

F.Y.B.Sc.(IT)
Syllabus : Fundamentals of Digital Computing

Unit – I	<p>Data and Information</p> <p>Features of Digital Systems, Number Systems-Decimal, Binary, Octal, Hexadecimal and their inter conversions, Representation of Data: Signed Magnitude, one's complement and two's complement, Binary Arithmetic, Fixed point representation and Floating point representation of numbers.</p> <p>Codes</p> <p>BCD, XS-3, Gray code, hamming code, alphanumeric codes (ASCII, EBCDIC, UNICODE), Error detecting and error correcting codes.</p>
Unit- II	<p>Boolean Algebra:</p> <p>Basic gates (AND, OR, NOT gates), Universal gates (NAND and NOR gates), other gates (XOR, XNOR gates). Boolean identities, De Morgan Laws.</p> <p>Karnaugh maps:</p> <p>SOP and POS forms, Quine McClusky method.</p>
Unit -III	<p>Combinational Circuits:</p> <p>Half adder, full adder, code converters, combinational circuit design, Multiplexers and demultiplexers, encoders, decoders, Combinational design using mux and demux.</p>
Unit - IV	<p>Sequential Circuit Design:</p> <p>Flip flops (RS, Clocked RS, D, JK, JK Master Slave, T, Counters, Shift registers and their types, Counters: Synchronous and Asynchronous counters.</p>
Unit- V	<p>Computers:</p> <p>Basic Organisation, Memory: ROM, RAM, PROM, EPROM, EEPROM, Secondary Memory: Hard Disk and optical Disk, Cache Memory, I/O devices</p>
Unit -VI	<p>Operating Systems:</p> <p>Types (real Time, Single User / Single Tasking, Single user / Multi tasking, Multi user / Multi tasking, GUI based OS. Overview of desktop operating systems- Windows and LINUX.</p>

Text Books:

- Modern Digital Electronics by R. P. Jain, 3rd Edition, McGraw Hill
- Digital Design and Computer Organisation by Dr. N. S. Gill and J. B. Dixit, University Science Press
- Linux Commands by Bryan Pfaffaenberger BPB Publications
- UNIX by Sumitabha Das, TMH
- References:
- Digital Principles and Applications by Malvino and Leach, McGrawHill
- Introduction to Computers by Peter Norton, McGraw Hill
- Introduction to Computers by Balagurusamy



DATA AND INFORMATION

Unit Structure

- 1.0 Objectives
- 1.1 Data & Information
- 1.2 Analog Versus Digital
- 1.3 Number Systems
- 1.4 Decimal Verses Binary Number System
- 1.5 Octal & Hexadecimal Number System
- 1.6 Conversion from Decimal Number System
- 1.7 Unsigned & Signed Integers
- 1.8 Signed Integers
- 1.9 1's Complement
- 1.10 2's Complement
- 1.11 Binary Arithmetic
 - 1.11.1 Addition
 - 1.11.2 Subtraction
 - 1.11.3 Multiplication
 - 1.11.4 Division
 - 1.11.5 Binary Subtraction Using 1's Complement
 - 1.11.6 Binary Subtraction Using 2's Complement
- 1.12 Questions
- 1.13 Further Reading

1.0 OBJECTIVES

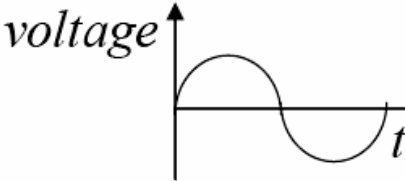
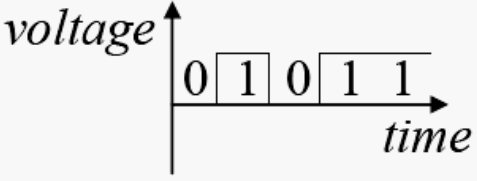
After completing this chapter, you will be able to:

- ❖ Understand the concept of Data and Information.
- ❖ Differentiate between the Analog Verses digital Signals.
- ❖ Deal with the different number system in arithmetic.
- ❖ Understand the number system conversions.
- ❖ Solve the Arithmetic examples based on Binary arithmetic.

1.1 DATA AND INFORMATION

Data is raw collection of samples. When this raw data is processed it becomes information. Hence information is processed data. Information is processed data.

1.2 ANALOG VERSUS DIGITAL

Analog Systems	Digital Systems
	
Continuous time-varying voltages and/or Currents	Discrete signals sampled in time
All possible values are present.	<ul style="list-style-type: none"> – Two possible values • 0V, low, false (logic 0) • 5V, high, true (logic 1)
<ul style="list-style-type: none"> – Basic elements of analog circuits: <ul style="list-style-type: none"> • Resistors • Capacitors • Inductors • Transistors 	<ul style="list-style-type: none"> – Basic elements of digital circuits: <ul style="list-style-type: none"> • Logic gates: AND, OR, NOT

Advantages of Digital Systems

- Reproducible results
- Relative ease of design
- Flexibility and functionality
- High speed
- Small size
- Low cost
- Low power
- Steadily advancing technology
- Programmable logic devices

Digital techniques and systems have the advantages of being relatively much easier to design and having higher accuracy, programmability, noise immunity, easier storage of data and ease of fabrication in integrated circuit form, leading to availability of more complex functions in a smaller size. The real world, however, is analogue. Most physical quantities – position, velocity, acceleration, force, pressure, temperature and flowrate, for

example – are analogue in nature. That is why analog variables representing these quantities need to be digitized or discretized at the input if we want to benefit from the features and facilities that come with the use of digital techniques. In a typical system dealing with analog inputs and outputs, analog variables are digitized at the input with the help of an analog-to-digital converter block and reconverted back to analogue form at the output using a digital-to-analog converter block.

1.3 NUMBER SYSTEMS

The expression of numerical quantities is something we tend to take for granted. This is both a good and a bad thing in the study of electronics. It is good, in that we are accustomed to the use and manipulation of numbers for the many calculations used in analyzing electronic circuits. On the other hand, the particular system of notation we have been taught from primary school onwards is *not* the system used internally in modern electronic computing devices and learning any different system of notation requires some re-examination of deeply ingrained assumptions.

First, we have to distinguish the difference between numbers and the symbols we use to represent numbers. 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: $3 - j4$, $34.5 \angle 20^\circ$

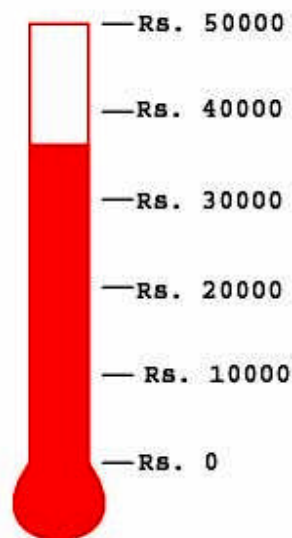
Different types of numbers find different application in the physical world. Whole numbers work well for counting discrete objects, such as the number of resistors in a circuit. Integers are needed when negative equivalents of whole numbers are required.

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.

We are familiar with an analog representation of money, and didn't realize it for what it was. Have you ever seen a fund-raising poster made with a picture of a thermometer on it, where the height of the red column indicated the amount of money collected for the cause? The more money collected, the taller the column of red ink on the poster.

*An analog representation
of a numerical quantity*

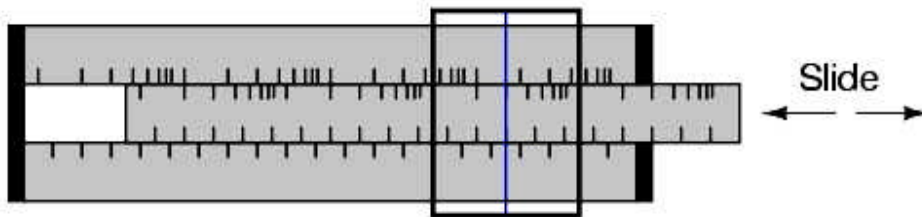


This is an example of an analog representation of a number. There is no real limit to how finely divided the height of that column can be made to symbolize the amount of money in the account. Changing the height of that column is something that can be done without changing the essential nature of what 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:
Rs. 35,955.38

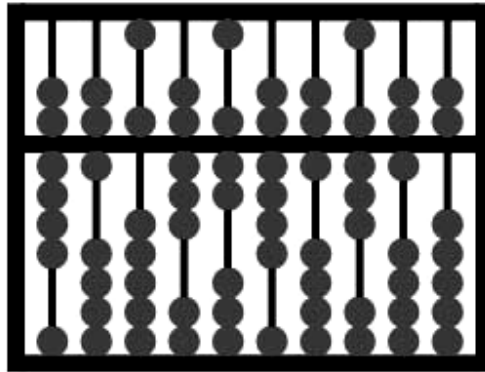
Unlike the "thermometer" poster with its red column, those symbolic characters above cannot be finely divided: that particular combination of ciphers stand for one quantity and one quantity only. If more money is added to the account (+ \$40.12), different symbols must be used to represent the new balance (\$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.

Lets contrast these two methods of numerical representation:

ANALOG

DIGITAL

Intuitively understood ----- Requires training to interpret

Infinitely divisible ----- Discrete

Prone to errors of precision ----- Absolute precision

Interpretation of numerical symbols is something we tend to take for granted, because it has been taught to us for many years. However, if you were to try to communicate a quantity of something to a person ignorant of decimal numerals, that person could still understand the simple thermometer chart!

The infinitely divisible vs. discrete and precision comparisons are really flip-sides of the same coin. The fact that 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).

Please don't misunderstand this difference in precision by thinking that digital representation is necessarily more *accurate* than analog. Just because a clock is digital doesn't mean that it will always read time more accurately than an analog clock, it just means that the *interpretation* of its display is less ambiguous.


Divisibility of analog versus digital representation can be further illuminated by talking about the representation of irrational numbers. Numbers such as π are called irrational, because they cannot be exactly expressed as the fraction of integers, or whole numbers. Although you might have learned in the past that the fraction $22/7$ can be used for π in calculations, this is just an approximation. The actual number "pi" cannot be exactly expressed by any finite, or limited, number of decimal places. The digits of π go on forever:
 3.1415926535897932384

It is possible, at least theoretically, to set a slide rule (or even a thermometer column) so as to perfectly represent the number π , because analog symbols have no minimum limit to the degree that they can be increased or decreased. If my slide rule shows a figure of 3.141593 instead of 3.141592654, I can bump the slide just a bit more (or less) to get it closer yet. However, with digital representation, such as with an abacus, I would need additional rods (place holders, or digits) to represent π to further degrees of precision. An abacus with 10 rods simply cannot represent any more than 10 digits worth of the number π , no matter how I set the beads. To perfectly represent π , an abacus would have to have an infinite number of beads and rods! The tradeoff, of course, is the practical limitation to adjusting, and reading, analog symbols. Practically speaking, one cannot read a slide rule's scale to the 10th digit of precision, because the marks on the scale are too coarse and human vision is too limited. An abacus, on the other hand, can be set and read with no interpretational errors at all.

Furthermore, analog symbols require some kind of standard by which they can be compared for precise interpretation. Slide rules have markings printed along the length of the slides to translate length into standard quantities. Even the thermometer chart has numerals written along its height to show how much money (in Rupees) the red column represents for any given amount of height. Imagine if we all tried to communicate simple numbers to each other by spacing our hands apart varying distances. The number 1 might be signified by holding our hands 1 inch apart, the number 2 with 2 inches, and so on. If someone held their hands 17 inches apart to represent the number 17, would everyone around them be able to immediately and accurately interpret that distance as 17? Probably not. Some would guess short (15 or 16) and some would guess long (18 or 19). Of course, fishermen who brag about their catches don't mind overestimations in quantity!

Perhaps this is why people have generally settled upon digital symbols for representing numbers, especially whole numbers and integers, which find the most application in everyday life. Using the fingers on our hands, we have a ready means of

symbolizing integers from 0 to 10. We can make tally marks on paper, wood, or stone to represent the same quantities quite easily:

$$5 + 5 + 3 = 13$$


For large numbers, though, the "tally mark" number system is too inefficient.

1.4 DECIMAL VERSUS BINARY NUMBER SYSTEMS

Let's count from zero to twenty using four different kinds of number systems: Roman numerals, decimal and binary:

Decimal	Roman	Binary	Octal	Hexadecimal
0	-	0	0	0
1	I	1	1	1
2	II	01	2	2
3	III	11	3	3
4	IV	100	4	4
5	V	101	5	5
6	VI	110	6	6
7	VII	111	7	7
8	VIII	1000	10	8
9	IX	1001	11	9
10	X	1010	12	A
11	XI	1011	13	B
12	XII	1100	14	C
13	XIII	1101	15	D
14	XIV	1110	16	E
15	XV	1110	17	F

The Roman system is not very practical for symbolizing large numbers. Obviously, place-weighted systems such as decimal and binary are more efficient for the task. Notice, though, how much shorter decimal notation is over binary notation, for the same

number of quantities. What takes five bits in binary notation only takes two digits in decimal notation.

This raises an interesting question regarding different number systems: how large of a number can be represented with a limited number of cipher positions, or places? With the crude hash-mark system, the number of places IS the largest number that can be represented, since one hash mark "place" is required for every integer step. For place-weighted systems of numeration, however, the answer is found by taking base of the number system (10 for decimal, 2 for binary) and raising it to the power of the number of places. For example, 5 digits in a decimal numeration system can represent 100,000 different integer number values, from 0 to 99,999 (10 to the 5th power = 100,000). 8 bits in a binary numeration system can represent 256 different integer number values, from 0 to 11111111 (binary), or 0 to 255 (decimal), because 2 to the 8th power equals 256. With each additional place position to the number field, the capacity for representing numbers increases by a factor of the base (10 for decimal, 2 for binary).

An interesting footnote for this topic is the one of the first electronic digital computers, the Eniac. The designers of the Eniac chose to represent numbers in decimal form, digitally, using a series of circuits called "ring counters" instead of just going with the binary numeration system, in an effort to minimize the number of circuits required to represent and calculate very large numbers. This approach turned out to be counter-productive, and virtually all digital computers since then have been purely binary in design.

To convert a number in binary numeration to its equivalent in decimal form, all you have to do is calculate the sum of all the products of bits with their respective place-weight constants. To illustrate:

Convert 11001101_2 to decimal form:

bits =	1	1	0	0	1	1	0	1
	.		-	-	-	-	-	-
weight =	1	6	3	1	8	4	2	1
(in decimal notation)	2	4	2	6				8

The bit on the far right side is called the Least Significant Bit (LSB), because it stands in the place of the lowest weight (the one's place). The bit on the far left side is called the Most Significant Bit (MSB), because it stands in the place of the highest weight (the one hundred twenty-eight's place). Remember, a bit value of "1" means that the respective place weight gets added to the total value, and a bit value of "0" means that the respective place weight does *not* get added to the total value. With the above example, we have:

$$128_{10} + 64_{10} + 8_{10} + 4_{10} + 1_{10} = 205_{10}$$

Number	Decimal	Binary	Octal	Hexadecimal
Zero	0	0	0	0
One	1	1	1	1
Two	2	10	2	2
Three	3	11	3	3
Four	4	100	4	4
Five	5	101	5	5
Six	6	110	6	6
Seven	7	111	7	7
Eight	8	1000	10	8
Nine	9	1001	11	9
Ten	10	1010	12	A
Eleven	11	1011	13	B
Twelve	12	1100	14	C
Thirteen	13	1101	15	D
Fourteen	14	1110	16	E
Fifteen	15	1111	17	F
Sixteen	16	10000	20	10
Seventeen	17	10001	21	11
Eighteen	18	10010	22	12
Nineteen	19	10011	23	13
Twenty	20	10100	24	14

Octal and hexadecimal number systems would be pointless if not for their ability to be easily converted to and from binary notation. Their primary purpose in being is to serve as a "shorthand" method of denoting a number represented electronically in binary form. Because the bases of octal (eight) and hexadecimal (sixteen) are even multiples of binary's base (two), binary bits can be grouped together and directly converted to or from their respective octal or hexadecimal digits. With octal, the binary bits are grouped in three's (because $2^3 = 8$), and with hexadecimal, the binary bits are grouped in four's (because $2^4 = 16$):

BINARY TO OCTAL CONVERSION

Convert 10110111.1_2 to octal:

```

..                implied zero  implied zeros
.                |                ||
.                010  110  111  100
.
Convert each group of bits --- --- --- . ---
to its octal equivalent:   2   6   7   4
.

```

Answer: $10110111.1_2 = 267.4_8$

We had to group the bits in three's, from the binary point left, and from the binary point right, adding (implied) zeros as necessary to make complete 3-bit groups. Each octal digit was translated from the 3-bit binary groups. Binary-to-Hexadecimal conversion is much the same:

BINARY TO HEXADECIMAL CONVERSION

Convert 10110111.1_2 to hexadecimal:

```

.
.
.           implied zeros
.           |||
.           1011 0111 1000
.
Convert each group of bits  ----  ----  .  ----
to its hexadecimal equivalent:  B   7   8
.

```

Answer: $10110111.1_2 = B7.8_{16}$

Here we had to group the bits in four's, from the binary point left, and from the binary point right, adding (implied) zeros as necessary to make complete 4-bit groups:

Likewise, the conversion from either octal or hexadecimal to binary is done by taking each octal or hexadecimal digit and converting it to its equivalent binary (3 or 4 bit) group, then putting all the binary bit groups together.

Incidentally, hexadecimal notation is more popular, because binary bit groupings in digital equipment are commonly multiples of eight (8, 16, 32, 64, and 128 bit), which are also multiples of 4. Octal, being based on binary bit groups of 3, doesn't work out evenly with those common bit group sizings.

1.6 CONVERSION FROM DECIMAL NUMBER SYSTEM

Because octal and hexadecimal number systems have bases that are multiples of binary (base 2), conversion back and forth between either hexadecimal or octal and binary is very easy. Also, because we are so familiar with the decimal system, converting binary, octal, or hexadecimal to decimal form is relatively easy (simply add up the products of cipher values and place-weights). However, conversion from decimal to any of these "strange" number systems is a different matter.

The method which will probably make the most sense is the "trial-and-fit" method, where you try to "fit" the binary, octal, or hexadecimal notation to the desired value as represented in decimal form. For example, let's say that I wanted to represent the decimal value of 87 in binary form. Let's start by drawing a binary number field, complete with place-weight values:

```

.
.   - - - - -
.

```

weight = 1 6 3 1 8 4 2 1
 (in decimal 2 4 2 6
 notation) 8

Well, we know that we won't have a "1" bit in the 128's place, because that would immediately give us a value greater than 87. However, since the next weight to the right (64) is less than 87, we know that we must have a "1" there.

. 1
 . - - - - - Decimal value so far = 64₁₀

weight = 6 3 1 8 4 2 1
 (in decimal 4 2 6
 notation)

If we were to make the next place to the right a "1" as well, our total value would be 64₁₀ + 32₁₀, or 96₁₀. This is greater than 87₁₀, so we know that this bit must be a "0". If we make the next (16's) place bit equal to "1," this brings our total value to 64₁₀ + 16₁₀, or 80₁₀, which is closer to our desired value (87₁₀) without exceeding it:

. 1 0 1
 . - - - - - Decimal value so far = 80₁₀

weight = 6 3 1 8 4 2 1
 (in decimal 4 2 6
 notation)

By continuing in this progression, setting each lesser-weight bit as we need to come up to our desired total value without exceeding it, we will eventually arrive at the correct figure:

. 1 0 1 0 1 1 1
 . - - - - - Decimal value so far = 87₁₀

weight = 6 3 1 8 4 2 1
 (in decimal 4 2 6
 notation)

This trial-and-fit strategy will work with octal and hexadecimal conversions, too. Let's take the same decimal figure, 87₁₀, and convert it to octal number system:

.
 . - - -
 weight = 6 8 1
 (in decimal 4
 notation)

If we put a cipher of "1" in the 64's place, we would have a total value of 64₁₀ (less than 87₁₀). If we put a cipher of "2" in the 64's place, we would have a total value of 128₁₀ (greater than 87₁₀). This tells us that our octal number system must start with a "1" in the 64's place:

. 1
 . weight = - - - Decimal value so far = 64_{10}
 6 8 1
 (in decimal 4
 notation)

Now, we need to experiment with cipher values in the 8's place to try and get a total (decimal) value as close to 87 as possible without exceeding it. Trying the first few cipher options, we get:

"1" = $64_{10} + 8_{10} = 72_{10}$
 "2" = $64_{10} + 16_{10} = 80_{10}$
 "3" = $64_{10} + 24_{10} = 88_{10}$

A cipher value of "3" in the 8's place would put us over the desired total of 87_{10} , so "2" it is!

. 1 2
 . - - - Decimal value so far = 80_{10}
 weight = 6 8 1
 (in decimal 4
 notation)

Now, all we need to make a total of 87 is a cipher of "7" in the 1's place:

. 1 2 7
 . - - - Decimal value so far = 87_{10}
 weight = 6 8 1
 (in decimal 4
 notation)

Of course, if you were paying attention during the last section on octal/binary conversions, you will realize that we can take the binary representation of (decimal) 87_{10} , which we previously determined to be 1010111_2 , and easily convert from that to octal to check our work:

. Implied zeros
 . ||
 . 001 010 111 Binary
 . --- --- ---
 . 1 2 7 Octal

.
 Answer: $1010111_2 = 127_8$

Can we do decimal-to-hexadecimal conversion the same way? Sure, but who would want to? This method is simple to understand, but laborious to carry out. There is another way to do these conversions, which is essentially the same (mathematically), but easier to accomplish.

This other method uses repeated cycles of division (using decimal notation) to break the decimal number system down into multiples of binary, octal, or hexadecimal place-weight values. In the first cycle of division, we take the original decimal number and divide it by the base of the number system that we're converting to (binary=2, octal=8, hex=16). Then, we take the whole-number portion of division result (quotient) and divide it by the base value again, and so on, until we end up with a quotient of less than 1. The binary, octal, or hexadecimal digits are determined by the "remainders" left over by each division step. Let's see how this works for binary, with the decimal example of 87_{10} :

. 87	Divide 87 by 2, to get a quotient of 43.5
. --- = 43.5	Division "remainder" = 1, or the < 1 portion
. 2	of the quotient times the divisor (0.5×2)
.	
. 43	Take the whole-number portion of 43.5 (43)
. --- = 21.5	and divide it by 2 to get 21.5, or 21 with
. 2	a remainder of 1
.	
. 21	And so on . . . remainder = 1 (0.5×2)
. --- = 10.5	
. 2	
.	
. 10	And so on . . . remainder = 0
. --- = 5.0	
. 2	
.	
. 5	And so on . . . remainder = 1 (0.5×2)
. --- = 2.5	
. 2	
.	
. 2	And so on . . . remainder = 0
. --- = 1.0	
. 2	
.	
. 1	. . . until we get a quotient of less than 1
. --- = 0.5	remainder = 1 (0.5×2)
. 2	

The binary bits are assembled from the remainders of the successive division steps, beginning with the LSB and proceeding to the MSB. In this case, we arrive at a binary notation of 1010111_2 .

When we divide by 2, we will always get a quotient ending with either ".0" or ".5", i.e. a remainder of either 0 or 1. As was said before, this repeat-division technique for conversion will work for number systems other than binary. If we were to perform successive divisions using a different number, such as 8 for conversion to octal, we will necessarily get remainders between 0 and 7. Let's try this with the same decimal number, 87_{10} :

```
. 87          Divide 87 by 8, to get a quotient of 10.875
. --- = 10.875  Division "remainder" = 7, or the < 1 portion
. 8           of the quotient times the divisor (.875 x 8)
.
. 10
. --- = 1.25    Remainder = 2
. 8
.
. 1
. --- = 0.125   Quotient is less than 1, so we'll stop here.
. 8           Remainder = 1
.
. RESULT:  $87_{10} = 127_8$ 
```

We can use a similar technique for converting number systems dealing with quantities less than 1, as well. For converting a decimal number less than 1 into binary, octal, or hexadecimal, we use repeated multiplication, taking the integer portion of the product in each step as the next digit of our converted number. Let's use the decimal number 0.8125_{10} as an example, converting to binary:

```
. 0.8125 x 2 = 1.625  Integer portion of product = 1
.
. 0.625 x 2 = 1.25    Take < 1 portion of product and remultiply
.                    Integer portion of product = 1
.
. 0.25 x 2 = 0.5     Integer portion of product = 0
.
. 0.5 x 2 = 1.0      Integer portion of product = 1
.                    Stop when product is a pure integer
.                    (ends with .0)
.
. RESULT:  $0.8125_{10} = 0.1101_2$ 
```

As with the repeat-division process for integers, each step gives us the next digit (or bit) further away from the "point." With integer (division), we worked from the LSB to the MSB (right-to-left), but with repeated multiplication, we worked from the left to the right. To convert a decimal number greater than 1, with a < 1

component, we must use *both* techniques, one at a time. Take the decimal example of 54.40625_{10} , converting to binary:

REPEATED DIVISION FOR THE INTEGER PORTION:

```

. 54
. --- = 27.0      Remainder = 0
. 2
.
. 27
. --- = 13.5      Remainder = 1 (0.5 x 2)
. 2
.
. 13
. --- = 6.5       Remainder = 1 (0.5 x 2)
. 2
.
. 6
. --- = 3.0       Remainder = 0
. 2
.
. 3
. --- = 1.5       Remainder = 1 (0.5 x 2)
. 2
.
. 1
. --- = 0.5       Remainder = 1 (0.5 x 2)
. 2
.

```

PARTIAL ANSWER: $54_{10} = 110110_2$

REPEATED MULTIPLICATION FOR THE < 1 PORTION:

```

.  $0.40625 \times 2 = 0.8125$  Integer portion of product = 0
.  $0.8125 \times 2 = 1.625$  Integer portion of product = 1
.  $0.625 \times 2 = 1.25$  Integer portion of product = 1
.  $0.25 \times 2 = 0.5$  Integer portion of product = 0
.  $0.5 \times 2 = 1.0$  Integer portion of product = 1
.

```

PARTIAL ANSWER: $0.40625_{10} = 0.01101_2$

COMPLETE ANSWER: $54_{10} + 0.40625_{10} = 54.40625_{10}$
 $110110_2 + 0.01101_2 = 110110.01101_2$

1.7 UNSIGNED AND SIGNED INTEGERS

An integer is a number with no fractional part; it can be positive, negative or zero. In ordinary usage, one uses a minus sign

to designate a negative integer. However, a computer can only store information in bits, which can only have the values zero or one. We might expect, therefore, that the storage of negative integers in a computer might require some special technique. It is for that reason that we began this section with a discussion of unsigned integers.

As you might imagine, an **unsigned integer** is either positive or zero. Consider a single digit decimal number: in a single decimal digit, you can write a number between 0 and 9. In two decimal digits, you can write a number between 0 and 99, and so on. Since nine is equivalent to $10^1 - 1$, 99 is equivalent to $10^2 - 1$, etc., in n decimal digits, you can write a number between 0 and $10^n - 1$. Analogously, in the binary number system, **an unsigned integer containing n bits can have a value between 0 and $2^n - 1$ (which is 2^n different values).**

This fact is one of the most important and useful things to know about computers. When a computer program is written, the programmer, either explicitly or implicitly, must decide how many bits are used to store any given quantity. Once the decision is made to use n bits to store it, the program has an inherent limitation: that quantity can only have a value between 0 and $2^n - 1$. You will meet these limitations in one form or another in every piece of hardware and software that you will learn about during your career:

- the BIOS (Basic Input Output Software) in older PCs uses 10 bits to store the cylinder number on the hard drive where your operating system begins; therefore those PCs cannot boot an operating system from a cylinder greater than $2^{10} - 1$, or 1023.
- a FAT12 file system (used on Windows diskettes), which allocates file space in units called "clusters", uses 12 bits to store cluster numbers; therefore there can be no more than $2^{12} - 1$ or 4,095 clusters in such a file system.
- a UNIX system keeps track of the processes (programs) it runs using a PID (Process IDentifier); for typical memory sizes, the PID is 16 bits long and so after $2^{16} - 1$ or 65,535 processes, the PIDs must start over at the lowest number not currently in use.

• These are just a few examples of this basic principle. Most modern computers store memory in units of 8 bits, called a "**byte**" (also called an "**octet**"). Arithmetic in such computers can be done in bytes, but is more often done in larger units called "**(short) integers**" (16 bits), "**long integers**" (32 bits) or "**double integers**" (64 bits). Short integers can be used to store numbers between 0 and $2^{16} - 1$, or 65,535. Long integers can be used to store numbers

between 0 and $2^{32} - 1$, or 4,294,967,295. and double integers can be used to store numbers between 0 and $2^{64} - 1$, or 18,446,744,073,709,551,615. (Check these!)

When a computer performs an unsigned integer arithmetic operation, there are three possible problems which can occur:

1. if the result is too large to fit into the number of bits assigned to it, an "**overflow**" is said to have occurred. For example if the result of an operation using 16 bit integers is larger than 65,535, an overflow results.
2. in the division of two integers, if the result is not itself an integer, a "**truncation**" is said to have occurred: 10 divided by 3 is truncated to 3, and the extra $1/3$ is lost. This is not a problem, of course, if the programmer's intention was to ignore the remainder!
3. any division by zero is an error, since division by zero is not possible in the context of arithmetic.

1.8 SIGNED INTEGERS

Signed integers are stored in a computer using 2's complement. As you recall, when computing the 2's complement of a number it was necessary to know how many bits were to be used in the final result; leading zeroes were appended to the most significant digit in order to make the number the appropriate length. Since the process of computing the 2's complement involves first computing the 1's complement, these leading zeros become leading ones, and the left most bit of a negative number is therefore always 1. In computers, the left most bit of a signed integer is called the "**sign bit**".

Consider an 8 bit signed integer: let us begin with $0\ 0\ 0\ 0\ 0\ 0\ 0\ 0_2$ and start counting by repeatedly adding 1:

- When you get to 127, the integer has a value of $0\ 1\ 1\ 1\ 1\ 1\ 1\ 1_2$; this is easy to see because you know now that a 7 bit integer can contain a value between 0 and $2^7 - 1$, or 127. What happens when we add 1?
- If the integer were unsigned, the next value would be $1\ 0\ 0\ 0\ 0\ 0\ 0\ 0_2$, or 128 (2^7). But since this is a signed integer, $1\ 0\ 0\ 0\ 0\ 0\ 0\ 0_2$ is a negative value: the sign bit is 1!
- Since this is the case, we must ask the question: what is the decimal value corresponding to the signed integer $1\ 0\ 0\ 0\ 0\ 0\ 0\ 0_2$? To answer this question, we must take the 2's

complement of that value, by first taking the 1's complement and then adding one.

- The 1's complement is $0\ 1\ 1\ 1\ 1\ 1\ 1\ 1_2$, or decimal 127. Since we must now add 1 to that, our conclusion is that the signed integer $1\ 0\ 0\ 0\ 0\ 0\ 0\ 0_2$ must be equivalent to decimal -128! Odd as this may seem, it is in fact the only consistent way to interpret 2's complement signed integers. Let us continue now to "count" by adding 1 to $1\ 0\ 0\ 0\ 0\ 0\ 0\ 0_2$:

- $1\ 0\ 0\ 0\ 0\ 0\ 0\ 0_2 + 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1_2$ is $1\ 0\ 0\ 0\ 0\ 0\ 0\ 1_2$.

- To find the decimal equivalent of $1\ 0\ 0\ 0\ 0\ 0\ 0\ 1_2$, we again take the 2's complement: the 1's complement is $0\ 1\ 1\ 1\ 1\ 1\ 1\ 0_2$ and adding 1 we get $0\ 1\ 1\ 1\ 1\ 1\ 1\ 1_2$ (127) so $1\ 0\ 0\ 0\ 0\ 0\ 0\ 1_2$ is equivalent to -127.

- We see then that once we have accepted the fact that $1\ 0\ 0\ 0\ 0\ 0\ 0\ 0_2$ is decimal -128, counting by adding one works as we would expect.

- Note that the most negative number which we can store in an 8 bit signed integer is -128, which is -2^{8-1} , and that the largest positive signed integer we can store in an 8 bit signed integer is 127, which is $2^{8-1} - 1$.

- The number of integers between -128 and + 127 (inclusive) is 256, which is 2^8 ; this is the same number of values which an unsigned 8 bit integer can contain (from 0 to 255).

- Eventually we will count all the way up to $1\ 1\ 1\ 1\ 1\ 1\ 1\ 1_2$. The 1's complement of this number is obviously 0, so $1\ 1\ 1\ 1\ 1\ 1\ 1\ 1_2$ must be the decimal equivalent of -1. Using our deliberations on 8 bit signed integers as a guide, we come to the following observations about signed integer arithmetic in general:

- **if a signed integer has n bits, it can contain a number between -2^{n-1} and $(2^{n-1} - 1)$.**

- **since both signed and unsigned integers of n bits in length can represent 2^n different values, there is no inherent way to distinguish signed integers from unsigned integers simply by looking at them; the software designer is responsible for using them correctly.**

- no matter what the length, if a signed integer has a binary value of all 1's, it is equal to decimal -1.

You should verify that a signed short integer can hold decimal values from -32,768 to +32,767, a signed long integer can contain values from -2,147,483,648 to +2,147,483,647 and a signed double integer can represent decimal values from -9,223, 372, 036, 854, 775, 808 to +9,223, 372, 036, 854, 775, 807.

There is an interesting consequence to the fact that in 2's complement arithmetic, one expects to throw away the final carry: in unsigned arithmetic a carry out of the most significant digit means that there has been an overflow, but in signed arithmetic an overflow is not so easy to detect. In fact, signed arithmetic overflows are detected by checking the consistency of the signs of the operands and the final answer. A signed overflow has occurred in an addition or subtraction if:

- the sum of two positive numbers is negative;
- the sum of two negative numbers is non-negative;
- subtracting a positive number from a negative one yields a positive result; or
- subtracting a negative number from a non-negative one yields a negative result.

Integer arithmetic on computers is often called "**fixed point**" arithmetic and the integers themselves are often called fixed point numbers. Real numbers on computers (which may have fractional parts) are often called "floating point" numbers.

1.9 1'S COMPLEMENT

Binary numbers can also be represented by 'radix' and 'radix -1' forms. 1's complement of a binary number N is obtained by the formula : $-(2^n - 1) - N$ where n is the no of bits in number N

Example

Convert binary number 111001101 to 1's complement.

Method:

$$N = 111001101$$

$$n = 9$$

$$2^n = 256 = 100000000$$

$$2^n - 1 = 255 = 111111111$$

$$1's \text{ complement of } N = (100000000 - 1) - 111001101$$

$$011111111$$

$$- 111001101$$

$$= 000110010$$

Answer:

1's complement of N is 000110010

Trick:

Invert all the bits of the binary number

$$N = 111001101$$

1's complement of N is 000110010

1.10 2'S COMPLEMENT

2's complement of a binary number N is obtained by the formula $(2^n) - N$
 where n is the no of bits in number N

Example:

Convert binary number 111001101 to 2's complement

Method

2's complement of a binary no can be obtained by two step process

Step 1

1's complement of number N = 000110010

Step 2

1's complement + 1

$$\begin{array}{r} 000110010 \\ + 000000001 \\ \hline = 000110011 \end{array}$$

Answer

2's complement of a binary no 111001101 is 000110011

Trick : 2's complement can be represented by keeping all lower significant bits till first 1 as it is and taking complement of all upper bits after that.

1.11 BINARY ARITHMETIC

1.11.1 Addition

- $0 + 0 = 0$
- $0 + 1 = 1$
- $1 + 0 = 1$
- $1 + 1 = 0$, and carry 1 to the next more significant bit

For example,

$$\begin{array}{r} 00011010 + 00001100 = 00100110 \\ \quad 1 \quad 1 \quad \quad \quad \text{carries} \\ 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 0 = 26_{(\text{base } 10)} \\ + 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 = 12_{(\text{base } 10)} \\ \hline \end{array}$$

$$\begin{array}{r} 0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 = 38_{(\text{base } 10)} \\ 00010011 + 00111110 = 01010001 \\ \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad \quad \quad \text{carries} \\ 0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 = 19_{(\text{base } 10)} \\ + 0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 0 = 62_{(\text{base } 10)} \\ \hline 0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0 \quad 1 = 81_{(\text{base } 10)} \end{array}$$

1.11.2 Subtraction

- $0 - 0 = 0$
- $0 - 1 = 1$, and borrow 1 from the next more significant bit
- $1 - 0 = 1$
- $1 - 1 = 0$

For example,

$$\begin{array}{r}
 00100101 - 00010001 = 00010100 \\
 \begin{array}{r}
 0 \qquad \qquad \qquad \text{borrows} \\
 0\ 0\ 4\ 1\ 0\ 0\ 1\ 0\ 1 = 37_{(\text{base } 10)} \\
 -0\ 0\ 0\ 1\ 0\ 0\ 0\ 1 = 17_{(\text{base } 10)} \\
 \hline
 0\ 0\ 0\ 1\ 0\ 1\ 0\ 0 = 20_{(\text{base } 10)}
 \end{array}
 \end{array}$$

$$\begin{array}{r}
 00110011 - 00010110 = 00011101 \\
 \begin{array}{r}
 0\ 1\ 0\ 1 \qquad \qquad \qquad \text{borrows} \\
 0\ 0\ 4\ 4\ 0\ 1\ 1 = 51_{(\text{base } 10)} \\
 -0\ 0\ 0\ 1\ 0\ 1\ 1\ 0 = 22_{(\text{base } 10)} \\
 \hline
 0\ 0\ 0\ 1\ 1\ 1\ 0\ 1 = 29_{(\text{base } 10)}
 \end{array}
 \end{array}$$

1.11.3 Multiplication

- $0 \times 0 = 0$
- $0 \times 1 = 0$
- $1 \times 0 = 0$
- $1 \times 1 = 1$, and no carry or borrow bits

For example,

$$00101001 \times 00000110 = 11110110$$

$$\begin{array}{r}
 \begin{array}{r}
 0\ 0\ 1\ 0\ 1\ 0\ 0\ 1 = 41_{(\text{base } 10)} \\
 \times 0\ 0\ 0\ 0\ 0\ 1\ 1\ 0 = 6_{(\text{base } 10)} \\
 \hline
 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\
 0\ 0\ 1\ 0\ 1\ 0\ 0\ 1 \\
 0\ 0\ 1\ 0\ 1\ 0\ 0\ 1 \\
 \hline
 0\ 0\ 1\ 1\ 1\ 1\ 0\ 1\ 1\ 0 = 246_{(\text{base } 10)}
 \end{array}
 \end{array}$$

$$00010111 \times 00000011 = 01000101$$

$$\begin{array}{r} 00010111 = 23_{(\text{base } 10)} \\ \times 00000011 = 3_{(\text{base } 10)} \end{array}$$

$$\begin{array}{r} 1111 \\ 10111 \\ 00010111 \\ \hline 001000101 = 69_{(\text{base } 10)} \end{array} \quad \text{carries}$$

Another Method: Binary multiplication is the same as repeated binary addition; add the multiplicand to itself the multiplier number of times.

$$00001000 \times 00000011 = 00011000$$

$$\begin{array}{r} 1 \\ 00001000 = 8_{(\text{base } 10)} \\ 00001000 = 8_{(\text{base } 10)} \\ + 00001000 = 8_{(\text{base } 10)} \\ \hline 00011000 = 24_{(\text{base } 10)} \end{array} \quad \text{carries}$$

1.11.4 Division

Binary division is the repeated process of subtraction, just as in decimal division.

For example,

$$00101010 \div 00000110 = 00000111$$

$$\begin{array}{r} 111 = 7_{(\text{base } 10)} \\ \hline 110 \) \ 004 \overset{1}{0} \ 1 \ 0 \ 1 \ 0 = 42_{(\text{base } 10)} \\ - \ 1 \ 1 \ 0 = 6_{(\text{base } 10)} \\ \hline 1 \\ 4 \ 0 \ \overset{1}{0} \ 1 \\ - \ 1 \ 1 \ 0 \\ \hline 1 \ 1 \ 0 \\ - \ 1 \ 1 \ 0 \\ \hline 0 \end{array} \quad \text{borrows}$$

Now let's look at an example where our problem does not generate an overflow bit. We will subtract 7_{10} from 1_{10} using 1's complement.

1. First, we state our problem in binary.

$$\begin{array}{r} 0001 \quad (1) \\ - 0111 \quad -(7) \\ \hline \end{array}$$
2. Next, we convert 0111_2 to its negative equivalent and add this to 0001_2 . Add 1_2 to it.

$$\begin{array}{r} 0001 \quad (1) \\ + 1000 \quad +(-7) \\ \hline 1001 \quad (?) \end{array}$$
3. This time our result does not cause an overflow, so we do not need to adjust the sum. Notice that our final answer is a negative number since it begins with a 1. Remember that our answer is in 2's complement notation so the correct decimal value for our answer is -6_{10} and not 9_{10} .
(In unsigned representation, we re-complement the answer 0101 add 1 to it 0110 and attach a – sign.)

$$\begin{array}{r} 0001 \quad (1) \\ + 1001 \quad +(-7) \\ \hline 1010 \quad (-6) \end{array}$$

1.12 QUESTIONS

1. Explain the following terms:
 - a. Radix
 - b. Decimal number system
 - c. Binary number system
 - d. Octal number system
 - e. Hexadecimal number system
 - f. 1's complement
 - g. 2's complement
2. Convert the following numbers from decimal to binary:
 - a. 512
 - b. 255
 - c. 56.21
 - d. 178.71
 - e. 223.25
3. Convert the following numbers from binary to decimal:
 - a. 11011000111
 - b. 11111111111
 - c. 10101010
 - d. 1111.011
 - e. 1101101.11
4. Convert the following numbers from decimal to octal:
 - a. 1024
 - b. 800
 - c. 789
 - d. 826
 - e. 425

5. Convert the following numbers from octal to decimal:
 - a. 145
 - b. 512
 - c. 677
 - d. 177
 - e. 1024
6. Convert the following numbers from decimal to hexadecimal:
 - a. 4096
 - b. 975
 - c. 1263
 - d. 127
 - e. 659
7. Convert the following numbers from hexadecimal to decimal, binary and octal:
 - a. FFFF
 - b. A2C3
 - c. 8ABC
 - d. 1235
 - e. 6857
8. Perform the following subtraction in binary using 1's complement and 2's complement:
 - a. $45 - 26$
 - b. $87 - 96$
 - c. $128 - 65$
 - d. $142 - 220$
 - e. $100 - 56$

1.13 FUTHER READING

- ❖ **Computer Architecture and Organization** by William Stallings
- ❖ **Fundamentals of Computer organization and Design** by Sivarama P. Dandamudi
- ❖ **Digital Electronics** - An Introduction to Theory and Practice by W H Gothmann
- ❖ **Binary Functions and Their Applications** by Stormer, H., Beckmann, Martin J
- ❖ The Number System by H. A. Thurston
- ❖ http://en.wikipedia.org/wiki/Analog_signal
- ❖ http://en.wikipedia.org/wiki/Binary_numeral_system



CODES

Unit Structure

2.0 Objectives

2.1 Binary codes

2.1.1 Binary-coded-decimal Numbers

2.2 Geometric Representation of binary Numbers

2.3 Distance

2.4 Unit-distance codes

2.5 Symmetries of the n-cube

2.6 Error-detecting and error-correcting codes

2.7 Single-error-correcting codes

2.8 Ascii code

2.9 Ebcddic code

2.10 Unicode

2.11 Questions

2.12 Further Reading

3.0 OBJECTIVES

After completing this chapter, you will be able to:

- ❖ Understand the binary codes and arithmetic with binary codes.
- ❖ Learning the geometrical representation of Binary Numbers.
- ❖ Understand the Unit – Distance codes.
- ❖ Learn the Symmetries Of The N-Cube.
- ❖ Work with error handling and error detection codes.
- ❖ Learn the basics about the ASCII, EBCDIC & UNICODE and use the codes in arithmetic.

3.1 BINARY CODES

The binary number system has many advantages and is widely used in digital systems. However, there are times when

binary numbers are not appropriate. Since we think much more readily in terms of decimal numbers than binary numbers, facilities are usually provided so that data can be entered into the system in decimal form, the conversion to binary being performed automatically inside the system. In fact, many computers have been designed which work entirely with decimal numbers. For this to be possible, a scheme for representing each of the 10 decimal digits as a sequence of binary digits must be used.

3.1.1 Binary-Coded-Decimal Numbers

To represent 10 decimal digits, it is necessary to use at least 4 binary digits, since there are 24, or 16, different combinations of 4 binary digits but only 23, or 8, different combinations of 3 binary digits. If 4 binary digits, or **bits**, are used and only one combination of bits is used to represent each decimal digit, there will be six unused or invalid code words. In general, any arbitrary assignment of combinations of bits to digits can be used so that there are $16!/6!$ or approximately 2.9×10^{10} possible codes.

Binary Codes:

Decimal digit	8 b_3	4 b_2	2 b_1	1 b_0	8	4	-2	-1	2	4	2	1	Excess-3			
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	1	1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	1	0	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	0	1	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	0	0	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	1	1	1	0	1	1	1	0	0	0
6	0	1	1	0	1	0	1	0	1	1	0	0	1	0	0	1
7	0	1	1	1	1	0	0	1	1	1	0	1	1	0	1	0
8	1	0	0	0	1	0	0	0	1	1	1	0	1	0	1	1
9	1	0	0	1	1	1	1	1	1	1	1	1	1	1	0	0

Only a few of these codes have ever been used in any system, since the arithmetic operations are very difficult in almost all of the possible codes. Several of the more common 4-bit decimal codes are shown in Table. The 8,4,2,1 code is obtained by taking the first 10 binary numbers and assigning them to the corresponding decimal digits. This code is an example of a **weighted code**, since the decimal digits can be determined from the binary digits by forming the sum $d = 8b_3 + 4b_2 + 2b_1 + b_0$. The coefficients 8, 4, 2, 1 are known as the **code weights**. The number 462 would be represented as 0100 0110 0010 in the 8,4,2,1 code. It has been shown in that there are only 17 different sets of weights possible for a positively weighted code: (3,3,3,1), (4,2,2,1), (4,3,1,1), (5,2,1,1), (4,3,2,1), (4,4,2,1), (5,2,2,1), (5,3,1,1), (5,3,2,1), (5,4,2,1), (6,2,2,1), (6,3,1,1), (6,3,2,1), (6,4,2,1), (7,3,2,1), (7,4,2,1), (8,4,2,1).

It is also possible to have a weighted code in which some of the weights are negative, as in the 8,4,-2,-1 code shown in Table. This code has the useful property of being **self-complementing**: if a code word is formed by complementing each bit individually (changing 1's to 0's and 0's to 1's), then this new code word represents the 9's complement of the digit to which the original code word corresponds. For example, 0101 represents 3 in the 8,4,-2,-1 code, and 1010 represents 6 in this code. In general, if b'_i denotes the complement of b_i , then a code is self-complementing if, for any code word $b_3b_2b_1b_0$ representing a digit d_i , the code word $b'_3b'_2b'_1b'_0$ represents $9 - d_i$. The 2,4,2,1 code of Table is an example of a self-complementing code having all positive weights, and the excess-3 code is an example of a code which is self-complementing but not weighted. The excess-3 code is obtained from the 8,4,2,1 code by adding (using binary arithmetic) 0011 (or 3) to each 8,4,2,1 code word to obtain the corresponding excess-3 code word.

Binary codes using more than 4-bits

Decimal digit	2-out-of-5	Biquinary 5043210
0	00011	010001
1	00101	0100010
2	00110	0100100
3	01001	0101000
4	01010	0110000
5	01100	1000001
6	10001	1000010
7	10010	1000100
8	10100	1001000
9	11000	1010000

Although 4 bits are sufficient for representing the decimal digits, it is sometimes expedient to use more than 4 bits in order to achieve arithmetic simplicity or ease in error detection. The 2-out-of-5 code shown in Table has the property that each code word has exactly two 1's. A single error which complements 1 of the bits will always produce an invalid code word and is therefore easily detected. This is an unweighted code. The biquinary code shown in Table is a weighted code in which 2 of the bits specify whether the digit is in the range 0 to 4 or the range 5 to 9 and the other 5 bits identify where in the range the digit occurs.

3.2 GEOMETRIC REPRESENTATION OF BINARY NUMBERS

An n -bit binary number can be represented by what is called a **point in n -space**. To see just what is meant by this, consider the set of 1-bit binary numbers, that is, 0 and 1. This set can be represented by two points in 1-space, i.e., by two points on a line. Such a presentation is called a **1-cube** and is shown in Figure. (A **0-cube** is a single point in 0-space.) Now consider the set of 2-bit binary numbers, that is, 00, 01, 10, 11 (or, decimally, 0, 1, 2, 3). This set can be represented by four points (also called **vertices**, or **nodes**) in 2-space. This representation is called a **2-cube** and is shown in Figure. Note that this figure can be obtained by projecting the 1-cube (i.e., the horizontal line with two points) downward and by prefixing a 0 to 0 and 1 on the original 1-cube and a 1 to 0 and 1 on the projected 1-cube. A similar projection procedure can be followed in obtaining any next-higher-dimensional figure. For example, the representation for the set of 3-bit binary numbers is obtained by projecting the 2-cube representation of Figure c.

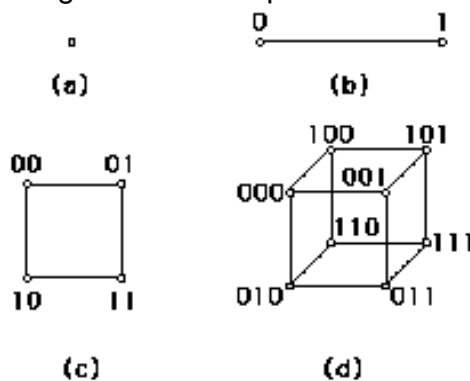


Figure: n -Cubes for $n = 0, 1, 2, 3$: (a) 0-cube; (b) 1-cube; (c) 2-cube; (d) 3-cube.

A 0 is prefixed to the bits on the original 2-cube, and a 1 is prefixed to the bits on the projection of the 2-cube. Thus, the 3-bit representation, or **3-cube**, is shown in Figure d. A more formal statement for the projection method of defining an n -cube is as follows:

1. A 0-cube is a single point with no designation.
2. An n -cube is formed by projecting an $(n-1)$ -cube. A 0 is prefixed to the designations of the points of the original $(n-1)$ -cube, and a 1 is prefixed to the designations of the points of the projected $(n-1)$ -cube.

There are 2^n points in an n -cube. A **p -subcube** of an n -cube. ($p < n$) is defined as a collection of any 2^p points which have exactly $(n - p)$ corresponding bits all the same. For example, the points 100, 101, 000, and 001 in the 3-cube (Figure. d) form a 2-subcube, since there are $2^2 = 4$ total points and $3 - 2 = 1$ of the bits (the second) is the same for all four points. In general, there are $(n!2^{n-p})/[(n-p)!p!]$ different p -subcubes in an n -cube, since there are $\binom{n}{n-p} = \frac{n!}{(n-p)!p!}$ (number of ways of selecting n things taken $n - p$ at a time) ways in which $n - p$ of the bits may be the same, and there are 2^{n-p} combinations which these bits may take on. For example, there are $(3!2^2)/(2!1!) = 12$ 1-subcubes (line segments) in a 3-cube, and there are $(3!2^1)/(1!2!) = 6$ 2-subcubes ("squares") in a 3-cube.

Besides the form shown in Figure, there are two other methods of drawing an n -cube which are frequently used. The first of these is shown in Figure 2 for the 3-and 4-cubes. It is seen that these still agree with the projection scheme and are merely a particular way of drawing the cubes. The lines which are dotted are usually omitted for convenience in drawing.

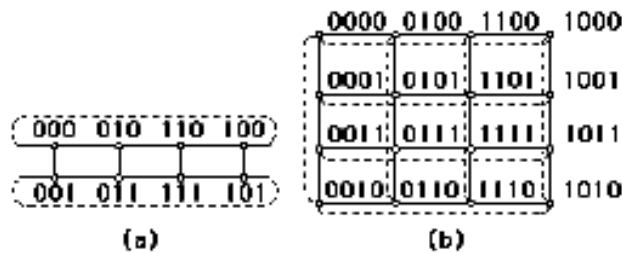


Figure: Alternative representations: (a) 3-cube; (b) 4-cube.

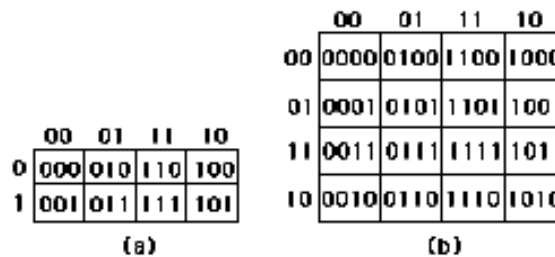


Figure n -Cube maps for $n = 3$ (a) and $n = 4$ (b).

If in the representation of Figure we replace each dot by a square area, we have what is known as an **n -cube map**. This representation is shown for the 3- and 4- cubes in Figure Maps will be of considerable use to us later. Notice that the appropriate entry for each cell of the maps of Figure can be determined from the

corresponding row and column labels. It is sometimes convenient to represent the points of an n -cube by the decimal equivalents of their binary designations. For example, Figure shows the 3- and 4-cube maps represented this way. It is of interest to note that, if a point has the decimal equivalent N_i in an n -cube, in an $(n + 1)$ -cube this point and its projection (as defined) become N_i and $N_i + 2^n$.

3.3 DISTANCE

A concept which will be of later use is that of the distance between two points on an n -cube. Briefly, the **distance** between two points on an n -cube is simply the number of coordinates (bit positions) in which the binary representations of the two points differ. This is also called the **Hamming distance**.

For example, 10110 and 01101 differ in all but the third coordinate (from left or right). Since the points differ in four coordinates, the distance between them is 4. A more formal definition is as follows: First, define the **mod 2 sum** of two bits, a and b , by

$$\begin{aligned} 0 \oplus 0 &= 0 & 1 \oplus 0 &= 1 \\ 0 \oplus 1 &= 1 & 1 \oplus 1 &= 0 \end{aligned}$$

That is, the sum is 0 if the 2 bits are alike, and it is 1 if the 2 bits are different. Now consider the binary representations of two points, $P_i = (a_n \bar{a}_{n-1} \bar{a}_{n-2} \dots a_0)$ and $P_j = (b_n \bar{b}_{n-1} \bar{b}_{n-2} \dots b_0)$, on the n -cube. The mod 2 sum of these two points is defined as

$$P_k = P_i \oplus P_j = (a_n \bar{b}_n \oplus b_n \bar{a}_n, \bar{a}_{n-1} \oplus b_{n-1}, \bar{a}_{n-2} \oplus b_{n-2}, \dots, a_0 \oplus b_0)$$

This sum P_k is the binary representation of another point on the n -cube. The number of 1's in the binary representation P_i is defined as the **weight** of P_i and is given the symbol $|P_i|$. Then the distance (or **metric**) between two points is defined as

$$D(P_i, P_j) = |P_i \oplus P_j|$$

The distance function satisfies the following three properties:

- $D(P_i, P_j) = 0$ if and only if $P_i = P_j$
- $D(P_i, P_j) = D(P_j, P_i) > 0$ if $P_i \neq P_j$
- $D(P_i, P_j) + D(P_j, P_k) \geq D(P_i, P_k)$ Triangle inequality

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

(a)

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

(b)

Figure: Decimal labels in n -cube maps: (a) 3-cube map; (b) 4-cube map.

To return to the more intuitive approach, since two adjacent points (connected by a single line segment) on an n -cube form a 1-subcube, they differ in exactly one coordinate and thus are distance 1 apart. We see then that, to any two points which are distance D apart, there corresponds a **path** of D connected line segments on the n -cube joining the two points. Furthermore, there will be more than one path of length D connecting the two points (for $D > 1$ and $n \geq 2$), but there will be no path shorter than length D connecting the two points. A given shortest path connecting the two points, thus, cannot intersect itself, and $D + 1$ nodes (including the end points) will occur on the path.

3.4 UNIT-DISTANCE CODES

In terms of the geometric picture, a code is simply the association of the decimal integers (0,1,2,...) with the points on an n -cube. There are two types of codes which are best described in terms of their geometric properties. These are the so-called **unitdistance codes** and **error-detecting** and **error-correcting codes**. A unit-distance code is simply the association of the decimal integers (0,1,2,...) with the points on a connected path in the n -cube such that the distance is 1 between the point corresponding to any integer i and the point corresponding to integer $i + 1$ (see Figure). That is, if P_i is the binary-code word for decimal integer i , then we must have

$$D(P_i, P_{i+1}) = 1 \quad i = 0, 1, 2, \dots$$

Unit-distance codes are used in devices for converting analog or continuous signals such as voltages or shaft rotations into binary numbers which represent the magnitude of the signal. Such a device is called an **analog-digital converter**. In any such device there must be boundaries between successive digits, and it is always possible for there to be some misalignment among the different bit positions at such a boundary. For example, if the seventh position is represented by 0111 and the eighth position by 1000, misalignment could cause signals corresponding to 1111 to be generated at the boundary between 7 and 8. If binary numbers were used for such a device, large errors could thus occur. By using a unit-distance code in which adjacent positions differ only in 1 bit, the error due to misalignment can be eliminated.

The highest integer to be encoded may or may not be required to be distance 1 from the code word for 0. If it is distance 1, then the path is closed. Of particular interest is the case of a closed nonintersecting path which goes through all 2^n points of the n -cube. In graph theory such a path is known as a (closed) **Hamilton line**. Any unit-distance code associated with such a path is sometimes called a **Gray code**, although this term is usually

reserved for a particular one of these codes. To avoid confusing terminology, we shall refer to a unit-distance code which corresponds to a closed Hamilton line as a **closed n code**. This is a unit-distance code containing $2n$ code words in which the code word for the largest integer ($2n - 1$) is distance 1 from the code word for the least integer (0). An **open n code** is similar except that the code words for the least and largest integer, respectively, are not distance 1 apart. The most useful unit distance code is the Gray code which is shown in Table The attractive feature of this code is the simplicity of the algorithm for translating from the binary number system into the Gray code.

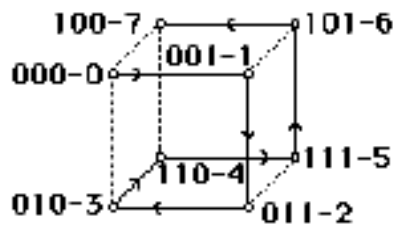


Figure: Path on a 3-cube corresponding to a unit-distance code.

Unit-distance code

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

This algorithm is described by the expression

$$g_i = b_i \oplus b_{i+1}$$

Decimal	Binary				Gray			
	b_3	b_2	b_1	b_0	g_3	g_2	g_1	g_0
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1
2	0	0	1	0	0	0	1	1
3	0	0	1	1	0	0	1	0
4	0	1	0	0	0	1	1	0
5	0	1	0	1	0	1	1	1
6	0	1	1	0	0	1	0	1
7	0	1	1	1	0	1	0	0
8	1	0	0	0	1	1	0	0
9	1	0	0	1	1	1	0	1
10	1	0	1	0	1	1	1	1
11	1	0	1	1	1	1	1	0
12	1	1	0	0	1	0	1	0
13	1	1	0	1	1	0	1	1
14	1	1	1	0	1	0	0	1
15	1	1	1	1	1	0	0	0

The Gray code

3.5 SYMMETRIES OF THE N-CUBE

A **symmetry** of the n -cube is defined to be any one-to-one translation of the binary point representations on the n -cube which leaves all pairwise distances the same. If we consider the set of binary numbers, we see that there are only two basic translation schemes which leave pairwise distances the same. (1) The bits of one coordinate may be interchanged with the bits of another coordinate in all code words. (2) The bits of one coordinate may be complemented (i.e., change 1's to 0's and 0's to 1's) in all code words. Since there are $n!$ translation schemes possible using (1), and since there are $2n$ ways in which coordinates may be complemented, there are $2n$ translation schemes possible using (2). Thus, in all there are $2n(n!)$ symmetries of the n -cube. This means that for any n -bit code there are $2n(n!)-1$ rather trivial modifications of the original code (in fact, some of these may result in the original code) which can be obtained by interchanging and complementing coordinates. The pairwise distances are the same in all these codes.

It is sometimes desired to enumerate the different types of a class of codes. Two codes are said to be of the same **type** if a symmetry of the n -cube translates one code into the other (i.e., by interchanging and complementing coordinates). As an example, we might ask: What are the types of closed n codes? It turns out that for $n < 4$ there is just one type, and this is the type of the conventional Gray code. For $n = 4$, there are nine types. Rather than specify a particular code of each type, we can list these types by specifying the sequence of coordinate changes for a closed path

of that type. On the assumption that the coordinates are numbered (3210), the nine types are shown in Table

TABLE Nine different types of unit-distance 4-bit code

Type																
1 (Gray)	0	1	0	2	0	1	0	3	0	1	0	2	0	1	0	3
2	1	0	1	3	1	0	1	2	0	1	0	3	0	1	0	2
3	1	0	1	3	0	1	0	2	1	0	1	3	0	1	0	2
4	1	0	1	3	2	3	1	0	1	3	1	0	2	0	1	3
5	1	0	1	3	2	0	1	3	1	0	1	3	2	0	1	3
6	1	0	1	3	2	3	1	3	2	0	1	2	1	3	1	2
7	1	0	1	3	2	0	2	1	0	2	0	3	0	1	0	2
8	1	0	1	3	2	1	2	0	1	2	1	3	0	1	0	2
9	1	0	1	3	2	3	1	0	3	0	2	0	1	2	3	2

3.6 ERROR-DETECTING AND ERROR-CORRECTING CODES

Special features are included in many digital systems for the purpose of increasing system reliability. In some cases circuits are included which indicate when an error has occurred—error detection—and perhaps provide some information as to where the error is—error diagnosis. Sometimes it is more appropriate to provide **error correction**: circuits not only detect a malfunction but act to automatically correct the erroneous indications caused by it. One technique used to improve reliability is to build two duplicate systems and then to run them in parallel, continually comparing the outputs of the two systems. When a mismatch is detected, actions are initiated to determine the source of the error and to correct it. Another approach uses three copies of each system module and relies on voter elements to select the correct output in case one of the three copies has a different output from the other two. This technique is called triple modular redundancy (TMR). Such costly designs are appropriate either when the components are not sufficiently reliable or in systems where reliability is very important as in real-time applications such as telephony, airline reservations, or space vehicles.

In many other applications where such massive redundancy is not justified it is still important to introduce some (less costly) techniques to obtain some improvement in reliability. A very basic and common practice is to introduce some redundancy in encoding the information manipulated in the system. For example, when the 2-out-of-5 code is used to represent the decimal digits, any error in only one bit is easily detected since if any single bit is changed the resulting binary word no longer contains exactly two 1's. While it is true that there are many 2-bit errors which will not be detected by

this code, it is possible to argue that in many situations multiple errors are so much less likely than single errors that it is reasonable to ignore all but single errors. Suppose it is assumed that the probability of any single bit being in error is p and that this probability is independent of the condition of any other bits. Also suppose that p is very much less than one, (i.e., that the components are very reliable). Then the probability of all 5 bits representing one digit being correct is $P_0 = (\bar{1}p)^5$, the probability of exactly one error is $P_1 = 5(\bar{1}p)^4p$ and the probability of two errors is $P_2 = 10(\bar{1}p)^3p^2$. Taking the ratio $P_2/P_1 = 2p/(\bar{1}p) \ll 2p/(1+p) \ll 1$, showing that the probability of a double error is much smaller than that of a single error. Arguments such as this are the basis for the very common emphasis on handling only single errors. It is possible to easily convert any of the 4-bit decimal codes to single-error detecting codes by the addition of a single bit a parity bit as is illustrated for the 8421 code in Table. The **parity bit** p is added to each code word so as to make the total number of 1's in the resultant 5-bit word even;

$$\text{i.e., } p = b_0 \oplus b_1 \oplus b_2 \oplus b_3$$

If any one bit is reversed it will change the overall parity (number of 1's) from even to odd and thus provide an error indication. This technique of adding a parity bit to a set of binary words is not peculiar to binary-coded-decimal schemes but is generally applicable. It is common practice to add a parity bit to all information recorded on magnetic tapes.

TABLE 8421 code with parity bit added

Decimal digit	8 b_3	4 b_2	2 b_1	1 b_0	Parity, p
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	1
3	0	0	1	1	0
4	0	1	0	0	1
5	0	1	0	1	0
6	0	1	1	0	0
7	0	1	1	1	1
8	1	0	0	0	1
9	1	0	0	1	0

The 8421 code with a parity bit added is shown plotted on the 5-cube map of Figure Inspection of this figure shows that the minimum distance between any two words is two as must be true for any single-error-detecting code.

In summary, *any single-error-detecting code must have a minimum distance between any two code words of at least two*, and

any set of binary words with minimum distance between words of at least two can be used as a single-error-detecting code.

Also the addition of a parity bit to any set of binary words will guarantee that the minimum distance between any two words is at least two.

		$p=0$			
		b_3b_2			
b_1b_0		00	01	11	10
00		0			
01			5		9
11		3			
10			6		

		$p=1$			
		b_3b_2			
b_1b_0		00	01	11	10
00			4		8
01		1			
11			7		
10		2			

Figure Five-cube map for the 8421 BCD code with parity bit p

3.7 SINGLE-ERROR-CORRECTING CODES

A parity check over all the bits of a binary word provides an indication if one of the bits is reversed; however, it provides no information about which bit was changed, all bits enter into the parity check in the same manner. If it is desired to use parity checks to not only detect an altered bit but also to identify the altered bit, it is necessary to resort to several parity checks, each checking a different set of bits in the word. For example, consider the situation in Table in which there are three bits, M_1 , M_2 , and M_3 , which are to be used to represent eight items of information and there are two parity check bits C_1 and C_2 . The information bits, M_i , are often called **message bits** and the C_i bits **check bits**. As indicated in the table C_1 is obtained as a parity check over bits M_1 and M_3 , while C_2 checks bits M_2 and M_3 .

At first glance it might seem that this scheme might result in a single-error-correcting code since an error in M_3 alters both parity checks while an error in M_1 or M_2 each alters a distinct single parity check. This reasoning overlooks the fact that it is possible to have an error in a check bit as well as an error in a message bit. Parity check one could fail as a result of an error either in message bit M_1 or in check bit C_1 . Thus in this situation it would not be clear whether M_1 should be changed or not. In order to obtain a true single-error-correcting code it is necessary to add an additional check bit as in Table.

A parity check table

	<table style="margin: auto; border: 1px solid black; padding: 5px;"> <tr> <td style="padding: 2px 10px;">M_1</td> <td style="padding: 2px 10px;">M_2</td> <td style="padding: 2px 10px;">M_3</td> <td style="padding: 2px 10px;">C_1</td> <td style="padding: 2px 10px;">C_2</td> </tr> <tr> <td style="padding: 2px 10px; text-align: center;">×</td> <td></td> <td style="padding: 2px 10px; text-align: center;">×</td> <td style="padding: 2px 10px; text-align: center;">×</td> <td></td> </tr> <tr> <td></td> <td style="padding: 2px 10px; text-align: center;">×</td> <td style="padding: 2px 10px; text-align: center;">×</td> <td></td> <td style="padding: 2px 10px; text-align: center;">×</td> </tr> </table>	M_1	M_2	M_3	C_1	C_2	×		×	×			×	×		×																																																	
M_1	M_2	M_3	C_1	C_2																																																													
×		×	×																																																														
	×	×		×																																																													
	$C_1 = M_1 \oplus M_3, C_2 = M_2 \oplus M_3$																																																																
(a)	(b)																																																																
	<table style="margin: auto; border: 1px solid black; padding: 5px;"> <tr> <td style="padding: 2px 10px;">M_1</td> <td style="padding: 2px 10px;">M_2</td> <td style="padding: 2px 10px;">M_3</td> <td style="padding: 2px 10px;">C_1</td> <td style="padding: 2px 10px;">C_2</td> <td style="padding: 2px 10px;">C_3</td> </tr> <tr> <td style="padding: 2px 10px; text-align: center;">×</td> <td></td> <td style="padding: 2px 10px; text-align: center;">×</td> <td style="padding: 2px 10px; text-align: center;">×</td> <td></td> <td></td> </tr> <tr> <td></td> <td style="padding: 2px 10px; text-align: center;">×</td> <td style="padding: 2px 10px; text-align: center;">×</td> <td></td> <td style="padding: 2px 10px; text-align: center;">×</td> <td></td> </tr> <tr> <td style="padding: 2px 10px; text-align: center;">×</td> <td style="padding: 2px 10px; text-align: center;">×</td> <td></td> <td></td> <td></td> <td style="padding: 2px 10px; text-align: center;">×</td> </tr> </table>	M_1	M_2	M_3	C_1	C_2	C_3	×		×	×				×	×		×		×	×				×	$C_1 = M_1 \oplus M_3$ $C_2 = M_2 \oplus M_3$ $C_3 = M_1 \oplus M_2$																																							
M_1	M_2	M_3	C_1	C_2	C_3																																																												
×		×	×																																																														
	×	×		×																																																													
×	×				×																																																												
	(c)																																																																
	<table style="margin: auto; border-collapse: collapse;"> <thead> <tr> <th style="padding: 2px 10px;"></th> <th style="padding: 2px 10px;">M_1</th> <th style="padding: 2px 10px;">M_2</th> <th style="padding: 2px 10px;">M_3</th> <th style="padding: 2px 10px;">C_1</th> <th style="padding: 2px 10px;">C_2</th> <th style="padding: 2px 10px;">C_3</th> </tr> </thead> <tbody> <tr> <td style="padding: 2px 10px;"><i>a</i></td> <td style="padding: 2px 10px; text-align: center;">0</td> <td style="padding: 2px 10px; text-align: center;">0</td> <td style="padding: 2px 10px; text-align: center;">0</td> <td style="padding: 2px 10px; text-align: center;">0</td> <td style="padding: 2px 10px; text-align: center;">0</td> <td style="padding: 2px 10px; text-align: center;">0</td> </tr> <tr> <td style="padding: 2px 10px;"><i>b</i></td> <td style="padding: 2px 10px; text-align: center;">0</td> <td style="padding: 2px 10px; text-align: center;">0</td> <td style="padding: 2px 10px; text-align: center;">1</td> <td style="padding: 2px 10px; text-align: center;">1</td> <td style="padding: 2px 10px; text-align: center;">1</td> <td style="padding: 2px 10px; text-align: center;">0</td> </tr> <tr> <td style="padding: 2px 10px;"><i>c</i></td> <td style="padding: 2px 10px; text-align: center;">0</td> <td style="padding: 2px 10px; text-align: center;">1</td> <td style="padding: 2px 10px; text-align: center;">0</td> <td style="padding: 2px 10px; text-align: center;">0</td> <td style="padding: 2px 10px; text-align: center;">1</td> <td style="padding: 2px 10px; text-align: center;">1</td> </tr> <tr> <td style="padding: 2px 10px;"><i>d</i></td> <td style="padding: 2px 10px; text-align: center;">0</td> <td style="padding: 2px 10px; text-align: center;">1</td> <td style="padding: 2px 10px; text-align: center;">1</td> <td style="padding: 2px 10px; text-align: center;">1</td> <td style="padding: 2px 10px; text-align: center;">0</td> <td style="padding: 2px 10px; text-align: center;">1</td> </tr> <tr> <td style="padding: 2px 10px;"><i>e</i></td> <td style="padding: 2px 10px; text-align: center;">1</td> <td style="padding: 2px 10px; text-align: center;">0</td> <td style="padding: 2px 10px; text-align: center;">0</td> <td style="padding: 2px 10px; text-align: center;">1</td> <td style="padding: 2px 10px; text-align: center;">0</td> <td style="padding: 2px 10px; text-align: center;">1</td> </tr> <tr> <td style="padding: 2px 10px;"><i>f</i></td> <td style="padding: 2px 10px; text-align: center;">1</td> <td style="padding: 2px 10px; text-align: center;">0</td> <td style="padding: 2px 10px; text-align: center;">1</td> <td style="padding: 2px 10px; text-align: center;">0</td> <td style="padding: 2px 10px; text-align: center;">1</td> <td style="padding: 2px 10px; text-align: center;">1</td> </tr> <tr> <td style="padding: 2px 10px;"><i>g</i></td> <td style="padding: 2px 10px; text-align: center;">1</td> <td style="padding: 2px 10px; text-align: center;">1</td> <td style="padding: 2px 10px; text-align: center;">0</td> <td style="padding: 2px 10px; text-align: center;">1</td> <td style="padding: 2px 10px; text-align: center;">1</td> <td style="padding: 2px 10px; text-align: center;">0</td> </tr> <tr> <td style="padding: 2px 10px;"><i>h</i></td> <td style="padding: 2px 10px; text-align: center;">1</td> <td style="padding: 2px 10px; text-align: center;">1</td> <td style="padding: 2px 10px; text-align: center;">1</td> <td style="padding: 2px 10px; text-align: center;">0</td> <td style="padding: 2px 10px; text-align: center;">0</td> <td style="padding: 2px 10px; text-align: center;">0</td> </tr> </tbody> </table>		M_1	M_2	M_3	C_1	C_2	C_3	<i>a</i>	0	0	0	0	0	0	<i>b</i>	0	0	1	1	1	0	<i>c</i>	0	1	0	0	1	1	<i>d</i>	0	1	1	1	0	1	<i>e</i>	1	0	0	1	0	1	<i>f</i>	1	0	1	0	1	1	<i>g</i>	1	1	0	1	1	0	<i>h</i>	1	1	1	0	0	0	
	M_1	M_2	M_3	C_1	C_2	C_3																																																											
<i>a</i>	0	0	0	0	0	0																																																											
<i>b</i>	0	0	1	1	1	0																																																											
<i>c</i>	0	1	0	0	1	1																																																											
<i>d</i>	0	1	1	1	0	1																																																											
<i>e</i>	1	0	0	1	0	1																																																											
<i>f</i>	1	0	1	0	1	1																																																											
<i>g</i>	1	1	0	1	1	0																																																											
<i>h</i>	1	1	1	0	0	0																																																											

TABLE Eight-word single-error-correcting code: (a) Parity check table; (b) parity check equations; (c) Single-error-correcting code

Inspection of the parity check table in Table shows that an error in any one of the check bits will cause exactly one parity check violation while an error in any one of the message bits will cause violations of a distinct pair of parity checks. Thus it is possible to uniquely identify any single error. The code words of Table are shown plotted on the 6-cube map of Figure Each code word is indicated by the corresponding letter and all cells distance 1 away from a code word are marked with an \square .

The fact that no cell has more than one \square shows that no cell is distance one away from two code words. Since a single error changes a code word into a new word distance one away and each of such words is distance one away from only one code word it is possible to correct all single errors. A necessary consequence of the fact that no word is distance one away from more than one code word is the fact that the minimum distance between any pair of code words is three. In fact the *necessary and sufficient conditions for any set of binary words to be a single-error-correcting code is that the minimum distance between any pair of words be three.*

A single error correcting code can be obtained by any procedure which results in a set of words which are minimum distance three apart. The procedure illustrated in Table is due to and due to its systematic nature is almost universally used for single-error-codes. With three parity check bits it is possible to obtain a single-error-correcting code of more than eight code words. In fact up to sixteen code words can be obtained. The parity check table for a code with three check bits, C_1 , C_2 , and C_3 , and four message bits M_3 , M_5 , M_6 and M_7 is shown in Table. The peculiar numbering of the bits has been adopted to demonstrate the fact that it is possible to make a correspondence between the bit positions and the entries of the parity check table. If the blanks in the table are replaced by 0's and the \square 's by 1's then each column will be a binary number which is the equivalent of the subscript on the corresponding code bit. The check bits are placed in the bit positions corresponding to binary powers since they then enter into only one parity check making the formation of the parity check equations very straightforward.

The fact that Table leads to a single-error-correcting code follows from the fact that each code bit enters into a unique set of parity checks. In fact, *the necessary and sufficient conditions for a parity check table to correspond to a single-error-correcting code are that each column of the table be distinct (no repeated columns) and that each column contain at least one entry*. It follows from this that with K check bits it is possible to obtain a single-error-correcting code having at most 2^K total bits.¹ There are 2^K different columns possible but the empty column must be excluded leaving $2^K - 1$ columns.

		00				01				11						
		$M_3 C_1$				$M_1 M_2$				$M_3 C_1$						
$C_2 C_3$		00	01	11	10	$C_2 C_3$	00	01	11	10	$C_2 C_3$	00	01	11	10	
0	0	a	x	x	x	0	0	x		x	x	0	0	x	x	x
0	1	x	x	x		0	1	x	x	d	x	0	1		x	x
1	1	x		x	x	1	1	c	x	x	x	1	1	x	x	
1	0	x	x	b	x	1	0	x	x	x		1	0	x	g	x
		10				11				10						
		$M_3 C_1$				$M_3 C_1$				$M_3 C_1$						
$C_2 C_3$		00	01	11	10	$C_2 C_3$	00	01	11	10	$C_2 C_3$	00	01	11	10	
0	0	x	x		x	0	0	x	x	x	h	0	0	x	x	x
0	1	x	e	x	x	0	1		x	x	x	0	1	x	x	
1	1	x	x	x	f	1	1	x	x		x	1	1	x	x	
1	0		x	x	x	1	0	x	g	x	x	1	0	x	x	

Figure Six-cube map

TABLE Parity check table for a single-error-correcting code with 3 check bits and 4 message bits

C_1	C_2	M_3	C_4	M_5	M_6	M_7
			x	x	x	x
	x	x			x	x
x		x		x		x

$$C_1 = M_3 \oplus M_5 \oplus M_7$$

$$C_2 = M_3 \oplus M_6 \oplus M_7$$

$$C_4 = M_5 \oplus M_6 \oplus M_7$$

3.8 ASCII CODE

Most programming languages have a means of defining a character as a numeric code and, conversely, converting the code back to the character.

ASCII - American Standard Code for Information Interchange. A coding standard for characters, numbers, and symbols that is the same as the first 128 characters of the ASCII character set but differs from the remaining characters.

The ASCII character set (excluding the extended characters defined by IBM) is divided into four groups of 32 characters.

The first 32 characters, ASCII codes 0 through 1Fh, form a special set of non-printing characters called the control characters. We call them control characters because they perform various printer/display control operations rather than displaying symbols.

Examples of common control characters include:

- carriage return (ASCII code 0Dh), which positions the cursor to the left side of the current line of characters,
- line feed (ASCII code 0Ah), which moves the cursor down one line on the output device
- back space (ASCII code 08h), which moves the cursor back one position to the left

Unfortunately, different control characters perform different operations on different output devices. There is very little standardization among output devices. To find out exactly how a control character affects a particular device, you will need to consult its manual.

The second group of 32 ASCII character codes comprise various punctuation symbols, special characters, and the numeric digits. The most notable characters in this group include the:

- space character (ASCII code 20h)
- numeric digits 0 through 9 (ASCII codes 30h through 39h)

Note that the numeric digits differ from their numeric values only in the high order nibble. By subtracting 30h from the ASCII code for any particular digit you can obtain the numeric equivalent of that digit.

The third group of 32 ASCII characters is reserved for the upper case alphabetic characters.

The ASCII codes for the characters "A" through "Z" lie in the range 41h through 5Ah. Since there are only 26 different alphabetic characters, the remaining six codes hold various special symbols.

The fourth, and final, group of 32 ASCII character codes are reserved for the lower case alphabetic symbols, five additional special symbols, and another control character (delete).

Note that the lower case character symbols use the ASCII codes 61h through 7Ah. If you compare the ASCII codes for the upper and lower case characters to binary, you will notice that the upper case symbols differ from their lower case equivalents in exactly one bit position.

The only place these two codes differ is in bit five. Upper case characters always contain a zero in bit five; lower case alphabetic characters always contain a one in bit five. You can use this fact to quickly convert between upper and lower case. If you have an upper case character you can force it to lower case by setting bit five to one. If you have a lower case character and you wish to force it to upper case, you can do so by setting bit five to zero. You can toggle an alphabetic character between upper and lower case by simply inverting bit five.

The decimal representations of these ASCII codes are not very enlightening. However, the hexadecimal representation of these ASCII codes reveals something very important; the low order nibble of the ASCII code is the binary equivalent of the represented number.

By stripping away (i.e., setting to zero) the high order nibble of a numeric character, you can convert that character code to the corresponding binary representation. Conversely, you can convert a binary value in the range 0 through 9 to its ASCII character representation by simply setting the high order nibble to three. Note

that you can use the logical-AND operation to force the high order bits to zero; likewise, you can use the logical-OR operation to force the high order bits to 0011 (three).

Note that you cannot convert a string of numeric characters to their equivalent binary representation by simply stripping the high order nibble from each digit in the string. Converting 123 (31h 32h 33h) in this fashion yields three bytes: 010203h, not the correct value which is 7Bh. Converting a string of digits to an integer requires more sophistication than this; the conversion above works only for single digits.

Bit seven in standard ASCII is always zero. This means that the ASCII character set consumes only half of the possible character codes in an eight bit byte. The PC uses the remaining 128 character codes for various special characters including international characters (those with accents, etc.), math symbols, and line drawing characters. Note that these extra characters are a non-standard extension to the ASCII character set. Most printers support the PC's extended character set.

Should you need to exchange data with other machines which are not PC-compatible, you have only two alternatives: stick to standard ASCII or ensure that the target machine supports the extended IBM-PC character set. Some machines, like the Apple Macintosh, do not provide native support for the extended IBM-PC character set. However you may obtain a PC font which lets you display the extended character set. Other computers (e.g., Amiga and Atari ST) have similar capabilities. However, the 128 characters in the standard ASCII character set are the only ones you should count on transferring from system to system.

Despite the fact that it is a "standard", simply encoding your data using standard ASCII characters does not guarantee compatibility across systems. While it's true that an "A" on one machine is most likely an "A" on another machine, there is very little standardization across machines with respect to the use of the control characters. Indeed, of the 32 control codes plus delete, there are only four control codes commonly supported ^P; backspace (BS), tab, carriage return (CR), and line feed (LF). Worse still, different machines often use these control codes in different ways. End of line is a particularly troublesome example. MS-DOS, CP/M, and other systems mark end of line by the two-character sequence CR/LF. Apple Macintosh, Apple II, and many other systems mark the end of line by a single CR character. UNIX systems mark the end of a line with a single LF character. Needless to say, attempting to exchange simple text files between such systems can be an experience in frustration. Even if you use standard ASCII characters in all your files on these systems, you

will still need to convert the data when exchanging files between them. Fortunately, such conversions are rather simple.

Despite some major shortcomings, ASCII data is the standard for data interchange across computer systems and programs. Most programs can accept ASCII data; likewise most programs can produce ASCII data. If you will program in the assembly language you will be dealing with ASCII characters, and it would for that reason be wise to study the layout of the character set and memorize a few key ASCII codes (e.g., "0", "A", "a", etc.).

3.9 EBCDIC CODE

EBCDIC (Extended Binary Coded Decimal Information Code) is an eight-bit character set that was developed by International Business Machines (IBM). It was the character set used on most computers manufactured by IBM prior to 1981.

EBCDIC is not used on the IBM PC and all subsequent "PC clones". These computer systems use ASCII as the primary character and symbol coding system. (Computer makers other than IBM used the ASCII system since its inception in the 1960s.)

EBCDIC is widely considered to be an obsolete coding system, but is still used in some equipment, mainly in order to allow for continued use of software written many years ago that expects an EBCDIC communication environment.

The EBCDIC code assignments are shown in the following table.

Least Significant Bits																
Most Sig. Bits V	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0 0000	NUL (0) 00	SOH (1) 01	STX (2) 02	ETX (3) 03	PF (4) 04	HT (5) 05	LC (6) 06	DEL (7) 07	GE (8) 08	RLF (9) 09	SMM (10) 0A	VT (11) 0B	FF (12) 0C	CR (13) 0D	SO (14) 0E	SI (15) 0F
1 0001	DLE (16) 10	DC1 (17) 11	DC2 (18) 12	TM (19) 13	RES (20) 14	NL (21) 15	BS (22) 16	IL (23) 17	CAN (24) 18	EM (25) 19	CC (26) 1A	CUI (27) 1B	IFS (28) 1C	IGS (29) 1D	IRS (30) 1E	IUS (31) 1F
2 0010	DS (32) 20	SOS (33) 21	FS (34) 22		BYP (36) 24	LF (37) 25	ETB (38) 26	ESC (39) 27			SM (42) 2A	CU2 (43) 2B		ENQ (45) 2D	ACK (46) 2E	BEL (47) 2F
3 0011			SYN (50) 32		PN (52) 34	RS (53) 35	US (54) 36	EOT (55) 37				CU3 (59) 3A	DC4 (60) 3C	NAK (61) 3D		SUB (63) 3F
4 0100	SP (64) 40										¢ (74) 4A	. (75) 4B	< (76) 4C	((77) 4D	+ (78) 4E	Note1 (79) 4F

5 0101	& (80) 50	(81) 51	(82) 52	(83) 53	(84) 54	(85) 55	(86) 56	(87) 57	(88) 58	(89) 59	! (90) 5A	\$ (91) 5B	* (92) 5C) (93) 5D	; (94) 5E	~ (95) 5F
6 0110	- (96) 60	/ (97) 61	(98) 62	(99) 63	(100) 64	(101) 65	(102) 66	(103) 67	(104) 68	(105) 69	 (106) 6A	, (107) 6B	% (108) 6C	_ (109) 6D	> (110) 6E	? (111) 6F
7 0111	(112) 70	(113) 71	(114) 72	(115) 73	(116) 74	(117) 75	(118) 76	(119) 77	(120) 78	(121) 79	: (122) 7A	# (123) 7B	@ (124) 7C	' (125) 7D	= (126) 7E	" (127) 7F
8 1000	(128) 80	a (129) 81	b (130) 82	c (131) 83	d (132) 84	e (133) 85	f (134) 86	g (135) 87	h (136) 88	i (137) 89	(138) 8A	(139) 8B	(140) 8C	(141) 8D	(142) 8E	(143) 8F
9 1001	(144) 90	j (145) 91	k (146) 92	l (147) 93	m (148) 94	n (149) 95	o (150) 96	p (151) 97	q (152) 98	r (153) 99	(154) 9A	(155) 9B	(156) 9C	(157) 9D	(158) 9E	(159) 9F
A 1010	(160) A0	~ (161) A1	s (162) A2	t (163) A3	u (164) A4	v (165) A5	w (166) A6	x (167) A7	y (168) A8	z (169) A9	(170) AA	(171) AB	(172) AC	(173) AD	(174) AE	(175) AF
B 1011	(176) B0	(177) B1	(178) B2	(179) B3	(180) B4	(181) B5	(182) B6	(183) B7	(184) B8	(185) B9	(186) BA	(187) BB	(188) BC	(189) BD	(190) BE	(191) BF
C 1100	{ (192) C0	A (193) C1	B (194) C2	C (195) C3	D (196) C4	E (197) C5	F (198) C6	G (199) C7	H (200) C8	I (201) C9	(202) CA	(203) CB	Note2 (204) CC	(205) CD	Note3 (206) CE	(207) CF
D 1101	} (208) D0	J (209) D1	K (210) D2	L (211) D3	M (212) D4	N (213) D5	O (214) D6	P (215) D7	Q (216) D8	R (217) D9	(218) DA	(219) DB	(220) DC	(221) DD	(222) DE	(223) DF
E 1110	\ (224) E0	(225) E1	S (226) E2	T (227) E3	U (228) E4	V (229) E5	W (230) E6	X (231) E7	Y (232) E8	Z (233) E9	(234) EA	(235) EB	Note4 (236) EC	(237) ED	(238) EE	(239) EF
F 1111	0 (240) F0	1 (241) F1	2 (242) F2	3 (243) F3	4 (244) F4	5 (245) F5	6 (246) F6	7 (247) F7	8 (248) F8	9 (249) F9	(250) FA	(251) FB	(252) FC	(253) FD	(254) FE	(255) FF

The following EBCDIC characters have no equivalents in the ASCII or ISO-8859 character sets used on the Internet, and cannot be displayed in this table.

- (1) Code 79 is a solid vertical bar, similar to the broken vertical bar (character 106).
- (2) Code 204 is the mathematics integration symbol.
- (3) Code 206 is a "Y" drawn with only right angles.
- (4) Code 236 is a horizontally-flipped "h".

In this table, the code or symbol name is shown on the first line, followed by the decimal value for that code or symbol, followed by the hexadecimal value. The binary value can be computed based on the row and column where the code or symbol resides, or directly from the hexadecimal value. For example, the character "+" has the binary value "0100 1110", with "0100" taken from the row and "1110" taken from the column. Similarly, the lowercase letter 'p' has the binary value "1001 0111".

3.10 UNICODE

Fundamentally, computers just deal with numbers. They store letters and other characters by assigning a number for each one. Before Unicode was invented, there were hundreds of different encoding systems for assigning these numbers. No single encoding could contain enough characters: for example, the European Union alone requires several different encodings to cover all its languages. Even for a single language like English no single encoding was adequate for all the letters, punctuation, and technical symbols in common use.

These encoding systems also conflict with one another. That is, two encodings can use the same number for two *different* characters, or use different numbers for the *same* character. Any given computer (especially servers) needs to support many different encodings; yet whenever data is passed between different encodings or platforms, that data always runs the risk of corruption.

Unicode is changing all that!

Unicode provides a unique number for every character, no matter what the platform, no matter what the program, no matter what the language. The Unicode Standard has been adopted by such industry leaders as Apple, HP, IBM, JustSystems, Microsoft, Oracle, SAP, Sun, Sybase, Unisys and many others. Unicode is required by modern standards such as XML, Java, ECMAScript (JavaScript), LDAP, CORBA 3.0, WML, etc., and is the official way to implement ISO/IEC 10646. It is supported in many operating systems, all modern browsers, and many other products. The emergence of the Unicode Standard, and the availability of tools supporting it, are among the most significant recent global software technology trends.

Incorporating Unicode into client-server or multi-tiered applications and websites offers significant cost savings over the use of legacy character sets. Unicode enables a single software product or a single website to be targeted across multiple platforms, languages and countries without re-engineering. It allows data to be transported through many different systems without corruption.

3.11 QUESTIONS

1. What are binary codes? Why are they used?
2. Explain the error detecting codes and how are they used with examples.
3. Explain the error correcting codes and how are they used with examples.

4. Explain geometric representation of binary numbers.
5. What is ASCII code? Explain.
6. Explain UNICODE.

3.12 FURTHER READING

- ❖ **Digital Electronics** - An Introduction to Theory and Practice by W H Gothmann
- ❖ **Computer Architecture and Parallel Processing** by Kai Hwang, Faye A Briggs , McGraw Hill
- ❖ **Computer Architecture and Organization** by William Stallings
- ❖ **Fundamentals of Computer organization and Design** by Sivarama P. Dandamudi

- ❖ http://en.wikipedia.org/wiki/Binary_codes
- ❖ <http://en.wikipedia.org/wiki/Distance>
- ❖ http://en.wikipedia.org/wiki/Error_detection
- ❖ <http://en.wikipedia.org/wiki/ASCII>



BOOLEAN ALGEBRA AND LOGIC GATES

Unit Structure

- 3.0: Objectives
- 3.1 Boolean Logic
- 3.2 Logic Gates
 - 3.2.1 Not gate
 - 3.2.2 The "buffer" gate
 - 3.2.3 The AND gate
 - 3.2.4 The OR gate
- 3.3 Universal Gates
 - 3.3.1 Nand gate
 - 3.3.2 Nor gate
- 3.4 Other Gates
 - 3.4.1 The Exclusive-Or gate
 - 3.4.2 The Exclusive-Nor gate
- 3.5 Boolean algebra
- 3.6 Laws of boolean algebra
- 3.7 Questions
- 3.8 Further reading

3.0 OBJECTIVES:

After completing this chapter, you will be able to:

- ❖ Understand the concept of Boolean Logic
- ❖ Learn the concept of Logic gates with the help of Diagrams.
- ❖ Understanding the Universal Gates and their circuit implications.
- ❖ Learn about Exclusive OR & NOR gates.
- ❖ Understand the Boolean algebra and laws of Boolean Algebra for use in implementation.

3.1 BOOLEAN LOGIC:

Boolean logic is named after a 19th Century English clergyman, George Boole, who was also a keen amateur

mathematician and formalised the mathematics of a logic based on a two-valued (TRUE and FALSE) system.

In Boolean logic any variable can have one of only two possible values, TRUE or FALSE. In some systems these may be called HIGH and LOW or perhaps ONE and ZERO, but whatever the names there are only two possible values. There is no such thing as "in between" or "it has no value". If a variable is not TRUE, then it must be FALSE.

3.2 LOGIC GATES:

Practical devices which obey the laws of Boolean logic can be made in a variety of different forms. In the earlier part of the Century, relays were used extensively for simple logic systems such as the controllers for lifts (elevators) and traffic lights. Victorian signal boxes used mechanical levers and rods for operating points and signals which had logical interlocks, and in some hazardous environments where the use of electricity is avoided, fluid logic circuits are used. In this course we are interested in electronic logic based on the modern, silicon, integrated-circuit technology.

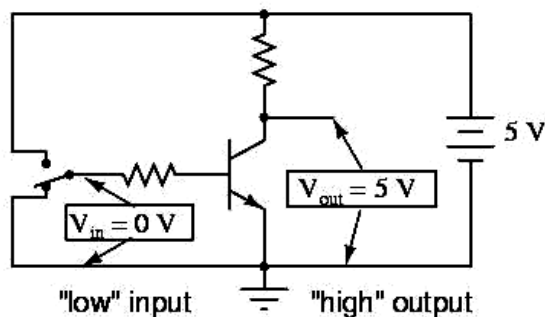
In electronic logic, the value of a logic signal on an input or output is usually defined by the voltage. If the voltage is above a certain value, it is a logic ONE. If it is below that value it is a logic ZERO. Circuits are produced (by the million!) which have a (Boolean) output value, or values, which are directly determined by the (Boolean) values on the inputs. In these notes a logical value of ONE or TRUE will often be written as '1' and a value which is ZERO or FALSE written as '0'.

3.2.1 NOT gate

While the binary numeration system is an interesting mathematical abstraction, we haven't yet seen its practical application to electronics. This chapter is devoted to just that: practically applying the concept of binary bits to circuits. What makes binary numeration so important to the application of digital electronics is the ease in which bits may be represented in physical terms. Because a binary bit can only have one of two different values, either 0 or 1, any physical medium capable of switching between two saturated states may be used to represent a bit. Consequently, any physical system capable of representing binary bits is able to represent numerical quantities, and potentially has the ability to manipulate those numbers. This is the basic concept underlying digital computing.

Electronic circuits are physical systems that lend themselves well to the representation of binary numbers. Transistors, when operated at their bias limits, may be in one of two different states: either cutoff (no controlled current) or saturation (maximum controlled current). If a transistor circuit is designed to maximize the probability of falling into either one of these states (and not operating in the linear, or *active*, mode), it can serve as a physical representation of a binary bit. A voltage signal measured at the output of such a circuit may also serve as a representation of a single bit, a low voltage representing a binary "0" and a (relatively) high voltage representing a binary "1." Note the following transistor circuit:

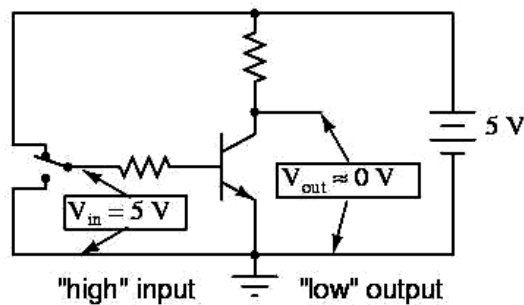
Transistor in cutoff



0 V = "low" logic level (0)

5 V = "high" logic level (1)

Transistor in saturation



0 V = "low" logic level (0)

5 V = "high" logic level (1)

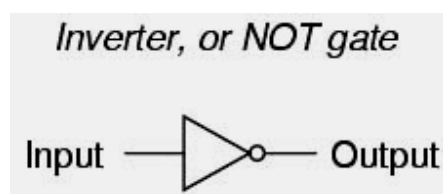
In this circuit, the transistor is in a state of saturation by virtue of the applied input voltage (5 volts) through the two-position switch. Because it's saturated, the transistor drops very little voltage between collector and emitter, resulting in an output voltage of (practically) 0 volts. If we were using this circuit to represent binary bits, we would say that the input signal is a binary "1" and that the

output signal is a binary "0." Any voltage close to full supply voltage (measured in reference to ground, of course) is considered a "1" and a lack of voltage is considered a "0." Alternative terms for these voltage levels are *high* (same as a binary "1") and *low* (same as a binary "0"). A general term for the representation of a binary bit by a circuit voltage is *logic level*.

Moving the switch to the other position, we apply a binary "0" to the input and receive a binary "1" at the output:

What we've created here with a single transistor is a circuit generally known as a *logic gate*, or simply *gate*. A gate is a Special type of amplifier circuit designed to accept and generate voltage signals corresponding to binary 1's and 0's. As such, gates are not intended to be used for amplifying analog signals (voltage signals *between* 0 and full voltage). Used together, multiple gates may be applied to the task of binary number storage (memory circuits) or manipulation (computing circuits), each gate's output representing one bit of a multi-bit binary number. Just how this is done is a subject for a later chapter. Right now it is important to focus on the operation of individual gates.

The gate shown here with the single transistor is known as an *inverter*, or NOT gate, because it outputs the exact opposite digital signal as what is input. For convenience, gate circuits are generally represented by their own symbols rather than by their constituent transistors and resistors. The following is the symbol for an inverter:

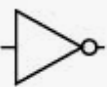


An alternative symbol for an inverter is shown here:



One common way to express the particular function of a gate circuit is called a *truth table*. Truth tables show all combinations of input conditions in terms of logic level states (either "high" or "low," "1" or "0," for each input terminal of the gate), along with the corresponding output logic level, either "high" or "low." For the inverter, or NOT, circuit just illustrated, the truth table is very simple indeed:

NOT gate truth table

Input  Output

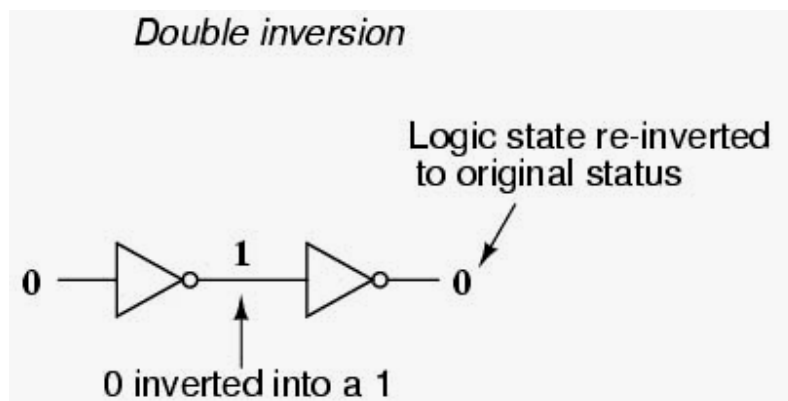
Input	Output
0	1
1	0

Equation for NOT gate

$$Y = \bar{A}$$

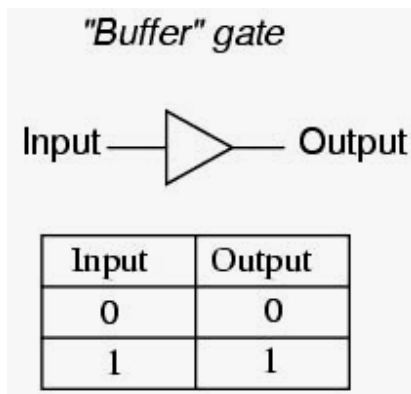
3.2.2 The "buffer" gate

If we were to connect two inverter gates together so that the output of one fed into the input of another, the two inversion functions would "cancel" each other out so that there would be no inversion from input to final output:



While this may seem like a pointless thing to do, it does have practical application. Remember that gate circuits are signal amplifiers, regardless of what logic function they may perform. A weak signal source (one that is not capable of sourcing or sinking very much current to a load) may be boosted by means of two inverters like the pair shown in the previous illustration. The logic level is unchanged, but the full current-sourcing or -sinking capabilities of the final inverter are available to drive a load resistance if needed.

For this purpose, a special logic gate called a buffer is manufactured to perform the same function as two inverters. Its symbol is simply a triangle, with no inverting "bubble" on the output terminal:



Equation for Buffer gate

$$Y = A$$

Inverters and buffers exhaust the possibilities for single-input gate circuits. What more can be done with a single logic signal but to buffer it or invert it? To explore more logic gate possibilities, we must add more input terminals to the circuit(s).

Adding more input terminals to a logic gate increases the number of input state possibilities. With a single-input gate such as the inverter or buffer, there can only be two possible input states: either the input is "high" (1) or it is "low" (0). As was mentioned previously in this chapter, a two input gate has *four* possibilities (00, 01, 10, and 11). A three-input gate has *eight* possibilities (000, 001, 010, 011, 100, 101, 110, and 111) for input states. The number of possible input states is equal to two to the power of the number of inputs:

$$\text{Number of possible input states} = 2^n$$

Where,

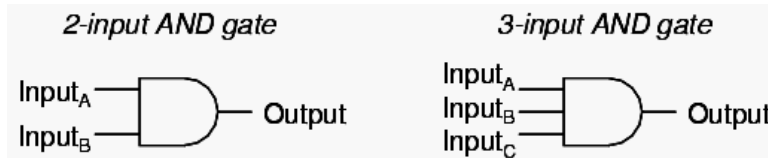
$$n = \text{Number of inputs}$$

This increase in the number of possible input states obviously allows for more complex gate behavior. Now, instead of merely inverting or amplifying (buffering) a single "high" or "low" logic level, the output of the gate will be determined by whatever *combination* of 1's and 0's is present at the input terminals.

Since so many combinations are possible with just a few input terminals, there are many different types of multiple-input gates, unlike single-input gates which can only be inverters or buffers.

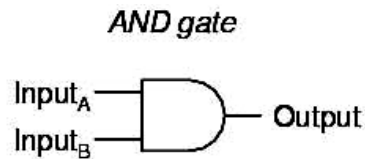
3.2.3 The AND gate

One of the easiest multiple-input gates to understand is the AND gate, so-called because the output of this gate will be "high" (1) if and only if all inputs (first input and the second input and . . .) are "high" (1). If any input(s) are "low" (0), the output is guaranteed to be in a "low" state as well.



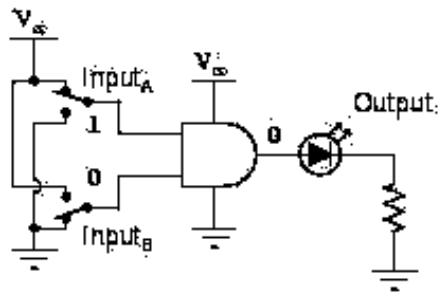
In case you might have been wondering, AND gates are made with more than three inputs, but this is less common than the simple two-input variety.

The logic diagram and the truth table of AND gate are shown below:

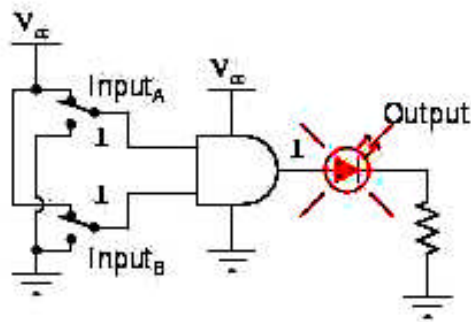


A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

What this truth table means in practical terms is shown in the following sequence of illustrations, with the 2-input AND gate subjected to all possibilities of input logic levels. An LED (Light-Emitting Diode) provides visual indication of the output logic level:



Input_A = 1
 Input_B = 0
 Output = 0 (no light)

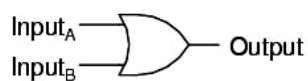


Input_A = 1
 Input_B = 1
 Output = 1 (light!)

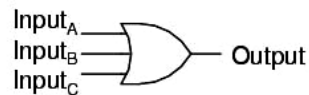
3.2.4 The OR gate

Our next gate to investigate is the OR gate, so-called because the output of this gate will be "high" (1) if any of the inputs (first input or the second input or . . .) are "high" (1). The output of an OR gate goes "low" (0) if and only if all inputs are "low" (0).

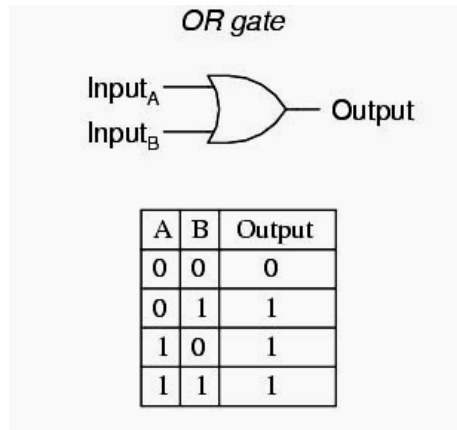
2-input OR gate



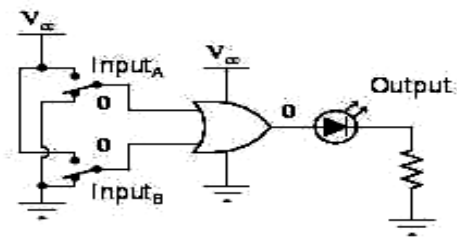
3-input OR gate



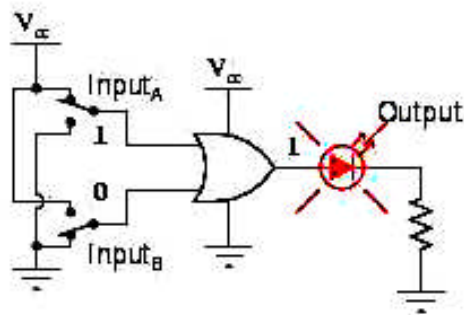
The logic diagram and the truth table of OR gate are shown below:



The following sequence of illustrations demonstrates the OR gate's function, with the 2-inputs experiencing all possible logic levels. An LED (Light-Emitting Diode) provides visual indication of the gate's output logic level:



Input_A = 0
 Input_B = 0
 Output = 0 (no light)



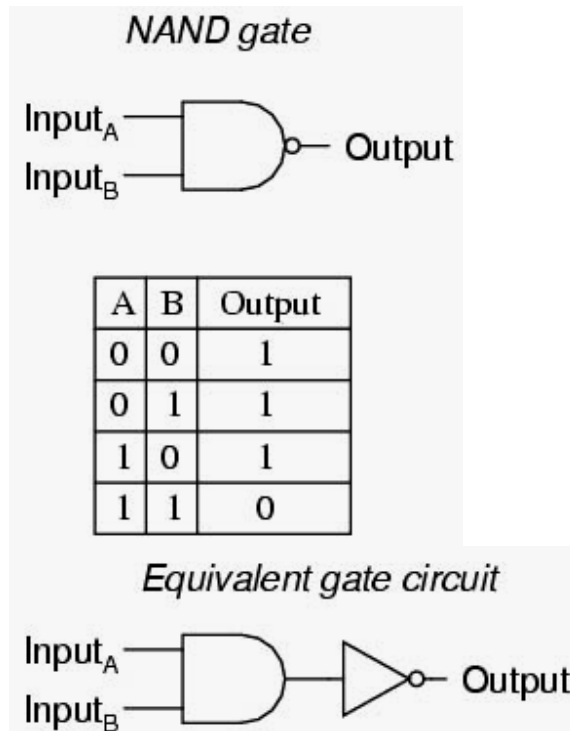
Input_A = 1
 Input_B = 0
 Output = 1 (light!)

3.3 UNIVERSAL GATES

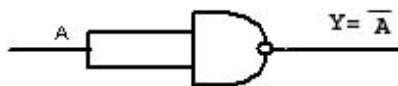
NOT, AND and OR gates are called as basic gates. The NAND and NOR gates are called as Universal gates as any gate can be derived using these gates.

3.3.1 NAND gate:

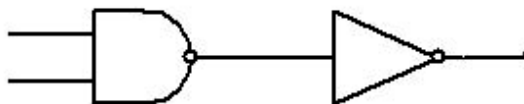
A variation on the idea of the AND gate is called the NAND gate. The word "NAND" is a verbal contraction of the words NOT and AND. Essentially, a NAND gate behaves the same as an AND gate with a NOT (inverter) gate connected to the output terminal. To symbolize this output signal inversion, the NAND gate symbol has a bubble on the output line. The truth table for a NAND gate is as one might expect, exactly opposite as that of an AND gate:



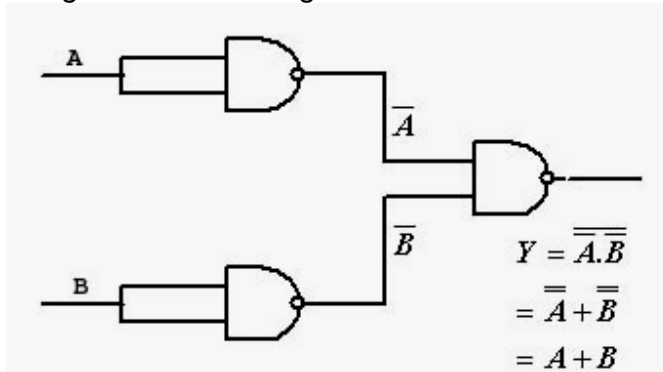
NOT gate from NAND gate:



AND gate from NAND gate:



OR gate from NAND gate:

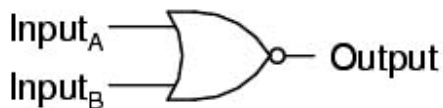


3.3.2 NOR gate:

As you might have guessed, the NOR gate is an OR gate with its output inverted, just like a NAND gate is an AND gate with an inverted output.

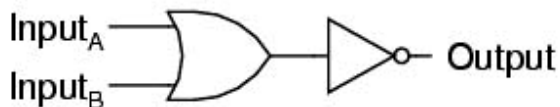
The NOR gate has output high when both of its inputs are low. The symbolic diagram and the truth table of NOR gate is shown below:

NOR gate



A	B	Output
0	0	1
0	1	0
1	0	0
1	1	0

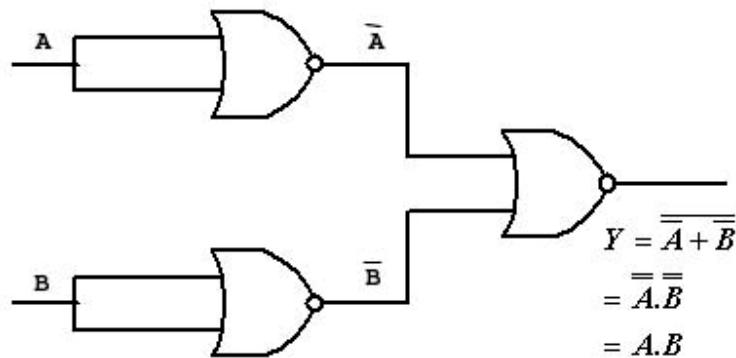
Equivalent gate circuit



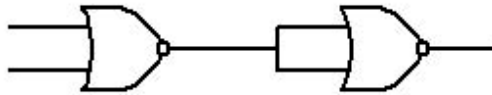
NOT gate from NOR gate:



AND gate from NOR gate:



OR gate from NOR gate:



So we have seen that all the basic gates can be constructed using NAND and NOR gates and hence they are called Universal gates.

3.4 OTHER GATES

3.4.1 The Exclusive-OR gate

Exclusive-OR gates output a "high" (1) logic level if the inputs are at different logic levels, either 0 and 1 or 1 and 0. Conversely, they output a "low" (0) logic level if the inputs are at the same logic levels. The Exclusive-OR (sometimes called XOR) gate has both a symbol and a truth table pattern that is unique:

Exclusive-OR gate

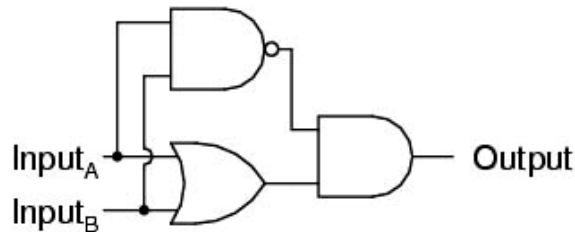


A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

There are equivalent circuits for an Exclusive-OR gate made up of AND, OR, and NOT gates, just as there were for NAND, NOR, and the negative-input gates. A rather direct approach to

simulating an Exclusive-OR gate is to start with a regular OR gate, then add additional gates to inhibit the output from going "high" (1) when both inputs are "high" (1):

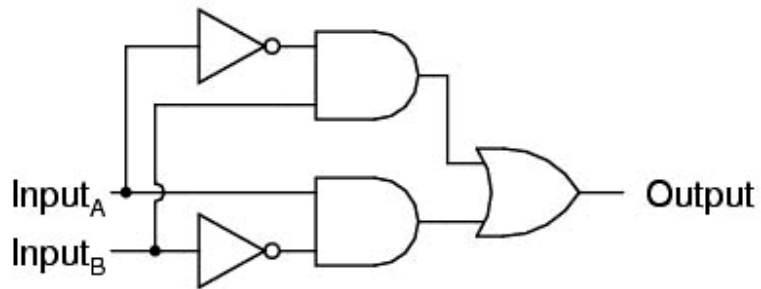
Exclusive-OR equivalent circuit



A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

In this circuit, the final AND gate acts as a buffer for the output of the OR gate whenever the NAND gate's output is high, which it is for the first three input state combinations (00, 01, and 10). However, when both inputs are "high" (1), the NAND gate outputs a "low" (0) logic level, which forces the final AND gate to produce a "low" (0) output.

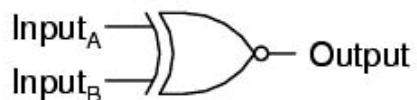
Another equivalent circuit for the Exclusive-OR gate uses a strategy of two AND gates with inverters, set up to generate "high" (1) outputs for input conditions 01 and 10. A final OR gate then allows either of the AND gates' "high" outputs to create a final "high" output:

Exclusive-OR equivalent circuit

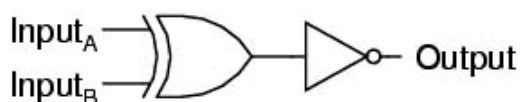
A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

3.4.2 The Exclusive-NOR gate

Finally, our last gate for analysis is the Exclusive-NOR gate, otherwise known as the XNOR gate. It is equivalent to an Exclusive-OR gate with an inverted output. The truth table for this gate is exactly opposite as for the Exclusive-OR gate:

Exclusive-NOR gate

A	B	Output
0	0	1
0	1	0
1	0	0
1	1	1

Equivalent gate circuit

3.5 BOOLEAN ALGEBRA

The most obvious way to simplify Boolean expressions is to manipulate them in the same way as normal algebraic expressions are manipulated. With regards to logic relations in digital forms, a set of rules for symbolic manipulation is needed in order to solve for the unknowns.

A set of rules formulated by the English mathematician *George Boole* describe certain propositions whose outcome would be either *true* or *false*. With regard to digital logic, these rules are used to describe circuits whose state can be either, *1 (true)* or *0 (false)*. In order to fully understand this, the relation between the AND gate, OR gate and NOT gate operations should be appreciated. A number of rules can be derived from these relations as the following Table demonstrates.

- P1: $X = 0$ or $X = 1$
- P2: $0 \cdot 0 = 0$
- P3: $1 + 1 = 1$
- P4: $0 + 0 = 0$
- P5: $1 \cdot 1 = 1$
- P6: $1 \cdot 0 = 0 \cdot 1 = 0$
- P7: $1 + 0 = 0 + 1 = 1$

3.6 LAWS OF BOOLEAN ALGEBRA

[The](#) following table shows the basic Boolean laws. Note that every law has two expressions, (a) and (b). This is known as *duality*. These are obtained by changing every AND(.) to OR(+), every OR(+) to AND(.) and all 1's to 0's and vice-versa. It has become conventional to drop the . (AND symbol) i.e. A.B is written as AB.

T1 : Commutative Law

(a) $A + B = B + A$

(b) $A B = B A$

T2 : Associate Law

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

(b) $(A B) C = A (B C)$

T3 : Distributive Law

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

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

T4 : Identity Law

(a) $A + A = A$

(b) $A A = A$

T5 :

- (a) $AB + A\bar{B} = A$
 (b) $(A+B)(A+\bar{B}) = A$

T6 : Redundance Law

- (a) $A + AB = A$
 (b) $A(A + B) = A$

T7 :

- (a) $0 + A = A$
 (b) $0A = 0$

T8 :

- (a) $1 + A = 1$
 (b) $1A = A$

T9 :

- (a) $\bar{A} + A = 1$
 (b) $\bar{A}A = 0$

T10 :

- (a) $A + \bar{A}B = A + B$
 (b) $A(\bar{A} + B) = AB$

T11 : De Morgan's Theorem

- (a) $\overline{(A+B)} = \bar{A}\bar{B}$
 (b) $\overline{AB} = \bar{A} + \bar{B}$

Examples:

$$ab + ab' + a'b = a(b+b') + a'b$$

$$= a \cdot 1 + a'b \text{ By P5}$$

$$= a + a'b \text{ By}$$

$$= a + a'b + 0$$

$$= a + a'b + aa'$$

$$= a + b(a + a')$$

$$= a + b \cdot 1$$

$$= a + b$$

$$(a'b + a'b' + b')' = (a'(b+b') + b')'$$

$$= (a' + b')'$$

$$= ((ab)')'$$

$$= ab$$

$$b(a+c) + ab' + bc' + c = ba + bc + ab' + bc' + c$$

$$= a(b+b') + b(c + c') + c$$

$$= a \cdot 1 + b \cdot 1 + c$$

$$= a + b + c$$

3.7 QUESTIONS:

1. What are basic gates? Explain.
2. What are universal gates? Why are they so called?
3. Explain XOR and XNOR gates.
4. Construct the following gates from universal (both NAND and NOR) gates:
 - a. AND GATE
 - b. OR GATE
 - c. NOT GATE
 - d. XOR GATE
 - e. XNOR GATE
5. State and prove De' Morgans Laws.
6. State and prove the laws of Boolean algebra.
7. Prove the following using Laws of Boolean Algebra:
 - i. $(A + \overline{AB})(A + \overline{AB})(CD + \overline{CDA} + \overline{CD} + \overline{CDA}) = A$
 - ii. $\overline{XYZ} + \overline{XYZ} + \overline{XYZ} + \overline{XYZ} = \overline{XY} + \overline{Z}$
 - iii. $(A + \overline{BC})(\overline{AB} + \overline{ABC}) = \overline{ABC}$
 - iv. $(\overline{AB} + \overline{ABC})\overline{ABC} = 0$

3.8 FURTHER READING:

- ❖ **Digital Electronics** - An Introduction to Theory and Practice by W H Gothmann
- ❖ **Computer Architecture and Parallel Processing** by Kai Hwang, Faye A Briggs , McGraw Hill
- ❖ **Computer Architecture and Organization** by William Stallings
- ❖ **Fundamentals of Computer organization and Design** by Sivarama P. Dandamudi
- ❖ http://en.wikipedia.org/wiki/Logic_gates
- ❖ http://en.wikipedia.org/wiki/Universal_logic_gate
- ❖ http://en.wikipedia.org/wiki/Boolean_algebra



CANONICAL FORMS AND KARNAUGH MAPS

Unit Structure

- 4.0 Objectives
- 4.1 Canonical Forms
- 4.2 Karnaugh Maps
- 4.3 Simplifying Boolean Expressions Using Karnaugh Maps
- 4.4 Maxterms and Minterms
- 4.5 Quine McClusky Method
- 4.6 Questions
- 4.7 Further Reading

4.0 OBJECTIVES

After completing this chapter, you will be able to:

- ❖ Learn about the CANONICAL FORMS.
- ❖ Understand the building and working of KARNAUGH MAP.
- ❖ Using the KARNAUGH MAP for solving the Boolean expression.
- ❖ Understanding the concept of Maxterms and Minterms
- ❖ Learn the Quine McClusky Method for finding a minimum-cost sum-of-products.

4.1 CANONICAL FORMS

Since there are a finite number of boolean functions of n input variables, yet an infinite number of possible logic expressions you can construct with those n input values, clearly there are an infinite number of logic expressions that are equivalent (i.e., they produce the same result given the same inputs). To help eliminate possible confusion, logic designers generally specify a boolean function using a *canonical*, or standardized, form. For any given boolean function there exists a unique canonical form. This eliminates some confusion when dealing with boolean functions.

Actually, there are several different canonical forms. We will discuss only two here and employ only the first of the two. The first is the so-called *sum of minterms* and the second is the *product of maxterms*. Using the duality principle, it is very easy to convert between these two.

A *term* is a variable or a product (logical AND) of several different literals. For example, if you have two variables, A and B, there are eight possible terms: A, B, A', B', A'B', A'B, AB', and AB. For three variables we have 26 different terms: A, B, C, A', B', C', A'B', A'B, AB', AB, A'C', A'C, AC', AC, B'C', B'C, BC', BC, A'B'C', AB'C', A'BC', ABC', A'B'C, AB'C, A'BC, and ABC. As you can see, as the number of variables increases, the number of terms increases dramatically. A *minterm* is a product containing exactly n literals. For example, the minterms for two variables are A'B', AB', A'B, and AB. Likewise, the minterms for three variables A, B, and C are A'B'C', AB'C', A'BC', ABC', A'B'C, AB'C, A'BC, and ABC. In general, there are 2^n minterms for n variables. The set of possible minterms is very easy to generate since they correspond to the sequence of binary numbers:

Binary Equivalent (CBA)	Minterm
000	A'B'C'
001	AB'C'
010	A'BC'
011	ABC'
100	A'B'C
101	AB'C
110	A'BC
111	ABC

4.2 KARNAUGH MAPS

A **Karnaugh map** comprises a box for every line in the truth table; the binary value for each box is the binary value of the input terms in the corresponding table row. Unlike a truth table, in which the input values typically follow a standard binary sequence (00, 01, 10, 11), the Karnaugh map's input values must be ordered such that the values for adjacent columns vary by only a single bit, for example, 00, 01, 11, and 10. This ordering is known as a *Gray code*.

We use a **Karnaugh map** to obtain the simplest possible Boolean expression that describes a truth table.

Each row in the table (or minterm) is equivalent to a cell on the Karnaugh Map.

Example #1:

Here is a two-input truth table for a digital circuit:

Row	Inputs		Output
	A	B	
Row # 0	0	0	0
Row # 1	0	1	1
Row # 2	1	0	1
Row # 3	1	1	1

The corresponding K-map is

		B	
		0	1
A	0	Row # 0: 0	Row # 1: 1
	1	Row # 2: 1	Row # 3: 1

Example #2:

Here is a three-input truth table for a digital

Row	Inputs			Output
	A	B	C	
Row # 0	0	0	0	0
Row # 1	0	0	1	1
Row # 2	0	1	0	1
Row # 3	0	1	1	1
Row # 4	1	0	0	1
Row # 5	1	0	1	1
Row # 6	1	1	0	0
Row # 7	1	1	1	1

The corresponding K-map is

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

Example #3:

Here is a four-input truth table for a digital circuit:

Row	Inputs				Output
	A	B	C	D	F
Row # 0	0	0	0	0	0
Row # 1	0	0	0	1	1
Row # 2	0	0	1	0	1
Row # 3	0	0	1	1	1
Row # 4	0	1	0	0	1
Row # 5	0	1	0	1	1
Row # 6	0	1	1	0	0
Row # 7	0	1	1	1	1
Row # 8	1	0	0	0	1
Row # 9	1	0	0	1	0
Row # 10	1	0	1	0	1
Row # 11	1	0	1	1	1
Row # 12	1	1	0	0	1
Row # 13	1	1	0	1	1
Row # 14	1	1	1	0	1
Row # 15	1	1	1	1	0

The corresponding K-map is

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

4.3 SIMPLIFYING BOOLEAN EXPRESSIONS USING KARNAUGH MAP

To simplify the resulting Boolean expression using a Karnaugh map adjacent cells containing one are looped together. This step eliminated any terms of the form AA .

Adjacent cells means:

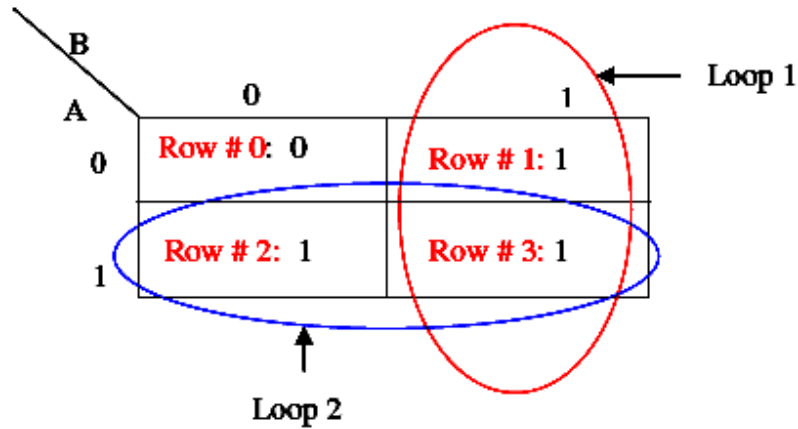
1. Cells that are side by side in the horizontal and vertical directions (but not diagonal).
2. For a map row: the leftmost cell and the rightmost cell.
3. For a map column: the topmost cell and the bottom most cell.
4. For a 4 variable map: cells occupying the four corners of the map.

Cells may only be looped together in twos, fours, or eights. As few groups as possible must be formed. Groups may overlap one another and may contain only one cell.

The larger the number of 1s looped together in a group the simpler is the product term that the group represents.

Example #1:

Simplifying the corresponding K-map of a two-input truth table for a digital circuit:



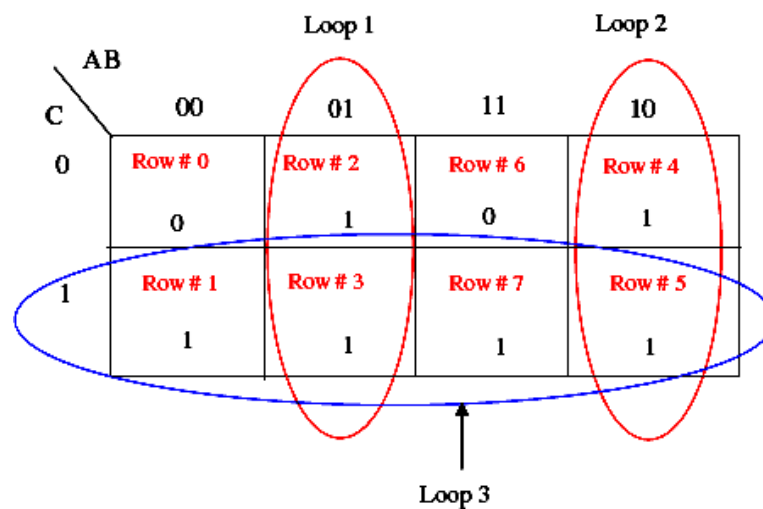
In Loop 1 the variable A has both logic 0 and logic 1 values in the same loop. B has a value of 1. Hence minterm equation is: $F = B$.

In Loop 2 Variable B has both logic 0 and 1 values in the same loop. $A = 1$, hence minterm equation is: $F = A$.

The overall Boolean expression for F is therefore: $F = A + B$

Example #2:

Simplifying the corresponding K-map of a three-input truth table for a digital circuit:



In Loop 1 the variable C has both logic 0 and logic 1 values in the same loop. A has a value of 0 and B has a logic value of 1. Hence minterm equation is: $F = \bar{A}B$

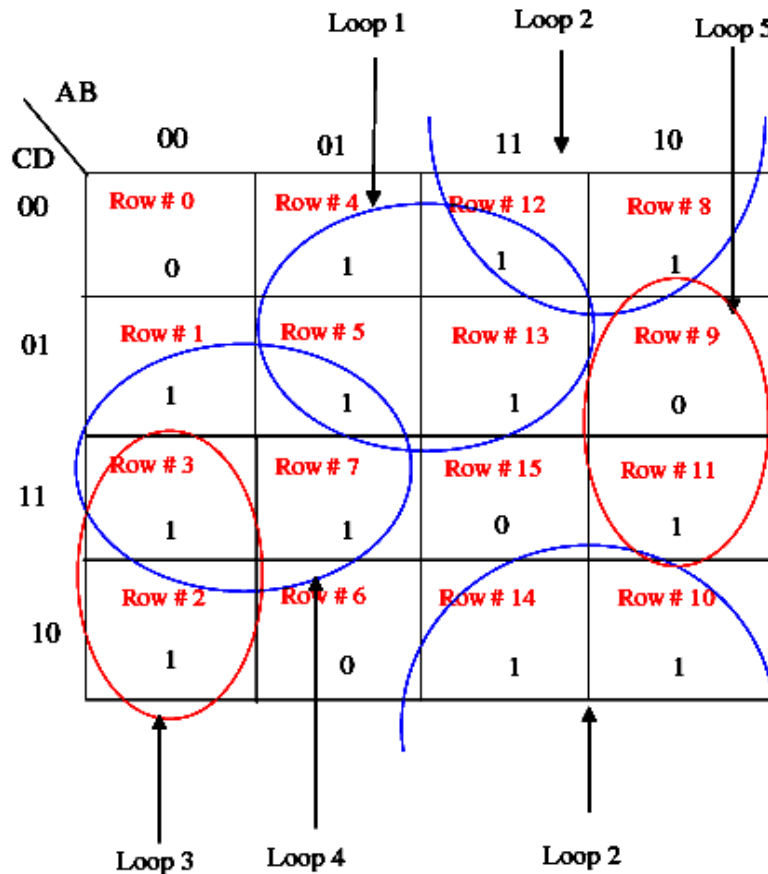
In Loop 2 the variable C has both logic 0 and 1 values in the same loop. $A = 1$ and $B = 0$, hence minterm equation is: $F = A\bar{B}$

In Loop 3 the two variables A and B both have logic 0 and logic 1 values in the same loop. C has a value of 1. Hence minterm equation is: $F = C$.

The overall Boolean expression for F is therefore: $F = \bar{A}B + A\bar{B} + C$

Example #3:

Simplifying the corresponding K-map of a four-input truth table for a digital circuit:



In Loop 1 the two variables A and D both have logic 0 and logic 1 values in the same loop. C has a value of 0 and B has a value of 1. Hence minterm equation is: $F = \bar{A}B$

In Loop 2 the two variables B and C both have logic 0 and logic 1 values in the same loop. A has a value of 1 and D has a value of 0. Hence minterm equation is: $F = A\bar{D}$.

In Loop 3 the variable D has logic 0 and logic 1 values in the same loop. A and B both have a value of 0 and C has a value of 1. Hence minterm equation is: $F = \bar{A}\bar{B}C$.

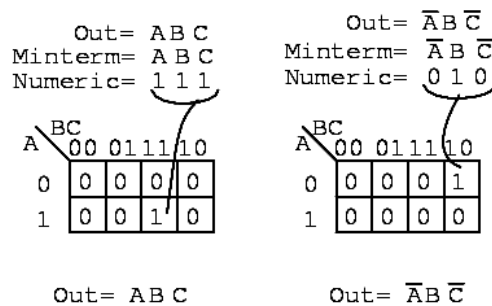
In Loop 4 the two variables B and C both have logic 0 and logic 1 values in the same loop. A has a value of 0 and D has a value of 1. Hence minterm equation is: $F = \bar{A}D$.

In Loop 5 the variable C has logic 0 and logic 1 values in the same loop. A and D both have a value of 1 and B has a value of 0. Hence minterm equation is: $F = A\bar{B}D$.

The overall Boolean expression for F is therefore: $F = A\bar{D} + \bar{A}\bar{B}C + \bar{A}D + A\bar{B}D$.

4.4 MAXTERMS AND MINTERMS

Sum-Of-Product (SOP) and Product-Of-Sums solution (POS):



A *minterm* is a Boolean expression resulting in 1 for the output of a single cell, and 0s for all other cells in a Karnaugh map, or truth table. If a minterm has a single 1 and the remaining cells as 0s, it would appear to cover a minimum area of 1s. The illustration above left shows the minterm ABC , a single product term, as a single 1 in a map that is otherwise 0s. We have not shown the 0s in our Karnaugh maps up to this point, as it is customary to omit them unless specifically needed. Another minterm $A'BC'$ is shown above right. The point to review is that the address of the cell corresponds directly to the minterm being mapped. That is, the cell 111 corresponds to the minterm ABC above left. Above right we see that the minterm $A'BC'$ corresponds directly to the cell 010. A Boolean expression or map may have multiple minterms.

Referring to the above figure, Let's summarize the procedure for placing a minterm in a K-map:

- Identify the minterm (product term) term to be mapped.
- Write the corresponding binary numeric value.
- Use binary value as an address to place a **1** in the K-map
- Repeat steps for other minterms (P-terms within a Sum-Of-Products).

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

	BC			
A	00	01	11	10
0	0	0	0	1
1	0	0	1	0

Numeric = $\underline{010}$ $\underline{111}$
 Minterm = $\bar{A}\bar{B}\bar{C}$ ABC
 $\text{Out} = \bar{A}\bar{B}\bar{C} + ABC$

A Boolean expression will more often than not consist of multiple minterms corresponding to multiple cells in a Karnaugh map as shown above. The multiple minterms in this map are the individual minterms which we examined in the previous figure above. The point we review for reference is that the **1s** come out of the K-map as a binary cell address which converts directly to one or more product terms. By directly we mean that a **0** corresponds to a complemented variable, and a **1** corresponds to a true variable. Example: **010** converts directly to **A'BC'**. There was no reduction in this example. Though, we do have a Sum-Of-Products result from the minterms.

Referring to the above figure, Let's summarize the procedure for writing the Sum-Of-Products reduced Boolean equation from a K-map:

- Form largest groups of **1s** possible covering all minterms. Groups must be a power of 2.
- Write binary numeric value for groups.
- Convert binary value to a product term.
- Repeat steps for other groups. Each group yields a p-terms within a Sum-Of-Products.

Nothing new so far, a formal procedure has been written down for dealing with minterms. This serves as a pattern for dealing with maxterms.

Next we attack the Boolean function which is **0** for a single cell and **1s** for all others.

$$\begin{aligned}
 \text{Out} &= (A+B+C) \\
 \text{Maxterm} &= A+B+C \\
 \text{Numeric} &= 1\ 1\ 1 \\
 \text{Complement} &= 0\ 0\ 0
 \end{aligned}$$

A	BC	00	01	11	10
0		0	1	1	1
1		1	1	1	1

A *maxterm* is a Boolean expression resulting in a **0** for the output of a single cell expression, and **1**s for all other cells in the Karnaugh map, or truth table. The illustration above left shows the maxterm $(A+B+C)$, a single sum term, as a single **0** in a map that is otherwise **1**s. If a maxterm has a single **0** and the remaining cells as **1**s, it would appear to cover a maximum area of **1**s.

There are some differences now that we are dealing with something new, maxterms. The maxterm is a **0**, not a **1** in the Karnaugh map. A maxterm is a sum term, $(A+B+C)$ in our example, not a product term.

It also looks strange that $(A+B+C)$ is mapped into the cell **000**. For the equation $\text{Out}=(A+B+C)=0$, all three variables (**A**, **B**, **C**) must individually be equal to **0**. Only $(0+0+0)=0$ will equal **0**. Thus we place our sole **0** for minterm $(A+B+C)$ in cell **A,B,C=000** in the K-map, where the inputs are all **0**. This is the only case which will give us a **0** for our maxterm. All other cells contain **1**s because any input values other than $((0,0,0)$ for $(A+B+C)$ yields **1**s upon evaluation.

Referring to the above figure, the procedure for placing a maxterm in the K-map is:

- Identify the Sum term to be mapped.
- Write corresponding binary numeric value.
- Form the complement
- Use the complement as an address to place a **0** in the K-map
- Repeat for other maxterms (Sum terms within Product-of-Sums expression).

$$\begin{aligned} \text{Out} &= (\overline{A} + \overline{B} + \overline{C}) \\ \text{Maxterm} &= \overline{A} + \overline{B} + \overline{C} \\ \text{Numeric} &= 0 \ 0 \ 0 \\ \text{Complement} &= 1 \ 1 \ 1 \end{aligned}$$

	BC	00	01	11	10
A	0	1	1	1	1
	1	1	1	0	1

Another maxterm $A'+B'+C'$ is shown above. Numeric **000** corresponds to $A'+B'+C'$. The complement is **111**. Place a **0** for maxterm $(A'+B'+C')$ in this cell **(1,1,1)** of the K-map as shown above.

Why should $(A'+B'+C')$ cause a **0** to be in cell **111**? When $A'+B'+C'$ is $(1'+1'+1')$, all **1s** in, which is $(0+0+0)$ after taking complements, we have the only condition that will give us a **0**. All the **1s** are complemented to all **0s**, which is **0** when **ORed**.

$$\begin{aligned} \text{Out} &= (A+B+C)(A+B+\overline{C}) \\ \text{Maxterm} &= (A+B+C) & \text{Maxterm} &= (A+B+\overline{C}) \\ \text{Numeric} &= 1 \ 1 \ 1 & \text{Numeric} &= 1 \ 1 \ 0 \\ \text{Complement} &= 0 \ 0 \ 0 & \text{Complement} &= 0 \ 0 \ 1 \end{aligned}$$

	BC	00	01	11	10
A	0	0	0	1	1
	1	1	1	1	1

A Boolean Product-Of-Sums expression or map may have multiple maxterms as shown above. Maxterm $(A+B+C)$ yields numeric **111** which complements to **000**, placing a **0** in cell **(0,0,0)**. Maxterm $(A+B+C')$ yields numeric **110** which complements to **001**, placing a **0** in cell **(0,0,1)**.

Now that we have the k-map setup, what we are really interested in is showing how to write a Product-Of-Sums reduction. Form the **0s** into groups. That would be a group of two below. Write the binary value corresponding to the sum-term which is **(0,0,X)**. Both **A** and **B** are **0** for the group. But, **C** is both **0** and **1** so we write an **X** as a place holder for **C**. Form the complement **(1,1,X)**. Write the Sum-term **(A+B)** discarding the **C** and the **X** which held its' place. In general, expect to have more sum-terms multiplied together in the Product-Of-Sums result. Though, we have a simple example here.

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

	BC			
A	00	01	11	10
0	0	0	1	1
1	1	1	1	1

$A \ B \ C = 0 \ 0 \ X$
 Complement = $1 \ 1 \ X$
 Sum-term = $(A+B)$
 Out = $(A+B)$

Let's summarize the procedure for writing the Product-Of-Sums Boolean reduction for a K-map:

- Form largest groups of 0s possible, covering all maxterms. Groups must be a power of 2.
- Write binary numeric value for group.
- Complement binary numeric value for group.
- Convert complement value to a sum-term.
- Repeat steps for other groups. Each group yields a sum-term within a Product-Of-Sums result.

Example:

Simplify the Product-Of-Sums Boolean expression below, providing a result in POS form.

$$\text{Out} = (A+B+C+\bar{D})(A+B+\bar{C}+D)(A+\bar{B}+C+\bar{D})(A+\bar{B}+\bar{C}+D) \\ (\bar{A}+\bar{B}+\bar{C}+D)(\bar{A}+B+C+\bar{D})(\bar{A}+B+\bar{C}+D)$$

Solution:

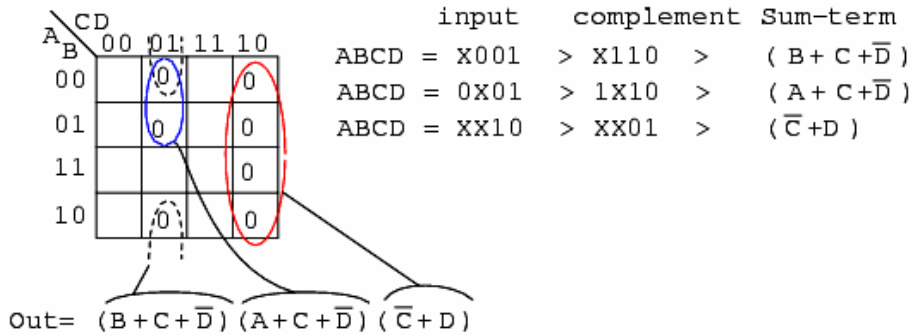
Transfer the seven maxterms to the map below as **0s**. Be sure to complement the input variables in finding the proper cell location.

$$\text{Out} = (A+B+C+\bar{D})(A+B+\bar{C}+D)(A+\bar{B}+C+\bar{D})(A+\bar{B}+\bar{C}+D) \\ (\bar{A}+\bar{B}+\bar{C}+D)(\bar{A}+B+C+\bar{D})(\bar{A}+B+\bar{C}+D)$$

	CD			
A	00	01	11	10
B				
00		0		0
01		0		0
11				0
10		0		0

We map the 0s as they appear left to right top to bottom on the map above. We locate the last three maxterms with leader lines..

Once the cells are in place above, form groups of cells as shown below. Larger groups will give a sum-term with fewer inputs. Fewer groups will yield fewer sum-terms in the result.



We have three groups, so we expect to have three sum-terms in our POS result above. The group of 4-cells yields a 2-variable sum-term. The two groups of 2-cells give us two 3-variable sum-terms. Details are shown for how we arrived at the Sum-terms above. For a group, write the binary group input address, then complement it, converting that to the Boolean sum-term. The final result is product of the three sums.

Example:

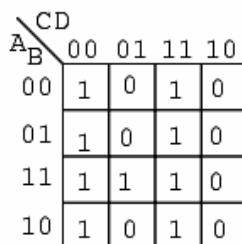
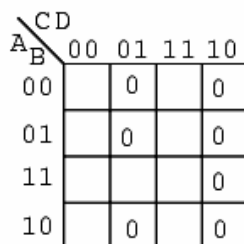
Simplify the Product-Of-Sums Boolean expression below, providing a result in SOP form.

$$\text{Out} = (A+B+C+\bar{D})(A+B+\bar{C}+D)(A+\bar{B}+C+\bar{D})(A+\bar{B}+\bar{C}+D)(\bar{A}+\bar{B}+\bar{C}+D)(\bar{A}+B+C+\bar{D})(\bar{A}+B+\bar{C}+D)$$

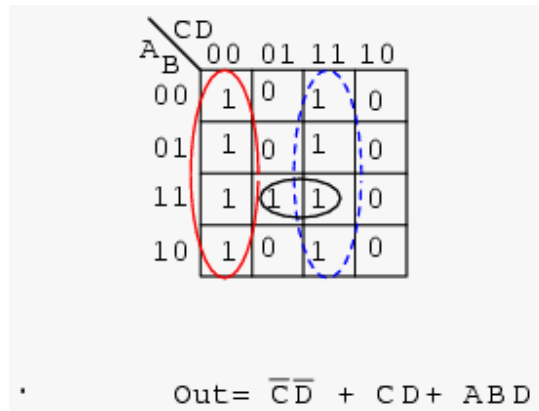
Solution:

This looks like a repeat of the last problem. It is except that we ask for a Sum-Of-Products Solution instead of the Product-Of-Sums which we just finished. Map the maxterm 0s from the Product-Of-Sums given as in the previous problem, below left.

$$\text{Out} = (A+B+C+\bar{D})(A+B+\bar{C}+D)(A+\bar{B}+C+\bar{D})(A+\bar{B}+\bar{C}+D)(\bar{A}+\bar{B}+\bar{C}+D)(\bar{A}+B+C+\bar{D})(\bar{A}+B+\bar{C}+D)$$

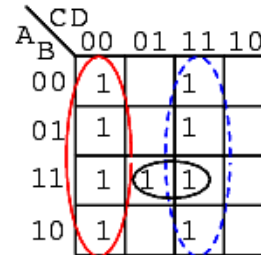
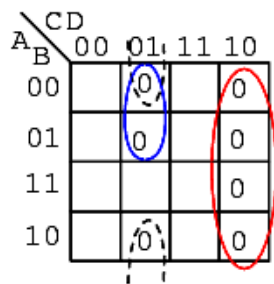


Then fill in the implied 1s in the remaining cells of the map above right.



Form groups of 1s to cover all 1s. Then write the Sum-Of-Products simplified result as in the previous section of this chapter. This is identical to a previous problem.

$$Out = (A+B+C+\overline{D})(A+B+\overline{C}+D)(A+\overline{B}+C+\overline{D})(A+\overline{B}+\overline{C}+D) \\ (\overline{A}+\overline{B}+\overline{C}+D)(\overline{A}+B+C+\overline{D})(\overline{A}+B+\overline{C}+D)$$

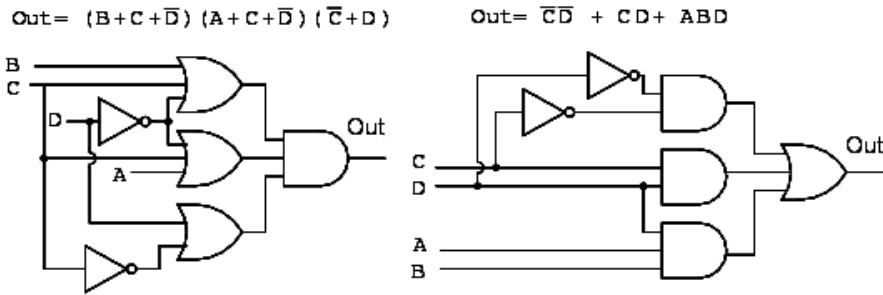


$$Out = \overline{C}\overline{D} + CD + ABD$$

$$Out = (B+C+\overline{D})(A+C+\overline{D})(\overline{C}+D)$$

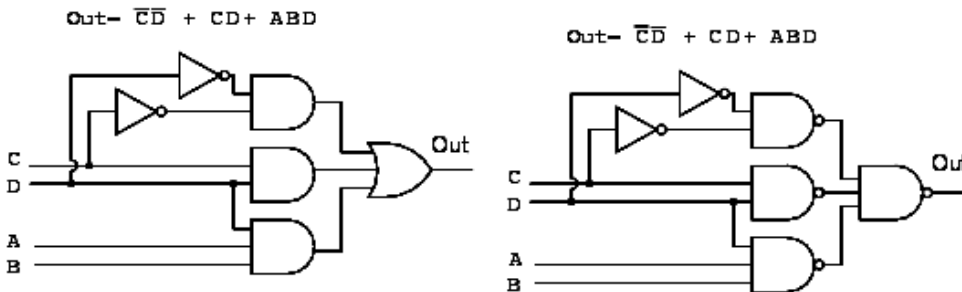
Above we show both the Product-Of-Sums solution, from the previous example, and the Sum-Of-Products solution from the current problem for comparison. Which is the simpler solution? The POS uses 3-OR gates and 1-AND gate, while the SOP uses 3-AND gates and 1-OR gate. Both use four gates each. Taking a closer look, we count the number of gate inputs. The POS uses 8-inputs; the SOP uses 7-inputs. By the definition of minimal cost solution, the SOP solution is simpler. This is an example of a technically correct answer that is of little use in the real world.

The better solution depends on complexity and the logic family being used. The SOP solution is usually better if using the TTL logic family, as NAND gates are the basic building block, which works well with SOP implementations. On the other hand, A POS solution would be acceptable when using the CMOS logic family since all sizes of NOR gates are available.

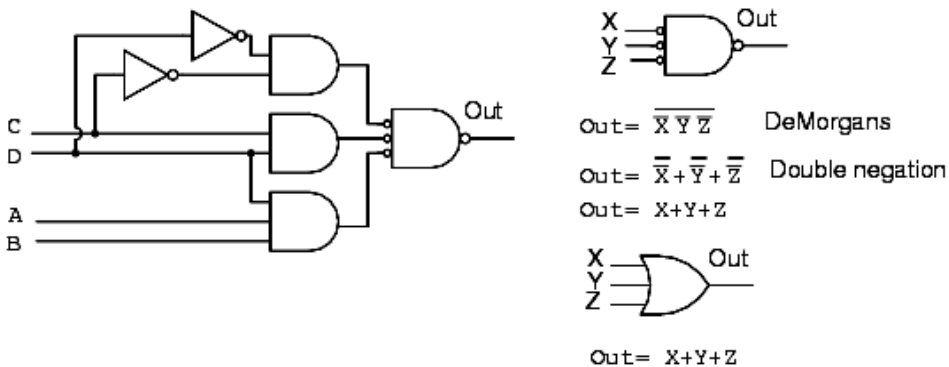


The gate diagrams for both cases are shown above, Product-Of-Sums left, and Sum-Of-Products right.

Below, we take a closer look at the Sum-Of-Products version of our example logic, which is repeated at left.



Above all AND gates at left have been replaced by NAND gates at right.. The OR gate at the output is replaced by a NAND gate. To prove that AND-OR logic is equivalent to NAND-NAND logic, move the inverter invert bubbles at the output of the 3-NAND gates to the input of the final NAND as shown in going from above right to below left.



Above right we see that the output NAND gate with inverted inputs is logically equivalent to an OR gate by DeMorgan's theorem and double negation. This information is useful in building digital logic in a laboratory setting where TTL logic family NAND gates are more readily available in a wide variety of configurations than other types.

The Procedure for constructing NAND-NAND logic, in place of AND-OR logic is as follows:

- Produce a reduced Sum-Of-Products logic design.
- When drawing the wiring diagram of the SOP, replace all gates (both AND and OR) with NAND gates.
- Unused inputs should be tied to logic High.
- In case of troubleshooting, internal nodes at the first level of NAND gate outputs do NOT match AND-OR diagram logic levels, but are inverted. Use the NAND-NAND logic diagram. Inputs and final output are identical, though.
- Label any multiple packages U1, U2,.. etc.
- Use data sheet to assign pin numbers to inputs and outputs of all gates.

Example:

Let us revisit a previous problem involving an SOP minimization. Produce a Product-Of-Sums solution. Compare the POS solution to the previous SOP.

$$\begin{aligned} \text{Out} = & \bar{A}\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}\bar{C}D + \bar{A}\bar{B}CD \\ & + \bar{A}B\bar{C}\bar{D} + \bar{A}B\bar{C}D + \bar{A}BCD \\ & + AB\bar{C}\bar{D} + AB\bar{C}D + ABCD \end{aligned}$$

		CD			
		00	01	11	10
A B	00	1	1	1	
	01	1	1	1	
	11	1	1	1	
	10				

		CD			
		00	01	11	10
A B	00	1	1	1	0
	01	1	1	1	0
	11	1	1	1	0
	10	0	0	0	0

		CD			
		00	01	11	10
A B	00	1	1	1	0
	01	1	1	1	0
	11	1	1	1	0
	10	0	0	0	0

$$\text{Out} = \bar{A}\bar{C} + \bar{A}D + B\bar{C} + BD$$

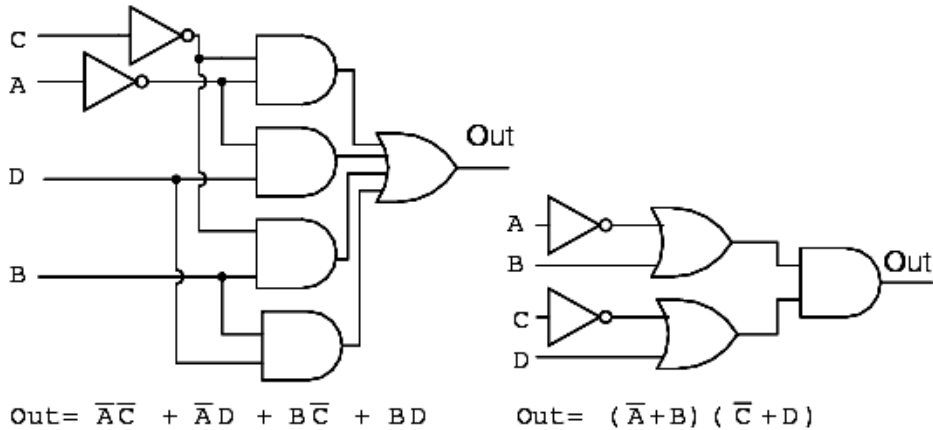
$$\text{Out} = (\bar{A}+B)(\bar{C}+D)$$

Solution:

Above left we have the original problem starting with a 9-minterm Boolean unsimplified expression. Reviewing, we formed four groups of 4-cells to yield a 4-product-term SOP result, lower left.

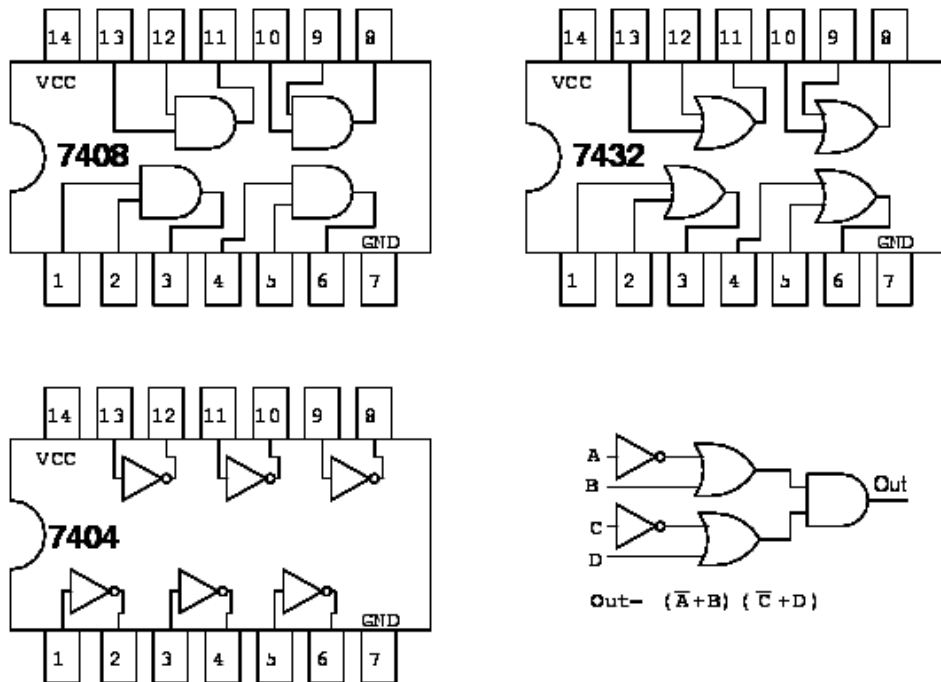
In the middle figure, above, we fill in the empty spaces with the implied 0s. The 0s form two groups of 4-cells. The solid blue group is $(A'+B)$, the dashed red group is $(C'+D)$. This yields two sum-terms in the Product-Of-Sums result, above right **Out** = $(A'+B)(C'+D)$

Comparing the previous SOP simplification, left, to the POS simplification, right, shows that the POS is the least cost solution. The SOP uses 5-gates total, the POS uses only 3-gates. This POS solution even looks attractive when using TTL logic due to simplicity of the result. We can find AND gates and an OR gate with 2-inputs.



The SOP and POS gate diagrams are shown above for our comparison problem.

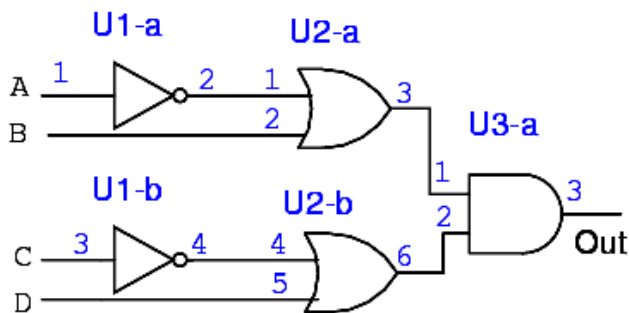
Given the pin-outs for the TTL logic family integrated circuit gates below, label the maxterm diagram above right with Circuit designators (U1-a, U1-b, U2-a, etc), and pin numbers.



Each integrated circuit package that we use will receive a circuit designator: U1, U2, U3. To distinguish between the individual

gates within the package, they are identified as a, b, c, d, etc. The 7404 hex-inverter package is U1. The individual inverters in it are U1-a, U1-b, U1-c, etc. U2 is assigned to the 7432 quad OR gate. U3 is assigned to the 7408 quad AND gate. With reference to the pin numbers on the package diagram above, we assign pin numbers to all gate inputs and outputs on the schematic diagram below.

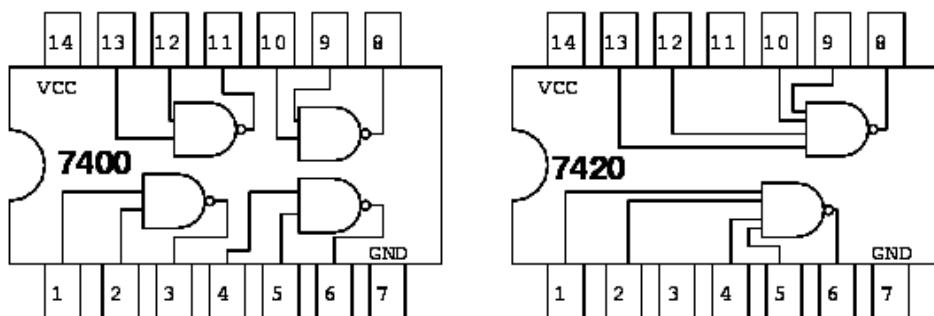
We can now build this circuit in a laboratory setting. Or, we could design a *printed circuit board* for it. A printed circuit board contains copper foil "wiring" backed by a non conductive substrate of phenolic, or epoxy-fiberglass. Printed circuit boards are used to mass produce electronic circuits. Ground the inputs of unused gates.



$$\text{Out} = (\bar{A} + B) (\bar{C} + D)$$

- U1 = 7404
- U2 = 7432
- U3 = 7408

Label the previous POS solution diagram above left (third figure back) with Circuit designators and pin numbers. This will be similar to what we just did.

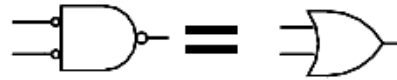


We can find 2-input AND gates, 7408 in the previous example. However, we have trouble finding a 4-input OR gate in our TTL catalog. The only kind of gate with 4-inputs is the 7420 NAND gate shown above right.

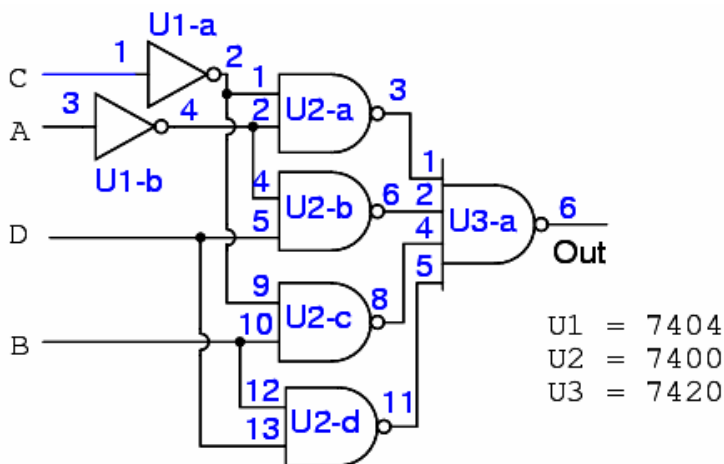
We can make the 4-input NAND gate into a 4-input OR gate by inverting the inputs to the NAND gate as shown below. So we

will use the 7420 4-input NAND gate as an OR gate by inverting the inputs.

$$\begin{aligned} \bar{Y} &= \bar{A} \bar{B} = \overline{A+B} && \text{DeMorgan's} \\ Y &= A+B && \text{Double negation} \end{aligned}$$



We will not use discrete inverters to invert the inputs to the 7420 4-input NAND gate, but will drive it with 2-input NAND gates in place of the AND gates called for in the SOP, minterm, solution. The inversion at the output of the 2-input NAND gates supply the inversion for the 4-input OR gate.



$$\text{Out} = \overline{(\bar{A}\bar{C}) (\bar{A}D) (B\bar{C}) (BD)} \quad \text{Boolean from diagram}$$

$$\text{Out} = \overline{\bar{A}\bar{C} + \bar{A}D + B\bar{C} + BD} \quad \text{DeMorgan's}$$

$$\text{Out} = \bar{A}\bar{C} + \bar{A}D + B\bar{C} + BD \quad \text{Double negation}$$

The result is shown above. It is the only practical way to actually build it with TTL gates by using NAND-NAND logic replacing AND-OR logic.

4.5 QUINE MCCLUSKY METHOD

The Quine-McCluskey method is an exact algorithm which finds a minimum-cost sum-of-products implementation of a Boolean function. This handout introduces the method and applies it to several examples.

There are 4 main steps in the Quine-McCluskey algorithm:

1. Generate Prime Implicants
2. Construct Prime Implicant Table

3. Reduce Prime Implicant Table
 1. Remove Essential Prime Implicants
 2. Row Dominance
 3. Column Dominance
4. Solve Prime Implicant Table

In Step #1, the prime implicants of a function are generated using an iterative procedure. In Step #2, a prime implicant table is constructed. The columns of the table are the prime implicants of the function. The rows are minterms of where the function is 1, called *ON-set minterms*. The goal of the method is to cover all the rows using a minimum-cost *cover* of prime implicants.

The reduction step (Step #3) is used to reduce the size of the table. This step has three sub-steps **which are iterated until no further table reduction is possible!** At this point, the reduced table is either (i) empty or (ii) non-empty. If the reduced table is empty, the removed essential prime implicants form a minimum-cost solution. However, if the reduced table is *not* empty, the table must be "solved" (Step #4). The table can be solved using either "Petrick's method" or the "branching method". This handout focuses on Petrick's method. The branching method is discussed in the books by McCluskey, Roth, etc., but you will not be responsible for the branching method.

The remainder of this handout illustrates the details of the Quine-McCluskey method on 3 examples. Example #1 is fairly straightforward, Examples #2 is more involved, and Example #3 applies the method to a function with "don't-cares". But first, we motivate the need for *column dominance* and *row dominance*.

Example #1:

$$F(A, B, C, D) = \Sigma m(0, 2, 5, 6, 7, 8, 10, 12, 13, 14, 15)$$

The Notation. The above notation is a shorthand to describe the Karnaugh map for F . First, it indicates that F is a Boolean function of 4 variables: A , B , C , and D . Second, each *ON-set minterm* of F is listed above, that is, minterms where the function is 1: $0, 2, 5, \dots$. Each of these numbers corresponds to one entry (or square) in the Karnaugh map. For example, the decimal number 2 corresponds to the minterm $ABCD = 0010$, (0010 is the binary representation of 2). That is, $ABCD = 0010$ is an ON-set minterm of F ; *i.e.*, it is a 1 entry. All remaining minterms, not listed above, are assumed to be 0.

Step 1: Generate Prime Implicants.**List Minterms**

<i>Column I</i>	
0	0000
2	0010
8	1000
5	0101
6	0110
10	1010
12	1100
7	0111
13	1101
14	1110
15	1111

Combine Pairs of Minterms from Column I

A check (✓) is written next to every minterm which can be combined with another minterm.

<i>Column I</i>	<i>Column II</i>
0 0000 ✓	(0,2) 00-0
2 0010 ✓	(0,8) -000
8 1000 ✓	(2,6) 0-10
5 0101 ✓	(2,10) -010
6 0110 ✓	(8,10) 10-0
10 1010 ✓	(8,12) 1-00
12 1100 ✓	(5,7) 01-1
7 0111 ✓	(5,13) -101
13 1101 ✓	(6,7) 011-

14 1110 ✓ (6,14) -110

15 1111 ✓ (10,14) 1-10

(12,13) 110-

(12,14) 11-0

(7,15) -111

(13,15) 11-1

(14,15) 111-

Combine Pairs of Products from Column II

A check (✓) is written next to every product which can be combined with another product.

Column III contains a number of duplicate entries, e.g. (0,2,8,10) and (0,8,2,10). Duplicate entries appear because a product in Column III can be formed in several ways. For example, (0,2,8,10) is formed by combining products (0,2) and (8,10) from Column II, and (0,8,2,10) (the same product) is formed by combining products (0,8) and (2,10).

Duplicate entries should be crossed out. The remaining unchecked products cannot be combined with other products. These are the prime implicants: (0,2,8,10), (2,6,10,14), (5,7,13,15), (6,7,14,15), (8,10,12,14) and (12,13,14,15); or, using the usual product notation: $B'D'$, CD' , BD , BC , AD' and AB .

Column I			Column II			Column III		
0	0000	✓	(0,2)	00-0	✓	(0,2,8,10)	-0-0	
2	0010	✓	(0,8)	-000	✓	(0,8,2,10)	-0-0	
8	1000	✓	(2,6)	0-10	✓	(2,6,10,14)	-10	
5	0101	✓	(2,10)	-010	✓	(2,10,6,14)	-10	
6	0110	✓	(8,10)	10-0	✓	(8,10,12,14)	1-0	
10	1010	✓	(8,12)	1-00	✓	(8,12,10,14)	1-0	

12	1100	✓	(5,7)	01-1	✓	(5,7,13,15)	-1-1
7	0111	✓	(5,13)	-101	✓	(5,13,7,15)	-1-1
13	1101	✓	(6,7)	011-	✓	(6,7,14,15)	-11-
14	1110	✓	(6,14)	-110	✓	(6,14,7,15)	-11-
15	1111	✓	(10,14)	1-10	✓	(12,13,14,15)	11-
			(12,13)	110-	✓	(12,14,13,15)	11-
			(12,14)	11-0	✓		
			(7,15)	-111	✓		
			(13,15)	11-1	✓		
			(14,15)	111-	✓		

Step 2: Construct Prime Implicant Table.

	<i>B'D'</i>	<i>CD'</i>	<i>BD</i>	<i>BC</i>	<i>AD'</i>	<i>AB</i>
	(0,2,8,10)	(2,6,10,14)	(5,7,13,15)	(6,7,14,15)	(8,10,12,14)	(12,13,14,15)
0	X					
2	X	X				
5			X			
6		X		X		
7			X	X		
8	X				X	
10	X	X			X	
12					X	X

13			X			X
14		X		X	X	X
15			X	X		X

Step 3: Reduce Prime Implicant Table.

Iteration #1.

(i) Remove Primary Essential Prime Implicants

	$B'D'(*)$	CD'	$BD(*)$	BC	AD'	AB
	(0,2,8,10)	(2,6,10,14)	(5,7,13,15)	(6,7,14,15)	(8,10,12,14)	(12,13,14,15)
(o) 0	X					
2	X	X				
(o) 5			X			
6		X		X		
7			X	X		
8	X				X	
10	X	X			X	
12					X	X
13			X			X
14		X		X	X	X
15			X	X		X

* indicates an essential prime implicant

o indicates a distinguished row, i.e. a row covered by only 1 prime implicant

In step #1, *primary essential prime implicants* are identified. These are implicants which will appear in *any* solution. A row which is covered by only 1 prime implicant is called a *distinguished row*. The prime implicant which covers it is an *essential prime implicant*. In this step, essential prime implicants are identified and removed. The corresponding column is crossed out. Also, each row where the column contains an **X** is completely crossed out, since these minterms are now covered. These essential implicants will be

added to the final solution. In this example, $B'D'$ and BD are both primary essentials.

(ii) Row Dominance

The table is simplified by removing rows and columns which were crossed out in step (i). (*Note*: you do not need to do this, but it makes the table easier to read. Instead, you can continue to mark up the original table.)

	CD'	BC	AD'	AB
	(2,6,10,14)	(6,7,14,15)	(8,10,12,14)	(12,13,14,15)
6	X	X		
12			X	X
14	X	X	X	X

Row 14 *dominates* both row 6 and row 12. That is, row 14 has an "X" in every column where row 6 has an "X" (and, in fact, row 14 has "X"s in other columns as well). Similarly, row 14 has an "X" in every column where row 12 has an "X". Rows 6 and 12 are said to be *dominated* by row 14.

A *dominating* row can always be eliminated. To see this, note that every product which covers row 6 also covers row 14. That is, if some product covers row 6, row 14 is *guaranteed* to be covered. Similarly, any product which covers row 12 will also cover row 14. Therefore, row 14 can be crossed out.

(iii) Column Dominance

	CD'	BC	AD'	AB
	(2,6,10,14)	(6,7,14,15)	(8,10,12,14)	(12,13,14,15)
6	X	X		
12			X	X

Column CD' *dominates* column BC . That is, column CD' has an "X" in every row where column BC has an "X". In fact, in this example, column BC also dominates column CD' , so each is *dominated* by the other. (Such columns are said to *co-dominate* each other.) Similarly, columns AD' and AB dominate each other, and each is dominated by the other.

A *dominated* column can always be eliminated. To see this, note that every row covered by the dominated column is also

covered by the dominating column. For example, $C'D$ covers every row which BC covers. Therefore, the dominating column can always replace the dominated column, so the dominated column is crossed out. In this example, CD' and BC dominate each other, so either column can be crossed out (but not both). Similarly, AD' and AB dominate each other, so either column can be crossed out.

Iteration #2.

(i) Remove Secondary Essential Prime Implicants

	CD' (**)	AD' (**)
	(2,6,10,14)	(8,10,12,14)
(\circ)6	X	
(\circ)12		X

** indicates a secondary essential prime implicant

\circ indicates a distinguished row

In iteration #2 and beyond, *secondary essential prime implicants* are identified. These are implicants which will appear in *any* solution, *given* the choice of column-dominance used in the previous steps (if 2 columns co-dominated each other in a previous step, the choice of which was deleted can affect what is an "essential" at this step). As before, a row which is covered by only 1 prime implicant is called a *distinguished row*. The prime implicant which covers it is a (*secondary*) *essential prime implicant*.

Secondary essential prime implicants are identified and removed. The corresponding columns are crossed out. Also, each row where the column contains an X is completely crossed out, since these minterms are now covered. These essential implicants will be added to the final solution. In this example, both CD' and AD' are secondary essentials.

Step 4: Solve Prime Implicant Table.

No other rows remain to be covered, so no further steps are required. Therefore, the minimum-cost solution consists of the primary and secondary essential prime implicants $B'D'$, BD , CD' and AD' :

$$F = B'D' + BD + CD' + AD'$$

Example #2:

$$F(A, B, C, D) = \Sigma m(0, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13)$$

Step 1: Generate Prime Implicants.

Use the method described in Example #1.

Step 2: Construct Prime Implicant Table.

	$A'D'$	$B'D'$	$C'D'$	$A'C$	$B'C$	$A'B$	BC'	AB'	AC'
0	X	X	X						
2	X	X		X	X				
3				X	X				
4	X		X			X	X		
5						X	X		
6	X			X		X			
7				X		X			
8		X	X					X	X
9								X	X
10		X			X			X	
11					X			X	
12			X				X		X
13							X		X

Step 3: Reduce Prime Implicant Table.

Iteration #1.

(i) Remove Primary Essential Prime Implicants

There are no primary essential prime implicants: each row is covered by at least two products.

(ii) Row Dominance

	$A'D'$	$B'D'$	$C'D'$	$A'C$	$B'C$	$A'B$	BC'	AB'	AC'
0	X	X	X						
2	X	X		X	X				
3				X	X				
4	X		X			X	X		
5						X	X		

6	X			X		X			
7				X		X			
8		X	X					X	X
9								X	X
10		X			X			X	
11					X			X	
12			X				X		X
13							X		X

There are many instances of row dominance. Row 2 dominates 3, 4 dominates 5, 6 dominates 7, 8 dominates 9, 10 dominates 11, 12 dominates 13. Dominating rows are removed.

(iii) Column Dominance

	$A'D'$	$B'D'$	$C'D'$	$A'C$	$B'C$	$A'B$	BC'	AB'	AC'
0	X	X	X						
3				X	X				
5						X	X		
7				X		X			
9								X	X
11					X			X	
13							X		X

Columns $A'D'$, $B'D'$ and $C'D'$ each dominate one another. We can remove any two of them.

Iteration #2.

(i) Remove Secondary Essential Prime Implicants

	$A'D'(**)$	$A'C$	$B'C$	$A'B$	BC'	AB'	AC'
(o)0	X						
3		X	X				
5				X	X		
7		X		X			
9						X	X

11			X			X	
13					X		X

** indicates a secondary essential prime implicant

○ indicates a distinguished row

Product $A'D'$ is a secondary essential prime implicant; it is removed from the table.

(ii) Row Dominance

No further row dominance is possible.

(iii) Row Dominance

No further column dominance is possible.

	$A'C$	$B'C$	$A'B$	BC'	AB'	AC'
3	X	X				
5			X	X		
7	X		X			
9					X	X
11		X			X	
13				X		X

There are no additional secondary essential prime implicants, and no further row- or column-dominance is possible.

There are two solutions. Both solutions have a minimal number of prime implicants, so either can be used. With either choice, we must include the secondary essential prime implicant, $A'D'$, identified earlier. Therefore, the two minimum-cost solutions are:

$$F = A'D' + A'C + BC' + AB'$$

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

Example #3: Don't-Cares

$$F(A, B, C, D) = \Sigma m(2, 3, 7, 9, 11, 13) + \Sigma d(1, 10, 15)$$

Step 1: Generate Prime Implicants.

The don't-cares are *included* when generating prime implicants.

Note: As indicated earlier, you should learn this basic method for generating prime implicants (Step #1).

List Minterms

<i>Column I</i>		
1	0001	
2	0010	
3	0011	
9	1001	
10	1010	
7	0111	
11	1011	
13	1101	
15	1111	

Combine Pairs of Minterms from Column I

A check (✓) is written next to every minterm which can be combined with another minterm.

<i>Column I</i>	<i>Column II</i>
1 0001 ✓	(1,3) 00-1
2 0010 ✓	(1,9) -001
3 0011 ✓	(2,3) 001-
9 1001 ✓	(2,10) -010
10 1010 ✓	(3,7) 0-11
7 0111 ✓	(3,11) -011
11 1011 ✓	(9,11) 10-1
13 1101 ✓	(9,13) 1-01

15 1111 ✓ (10,11) 101-
 (7,15) -111
 (11,15) 1-11
 (13,15) 11-1

Combine Pairs of Products from Column II

A check (✓) is written next to every product which can be combined with another product.

<i>Column I</i>			<i>Column II</i>			<i>Column III</i>		
1	0001	✓	(1,3)	00-1	✓	(1,3,9,11)	-0-1	
2	0010	✓	(1,9)	-001	✓	(2,3,10,11)	-01-	
3	0011	✓	(2,3)	001-	✓	(3,7,11,15)	-11	
9	1001	✓	(2,10)	-010	✓	(9,11,13,15)	1-1	
10	1010	✓	(3,7)	0-11	✓			
7	0111	✓	(3,11)	-011	✓			
11	1011	✓	(9,11)	10-1	✓			
13	1101	✓	(9,13)	1-01	✓			
15	1111	✓	(10,11)	101-	✓			
			(7,15)	-111	✓			
			(11,15)	1-11	✓			
			(13,15)	11-1	✓			

The unchecked products cannot be combined with other products. These are the prime implicants: (1,3,9,11), (2,3,10,11), (3,7,11,15) and (9,11,13,15); or, using the usual product notation: $B'D$, $B'C$, CD and AD .

Step 2: Construct Prime Implicant Table.

The don't-cares are *omitted* when constructing the prime implicant table, since they do not need to be covered.

	$B'D$	$B'C$	CD	AD
	(1,3,9,11)	(2,3,10,11)	(3,7,11,15)	(9,11,13,15)
2		X		
3	X	X	X	
7			X	
9	X			X
11	X	X	X	X
13				X

Step 3: Reduce Prime Implicant Table.**(i) Remove Essential Prime Implicants**

	$B'D$	$B'C(*)$	$CD(*)$	$AD(*)$
	(1,3,9,11)	(2,3,10,11)	(3,7,11,15)	(9,11,13,15)
2		X		
3	X	X	X	
(◦)7			X	
9	X			X
11	X	X	X	X
(◦)13				X

* indicates an essential prime implicant

◦ indicates a distinguished row

Step 4: Solve Prime Implicant Table.

The essential prime implicants cover all the rows, so no further steps are required. Therefore, the minimum-cost solution consists of the essential prime implicants $B'C$, CD and AD :

$$F = B'C + CD + AD$$

4.6 QUESTIONS :

1. Minimise the following expression using Quine Mc Cluskey method:
 - a. $f(A, B, C, D, E) = \sum m(8,9,10,11,13,15,16,18,21,24,25,26,27,30,31)$
 - b. $f(A, B, C, D, E) = \sum m(1,3,4,5,6,8,12,14,15)$
2. Simplify the following using K-map and realize it using 2-input gates:
 - a. $f(A, B, C, D) = \sum m(1,2,9,10,11,14,15)$
 - b. $f(A, B, C, D) = \sum m(0,1,5,9,13,14,15) + d(3,4,7,10,11)$
 - c. $F(A, B, C, D) = \Pi M(4,6,8,9,10,12,13,14) + d(0,2,5)$

4.7 FURTHER READING:

- ❖ **Digital Electronics** - An Introduction to Theory and Practice by W H Gothmann
- ❖ **Computer Architecture and Parallel Processing** by Kai Hwang, Faye A Briggs , McGraw Hill
- ❖ **Computer Architecture and Organization** by William Stallings
- ❖ **Fundamentals of Computer organization and Design** by Sivarama P. Dandamudi
- ❖ http://en.wikipedia.org/wiki/Canonical_form
- ❖ <http://en.wikipedia.org/wiki/Minterm>
- ❖ http://en.wikipedia.org/wiki/Quine-McCluskey_algorithm
- ❖ http://en.wikipedia.org/wiki/Karnaugh_map



COMBINATIONAL LOGIC DESIGN

Unit Structure

5.0 Objectives

5.1 Combinational Circuits

5.2 Adders and Subtractors

5.2.1 Half Adder

5.2.2 Full Adder

5.2.3: Half Subtractor

5.2.4 Full Subtractor

5.3 Code Converters

5.3.1 Bcd to Excess – 3 ($X_8 - 3$) Code Conversion

5.3.2 Binary to Gray Code Conversion

5.3.3 Gray to Binary Code Conversion

5.4 Questions

5.6 Further Reading

5.0 OBJECTIVES

After completing this chapter, you will be able to:

- ❖ Understand the basics of Combinational Circuits.
- ❖ Understand the structure and working of Adders & Subtractor with the help of suitable diagrams.
- ❖ Use the Adders & Subtractor in real life applications.
- ❖ Understand the functioning and working of different types of Code Converters with the help of appropriate diagram.
- ❖ Using the code converters in applications.

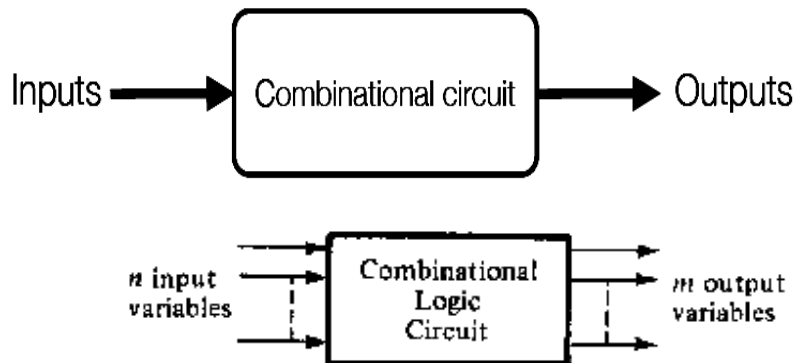
5.1 COMBINATIONAL CIRCUITS

Logic circuits for digital systems may be combinational or sequential. A combinational circuit consists of logic gates whose outputs at any time are determined by combining the values of the applied inputs using logic operations. A combinational circuit performs an operation that can be specified logically by a set of Boolean expression. In addition to using logic gates, sequential circuits employ elements that store bit values. Sequential circuit outputs are a function of inputs and the bit value in storage

elements. These values, in turn, are a function of previously applied inputs and stored values. As a consequence, the outputs of a sequential circuit depend not only on the presently applied values of the inputs, but also on past inputs, and the behavior of the circuit must be specified by a sequence in time of inputs and internal stored bit values.

A combinational circuit consists of input variables, output variables, logic gates and interconnections. The interconnected logic gates accept signals from the inputs and generate signals at the output. The n input variables come from the environment of the circuit, and the m output variables are available for use by the environment. Each input and output variable exists physically as a binary signal that represents logic 1 or logic 0.

For n input variables, there are 2^n possible binary input combinations. For each binary combination of the input variables, there is one possible binary value on each output. Thus, a combinational circuit can be specified by a truth table that lists the output values for each combination of the input variables. A combinational circuit can also be described by m Boolean functions, one for each output variable. Each such function is expressed as a function of the n input variables.



Combinational Circuit Design

The design of combinational circuit starts from a specification of the problem and culminates in a logic diagram or set of Boolean equations from which the logic diagram can be obtained. The procedure involves the following steps:

1. From the specifications of the circuit, determine the required number of inputs and outputs, and assign a letter symbol to each.
2. Derive the truth table that defines the required relationship between inputs and outputs.

3. Obtain the simplified Boolean functions of each outputs as function of the input variables.
4. Draw the logic diagram.
5. Verify the correctness of the design.

5.2 ADDERS AND SUBTRACTORS

Digital computers perform a variety of information-processing tasks. Among the basic functions encountered are the various arithmetic operations. The most basic arithmetic operation, no doubt, is the addition of two binary digits. This simple addition consists of four possible elementary operations, namely, $0 + 0 = 0$, $0 + 1 = 1$, $1 + 0 = 1$ and $1 + 1 = 10$. The first three operations produce a sum whose length is one digit, but when both bits are equal to 1, the binary sum consists of two digits. The higher significant bit of this result is called a **carry**. When the bits contain more significant digits, the carry obtained from the addition of two bits is added to the next higher-order pair of significant bits. A combinational circuit that performs the addition of two bits is called a **half-adder**. One that performs the addition of three bits (two significant bits and a previous carry) is a **full-adder**. The name of the former stems from the fact that two half-adders can be employed to implement a full-adder.

5.2.1 Half Adder

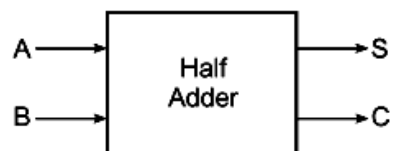
A *half-adder* is an arithmetic circuit block that can be used to add two bits. Such a circuit thus has two inputs that represent the two bits to be added and two outputs, with one producing the SUM output and the other producing the CARRY. The figure shows the truth table of a half-adder, showing all possible input combinations and the corresponding outputs.

The Boolean expressions for the SUM and CARRY outputs are given by the equations

$$\text{SUM} = A\bar{B} + \bar{A}B$$

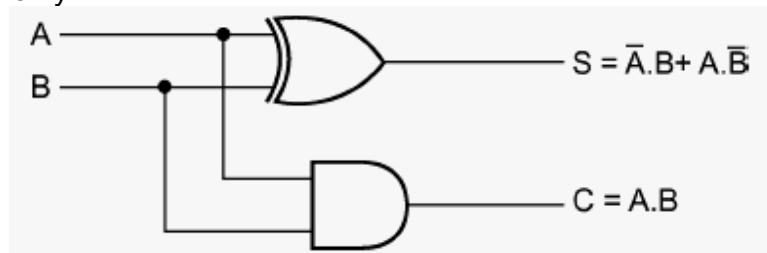
$$\text{CARRY} = \bar{A}B$$

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



An examination of the two expressions tells that there is no scope for further simplification. While the first one representing the SUM output is that of an EX-OR gate, the second one representing

the CARRY output is that of an AND gate. However, these two expressions can certainly be represented in different forms using various laws and theorems of Boolean algebra to illustrate the flexibility that the designer has in hardware-implementing as simple a combinational function as that of a half-adder. The simplest way to hardware-implement a half-adder would be to use a two-input EX-OR gate for the SUM output and a two-input AND gate for the CARRY output, as shown in figure, it could also be implemented by using an appropriate arrangement of either NAND or NOR gates. The figure shows the implementation of a half-adder with NAND gates only.



5.2.2 Full Adder

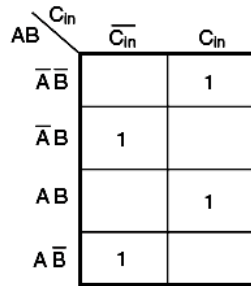
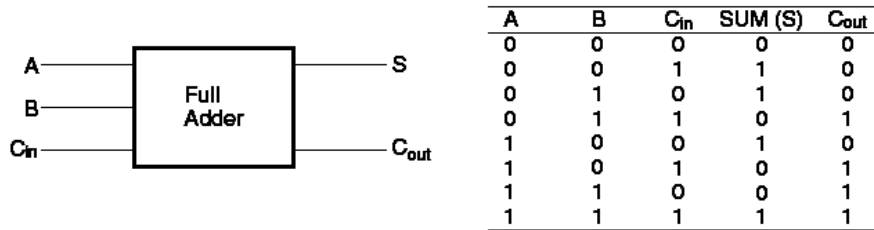
A *full adder* circuit is an arithmetic circuit block that can be used to add three bits to produce a SUM and a CARRY output. Such a building block becomes a necessity when it comes to adding binary numbers with a large number of bits. The full adder circuit overcomes the limitation of the half-adder, which can be used to add two bits only. Let us recall the procedure for adding larger binary numbers. We begin with the addition of LSBs of the two numbers. We record the sum under the LSB column and take the carry, if any, forward to the next higher column bits. As a result, when we add the next adjacent higher column bits, we would be required to add three bits if there were a carry from the previous addition. We have a similar situation for the other higher column bits also until we reach the MSB. A full adder is therefore essential for the hardware implementation of an adder circuit capable of adding larger binary numbers. A half-adder can be used for addition of LSBs only.

Figure shows the truth table of a full adder circuit showing all possible input combinations and corresponding outputs. In order to arrive at the logic circuit for hardware implementation of a full adder, we will firstly write the Boolean expressions for the two output variables, that is, the SUM and CARRY outputs, in terms of input variables. These expressions are then simplified by using any of the simplification techniques described in the previous chapter. The Boolean expressions for the two output variables are given in Equations below for (S) and CARRY output(Cout)

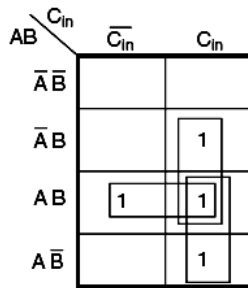
$$S = \bar{A}\bar{B}C_{in} + \bar{A}B\bar{C}_{in} + A\bar{B}\bar{C}_{in} + ABC_{in}$$

$$Cout = \bar{A}BC_{in} + A\bar{B}C_{in} + AB\bar{C}_{in} + ABC_{in}$$

The next step is to simplify the two expressions. We will do so with the help of the Karnaugh mapping technique. Karnaugh maps for the two expressions are given in Figure.



(a)

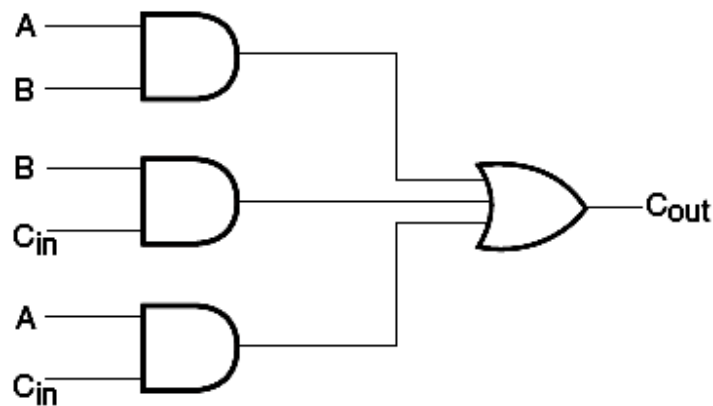
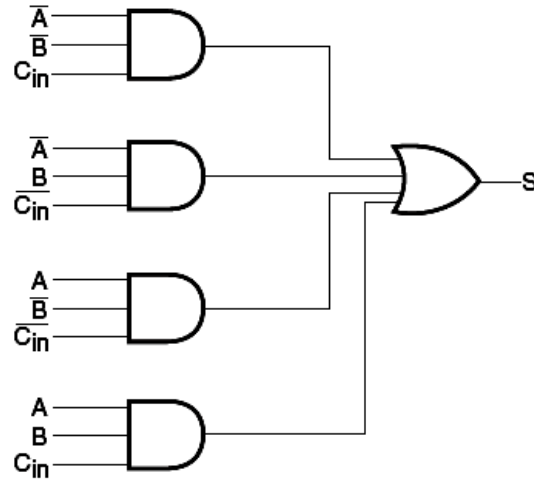
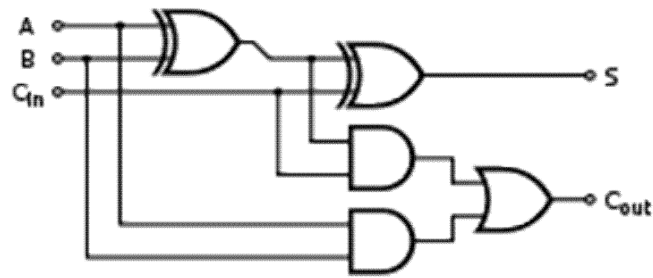


(b)

As is clear from the two maps, the expression for the SUM (S_{output}) cannot be simplified any further, whereas the simplified Boolean expression for Cout is given by the equation

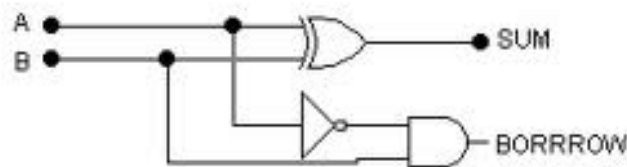
$$C_{out} = B.C_{in} + A.B + A.C_{in}$$

$$C_{out} = A.B + C_{in}.(\overline{A}.B + A.\overline{B})$$



5.2.3 Half Subtractor

The half-subtractor is a combinational circuit which is used to perform subtraction of two bits. It has two inputs, X (minuend) and Y (subtrahend) and two outputs D (difference) and B (borrow).



**DIFFERENC
E**



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

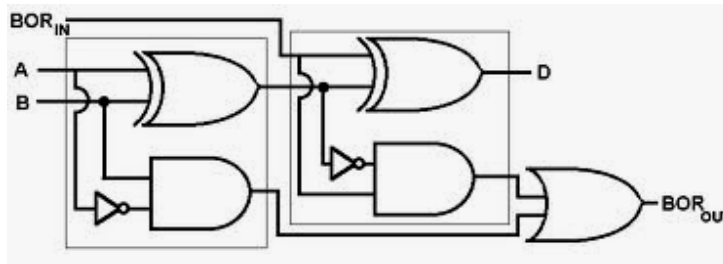
5.2.4 Full Subtractor

The full-subtractor is a combinational circuit which is used to perform subtraction of three bits. It has three inputs, X (minuend) and Y (subtrahend) and Z (subtrahend) and two outputs D (difference) and B (borrow).

Easy way to write truth table

D=A-B-BOR_{IN} (don't bother about sign)

BOR_{OUT} = 1 If A<(B+ BOR_{IN})



A	B	BOR _{IN}	D	BOR _{OUT}
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

So, Logic equations are:

$$D = X \oplus Y \oplus Z$$

$$B = Z \cdot (X \oplus Y) + \bar{X} \cdot Y$$

Where X= A, Y = B, Z = BOR_{IN} B=BOR_{OUT}

5.3 CODE CONVERTERS

The availability of a large Variety of codes for the same discrete elements of information results in the use of different codes

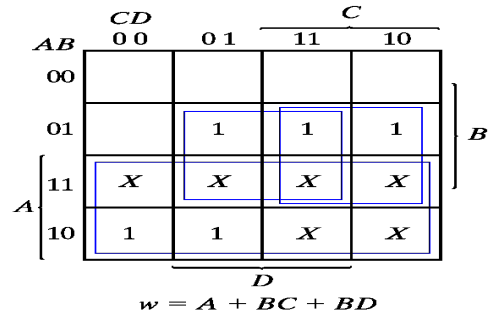
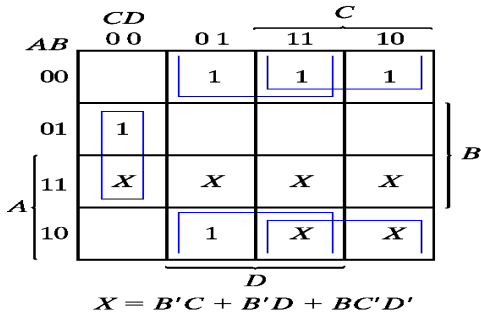
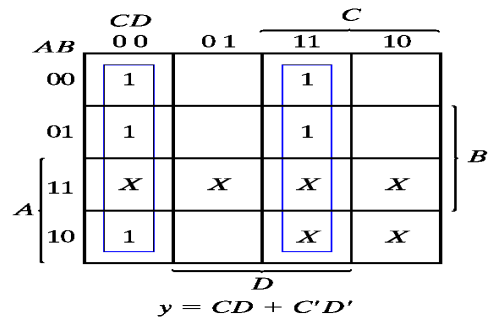
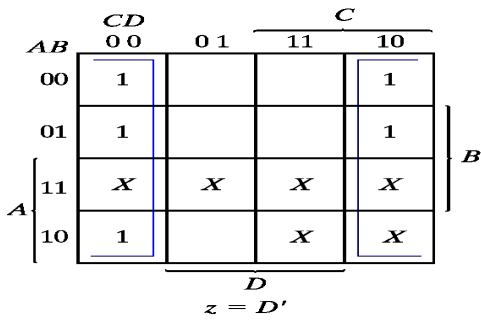
by different digital systems. It is sometimes necessary to use the output of one system as the input to another. A conversion circuit must be inserted between the two systems if each uses different codes for the same information. Thus, a code converter is a circuit that makes the two systems compatible even though each uses a different binary code. To convert from binary code A to binary code B, the input lines must supply the bit combination of elements as specified by code A and the output lines must generate the corresponding bit combination of code B. A combinational circuit performs this transformation by means of logic gates.

5.3.1 BCD to EXCESS – 3 (XS – 3) Code Conversion

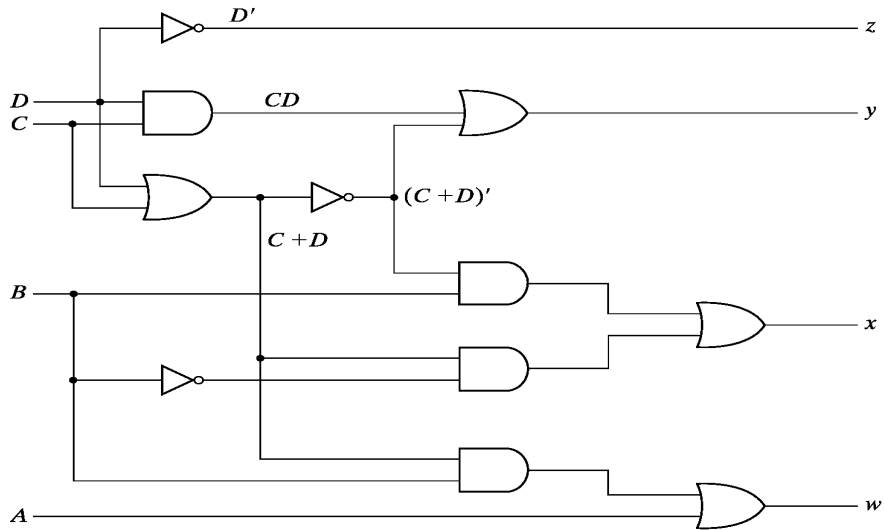
Input (Std BCD code)

Output (XS3 Code)

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
1	0	1	0		X	X	X	X
1	0	1	1		X	X	X	X
1	1	0	1		X	X	X	X
1	1	1	0		X	X	X	X
1	1	1	1		X	X	X	X



$z = D'$
 $y = CD + C'D' = CD + (C + D)'$
 $x = B'C + B'D + BC'D' = B'(C + D) + BC'D'$
 $w = A + BC + BD = A + B(C + D)$



5.3.2 Binary to Gray code Conversion

Binary				Gray			
A	B	C	D	G3	G2	G1	G0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	0
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	0

	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	0	0	0	0
$\bar{A}B$	0	0	0	0
AB	1	1	1	1
$A\bar{B}$	1	1	1	1

$G3=A$

	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	0	0	0	0
$\bar{A}B$	1	1	1	1
AB	0	0	0	0
$A\bar{B}$	1	1	1	1

$G2= A\oplus B$

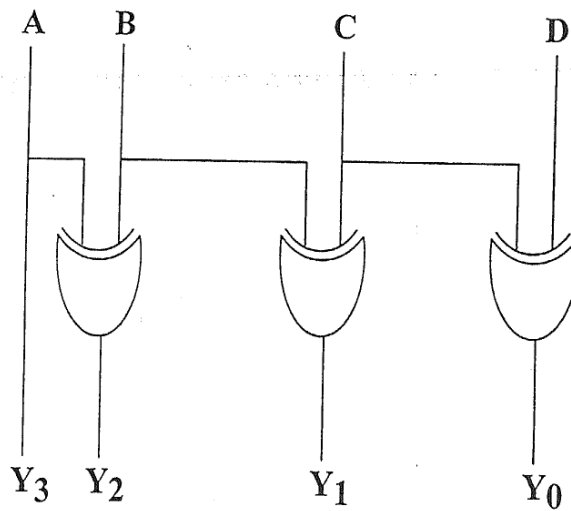
	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	0	0	(1 1)	
$\bar{A}B$	(1 1)		0	0
AB	(1 1)		0	0
$A\bar{B}$	0	0	(1 1)	

$G_0 = C \oplus D$

	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	0	(1)	0	(1)
$\bar{A}B$	0	(1)	0	(1)
AB	0	(1)	0	(1)
$A\bar{B}$	0	(1)	0	(1)

$G_1 = B \oplus C$

Binary Input



Gray Code Output

5.3.3 Gray to binary code Conversion

A given Gray code number can be converted into its binary equivalent by going through the following steps:

1. Begin with the most significant bit (MSB). The MSB of the binary number is the same as the MSB of the Gray code number.
2. The bit next to the MSB (the second MSB) in the binary number is obtained by adding the MSB in the binary number to the second MSB in the Gray code number and disregarding the carry, if any.
3. The third MSB in the binary number is obtained by adding the second MSB in the binary number to the third MSB in the Gray code number. Again, carry, if any, is to be ignored.
4. The process continues until we obtain the LSB of the binary number.

The conversion process is further illustrated with the help of an example showing step-by-step conversion of the Gray code number 1110 into its binary equivalent:

Gray code 1110

Binary 1 - - -

Gray code 1110

Binary 10 - -

Gray code 1110

Binary 101

Gray code 1110

Binary 1011

Gray				Binary			
A	B	C	D	G3	G2	G1	G0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	1
1	0	0	0	1	1	1	1
1	0	0	1	1	1	1	0
1	0	1	0	1	1	0	0
1	0	1	1	1	1	0	1
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	1
1	1	1	0	1	0	1	1
1	1	1	1	1	0	1	0

(Getting the expressions using K-maps and drawing the Logic diagram using gates is left as an exercise for students)

The Digital Comparator

Another common and very useful combinational logic circuit is that of the **Digital Comparator** circuit. Digital or Binary Comparators are made up from standard AND, NOR and NOT gates that compare the digital signals present at their input terminals and produce an output depending upon the condition of those inputs. For example, along with being able to add and subtract binary numbers we need to be able to compare them and determine whether the value of input A is greater than, smaller than or equal to the value at input B etc. The digital comparator accomplishes this using several logic gates that operate on the principles of Boolean algebra. There are two main types of digital comparator available and these are.

- Identity Comparator - is a digital comparator that has only one output terminal for when $A = B$ either "HIGH" $A = B = 1$ or "LOW" $A = B = 0$
- Magnitude Comparator - is a type of digital comparator that has three output terminals, one each for equality, $A = B$ greater than, $A > B$ and less than $A < B$

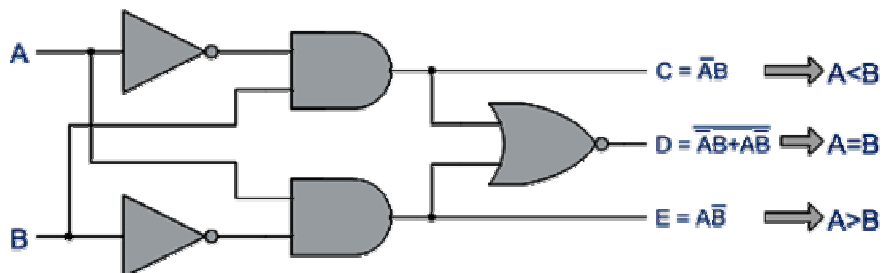
The purpose of a **Digital Comparator** is to compare a set of variables or unknown numbers, for example A ($A_1, A_2, A_3, \dots, A_n$, etc) against that of a constant or unknown value such as B ($B_1, B_2, B_3, \dots, B_n$, etc) and produce an output condition or flag depending upon the result of the comparison. For example, a magnitude comparator of two 1-bits, (A and B) inputs would produce the following three output conditions when compared to each other.

$$A > B, \quad A = B, \quad A < B$$

Which means: A is greater than B, A is equal to B, and A is less than B

This is useful if we want to compare two variables and want to produce an output when any of the above three conditions are achieved. For example, produce an output from a counter when a certain count number is reached. Consider the simple 1-bit comparator below.

1-bit Comparator



Then the operation of a 1-bit digital comparator is given in the following Truth Table.

Truth Table

Inputs		Outputs		
B	A	A > B	A = B	A < B
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0

You may notice two distinct features about the comparator from the above truth table. Firstly, the circuit does not distinguish between either two "0" or two "1"s as an output $A = B$ is produced when they are both equal, either $A = B = "0"$ or $A = B = "1"$. Secondly, the output condition for $A = B$ resembles that of a commonly available logic gate, the Exclusive-NOR or Ex-NOR function (equivalence) on each of the n-bits giving: $Q = A \oplus B$

Digital comparators actually use Exclusive-NOR gates within their design for comparing their respective pairs of bits. When we are comparing two binary or BCD values or variables against each other, we are comparing the "magnitude" of these values, a logic "0" against a logic "1" which is where the term **Magnitude Comparator** comes from.

Example:

In a car, we have the following components:

A Day-night sensor: Day-1, Night-0

B Lamps on: On-1, Off-0

C Ignition on: On-1, Off-0

D Warning light for lamps-on

In this case, the truth table for the logic D would be

A	B	C	D
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Therefore,

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

which can be written as in the sum of product form.

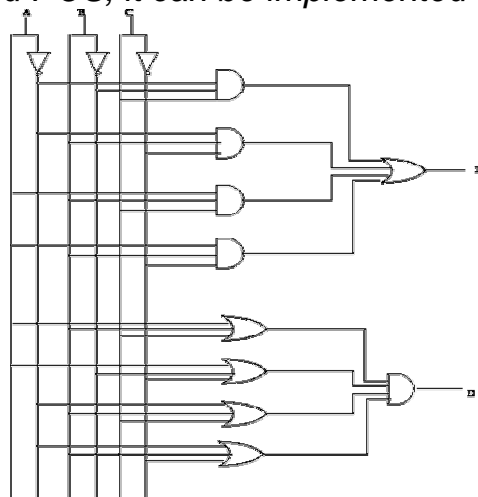
$$\sum 1,2,6,7$$

We arrive at this by looking at the combinations when the output is one.

We can alternatively, express this in the product of sums form by looking at the combinations when the output is low as

$$D = (A + B + C) \cdot (A + \overline{B} + \overline{C}) \cdot (\overline{A} + B + C) \cdot (\overline{A} + B + \overline{C}) = \prod 0,4,5,6$$

Using SOP and POS, it can be implemented as follows:



Next, we will try to reduce the number of gates by combining terms suitably.

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

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

We can get the above by clubbing the 1's in the k-map shown.

Now, if we club the zeroes together in the k-map,

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

Check that we get the same expression by simplifying the product of sums expression (by using $(X+Y)(X+Z)=X+YZ$)

5.4 QUESTIONS:

1. Design a 4-bit gray to 4-bit binary code converter.
2. Design a 4-bit binary to 4-bit gray code converter.
3. A step in space vehicle checkout depends on FOUR sensors S_1 , S_2 , S_3 and S_4 . Every circuit is working properly if sensor S_1 and at least two of the other three sensors are at logic 1. Assuming that the output is 1 when the circuit is working properly,

implement the system using NAND gates only after finding minimal SOP expression for the output.

4. Design a two-bit by two-bit multiplier circuit. Implement using minimum hardware.
5. Design a two bit comparator circuit.

5.5 FURTHER READING:

- ❖ **Digital Electronics** - An Introduction to Theory and Practice by W H Gothmann
- ❖ **Computer Architecture and Parallel Processing** by Kai Hwang, Faye A Briggs , McGraw Hill
- ❖ **Computer Architecture and Organization** by William Stallings
- ❖ **Fundamentals of Computer organization and Design** by Sivarama P. Dandamudi
- ❖ http://en.wikipedia.org/wiki/Combinational_logic
- ❖ [http://en.wikipedia.org/wiki/Adder_\(electronics\)](http://en.wikipedia.org/wiki/Adder_(electronics))
- ❖ <http://en.wikipedia.org/wiki/Subtractor>



MULTIPLEXERS AND DEMULTIPLEXERS

Unit Structure

6.0 Objectives

6.1 Multiplexers

6.1.1 4-to-1 Channel Multiplexer

6.1.2 4 Channel Multiplexer using Logic Gates

6.1.3 4-to-2 Channel Multiplexer

6.1.4 The Demultiplexer

6.1.5 1-to-4 Channel De-multiplexer

6.1.6 4 Channel Demultiplexer using Logic Gates

6.1.7 The Digital Encoder

6.1.8 4-to-2 Bit Binary Encoder

6.1.9 Priority Encoder

6.1.10 8-to-3 Bit Priority Encoder

6.1.11 Encoder Applications

6.1.12 A 4-to-16 Binary Decoder Configuration

6.2 Questions

6.3 Further Reading

6.0 OBJECTIVES:

After completing this chapter, you will be able to:

- ❖ Learning the basics about Multiplexer Electronic Circuit.
- ❖ Understand the Structure & working of different types of Multiplexers with help of suitable diagrams.
- ❖ Learning the basics about Demultiplexer Electronic Circuit.
- ❖ Understand the basics of Digital Encoders.
- ❖ Aware about applications of Encoders.

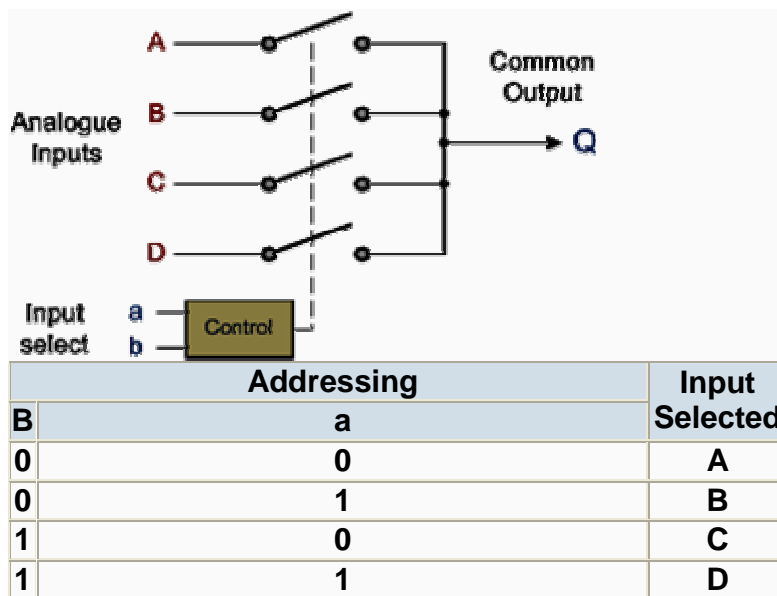
6.1 MULTIPLEXERS

A data selector, more commonly called a **Multiplexer**, shortened to "Mux" or "MPX", are combinational logic switching devices that operate like a very fast acting multiple position rotary switch. They connect or control, multiple input lines called

"channels" consisting of either 2, 4, 8 or 16 individual inputs, one at a time to an output. Then the job of a multiplexer is to allow multiple signals to *share* a single common output. For example, a single 8-channel multiplexer would connect one of its eight inputs to the single data output. Multiplexers are used as one method of reducing the number of logic gates required in a circuit or when a single data line is required to carry two or more different digital signals.

Digital **Multiplexers** are constructed from individual **analogue switches** encased in a single IC package as opposed to the "mechanical" type selectors such as normal conventional switches and relays. Generally, multiplexers have an even number of data inputs, usually an even power of two, n^2 , a number of "control" inputs that correspond with the number of data inputs and according to the binary condition of these control inputs, the appropriate data input is connected directly to the output. An example of a **Multiplexer** configuration is shown below.

6.1.1:4-to-1 Channel Multiplexer



The Boolean expression for this 4-to-1 **Multiplexer** above with inputs A to D and data select lines a, b is given as:

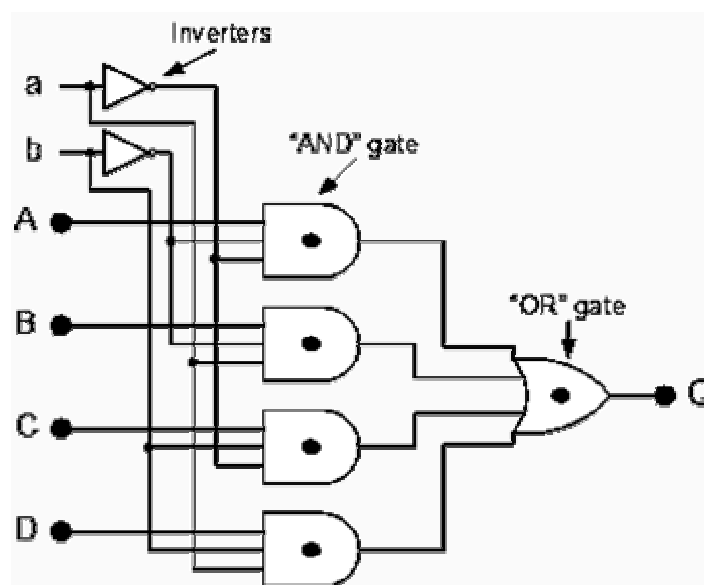
$$Q = abA + abB + abC + abD$$

In this example at any one instant in time only ONE of the four analogue switches is closed, connecting only one of the input lines A to D to the single output at Q. As to which switch is closed depends upon the addressing input code on lines "a" and "b", so for this example to select input B to the output at Q, the binary input

address would need to be "a" = logic "1" and "b" = logic "0". Adding more control address lines will allow the multiplexer to control more inputs but each control line configuration will connect only ONE input to the output.

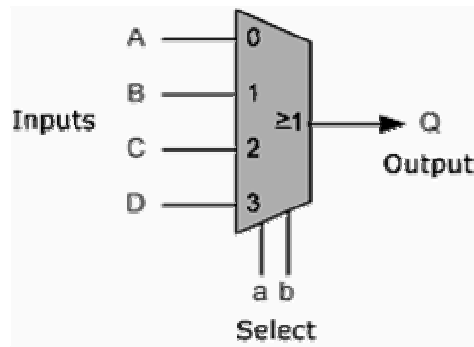
Then the implementation of this Boolean expression above using individual logic gates would require the use of seven individual gates consisting of AND, OR and NOT gates as shown.

6.1.2: 4 Channel Multiplexer using Logic Gates



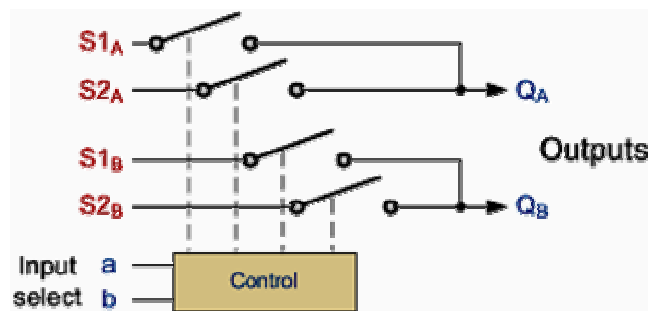
The symbol used in logic diagrams to identify a multiplexer is as follows.

Multiplexer Symbol



Multiplexers are not limited to just switching a number of different input lines or channels to one common single output. There are also types that can switch their inputs to multiple outputs and have arrangements of 4 to 2, 8 to 3 or even 16 to 4 etc configurations and an example of a simple Dual channel 4 input multiplexer (4 to 2) is given below:

6.1.3 :4-to-2 Channel Multiplexer

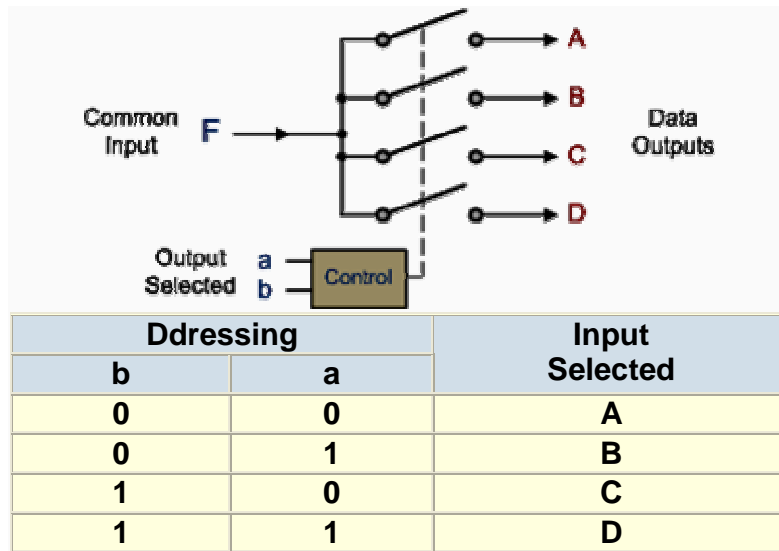


Here in this example the 4 input channels are switched to 2 individual output lines but larger arrangements are also possible. This simple 4 to 2 configuration could be used for example, to switch audio signals for stereo pre-amplifiers or mixers.

6.1.4 :The Demultiplexer

The data distributor, known more commonly as a **Demultiplexer** or "Demux", is the exact opposite of the **Multiplexer** we saw in the previous tutorial. The demultiplexer takes one single input data line and then switches it to any one of a number of individual output lines one at a time. The **demultiplexer** converts a serial data signal at the input to a parallel data at its output lines as shown below.

6.1.5:1-to-4 Channel De-multiplexer



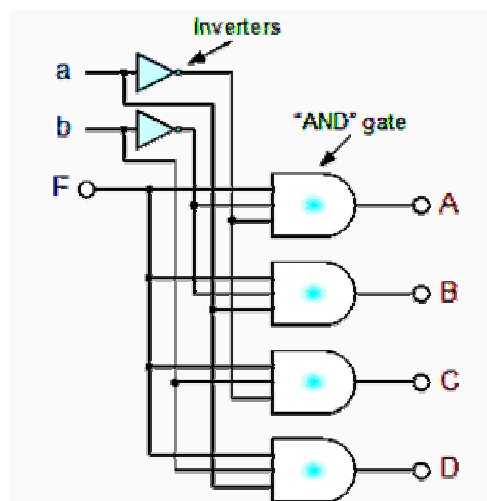
The Boolean expression for this 1-to-4 **Demultiplexer** above with outputs A to D and data select lines a, b is given as:

$$F = ab A + abB + abC + abD$$

The function of the **Demultiplexer** is to switch one common data input line to any one of the 4 output data lines A to D in our example above. As with the multiplexer the individual solid state switches are selected by the binary input address code on the output select pins "a" and "b" and by adding more address line inputs it is possible to switch more outputs giving a 1-to- 2^n data line outputs. Some standard demultiplexer IC's also have an "enable output" input pin which disables or prevents the input from being passed to the selected output. Also some have latches built into their outputs to maintain the output logic level after the address inputs have been changed. However, in standard decoder type circuits the address input will determine which single data output will have the same value as the data input with all other data outputs having the value of logic "0".

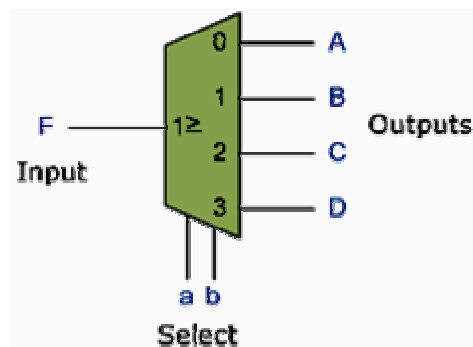
The implementation of the Boolean expression above using individual logic gates would require the use of six individual gates consisting of AND and NOT gates as shown.

6.1.6:4 Channel Demultiplexer using Logic Gates



The symbol used in logic diagrams to identify a demultiplexer is as follows.

Demultiplexer Symbol



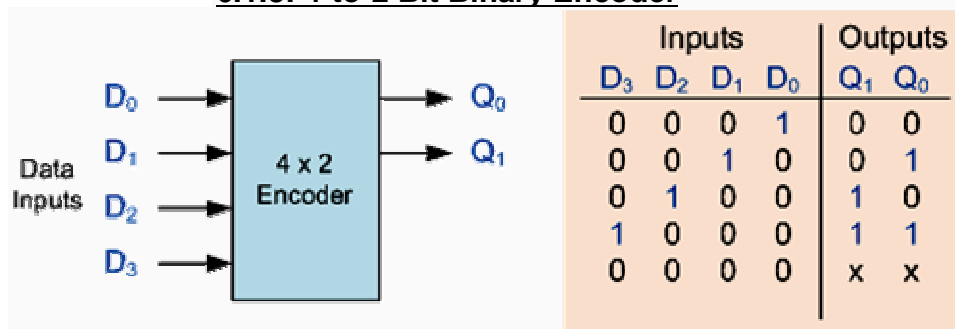
Standard **Demultiplexer** IC packages available are the TTL 74LS138 1 to 8-output demultiplexer, the TTL 74LS139 Dual 1-to-4 output demultiplexer or the CMOS CD4514 1-to-16 output demultiplexer. Another type of demultiplexer is the 24-pin, 74LS154 which is a 4-bit to 16-line demultiplexer/decoder. Here the individual output positions are selected using a 4-bit binary coded input. Like multiplexers, demultiplexers can also be cascaded together to form higher order demultiplexers.

Unlike multiplexers which convert data from a single data line to multiple lines and demultiplexers which convert multiple lines to a single data line, there are devices available which convert data to and from multiple lines and in the next tutorial about combinational logic devices, we will look at [Encoders](#) which convert multiple input lines into multiple output lines, converting the data from one form to another.

6.1.7 :The Digital Encoder

Unlike a multiplexer that selects one individual data input line and then sends that data to a single output line or switch, a **Digital Encoder** more commonly called a **Binary Encoder** takes *ALL* its data inputs one at a time and then converts them into a single encoded output. So we can say that a binary encoder, is a multi-input combinational logic circuit that converts the logic level "1" data at its inputs into an equivalent binary code at its output. Generally, digital encoders produce outputs of 2-bit, 3-bit or 4-bit codes depending upon the number of data input lines. An "n-bit" binary encoder has 2^n input lines and n-bit output lines with common types that include 4-to-2, 8-to-3 and 16-to-4 line configurations. The output lines of a digital encoder generate the binary equivalent of the input line whose value is equal to "1" and are available to encode either a decimal or hexadecimal input pattern to typically a binary or B.C.D. output code.

6.1.8: 4-to-2 Bit Binary Encoder



One of the main disadvantages of standard digital encoders is that they can generate the wrong output code when there is more than one input present at logic level "1". For example, if we make inputs D₁ and D₂ HIGH at logic "1" at the same time, the resulting output is neither at "01" or at "10" but will be at "11" which is an output binary number that is different to the actual input present. Also, an output code of all logic "0"s can be generated when all of its inputs are at "0" OR when input D₀ is equal to one.

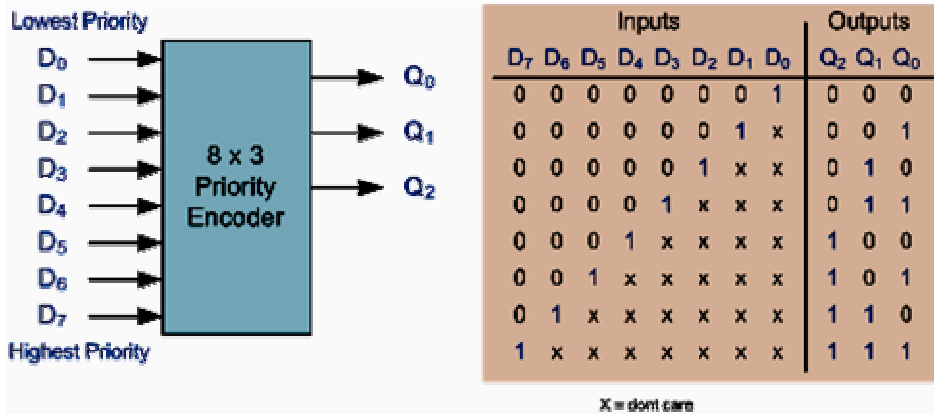
One simple way to overcome this problem is to "Prioritise" the level of each input pin and if there was more than one input at logic level "1" the actual output code would only correspond to the input with the highest designated priority. Then this type of digital encoder is known commonly as a **Priority Encoder** or **P-encoder** for short.

6.1.9: Priority Encoder

The **Priority Encoder** solves the problems mentioned above by allocating a priority level to each input. The *priority*

encoders output corresponds to the currently active input which has the highest priority. So when an input with a higher priority is present, all other inputs with a lower priority will be ignored. The priority encoder comes in many different forms with an example of an 8-input priority encoder along with its truth table shown below.

6.1.10 :8-to-3 Bit Priority Encoder



Priority encoders are available in standard IC form and the TTL 74LS148 is an 8-to-3 bit priority encoder which has eight active LOW (logic "0") inputs and provides a 3-bit code of the highest ranked input at its output. Priority encoders output the highest order input first for example, if input lines "D2", "D3" and "D5" are applied simultaneously the output code would be for input "D5" ("101") as this has the highest order out of the 3 inputs. Once input "D5" had been removed the next highest output code would be for input "D3" ("011"), and so on.

The truth table for a 8-to-3 bit priority encoder is given as:

Digital Inputs								Binary Output		
D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀	Q ₂	Q ₁	Q ₀
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	X	0	0	1
0	0	0	0	0	1	X	X	0	1	0
0	0	0	0	1	X	X	X	0	1	1
0	0	0	1	X	X	X	X	1	0	0
0	0	1	X	X	X	X	X	1	0	1
0	1	X	X	X	X	X	X	1	1	0
1	X	X	X	X	X	X	X	1	1	1

From this truth table, the Boolean expression for the encoder above with inputs D₀ to D₇ and outputs Q₀, Q₁, Q₂ is given as:

Output Q₀

$$Q_0 = \Sigma(1, 3, 5, 7)$$

$$Q_0 = \Sigma(\bar{D}_7\bar{D}_6\bar{D}_5\bar{D}_4\bar{D}_3\bar{D}_2D_1 + \bar{D}_7\bar{D}_6\bar{D}_5\bar{D}_4D_3 + \bar{D}_7\bar{D}_6D_5 + D_7)$$

$$Q_0 = \Sigma(\bar{D}_6\bar{D}_4\bar{D}_2D_1 + \bar{D}_6\bar{D}_4D_3 + \bar{D}_6D_5 + D_7)$$

$$Q_0 = \Sigma(\bar{D}_6(\bar{D}_4\bar{D}_2D_1 + \bar{D}_4D_3 + D_5) + D_7)$$

Output Q_1

$$Q_1 = \Sigma(2, 3, 6, 7)$$

$$Q_1 = \Sigma(\bar{D}_7\bar{D}_6\bar{D}_5\bar{D}_4\bar{D}_3D_2 + \bar{D}_7\bar{D}_6\bar{D}_5\bar{D}_4D_3 + \bar{D}_7D_6 + D_7)$$

$$Q_1 = \Sigma(\bar{D}_5\bar{D}_4D_2 + \bar{D}_5\bar{D}_4D_3 + D_6 + D_7)$$

$$Q_1 = \Sigma(\bar{D}_5\bar{D}_4(D_2 + D_3) + D_6 + D_7)$$

Output Q_2

$$Q_2 = \Sigma(4, 5, 6, 7)$$

$$Q_2 = \Sigma(\bar{D}_7\bar{D}_6\bar{D}_5D_4 + \bar{D}_7\bar{D}_6D_5 + \bar{D}_7D_6 + D_7)$$

$$Q_2 = \Sigma(D_4 + D_5 + D_6 + D_7)$$

Then the final Boolean expression for the priority encoder including the zero inputs is defined as:

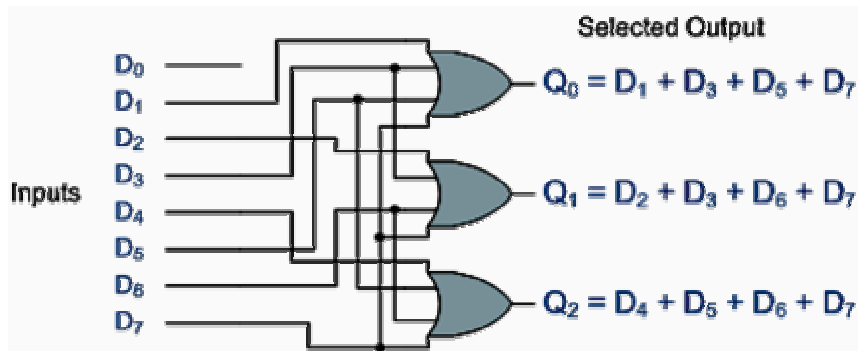
$$Q_0 = \Sigma(\bar{D}_6(\bar{D}_4\bar{D}_2D_1 + \bar{D}_4D_3 + D_5) + D_7)$$

$$Q_1 = \Sigma(\bar{D}_5\bar{D}_4(D_2 + D_3) + D_6 + D_7)$$

$$Q_2 = \Sigma(D_4 + D_5 + D_6 + D_7)$$

In practice these zero inputs would be ignored allowing the implementation of the final Boolean expression for the outputs of the 8-to-3 **priority encoder** above to be constructed using individual ORgates as follows.

Digital Encoder using Logic Gates



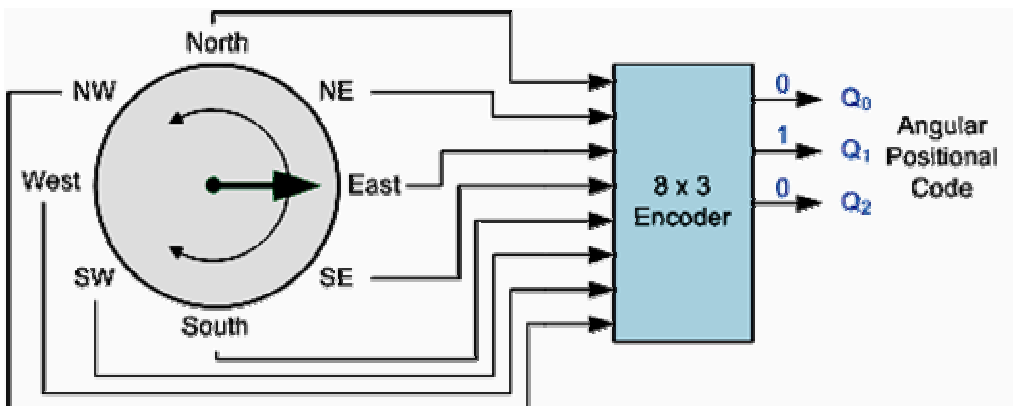
6.1.11 :Encoder Applications

Keyboard Encoder

Priority encoders can be used to reduce the number of wires needed in a particular circuits or application that have multiple inputs. For example, assume that a microcomputer needs to read the 104 keys of a standard QWERTY keyboard where only one key would be pressed either "HIGH" or "LOW" at any one time. One way would be to connect all 104 wires from the keys directly to the computer but this would be impractical for a small home PC, but another better way would be to use a priority encoder. The 104 individual buttons or keys could be encoded into a standard ASCII code of only 7-bits (0 to 127 decimal) to represent each key or character of the keyboard and then inputted as a much smaller 7-bit B.C.D code directly to the computer. Keypad encoders such as the 74C923 20-key encoder are available to do just that.

Positional Encoders

Another more common application is in magnetic positional control as used on ships or robots etc. Here the angular or rotary position of a compass is converted into a digital code by an encoder and inputted to the systems computer to provide navigational data and an example of a simple 8 position to 3-bit output compass encoder is shown below. Magnets and reed switches could be used to indicate the compasses angular position.



Compass Direction	Binary Output		
	Q ₀	Q ₁	Q ₂
North	0	0	0
North-East	0	0	1
East	0	1	0
South-East	0	1	1
South	1	0	0
South-West	1	0	1
West	1	1	0
North-West	1	1	1

Interrupt Requests

Other applications especially for **Priority Encoders** may include detecting interrupts in microprocessor applications. Here the microprocessor uses interrupts to allow peripheral devices such as the disk drive, scanner, mouse, or printer etc, to communicate with it, but the microprocessor can only "talk" to one peripheral device at a time. The processor uses "Interrupt Requests" or "IRQ" signals to assign priority to the devices to ensure that the most important peripheral device is serviced first. The order of importance of the devices will depend upon their connection to the priority encoder.

IRQ Number	Typical Use	Description
IRQ 0	System timer	Internal System Timer.
IRQ 1	Keyboard	Keyboard Controller.
IRQ 3	COM2 & COM4	Second and Fourth Serial Port.
IRQ 4	COM1 & COM3	First and Third Serial Port.
IRQ 5	Sound	Sound Card.
IRQ 6	Floppy disk	Floppy Disk Controller.
IRQ 7	Parallel port	Parallel Printer.

IRQ 12	Mouse	PS/2 Mouse.
IRQ 14	Primary IDE	Primary Hard Disk Controller.
IRQ 15	Secondary IDE	Secondary Hard Disk Controller.

Because implementing such a system using priority encoders such as the standard 74LS148 priority encoder IC involves additional logic circuits, purpose built integrated circuits such as the 8259 Programmable Priority Interrupt Controller is available.

Digital Encoder Summary

Then to summarise, the **Digital Encoder** is a combinational circuit that generates a specific code at its outputs such as binary or BCD in response to one or more active inputs. There are two main types of digital encoder. The **Binary Encoder** and the **Priority Encoder**.

The **Binary Encoder** converts one of 2^n inputs into an n-bit output. Then a binary encoder has fewer output bits than the input code. Binary encoders are useful for compressing data and can be constructed from simple AND or OR gates. One of the main disadvantages of a standard binary encoder is that it would produce an error at its outputs if more than one input were active at the same time. To overcome this problem priority encoders were developed.

The **Priority Encoder** is another type of combinational circuit similar to a binary encoder, except that it generates an output code based on the highest prioritised input. Priority encoders are used extensively in digital and computer systems as microprocessor interrupt controllers where they detect the highest priority input.

In the next tutorial about combinational logic devices, we will look at complementary function of the encoder called a **Decoder** which convert an n-bit input code to one of its 2^n output lines.

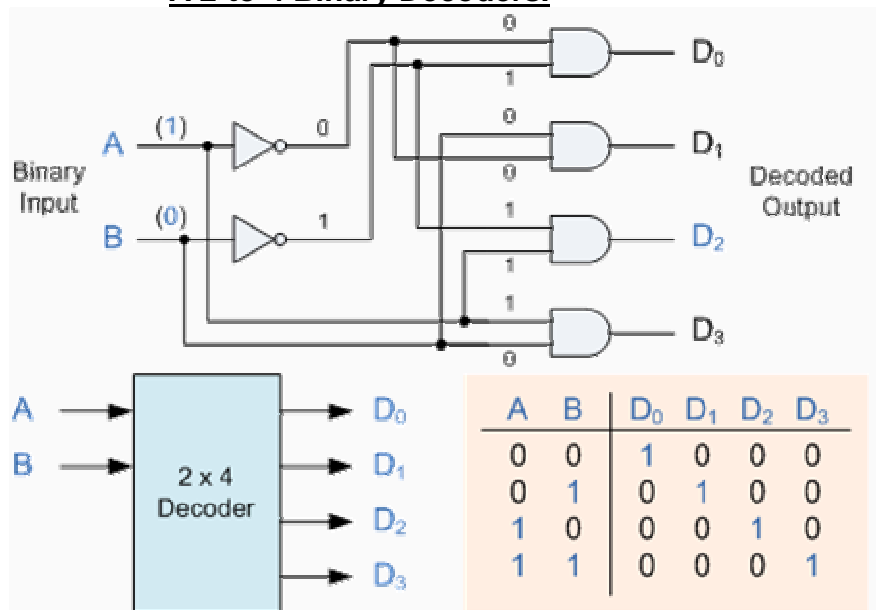
Binary Decoder

A **Decoder** is the exact opposite to that of an "Encoder" we looked at in the last tutorial. It is basically, a combinational type logic circuit that converts the binary code data at its input into one of a number of different output lines, one at a time producing an equivalent decimal code at its output. **Binary Decoders** have inputs

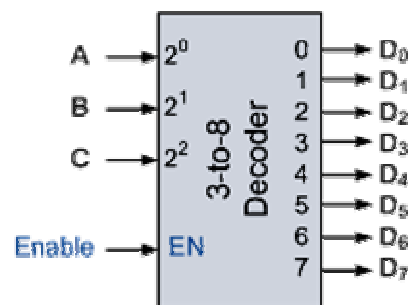
of 2-bit, 3-bit or 4-bit codes depending upon the number of data input lines, and a n-bit decoder has 2^n output lines. Therefore, if it receives n inputs (usually grouped as a binary or Boolean number) it activates one and only one of its 2^n outputs based on that input with all other outputs deactivated. A decoder's output code normally has more bits than its input code and practical binary decoder circuits include, 2-to-4, 3-to-8 and 4-to-16 line configurations.

A binary decoder converts coded inputs into coded outputs, where the input and output codes are different and decoders are available to "decode" either a Binary or BCD (8421 code) input pattern to typically a Decimal output code. Commonly available BCD-to-Decimal decoders include the TTL 7442 or the CMOS 4028. An example of a 2-to-4 line decoder along with its truth table is given below. It consists of an array of four NAND gates, one of which is selected for each combination of the input signals A and B.

A 2-to-4 Binary Decoders.



In this simple example of a 2-to-4 line binary decoder, the binary inputs A and B determine which output line from D₀ to D₃ is "HIGH" at logic level "1" while the remaining outputs are held "LOW" at logic "0" so only one output can be active (HIGH) at any one time. Therefore, whichever output line is "HIGH" identifies the binary code present at the input, in other words it "de-codes" the binary input and these types of binary decoders are commonly used as **Address Decoders** in microprocessor memory applications.

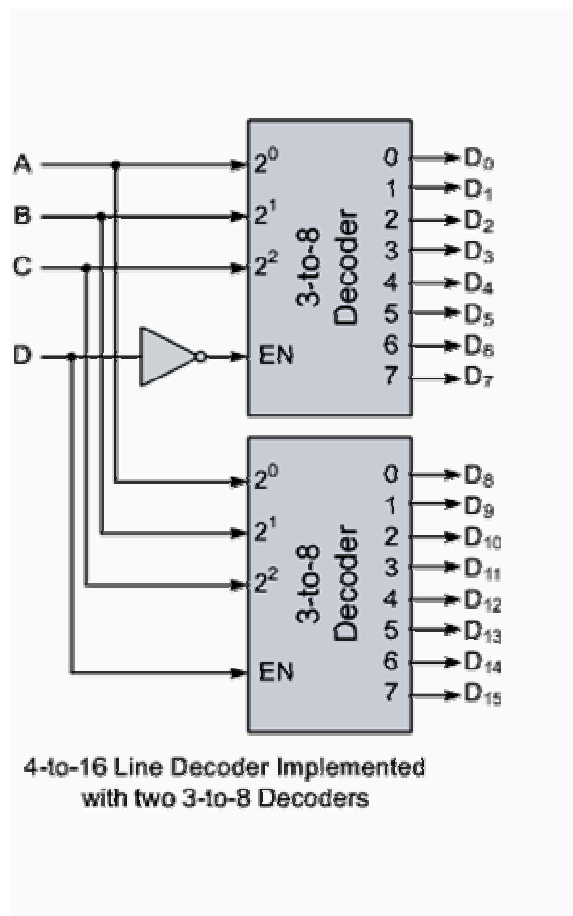


74LS138 Binary Decoder

Some binary decoders have an additional input labelled "Enable" that controls the outputs from the device. This allows the decoders outputs to be turned "ON" or "OFF" and we can see that the logic diagram of the basic decoder is identical to that of the basic demultiplexer. Therefore, we say that a demultiplexer is a decoder with an additional data line that is used to enable the decoder. An alternative way of looking at the decoder circuit is to regard inputs A, B and C as address signals. Each combination of A, B or C defines a unique address which can access a location having that address.

Sometimes it is required to have a **Binary Decoder** with a number of outputs greater than is available, or if we only have small devices available, we can combine multiple decoders together to form larger decoder networks as shown. Here a much larger 4-to-16 line binary decoder has been implemented using two smaller 3-to-8 decoders.

6.1.12 :A 4-to-16 Binary Decoder Configuration.



Inputs A, B, C are used to select which output on either decoder will be at logic "1" (HIGH) and input D is used with the enable input to select which encoder either the first or second will output the "1".

Example 1: 2x1 Mux

A 2x1 Mux has 2 input lines (D_0 & D_1), one select input (S), and one output line (Y). (see Figure 2)

IF $S=0$, then $Y= D_0$
 Else ($S=1$) $Y= D_1$

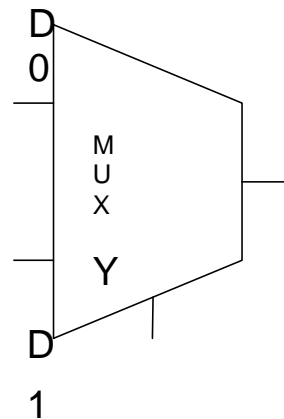


Figure 2: A 2 X 1 Multiplexer

Thus, the output signal Y can be expressed as:

$$Y = S_1 D_0 + \bar{S}_1 D_1$$

Example 2: 4x1 Mux

A 4x1 Mux has 4 input lines (D_0, D_1, D_2, D_3), two select inputs (S_0 & S_1), and one output line Y. (see Figure 3)

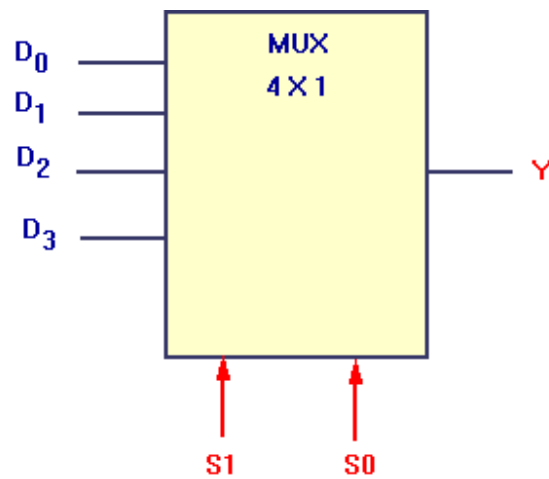
IF $S_1 S_0 = 00$,	
then	$Y = D_0$
IF $S_1 S_0 = 01$,	
then	$Y = D_1$
IF $S_1 S_0 = 10$,	
then	$Y = D_2$
IF $S_1 S_0 = 11$,	
then	$Y = D_3$

Thus, the output signal Y can be expressed as:

$$Y = \underbrace{\bar{S}_1 \bar{S}_0}_{m_0} D_0 + \underbrace{\bar{S}_1 S_0}_{m_1} D_1 + \underbrace{S_1 \bar{S}_0}_{m_2} D_2 + \underbrace{S_1 S_0}_{m_3} D_3$$

minterm minterm minterm minterm
 m_0 m_1 m_2 m_3

Obviously, the input selected to be passed to the output depends on the minterm expressions of the select inputs.



**Figure 3: A 4
X1
Multiplexer**

In General,

For MUXes with n select inputs, the output Y is given by

$$Y = m_0 D_0 + m_1 D_1 + m_2 D_2 + \dots + m_{2^n - 1} D_{2^n - 1}$$

Where $m_i = i^{\text{th}}$ minterm of the Select Inputs

Thus

$$Y = \sum_{i=0}^{2^n - 1} m_i D_i$$

Example 3: Quad 2X1 Mux

Given two 4-bit numbers A and B , design a multiplexer that selects one of these 2 numbers based on some select signal S . Obviously, the output (Y) is a 4-bit number.

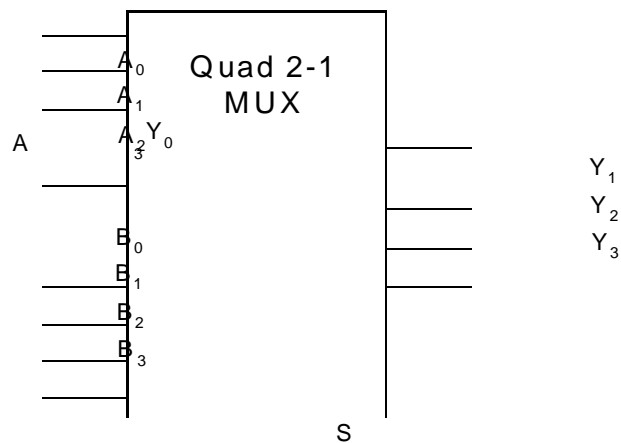


Figure 4: Quad 2 X 1 Multiplexer

The 4-bit output number Y is defined as follows:

$$Y = A \text{ IF } S=0, \text{ otherwise } Y = B$$

The circuit is implemented using four 2x1 Muxes, where the output of each of the Muxes gives one of the outputs (Y_i).

Combinational Circuit Implementation using Muxes

Problem Statement:

Given a function of n -variables, show how to use a MUX to implement this function.

This can be accomplished in one of 2 ways:

Using a Mux with n -select inputs

Using a Mux with $n-1$ select inputs

Method 1: Using a Mux with n -select inputs

n variables need to be connected to n select inputs. For a MUX with n select inputs, the output Y is given by:

$$Y = m_0 D_0 + m_1 D_1 + m_2 D_2 + \dots + m_{2^n - 1} D^{n-1}$$

Alternatively,

$$Y = \sum_{i=0}^{2^n - 1} m_i D_i$$

Where $m_i = i^{th}$ minterm of the Select Inputs

The MUX output expression is a SUM of minterms expression for all minterms (m_i) which have their corresponding inputs (D_i) equal to 1.

Thus, it is possible to implement any function of n -variables using a MUX with n -select inputs by proper assignment of the input values ($D_i \in \{0, 1\}$).

$$Y(S_{n-1} \dots S_1 S_0) = \sum(\text{minterms})$$

Example 4: Implement the function $F(A, B, C) = \sum(1, 3, 5, 6)$ (see Figure 5)

Since number of variables $n = 3$, this requires a Mux with 3 select inputs, i.e. an 8x1 Mux

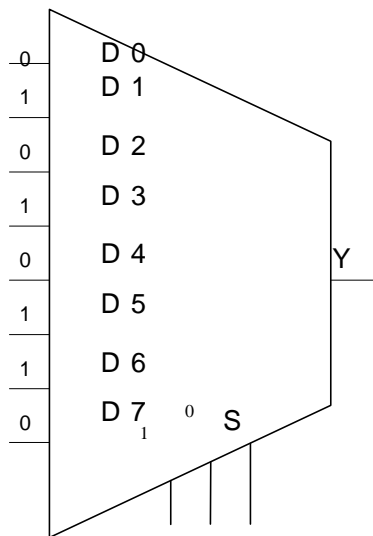
The most significant variable A is connected to the most significant select input S_2 while the least significant variable C is connected to the least significant select input S_0 , thus:

$$S_2 = A, S_1 = B, \text{ and } S_0 = C$$

For the MUX output expression (sum of minterms) to include minterm 1 we assign $D_1 = 1$

Likewise, to include minterms 3, 5, and 6 in the sum of minterms expression while excluding minterms 0, 2, 4, and 7, the following input (D_i) assignments are made

$$\begin{aligned} D_1 &= D_3 = \\ D_5 &= D_6 = 1 \\ D_0 &= D_2 = \\ D_4 &= D_7 = 0 \end{aligned}$$



$$F(A, B, C) = \sum (1, 3, 5, 6) A B C$$

Figure 5: Implementing function with Mux with n select inputs

Method 2: Using a Mux with (n-1) select inputs

Any n-variable logic function can be implemented using a Mux with only (n-1) select inputs (e.g. 4-to-1 mux to implement any 3 variable function)

This can be accomplished as follows:

Express function in canonical sum-of-minterms form.

Choose n-1 variables to be connected to the mux select lines.

Construct the truth table of the function, but grouping the n-1 select input variables together (e.g. by making the n-1 select variables as most significant inputs).

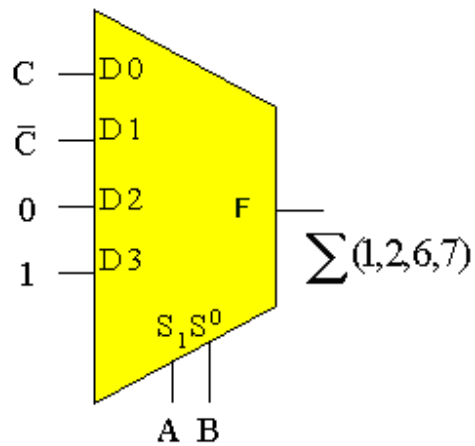
The values of D_j (mux input line) will be 0, or 1, or n^{th} variable or complement of n^{th} variable of value of function F, as will be clarified by the following example.

Example 5: Implement the function $F(A, B, C) = \sum (1, 2, 6, 7)$ (see figure 6) This function can be implemented with a 4-to-1 line MUX.

A and B are applied to the select line, that is

$$A = S_1, B = S_0$$

The truth table of the function and the implementation are as shown:



	A	B	C	F	
1	0	0	0	0	F = C
	0	0	1	1	
2	0	1	0	1	F = C'
	0	1	1	0	
3	1	0	0	0	F = 0
	0	1	1	0	
4	1	1	0	1	F = 1
	1	1	1	1	

Figure 6: Implementing function with Mux with n-1 select inputs

Example 6: Consider the function
 $F(A,B,C,D) = \sum(1,3,4,11,12,13,14,15)$

This function can be implemented with an 8-to-1 line MUX (see Figure 7) A, B, and C are applied to the select inputs as follows:

$$A \square S_2, B \square S_1, C \square S_0$$

The truth table and implementation are shown.

A	B	C	D	F	
0	0	0	0	0	F = D
0	0	0	1	1	
0	0	1	0	0	F = D
0	0	1	1	1	
0	1	0	0	1	F = D-bar
0	1	0	1	0	

Figure 7: Implementing function of Example 6

Example 7: A 1-to-4 line Demux

The input E is directed to one of the outputs, as specified by the two select lines S_1 and

S_0 .

$$D_0 = E \text{ if } S_1 S_0 = 00 \quad \square \quad D_0 = S_1' S_0' E$$

$$D_1 = E \text{ if } S_1 S_0 = 01 \quad \square \quad D_1 = S_1' S_0 E$$

$$D_2 = E \text{ if } S_1 S_0 = 10 \quad \square \quad D_2 = S_1 S_0' E$$

$$D_3 = E \text{ if } S_1 S_0 = 11 \quad \square \quad D_3 = S_1 S_0 E$$

A careful inspection of the Demux circuit shows that it is

identical to a 2 to 4 decoder with enable input.

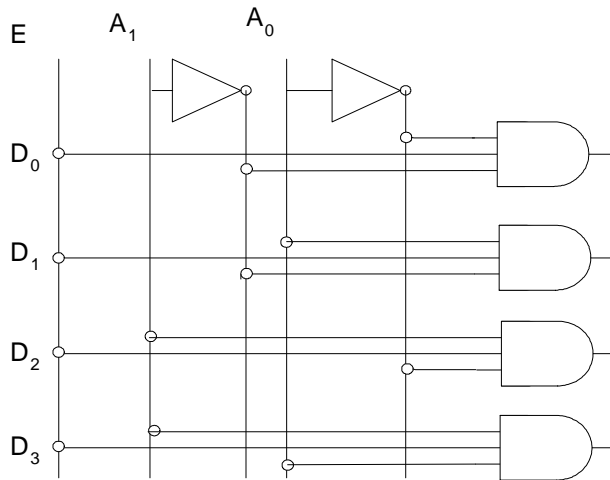


Figure 8: A 1-to-4 line demultiplexer

For the decoder, the inputs are A_1 and A_0 , and the enable is input E. (see figure 9)

For demux, input E provides the data, while other inputs accept the selection variables.

Although the two circuits have different applications, their logic diagrams are exactly the same.

Decimal value	Inputs			Outputs			
	E	A_1	A_0	D_0	D_1	D_2	D_3
	0	X	X	0	0	0	0
0	1	0	0	1	0	0	0
1	1	0	1	0	1	0	0
2	1	1	0	0	0	1	0
3	1	1	1	0	0	0	1

Figure 9: Table for 1-to-4 line demultiplexer

6.2 QUESTIONS:

1. Define Multiplexer.
2. Write short note on 4 –to- 1 Channel Multiplexer.
3. Explain 4- to -2 Channel Multiplexer With help of suitable diagram.
4. What is De-Multiplexer?
5. What is Digital Encoder? Explain 4- to -2 Bit Binary Encoder with help of suitable diagram.

6. What is Priority Encoder? Explain 8- to -3 Priority Encoder with the help of suitable diagram.
7. List and Explain Applications of Encoder.
8. Explain in detail Binary Decoder.
9. Implement the function $F(A,B,C,D)=\sum(1,3,5,8,12,13,16,18)$ using 8-to-1 line MUX.

6.3 FURTHER READING:

- ❖ **Digital Electronics** - An Introduction to Theory and Practice by W H Gothmann
- ❖ **Computer Architecture and Parallel Processing** by Kai Hwang, Faye A Briggs , McGraw Hill
- ❖ **Computer Architecture and Organization** by William Stallings
- ❖ **Fundamentals of Computer organization and Design** by Sivarama P. Dandamudi
- ❖ <http://en.wikipedia.org/wiki/Multiplexer>



SEQUENTIAL CIRCUITS

Unit structure

7.0 Objectives

7.1 What is sequential logic?

7.2 Flip-Flops

7.1.1 Rs Flip-Flop

7.1.2 J-K Flip-Flop

7.1.3 Master–Slave Flip-Flops

7.1.4 Toggle Flip-Flop (*T* Flip-Flop)

7.1.5 D Flip-Flop

7.1.6 D Latch

7.3 Questions

7.4 Further Reading

7.0 OBJECTIVES:

After completing this chapter, you will be able to:

- ❖ Understand the basics of Sequential Logic Circuits.
- ❖ Learn different types of Flips –Flops, their working and applications with the help of suitable diagrams.
- ❖ Understand the D Flip-Flop & D Latch with their functioning.

7.1 WHAT IS SEQUENTIAL LOGIC?

Sequential logic elements perform as many different functions as combinational logic elements; however, they do carry out certain well-denned functions, which have been given names.

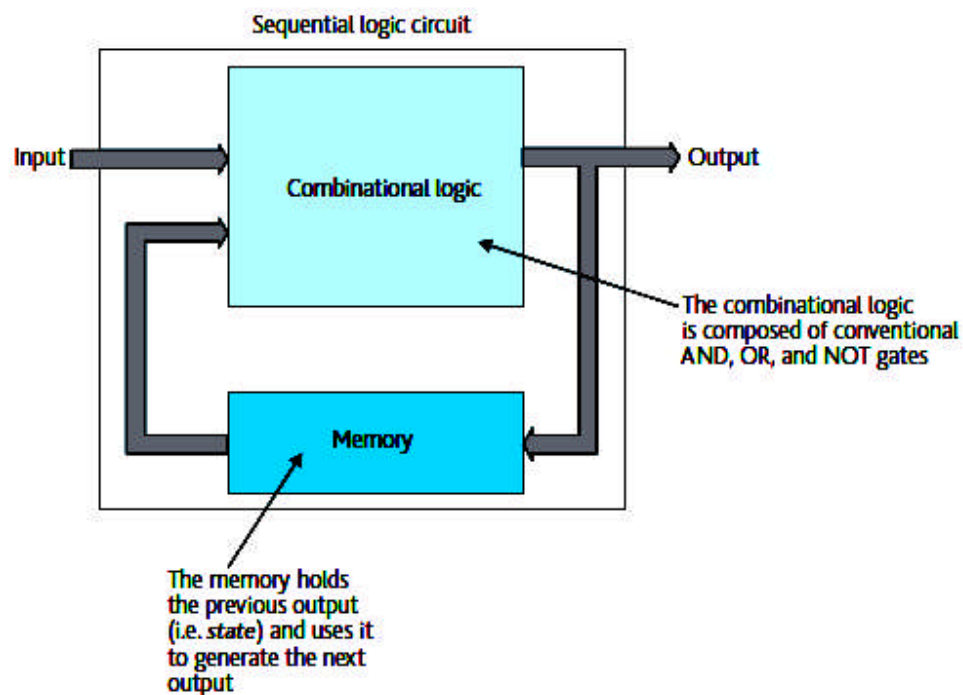
Latch A latch is a 1-bit memory element. You can capture a single bit in a latch at one instant and then use it later; for example, when adding numbers you can capture the carry-out in a latch and use it as a carry-in in the next calculation.

Register The register is just m latches in a row and is able to store an m -bit word; that is, the register is a device that stores

one memory word. A computer's memory is just a very large array of registers.

Shift register A shift register is a special-purpose register that can move the bits of the word it holds left or right; for example the 8-bit word 00101001 can be shifted left to give 01010010. Counter A counter is another special-purpose register that holds an m-bit word. However, when a counter is triggered (i.e. clocked) its contents increase by 1; for example, if a counter holding the binary equivalent of 42 is clocked, it will hold the value 43. Counters can count up or down, by 1 or any other number, or they can count through any arbitrary sequence.

State machines A state machine is a digital system that moves from one state to another each time it is triggered. You can regard a washing machine controller as a state machine that steps through all the processes involved in washing (at a rate depending on the load, the temperature, and its preselected functions). Ultimately, the computer itself is nothing more than a state machine controlled by a program and its data.



7.2 FILP-FLOPS

In electronics, a **flip-flop** or **latch** is a circuit that has two stable states and can be used to store state information. The circuit can be made to change state by signals applied to one or more control inputs and will have one or two outputs. It is the basic storage element in sequential logic. Flip-flops and latches are a

fundamental building block of digital electronics systems used in computers, communications, and many other types of systems.

Flip-flops and latches are used as data storage elements. Such data storage can be used for storage of *state*, and such a circuit is described as sequential logic. When used in a finite-state machine, the output and next state depend not only on its current input, but also on its current state (and hence, previous inputs). It can also be used for counting of pulses, and for synchronizing variably-timed input signals to some reference timing signal.

Flip-flops can be either simple (transparent or opaque) or clocked (synchronous or edge-triggered); the simple ones are commonly called latches. The word *latch* is mainly used for storage elements, while clocked devices are described as *flip-flops*.

7.2.1: RS Flip-Flop

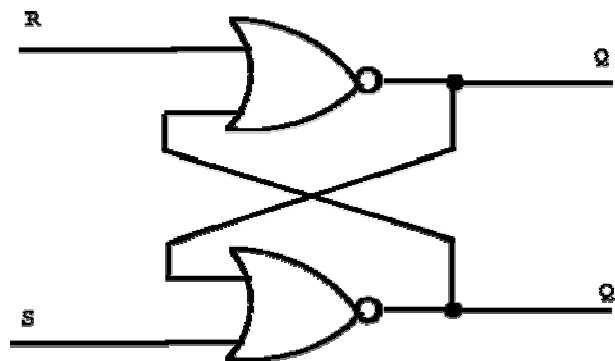
- A RS-flipflop is the simplest possible memory element.
- It is constructed by feeding the outputs of two NOR gates back to the other NOR gates input.
- The inputs R and S are referred to as the Reset and Set inputs, respectively.
- To understand the operation of the RS-flipflop (or RS-latch) consider the following scenarios:
 - **S=1 and R=0:** The output of the bottom NOR gate is equal to zero, $Q'=0$.
 - Hence both inputs to the top NOR gate are equal to one, thus, $Q=1$.
 - Hence, the input combination $S=1$ and $R=0$ leads to the flipflop being set to $Q=1$.
 - **S=0 and R=1:** Similar to the arguments above, the outputs become $Q=0$ and $Q'=1$.
 - We say that the flipflop is reset.
 - **S=0 and R=0:** Assume the flipflop is set ($Q=0$ and $Q'=1$), then the output of the top NOR gate remains at $Q=1$ and the bottom NOR gate stays at $Q'=0$.
 - Similarly, when the flipflop is in a reset state ($Q=1$ and $Q'=0$), it will remain there with this input combination.
 - Therefore, with inputs $S=0$ and $R=0$, the flipflop remains in its state.
 - **S=1 and R=1:** This input combination must be avoided.

- We can summarize the operation of the RS-flipflop by the following truth table.

R	S	Q	Q'	Comment
0	0	Q	Q'	Hold state
0	1	1	0	Set
1	0	0	1	Reset
1	1	?	?	Avoid

- Note, the output Q' is simply the inverse of Q.
- An RS flipflop can also be constructed from NAND gates.

RS Flip-Flop composed of two NOR Gates.



R-S Flip-Flop with Active LOW Inputs

The Figure below shows a NAND gate implementation of an R-S flip-flop with active LOW inputs. The two NAND gates are cross-coupled. That is, the output of NAND 1 is fed back to one of the inputs of NAND 2, and the output of NAND 2 is fed back to one of the inputs of NAND 1. The remaining inputs of NAND 1 and NAND 2 are the S and R inputs. The outputs of NAND 1 and NAND 2 are respectively Q and Q outputs.

The fact that this configuration follows the function table can be explained. We will look at different entries of the function table, one at a time.

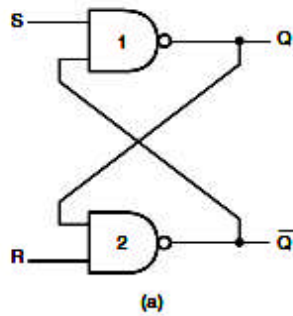
Let us take the case of $R = S = 1$ (the first entry in the function table). We will prove that, for $R = S = 1$, the Q output remains in its existing state. In the truth table, Q_n represents the existing state and Q_{n+1} represents the state of the flip-flop after it has been triggered by an appropriate pulse at the R or S input. Let us assume that $Q = 0$ initially. This '0' state fed back to one of the

inputs of gate 2 ensures that $Q = 1$. The '1' state of Q fed back to one of the inputs of gate 1 along with $S = 1$ ensures that $Q = 0$. Thus, $R = S = 1$ holds the existing stage. Now, if Q was initially in the '1' state and not the '0' state, this '1' fed back to one of the inputs of gate 2 along with $R = 1$ forces Q to be in the '0' state. The '0' state, when fed back to one of the inputs of gate 1, ensures that Q remains in its existing state of logic '1'. Thus, whatever the state of Q , $R = S = 1$ holds the existing state.

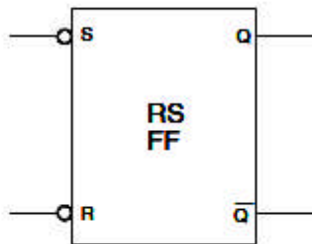
Let us now look at the second entry of the function table where $S = 0$ and $R = 1$. We can see that such an input combination forces the Q output to the '1' state. On similar lines, the input combination $S = 1$ and $R = 0$ (third entry of the truth table) forces the Q output to the '0' state. It would be interesting to analyze what happens when $S = R = 0$. This implies that both Q and \bar{Q} outputs should go to the '1' state, as one of the inputs of a NAND gate being a logic '0' should force its output to the logic '1' state irrespective of the status of the other input. This is an undesired state as Q and \bar{Q} outputs are to be the complement of each other. The input condition (i.e. $R=S=0$) that causes such a situation is therefore considered to be an invalid condition and is forbidden. Figure shows the logic symbol of such a flip-flop. The R and S inputs here have been shown as active LOW inputs, which is obvious as this flip-flop of Fig. 10.17(a) is SET (that is, $Q=1$) when $S=0$ and RESET (that is, $Q=0$) when $R=0$. Thus, R and S are active when LOW. The term CLEAR input is also used sometimes in place of RESET. The operation of the R-S flip-flop can be summarized as follows:

1. $SET=RESET=1$ is the normal resting condition of the flip-flop. It has no effect on the output state of the flip-flop. Both Q and \bar{Q} outputs remain in the logic state they were in prior to this input condition.
2. $SET=0$ and $RESET=1$ sets the flip-flop. Q and \bar{Q} respectively go to the '1' and '0' state.
3. $SET=1$ and $RESET=0$ resets or clears the flip-flop. Q and \bar{Q} respectively go to the '0' and '1' state.
4. $SET=RESET=0$ is forbidden as such a condition tries to set (that is, $Q=1$) and reset (that is, $Q=0$) the flip-flop at the same time. To be more precise, SET and RESET inputs in the R-S flip-flop cannot be active at the same time.

The R-S flip-flop is also referred to as an R-S latch. This is because any combination at the inputs immediately manifests itself at the output as per the truth table.



(a)



(b)

Operation Mode	S	R	Q_{n+1}
No change	1	1	Q_n
SET	0	1	1
RESET	1	0	0
Forbidden	0	0	—

(c)

Clocked R-S Flip-Flop

In the case of a clocked R-S flip-flop, or for that matter any clocked flip-flop, the outputs change states as per the inputs only on the occurrence of a clock pulse. The clocked flip-flop could be a level-triggered one or an edge-triggered one. The two types are discussed. First let us see how the flip-flop of the previous section can be transformed into a clocked flip-flop. Figure (a) shows the logic implementation of a clocked flip-flop that has active HIGH inputs. The function table for the same is shown in Fig. (b) and is self-explanatory.

The basic flip-flop is the same as that shown in Fig. (a). The two NAND gates at the input have been used to couple the R and S inputs to the flip-flop inputs under the control of the clock signal. When the clock signal is HIGH, the two NAND gates are enabled and the S and R inputs are passed on to flip-flop inputs with their status complemented. The outputs can now change states as per the status of R and S at the flip-flop inputs. For instance, when $S = 1$ and $R = 0$ it will be passed on as 0 and 1 respectively when the clock is HIGH. When the clock is LOW, the two NAND gates produce a '1' at their outputs, irrespective of the S and R status.

This produces a logic '1' at both inputs of the flip-flop, with the result that there is no effect on the output states. Figure shows the clocked R-S flip-flop with active LOW R and S inputs. The logic implementation here is a modification of the basic R-S flip-flop in Fig. The truth table of this flip-flop, as given in Fig., is self-explanatory.

Q _n	S	R	Q _{n+1}
0	0	0	Indeter
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	Indeter
1	0	1	1
1	1	0	0
1	1	1	1

(a)

Q _n	S	R	Q _{n+1}
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	Indeter
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	Indeter

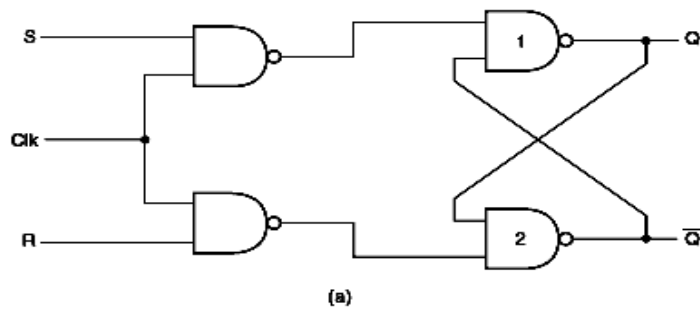
(b)

		SR			
		00	01	11	10
Q _n	0	X	1		
	1	X	1	1	

(c)

		SR			
		00	01	11	10
Q _n	0			X	1
	1	1		X	1

(d)

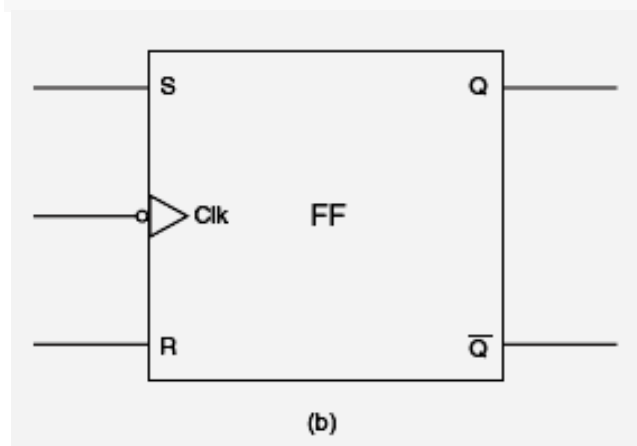
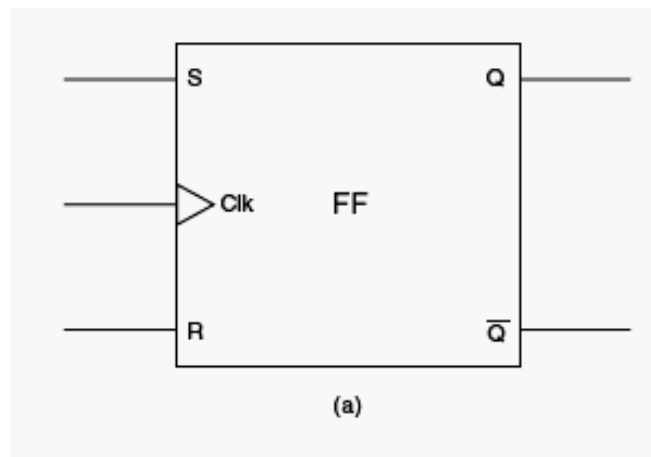
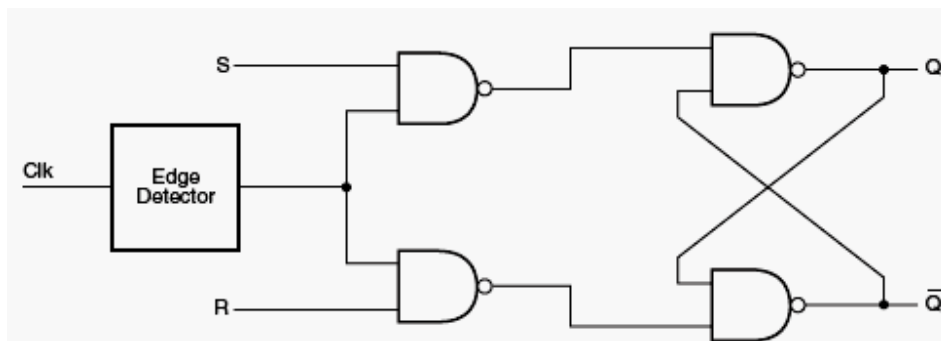


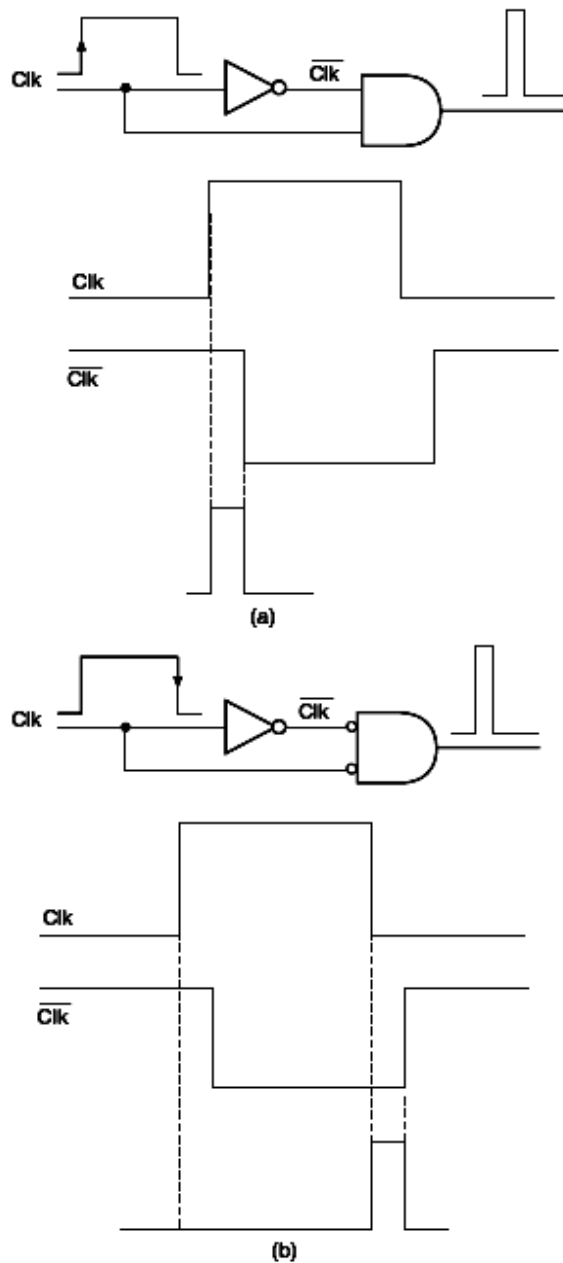
S	R	Clk	Q_{n+1}
0	0	0	Q_n
0	0	1	Q_n
0	1	0	Q_n
0	1	1	0
1	0	0	Q_n
1	0	1	1
1	1	0	Q_n
1	1	1	Invalid

Level-Triggered and Edge-Triggered Flip-Flops

In a *level-triggered* flip-flop, the output responds to the data present at the inputs during the time the clock pulse level is HIGH (or LOW). That is, any changes at the input during the time the clock is active (HIGH or LOW) are reflected at the output as per its function table. The clocked R-S flip-flop described is a level-triggered flip-flop that is active when the clock is HIGH.

In an *edge-triggered* flip-flop, the output responds to the data at the inputs only on LOW-to-HIGH or HIGH-to-LOW transition of the clock signal. The flip-flop in the two cases is referred to as positive edge triggered and negative edge triggered respectively. Any changes in the input during the time the clock pulse is HIGH (or LOW) do not have any effect on the output. In the case of an edge triggered flip-flop, an edge detector circuit transforms the clock input into a very narrow pulse that is a few nanoseconds wide. This narrow pulse coincides with either LOW-to-HIGH or HIGH-to-LOW transition of the clock input, depending upon whether it is a positive edge-triggered flip-flop or a negative edge-triggered flip-flop. This pulse is so narrow that the operation of the flip-flop can be considered to have occurred on the edge itself. Figure shows the clocked R-S flip-flop of Fig. with the edge detector block incorporated in the clock circuit. Figures (a) and (b) respectively show typical edge detector circuits for positive and negative edge triggering. The width of the narrow pulse generated by this edge detector circuit is equal to the propagation delay of the inverter. Figure 10.25 shows the circuit symbol for the flip-flop of Fig. for the positive edge-triggered mode and the negative edge-triggered mode.





7.2.2: J-K Flip-Flop

A J-K flip-flop behaves in the same fashion as an R-S flip-flop except for one of the entries in the function table. In the case of an R-S flip-flop, the input combination $S = R = 1$ (in the case of a flip-flop with active HIGH inputs) and the input combination $S = R = 0$ (in the case of a flip-flop with active LOW inputs) are prohibited. In the case of a J-K flip-flop with active HIGH inputs, the output of the flip-flop toggles, that is, it goes to the other state, for $J = K = 1$. The output toggles for $J = K = 0$ in the case of the flip-flop having active LOW inputs. Thus, a J-K flip-flop overcomes the problem of a forbidden input combination of the R-S flip-flop. Figures (a) and (b)

respectively show the circuit symbol of level-triggered J-K flip-flops with active HIGH and active LOW inputs, along with their function tables.

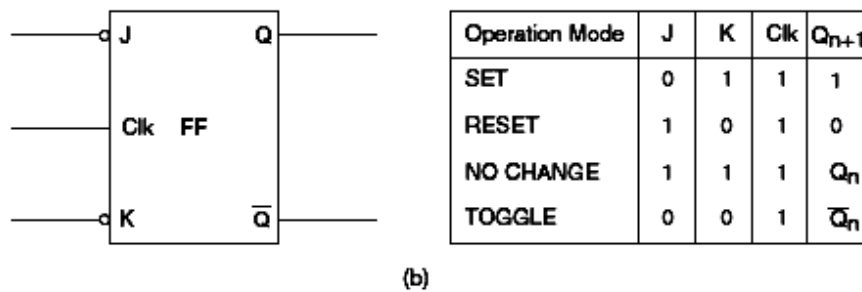
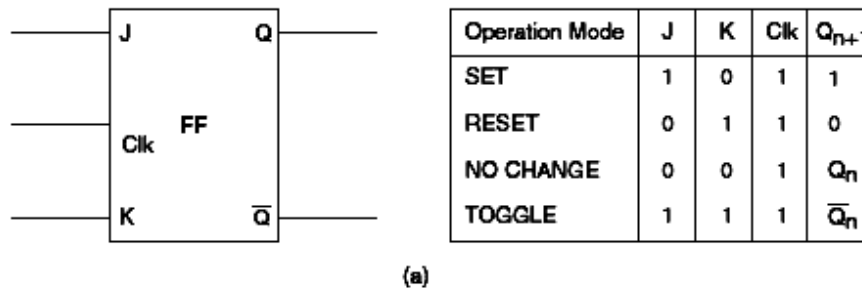
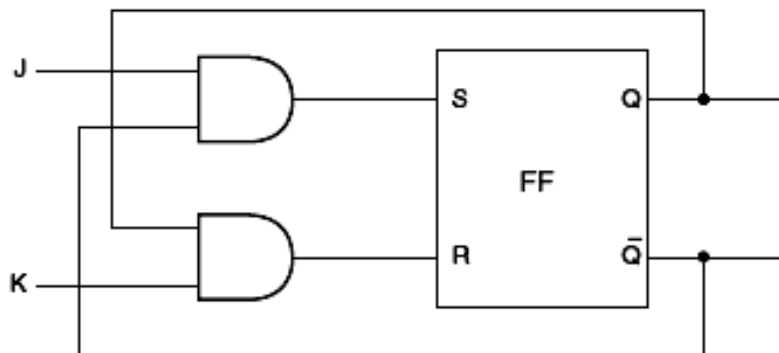


Figure shows the realization of a J-K flip-flop with an R-S flip-flop. The characteristic tables for a J-K flip-flop with active HIGH J and K inputs and a J-K flip-flop with active LOW J and K inputs are respectively shown in Figs (a) and (b). The corresponding Karnaugh maps are shown in Fig. (c) for the characteristics table of Fig. (a) and in Fig. (d) for the characteristic table of Fig. (b). The characteristic equations for the Karnaugh maps of Fig (c) and (d) are respectively

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

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



Q_n	J	K	Q_{n+1}
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

(a)

Q_n	J	K	Q_{n+1}
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

(b)

Q_n \ JK	JK			
	00	01	11	10
0			1	1
1	1			1

(c)

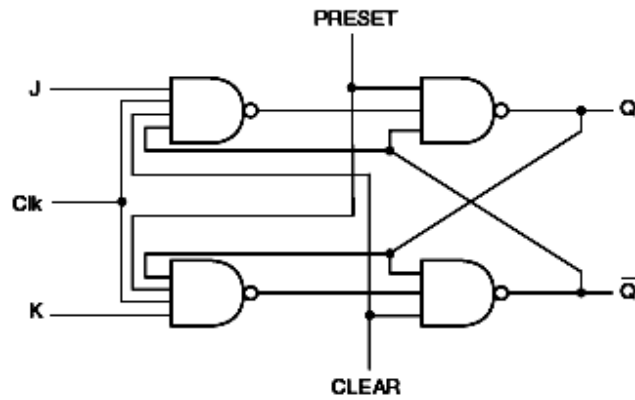
Q_n \ JK	JK			
	00	01	11	10
0	1	1		
1		1	1	

(d)

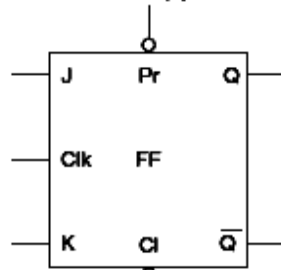
J-K Flip-Flop with PRESET and CLEAR Inputs

It is often necessary to clear a flip-flop to a logic '0' state ($Q_n = 0$) or preset it to a logic '1' state ($Q_n = 1$). An example of how this is realized is shown in Fig. The flip-flop is cleared (that is, $Q_n = 0$) whenever the CLEAR input is '0' and the PRESET input is '1'. The flip-flop is preset to the logic '1' state whenever the PRESET input is '0' and the CLEAR input is '1'. Here, the CLEAR and PRESET inputs are active when LOW. Figure shows the circuit symbol of this presettable, clearable, clocked J-K flip-flop. Figure shows the function table of such a flip-flop. It is evident from the function table that, whenever the PRESET input is active, the output goes to the '1' state irrespective of the status of the clock, J and K inputs.

Similarly, when the flip-flop is cleared, that is, the CLEAR input is active, the output goes to the '0' state irrespective of the status of the clock, J and K inputs. In a flip-flop of this type, both PRESET and CLEAR inputs should not be made active at the same time.



(a)



(b)

PR	CL	CLK	J	K	Q_{n+1}	\overline{Q}_{n+1}
0	1	X	X	X	1	0
1	0	X	X	X	0	1
0	0	X	X	X	--	--
1	1	1	0	0	Q_n	\overline{Q}_n
1	1	1	1	0	1	0
1	1	1	0	1	0	1
1	1	1	1	1	Toggle	
1	1	0	X	X	Q_n	\overline{Q}_n

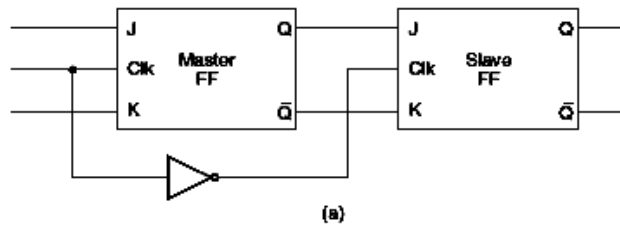
(c)

7.2.3: Master–Slave Flip-Flops

Whenever the width of the pulse clocking the flip-flop is greater than the propagation delay of the flip-flop, the change in state at the output is not reliable. In the case of edge-triggered flip-flops, this pulse width would be the trigger pulse width generated by the edge detector portion of the flip-flop and not the pulse width of the input clock signal. This phenomenon is referred to as the *race problem*. As the propagation delays are normally very small, the

likelihood of the occurrence of a race condition is reasonably high. One way to get over this problem is to use a *master–slave* configuration. Figure (a) shows a master–slave flip-flop constructed with two J-K flip-flops.

The first flip-flop is called the master flip-flop and the second is called the slave. The clock to the slave flip-flop is the complement of the clock to the master flip-flop. When the clock pulse is present, the master flip-flop is enabled while the slave flip-flop is disabled. As a result, the master flip-flop can change state while the slave flip-flop cannot. When the clock goes LOW, the master flip-flop gets disabled while the slave flip-flop is enabled. Therefore, the slave J-K flip-flop changes state as per the logic states at its J and K inputs. The contents of the master flip-flop are therefore transferred to the slave flip-flop, and the master flip-flop, being disabled, can acquire new inputs without affecting the output. As would be clear from the description above, a master–slave flip-flop is a pulse-triggered flip-flop and not an edge-triggered one. Figure (b) shows the truth table of a master–slave J-K flip-flop with active LOW PRESET and CLEAR inputs and active HIGH J and K inputs. The master–slave configuration has become obsolete. The newer IC technologies such as 74LS, 74AS, 74ALS, 74HC and 74HCT do not have master–slave flip-flops in their series.



PR	CLR	CLK	J	K	Q_{n+1}	\overline{Q}_{n+1}
0	1	X	X	X	1	0
1	0	X	X	X	0	1
0	0	X	X	X	Unstable	
1	1		0	0	Q_n	\overline{Q}_n
1	1		1	0	1	0
1	1		0	1	0	1
1	1		1	1	Toggle	

(b)

7.2.4: Toggle Flip-Flop (T Flip-Flop)

The output of a *toggle flip-flop*, also called a T flip-flop, changes state every time it is triggered at its T input, called the toggle input. That is, the output becomes ‘1’ if it was ‘0’ and ‘0’ if it was ‘1’. Figures (a) and (b) respectively show the circuit symbols of

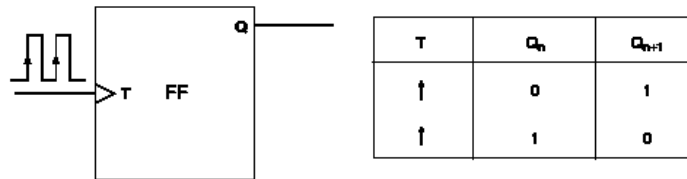
positive edge-triggered and negative edge-triggered T flip-flops, along with their function tables.

If we consider the T input as active when HIGH, the characteristic table of such a flip-flop is shown in Fig. (c). If the T input were active when LOW, then the characteristic table would be as shown in Fig. (d). The Karnaugh maps for the characteristic tables of Figs 10.34(c) and (d) are shown in Figs(e) and (f) respectively. The characteristic equations as written from the Karnaugh maps are as follows:

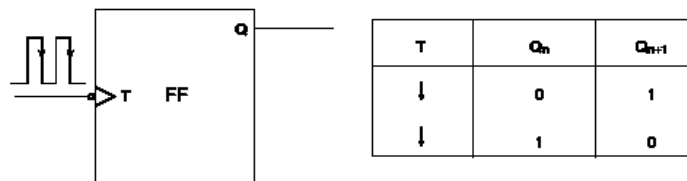
$$Q_{n+1} = T \cdot \overline{Q_n} + \overline{T} \cdot Q_n$$

$$Q_{n+1} = \overline{T} \cdot \overline{Q_n} + T \cdot Q_n$$

It is obvious from the operational principle of the T flip-flop that the frequency of the signal at the Q output is half the frequency of the signal applied at the T input. A cascaded arrangement of nT flip-flops, where the output of one flip-flop is connected to the T input of the following flip-flop, can be used to divide the input signal frequency by a factor of 2^n . Figure shows a divide-by-16 circuit built around a cascaded arrangement of four T flip-flops.



(a)



(b)

Q_n	T	Q_{n+1}
0	0	0
0	1	1
1	0	1
1	1	0

(c)

Q_n	T	Q_{n+1}
0	0	1
0	1	0
1	0	0
1	1	1

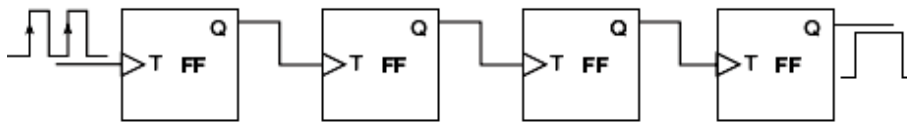
(d)

T		0	1
Q _n	0		1
1		1	

(e)

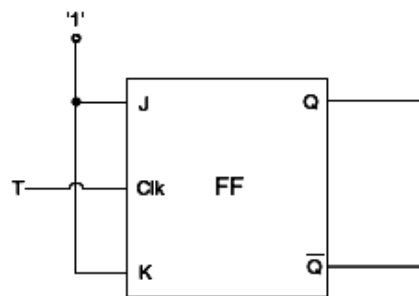
T		0	1
Q _n	0	1	
1			1

(f)



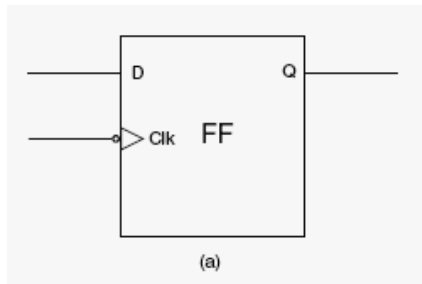
J-K Flip-Flop as a Toggle Flip-Flop

If we recall the function table of a J-K flip-flop, we will see that, when both J and K inputs of the flip-flop are tied to their active level ('1' level if J and K are active when HIGH, and '0' level when J and K are active when LOW), the flip-flop behaves like a toggle flip-flop, with its clock input serving as the T input. In fact, the J-K flip-flop can be used to construct any other flip-flop. That is why it is also sometimes referred to as a universal flip-flop. Figure shows the use of a J-K flip-flop as a T flip-flop.



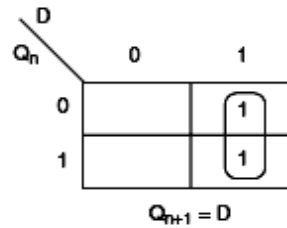
7.2.5: D Flip-Flop

A D flip-flop, also called a *delay flip-flop*, can be used to provide temporary storage of one bit of information. Figure (a) shows the circuit symbol and function table of a negative edge-triggered D flip-flop. When the clock is active, the data bit (0 or 1) present at the D input is transferred to the output. In the D flip-flop of Fig., the data transfer from D input to Q output occurs on the negative-going (HIGH-to-LOW) transition of the clock input. The D input can acquire new status



D	Clk	Q
0		0
1		1

Q_n	D	Q_{n+1}
0	0	0
0	1	1
1	0	0
1	1	1



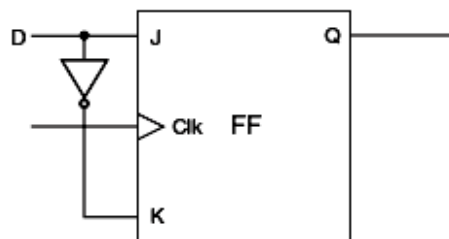
(c)

(d)

J-K Flip-Flop as D Flip-Flop

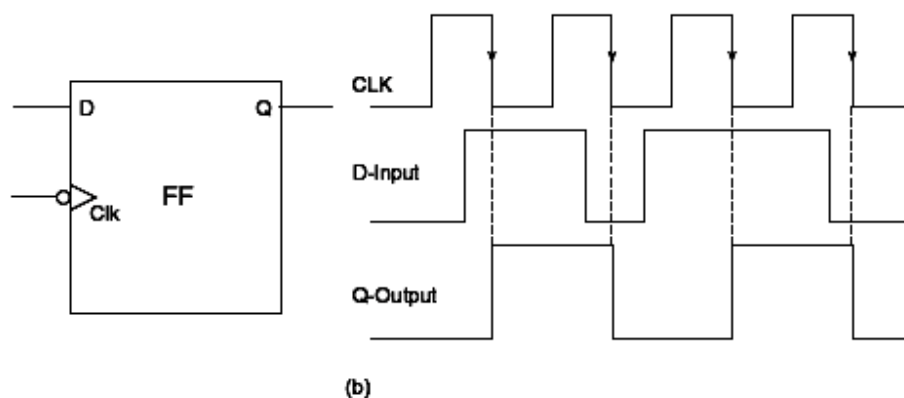
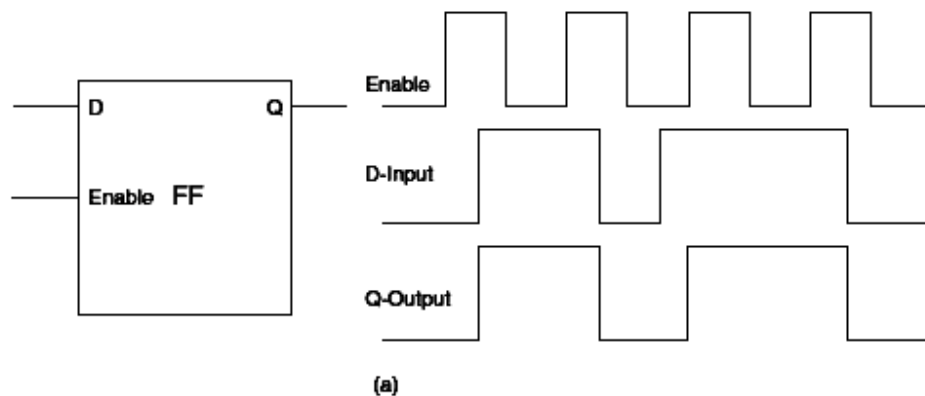
Figure shows how a J-K flip-flop can be used as a D flip-flop. When the D input is a logic '1', the J and K inputs are a logic '1' and '0' respectively. According to the function table of the J-K flip-flop, under these input conditions, the Q output will go to the logic '1' state when clocked.

Also, when the D input is a logic '0', the J and K inputs are a logic '0' and '1' respectively. Again, according to the function table of the J-K flip-flop, under these input conditions, the Q output will go to the logic '0' state when clocked. Thus, in both cases, the D input is passed on to the output when the flip-flop is clocked.



7.2.6 D Latch

In a D latch, the output Q follows the D input as long as the clock input (also called the ENABLE input) is HIGH or LOW, depending upon the clock level to which it responds. When the ENABLE input goes to the inactive level, the output holds on to the logic state it was in just prior to the ENABLE input becoming inactive during the entire time period the ENABLE input is inactive.



A D flip-flop should not be confused with a D latch. In a D flip-flop, the data on the D input are transferred to the Q output on the positive- or negative-going transition of the clock signal, depending upon the flip-flop, and this logic state is held at the output until we get the next effective clock transition. The difference between the two is further illustrated in Figs (a) and (b) depicting the functioning of a D latch and a D flip-flop respectively.

7.3 QUESTIONS:

1. What is Sequential Logic?
2. What is Shift Register?
3. Explain state machine with help of suitable diagram.
4. What is Flip-Flop? Where it is used?
5. Write detail Note on RS Flip-Flop.
6. Explain Level- Triggered and Edged –Triggered Flip-Flops.
7. Explain J-K Flip- Flop with help of suitable diagram.
8. Write short note on Master –Slave Flip-Flop.
9. Explain T Flip-Flop.
10. Explain the J-K Flip-Flop as D Flip-Flop.
11. Write Short note on D Latch.

7.4 FURTHER READING:

- ❖ **Digital Electronics** - An Introduction to Theory and Practice by W H Gothmann
- ❖ **Computer Architecture and Parallel Processing** by Kai Hwang, Faye A Briggs , McGraw Hill
- ❖ **Computer Architecture and Organization** by William Stallings
- ❖ **Fundamentals of Computer organization and Design** by Sivarama P. Dandamudi



COUNTERS AND REGISTERS

Unit Structure

8.0 Objectives

8.1 Introduction

8.2 Counters

8.2.1: Ripple (Asynchronous) Counter

8.2.2: Binary Ripple Counter

8.2.3: Synchronous Counter

8.2.4: UP/DOWN Counters

8.2.5: Presetable Counters

8.3 Shift Registers

8.3.1: Serial-in to Parallel-out (SIPO) 4-bit Serial-in to Parallel-out Shift Register

8.3.2: 4-bit Serial-in to Serial-out Shift Register

8.3.3: 4-bit Parallel-in to Serial-out Shift Register

8.3.4: 4-bit Parallel-in to Parallel-out Shift Register

8.3.5: Summary of Shift Registers

8.4 Questions

8.5 Further Reading

8.0 OBJECTIVES:

After completing this chapter, you will be able to:

- ❖ Understand the electronics parts like counters & Shift Registers.
- ❖ Understand the structure and working of Asynchronous counters with help of suitable diagrams.
- ❖ Understand the structure and working of Synchronous counters with help of suitable diagrams.
- ❖ Learn the basics of Shift Register of different types with help of appropriate diagrams.

8.1 INTRODUCTION:

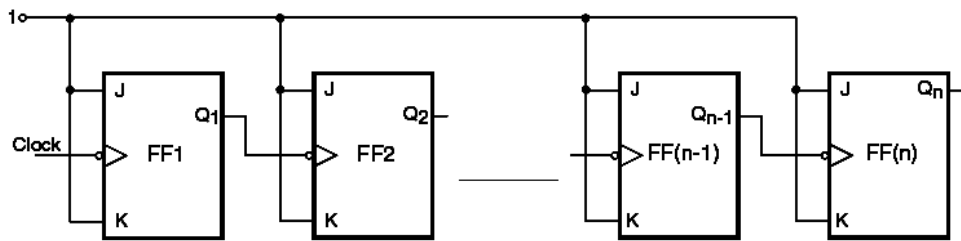
Counters and *registers* belong to the category of MSI sequential logic circuits. They have similar architecture, as both counters and registers comprise a cascaded arrangement of more than one flip-flop with or without combinational logic devices. Both constitute very important building blocks of sequential logic, and different types of counter and register available in integrated circuit (IC) form are used in a wide range of digital systems. While counters are mainly used in counting applications, where they either measure the time interval between two unknown time instants or measure the frequency of a given signal, registers are primarily used for the temporary storage of data present at the output of a digital circuit before they are fed to another digital circuit. We are all familiar with the role of different types of register used inside a microprocessor, and also their use in microprocessor-based applications. Because of the very nature of operation of registers, they form the basis of a very important class of counters called *shift counters*.

8.2 COUNTERS:

8.2.1: Ripple (Asynchronous) Counter

A *ripple counter* is a cascaded arrangement of flip-flops where the output of one flip-flop drives the clock input of the following flip-flop. The number of flip-flops in the cascaded arrangement depends upon the number of different logic states that it goes through before it repeats the sequence, a parameter known as the modulus of the counter.

In a ripple counter, also called an *asynchronous counter* or a *serial counter*, the clock input is applied only to the first flip-flop, also called the input flip-flop, in the cascaded arrangement. The clock input to any subsequent flip-flop comes from the output of its immediately preceding flip-flop. For instance, the output of the first flip-flop acts as the clock input to the second flip-flop, the output of the second flip-flop feeds the clock input of the third flip-flop and so on. In general, in an arrangement of n flip-flops, the clock input to the n th flip-flop comes from the output of the $(n - 1)^{\text{th}}$ flip-flop for $n > 1$. Figure shows the generalized block schematic arrangement of an n -bit binary ripple counter.



As a natural consequence of this, not all flip-flops change state at the same time. The second flip-flop can change state only after the output of the first flip-flop has changed its state. That is, the second flip-flop would change state a certain time delay after the occurrence of the input clock pulse owing to the fact that it gets its own clock input from the output of the first flip-flop and not from the input clock. This time delay here equals the sum of propagation delays of two flip-flops, the first and the second flip-flops. In general, the n th flip-flop will change state only after a delay equal to n times the propagation delay of one flip-flop. The term 'ripple counter' comes from the mode in which the clock information ripples through the counter. It is also called an 'asynchronous counter' as different flip-flops comprising the counter do not change state in synchronization with the input clock. In a counter like this, after the occurrence of each clock input pulse, the counter has to wait for a time period equal to the sum of propagation delays of all flip-flops before the next clock pulse can be applied. The propagation delay of each flip-flop, of course, will depend upon the logic family to which it belongs.

Modulus of a Counter

The *modulus* (MOD number) of a counter is the number of different logic states it goes through before it comes back to the initial state to repeat the count sequence. An n -bit counter that counts through all its natural states and does not skip any of the states has a modulus of 2^n . We can see that such counters have a modulus that is an integral power of 2, that is, 2, 4, 8, 16 and so on. These can be modified with the help of additional combinational logic to get a modulus of less than 2^n .

To determine the number of flip-flops required to build a counter having a given modulus, identify the smallest integer m that is either equal to or greater than the desired modulus and is also equal to an integral power of 2. For instance, if the desired modulus is 10, which is the case in a decade counter, the smallest integer greater than or equal to 10 and which is also an integral power of 2 is 16. The number of flip-flops in this case would be 4, as $16 = 2^4$. On the same lines, the number of flip-

flops required to construct counters with MOD numbers of 3, 6, 14, 28 and 63 would be 2, 3, 4, 5 and 6 respectively. In general, the arrangement of a minimum number of N flip-flops can be used to construct any counter with a modulus given by the equation

$$2^N - 1 + 1 \leq \text{modulus} \leq 2^N$$

8.2.2: Binary Ripple Counter

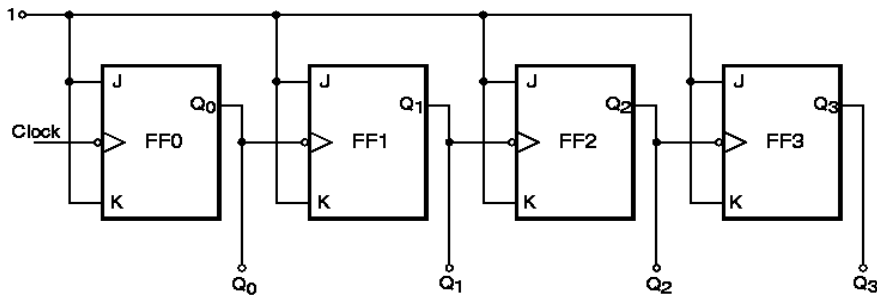
The operation of a binary ripple counter can be best explained with the help of a typical counter of this type. Figure (a) shows a four-bit ripple counter implemented with negative edge-triggered J - K flip-flops wired as toggle flip-flops. The output of the first flip-flop feeds the clock input of the second, and the output of the second flip-flop feeds the clock input of the third, the output of which in turn feeds the clock input of the fourth flip-flop. The outputs of the four flip-flops are designated as Q_0 (LSB flip-flop), Q_1 , Q_2 and Q_3 (MSB flip-flop). Figure (b) shows the waveforms appearing at Q_0 , Q_1 , Q_2 and Q_3 outputs as the clock signal goes through successive cycles of trigger pulses. The counter functions as follows.

Let us assume that all the flip-flops are initially cleared to the '0' state. On HIGH-to-LOW transition of the first clock pulse, Q_0 goes from '0' to '1' owing to the toggling action. As the flip-flops used are negative edge-triggered ones, the '0' to '1' transition of Q_0 does not trigger flip-flop FF1. FF1, along with FF2 and FF3, remains in its '0' state. So, on the occurrence of the first negative-going clock transition, $Q_0 = 1$, $Q_1 = 0$, $Q_2 = 0$ and $Q_3 = 0$.

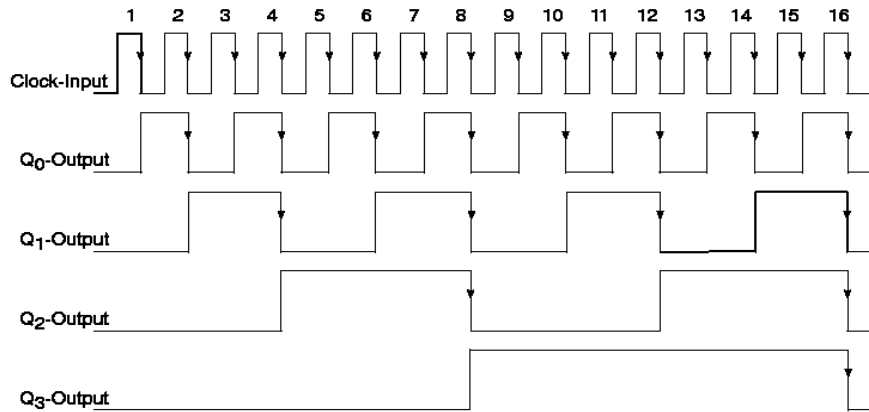
On the HIGH-to-LOW transition of the second clock pulse, Q_0 toggles again. That is, it goes from '1' to '0'. This '1' to '0' transition at the Q_0 output triggers FF1, the output Q_1 of which goes from '0' to '1'. The Q_2 and Q_3 outputs remain unaffected. Therefore, immediately after the occurrence of the second HIGH-to-LOW transition of the clock signal, $Q_0 = 0$, $Q_1 = 1$, $Q_2 = 0$ and $Q_3 = 0$. On similar lines, we can explain the logic status of Q_0 , Q_1 , Q_2 and Q_3 outputs immediately after subsequent clock transitions. The logic status of outputs for the first 16 relevant (HIGH-to-LOW in the present case) clock signal transitions is summarized in Table.

Thus, we see that the counter goes through 16 distinct states from 0000 to 1111 and then, on the occurrence of the desired transition of the sixteenth clock pulse, it resets to the original state of 0000 from where it had started. In general, if we had N flip-flops, we could count up to 2^N pulses before the

counter resets to the initial state. We can also see from the Q_0 , Q_1 , Q_2 and Q_3 waveforms, as shown



(a)



(b)

Clock signal transition number	Q_0	Q_1	Q_2	Q_3
After first clock transition	1	0	0	0
After second clock transition	0	1	0	0
After third clock transition	1	1	0	0
After fourth clock transition	0	0	1	0
After fifth clock transition	1	0	1	0
After sixth clock transition	0	1	1	0
After seventh clock transition	1	1	1	0
After eighth clock transition	0	0	0	1
After ninth clock transition	1	0	0	1
After tenth clock transition	0	1	0	1
After eleventh clock transition	1	1	0	1
After twelfth clock transition	0	0	1	1
After thirteenth clock transition	1	0	1	1
After fourteenth clock transition	0	1	1	1
After fifteenth clock transition	1	1	1	1
After sixteenth clock transition	0	0	0	0

in Fig. 11.2(b), that the frequencies of the Q_0 , Q_1 , Q_2 and Q_3 waveforms are $f/2$, $f/4$, $f/8$ and $f/16$ respectively. Here, f is the frequency of the clock input. This implies that a counter of this type can be used as a divide-by- 2^N circuit, where N is the number of flip-flops in the counter chain. In fact, such a counter provides frequency-divided outputs of $f/2^N$, $f/2^{N-1}$, $f/2^{N-2}$, $f/2^{N-3}$, ..., $f/2$ at the outputs of the N th, $(N-1)$ th, $(N-2)$ th, $(N-3)$ th, ..., first flip-flops. In the case of a four-bit counter of the type shown in Fig. 11.2(a), outputs are available at $f/2$ from the Q_0 output, at $f/4$ from the Q_1 output, at $f/8$ from the Q_2 output and at $f/16$ from the Q_3 output. It may be noted that frequency division is one of the major applications of counters.

Binary Ripple Counters with a Modulus of Less than 2^N

An N -flip-flop binary ripple counter can be modified, as we will see in the following paragraphs, to have any other modulus less than 2^N with the help of simple externally connected combinational logic. We will illustrate this simple concept with the help of an example.

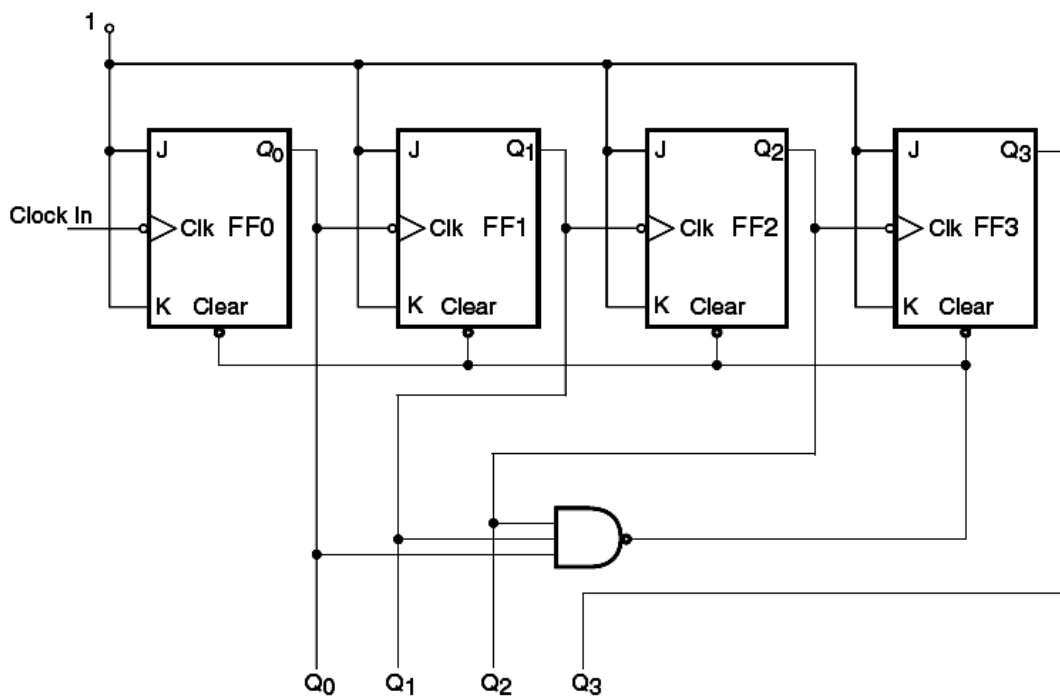
Consider the four-flip-flop binary ripple counter arrangement of Fig. (a). It uses J - K flip-flops with an active LOW asynchronous CLEAR input. The NAND gate in the figure has its output connected to the CLEAR inputs of all four flip-flops. The inputs to this three-input NAND gate are from the Q outputs of flip-flops FF0, FF1 and FF2. If we disregard the NAND gate for some time, this counter will go through its natural binary sequence from 0000 to 1111. But that is not to happen in the present arrangement. The counter does start counting from 0000 towards its final count of 1111. The counter keeps counting as long as the asynchronous CLEAR inputs of the different flip-flops are inactive. That is, the NAND gate output is HIGH. This is the case until the counter reaches 0110. With the seventh clock pulse it tends to go to 0111, which makes all NAND gate inputs HIGH, forcing its output to LOW. This HIGH-to-LOW transition at the NAND gate output clears all flip-flop outputs to the logic '0' state, thus disallowing the counter to settle at 0111. From the eighth clock pulse onwards, the counter repeats the sequence. The counter thus always counts from 0000 to 0110 and resets back to 0000. The remaining nine states, which include 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110 and 1111, are skipped, with the result that we get an MOD-7 counter.

Figure (b) shows the timing waveforms for this counter. By suitably choosing NAND inputs, one can get a counter with any MOD number less than 16. Examination of timing waveforms also

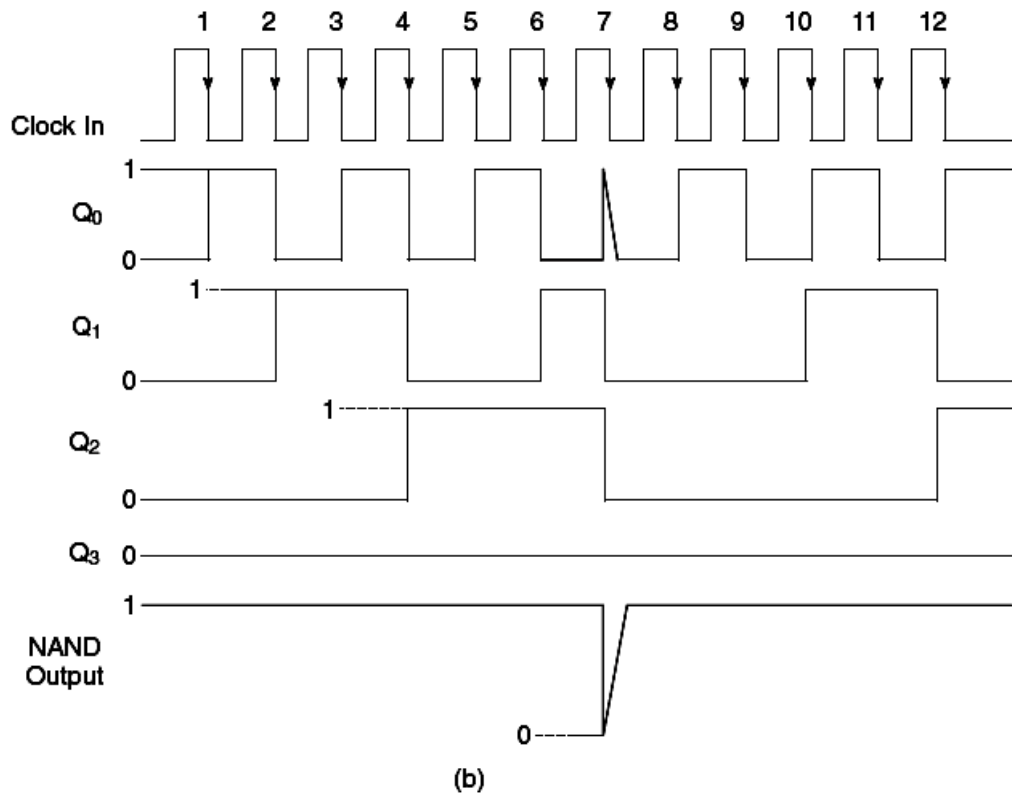
reveals that the frequency of the Q_2 output is one-seventh of the input clock frequency.

The waveform at the Q_2 output is, however, not symmetrical as it would be if the counter were to go through its full binary sequence. The Q_3 output stays in the logic LOW state. It is expected to be so because an MOD-7 counter needs a minimum of three flip-flops. That is why the fourth flip-flop, which was supposed to toggle on the HIGH-to-LOW transition of the eighth clock pulse, and on every successive eighth pulse thereafter, never gets to that stage. The counter is cleared on the seventh clock pulse and every successive seventh clock pulse thereafter.

As another illustration, if the NAND gate used in the counter arrangement of Fig. (a) is a two-input NAND and its inputs are from the Q_1 and Q_3 outputs, the counter will go through 0000 to 1001 and then reset to 0000 again, as, the moment the counter tends to switch from the 1001 to the 1010 state, the NAND gate goes from the '1' to the '0' state, clearing all flip-flops to the '0' state.



(a)



Steps to be followed to design any binary ripple counter that starts from 0000 and has a modulus of X are summarized as follows:

1. Determine the minimum number of flip-flops N so that $2^N \geq X$. Connect these flip-flops as a binary ripple counter. If $2^N = X$, do not go to steps 2 and 3.
2. Identify the flip-flops that will be in the logic HIGH state at the count whose decimal equivalent is X. Choose a NAND gate with the number of inputs equal to the number of flip-flops that would be in the logic HIGH state. As an example, if the objective were to design an MOD-12 counter, then, in the corresponding count, that is, 1100, two flip-flops would be in the logic HIGH state. The desired NAND gate would therefore be a two-input gate.
3. Connect the Q outputs of the identified flip-flops to the inputs of the NAND gate and the NAND gate output to asynchronous clear inputs of all flip-flops.

8.2.3: Synchronous Counter

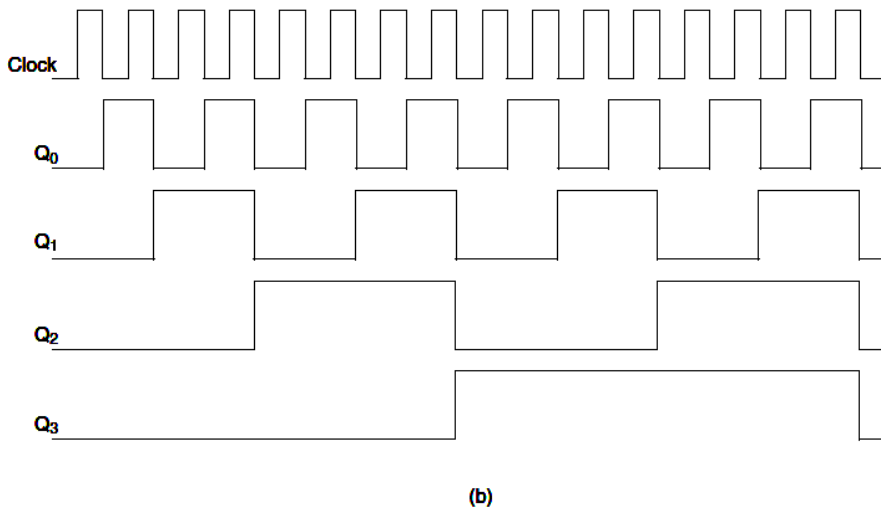
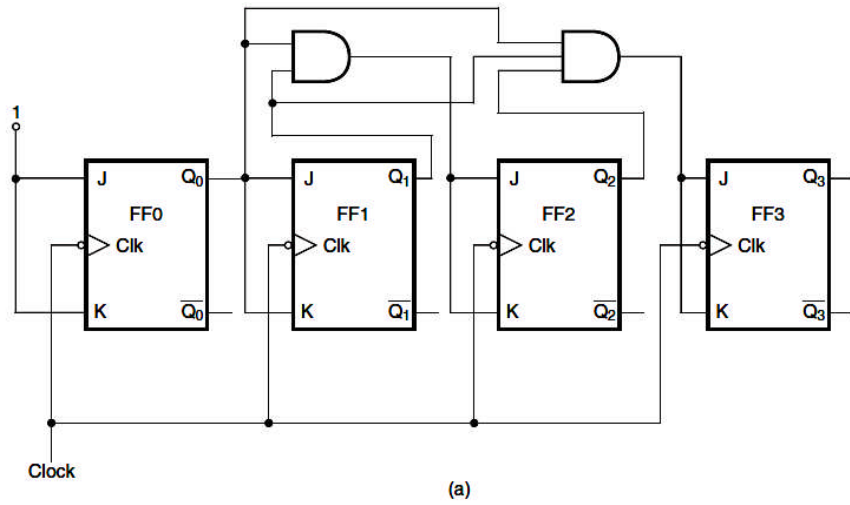
In a *synchronous counter*, also known as a *parallel counter*, all the flip-flops in the counter change state at the same time in synchronism with the input clock signal. The clock signal in this case is simultaneously applied to the clock inputs of all the flip-flops. The delay involved in this case is equal to the propagation

delay of one flip-flop only, irrespective of the number of flip-flops used to construct the counter. In other words, the delay is independent of the size of the counter.

Ripple counters discussed thus far in this chapter are asynchronous in nature as the different flipflops comprising the counter are not clocked simultaneously and in synchronism with the clock pulses. The total propagation delay in such a counter, as explained earlier, is equal to the sum of propagation delays due to different flip-flops. The propagation delay becomes prohibitively large in a ripple counter with a large count. On the other hand, in a synchronous counter, all flip-flops in the counter are clocked simultaneously in synchronism with the clock, and as a consequence all flip-flops change state at the same time. The propagation delay in this case is independent of the number of flip-flops used.

Since the different flip-flops in a synchronous counter are clocked at the same time, there needs to be additional logic circuitry to ensure that the various flip-flops toggle at the right time. For instance, if we look at the count sequence of a four-bit binary counter shown in Table, we find that flip-flop FF0 toggles with every clock pulse, flip-flop FF1 toggles only when the output of FF0 is in the '1' state, flip-flop FF2 toggles only with those clock pulses when the outputs of FF0 and FF1 are both in the logic '1' state and flip-flop FF3 toggles only with those clock pulses when Q0, Q1 and Q2 are all in the logic '1' state. Such logic can be easily implemented with AND gates. Figure (a) shows the schematic arrangement of a four-bit synchronous counter. The timing waveforms are shown in Fig. (b). The diagram is self-explanatory. As an example, ICs 74162 and 74163 are four-bit synchronous counters, with the former being a decade counter and the latter a binary counter.

A synchronous counter that counts in the reverse or downward sequence can be constructed in a similar manner by using complementary outputs of the flip-flops to drive the J and K inputs of the following flip-flops. Refer to the reverse or downward count sequence as given in Table. As is evident from the table, FF0 toggles with every clock pulse, FF1 toggles only when Q0 is logic '0', FF2 toggles only when both Q0 and Q1 are in the logic '0' state and FF3 toggles only when Q0, Q1 and Q2 are in the logic '0' state. Referring to the four-bit synchronous UP counter of Fig. (a), if the J and K inputs of flip-flop FF1 are fed from the Q0 output instead of the Q0 output, the inputs to the two-input AND gate are and instead of Q0 and Q1, and the inputs to the three-input AND gate are, and instead of Q0, Q1 and Q2, we get a counter that counts in reverse order. In that case it becomes a four-bit synchronous DOWN counter.



Count	Q ₃	Q ₂	Q ₁	Q ₀	Count	Q ₃	Q ₂	Q ₁	Q ₀
0	0	0	0	0	8	1	0	0	0
1	1	1	1	1	9	0	1	1	1
2	1	1	1	0	10	0	1	1	0
3	1	1	0	1	11	0	1	0	1
4	1	1	0	0	12	0	1	0	0
5	1	0	1	1	13	0	0	1	1
6	1	0	1	0	14	0	0	1	0
7	1	0	0	1	15	0	0	0	1

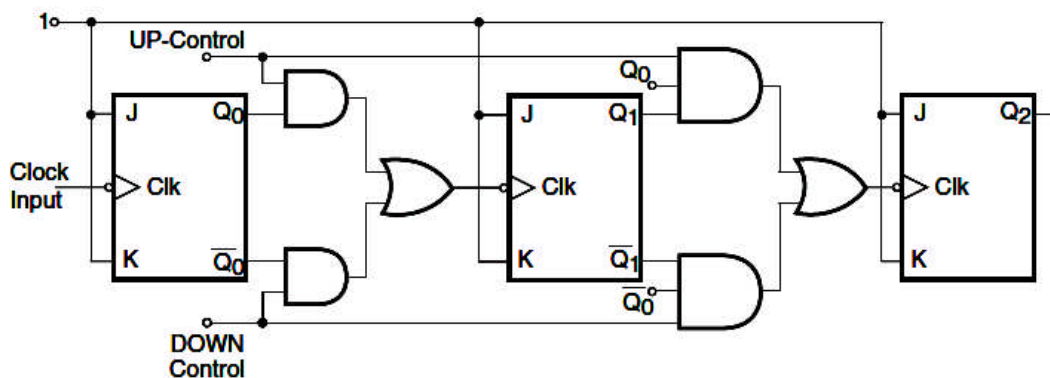
8.2.4: UP/DOWN Counters

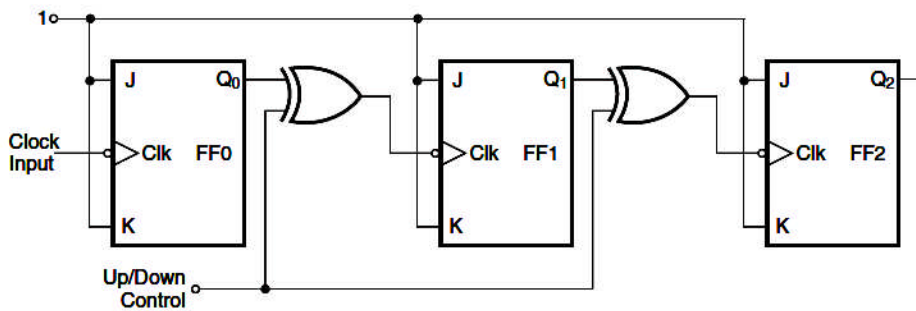
Counters are also available in integrated circuit form as UP/DOWN counters, which can be made to operate as either UP or DOWN counters. As outlined in Section 11.5, an UP counter is one that counts upwards or in the forward direction by one LSB

every time it is clocked. A four-bit binary UP counter will count as 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111, 0000, 0001 and so on. A DOWN counter counts in the reverse direction or downwards by one LSB every time it is clocked. The four-bit binary DOWN counter will count as 0000, 1111, 1110, 1101, 1100, 1011, 1010, 1001, 1000, 0111, 0110, 0101, 0100, 0011, 0010, 0001, 0000, 1111 and so on.

Some counter ICs have separate clock inputs for UP and DOWN counts, while others have a single clock input and an UP/DOWN control pin. The logic status of this control pin decides the counting mode. As an example, ICs 74190 and 74191 are four-bit UP/DOWN counters in the TTL family with a single clock input and an UP/DOWN control pin. While IC 74190 is a BCD decade counter, IC 74191 is a binary counter. Also, ICs 74192 and 74193 are four-bit UP/DOWN counters in the TTL family, with separate clock input terminals for UP and DOWN counts. While IC 74192 is a BCD decade counter, IC 74193 is a binary counter.

Figure shows a three-bit binary UP/DOWN counter. This is only one possible logic arrangement. As we can see, the counter counts upwards when UP control is logic '1' and DOWN control is logic '0'. In this case the clock input of each flip-flop other than the LSB flip-flop is fed from the normal output of the immediately preceding flip-flop. The counter counts downwards when the UP control input is logic '0' and DOWN control is logic '1'. In this case, the clock input of each flip-flop other than the LSB flip-flop is fed from the complemented output of the immediately preceding flip-flop. Figure shows another possible configuration for a three-bit binary ripple UP/DOWN counter. It has a common control input. When this input is in logic '1' state the counter counts downwards, and when it is in logic '0' state it counts upwards.





8.2.5: Presetable Counters

Presetable counters are those that can be preset to any starting count either asynchronously (independently of the clock signal) or synchronously (with the active transition of the clock signal). The presetting operation is achieved with the help of PRESET and CLEAR (or MASTER RESET) inputs available on the flip-flops. The presetting operation is also known as the 'preloading' or simply the 'loading' operation.

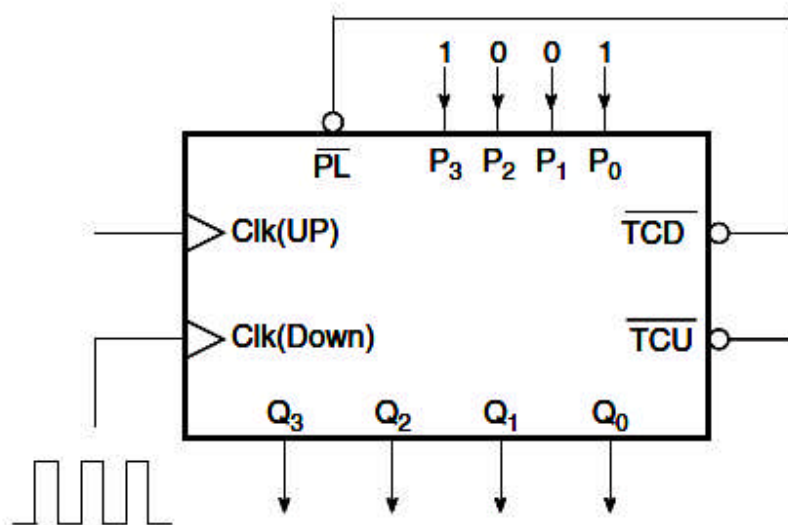
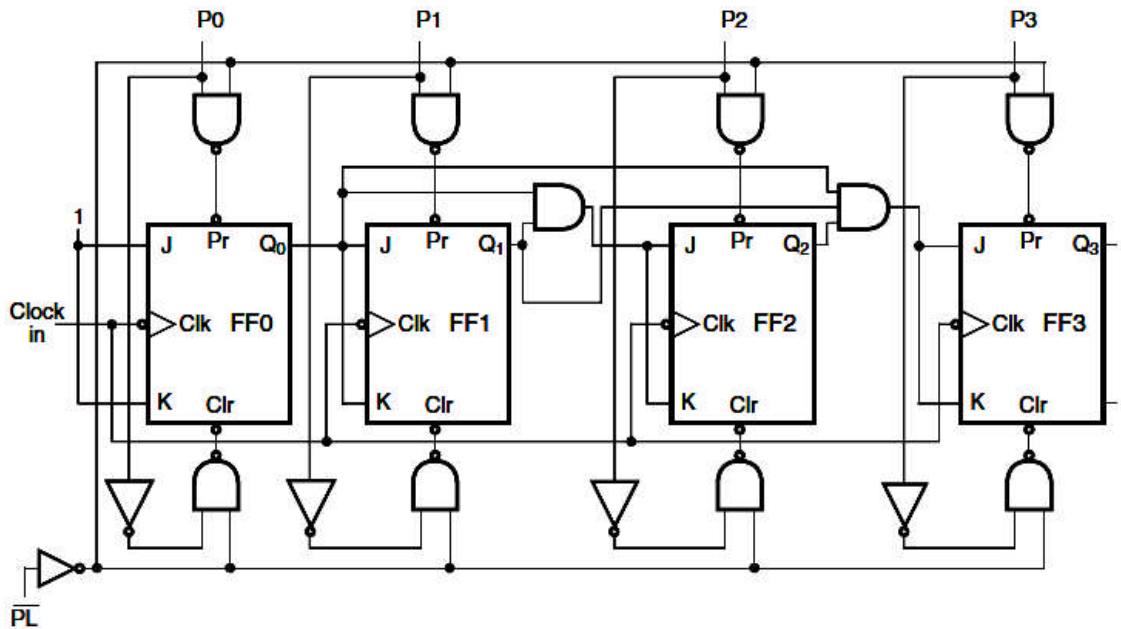
Presetable counters can be UP counters, DOWN counters or UP/DOWN counters. Additional inputs/outputs available on a presetable UP/DOWN counter usually include PRESET inputs, from where any desired count can be loaded, parallel load (*PL*) inputs, which when active allow the PRESET inputs to be loaded onto the counter outputs, and terminal count (*TC*) outputs, which become active when the counter reaches the terminal count.

Figure shows the logic diagram of a four-bit presetable synchronous UP counter. The data available on P_3 , P_2 , P_1 and P_0 inputs are loaded onto the counter when the parallel load input goes LOW.

When the parallel load input goes LOW, one of the inputs of all NAND gates, including the four NAND gates connected to the PRESET inputs and the four NAND gates connected to the CLEAR inputs, goes to the logic '1' state. What reaches the PRESET inputs of FF3, FF2, FF1 and FF0 is P_3 , P_2 , P_1 and P_0 respectively, and what reaches their CLEAR inputs is P_3 , P_2 , P_1 and P_0 respectively. Since PRESET and CLEAR are active LOW inputs, the counter flip-flops FF3, FF2, FF1 and FF0 will respectively be loaded with P_3 , P_2 , P_1 and P_0 . For example, if $P_3 = 1$, the PRESET and CLEAR inputs of FF3 will be in the '0' and '1' logic states respectively. This implies that the Q_3 output will go to the logic '1' state. Thus, FF3 has been loaded with P_3 . Similarly, if $P_3 = 0$, the PRESET and CLEAR inputs of flip-flop FF3 will be in the '1' and '0' states respectively. The flip-flop output (Q_3 output) will

be cleared to the '0' state. Again, the flip-flop is loaded with P3 logic status when the input becomes active.

Counter ICs 74190, 74191, 74192 and 74193 are asynchronously presettable synchronous UP/DOWN counters. Many synchronous counters use synchronous presetting whereby the counter is preset or loaded with the data on the active transition of the same clock signal that is used for counting. Presettable counters also have terminal count () outputs, which allow them to be cascaded together to get counters with higher MOD numbers. In the cascade arrangement, the terminal count output of the lower-order counter feeds the clock input of the next higher-order counter.



8.3 SHIFT REGISTERS:

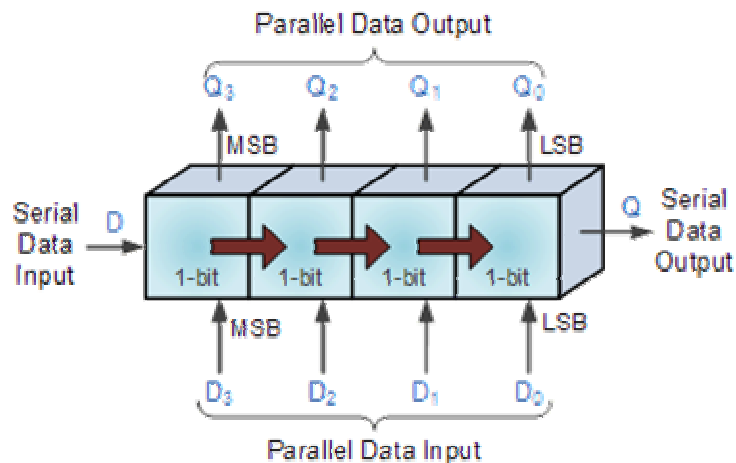
The **Shift Register** is another type of sequential logic circuit that is used for the storage or transfer of data in the form of binary numbers and then "shifts" the data out once every clock cycle, hence the name "shift register". It basically consists of several single bit "D-Type Data Latches", one for each bit (0 or 1) connected together in a serial or daisy-chain arrangement so that the output from one data latch becomes the input of the next latch and so on. The data bits may be fed in or out of the register serially, i.e. one after the other from either the left or the right direction, or in parallel, i.e. all together. The number of individual data latches required to make up a single **Shift Register** is determined by the number of bits to be stored with the most common being 8-bits wide, i.e. eight individual data latches.

The Shift Register is used for data storage or data movement and are used in calculators or computers to store data such as two binary numbers before they are added together, or to convert the data from either a serial to parallel or parallel to serial format. The individual data latches that make up a single shift register are all driven by a common clock (**Clk**) signal making them synchronous devices. Shift register IC's are generally provided with a *clear* or *reset* connection so that they can be "SET" or "RESET" as required.

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

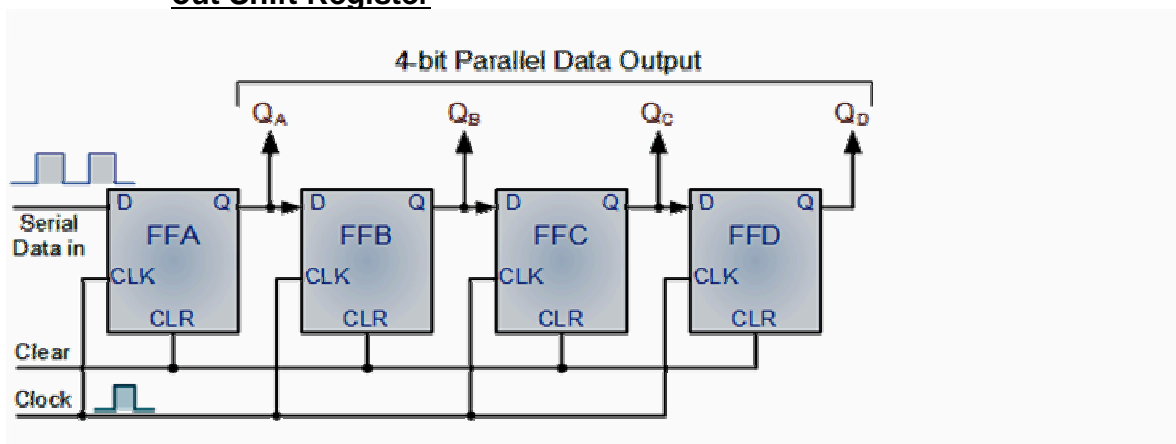
- Serial-in to Parallel-out (SIPO) - the register is loaded with serial data, one bit at a time, with the stored data being available in parallel form.
- Serial-in to Serial-out (SISO) - the data is shifted serially "IN" and "OUT" of the register, one bit at a time in either a left or right direction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

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



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

8.3.1: Serial-in to Parallel-out (SIPO) 4-bit Serial-in to Parallel-out Shift Register



The operation is as follows. Lets assume that all the flip-flops (FFA to FFD) have just been RESET (CLEAR input) and that all the outputs Q_A to Q_D are at logic level "0" i.e, no parallel data output. If a logic "1" is connected to the DATA input pin of FFA then on the first clock pulse the output of FFA and therefore the resulting Q_A will be set HIGH to logic "1" with all the other outputs still remaining LOW at logic "0". Assume now that the DATA input pin of FFA has returned LOW again to logic "0" giving us one data pulse or 0-1-0.

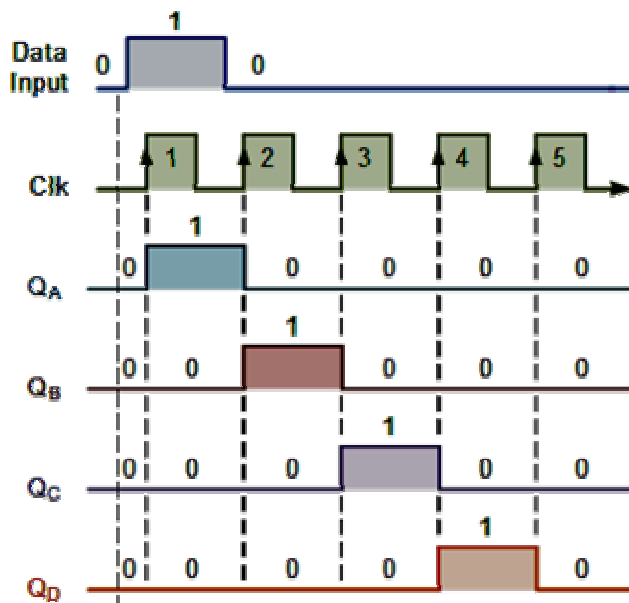
The second clock pulse will change the output of FFA to logic "0" and the output of FFB and Q_B HIGH to logic "1" as its input D has the logic "1" level on it from Q_A . The logic "1" has now moved or been "shifted" one place along the register to the right as

it is now at Q_A . When the third clock pulse arrives this logic "1" value moves to the output of FFC (Q_C) and so on until the arrival of the fifth clock pulse which sets all the outputs Q_A to Q_D back again to logic level "0" because the input to FFA has remained constant at logic level "0".

The effect of each clock pulse is to shift the data contents of each stage one place to the right, and this is shown in the following table until the complete data value of 0-0-0-1 is stored in the register. This data value can now be read directly from the outputs of Q_A to Q_D . Then the data has been converted from a serial data input signal to a parallel data output. The truth table and following waveforms show the propagation of the logic "1" through the register from left to right as follows.

Basic Movement of Data through a Shift Register

Clock Pulse No	Q_A	Q_B	Q_C	Q_D
0	0	0	0	0
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1
5	0	0	0	0



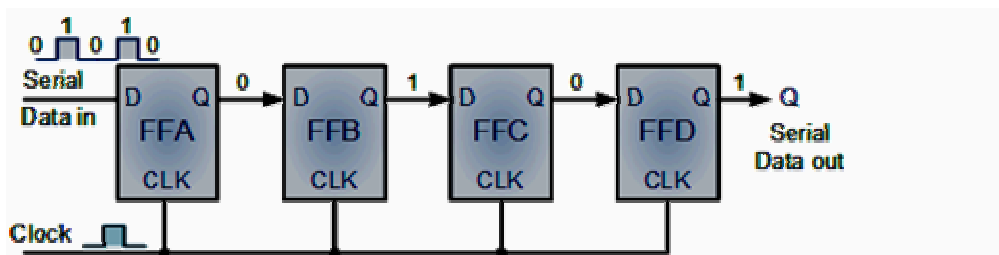
Note that after the fourth clock pulse has ended the 4-bits of data (0-0-0-1) are stored in the register and will remain there provided clocking of the register has stopped. In practice the input data to the register may consist of various combinations of logic "1" and "0". Commonly available SIPO IC's include the standard 8-bit 74LS164 or the 74LS594.

Serial-in to Serial-out (SISO)

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

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

8.3.2: 4-bit Serial-in to Serial-out Shift Register



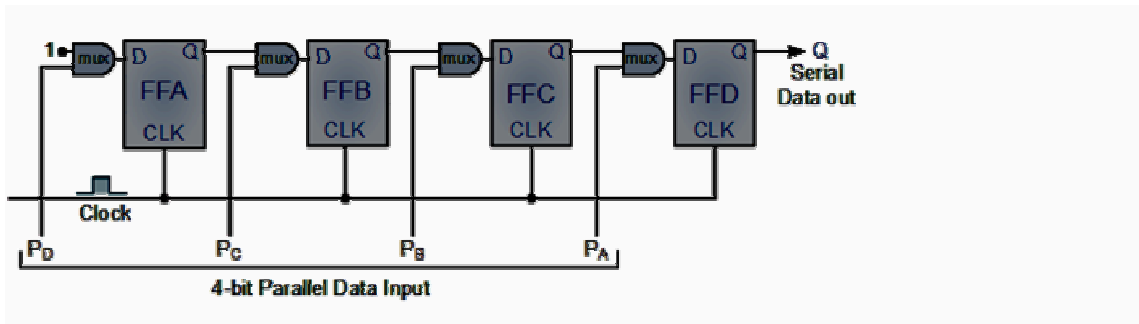
You may think what's the point of a SISO shift register if the output data is exactly the same as the input data. Well this type of **Shift Register** also acts as a temporary storage device or as a time delay device for the data, with the amount of time delay being controlled by the number of stages in the register, 4, 8, 16 etc or by varying the application of the clock pulses. Commonly available IC's include the 74HC595 8-bit Serial-in/Serial-out Shift Register all with 3-state outputs.

Parallel-in to Serial-out (PISO)

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

already present, but four clock pulses are required to unload the data.

8.3.3: 4-bit Parallel-in to Serial-out Shift Register

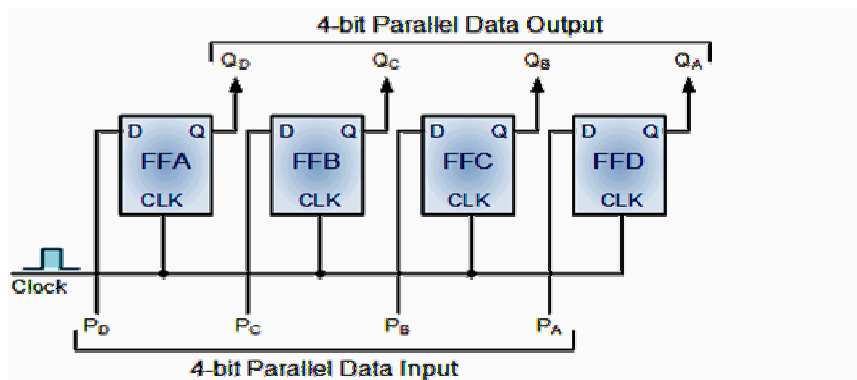


As this type of shift register converts parallel data, such as an 8-bit data word into serial format, it can be used to multiplex many different input lines into a single serial DATA stream which can be sent directly to a computer or transmitted over a communications line. Commonly available IC's include the 74HC166 8-bit Parallel-in/Serial-out Shift Registers.

Parallel-in to Parallel-out (PIPO)

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

8.3.4: 4-bit Parallel-in to Parallel-out Shift Register



The PIPO shift register is the simplest of the four configurations as it has only three connections, the parallel input

(PI) which determines what enters the flip-flop, the parallel output (PO) and the sequencing clock signal (Clk).

Similar to the Serial-in to Serial-out shift register, this type of register also acts as a temporary storage device or as a time delay device, with the amount of time delay being varied by the frequency of the clock pulses. Also, in this type of register there are no interconnections between the individual flip-flops since no serial shifting of the data is required.

8.3.5: Summary of Shift Registers

- Then to summarise.
- A simple **Shift Register** can be made using only D-type flip-Flops, one flip-Flop for each data bit.
- The output from each flip-Flop is connected to the D input of the flip-flop at its right.
- Shift registers hold the data in their memory which is moved or "shifted" to their required positions on each clock pulse.
- Each clock pulse shifts the contents of the register one bit position to either the left or the right.
- The data bits can be loaded one bit at a time in a series input (SI) configuration or be loaded simultaneously in a parallel configuration (PI).
- Data may be removed from the register one bit at a time for a series output (SO) or removed all at the same time from a parallel output (PO).
- One application of shift registers is converting between serial and parallel data.
- Shift registers are identified as SIPO, SISO, PISO, PIPO, and universal shift registers.

8.4 QUESTIONS:

1. Explain Ripple (Asynchronous) counter in short with help of suitable diagram.
2. Explain Binary Ripple counter in brief.
3. Explain construction & working of Synchronous Counter.
4. Write short note on Presettable Counters.
5. What is Shift Register? Explain 4 – bit Serial – in Parallel-out Shift register with help of suitable diagram.
6. Explain Serial-in to Serial-out Shift register in detail.
7. Write detail note on PISO Shift register.

8.5 FURTHER READING:

- ❖ **Digital Electronics** - An Introduction to Theory and Practice by W H Gothmann
- ❖ **Computer Architecture and Parallel Processing** by Kai Hwang, Faye A Briggs , McGraw Hill
- ❖ **Computer Architecture and Organization** by William Stallings
- ❖ **Fundamentals of Computer organization and Design** by Sivarama P. Dandamudi
- ❖ <http://en.wikipedia.org/wiki/Counters>
- ❖ http://en.wikipedia.org/wiki/Shift_registers



COMPUTER ORGANISATION

Unit Structure

9.0 Objectives

9.1 Computers

9.1.1: Functional Units

9.1.2: Control Unit (CU)

9.1.3: Memory System in a Computer

9.1.4: Secondary Storage

9.1.5: Input Output Devices

9.2 Questions

9.3 Further Reading

9.0 OBJECTIVES:

After completing this chapter, you will be able to:

- ❖ Learn the basics about computers.
- ❖ Understand the structure & working of computer.
- ❖ Understand of different parts of computers and their work.
- ❖ Learn about the memory organization within computers.

9.1 COMPUTERS:

A computer as shown in Fig. performs basically five major operations or functions irrespective of their size and make. These are 1) it accepts data or instructions by way of input, 2) it stores data, 3) it can process data as required by the user, 4) it gives results in the form of output, and 5) it controls all operations inside a computer. We discuss below each of these operations.

1. Input: This is the process of entering data and programs in to the computer system. You should know that computer is an electronic machine like any other machine which takes as inputs raw data and performs some processing giving out processed data. Therefore, the input unit takes data from us to the computer in an organized manner for processing.

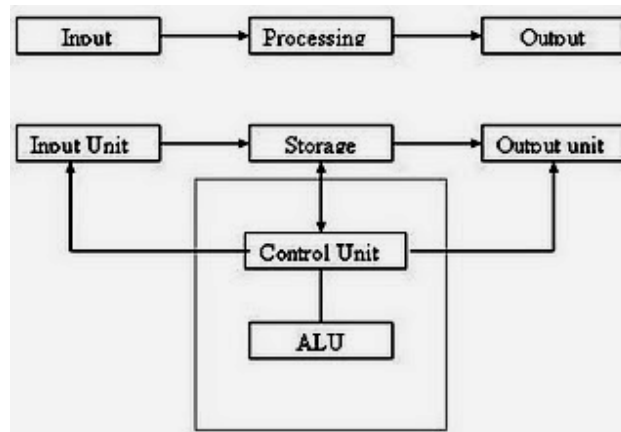


Fig. Basic computer Operations

2. Storage: The process of saving data and instructions permanently is known as storage. Data has to be fed into the system before the actual processing starts. It is because the processing speed of Central Processing Unit (CPU) is so fast that the data has to be provided to CPU with the same speed. Therefore the data is first stored in the storage unit for faster access and processing. This storage unit or the primary storage of the computer system is designed to do the above functionality. It provides space for storing data and instructions.

The storage unit performs the following major functions:

- All data and instructions are stored here before and after processing.
- Intermediate results of processing are also stored here.

3. Processing: The task of performing operations like arithmetic and logical operations is called processing. The Central Processing Unit (CPU) takes data and instructions from the storage unit and makes all sorts of calculations based on the instructions given and the type of data provided. It is then sent back to the storage unit.

4. Output: This is the process of producing results from the data for getting useful information. Similarly the output produced by the computer after processing must also be kept somewhere inside the computer before being given to you in human readable form. Again the output is also stored inside the computer for further processing.

5. Control: The manner how instructions are executed and the above operations are performed. Controlling of all operations like input, processing and output are performed by control unit. It takes care of step by step processing of all operations inside the computer.

9.1.1: FUNCTIONAL UNITS

In order to carry out the operations mentioned in the previous section the computer allocates the task between its various functional units. The computer system is divided into three separate units for its operation. They are 1) arithmetic logical unit, 2) control unit, and 3) central processing unit.

Arithmetic Logical Unit (ALU)

After you enter data through the input device it is stored in the primary storage unit. The actual processing of the data and instruction are performed by Arithmetic Logical Unit. The major operations performed by the ALU are addition, subtraction, multiplication, division, logic and comparison. Data is transferred to ALU from storage unit when required. After processing the output is returned back to storage unit for further processing or getting stored.

9.1.2:Control Unit (CU)

The next component of computer is the Control Unit, which acts like the supervisor seeing that things are done in proper fashion. The control unit determines the sequence in which computer programs and instructions are executed. Things like processing of programs stored in the main memory, interpretation of the instructions and issuing of signals for other units of the computer to execute them. It also acts as a switch board operator when several users access the computer simultaneously. Thereby it coordinates the activities of computer's peripheral equipment as they perform the input and output. Therefore it is the manager of all operations mentioned in the previous section.

Central Processing Unit (CPU)

The ALU and the CU of a computer system are jointly known as the central processing unit. You may call CPU as the brain of any computer system. It is just like brain that takes all major decisions, makes all sorts of calculations and directs different parts of the computer functions by activating and controlling the operations.

9.1.3: MEMORY SYSTEM IN A COMPUTER

There are two kinds of computer memory: *primary* and *secondary*. Primary memory is accessible directly by the processing unit. RAM is an example of primary memory. As soon as the computer is switched off the contents of the primary memory is lost. You can store and retrieve data much faster with primary

memory compared to secondary memory. Secondary memory such as floppy disks, magnetic disk, etc., is located outside the computer. Primary memory is more expensive than secondary memory. Because of this the size of primary memory is less than that of secondary memory. We will discuss about secondary memory later on.

Computer memory is used to store two things: i) instructions to execute a program and ii) data. When the computer is doing any job, the data that have to be processed are stored in the primary memory. This data may come from an input device like keyboard or from a secondary storage device like a floppy disk.

As program or the set of instructions is kept in primary memory, the computer is able to follow instantly the set of instructions. For example, when you book ticket from railway reservation counter, the computer has to follow the same steps: take the request, check the availability of seats, calculate fare, wait for money to be paid, store the reservation and get the ticket printed out. The programme containing these steps is kept in memory of the computer and is followed for each request.

But inside the computer, the steps followed are quite different from what we see on the monitor or screen. In computer's memory both programs and data are stored in the binary form. You have already been introduced with decimal number system, that is the numbers 1 to 9 and 0. The binary system has only two values 0 and 1. These are called *bits*. As human beings we all understand decimal system but the computer can only understand binary system. It is because a large number of integrated circuits inside the computer can be considered as switches, which can be made ON, or OFF. If a switch is ON it is considered 1 and if it is OFF it is 0. A number of switches in different states will give you a message like this: 110101....10. So the computer takes input in the form of 0 and 1 and gives output in the form 0 and 1 only. Is it not absurd if the computer gives outputs as 0's & 1's only? But you do not have to worry about. Every number in binary system can be converted to decimal system and vice versa; for example, 1010 meaning decimal 10. Therefore it is the computer that takes information or data in decimal form from you, convert it in to binary form, process it producing output in binary form and again convert the output to decimal form.

The primary memory as you know in the computer is in the form of IC's (Integrated Circuits). These circuits are called Random Access Memory (RAM). Each of RAM's locations stores one *byte* of information. (One *byte* is equal to 8 *bits*). A bit is an acronym for *binary digit*, which stands for one binary piece of information. This can be either 0 or 1. You will know more about RAM later. The

Primary or internal storage section is made up of several small storage locations (ICs) called cells. Each of these cells can store a fixed number of bits called *word length*.

Each cell has a unique number assigned to it called the address of the cell and it is used to identify the cells. The address starts at 0 and goes up to (N-1). You should know that the memory is like a large cabinet containing as many drawers as there are addresses on memory. Each drawer contains a word and the address is written on outside of the drawer.

Capacity of Primary Memory

You know that each cell of memory contains one character or 1 byte of data. So the capacity is defined in terms of byte or words. Thus 64 kilobyte (KB) memory is capable of storing $64 \times 1024 = 32,768$ bytes. (1 kilobyte is 1024 bytes). A memory size ranges from few kilobytes in small systems to several thousand kilobytes in large mainframe and super computer. In your personal computer you will find memory capacity in the range of 64 KB, 4 MB, 8 MB and even 16 MB (MB = Million bytes).

The following terms related to memory of a computer are discussed below:

1. **Random Access Memory (RAM):** The primary storage is referred to as random access memory (RAM) because it is possible to randomly select and use any location of the memory directly store and retrieve data. It takes same time to any address of the memory as the first address. It is also called read/write memory. The storage of data and instructions inside the primary storage is temporary. It disappears from RAM as soon as the power to the computer is switched off. The memories, which loose their content on failure of power supply, are known as **volatile** memories .So now we can say that RAM is volatile memory.
2. **Read Only Memory (ROM):** There is another memory in computer, which is called Read Only Memory (ROM). Again it is the ICs inside the PC that form the ROM. The storage of program and data in the ROM is permanent. The ROM stores some standard processing programs supplied by the manufacturers to operate the personal computer. The ROM can only be read by the CPU but it cannot be changed. The basic input/output program is stored in the ROM that examines and initializes various equipment attached to the PC when the switch is made ON. The memories, which do not loose their content on failure of power supply, are known as **non-volatile** memories. ROM is non-volatile memory.

3. **PROM** There is another type of primary memory in computer, which is called Programmable Read Only Memory (PROM). You know that it is not possible to modify or erase programs stored in ROM, but it is possible for you to store your program in PROM chip. Once the programmes are written it cannot be changed and remain intact even if power is switched off. Therefore programs or instructions written in PROM or ROM cannot be erased or changed.
4. **EPROM:** This stands for Erasable Programmable Read Only Memory, which over come the problem of PROM & ROM. EPROM chip can be programmed time and again by erasing the information stored earlier in it. Information stored in EPROM exposing the chip for some time ultraviolet light and it erases chip is reprogrammed using a special programming facility. When the EPROM is in use information can only be read.
5. **Cache Memory:** The speed of CPU is extremely high compared to the access time of main memory. Therefore the performance of CPU decreases due to the slow speed of main memory. To decrease the mismatch in operating speed, a small memory chip is attached between CPU and Main memory whose access time is very close to the processing speed of CPU. It is called CACHE memory. CACHE memories are accessed much faster than conventional RAM. It is used to store programs or data currently being executed or temporary data frequently used by the CPU. So each memory makes main memory to be faster and larger than it really is. It is also very expensive to have bigger size of cache memory and its size is normally kept small.
6. **Registers:** The CPU processes data and instructions with high speed, there is also movement of data between various units of computer. It is necessary to transfer the processed data with high speed. So the computer uses a number of special memory units called *registers*. They are not part of the main memory but they store data or information temporarily and pass it on as directed by the control unit.

9.1.4:SECONDARY STORAGE

1. You are now clear that the operating speed of primary memory or main memory should be as fast as possible to cope up with the CPU speed. These high-speed storage devices are very expensive and hence the cost per bit of storage is also very high. Again the storage capacity of the main memory is also very limited. Often it is necessary to store hundreds of millions of bytes of data for the CPU to process. Therefore additional memory is required in all the computer systems. This memory is called *auxiliary memory* or *secondary storage*.

2. In this type of memory the cost per bit of storage is low. However, the operating speed is slower than that of the primary storage. Huge volume of data are stored here on permanent basis and transferred to the primary storage as and when required. Most widely used secondary storage devices are *magnetic tapes* and *magnetic disk*.
3. **Magnetic Tape:** Magnetic tapes are used for large computers like mainframe computers where large volume of data is stored for a longer time. In PC also you can use tapes in the form of cassettes. The cost of storing data in tapes is inexpensive. Tapes consist of magnetic materials that store data permanently. It can be 12.5 mm to 25 mm wide plastic film-type and 500 meter to 1200 meter long which is coated with magnetic material. The deck is connected to the central processor and information is fed into or read from the tape through the processor. It similar to cassette tape recorder.

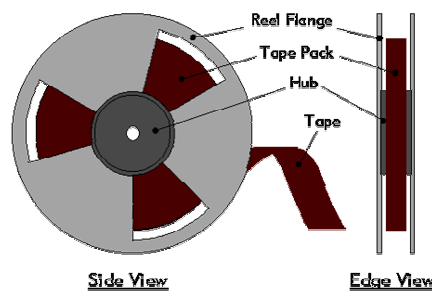


Fig: Magnetic Tape

1. **Advantages of Magnetic Tape:**
 - **Compact:** A 10-inch diameter reel of tape is 2400 feet long and is able to hold 800, 1600 or 6250 characters in each inch of its length. The maximum capacity of such tape is 180 million characters. Thus data are stored much more compactly on tape.
 - **Economical:** The cost of storing characters is very less as compared to other storage devices.
 - **Fast:** Copying of data is easier and fast.
 - **Long term Storage and Re-usability:** Magnetic tapes can be used for long term storage and a tape can be used repeatedly with out loss of data.
2. **Magnetic Disk:** You might have seen the gramophone record, which is circular like a disk and coated with magnetic material. Magnetic disks used in computer are made on the same principle. It rotates with very high speed inside the computer

drive. Data is stored on both the surface of the disk. Magnetic disks are most popular for *direct access* storage device. Each disk consists of a number of invisible *concentric circles* called *tracks*. Information is recorded on tracks of a disk surface in the form of tiny magnetic spots. The presence of a magnetic spot represents *one bit* and its absence represents zero bit. The information stored in a disk can be read many times without affecting the stored data. So the reading operation is non-destructive. But if you want to write a new data, then the existing data is erased from the disk and new data is recorded.

3. **Floppy Disk:** It is similar to magnetic disk discussed above. They are 5.25 inch or 3.5 inch in diameter. They come in single or double density and recorded on one or both surface of the diskette. The capacity of a 5.25-inch floppy is 1.2 mega bytes whereas for 3.5 inch floppy it is 1.44 mega bytes. It is cheaper than any other storage devices and is portable. The floppy is a low cost device particularly suitable for personal computer system. A floppy disk drive reads and writes data to a small, circular piece of metal-coated plastic similar to audio cassette tape.

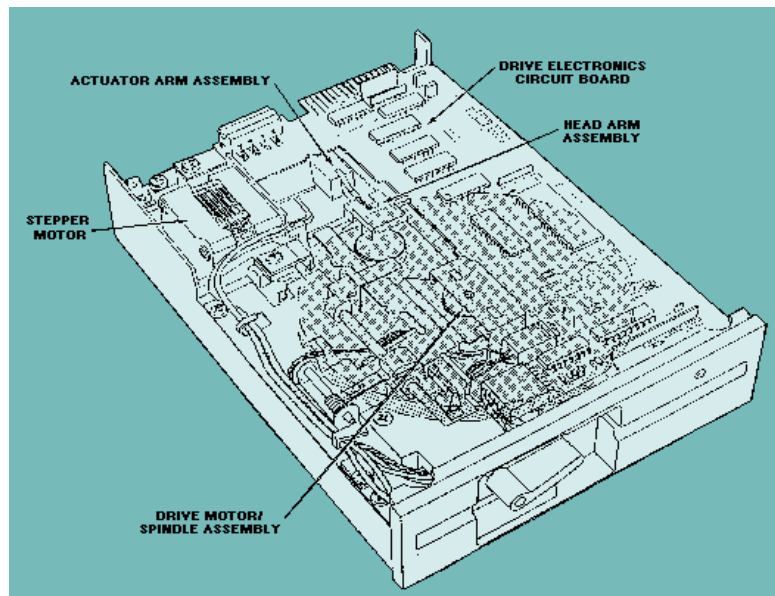


Fig: Floppy Disk

2. Optical Disk:

With every new application and software there is greater demand for memory capacity. It is the necessity to store large volume of data that has led to the development of optical disk storage medium. Optical disks can be divided into the following categories:

1. *Compact Disk/ Read Only Memory (CD-ROM)*: CD-ROM disks are made of reflective metals. CD-ROM is written during the process of manufacturing by high power *laser beam*. Here the storage density is very high, storage cost is very low and access time is relatively fast. Each disk is approximately 4 1/2 inches in diameter and can hold over 600 MB of data. As the CD-ROM can be *read only* we cannot write or make changes into the data contained in it.
2. *Write Once, Read Many (WORM)*: The inconvenience that we can not write any thing in to a CD-ROM is avoided in WORM. A WORM allows the user to write data permanently on to the disk. Once the data is written it can never be erased without physically damaging the disk. Here data can be recorded from keyboard, video scanner, OCR equipment and other devices. The advantage of WORM is that it can store vast amount of data amounting to gigabytes (10^9 bytes). Any document in a WORM can be accessed very fast, say less than 30 seconds.
3. *Erasable Optical Disk*: These are optical disks where data can be written, erased and re-written. This also applies a laser beam to write and re-write the data. These disks may be used as alternatives to traditional disks. Erasable optical disks are based on a technology known as *magnetic optical (MO)*. To write a data bit on to the erasable optical disk the MO drive's laser beam heats a tiny, precisely defined point on the disk's surface and magnetises it.

9.1.5: INPUT OUTPUT DEVICES

A computer is only useful when it is able to communicate with the external environment. When you work with the computer you feed your data and instructions through some devices to the computer. These devices are called Input devices. Similarly computer after processing, gives output through other devices called output devices.

For a particular application one form of device is more desirable compared to others. We will discuss various types of I/O devices that are used for different types of applications. They are also known as peripheral devices because they surround the CPU and make a communication between computer and the outer world.

1 Input Devices

Input devices are necessary to convert our information or data in to a form which can be understood by the computer. A good input device should provide timely, accurate and useful data to the main memory of the computer for processing followings are the most useful input devices.

1. **Keyboard:** - This is the standard input device attached to all computers. The layout of keyboard is just like the traditional typewriter of the type QWERTY. It also contains some extra command keys and function keys. It contains a total of 101 to 104 keys. A typical keyboard used in a computer is shown in Fig.. You have to press correct combination of keys to input data. The computer can recognise the electrical signals corresponding to the correct key combination and processing is done accordingly.



Fig: Keyboard

2. **Mouse:** - Mouse is an input device shown in Fig. 2.7 that is used with your personal computer. It rolls on a small ball and has two or three buttons on the top. When you roll the mouse across a flat surface the screen sensors the mouse in the direction of mouse movement. The cursor moves very fast with mouse giving you more freedom to work in any direction. It is easier and faster to move through a mouse.



Fig: Mouse

3. **Scanner:** The keyboard can input only text through keys provided in it. If we want to input a picture the keyboard cannot do that. Scanner is an optical device that can input any graphical matter and display it back. The common optical scanner devices are Magnetic Ink Character Recognition (MICR), Optical Mark Reader (OMR) and Optical Character Reader (OCR).

- **Magnetic Ink Character Recognition (MICR):** - This is widely used by banks to process large volumes of cheques and drafts. Cheques are put inside the MICR. As they enter the reading unit the cheques pass through the magnetic field which causes the read head to recognise the character of the cheques.
- **Optical Mark Reader (OMR):** This technique is used when students have appeared in objective type tests and they had to mark their answer by darkening a square or circular space by pencil. These answer sheets are directly fed to a computer for grading where OMR is used.
- **Optical Character Recognition (OCR):** - This technique unites the direct reading of any printed character. Suppose you have a set of hand written characters on a piece of paper. You put it inside the scanner of the computer. This pattern is compared with a site of patterns stored inside the computer. Whichever pattern is matched is called a character read. Patterns that cannot be identified are rejected. OCRs are expensive though better the MICR.

Output Devices

1. **Visual Display Unit:** The most popular input/output device is the Visual Display Unit (VDU). It is also called the monitor. A Keyboard is used to input data and Monitor is used to display the input data and to receive messages from the computer. A monitor has its own box which is separated from the main computer system and is connected to the computer by cable. In some systems it is compact with the system unit. It can be *color* or *monochrome*.
2. **Terminals:** It is a very popular interactive input-output unit. It can be divided into two types: hard copy terminals and *soft copy* terminals. A *hard copy* terminal provides a printout on paper whereas soft copy terminals provide visual copy on monitor. A terminal when connected to a CPU sends instructions directly to the computer. Terminals are also classified as dumb terminals or intelligent terminals depending upon the work situation.
3. **Printer:** It is an important output device which can be used to get a printed copy of the processed text or result on paper. There are different types of printers that are designed for different types of applications. Depending on their speed and approach of printing, printers are classified as *impact* and *non-impact* printers. Impact printers use the familiar typewriter approach of hammering a typeface against the paper and inked ribbon. *Dot-matrix printers* are of this type. Non-impact printers

do not hit or impact a ribbon to print. They use electro-static chemicals and ink-jet technologies. *Laser printers* and *Ink-jet printers* are of this type. This type of printers can produce color printing and elaborate graphics.

9.2 QUESTIONS:

1. Explain the components of computer organizations?
1. Define Computer? Explain basic Structure & Working of Computers.
2. Write short note on Functional units of computer.
3. What is difference between RAM and ROM?
4. Explain in Brief about types of memory in computers.
5. What secondary storage in computer?
6. Explain types of Secondary storage in computers.
7. Explain structure & working of following:
 - a) Optical Disk
 - b) Magnetic Tape
8. Write short note on input devices.
9. Explain in brief about output devices of computers.

9.3 FURTHER READING:

- ❖ **Computer Organization and Architecture** by William Stallings
- ❖ **Structured Computer Organization** by Tanenbaum



OPERATING SYSTEMS

Unit Structure

- 10.0 Objectives
- 10.1 Introduction
 - 10.1.1: Types of OS
- 10.2 Windows Operating System
- 10.3 Linux Operating System
- 10.4 Some Linux Commands
- 10.5 Questions
- 10.6 Further Reading

10.0: OBJECTIVES

After completing this chapter, you will be able to:

- ❖ Understand the Operating System and its structure.
- ❖ Different types of Operating Systems, their applications, use & history.
- ❖ Interact with LINUX OS using several commands

10.1 INTRODUCTION:

An **operating system (OS)** is a set of programs that manage computer hardware resources and provide common services for application software. The operating system is the most important type of system software in a computer system. A user cannot run an application program on the computer without an operating system, unless the application program is self booting. Time-sharing operating systems schedule tasks for efficient use of the system and may also include accounting for cost allocation of processor time, mass storage, printing, and other resources.

For hardware functions such as input and output and memory allocation, the operating system acts as an intermediary between application programs and the computer hardware, although the application code is usually executed directly by the hardware and will frequently call the OS or be interrupted by

it. Operating systems are found on almost any device that contains a computer — from cellular phones and video game consoles to supercomputers and web servers.

Examples of popular modern operating systems include Android, iOS, Linux, Mac OS X, all of which have their roots in Unix, and Microsoft Windows.

10.1.1: Types of OS:

Real-time

A real-time operating system is a multitasking operating system that aims at executing real-time applications. Real-time operating systems often use specialized scheduling algorithms so that they can achieve a deterministic nature of behavior. The main objective of real-time operating systems is their quick and predictable response to events. They have an event-driven or time-sharing design and often aspects of both. An event-driven system switches between tasks based on their priorities or external events while time-sharing operating systems switch tasks based on clock interrupts.

Multi-user vs. Single-user

A multi-user operating system allows multiple users to access a computer system concurrently. Time-sharing system can be classified as multi-user systems as they enable a multiple user access to a computer through the sharing of time. Single-user operating systems, as opposed to a multi-user operating system, are usable by a single user at a time. Being able to have multiple accounts on a Windows operating system does not make it a multi-user system. Rather, only the network administrator is the real user. But for a Unix-like operating system, it is possible for two users to login at a time and this capability of the OS makes it a multi-user operating system.

Multi-tasking vs. Single-tasking

When only a single program is allowed to run at a time, the system is grouped under a single-tasking system. However, when the operating system allows the execution of multiple tasks at one time, it is classified as a multi-tasking operating system. Multi-tasking can be of two types: pre-emptive or co-operative. In pre-emptive multitasking, the operating system slices the CPU time and dedicates one slot to each of the programs. Unix-like operating systems such as Solaris and Linux support pre-emptive multitasking, as does AmigaOS. Cooperative multitasking is achieved by relying on each process to give time to the other processes in a defined manner. MS Windows prior to Windows 2000 and Mac OS prior to OS X used to support cooperative multitasking.

Distributed

A distributed operating system manages a group of independent computers and makes them appear to be a single computer. The development of networked computers that could be linked and communicate with each other gave rise to distributed computing. Distributed computations are carried out on more than one machine. When computers in a group work in cooperation, they make a distributed system.

Embedded

Embedded operating systems are designed to be used in embedded computer systems. They are designed to operate on small machines like PDAs with less autonomy. They are able to operate with a limited number of resources. They are very compact and extremely efficient by design. Windows CE and Minix 3 are some examples of embedded operating systems.

10.2 WINDOWS OPERATING SYSTEM:

Introduction to Windows 7:

Windows 7 is an **operating system**, developed by the global giant Microsoft, which was released for public in October 2009. An **operating system** can be understood as a software program designed to facilitate the communication between computer hardware and software. Without an operating system, a computer is useless. **Windows 7** is more simpler and easier to use compared to its predecessor, Windows Vista. Windows Vista cannot be considered as a very successful launch of Microsoft. Windows 7 has a 64-bit along with the availability of 32-bit support which enables the users to use almost all the latest PCs. Be it desktops, laptops, notebooks, or anything,

Windows 7 supports them all.

The purpose behind the launching of Windows 7:

Windows 7 was launched with many new and advanced features beneficial to the users. The main aim was to cope up with the limitations present in the previous versions that were highly criticised. Windows Vista consisted of a big range of user friendly features but unfortunately, the system failed due to ever increasing complaints and negative reviews coming from the press. Contrary to this, Windows 7 was developed with a focus on rectifying the mistakes and providing noticeable upgrade to the product line of Windows.

This product of Microsoft has launched **six different editions**. They are listed as under:

- Starter
- Home Premium
- Professional
- Ultimate
- OEM
- Enterprise

System requirements for installing Windows 7:

The installation of Windows 7 operating system needs certain bare minimum requirements.

- RAM (Random Access Memory) ranging from 1GB to 2GB is enough here.
- Free Hard Disk space ranging from 16GB to 20GB is needed.
- DirectX 9 graphics device with WDDM 1.0 or higher driver is sufficient in both cases.

Various available options for the installation of Windows 7:

Microsoft has taken all the necessary steps to ensure that this version is developed in a manner that it is accepted by all. Hence, keeping this in mind, its installation process is made extremely simple.

Users wanting to go for this product, have the option of just upgrading directly from Windows Vista or even XP. Secondly they can opt for a clean install option available to them. And for the fresh users, they can directly buy new computers having this operating system already installed.

Most Striking features of Windows 7:

The most attractive feature of Windows 7 can be its stability and reliability. It's advanced graphical features, Aero Peek, and new taskbar, which is covered by **Windows Shell**. This taskbar has been given the name '**Superbar**' by its developers. This new Taskbar has been praised by the users so far. Although a bit difficult to get a grasp of, these new features have proved to be a hit among the users. The best improvement by Microsoft here is the replacement of Quick Launch Toolbar with pinning applications. This innovation has resulted in more easy to see icons. They are an absolute delight to use.

Another interesting feature added here is that of **Aero Snap**. This feature enables the user to maximize the window as soon as the edge of the screen is dragged. Just a little move by the user, and the window restores its previous state.

Users having the leisure of touch screen monitors, can also enjoy another interesting feature of **native touch**.

The developers have taken full care to see that performance of Windows 7 is much better compared to Vista and this can be proved with the introduction of innumerable new features including the following: support for virtual hard disks, Handwriting recognition, improved presentation on multi-core processors, improved boot performance, DirectAccess and kernel improvements, etc.

Windows components like **Internet Explorer** and **Windows Media Player 12** are included in this product of Microsoft. The new look that the developers have given to Windows Media Player here is incomparable. It is more stylish and sleek than ever, making it more enjoyable.

For Game lovers also, Windows 7 has proved to be better than its previous edition, Windows Vista. Many popular games like Internet Spades, Internet Backgammon, etc. which had disappeared from Vista have been restored in Windows 7.

By the launch of **Windows 7**, Microsoft has successfully tried to regain its lost popularity to a considerable extent.

10.3 LINUX OPERATING SYSTEM:

Linux is, in simplest terms, an operating system. It is the software on a computer that enables applications and the computer operator to access the devices on the computer to perform desired functions. The operating system (OS) relays instructions from an application to, for instance, the computer's processor. The processor performs the instructed task, then sends the results back to the application via the operating system.

Explained in these terms, Linux is very similar to other operating systems, such as Windows and OS X.

But something sets Linux apart from these operating systems. The Linux operating system represented a \$25 billion ecosystem in 2008. Since its inception in 1991, Linux has grown to become a force in computing, powering everything from the New York Stock Exchange to mobile phones to supercomputers to consumer devices.

As an open operating system, Linux is developed collaboratively, meaning no one company is solely responsible for its development or ongoing support. Companies participating in the

Linux economy share research and development costs with their partners and competitors. This spreading of development burden amongst individuals and companies has resulted in a large and efficient ecosystem and unheralded software innovation.

Over 1,000 developers, from at least 100 different companies, contribute to every kernel release. In the past two years alone, over 3,200 developers from 200 companies have contributed to the kernel--which is just one small piece of a Linux distribution.

This article will explore the various components of the Linux operating system, how they are created and work together, the communities of Linux, and Linux's incredible impact on the IT ecosystem.

Where is Linux?

One of the most noted properties of Linux is where it can be used. Windows and OS X are predominantly found on personal computing devices such as desktop and laptop computers. Other operating systems, such as Symbian, are found on small devices such as phones and PDAs, while mainframes and supercomputers found in major academic and corporate labs use specialized operating systems such as AS/400 and the Cray OS.

Linux, which began its existence as a server OS and Has become useful as a desktop OS, can also be used on all of these devices. „ÚFrom wristwatches to supercomputers,„À is the popular description of Linux' capabilities.

An abbreviated list of some of the popular electronic devices Linux is used on today includes:



Dell Inspiron Mini 9 and 12



Garmin Nuvi 860, 880, and 5000



Google Dev Phone 1 Android



HP Mini 1000



Lenovo IdeaPad S9



**Motorola MotoRokr
EM35 Phone**



**One Laptop Per Child
XO2**



**Sony Bravia
Television**



Sony Reader



**TiVo
Digital Video Recorder**



**Volvo In-Car
Navigation System**



**Yamaha Motif
Keyboard**

These are just the most recent examples of Linux-based devices available to consumers worldwide. This actual number of items that use Linux numbers in the thousands. The Linux Foundation is building a centralized database that will list all currently offered Linux-based products, as well as archive those devices that pioneered Linux-based electronics.

Linux Distributions:

Well-known Linux distributions include:

- Arch Linux, a minimalist distribution maintained by a volunteer community and primarily based on binary packages in the tar.gz and tar.xz format.
- Debian, a non-commercial distribution maintained by a volunteer developer community with a strong commitment to free software principles

- Knoppix, the first Live CD distribution to run completely from removable media without installation to a hard disk, derived from Debian
- Linux Mint Debian Edition (LMDE) is based directly on Debian's *testing* distribution.
- Ubuntu, a popular desktop and server distribution derived from Debian, maintained by Canonical Ltd.
- BackTrack, based off the Ubuntu Operating System. Used for digital forensics and penetration testing.
- Kubuntu, the KDE version of Ubuntu.
- Linux Mint, a distribution based on and compatible with Ubuntu.
- Xubuntu is the Xfce version of Ubuntu.
- Fedora, a community distribution sponsored by Red Hat
- Red Hat Enterprise Linux, which is a derivative of Fedora, maintained and commercially supported by Red Hat.
- CentOS, a distribution derived from the same sources used by Red Hat, maintained by a dedicated volunteer community of developers with both 100% Red Hat-compatible versions and an upgraded version that is not always 100% upstream compatible
- Oracle Enterprise Linux, which is a derivative of Red Hat Enterprise Linux, maintained and commercially supported by Oracle.
- Mandriva, a Red Hat derivative popular in France and Brazil, today maintained by the French company of the same name.
- Manthiran Linux is a popular linux distro introduced to this world by Quara Foundation.
- PCLinuxOS, a derivative of Mandriva, grew from a group of packages into a community-spawned desktop distribution.
- Gentoo, a distribution targeted at power users, known for its FreeBSD Ports-like automated system for compiling applications from source code
- openSUSE a community distribution mainly sponsored by Novell.
- SUSE Linux Enterprise, derived from openSUSE, maintained and commercially supported by Novell.
- Slackware, one of the first Linux distributions, founded in 1993, and since then actively maintained by Patrick J. Volkerding.
- Damn Small Linux, "DSL" is a Biz-card Desktop OS

Advantages of Linux Operating System:

Low cost:

There is no need to spend time and huge amount money to obtain licenses since Linux and much of it's software come with the GNU General Public License. There is no need to worry about any software's that you use in Linux.

Stability:

Linux has high stability compared with other operating systems. There is no need to reboot the Linux system to maintain performance levels. Rarely it freeze up or slow down. It has a continuous up-times of hundreds of days or more.

Performance:

Linux provides high performance on various networks. It has the ability to handle large numbers of users simultaneously.

Networking:

Linux provides a strong support for network functionality; client and server systems can be easily set up on any computer running Linux. It can perform tasks like network backup more faster than other operating systems.

Flexibility:

Linux is very flexible. Linux can be used for high performance server applications, desktop applications, and embedded systems. You can install only the needed components for a particular use. You can also restrict the use of specific computers.

Compatibility:

It runs all common Unix software packages and can process all common file formats.

Wider Choice:

There is a large number of Linux distributions which gives you a wider choice. Each organization develop and support different distribution. You can pick the one you like best; the core function's are the same.

Fast and easy installation:

Linux distributions come with user-friendly installation.

Better use of hard disk:

Linux uses its resources well enough even when the hard disk is almost full.

Multitasking:

Linux is a multitasking operating system. It can handle many things at the same time.

Security:

Linux is one of the most secure operating systems. File ownership and permissions make linux more secure.

Open source:

Linux is an Open source operating systems. You can easily get the source code for linux and edit it to develop your personal operating system.

Today, Linux is widely used for both basic home and office uses. It is the main operating system used for high performance business and in web servers. Linux has made a high impact in this world.

Comparison of Windows and Linux:

Both Linux and Windows are operating systems. An operating system is the most important program that runs on a computer. Every general-purpose computer must have an operating system to run other programs. Operating systems perform basic tasks, such as recognizing input from the keyboard, sending output to the display screen, keeping track of files and directories on the disk, and controlling peripheral devices such as disk drives and printers.

1. Reduces the risk of carpal tunnel syndrome

When linux is properly installed, there no longer a need to use the mouse. Chances of you using a mouse is close to zero.

2. Use the extra cash for rewards

Linux is 100% free while Windows Vista Ultimate costs \$398.99 at the time of writing. Companies that pay a licensing annually could have used the money for other things like buying an additional server to reduce the load or even give a bigger bonus to its loyal employees.

3. Formats are free, freedom is preserved

Linux file formats can be accessed in a variety of ways because they are free. Windows on the other hand makes you lock your own data in secret formats that can only be accessed with tools leased to you at the vendor's price. "What we will get with Microsoft is a three-year lease on a health record we need to keep for 100 years"

4. Zero risk in violating license agreements

Linux is open source so you are unlikely to violate any license agreement. All the software is happily yours. With MS Windows you likely already violate all kinds of licenses and you could be pronounced a computer pirate if only a smart lawyer was after you. The worldwide PC software piracy rate for 2004 is at 35%. Which means that 3 out of 10 people are likely to get into real trouble.

5. Transparent vs Proprietary

MS Windows is based on DOS, Linux is based on UNIX. MS Windows Graphical User Interface (GUI) is based on Microsoft-own marketing-driven specifications. Linux GUI is based on industry-standard network-transparent X-Windows.

6. Better network, processing capabilities

Linux beats Windows hands down on network features, as a development platform, in data processing capabilities, and as a scientific workstation. MS Windows desktop has a more polished appearance, simple general business applications, and many more games for kids (less intellectual games compared to linux's).

7. Customizable

Linux is customizable in a way that Windows is not. For example, NASlite is a version of Linux that runs off a single floppy disk and converts an old computer into a file server. This ultra small edition of Linux is capable of networking, file sharing and being a web server.

8. Flexibility

Windows must boot from a primary partition. Linux can boot from either a primary partition or a logical partition inside an extended partition. Windows must boot from the first hard disk. Linux can boot from any hard disk in the computer.

9. Mobility

Windows allows programs to store user information (files and settings) anywhere. This makes it impossibly hard to backup user data files and settings and to switch to a new computer. In contrast, Linux stores all user data in the home directory making it much easier to migrate from an old computer to a new one. If home directories are segregated in their own partition, you can even upgrade from one version of Linux to another without having to migrate user data and settings.

10. Proven Security

Why isn't Linux affected by viruses? Simply because its code has been open source for more than a decade, tested by people all around the world, and not by a single development team like in the case of Windows. This leads to a lightning fast finding and fixing for exploitable holes in Linux. So that proves Linux as having an extremely enhanced security and lesser chances of exploits compared to Windows.

10.4 SOME LINUX COMMANDS:

mkdir - make directories

Usage

mkdir [OPTION] DIRECTORY

Options

Create the DIRECTORY(ies), if they do not already exist.

Mandatory arguments to long options are mandatory for short options too.

-m, mode=MODE set permission mode (as in chmod), not rwxrwxrwx - umask

-p, parents no error if existing, make parent directories as needed

-v, verbose print a message for each created directory

-help display this help and exit

-version output version information and exit

cd - change directories

Use cd to change directories. Type cd followed by the name of a directory to access that directory. Keep in mind that you are always in a directory and can navigate to directories hierarchically above or below.

mv- change the name of a directory

Type mv followed by the current name of a directory and the new name of the directory.

Ex: mv testdir newnamedir

pwd - print working directory

will show you the full path to the directory you are currently in. This is very handy to use, especially when performing some of the other commands on this page

rmdir - Remove an existing directory**rm -r**

Removes directories and files within the directories recursively.

chown - change file owner and group

Usage

chown [OPTION] OWNER[:[GROUP]] FILE

chown [OPTION] :GROUP FILE

chown [OPTION] --reference=RFILE FILE

Options

Change the owner and/or group of each FILE to OWNER and/or GROUP. With --reference, change the owner and group of each FILE to those of RFILE.

-c, changes like verbose but report only when a change is made

-dereference affect the referent of each symbolic link, rather than the symbolic link itself

-h, no-dereference affect each symbolic link instead of any referenced file (useful only on systems that can change the ownership of a symlink)

-from=CURRENT_OWNER:CURRENT_GROUP

change the owner and/or group of each file only if its current owner and/or group match those specified here. Either may be omitted, in which case a match is not required for the omitted attribute.

-no-preserve-root do not treat '/' specially (the default)

-preserve-root fail to operate recursively on '/'

- f, -silent, -quiet suppress most error messages
- reference=RFILE use RFILE's owner and group rather than the specifying OWNER:GROUP values
- R, -recursive operate on files and directories recursively
- v, -verbose output a diagnostic for every file processed

The following options modify how a hierarchy is traversed when the -R option is also specified. If more than one is specified, only the final one takes effect.

- H if a command line argument is a symbolic link to a directory, traverse it
- L traverse every symbolic link to a directory encountered
- P do not traverse any symbolic links (default)

chmod - change file access permissions

Usage

chmod [-r] permissions filenames

r Change the permission on files that are in the subdirectories of the directory that you are currently in permission Specifies the rights that are being granted. Below is the different rights that you can grant in an alpha numeric format. filenames File or directory that you are associating the rights with Permissions

- u - User who owns the file.
- g - Group that owns the file.
- o - Other.
- a - All.
- r - Read the file.
- w - Write or edit the file.
- x - Execute or run the file as a program.

Numeric Permissions:

CHMOD can also to attributed by using Numeric Permissions:

- 400 read by owner
- 040 read by group
- 004 read by anybody (other)
- 200 write by owner
- 020 write by group
- 002 write by anybody

100 execute by owner
 010 execute by group
 001 execute by anybody

ls - Short listing of directory contents

-a list hidden files
 -d list the name of the current directory
 -F show directories with a trailing '/'
 executable files with a trailing '*'
 -g show group ownership of file in long listing
 -i print the inode number of each file
 -l long listing giving details about files and directories
 -R list all subdirectories encountered
 -t sort by time modified instead of name

cp - Copy files

cp myfile yourfile

Copy the files "myfile" to the file "yourfile" in the current working directory. This command will create the file "yourfile" if it doesn't exist. It will normally overwrite it without warning if it exists.

cp -i myfile yourfile

With the "-i" option, if the file "yourfile" exists, you will be prompted before it is overwritten.

cp -i /data/myfile

Copy the file "/data/myfile" to the current working directory and name it "myfile". Prompt before overwriting the file.

cp -dpr srcdir destdir

Copy all files from the directory "srcdir" to the directory "destdir" preserving links (-p option), file attributes (-p option), and copy recursively (-r option). With these options, a directory and all its contents can be copied to another dir

ln - Creates a symbolic link to a file.

ln -s test symlink

Creates a symbolic link named symlink that points to the file test
Typing "ls -i test symlink" will show the two files are different with different inodes. Typing "ls -l test symlink" will show that symlink points to the file test.

locate - A fast database driven file locator.

slocate -u

This command builds the slocate database. It will take several minutes to complete this command. This command must be used before searching for files, however cron runs this command periodically on most systems. locate whereis Lists all files whose names contain the string "whereis". directory.

more - Allows file contents or piped output to be sent to the screen one page at a time

less - Opposite of the more command

cat - Sends file contents to standard output. This is a way to list the contents of short files to the screen. It works well with piping.

whereis - Report all known instances of a command

wc - Print byte, word, and line counts

bg

bg jobs Places the current job (or, by using the alternative form, the specified jobs) in the background, suspending its execution so that a new user prompt appears immediately. Use the jobs command to discover the identities of background jobs.

cal month year - Prints a calendar for the specified month of the specified year.

cat files - Prints the contents of the specified files.

clear - Clears the terminal screen.

cmp file1 file2 - Compares two files, reporting all discrepancies. Similar to the diff command, though the output format differs.

diff file1 file2 - Compares two files, reporting all discrepancies. Similar to the cmp command, though the output format differs.

dmesg - Prints the messages resulting from the most recent system boot.

fg

fg jobs - Brings the current job (or the specified jobs) to the foreground.

file files - Determines and prints a description of the type of each specified file.

find path -name pattern -print

Searches the specified path for files with names matching the specified pattern (usually enclosed in single quotes) and prints their names. The findcommand has many other arguments and functions; see the online documentation.

finger users - Prints descriptions of the specified users.

free - Displays the amount of used and free system memory.

ftp hostname

Opens an FTP connection to the specified host, allowing files to be transferred. The FTP program provides subcommands for accomplishing file transfers; see the online documentation.

head files - Prints the first several lines of each specified file.

ispell files - Checks the spelling of the contents of the specified files.

kill process_ids

kill - signal process_ids

kill -l

Kills the specified processes, sends the specified processes the specified signal (given as a number or name), or prints a list of available signals.

killall program

killall - signal program

Kills all processes that are instances of the specified program or sends the specified signal to all processes that are instances of the specified program.

mail - Launches a simple mail client that permits sending and receiving email messages.

man title

man section title - Prints the specified man page.

ping host - Sends an echo request via TCP/IP to the specified host. A response confirms that the host is operational.

reboot - Reboots the system (requires root privileges).

shutdown minutes

shutdown -r minutes

Shuts down the system after the specified number of minutes elapses (requires root privileges). The -r option causes the system to be rebooted once it has shut down.

sleep time - Causes the command interpreter to pause for the specified number of seconds.

sort files - Sorts the specified files. The command has many useful arguments; see the online documentation.

split file - Splits a file into several smaller files. The command has many arguments; see the online documentation

sync - Completes all pending input/output operations (requires root privileges).

telnet host - Opens a login session on the specified host.

top - Prints a display of system processes that's continually updated until the user presses the q key.

traceroute host - Uses echo requests to determine and print a network path to the host.

uptime - Prints the system uptime.

w - Prints the current system users.

wall - Prints a message to each user except those who've disabled message reception. Type **Ctrl-D** to end the message

10.5 QUESTIONS:

1. What is Operating System? State & Explain types of O.S.
2. Write Short Note on
 - a) Windows 7 O.S.
 - b) Linux O.S.
3. Write short note on Linux Distributions.
4. State the advantages of Linux Operating System.
5. Compare windows with Linux.

6. Explain use and execution of following Linux Commands
- a) mkdir
 - b) cd
 - c) mv
 - d) pwd
 - e) rmdir
 - f) chmod
 - g) ls
 - h) cp
 - i) cat
 - j) ftp
 - k) kill
 - l) sleep
 - m) wall

10.6 FURTHER READING:

- ❖ **Operating System Concepts** by Abraham Silberschatz, Greg Gagne and Peter B. Galvin
- ❖ **Modern Operating Systems** by Tanenbaum
- ❖ **Computer System Architecture** by M Morris Mano , Prentice Hall of India, 2001
- ❖ **Computer Architecture and Organization** by John P Hayes, Tata McGraw Hill

