs together with the present state of the storage elements, determine the binary value of the outputs. A block diagram of a sequential circuit is given below. It comprises of a combinational circuit to which memory elements are connected to form a feedback path. It consists of a combinational circuit with a number of clocked flip-flops. In general, in sequential circuit, any number or type of flip-flops may be included.

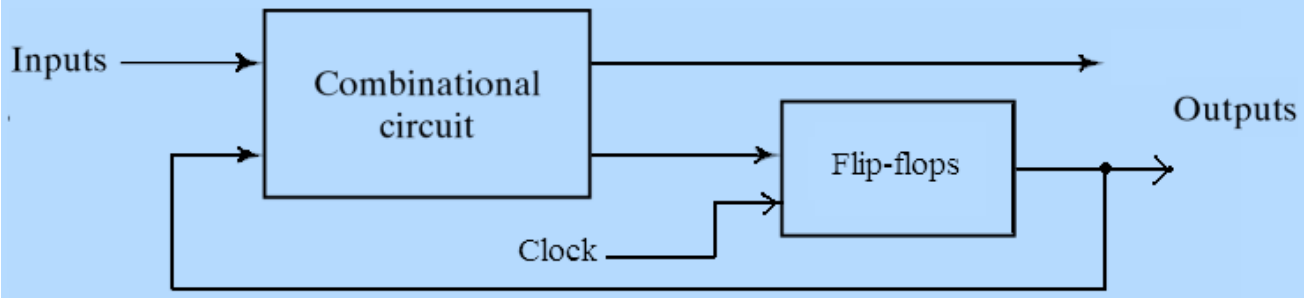


Fig. 5-1 Block Diagram of Sequential Circuit

In the diagram, the combinational circuit block receives binary signals from external inputs and from the outputs (states) of flip-flops. The outputs of the combinational circuit go to external outputs and to inputs of flip-flops. The binary value to be stored in the flip-flops after each clock transition is determined by the the gates inside the combinational circuit. The outputs of the flip-flops are applied to the combinational circuit input and determine the behavior of the circuit. In this way, the external outputs of a sequential circuit are functions of both external inputs and the present state of the flip-flops. Likewise, the next state of the flip-flops is also a function of their present state and external inputs. So, a sequential circuit is specified by a time sequence of external inputs, external outputs and internal flip-flop binary state.

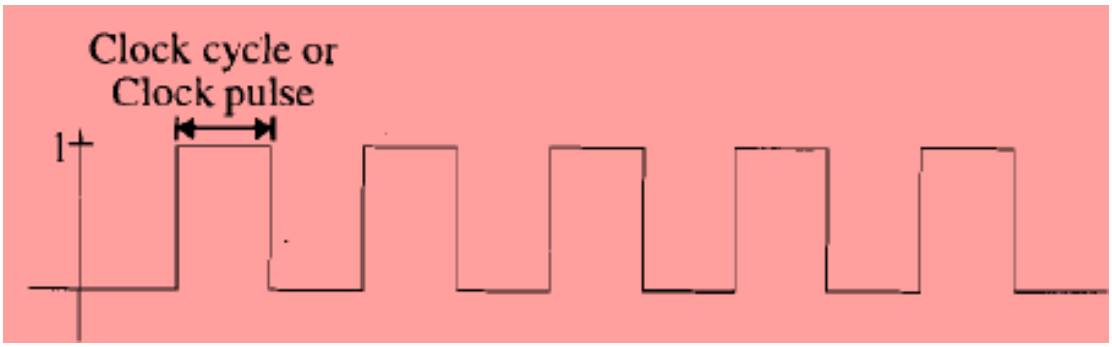
These are logic circuits whose present output depends on the past inputs. These circuits store and remember information. The sequential circuits unlike combinational circuits are time dependent. Normally the current output of a sequential circuit depends on the state of the circuit and on the current input to the circuit. It is a connection of flip-flops and gates

Types of a sequential Circuit

The sequential circuit can be categorized into two types. The classification depends up on the timing of their signals.

Asynchronous Sequential Circuits: The asynchronous sequential circuit is directly triggered by the input signals and can be affected at any instance of time. The Asynchronous sequential circuits may be regarded as combinational circuits with feedback path.

Synchronous Sequential Circuits: A synchronous sequential circuit employs signal that affect the storage elements only at discrete instants of time. Synchronization is achieved by a time device called clock that provides a periodic train of pulses. Synchronous circuits use flip-flops and their status can change only at discrete instants.



The synchronization in sequential circuits can be achieved using a clock pulse generator. It synchronizes the effect of input over output. It presents signal of the following form:

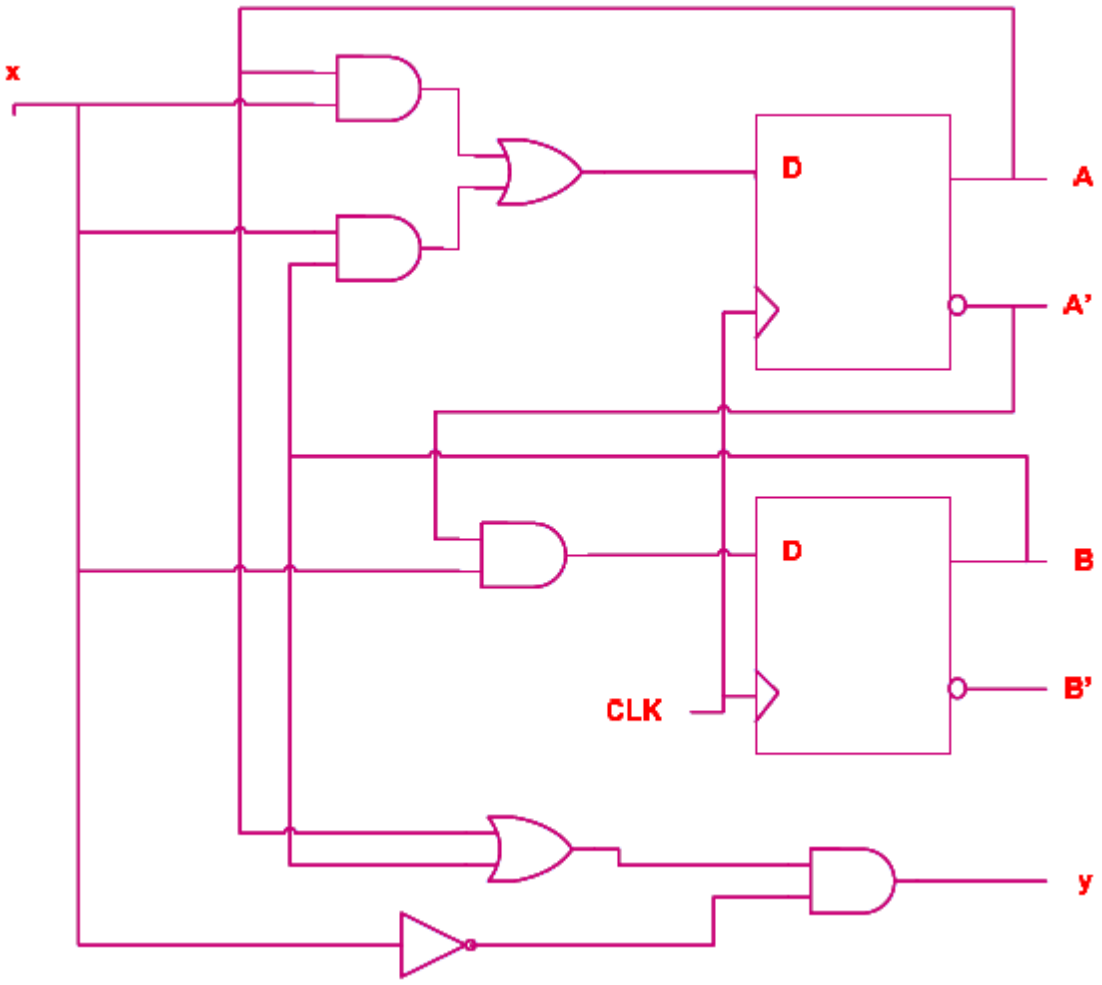
The signal produced by clock pulse generator is in the form of clock pulse or clock signal. These clock pulses are distributed throughout the computer system for synchronization. A clock

can have two states: an enable or active state, otherwise a disable or inactive state. Both of these states can be related to zero or one levels of clock signals (it depends on implementation). Normally, the flip-flops change their state only at the active state of the clock pulse. In certain designs the active state of the clock is triggered when the transition (this is sometimes termed as edge-triggered transition) from 0 to I or 1 to 0 is taking place in clock signal. A typical CPU is synchronized by a clock signal whose frequency is used as a basic measure of the CPU's speed of operation

State Table:

A State table specifies a sequential circuit that relates outputs and next states as a function of inputs and the present state of the sequential circuit. The behavior of the sequential circuit is determined by the input signals, the output of the circuit and the present states of the flip-flops. Both the outputs of the circuit and the next state of the flip-flops are a function of the input signals and the present sate of the flip-flops. The state table shows all the possible combinations of the inputs and outputs of the sequential circuit and present and next state of the flip-flops used in the circuit.

Let us take an sequential circuit.



The expression for the above circuit is:

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

Now, the state table will be.

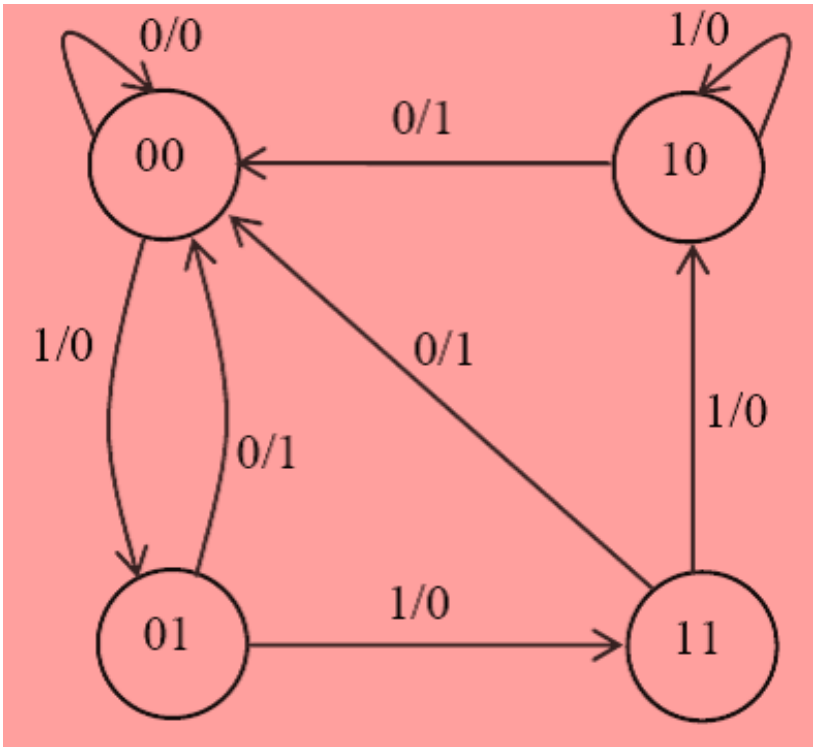
Present State		Input	Next State		Output
A	B	x	A	B	y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0

The state table can also be put in the following different form.

P.S.		N.S.				Output	
		x=0		x=1		x=0	x=1
A	B	A	B	A	B	y	y
0	0	0	0	0	1	0	0
0	1	0	0	1	1	1	0
1	0	0	0	1	0	1	0
1	1	0	0	1	0	1	0

State Diagram

The information in the state table can be presented pictorially in the form of a state diagram. A state is represented by a circle. A transition from present state to next state is represented by an arrow with the inputs and outputs written on that transition with a slash in between inputs and outputs. The state diagram of the previous sequential circuit is given below.


0/0
1/0
00
10
0/1
1/0
0/1
1/0
0/1
01
1/0
11

1. Introduction

A counter is a sequential machine that produces a specified count sequence. The count changes whenever the input clock is asserted.

There is a great variety of counter based on its construction.

1. Clock: Synchronous or Asynchronous
2. Clock Trigger: Positive edged or Negative edged
3. Counts: Binary, Decade
4. Count Direction: Up, Down, or Up/Down
5. Flip-flops: JK or T or D

- A counter can be constructed by a synchronous circuit or by an asynchronous circuit. With a synchronous circuit, all the bits in the count change synchronously with the assertion of the clock. With an asynchronous circuit, all the bits in the count do not all change at the same time.
- A counter may count up or count down or count up and down depending on the input control.

- Because of limited word length, the count sequence is limited. For an n-bit counter, the range of the count is [0, 2^n-1]. The count sequence usually repeats itself. When counting up, the count sequence goes in this manner: 0, 1, 2, … 2^n-2, 2^n-1, 0, 1, …etc. When counting down the count sequence goes in the same manner: 2^n-1, 2^n-2, … 2, 1, 0, 2^n- 1, 2^n-2, … etc.

  Example:

3-bit Up Counter	3-bit Down Counter
000	000
001	111
010	110
011	101
100	100
101	011
110	010
111	001

- The complement of the count sequence counts in reverse direction. If the uncomplemented output counts up, the complemented output counts down. If the uncomplemented output counts down, the complemented output counts up.

Example:

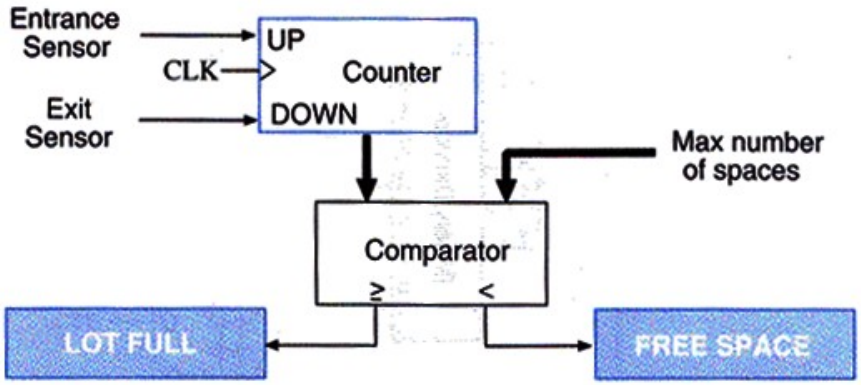
3-bit Up Counter	Complement of the Count
000	111
001	110
010	101
011	100
100	011
101	010
110	001
111	000

- The natural count sequence is to run through all possible combinations of the bit patterns before repeating itself. External logic can be used to arbitrary cause the counter to start at any count and terminate at any count.
- A binary counter produces a count sequence similar to the binary numbers. A decade counter counts from 0 to 9, thus making it suitable for human interface. Other counters count to 12 making them suitable for clocks.

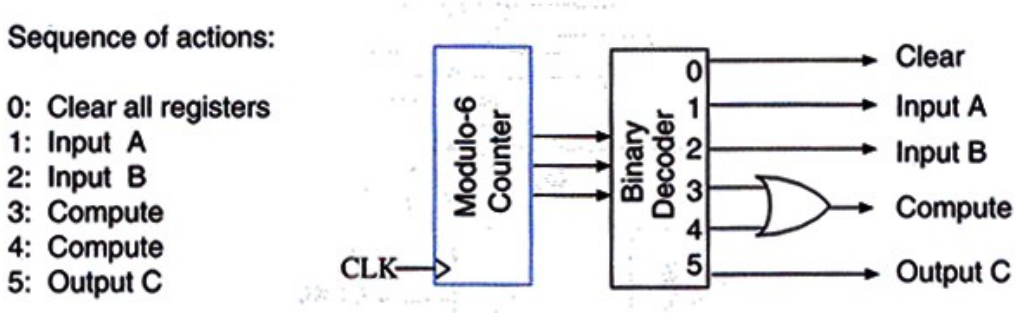
1.1 Uses of Counters

The most typical uses of counters are:

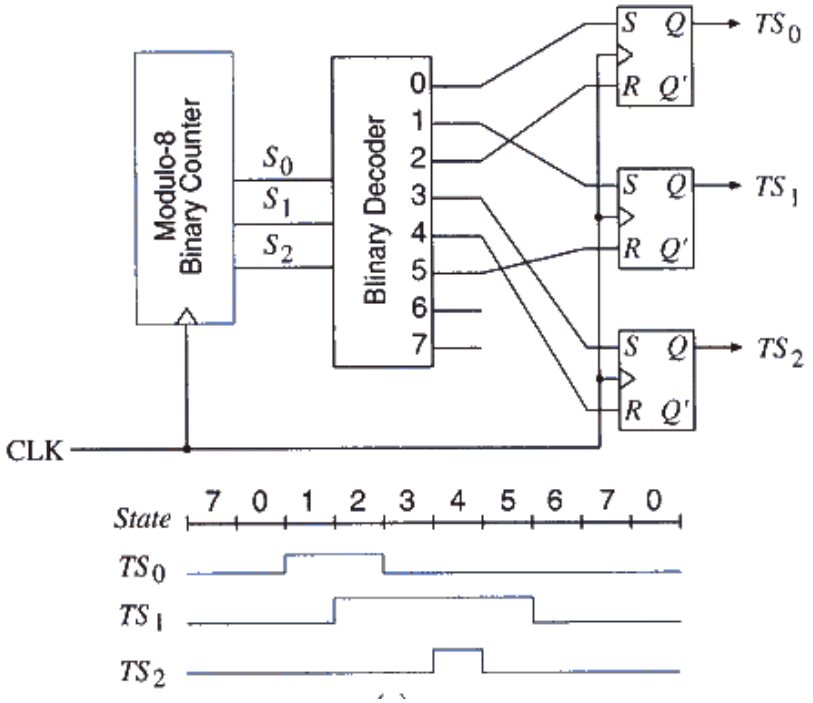
- To count the number of times that a certain event takes place; the occurrence of event to be counted is represented by the input signal to the counter



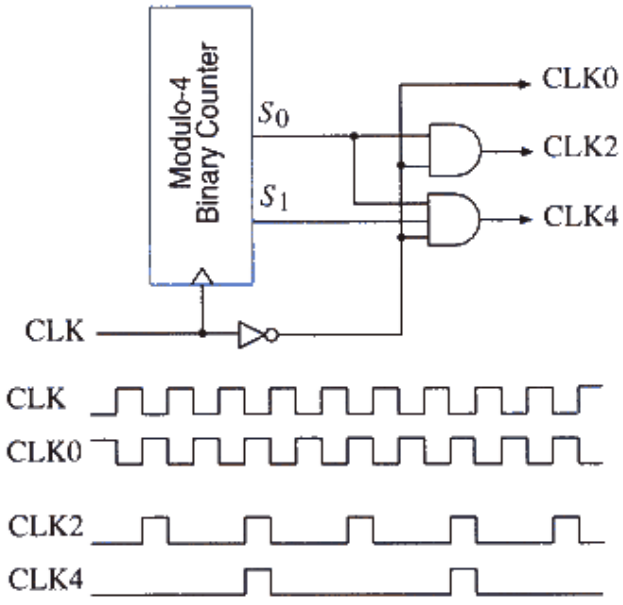
- To control a fixed sequence of actions in a digital system



- To generate timing signals



- To generate clocks of different frequencies



1.2 Two Classes of Counters

Counters are classified into two categories:

- Asynchronous Counters (Ripple counters)

- Synchronous Counters

Asynchronous & Synchronous

Asynchronous: The events do not have a fixed time relationship with each other and do not occur at the same time.
Synchronous: The events have a fixed time relationship with each other and do occur at the same time.

Counters are classified according to the way they are clocked. In asynchronous counters, the first flip-flop is clocked by the external clock pulse and then each successive flip-flop is by clocked the output of the preceding flip-flop. In synchronous counters, the clock input is connected to all of the flip-flop so that they are clocked simultaneously. Asynchronous (ripple) counters are counters where each flip flop is triggered by the transition of other flip-flops. In synchronous counters all flip-lops are triggered by the same clock pulses.

An asynchronous counter is one in which the flip-flop within the counter do not change states at exactly the same time because they do not have a common clock pulse.

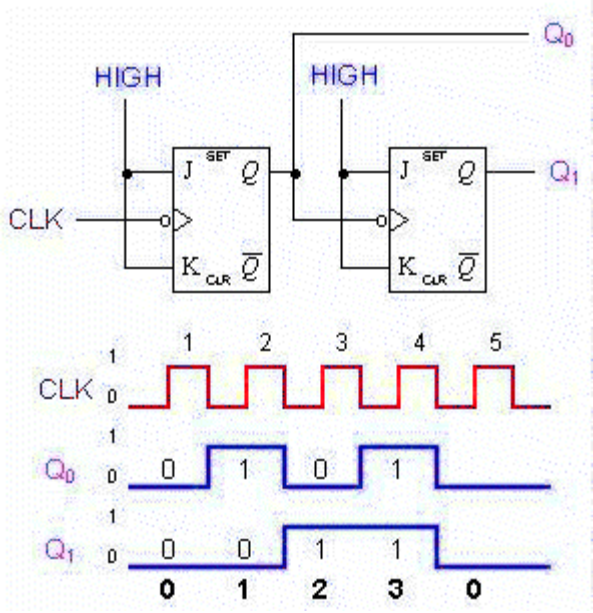
- 2 Bit asynchronous binary counter
- 3 Bit asynchronous binary counter
- 4 Bit asynchronous binary counter

The main characteristic of an asynchronous counter is each flip-flop derives its own clock from other flip-flops and is therefore

independent of the input clock. Consequently, the output of each flip-flop may change at different time, hence the term asynchronous. From the asynchronous counter diagram above, we observed that the output of the first flip-flop becomes the clock input for the second flip-flop, and the output of the second flip-flop becomes the clock input for the third flip-flop etc. For the first flip-flop, the output changes whenever there is a negative transition in the clock input. This means that the output of the first flipflop produces a series of square waves that is half the frequency of the clock input. Since the output of the first flip-flop becomes the clock of the second flip-flop, the output of the second flip-flop is half the frequency of its clock, i.e. the output of the first flip-flop that in turn is half the frequency of the clock input. This behaviour, in essence is captured by the binary bit pattern in the counting sequence.

2.1 2-Bit Asynchronous Binary Counter

The 2-bit counter is also called modulo-4 counter because it counts 4 numbers. It has two flip-flops.The logic diagram and the timing diagram of a 2-bit asynchronous counter is shown below:



- From the above figure, it is clear that, The external clock is connected to the clock input of the first flip-flop (FF0) only. So, FF0 changes state at the falling edge of each clock pulse, but FF1 changes only when triggered by the falling edge of the Q output of FF0. Because of the inherent propagation delay through a flip-flop, the transition of the input clock pulse and a transition of the Q output of FF0 can never occur at exactly the same time. Therefore, the flip-flops cannot be triggered simultaneously, producing an asynchronous operation.
- Note that for simplicity, the transitions of Q0, Q1 and CLK in the timing diagram above are shown as simultaneous even though this is an asynchronous counter. Actually, there is some small delay between the CLK, Q0 and Q1 transitions.
- Usually, all the CLEAR inputs are connected together, so that a single pulse can clear all the flip-flops before counting starts. The clock pulse fed into FF0 is rippled

through the other counters after propagation delays, like a ripple on water, hence the name Ripple Counter.

- The 2-bit ripple counter circuit above has four different states, each one corresponding to a count value. Similarly, a counter with n flipflops can have 2 to the power n states. The number of states in a counter is known as its mod (modulo) number. Thus a 2-bit counter is a mod-4 counter.
- A mod-n counter may also be described as a divide-by-n counter. This is because the most significant flip-flop (the furthest flip-flop from the original clock pulse) produces one pulse for every n pulses at the clock input of the least significant flip-flop (the one triggers by the clock pulse). Thus, the above counter is an example of a divide-by-4 counter.

Another Example:

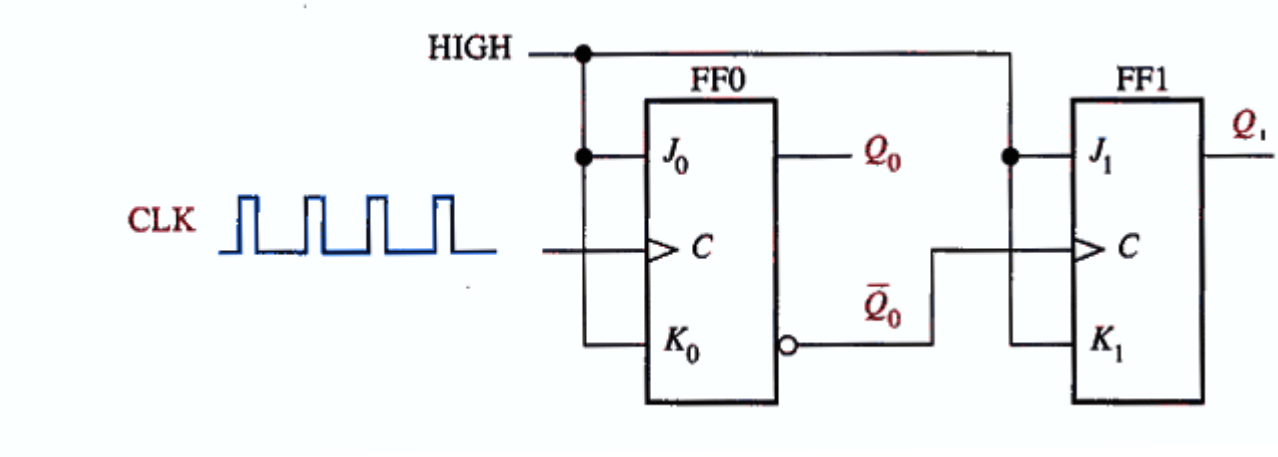


Fig. Two-bit asynchronous binary counter

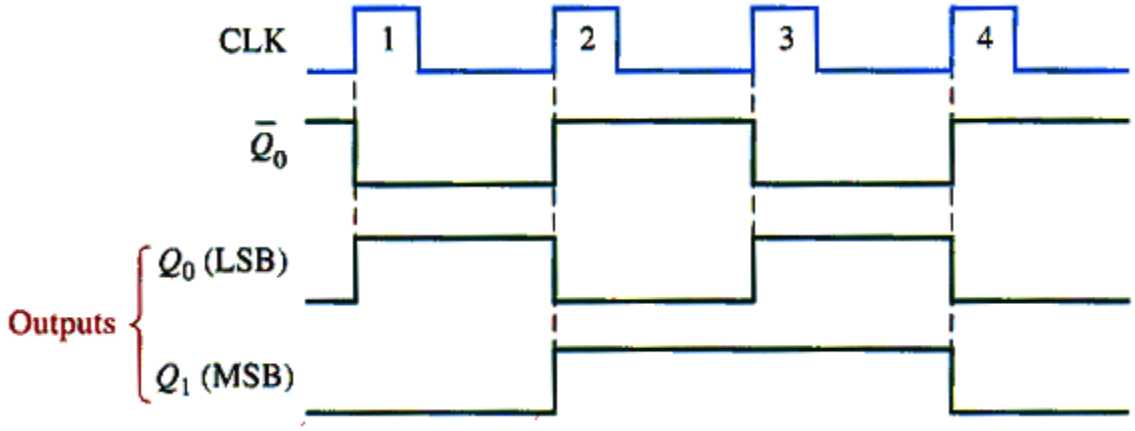


Fig. timing diagram

Fig. binary state sequence

2.2 3-Bit Asynchronous Binary Counter

The following is a three-bit asynchronous binary counter and its timing diagram for one cycle. It works exactly the same way as a two-bit asynchronous binary counter mentioned above, except it has eight states due to the third flip-flop.

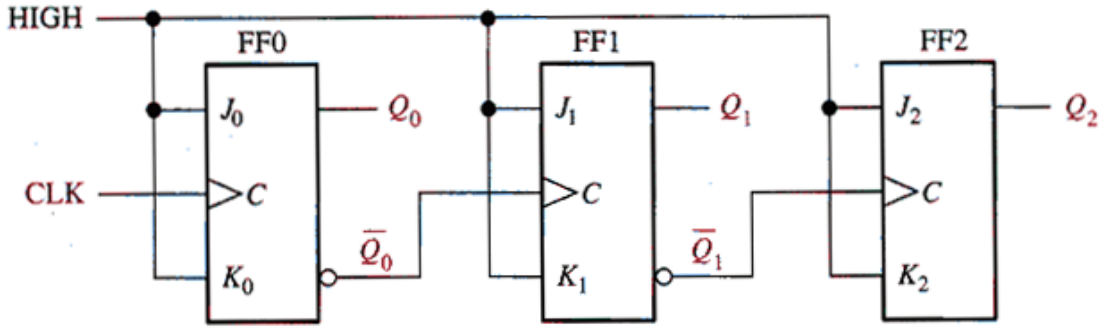


Fig. (a) Three-bit asynchronous binary counter

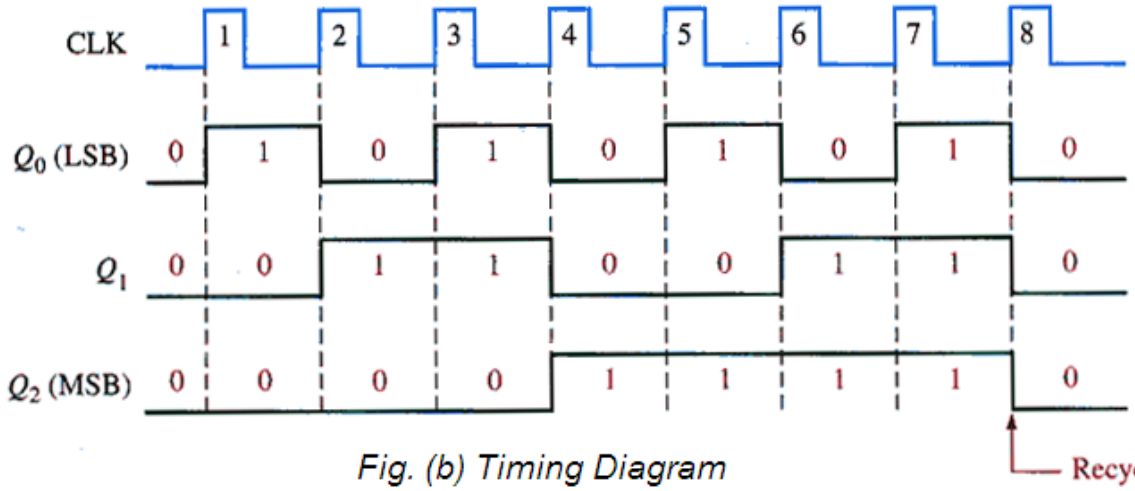


Fig. (b) Timing Diagram

Clock Pulse	Q_2	Q_1	Q_0
Initially	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1
8 (recycles)	0	0	0

Fig. (c) binary state sequence

2.3 4 Bit Asynchronous Binary Counter

The following is a 4-bit asynchronous binary counter and its timing diagram for one cycle. It works exactly the same way as a 2-bit or 3 bit asynchronous binary counter mentioned above, except it has 16 states due to the fourth flip-flop.

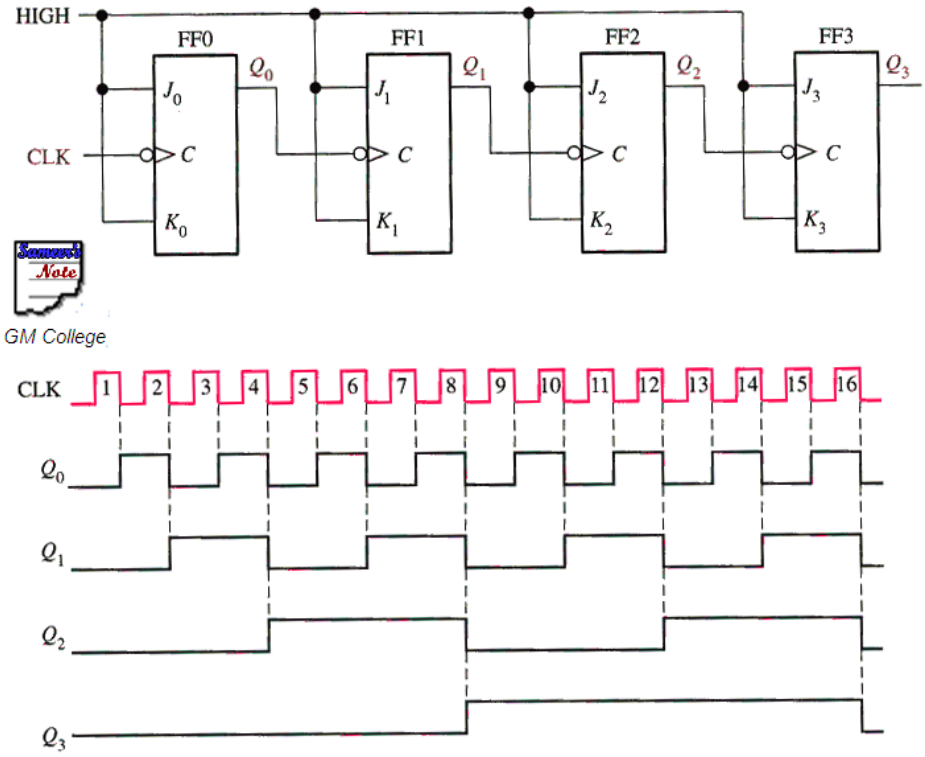
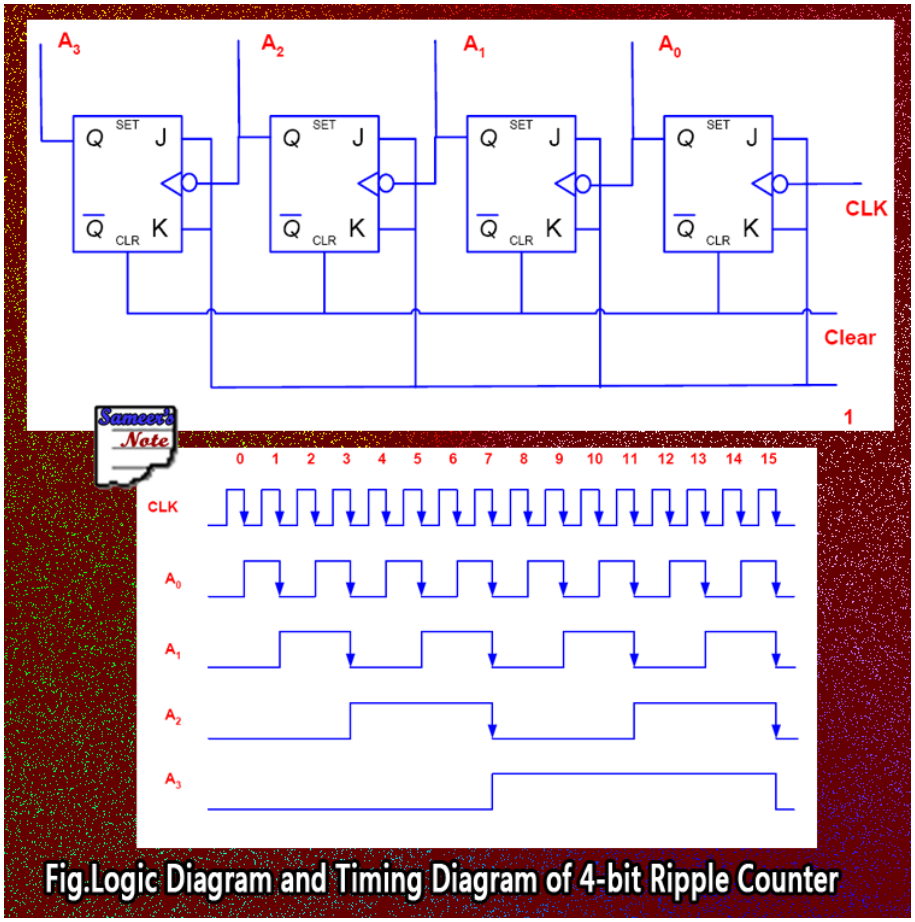
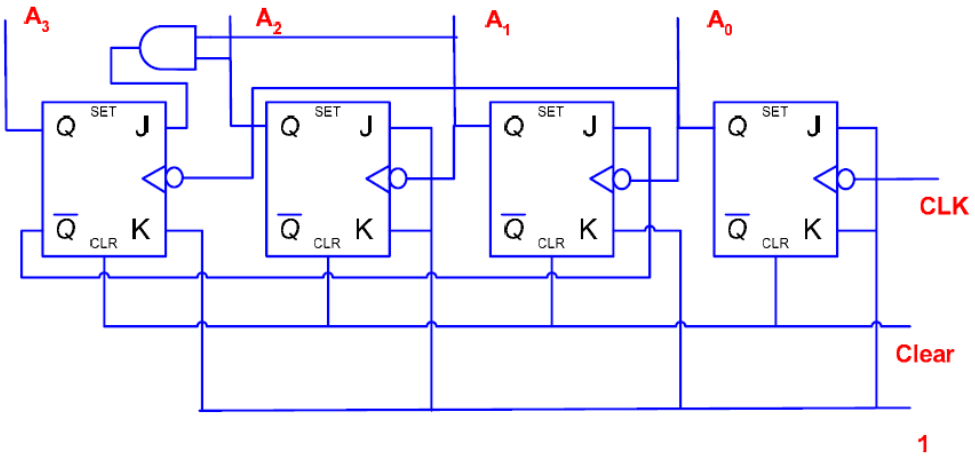


Fig. Four-bit asynchronous binary counter, timing diagram

Binary ripple counter consists of a series of complementing flip-flops. A binary counter consisting of n flip-flops has a count cycle of 2n and counts from 0 to 2n-1. A 4-bit binary ripple counter using T flip-flops is shown.



2.4 Asynchronous Decade Counters (BCD Ripple Counter)



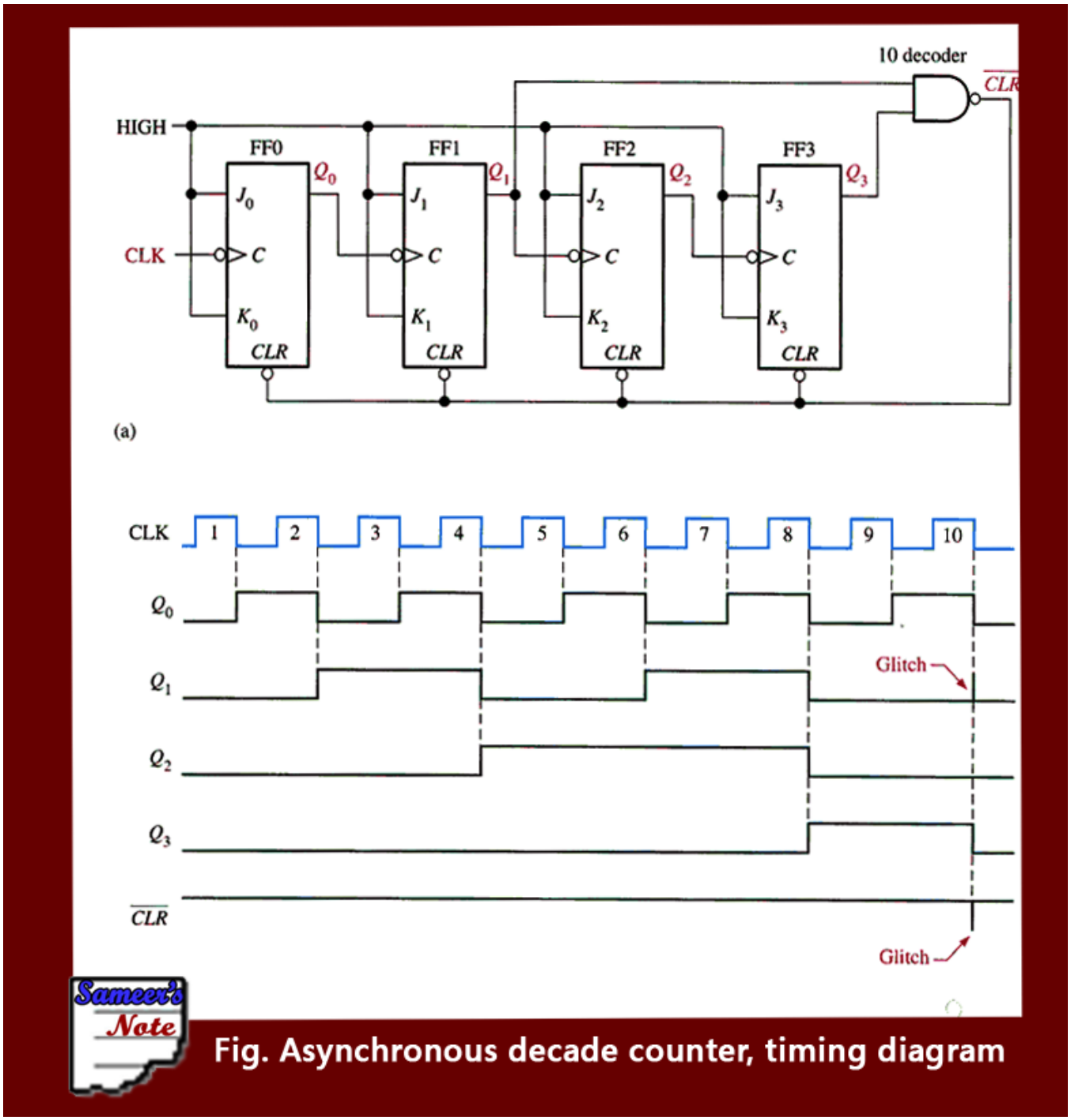
- The binary counters previously introduced have two to the power n states. But counters with states less than this number are also possible. They are designed to have the number of states in their sequences, which are called truncated sequences. These sequences are achieved by forcing the counter to recycle before going through all of its normal states.
- A common modulus for counters with truncated sequences is ten. A counter with ten states in its sequence is called a decade counter.

In order to arrive at the design of the BCD ripple counter, we must answer two questions concerning each flip-flop:

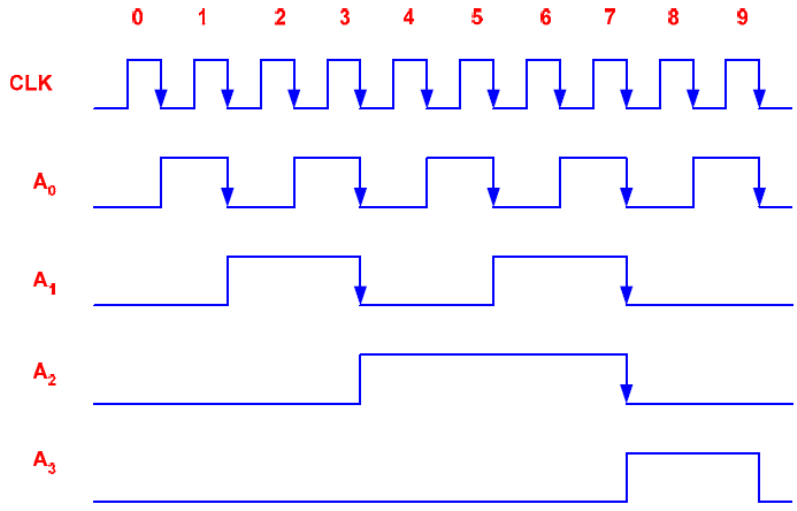
1. Which source should trigger the flip-flop?
2. What values should we make the J and K inputs of the flip-flop?

Using the timing diagram, it is clear that A0 is to be triggered by the input clock pulses and the J and K inputs should be made 1 and 1. A1 should be triggered by A0. In order to prevent it from setting on the negative edge of pulse 9, we should make the J and K inputs equal to A_3' and 1. A_2 is triggered by A_1 and the J and K inputs are 1 and 1. Finally, A_3 should be triggered by A_0. TO prevent A_3 from setting until we reach pulse 7, then we should make the J and K inputs equal to A_2A_1 and 1. The logic circuit of the BCD ripple counter is shown next.

- The circuit below is an implementation of a decade counter.



- Once the counter counts to ten (1010), all the flip-flops are being cleared. Notice that only Q1 and Q3 are used to decode the count of ten. This is called partial decoding, as none of the other states (zero to nine) have both Q1 and Q3 HIGH at the same time.



- The sequence of the decade counter is shown in the table below:

Clock Pulse	Q3	Q2	Q1	Q0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1

Sameer's Note

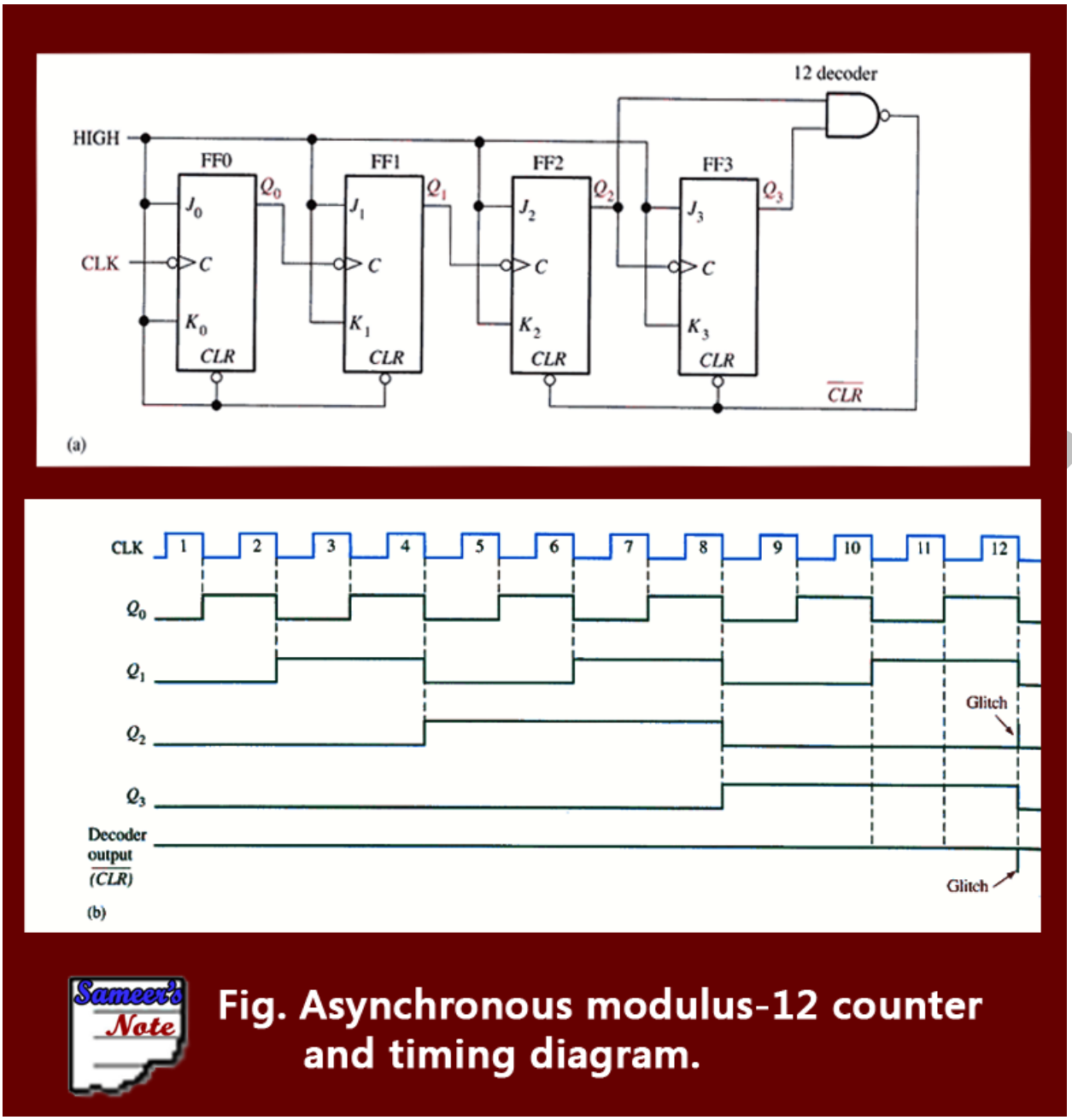
Fig. The sequence table of the decade counter

Glitch: Notice that there is a glitch on the Q1 waveform. The reason for this glitch is that Q1 must first go HIGH before the count of ten can be decoded. Not until several nanoseconds

after the counter goes to the count of ten does the output of the decoding gate go LOW (both inputs are HIGH). Therefore, the counter is in the 1010 state for a short time before it is reset to 0000, thus producing the glitch on Q_1 and the resulting glitch on the *CLR* line that resets the counter.

Modulus Twelve Asynchronous Counter

An Asynchronous counter can be implemented having a modulus of 12 with a straight binary sequence from 0000 through 1011.



Sameer's Note

Fig. Asynchronous modulus-12 counter and timing diagram.

2.5 Asynchronous Up-Down Counters

In certain applications a counter must be able to count both up and down. The circuit below is a 3-bit up-down counter. It counts up or down depending on the status of the control signals UP and DOWN. When the UP input is at 1 and the DOWN input is at 0, the NAND network between FF0 and FF1 will gate the non-inverted output (Q) of FF0 into the clock input of FF1.

Similarly, Q of FF1 will be gated through the other NAND network into the clock input of FF2. Thus the counter will count up.

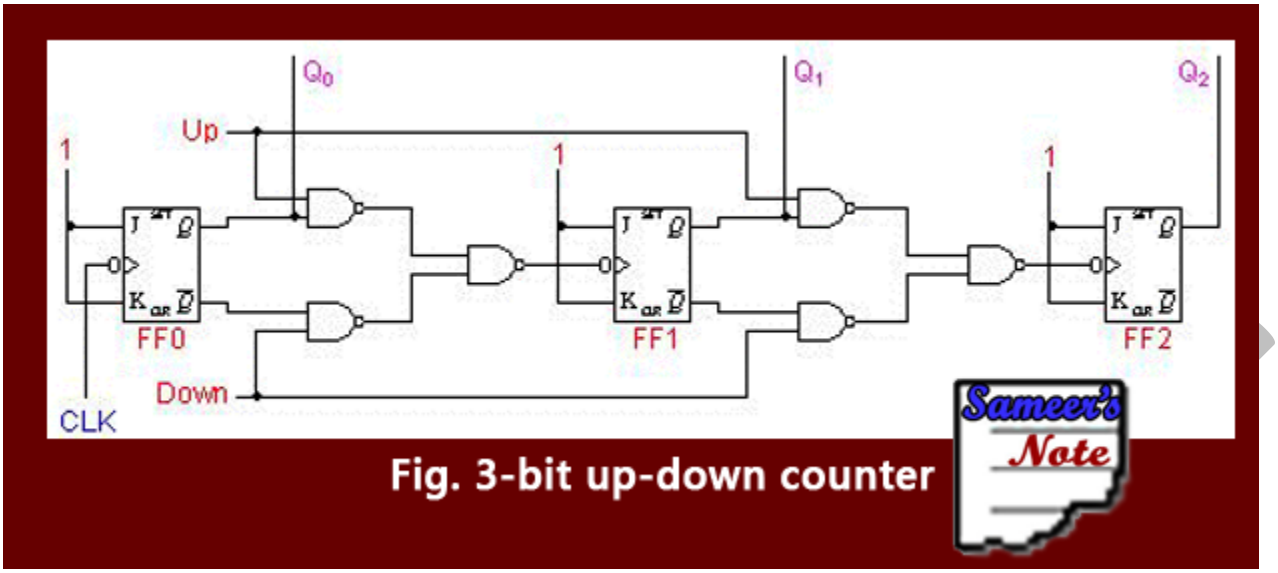


Fig. 3-bit up-down counter

When the control input UP is at 0 and DOWN is at 1, the inverted outputs of FF0 and FF1 are gated into the clock inputs of FF1 and FF2 respectively. If the flip-flops are initially reset to 0's, then the counter will go through the following sequence as input pulses are applied.

FF2	FF1	FF0
0	0	0
1	1	1
1	1	0
1	0	1
1	0	0
0	1	1
0	1	0
0	0	1

Sameer's Note

Fig. Sequence Table

Notice that an asynchronous up-down counter is slower than an up counter or a down counter because of the additional propagation delay introduced by the NAND networks.

Propagation Delay:

Asynchronous counters are commonly referred to as ripple counters for the following reason: The effect of the input clock pulse is first "felt" by FFO. This effect cannot get to FF1 immediately because of the propagation delay through FF0. Then there is the propagation delay through FF1 before FF2 can be triggered. Thus, the effect of an input clock pulse "ripples" through the counter, taking some time, due to propagation delays, to reach the last flip-flop.

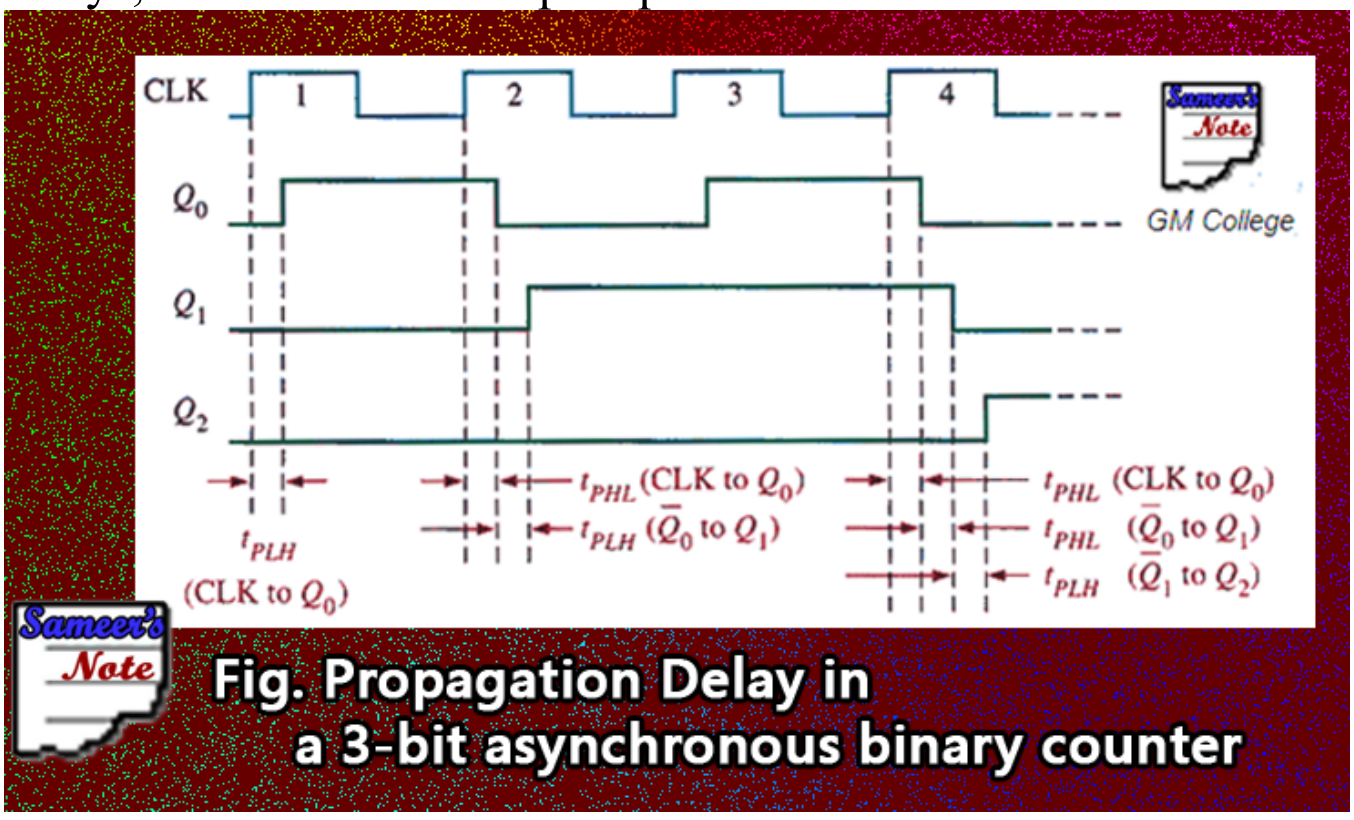


Fig. Propagation Delay in a 3-bit asynchronous binary counter

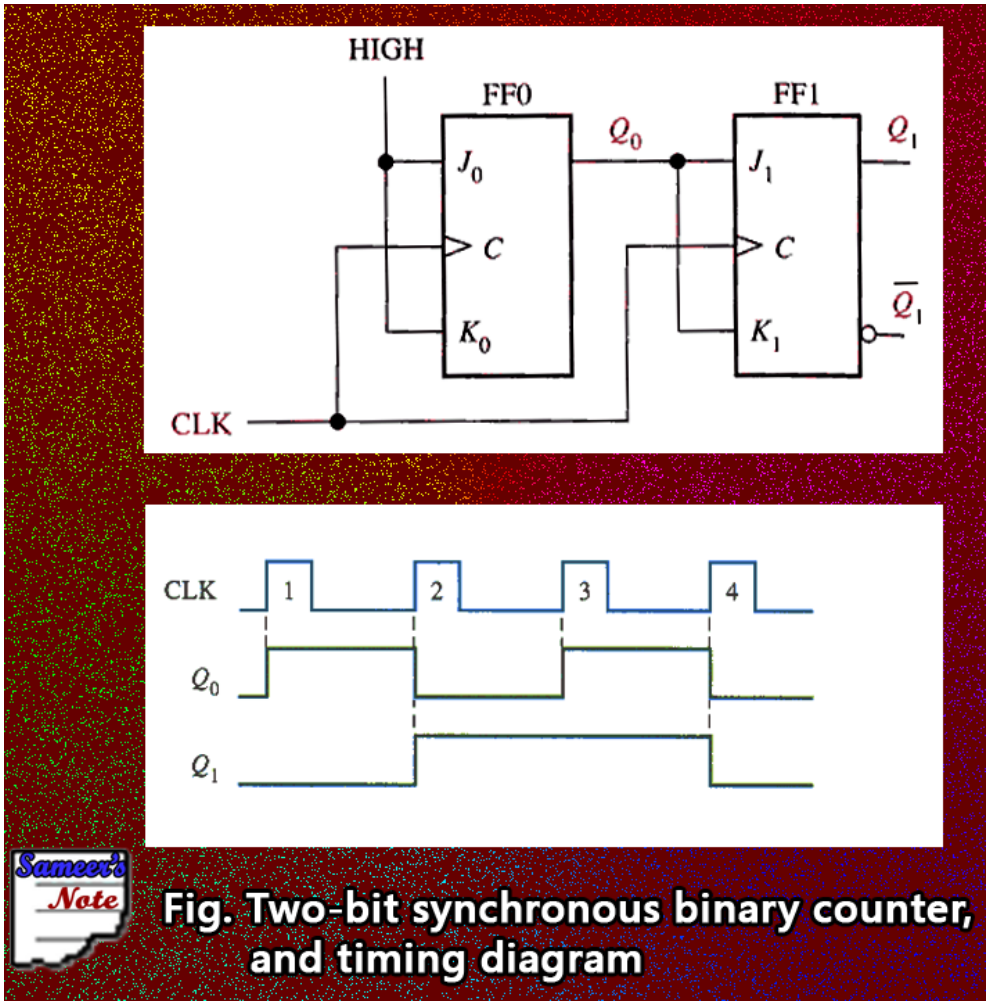
3.0 Synchronous Counters

In synchronous counters, the clock inputs of all the flip-flops are connected together and are triggered by the input pulses. Thus, all the flip-flops change state simultaneously (in parallel). Synchronous counters can be designed using the design procedure of the clocked sequential circuits. The count pulses are applied to the clock inputs of all flip-flops of the counter. Therefore the changes in the outputs occur at the same time. Several typical synchronous counters are presented in this ection and their operation explained.

3.1 2-Bit Synchronous Binary Counter

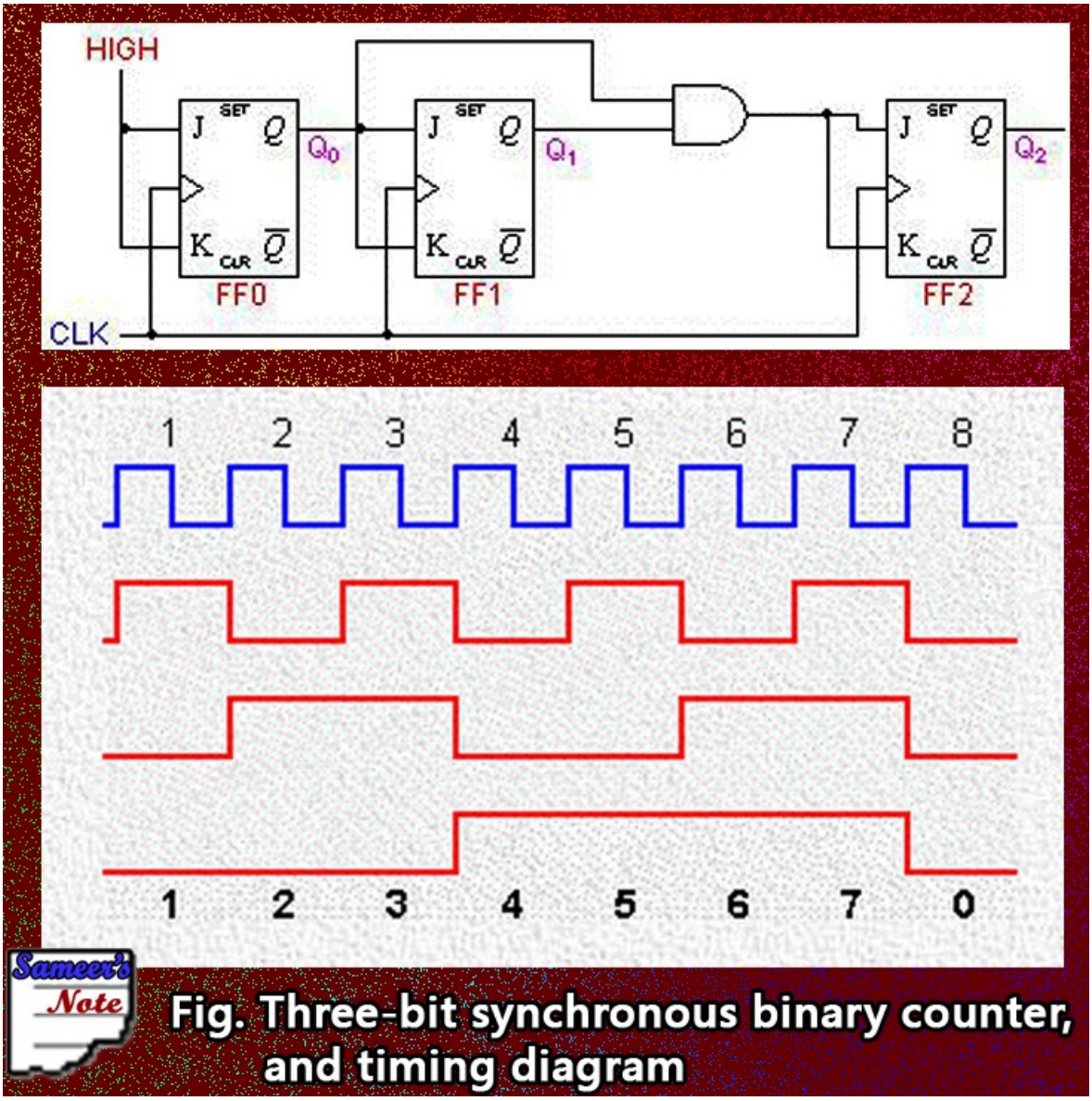
The operation of the synchronous binary counter is so simple and follows an easy complementing pattern that a simple design procedure can be followed. There is no need to follow the general procedure for the design of synchronous sequential circuits.

The circuit below is a basic 2-bit synchronous counter.



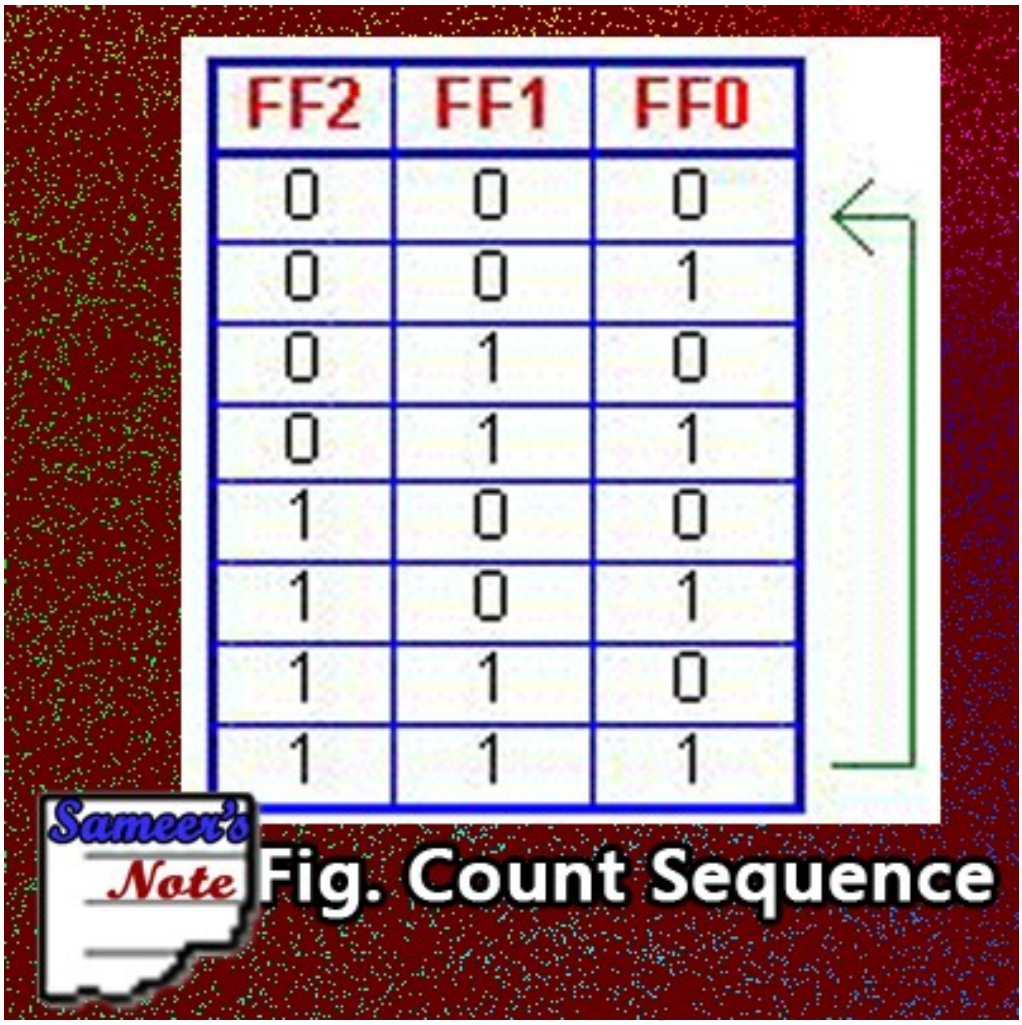
3.2 3-Bit Synchronous Binary Counter

The circuit below is a 3-bit synchronous counter. The J and K inputs of FF0 are connected to HIGH. FF1 has its J and K inputs connected to the output of FF0, and the J and K inputs of FF2 are connected to the output of an AND gate that is fed by the outputs of FF0 and FF1.



Pay attention to what happens after the 3rd clock pulse. Both outputs of FF0 and FF1 are HIGH. The positive edge of the 4th clock pulse will cause FF2 to change its state due to the AND gate.

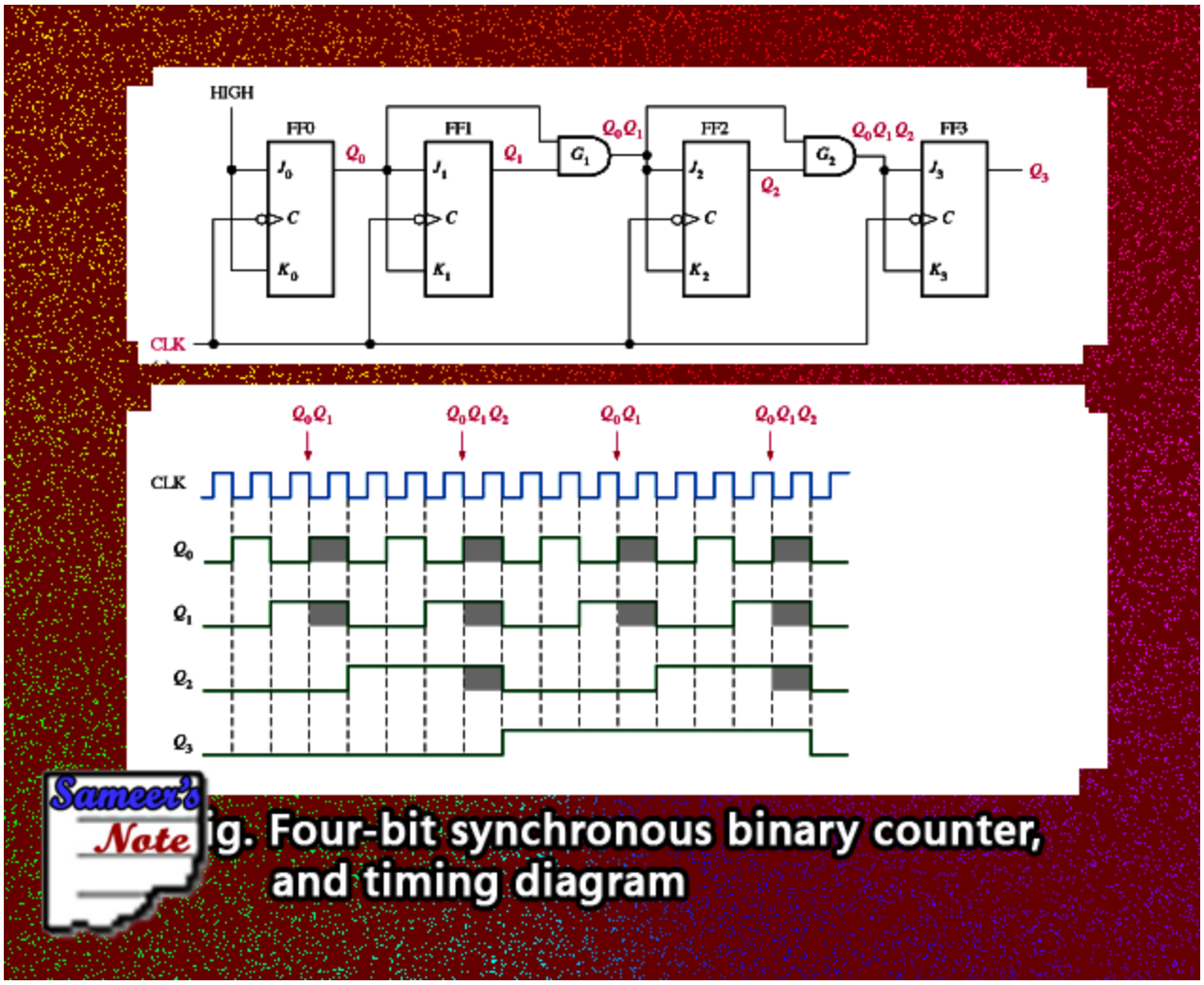
The count sequence for the 3-bit counter is shown in Figure



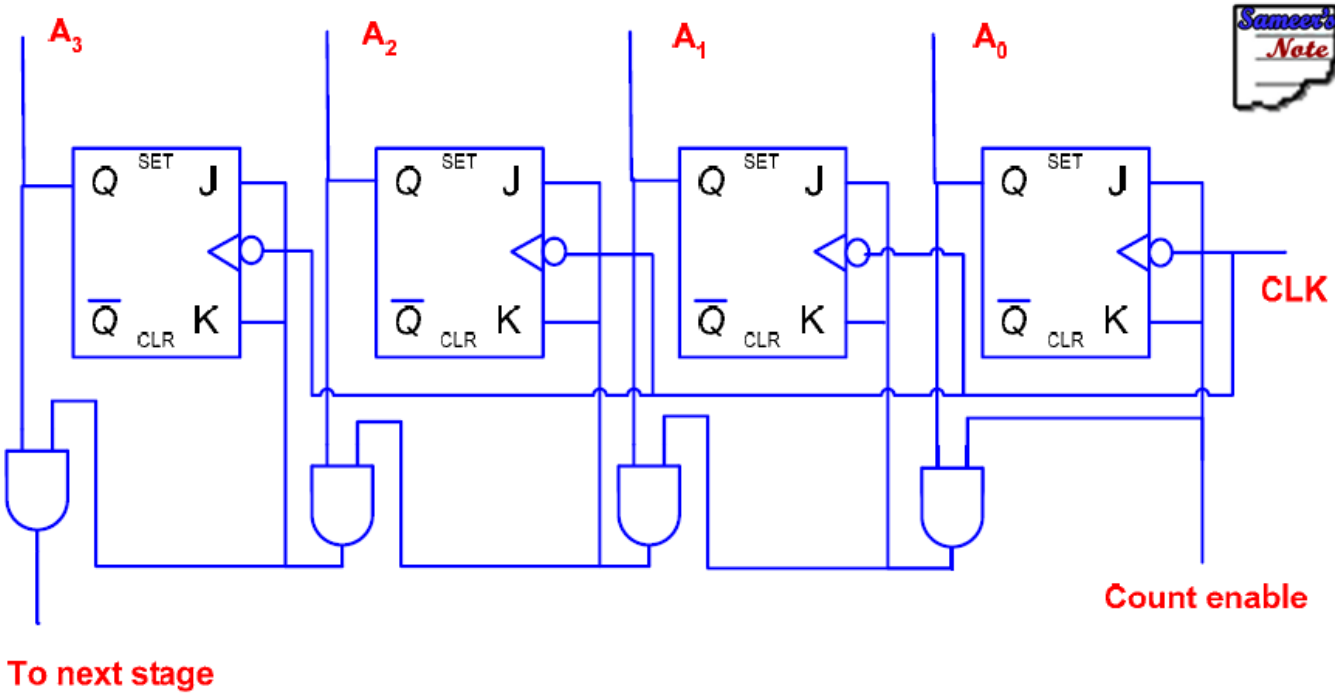
The most important advantage of synchronous counters is that there is no cumulative time delay because all flip-flops are triggered in parallel. Thus, the maximum operating frequency for this counter will be significantly higher than for the corresponding ripple counter.

3.3 4-Bit Synchronous Binary Counter

Following figure shows a 4-bit synchronous binary counter and its timing diagram.



The Logic Circuit is shown below:



If we inspect the count cycle or the timing diagram of the binary counter we find the following:

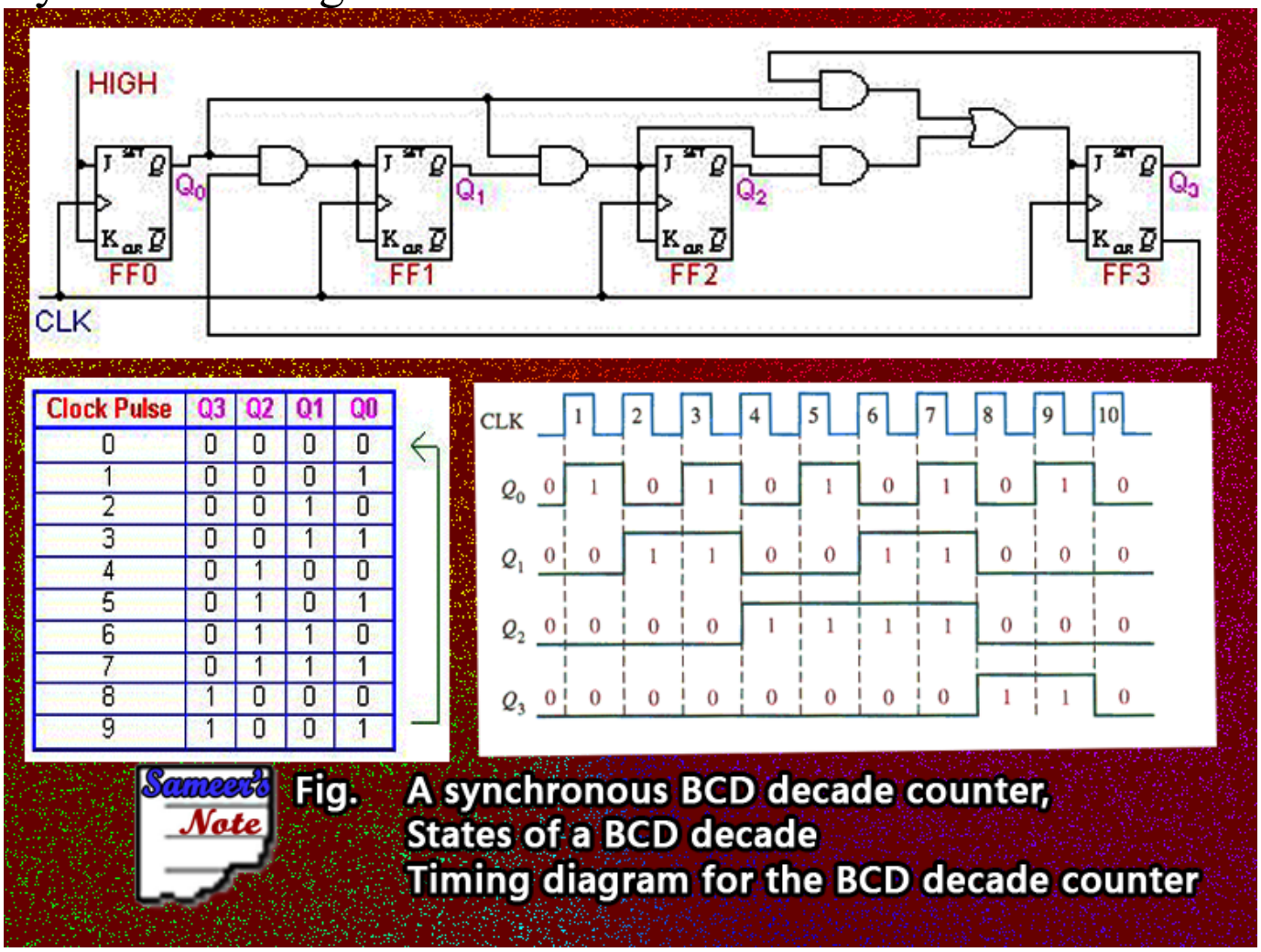
1. A_0 complements every time the count pulses go from 1 to 0.
2. A_1 complements only when A0 is 1 and goes to 0.
3. A_2 complements only when A1 and A0 are 1 and going to 0.
4. A_3 complements only when A_2, A_1 and A_0 are all 1 and going to 0.

From this pattern it becomes clear that the J and K inputs of flip-flop A0 should be kept at 1 (the count enable can be used in stead of 1). The count enable is Anded with A0 and applied to the J and k inputs of A1. This pattern is repeated by Anding the J and K input of each flip-flop with the flip-

flop output and applied to the J and K inputs of the next flip-flop.

3.4 Synchronous Decade Counters

Similar to an asynchronous decade counter, a synchronous decade counter counts from 0 to 9 and then recycles to 0 again. This is done by forcing the 1010 state back to the 0000 state. This so called truncated sequence can be constructed by the following circuit.



From the sequence in the above figure, we notice that:

- Q_0 toggles on each clock pulse.

- Q_1 changes on the next clock pulse each time $Q_0=1$ and $Q_3=0$.
- Q_2 changes on the next clock pulse each time $Q_0=Q_1=1$.
- Q_3 changes on the next clock pulse each time $Q_0=1$, $Q_1=1$ and $Q_2=1$ (count 7), or then $Q_0=1$ and $Q_3=1$ (count 9).

Flip-flop 2 (Q_2) changes on the next clock pulse each time both $Q_0=1$ and $Q_1=1$. Thus we must have

$$J2 = K2 = Q0Q1$$

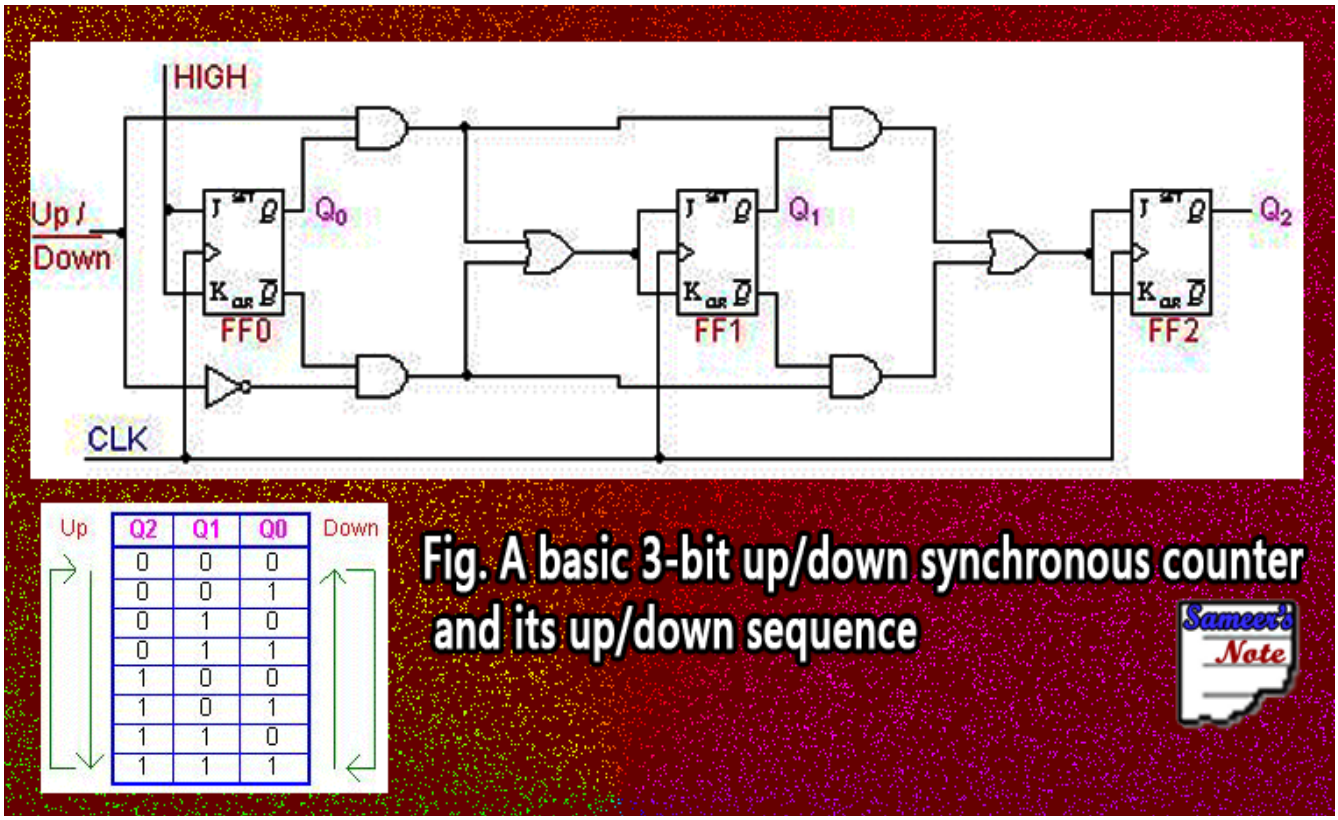
Flip-flop 3 (Q3) changes to the opposite state on the next clock pulse each time Q0=1, Q1=1, and Q2=1 (state 7), or when Q0=1 and Q3=1 (state 9). Thus we must have

$$J3 = K3 = Q0Q1Q2 + Q0Q3$$

These characteristics are implemented with the AND/OR logic connected as shown in the logic diagram.

3.5 Up-Down Synchronous Counters

A circuit of a 3-bit synchronous up-down counter and a table of its sequence are shown in Figure 3.6. Similar to an asynchronous up-down counter, a synchronous up-down counter also has an up-down control input. It is used to control the direction of the counter through a certain sequence.



An examination of the sequence table shows:

- For both the UP and DOWN sequences, Q_0 toggles on each clock pulse.
- For the UP sequence, Q_1 changes state on the next clock pulse when $Q_0=1$.
- For the DOWN sequence, Q_1 changes state on the next clock pulse when $Q_0=0$.
- For the UP sequence, Q_2 changes state on the next clock pulse when $Q_0=Q_1=1$.
- For the DOWN sequence, Q_2 changes state on the next clock pulse when $Q_0=Q_1=0$.

These characteristics are implemented with the AND, OR & NOT logic connected as shown in the above figure.

The BCD counter does not have a regular pattern. Therefore we have to follow the design procedure of synchronous

sequential circuits. Using T flip-flops, the state table of the counter will be given as follows:

P.S.				N.S.					Flip-flop inputs			
Q_8	Q_4	Q_2	Q_1	Q_8	Q_4	Q_2	Q_1	y	T_8	T_4	T_2	T_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

PS – Present State, NS – Next State

The flip-flop input equations can be simplified by means of Karnaugh maps:

$$T_1 = 1$$

$$T_2 = Q_8' Q_1$$

$$T_4 = Q_2 Q_1$$

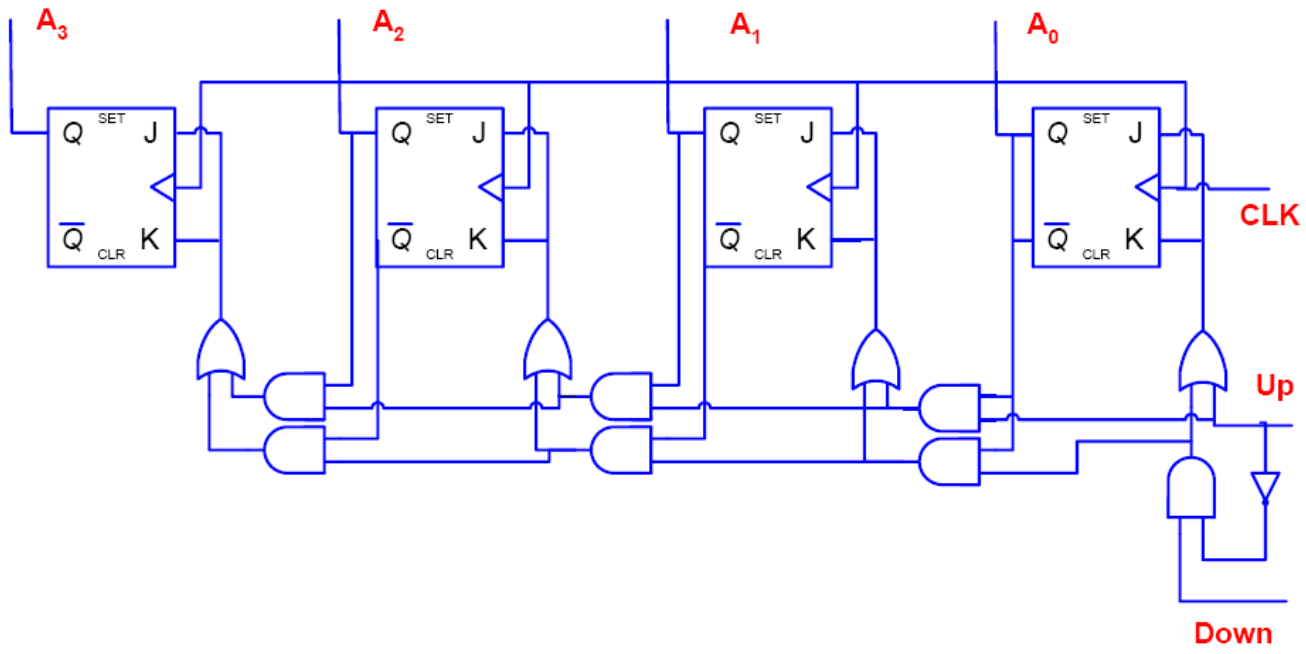
$$T_8 = Q_8 Q_1 + Q_4 Q_2 Q_1$$

And $$y = Q_8 Q_1$$

4-bit synchronous up-down counter

The down counter counts in reverse from 1111 to 0000 and then goes to 1111. If we inspect the count cycle, we find that each flip-flop will complement when the previous flip-flops are all 0 (this is the opposite of the up counter). The down

counter can be implemented similar to the up counter, except that the AND gate input is taken from Q' instead of Q. This is shown in the following Figure of a 4-bit up-down counter using T flip-flops.



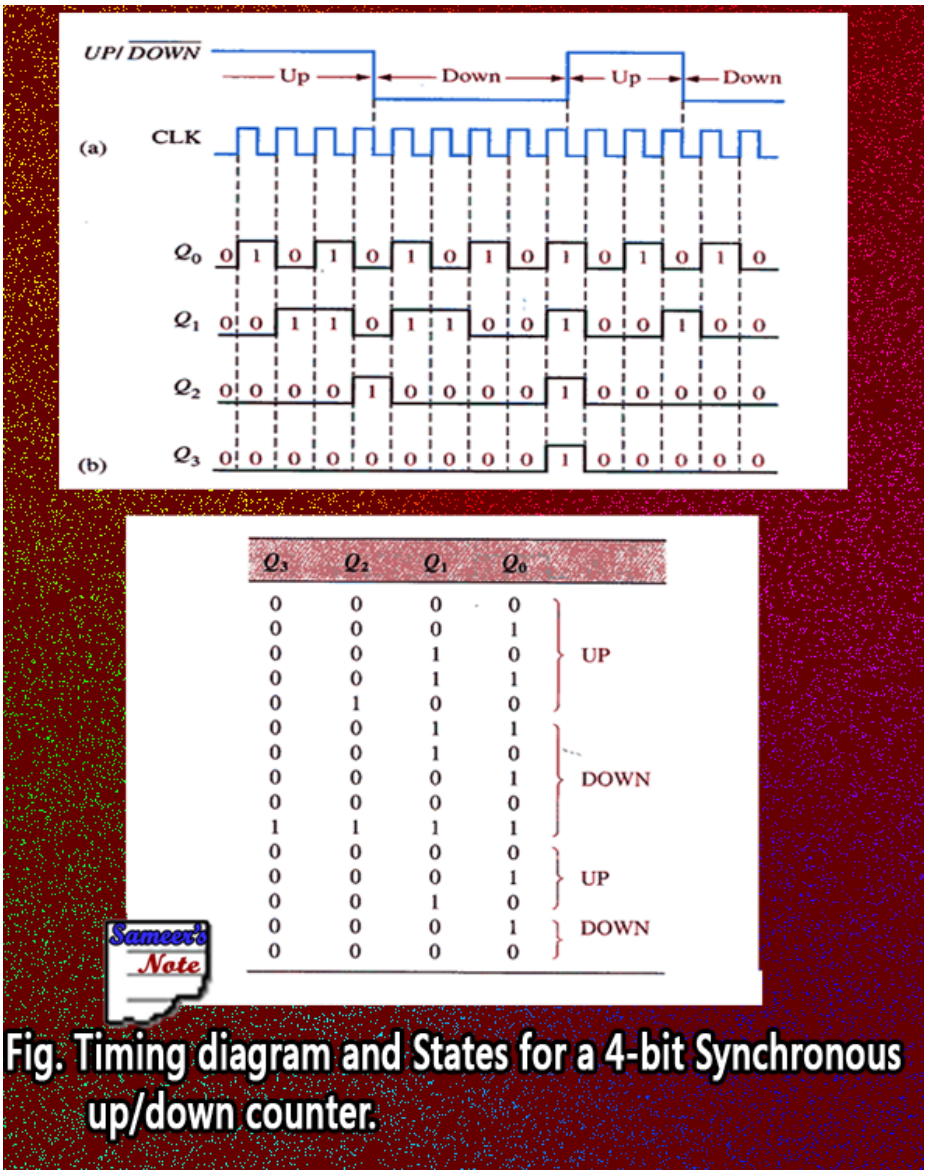
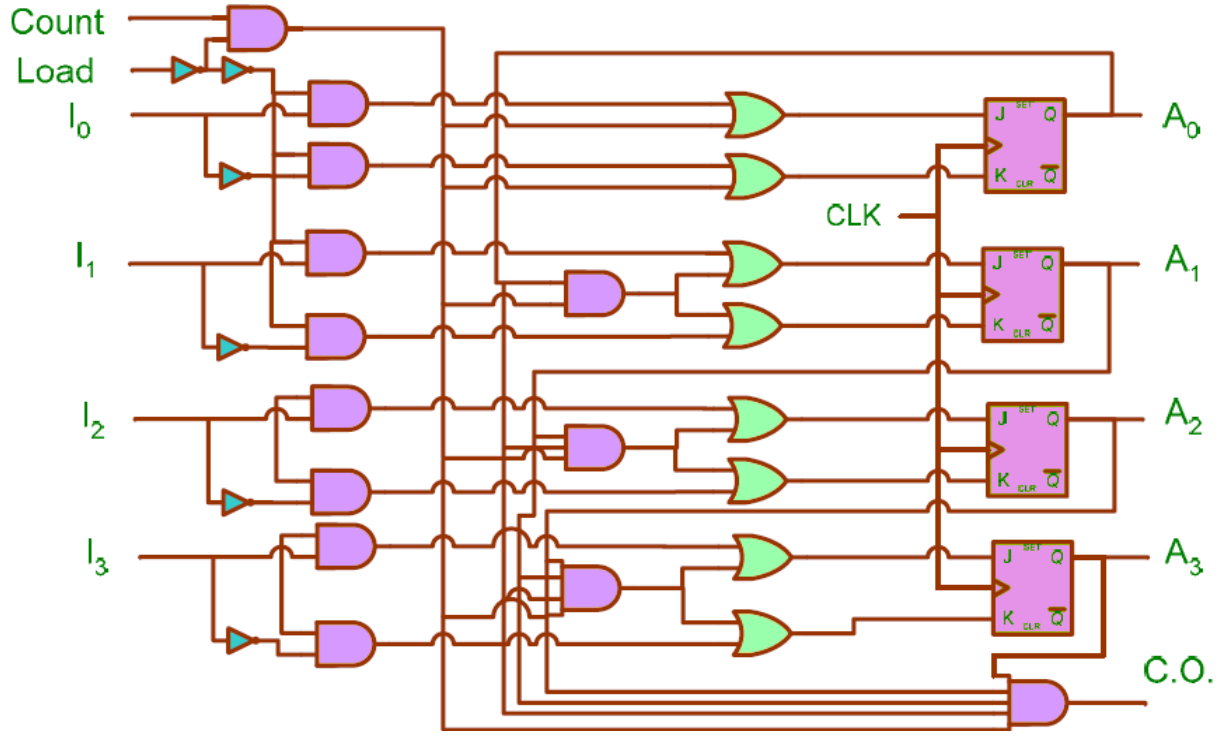


Fig. Timing diagram and States for a 4-bit Synchronous up/down counter.

Binary counter with parallel load

Parallel load capability is used for transmission of an initial binary number into the counter prior to the count operation. The logic diagram of a 4-bit register with parallel load, which can be used as a counter. When the input load is one, it disables the count operation and allows the transfer of the data inputs into the flip-flops. If the count and load inputs are both zero, then the outputs will not change. Only when the count is one and the load is zero, then the register will act as

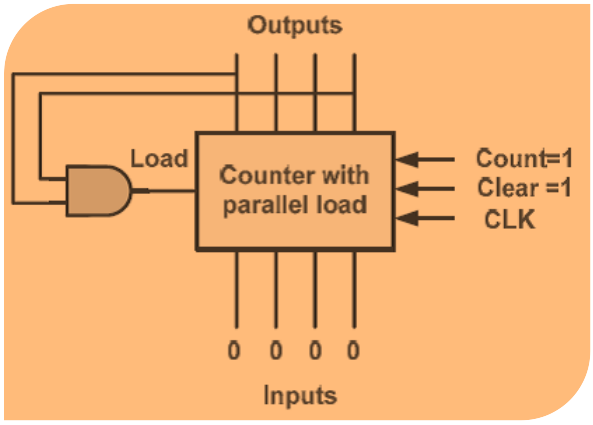
a counter and starts counting from the last output. The function table of this counter with parallel load is shown, followed by the logic diagram.



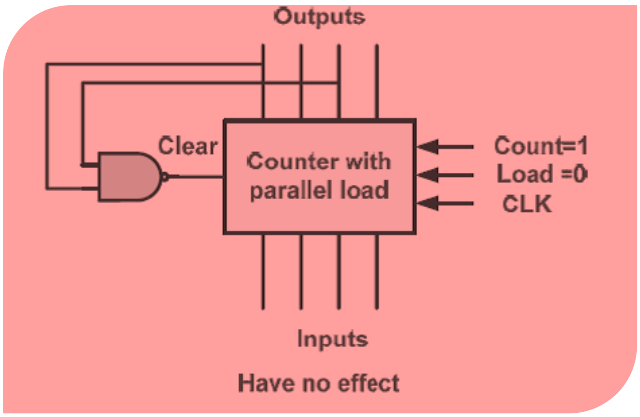
Clear	Clock	Load	Count	Function
0	X	X	X	Clear
1	↑	1	X	Load I's
1	↑	0	1	Count next
1	↑	0	0	No change
1	↓	0	0	No change

A counter with parallel load can be used to generate any desired count sequence. In the first example, the counter is used to generate the BCD count sequence by using the load input to load four zeros after reaching 1001. In the second example, the clear input is used to clear the counter after detecting that the count passed 1001 and attempts to go to 1010.

BCD counter , using the counter with parallel load and the load input:



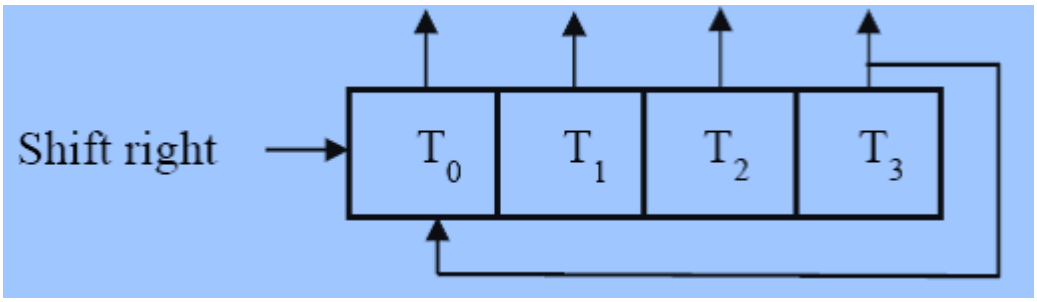
The second example is fro the BCD counter using the counter with parallel load. This time the clear input is used instead of the load input.



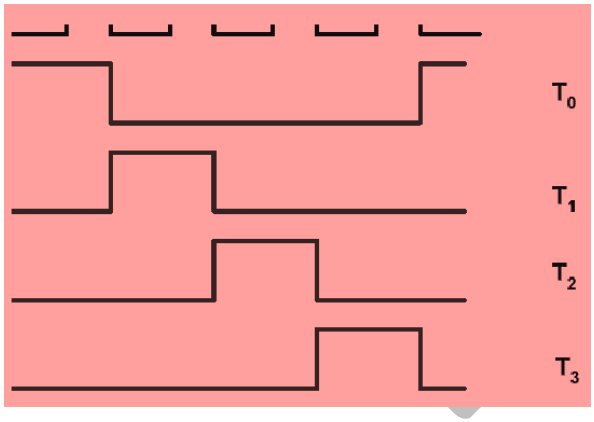
Ring Counter

Ring counter is useful in generating the timing signals that control the sequence of operations in a digital system. A ring counter is a circular shift register with only one flip-flop set at any particular time while all the others are cleared. The single bit is shifted from one flip-flop to the next to produce the sequence of timing signals. This is demonstrated by the

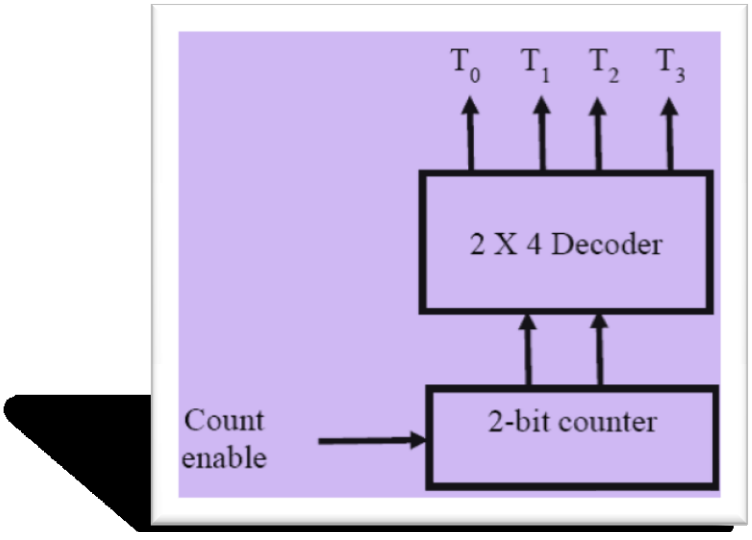
4-bit shift register connected as a ring counter. The generated timing signals are also shown.



Sequence of the four timing signals



The ring counter can also be implemented with a counter and a decoder. The 4-bit ring counter needs a 2-bit counter and a 2X4 decoder. This is illustrated in the following block diagram.



The Memory Unit

System Memory - The system memory is the place where the computer holds current programs and data that are in use. The term "memory" is somewhat ambiguous; it can refer to many different parts of the PC because there are so many different kinds of memory that a PC uses. However, when used by itself, "memory" usually refers to the main system memory, which holds the instructions that the processor executes and the data that those instructions work with. Your system memory is an important part of the main processing subsystem of the PC, tied in with the processor, cache, motherboard and chipset.

Memory plays a significant role in the following important aspects of your computer system:

Performance: The amount and type of system memory you have is an important contributing factor to overall performance. In many ways, it is more important than the processor, because insufficient memory can cause a processor to work at 50% or even more below its performance potential. This is an important point that is often overlooked.
Software Support: Newer programs require more memory than old ones. More memory will give you access to programs that you cannot use with a lesser amount.
Reliability and Stability: Bad memory is a leading cause of mysterious system problems. Ensuring you have high-quality memory will result in a PC that runs smoothly and exhibits fewer problems. Also, even high-quality memory will not work well if you use the wrong kind.
Upgradability: There are many different types of memory available, and some are more universal than others. Making a wise choice can allow you to migrate your memory it to a future system or continue to use it after you upgrade your motherboard.

Memory Technology Types - The system memory itself is made from DRAM, but the other types are explained here to show the other major technology types in use in the PC, and how they differ from DRAM.

ROM: Read-Only Memory. One major type of memory that is used in PCs is called read-only memory, or ROM for short. ROM is a type of memory that normally can only be read, as opposed to RAM which can be both read and written. There are two main reasons that read-only memory is used for certain functions within the PC: Permanence. The values stored in ROM are always there, whether the power is on or not. A ROM can be removed from the PC, stored for an indefinite period of time, and then replaced, and the data it contains will still be there. For this reason, it is called non-volatile storage. A hard disk is also non-volatile, for the same reason, but regular RAM is not.Security: The fact that ROM cannot easily be modified provides a measure of security against accidental (or malicious) changes to its contents. You are not going to find viruses infecting true ROMs, for example; it's just not possible. (It's technically possible with erasable EPROMs, though in practice never seen.)
Read-only memory is most commonly used to store system-level programs that we want to have available to the PC at all times. The most common example is the system BIOS program, which is stored in a ROM called (amazingly enough) the system BIOS ROM. Having this in a permanent ROM means it is available when the power is turned on so that the PC can use it to boot up the system. Remember that when you first turn on the PC the system memory is empty, so there has to be something for the PC to use when it starts up.
RAM: Random Access Memory. The kind of memory used for holding programs and data being executed is called random access memory or RAM. RAM differs from read-only memory (ROM) in that it can be both read and written. It is considered volatile storage because unlike ROM, the contents of RAM are lost when the power is turned off. RAM is also sometimes called read-write memory or RWM. It's a better name because calling RAM "random access" implies to some people that ROM isn't random access, which is not true. RAM is called "random access" because earlier read-write memories were sequential and did not allow random access. Sometimes old acronyms persist even when they don't make much sense any more. Obviously, RAM needs to be writable in order for it to do its job of holding programs and data that you are working on. The volatility of RAM also means that you risk losing what you are working on unless you save it frequently. RAM is much faster than ROM is, due to the nature of how it stores information. This is why RAM is often used to shadow the BIOS ROM to improve performance when executing BIOS code. There are many different types of RAMs, including static RAM (SRAM) and many flavors of dynamic RAM (DRAM).
SRAM: Static RAM. Static RAM is a type of RAM that holds its data without external refresh, for as long as power is supplied to the circuit. This is contrasted to dynamic RAM (DRAM), which must be refreshed many times per second in order to hold its data contents. SRAMs are used for specific applications within the PC, where their strengths outweigh their weaknesses compared to DRAM: Simplicity: SRAMs don't require external refresh circuitry or other work in order for them to keep their data intact. Speed: SRAM is faster than DRAM. In contrast, SRAMs have the following weaknesses, compared to DRAMs: Cost: SRAM is, byte for byte, several times more expensive than DRAM. Size: SRAMs take up much more space than DRAMs (which is part of why

the cost is higher).
These advantages and disadvantages taken together obviously show that performance-wise, SRAM is superior to DRAM, and we would use it exclusively if only we could do so economically. Unfortunately, 32 MB of SRAM would be prohibitively large and costly, which is why DRAM is used for system memory. SRAMs are used instead for level 1 cache and level 2 cache memory, for which it is perfectly suited; cache memory needs to be very fast, and not very large.
DRAM: Dynamic RAM. Dynamic RAM is a type of RAM that only holds its data if it is continuously accessed by special logic called a refresh circuit. Many hundreds of times each second, this circuitry reads the contents of each memory cell, whether the memory cell is being used at that time by the computer or not. Due to the way in which the cells are constructed, the reading action itself refreshes the contents of the memory. If this is not done regularly, then the DRAM will lose its contents, even if it continues to have power supplied to it. This refreshing action is why the memory is called dynamic.
All PCs use DRAM for their main system memory, instead of SRAM, even though DRAMs are slower than SRAMs and require the overhead of the refresh circuitry. It may seem weird to want to make the computer's memory out of something that can only hold a value for a fraction of a second. In fact, DRAMs are both more complicated and slower than SRAMs.
The reason that DRAMs are used is simple: they are much cheaper and take up much less space, typically 1/4 the silicon area of SRAMs or less. To build a 64 MB core memory from SRAMs would be very expensive. The overhead of the refresh circuit is tolerated in order to allow the use of large amounts of inexpensive, compact memory. The refresh circuitry itself is almost never a problem; many years of using DRAM has caused the design of these circuits to be all but perfected.
DRAMs are smaller and less expensive than SRAMs because SRAMs are made from four to six transistors (or more) per bit, DRAMs use only one, plus a capacitor. The capacitor, when energized, holds an electrical charge if the bit contains a "1" or no charge if it contains a "0". The transistor is used to read the contents of the capacitor. The problem with capacitors is that they only hold a charge for a short period of time, and then it fades away. These capacitors are tiny, so their charges fade particularly quickly. This is why the refresh circuitry is needed: to read the contents of every cell and refresh them with a fresh "charge" before the contents fade away and are lost. Refreshing is done by reading every "row" in the memory chip one row at a time; the process of reading the contents of each capacitor re-establishes the charge. For an explanation of how these "rows" are read, and thus how refresh is accomplished, refer to this section describing memory access.
DRAM is manufactured using a similar process to how processors are: a silicon substrate is etched with the patterns that make the transistors and capacitors (and support structures) that comprise each bit. DRAM costs much less than a processor because it is a series of simple, repeated structures, so there isn't the complexity of making a single chip with several million individually-located transistors.
There are many different kinds of specific DRAM technologies and speeds that they are available in. These have evolved over many years of using DRAM for system memory, and are discussed in more detail in the following.

Memory Access and Access Time - When memory is read or written, this is called a memory access. A specific procedure is used to control each access to memory, which consists of having the memory controller generate the correct signals to specify which memory location needs to be accessed, and then having the data show up on the data bus to be read by the processor or whatever other device requested it.

In order to understand how memory is accessed, it is first necessary to have a basic understanding of how memory chips are addressed. Let's take as an example a common 16Mbit chip, configured as 4Mx4. This means that there are 4M (4,194,304) addresses with 4 bits each; so there are 4,194,304 different memory locations--sometimes called cells--each of which contains 4 bits of data.. 4,194,304 is equal to 2^22, which means 22 bits are required to uniquely address that number of memory locations. Thus, in theory 22 address lines are required.

However, in practice, memory chips do not have this many address lines. They are instead logically organized as a "square" of rows and columns. The low-order 11 bits are considered the "row" and the high-order 11 bits the "column". First the row address is sent to the chip, and then the column address. For example, let's suppose that we want to access memory location 2,871,405 in this chip. This corresponds to a binary address of "10101111010 00001101101". First, "00001101101" would be sent to select the "row", and then "10101111010" would be sent to select the column. This combination selects the unique location of memory address 2,871,405. This is analogous to how you might select a particular cell on a spreadsheet: go to row #34, say, and then look at column "J" to find cell "J34".

Intuitively, it would seem that designing memory chips in this manner is both more complex and slower than just putting one address pin on the chip for each address line required to uniquely address the chip--why not just put 22 address pins on the chip? It may not surprise you to learn that the answer is "cost". By using the row/column

method, it is possible to greatly reduce the number of pins on the DRAM chip. Here, 11 address pins are required instead of 22 (though you lose a small part of the "savings" of 22-11=11 to additional control signals that are needed to manage the row/column timing.) You also save some of the buffers and other circuitry that are required for each address line. Certainly having to send the address in two "chunks" slows down the addressing process, but keeping the chip smaller and with fewer inputs allows it to use less power, which makes it possible to run the chip faster, partially offsetting the loss in access speed.

Of course, a PC doesn't have a single memory chip; most have dozens, depending on total memory capacity and the size of DRAMs being used. The chips are arranged into modules, and then into banks, and the memory controller manages which sets of chips are read from or written to. Since a modern PC reads or writes 64 bits at a time, each read or write involves simultaneous accesses to as many as 64 different DRAM chips.

Here is a simplified walkthrough of how a basic read memory access is performed. This is a conventional asynchronous read, because the timing signals are not tied to the main system clock; synchronous DRAM uses different timing signals:

The address for the memory location to be read is placed on the address bus.
The memory controller decodes the memory address and determines which chips are to be accessed.
The lower half of the address ("row") is sent to the chips to be read.
After allowing sufficient time for the row address signals to stabilize, the memory controller sets the row address strobe (sometimes called row address select) signal to zero. (This line is abbreviated as "RAS" with a horizontal line over it. The horizontal line is a short-hand code that tell engineers working with the circuit that the signal is "active low", meaning that the chip is looking for it to be set to zero as a signal to "do something". There's no way in HTML to reliably use this notation so instead, I will write "/RAS".)
When the /RAS signal has settled at zero, the entire row selected (all 2^11 columns in the example above, or 2048 different cells of 4 bits each) is read by the circuits in the chip. Note that this action refreshes all the cells in that row; refreshing is done one row at a time.
The higher half of the address ("column") is sent to the chips to be read.
After allowing sufficient time for the column address signals to stabilize, the memory controller sets the column address strobe (or column address select) signal to zero. This line is abbreviated as "CAS" with a horizontal line over it, or "/CAS".
When the /CAS signal has settled at zero, the selected column is fed to the output buffers of the chip.
The output buffers of all the accessed memory chips feed the data out onto the data bus, where the processor or other device that requested the data can read it.
Note that this is a very simplified example, since it doesn't mention all of the various timing signals, and it also ignores common performance enhancements such as multiple-banked modules, burst mode, etc. A write process is performed similarly, except of course that the data is read into the chips instead of being sent out by them. A special signal called "R/W" (actually written with a horizontal line over the "W") controls whether a read or write is being performed during the access.

The amount of time that it takes for the memory to produce the data required, from the start of the access until when the valid data is available for use, is called the memory's access time, sometimes abbreviated tAC. It is normally measured in nanoseconds (ns). Today's memory normally has access time ranging from 5 to 70 nanoseconds. This is the speed of the DRAM memory itself, which is not necessarily the same as the true speed of the overall memory system. Note that much of the difference in access times of various DRAM technologies has to do with how the memory chips are arranged and controlled, not anything different in the core DRAM chips themselves.

Asynchronous and Synchronous DRAM - Conventional DRAM, of the type that has been used in PCs since the original IBM PC days, is said to be asynchronous. This refers to the fact that the memory is not synchronized to the system clock. A memory access is begun, and a certain period of time later the memory value appears on the bus. The signals are not coordinated with the system clock at all, as described in the section discussing memory access. Asynchronous memory works fine in lower-speed memory bus systems but is not nearly as suitable for use in high-speed (>66 MHz) memory systems.

A newer type of DRAM, called "synchronous DRAM" or "SDRAM", is synchronized to the system clock; all signals are tied to the clock so timing is much tighter and better controlled. This type of memory is much faster than asynchronous DRAM and can be used to improve the performance of the system. It is more suitable to the higher-speed memory systems of the newest PCs.

Note that there are several different flavors of both asynchronous DRAM and synchronous DRAM; they are discussed on the page covering DRAM technologies.

DRAM Speed Ratings

There are two different ways that DRAM chips are rated for speed. Conventional asynchronous DRAM chips have a rated speed in nanoseconds (ns, or a billionth of a second), a speed which represents the minimum access time for doing a read or write to memory. This includes the entire access cycle. Most asynchronous memory in modern systems is 50, 60 or 70 ns in speed. Older systems (386 and earlier) use usually 70 or 80 ns RAM. Very old systems use even slower memory: 100, 120 or even 150 ns. Systems running with a clock speed of 60 MHz or higher generally require 60 ns or faster memory to function at peak efficiency. 70 ns is fine for 486 or older PCs.

Synchronous memory is much faster than conventional asynchronous RAM. It is usually rated at 12, 10 or even 7 nanoseconds; however you have to be careful here. An SDRAM module rated at 10 ns is not "5 times faster" than an EDO module rated at 50 ns. Since SDRAM is synchronized to the internal system clock, SDRAM speed ratings refer to the maximum speed at which the SDRAM module can burst data onto the bus. This does not include the addressing latency time the way asynchronous DRAM speed ratings do, which is why the numbers are much smaller. The core DRAMs inside the SDRAM module are usually not any faster than those of older technologies; the increase in usable speed is due to how the module is constructed and controlled. See the following section on SDRAM for more details.

DRAM chips are usually marked with their speed via a suffix at the end of the part number. If you look at the chips themselves, you'll see something like "-6" or "-60". This usually means 60 nanosecond DRAM. The suffix found on SDRAM chips is often "-12", "-10" or "-07". Note that older memory running at 100 or 120 ns also used "-10" and "-12" sometimes. This memory hasn't been used in years so there really shouldn't be any confusion between the two types. However, 70 ns memory uses "-7" and this can be readily confused with 7 ns SDRAM memory if you are not careful.

In addition to being referred to using a nanosecond speed rating, SDRAMs are also often rated in terms of their maximum frequency, in MHz. This is really the same thing, just expressed in a different way: for example, an SDRAM module with a 10ns rating would be called instead a "100 MHz SDRAM". 100 MHz is 100 million cycles per second, which is the reciprocal of 10ns, one-hundred-millionth of a second per cycle. This MHz number is not the same as saying that the SDRAM with that rating is designed for a system of that speed. A 100 MHz SDRAM may not function in a 100 MHz system bus PC. See here for more.

The rated speed of the memory is a maximum. In theory, the memory cannot support memory timing that requires a faster speed of RAM. However, in practice many companies rate their DRAM conservatively, so that the memory will function at a higher speed than what is indicated. This is why many Pentium systems running on a 66 MHz bus will work with 70 ns memory, even when set to 60 ns timing. However, this is not reliable and cannot be counted on (in a way, it is a form of overclocking) and is not recommended. You can usually compensate for slower memory by turning down the system timing level, which will cause a small performance decrease but give you better reliability.

SDRAM speed ratings and selection criteria are more complicated than those for conventional asynchronous RAM. Refer to the section on SDRAM.

Mixing DRAM Speeds - Mixing memory speeds refers to the use of DRAM of more than one speed in the same computer. For example, you might have bought a machine in 1994 that came with 70 ns DRAM (the fastest speed generally available then) and later upgraded with more memory in 1997 that was 60 ns. While it's generally preferable to avoid doing this, it can work without problems as long as certain caveats are followed:
Use Identical Memory Within a Bank: PCs read a certain bit width of memory at a time, typically 32 or 64, and the memory making up this width is called a bank. PCs always read data from more than one chip at a time within a single bank of memory. If different memory is used within a bank, some of the bits may arrive later than others and all sorts of system problems can result. For this reason you should only use the same type and speed of memory within a bank. This also means using the same technology--never mix EDO and FPM memory (for example) within a bank.
Put The Slowest Memory in the First Bank: Some memory systems automatically detect the speed of the memory being used, and set the system timing accordingly. They usually only look at the speed of the memory in the first bank when setting the timing. If you have 60 ns RAM in the first bank and 70 ns in the second bank, the system may set the timing at a rate that works fine for the 60 ns memory, but causes problems for the 70 ns.

You should put the 70 ns memory in the first bank instead. (If your system doesn't do autodetection this won't be an issue but it is still good practice). Note that the first bank of memory is often called "Bank 0".
Some systems just generally have a hard time working with dissimilar banks of memory. I once tried to upgrade a system that had a pair of 8 MB, fast page mode 60 ns SIMMs, with another pair of 8 MB, fast page mode 60 ns SIMMs of another brand. The two pairs just would not work together in any configuration, even though they worked fine separately. See this section of the Troubleshooting Expert for more on RAM problems.

Memory Bus Speed and Required DRAM Speed - Most modern systems generally require the DRAMs they use to be a certain minimum speed. The speed required is normally a function of the speed of the memory bus. Faster memory buses require faster speed DRAMs and in some cases, faster technologies. While it is not always cut-and-dried what speed is necessary, the table below is a guideline. It is usually possible to use slower memory in any system if you slow down the memory timing, but of course we're trying to make our systems run as fast as we can.

Processor Generation	Memory Bus Speed	Usual DRAM Technology	Usual Required DRAM Speed (ns)
First, Second	4.77-20	Conventional	100-120
Third, Fourth	16-40	Conventional, Page Mode, FPM, EDO	70-100
Fifth, Sixth	50-100	FPM, EDO, BEDO, SDRAM	8-10 (SDRAM) 50-70 (Asynchronous)
Future	125+	SDRAM, DDR SDRAM, DRDRAM, SLDRAM, Other?	?

DRAM Technologies - DRAM is available in several different technology types. At their core, all of these different memory types are similar. They differ mostly in the way that they are organized and how they are accessed. As processors get faster, memory needs to run increasingly faster and more efficiently. Memory companies have invented progressively faster memory architectures to allow memory speeds to increase.

In the real world, the differences between many of the DRAM technologies is not that great. Most requests for data by the processor are satisfied from either the primary or secondary caches on a modern PC, which masks much of the improvement in DRAM efficiency. Also, memory is just one piece of the puzzle in overall performance. Often, more system memory is more important to performance than better system memory.

Also bear in mind that at its core, DRAM is DRAM. The differences between the various acronyms of DRAM technologies are primarily a result of how the DRAM inside the module is connected, configured and addressed, in addition to any special enhancement circuits added to the device. For example, some fancy modules include SRAM (cache) directly in the DRAM module to improve performance.

Generally, there are Conventional DRAM, Fast Page Mode (FPM) DRAM, Extended Data Out (EDO) DRAM, Burst Extended Data Out (BEDO) DRAM, Synchronous DRAM (SDRAM), Double Data Rate SDRAM (DDR SDRAM), Direct Rambus DRAM (DRDRAM), Synchronous-Link DRAM (SLDRAM), Video RAM (VRAM) and Other Video DRAM Technologies. Today, the most commonlly used is SDRAM.

Synchronous DRAM (SDRAM) - A relatively new and different kind of RAM, Synchronous DRAM or SDRAM differs from earlier types in that it does not run asynchronously to the system clock the way older, conventional types of memory do. SDRAM is tied to the system clock and is designed to be able to read or write from memory in burst mode (after the initial read or write latency) at 1 clock cycle per access (zero wait states) at memory bus speeds up to 100 MHz or even higher. SDRAM supports 5-1-1-1 system timing when used with a supporting chipset. SDRAM accomplishes its faster access using a number of internal performance improvements, including internal interleaving, which allows half the module to begin an access while the other half is finishing one.

SDRAM is rapidly becoming the new memory standard for modern PCs. The reason is that its synchronized design permits support for the much higher bus speeds that have started to enter the market. SDRAM doesn't offer that much "real world" additional performance over EDO in many systems, due to the system cache masking much of that differential in speed, and the fact that most systems are running on relatively slow 66 MHz or lower system bus speeds. As 100 MHz bus system PCs become mainstream, SDRAM will largely replace

older technologies, since it is designed to work at these higher operating speeds and conventional asynchronous DRAM is not.

There are several important characteristics and concerns regarding SDRAMs that are relatively unique to the technology. In addition to the notes below, you will want to read this informative article that goes into more depth on choosing SDRAM modules:

Speed and Speed Matching: SDRAM modules are generally speed-rated in two different ways: First, they have a "nanosecond" rating like conventional asynchronous DRAMs, so SDRAMs are sometimes referred to as being "12 nanosecond" or "10 nanosecond". Second, they have a "MHz" rating, so they are called "83 MHz" or "100 MHz" SDRAMs for example. Because SDRAMs are, well, synchronous, they must be fast enough for the system in which they are being used. With asynchronous DRAMs such as EDO or FPM, it was common to add extra wait states to the access timing for the memory to compensate for memory that was too slow. With SDRAM however, the whole point of the technology is to be able to run with zero wait states. In order to do this, the memory must be fast enough for the bus speed of the system. One place where people run into trouble in this regard is that they take the reciprocal of the "nanosecond" rating of the module and conclude that the module can run at that speed. For example, the reciprocal of 10 ns is 100 MHz, so people assume that 10 ns modules will definitely be able to run on a 100 MHz system. The problem is that this allows absolutely no room for slack. In practice, you really want memory rated slightly higher than what is required, so 10 ns modules are really intended for 83 MHz operation. 100 MHz systems require faster memory, which is why the **PC100** specification was developed (see below).
Speed Rating: Due to the confusion inherent in the speed rating system described immediately above, and the likelihood of problems running slower SDRAM modules on new 100 MHz system bus motherboards, Intel created a formal specification for SDRAM capable of being used in these new PCs. Dubbed PC100, these modules generally are rated at 8 ns as previously mentioned, but there are other internal timing characteristics that must be met in order to have a module certified as PC100-compliant. While relying on a specification is never foolproof, it is definitely a good idea to ensure that any SDRAM you intend to use on a 100 MHz system bus motherboard is in fact PC100 specification compliant.
Latency: SDRAMs are still DRAMs, and therefore still have latency. The fast 12, 10 and 8 nanosecond numbers that everyone talks about refer only to the second, third and fourth accesses in a four-access burst. The first access is still a relatively slow 5 cycles, just as it is for conventional EDO and FPM memory.
2-Clock and 4-Clock Circuitry: There are two slight variations in the composition of SDRAM modules; these are commonly called 2-clock and 4-clock SDRAMs. They are almost exactly the same, and they use the same DRAM chips, but they differ in how they are laid out and accessed. A 2-clock SDRAM is structured so that each clock signal controls 2 different DRAM chips on the module, while a 4-clock SDRAM has clock signals that can control 4 different chips each. You need to make sure that you get the right kind for your motherboard. The current trend appears to be toward 4-clock SDRAMs.
Serial Presence Detect: Some motherboards are now being created that require the use of special SDRAM modules that include something called a Serial Presence Detect (SPD) chip. This is an EEPROM that contains speed and design information about the module. The motherboard queries the chip for information about the module and makes adjustments to system operation based on what it finds. A great idea in theory, but you won't think it's great if you buy an SDRAM module without the chip on it when your board requires SPD...
CAS2 vs. CAS3: "CAS" stands for column address strobe, one of the main signals used in accessing DRAM chips; see here for an explanation of what CAS is all about. The terms "CAS2" and "CAS3" are used to distinguish between slight variants in SDRAM modules. In fact, the term is a misnomer; the "2" and "3" refer to the latency of the CAS line, so the terms should be "CL2" and "CL3". Theoretically a "CAS2" module is slightly faster than a "CAS3" module, making it more likely to function if the system bus is being overclocked beyond 100 MHz, but the whole matter of "CAS2" and "CAS3" has been overhyped to the Nth degree by many vendors. Dean Kent's article on SDRAM terminology explains this in greater detail.
Packaging Concerns: To make matters even more confusing, SDRAM usually comes in DIMM packaging, which itself comes in several different formats (buffered and unbuffered, 3.3 volts and 5 volts). You need to make sure you get the right type of packaging as well; see DIMMs section for detail about it..
Now that you've read that, do you feel a bit confused about exactly what type of SDRAM you need? I don't blame you! This is why I strongly advise working closely with your motherboard manufacturer and/or a trusted vendor in choosing your SDRAM. It's one thing to try to figure all of this out from specifications, but it's much better to contact the company that made your motherboard and have them say definitively that you need "10 ns, 4-clock, unbuffered, 3.3 volt SDRAM modules with serial presence detect", or whatever.

Double Data Rate SDRAM (DDR SDRAM) - Only a few years ago, "regular" SDRAM was introduced as a proposed replacement for the older FPM and EDO asynchronous DRAM technologies. This was due to the limitations the older memory has when working with systems using higher bus speeds (over 75 MHz). In the

next couple of years, as system bus speeds increase further, the bell will soon toll on SDRAM itself. One of the proposed new standards to replace SDRAM is Double Data Rate SDRAM or DDR SDRAM.

Memory Size - Every system has a maximum amount of memory and maximum cacheable memory that it will support. There are in fact several limiting factors that dictate how much memory can be used in any system. See following for how it influences the system performance.

Memory Size and System Performance

The amount of memory in a PC has a significant impact on its overall performance. Using too little RAM can be the biggest anchor dragging down overall system speed. This is something that many PC users fail to realize; I will often see someone posting to USEnet saying that their PC is "too slow" and so they want to upgrade to a faster processor. Then I will find out they are trying to run multiple applications or high-end games under Windows 95 but have only 8 MB of RAM. In a situation like this, upgrading the processor is a waste of money until the system memory is brought up to a more reasonable level.

Strictly speaking, the amount of memory in the computer has no impact on the speed that the memory runs, nor on the speed that the processor, chipset, motherboard and other major system components run. However, this is only if all of the programs running on the PC fit into the system RAM! All multitasking operating systems use virtual memory, which lets the PC think it has more memory than the actual physical RAM; the extra virtual memory is stored in a swap file on the hard disk. When more programs and data are in use than physically fit in memory, the virtual memory manager swaps parts of memory to disk. This is described in detail in the section discussing virtual memory.

When the amount of virtual memory in use greatly exceeds the amount of real memory, the operating system spends a lot of time swapping pages of memory around, which greatly hampers performance. The reason is simple: the hard disk is thousands of times slower than the system memory, if not more. Remember that hard disk access time is measured in thousandths of a second; memory access time is measured in billionths of a second. This doesn't tell the whole story but it gives you a general idea of the difference.

Let's suppose you are running a word processor, spreadsheet and a calendar program on Windows 95 on a system with 8 MB of real system memory. The total amount of virtual memory required by the applications you are using combine to most likely about 24 MB, depending on what versions of the software you are using. Windows 95 itself needs about 8 MB for system tasks. Since you only have 8 MB of memory, this means basically every time you do anything, the PC will have to pause and swap information to disk before proceeding. If you were to increase your system RAM to 32 MB, you could hold most (if not all) of the data in memory and the hard disk would be quiet. The improvement in performance is dramatic.

The best way to understand the importance of having enough memory is to compare PCs with more memory and with less when running similar tasks. I have seen 80486DX2-66s that feel faster than Pentium 133s, when the 486 has 32 MB of memory and the Pentium has only 8 MB. (I'm not exaggerating.) Even though the Pentium has probably three times the raw power, it is wasted because the system is spending so much time thrashing to the hard disk (while the CPU figuratively twiddles its thumbs).

So how much memory do you need? This is not an easy question to answer. It depends entirely on what you are using the system for. If you are running a single DOS application on a slower PC, 4 MB can be sufficient as long as the application doesn't need more than that. Many high-end CAD or graphics workstations use 256 MB of RAM or more. The amount of memory needed by PCs continues to increase as programs and data get larger and larger. A few years ago, 8 MB was considered a configuration for a high-end system; last year this would have been considered "entry level"; today, it is considered totally unacceptable. The trend towards much larger quantities of RAM will continue in the future, since the price of RAM continues to drop dramatically, and is now in the environs of $1/MB.

Tip: Watch your hard disk LED. If you see it come on and flicker rapidly when switching between tasks for example, this probably means that your operating system is being forced to use virtual memory. If this happens often, it is a clue that you may need more memory. (Note that when loading programs or doing other obviously disk-intensive work, having the disk light come on does not necessarily imply anything about virtual memory, of course.)

In general, more memory is better, however you have to watch out for the cacheability issue, since some PCs will not cache memory above a certain value. In addition, the law of diminishing returns definitely applies to memory size: each time you increase your system's memory, you get less improvement than you did the previous time you increased it. Going from 8 MB to 16 MB results in a huge performance increase for most Windows 95 users. Going from 16 to 24 MB results in some improvement but much less than that resulting from going from 8 to 16, and so on.

For most systems, there is a point beyond which adding more memory does not add appreciably to system performance. Where this point lies depends a great deal on what you use your system for--if you are using huge multimedia files for example your PC can probably make use of as much memory as you can throw at it. For most users the point of serious diminishing returns lies between 24 and 48 MB. Above about 48 MB, you will not see much noticeable improvement unless you are doing high-end graphics work, manipulating large files, or multitasking like crazy.

The table below shows some sample operating systems and application types, along with general ranges for recommended memory. This is a guideline only and should not be taken as definitive law; if you are using Windows NT and several applications, I can guarantee that 32 MB is going to be too little, pretty quickly. Only you can assess how you use your PC and therefore what makes the most sense. (That said, I will also say that very few people will want to be below the numbers in the "minimum" column below):

Operating System and Application	Minimum RAM for Tolerable Performance	Minimum RAM for Good Performance	Typical Point of Diminishing Returns
DOS, Single Tasking (not games)	4 MB	8 MB	16 MB
Office work, Windows 3.x	8 MB	16 MB	32 MB
Windows 95 Multitasking, High-end DOS games	16 MB	32 MB	64 MB
Windows NT Workstation	24-32 MB	48 MB	128+ MB
High-end use, Graphics processing, Multimedia, Servers	32 MB	64+ MB	In many cases, none

There is one other way that memory size impacts on overall system performance. Having more memory allows you to dedicate some of it for use as a disk cache. A disk cache lets you store recently-used information from the disk in a special area of memory, to save time reading to the disk when it is needed again. This improves system performance by avoiding unnecessary reads and writes to the slow hard disk.

Memory Packaging - Like processors, memory is made from tiny semiconductor chips and must be packaged into something less fragile and tiny in order to be integrated with the rest of the system. However, in many cases the chip packages are themselves further integrated into larger packages. There are many different kinds of memory packages in the PC world today, and it can be difficult to know which type needs to be used with a given system design. The following are common memory packaging types: Dual Inline Packages (DIPs) and Memory Modules, Parity, Non-Parity and ECC Memory, Standard and Proprietary Memory Modules, Single Inline Memory Modules (SIMMs), Dual Inline Memory Modules (DIMMs).

Single Inline Memory Modules (SIMMs) - The single inline memory module or SIMM is still the most common memory module format in use in the PC world, largely due to the enormous installed base of PCs that use them (in new PCs, DIMMs are now overtaking SIMMs in popularity.) SIMMs are available in two flavors: 30 pin and 72 pin. 30-pin SIMMs are the older standard, and were popular on third and fourth generation motherboards. 72-pin SIMMs are used on fourth, fifth and sixth generation PCs.

SIMMs are placed into special sockets on the motherboard created to hold them. The sockets are specifically designed to ensure that once inserted, the SIMM will be held in place tightly. SIMMs are secured into their sockets (in most cases) by inserting them at an angle (usually about 60 degrees from the motherboard) into the base of the socket and then tilting them upward until they are perpendicular to the motherboard. Special metal clips on either side of the socket snap in place when the SIMM is inserted correctly. The SIMM is also keyed with a notch on one side, to make sure it isn't put in backwards.

The 30 pin SIMMs are generally available in sizes from 1 to 16 MB. Each one has 30 pins of course, and provides one byte of data (8 bits), plus 1 additional bit for parity with parity versions. 72-pin SIMMs provide four

bytes of data at a time (32 bits) plus 4 bits for parity/ECC in parity/ECC versions. Package bit width is discussed in detail here.

SIMMs are available in two styles: single-sided or double-sided. This refers to whether or not DRAM chips are found on both sides of the SIMM or only on one side. 30-pin SIMMs are all (I am pretty sure) single-sided. 72-pin SIMMs are either single-sided or double-sided. Some double-sided SIMMs are constructed as composite SIMMs. Internally, they are wired as if they were actually two single-sided SIMMs back to back. This doesn't change how many bits of data they put out or how many you need to use. However, some motherboards cannot handle composite SIMMs because they are slightly different electrically.

72-pin SIMMs that are 1 MB, 4 MB and 16 MB in size are normally single-sided, while those 2 MB, 8 MB and 32 MB in size are generally double-sided. This is why there are so many motherboards that will only work with 1 MB, 4 MB and 16 MB SIMMs. You should always check your motherboard to see what sizes of SIMMs it supports. Composite SIMMs will not work in a motherboard that doesn't support them. SIMMs with 32 chips on them are almost always composite.

Warning: Lately, some 16 MB and 64 MB SIMMs have been seen that are composite. These can cause significant problems with some motherboards, since they are specified to support 16 MB SIMMs on the expectation that 16 MB SIMMs will all be single-sided. You may not be able to use double-sided 16 MB SIMMs in some systems, especially older or cheaper ones.

Most motherboards support either 30-pin or 72-pin SIMMs, but not both. Some 486 motherboards do support both, however. In many cases these motherboards have significant restrictions on how these SIMMs can be used. For example, only one 72-pin socket may be usable if the 30-pin sockets are in use, or double-sided SIMMs may not be usable.

Dual Inline Memory Modules (DIMMs) - The dual inline memory module or DIMM is a newer memory module, intended for use in fifth- and sixth-generation computer systems. DIMMs are 168 pins in size, and provide memory 64 bits in width. They are a newer form factor and are becoming the de facto standard for new PCs; they are not used on older motherboards. They are also not generally available in smaller sizes such as 1 MB or 4 MB for the simple reason that newer machines are rarely configured with such small amounts of system RAM.

Physically, DIMMs differ from SIMMs in an important way. SIMMs have contacts on either side of the circuit board but they are tied together. So a 30-pin SIMM has 30 contacts on each side of the circuit board, but each pair is connected. This gives some redundancy and allows for more forgiving connections since each pin has two pads. This is also true of 72-pin SIMMs. DIMMs however have different connections on each side of the circuit board. So a 168-pin DIMM has 83 pads on each side and they are not redundant. This allows the packaging to be made smaller, but makes DIMMs a bit more sensitive to correct insertion and good electrical contact.

DIMMs are inserted into special sockets on the motherboard, similar to those used for SIMMs. They are generally available in 8 MB, 16 MB, 32 MB and 64 MB sizes, with larger DIMMs also available at a higher cost per megabyte. DIMMs are the memory format of choice for the newest memory technology, SDRAM. DIMMs are also used for EDO and other technologies as well.

DIMMs come in different flavors, and it is important to ensure that you get the right kind for the machine that you are using. They come in two different voltages: 3.3V and 5.0V, and they come in either buffered or unbuffered versions. This yields of course a total of four different combinations. The standard today is the 3.3 volt unbuffered DIMM, and most machines will use these. Consult your motherboard or system manual.

A smaller version of the DIMM is also sometimes seen; called the small outline DIMM or SODIMM, these packages are used primarily in laptop computers where miniaturization is key.

The Contents from M. Morris Mano

THE MEMORY UNIT

The registers in a digital computer may be classified as either operational or storage type. An *operational* register is capable of storing binary information in its flip-flops and, in addition, has combinational gates capable of data-processing tasks. A *storage* register is used solely for temporary storage of binary information. This information cannot be altered when transferred in and out of the register. A *memory unit is* a collection of storage registers together with the associated circuits needed to transfer information in and out of the registers. The storage registers in a memory unit are called *memory registers.*

The bulk of the registers in a digital computer are memory registers, to which information is transferred for storage and from which information is available when needed for processing. Comparatively few operational registers are found in the processor unit. When data processing takes place, the information from selected registers in the memory unit is first transferred to the operational registers n the processor unit. Intermediate and final results obtained in the operational registers are transferred back to selected memory registers. Similarly, binary information received from input devices is first stored in memory registers; information transferred to output devices is taken from registers in the memory unit.

The component that forms the binary cells of registers in a memory unit must nave certain basic properties, the most important of which are: (1) It must have a reliable two-state property for binary representation. (2) It must be small in size. 3) The cost per bit of storage should be as low as possible. (4) The time of access of a memory register should be reasonably fast. Examples of memory unit components are magnetic cores, semiconductor ICs, and magnetic surfaces on tapes, drums, or disks.

A memory unit stores binary information in groups called *words,* each word being stored in a memory register. A word in memory is an entity of n bits that moves in and out of storage as a unit. A memory word may represent an operand, an instruction, a group of alphanumeric characters, or any binary-coded information. The communication between a memory unit and its environment is achieved through two control signals and two external registers. The control signals specify the direction of transfer required, that is, whether a word is to be stored in a memory register or whether a word previously stored is to be transferred out of a memory register. One external register specifies the particular memory register chosen out of the thousands available; the other specifies the particular bit configuration of the word in question. The control signals and the registers are shown in the block diagram of Fig. 7-24.

The memory *address register* specifies the memory word selected. Each word in memory is assigned a number identification starting from 0 up to the maximum number of words available. To communicate with a specific memory word, its location number, or *address,* is transferred to the address register. The internal circuits of the memory unit accept this address from the register and open the paths needed to select the word called. An address register with n bits can specify up to 2" memory words. Computer memory units can range from 1024 words, requiring an address register of 10 bits, to 1,048,576 = 2^{20} words, requiring a 20-bit address register.

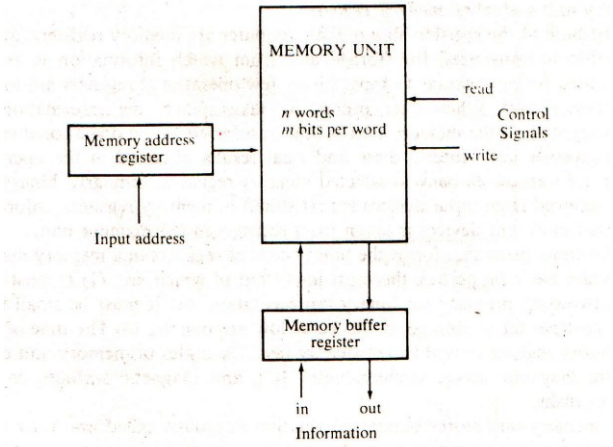


Figure Block diagram of a memory unit showing communication with environment

The two control signals applied to the memory unit are called *read* and *write.* A write signal specifies a transfer-in function; a read signal specifies a transfer-out function. Each is referenced from the memory unit. Upon accepting one of the control signals, the internal control circuits inside the memory unit provide the desired function. Certain types of storage units, because of their component characteristics, destroy the information stored in a cell when the bit in that cell is read out. Such a unit is said to be a destructive read-out memory, as opposed to a nondestructive memory where the information remains in the cell after it is read out. In either case, the old information is always destroyed when new information is written. The sequence of internal control in a destructive read-out memory must provide control signals that will cause the word to be restored into its binary cells it the application calls for a nondestructive function.

The information transfer to and from registers in memory and the external environment is communicated through one common register called the memory *buffer register* (other names are information *register* and *storage register).* When the memory unit receives a *write* control signal, the internal control interprets the contents of the buffer register to be the bit configuration of the word to be stored in a memory register. With *a read* control signal, the internal control sends the word from a memory register into the buffer register. In each case the contents of the address register specify the particular memory register referenced for writing or reading.

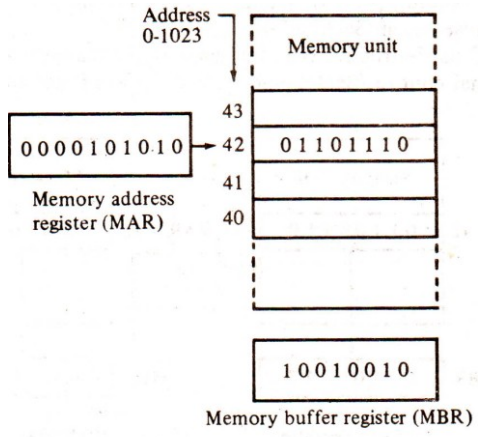


Figure Initial values of registers

Let us summarize the information transfer characteristics of a memory unit by an example. Consider a memory unit of 1024 words with eight bits per word. To specify 1024 words, we need an address of ten bits, since $2^{10} = 1024$. Therefore, the address register must contain ten flip-flops. The buffer register must have eight flip-flops to store the contents of words transferred into and out of memory. The memory unit has 1024 registers with assigned address numbers from 0 to 1023.

Figure 7-25 shows the initial contents of three registers: memory address register (MAR), memory buffer register (MBR), and the memory register addressed by MAR. Since the equivalent binary number in MAR is decimal 42, the memory register addressed by MAR is the one with address number 42.

The sequence of operations needed to communicate with the memory unit for the purpose of transferring a word out to the MBR is:

1. Transfer the address bits of the selected word into MAR. 2. Activate the *read* control input.

The result of the read operation is depicted in the following Fig (a). The binary information presently stored in memory register 42 is transferred into MBR. The sequence of operations needed to store a new word into memory is:

1. Transfer the address bits of the selected word into MAR.
2. Transfer the data bits of the word into MBR
3. Activate the *write* control input.

The result of the write operation is depicted in the following Fig (b). The data bits from MBR are stored in memory register 42.

In the above example, we assumed a memory unit with nondestructive read-out property. Such memories can be constructed with semiconductor ICs. They retain the information in the memory register when the register is sampled during the reading process so that no loss of information occurs. Another component commonly used in memory units is the magnetic core. A magnetic core characteristically has destructive read-out, i.e.., it loses the stored binary information during the reading process. Examples of semiconductor and magnetic-core memories are presented in Section 7-8.

Because of its destructive read-out property, a magnetic-core memory must provide additional control functions to restore the word into the memory register.

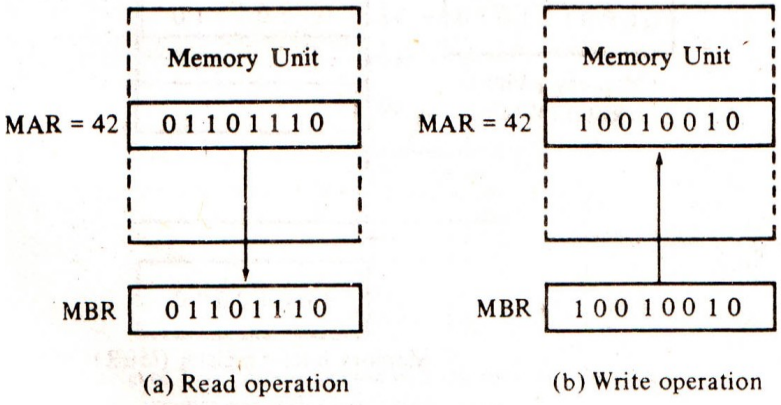


Figure Information transfer during read and write operations

A read control signal applied to a magnetic-core memory transfers the content of the addressed word into an external register and, at the same time, the memory register is automatically cleared. The sequence of internal control in a magnetic-core memory then provides appropriate signals to cause the restoration of the word into the memory register. The information transfer in a magnetic-core memory during a read operation is depicted in the following figure.

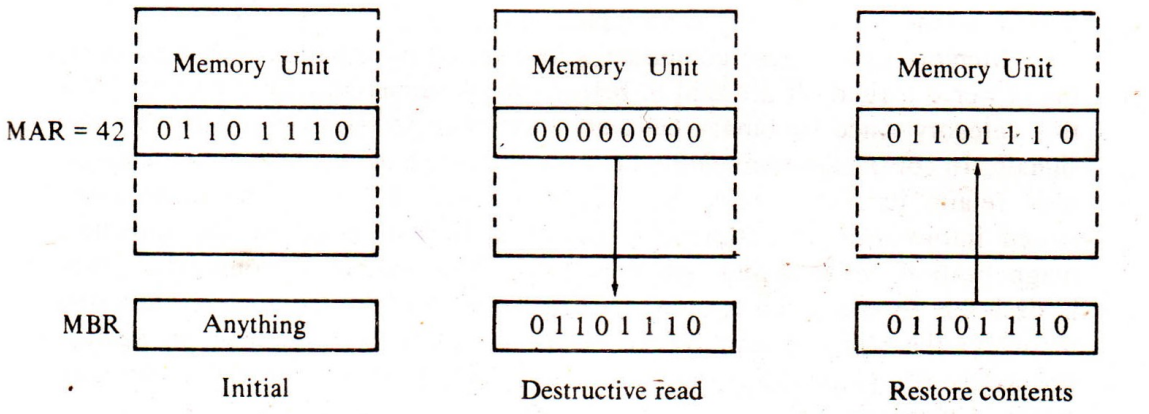


Figure Information transfer in a magnetic-core memory during a read operation

A destructive read operation transfers the selected word into MBR but leaves the memory register with all 0's. Normal memory operation requires that the content of the selected word remain in memory after a read operation. Therefore, it is necessary to go through *a restore* operation that writes the value in MBR into the selected memory register. During the restore operation, the contents of MAR and MBR must remain unchanged.

A write control input applied to a magnetic-core memory causes a transfer of information as depicted in following figure.

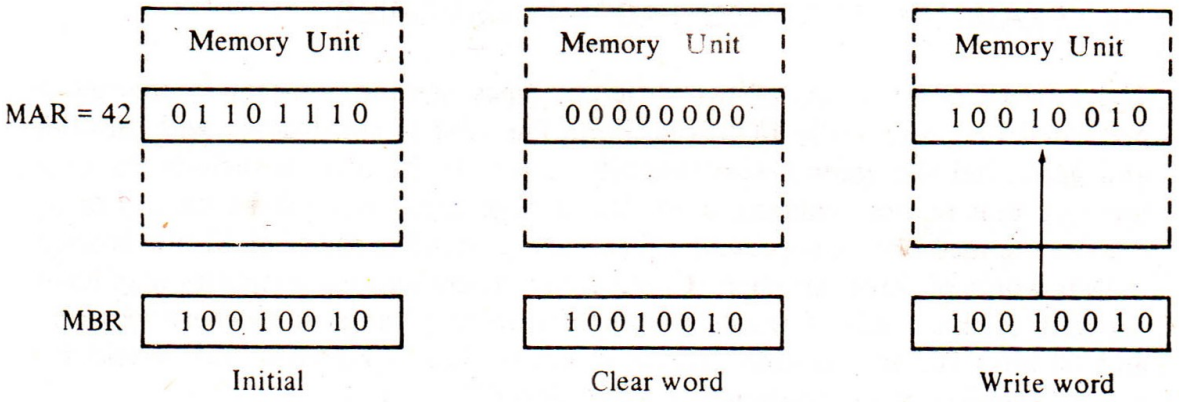


Figure Information transfer in a magnetic-core memory during a write operation

To transfer new information into a selected register, the old information must first be erased by clearing all the bits of the word to 0. After this is done, the content of MBR can be transferred to the selected word. MAR must not change during the operation to ensure that the same selected word that is cleared is the one that receives the new information.

A magnetic-core memory requires two half-cycles either for reading or writing. The time it takes for the memory to go through both half-cycles is called the *memory-cycle* time.

The mode of access of a memory system is determined by the type of components used. In *a random-access* memory, the registers may be thought of as being separated in space, with each register occupying one particular spatial location as in a magnetic-core memory. In *a sequential-access* memory, the information stored in some medium is not immediately accessible but is available only at certain intervals of time. A magnetic-tape unit is of this type. Each memory location passes the read and write heads in turn, but information is read out only when the requested word has been reached. The *access time* of a memory is the time required to select a word and either read or write it. In a random-access memory, the access time is always the same regardless of the word's particular location in space. In a sequential memory, the access time depends on the position of the word at the time of request. If the word is just emerging from storage at the time it is requested, the access time is just the time necessary to read or write it. But, if the word happens to be in the last position, the access time also includes the time required for all the other words to move past the terminals. Thus, the access time in a sequential memory is variable.

Memory units whose components lose stored information with time or when the power is turned off are said to be *volatile.* A semiconductor memory unit is of this category since its binary cells need external power to maintain the needed signals. In contrast, a nonvolatile memory unit, such as magnetic core or magnetic disk, retains its stored

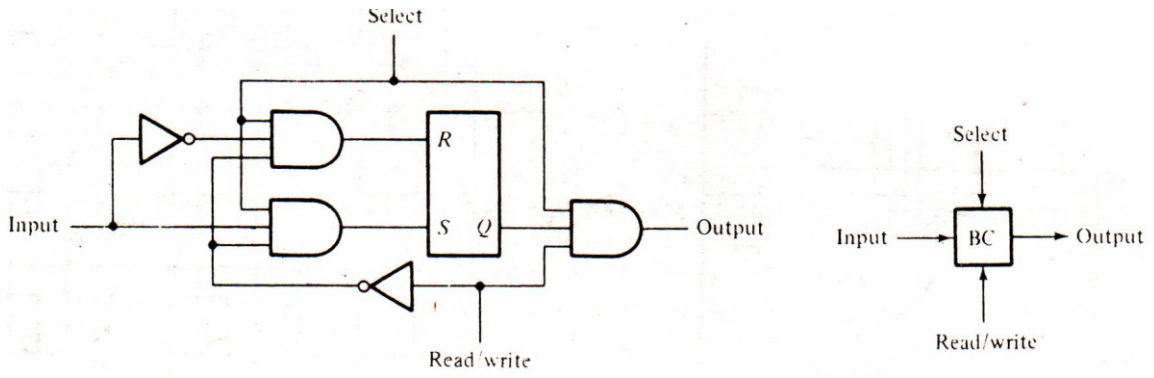
information after removal of power. This is because the stored information in magnetic components is manifested by the direction of magnetization, which is retained when power is turned off. A nonvolatile property is desirable in digital computers because many useful programs are left permanently in the memory unit. When power is turned off and then on again, the previously stored programs and other information are not lost but continue to reside in memory.

EXAMPLES OF RANDOM-ACCESS MEMORIES

The internal construction of two different types of random-access memories are presented diagramatically in this section. The first is constructed with flip-flops and gates and the second with magnetic cores. To be able to include the entire memory unit in one diagram, a limited storage capacity must be used. For this reason, the memory units presented here have a small capacity of 12 bits arranged in four words of three bits each. Commercial random-access memories may have a capacity of thousands of words and each word may range somewhere between 8 and 64 bits. The logical construction of large-capacity memory units would be a direct extension of the configuration shown here.

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 the associated logic for selecting individual words. The binary storage cell is the basic building block of a memory unit. The equivalent logic of a binary cell that stores one bit of information is shown in Fig. 7-29. Although the cell is shown to include gates and a flip-flop, internally it is constructed with two transistors having multiple inputs. A binary storage cell must be very small in order to be able to pack as many cells as possible in the small area available in the integrated-circuit chip. The binary cell has three inputs and one output. The select input enables the cell for reading or writing. The read/write input determines the cell operation when it is selected. A 1 in the read/write input forms a path from the flip-flop to the output terminal. The information in the input terminal is transferred into the flip-flop when the read/write control is 0. Note that the flip-flop operates without clock pulses and that its purpose is to store the information bit in the binary cell.

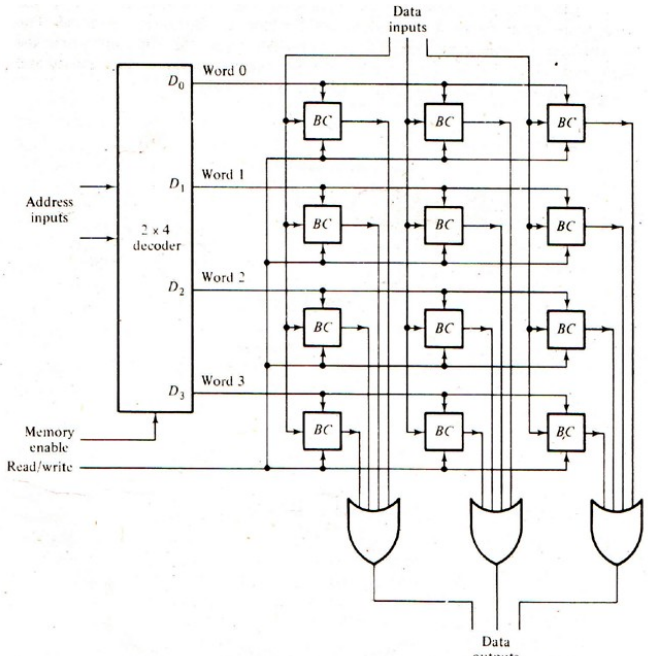
Integrated-circuit memories sometimes have a single line for the read and write control. One binary state in the single line specifies a read operation and the other state specifies a write operation. In addition, one or more enable lines are included to provide means for selecting the IC and for expanding several packages into a memory unit with a larger number of words. The logical construction of a IC RAM is shown in Fig. 7-30. It consists of 4 words of 3 bits each, for a total of 12 binary cells. The small boxes labeled BC represent a binary cell, and the three inputs and one output in each BC are as specified in the diagram of following figure.



(a) Logic diagram

(b) Block diagram

The two address input lines go through an internal 2-to-4 line decoder. The decoder is enabled with the memory-enable input. When the memory enable is 0, all the outputs of the decoder are 0 and none of the memory words are selected. With the memory enable at 1, one of the four words is selected, depending on the value of the two address lines. Now, with the read/write control at 1, the bits of the selected word go through the three OR gates to the output terminals. The nonselected binary cells produce 0's in the inputs of the OR gates and have no effect on the outputs. With the read/write control at 0, the information available on the input lines is transferred into the binary cells of the selected word. The nonselected binary cells in the other words are disabled by their selection inputs and their previous values remain unchanged. With the memory-enable control at 0, the contents of all cells in the memory remain unchanged, regardless of the value of the read/write control.



IC RAMs are constructed internally with cells having a wired-OR capability. This eliminates the need for the OR gates in the diagram. The external output lines can also form wired logic to facilitate the connection of two or more IC packages to form a memory unit with a larger number of words.

Magnetic-core Memory

A magnetic-core memory uses magnetic cores to store binary information. A magnetic core is a doughnut-shaped toroid made of magnetic material. In contrast to a semiconductor flip-flop that needs only one physical quantity such as voltage for its operation, a magnetic core employs three physical quantities: current, magnetic flux, and voltage. The signal that excites the core is *a current* pulse in a wire passing through the core. The binary information stored is represented by the direction of *magnetic flux* within the core. The output binary information is extracted from a wire linking the core in the form of a *voltage* pulse.

The physical property that makes a magnetic core suitable for binary storage is its hysteresis loop, shown in following figure (c). This is a plot of current vs. magnetic flux, and it has the shape of a square loop. With zero current, a flux which is either positive (counterclockwise direction) or negative (clockwise direction) remains in the magnetized core. One direction, say counterclockwise magnetization, is used to represent a 1 and the other to represent a 0.

A pulse of current applied to the winding through the core can shift the direction of magnetization. As shown in following figure (a), current in the downward direction produces flux in the clockwise direction, causing the core to go to the 0 state. Following figure (b) shows the current and flux directions for storing a 1. The path that the flux takes when the current pulse is applied is indicated by arrows in the hysteresis loop.

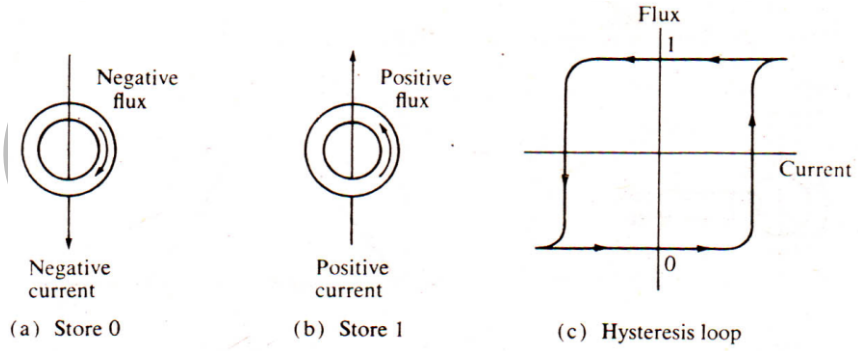


Figure Storing a bit into a magnetic core

Reading out the binary information stored in the core is complicated by the fact that flux cannot be detected when it is not changing. However, if flux is changing with respect to time, it induces a voltage in a wire that links the core. Thus, read-out could be accomplished by applying a current in the negative direction as shown in following figure. If the core is in the 1 state, the current reverses the direction of magnetization, and the resulting change of flux produces a voltage pulse in the sense wire. If the core is already in the 0 state, the negative current leaves the core magnetized in the same direction, causing a very slight disturbance of magnetic flux which results in a very small output voltage in the sense wire. Note that this is a destructive read-out, since the read current always returns the core to the 0 state. The previously stored value is lost.

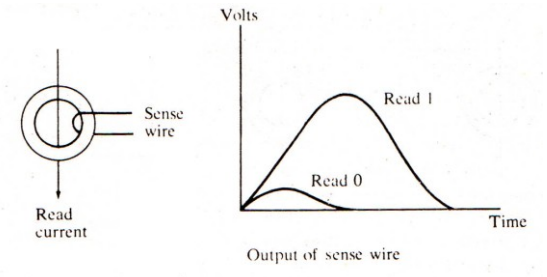


Figure Reading a bit from a magnetic core

The figure given below shows the organization of a magnetic-core memory containing four words with three bits each. Comparing it with the figure of IC memory unit, we note that the binary cell now is a magnetic core and the wires linking it. The excitation of the core is accomplished by means of a current pulse generated in a driver (DR). The output information goes through a sense amplifier (SA) whose outputs set corresponding flip-flops in the buffer register. Three wires link each core. The word wire is excited by a word driver and goes through the three cores of a word. A bit wire is excited by a bit driver and goes through four cores in the same bit position. The sense wire links the same cores as the bit wire and is applied to a sense amplifier that shapes the voltage pulse when a I is read and rejects the small disturbance when a 0 is read.

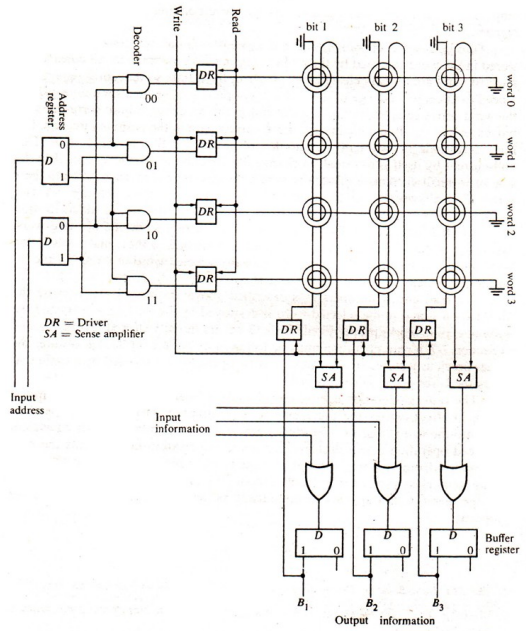


Figure Magnetic-core memory unit

During a read operation, a word-driver current pulse is applied to the cores of the word selected by the decoder. The read current is in the negative direction (Fig. 7-32) and causes all cores of the selected word to go to the 0 state, regardless of their previous state. Cores which previously contained a 1 switch their flux and induce a voltage into their sense wire. The flux of cores which already contained a 0 is not changed. The voltage pulse on a sense wire of cores with a previous I is amplified in the sense amplifier and sets the corresponding flip-flop in the buffer register.

During a write operation, the buffer register holds the information to be stored in the word specified by the address register. We assume that all cores in the selected word are initially cleared, i.e.., all are in the 0 state so that cores requiring a I need to undergo a change of state. A current pulse is generated simultaneously in the word driver selected by the decoder and in the bit driver, whose corresponding buffer register flip-flop contains a 1. Both currents are in the positive direction, but their magnitude is only half that needed to switch the flux to the 1 state. This half-current by itself is too small to change the direction of magnetization. But the sum of two half-currents is enough to switch the direction of magnetization to the 1 state. A core switches to the 1 state only if there is a coincidence of two half-currents from a word driver and a bit driver. The direction of magnetization of a core does not change if it receives only half-current from one of the drivers. The result is that the magnetization of cores is switched to the I state only if the' word and bit wires intersect, that is, only in the selected word and only in the bit position in which the buffer register is a 1.

The read and write operations described above are incomplete, because the information stored in the selected word is destroyed by the reading process and the write operation works properly only if the cores are initially cleared. As mentioned in Section 7-7, a read operation must be followed by another cycle that restores the values previously stored in the cores. A write operation is preceded by a cycle that clears the cores of the selected word.

The restore operation during a read cycle is equivalent to a write operation which, in effect, writes the previously read information from the buffer register back into the word selected. The clear operation during a write cycle is equivalent to a read operation which destroys the stored information but prevents the read information from reaching the buffer register by inhibiting the sense amplifier. Restore and clear cycles are normally initiated by the memory internal control, so that the memory unit appears to the outside as having a nondestructive read-out property.