EEE 3104 and ETI 3104 Digital Electronics I

School of Engineering

Bachelor of Science in Electrical and Electronic Engineering
Bachelor of Science in Telecommunication and Information Engineering
Bachelor of Education Technology in Electrical and Electronic Engineering

Instructor: L. Otieno

Notes prepared by A. M. Mbugua (with minor editing and revision)

May 2025

Dedan Kimathi University of Technology

Contents

Chapter 1

The Course

Course outline

Prerequisites

None

Purpose of the course

The aim of this course is to enable the student to understand number systems and codes, Boolean algebra and logic gates, and their application.

Expected learning outcomes

By the end of this course, the learner should be able to;

1. Understand number system and codes
2. Appreciate arithmetic and logic operations using different number systems
3. Use logic gates to design combinational and sequential circuits

Course content

1. *Number systems*: Decimal, binary, octal, and hexadecimal numbers; number system conversion, binary arithmetic, signed binary numbers, floating-point numbers.
2. *Number codes*: Binary Coded Decimal (BCD), Gray, Excess-3; alphanumeric codes, error detecting/correcting (encoding and decoding), code converters, and applications of various codes.
3. *Boolean algebra and logic gates*: circuits, truth tables, logic symbols, levels, Boolean expressions, axioms and postulates of Boolean algebra.
4. *Minimization of Boolean expressions*: Venn diagrams, laws and theorems. Karnaugh maps, SOP and POS, NAND and NOR logic implementation.
5. *Combinational logic circuits*: arithmetic circuits, comparators, encoders, decoders, multiplexers, demultiplexers, parity checker.
6. *Sequential logic circuits*: JK, SR, D, T, flip-flops, counters, Johnson and Ring counters, shift registers.

Laboratory/Practical Exercises

1. Verification of logic gates
2. Construction of adders, subtractors
3. Conversion of decimal to BCD and BCD to decimal (encoding and decoding)

Mode of delivery

- two (2) hour lectures per week
- two (2) hour tutorial per week
- at least five (5) 3-hour laboratory sessions per semester organized on a rotational basis

Course assessment

A total of 100%, the components of which are listed below.

1. Cats **10%**
2. Assignments **5%**
3. Labs **15%**
4. Exam **70%**

Core textbook

1. R. L. Tokheim, *Digital Electronics: Principles and Applications*, McGraw-Hill Education, 2014.

Reference textbooks

1. M. M. Mano and M. D. Ciletti, *Digital Design: With an Introduction to the Verilog HDL, VHDL, and SystemVerilog*, Pearson, 2021.
2. T. L. Floyd, *Digital Fundamentals*, Pearson, 2014.
3. K. M. Anil, *Digital electronics: principles, devices and applications*, McGraw-Hill, 2007.

Simulation software

Download logisim-evolution from GitHub. We will use it for most of our learning exercises.

Chapter 2

Number system

2.1 Introduction

Digital quantities can only take on discrete values while analog quantities vary over a continuous range of values. Examples of digital systems include electronic calculators, digital watches, digital voltmeters and digital computers. Examples of analog devices include pointer-type instruments like speedometers, voltmeters, analog computers, etc.

Advantages of digital systems over analog quantities are:

1. They are easier to design than analog systems.
2. Easy to store large quantities of information.
3. Accuracy and precision are greater.
4. digital circuits are less affected by noise.

2.2 Common number systems

All number systems are based on an ordered set of numbers called digits. The total number of digits used in a system is called the base or radix of the system e.g. base 10 (or radix 10) uses then ten digits 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. The four number systems that are used in digital systems are:

2.2.1 Decimal number systems

This is a positional-value system, that is, the value of a digit depends on its position in the number. It is possible to design a digital system with ten states (decimal) but this would not be easy to design as it would mean designing a circuit with ten discrete voltage levels.

2.2.2 Binary Number System

This is known as Base 2 or Radix 2. It uses only two digits, 0 and 1. In digital systems, these two digits are known as bits. The binary system is a positional value system, with the weights weights as shown in Figure 2.1

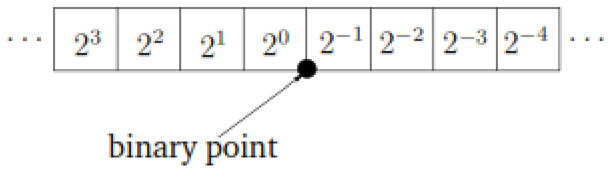


Fig. 2.1: Binary Weighting Factors

2.2.2.1 Binary to Decimal Conversion

The decimal equivalent of the binary number $a_n a_{n-1} \cdots a_1 a_0 . a_{-1} a_{-2} \cdots a_{m-1} a_m$ is given by

$$(2^n \times a_n) + (2^{n-1} \times a_{n-1}) + \cdots + (2^{-1} \times a - 1) + (2^0 \times a_0).(2^{-1} \times a_{-1}) +$$
$$(2^{-2} \times a_{-2}) + \cdots + (2^{-m+1} \times a_{-m-1}) + (2^{-m} \times a_{-m})$$

Exercise 1:

Convert 101.101_2 into decimal

Solution 1:

$$101.101_2 = (2^2 \times 1) + (2^1 \times 0) + (2^0 \times 1).(2^{-1} \times 1) + (2^{-2} \times 0) + (2^{-3} \times 1) = 5.625_{10}$$

2.2.2.2 Decimal to Binary Conversion

For **Integers**, apply repeated division by 2 and read the binary equivalent upwards

Exercise 2:

Covert 23_{10} into binary.

Solution 2:

2	23	
2	11	R 1
2	5	R 1
2	2	R 1
2	1	R 0
	0	R 1

$\uparrow \quad 23_{10} = 10111_2$

For **Fractions**, use repeated multiplication by 2 and read the number downwards as follows

Exercise 3:

Convert 0.8125_{10} into binary.

Solution 3:

0.8125×2	1.625	1	$\downarrow$
0.625×2	1.25	1	
0.25×2	0.5	0	
0.5×2	1.0	1	

$0.8125_{10} = 0.1101_2$

2.2.3 Octal Number System

This is a base 8 number system using the digits 0,1,2,3,4,5,6,7. The weighting factors are as shown in Figure 2.2

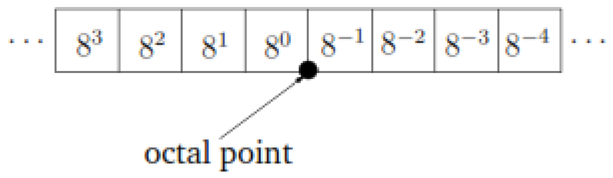


Fig. 2.2: Octal Weighting Factors

2.2.3.1 Octal to Decimal Conversion

The decimal equivalent of the binary number $a_n a_{n-1} \cdots a_1 a_0 . a_{-1} a_{-2} \cdots a_{m-1} a_m$ is given by

$$(8^n \times a_n) + (8^{n-1} \times a_{n-1}) + \cdots + (8^{-1} \times a - 1) + (8^0 \times a_0).(8^{-1} \times a_{-1}) + (8^{-2} \times a_{-2})$$
$$+ \cdots + (8^{-m+1} \times a_{-m-1}) + (8^{-m} \times a_{-m})$$

Exercise 4:

Convert 125.36_8 into decimal.

Solution 4:

$$125.36_8 = (8^2 \times 1) + (8^1 \times 2) + (8^0 \times 5).(8^{-1} \times 3) + (8^{-2} \times 6) = 85.46875_{10}$$

2.2.3.2 Decimal to Octal Conversion

For **Integers**, apply repeated division by 8 and read the octal equivalent upwards

Exercise 5:

Covert 459_{10} into octal.

Solution 5:

$$\begin{array}{c|cl} 8 & 459 & \\ \hline 8 & 57 & \text{R } 3 \\ \hline 8 & 7 & \text{R } 1 \\ \hline & 2 & \text{R } 7 \end{array} \uparrow$$

$459_{10} = 713_8$

For **Fractions**, use repeated multiplication by 8 and read the number downwards as follows

Exercise 6:

Covert 0.78125_{10} into binary

Solution 6:

0.78125×8	6.25	6	$\downarrow$
0.25×8	2.0	2	

$0.78125_{10} = 0.62_8$

2.2.3.3 Octal to Binary Conversion

The procedure here is to convert each octal digit to its 3-bit binary equivalent then to juxtapose these codes to give us the equivalent binary code. The 3-bit binary codes corresponding to the octal digits are shown on the table below:

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

Exercise 7:

Convert 713.62_8 to binary.

Solution 7:

From the above table, we can see that the 3-bit binary codes for 7, 1, 3, 6 and 2 are respectively 111, 001, 011, 110 and 010. We can therefore directly write that $713.62_8 = 111001011 : 110010_2$

2.2.3.4 Binary to Octal Conversion

In this case, we divide the binary number in groups of 3 bits, starting from the binary point. We then use the table shown in the previous section to get the corresponding octal digits.

Exercise 8:

Convert 10111101.1111_2 to octal.

Solution 8:

The procedure is illustrated on Figure 2.3

Hence

$$10111101.1111_2 = 275.74_8$$

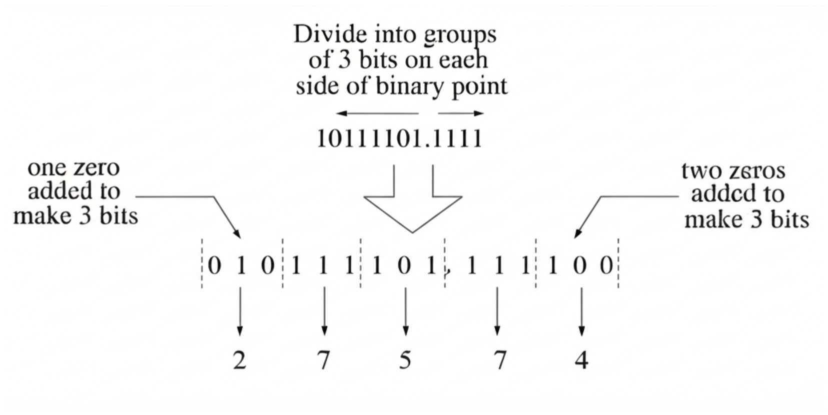

Divide into groups of 3 bits on each side of binary point
10111101.1111
one zero added to make 3 bits
two zeros added to make 3 bits
0 1 0 1 1 1 1 0 1 . 1 1 1 1 0 0
2 7 5 7 4

Fig. 2.3: Binary to octal conversion

2.2.4 Hexadecimal Number System

This is a base 8 number system using the digits 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F. The weighting factors are as shown in Figure 2.4

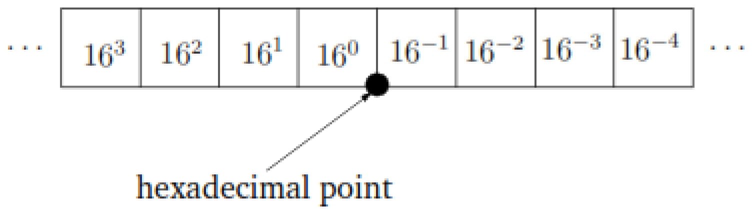


Fig. 2.4: Hexadecimal Weighting Factors

2.2.4.1 Hexadecimal to Decimal Conversion

The decimal equivalent of the binary number $a_n a_{n-1} \cdots a_1 a_0 . a_{-1} a_{-2} \cdots a_{m-1} a_m$ is given by

$$(16^n \times a_n) + (16^{n-1} \times a_{n-1}) + \cdots + (16^{-1} \times a - 1) + (16^0 \times a_0).(16^{-1} \times a_{-1})$$
$$+ (16^{-2} \times a_{-2}) + \cdots + (16^{-m} \times a_{-m})$$

Exercise 9:

Convert $2EA.B_{16}$ into decimal.

Solution 9:

$$2EA.B_{16} = (16^2 \times 2) + (16^1 \times 14) + (8^0 \times 10).(16^{-1} \times 11) = 746.6875_{10}$$

2.2.4.2 Decimal to Hexadecimal Conversion

For **Integers**, we use repeated division by 16 and read the hexadecimal number upwards. If the remainder exceeds 9, we replace it with the corresponding hexadecimal digit.

Exercise 10:

Convert 428_{10} to hexadecimal.

Solution 10:

16	428		
16	26	R 12	
16	1	R 10	
	0	R 1	↑

$428_{10} = 1AC_{16}$

For **Fractions**, use repeated multiplication by 16 and read the number downwards as follows
Exercise 11:

Convert 0.75390625_{10} into Hexadecimal.

Solution 11:

0.75390625×16	12.0625	12	↓
0.0625×16	1.0	1	

$0.75390625_{10} = 0.C1_{16}$

2.2.4.3 Hexadecimal to Binary Conversion

The procedure is similar to that of Octal to Binary conversion, the only difference being that we first convert each hexadecimal digit into 4-bit binary. The 4-bit binary codes representing each hexadecimal digit are tabulated below.

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

Exercise 12:

Convert $2EA.B_{16}$ to binary.

Solution 12:

From the above table, the 4-bit binary codes for 2, E, A, and B are respectively 0010, 1110, 1010 and 1011 so we can therefore directly write that

$$2EA.B16 = 001011101010.1011_2$$

2.2.4.4 Binary to Hexadecimal Conversion

In this case, we divide the binary number in groups of 4 bits, starting from the binary point. We then use the table shown above to get the corresponding hexadecimal digits.

Exercise 13:

Convert 110110011.01011_2 to Hexadecimal.

Solution 13:

The procedure is illustrated on Figure 2.5 below.

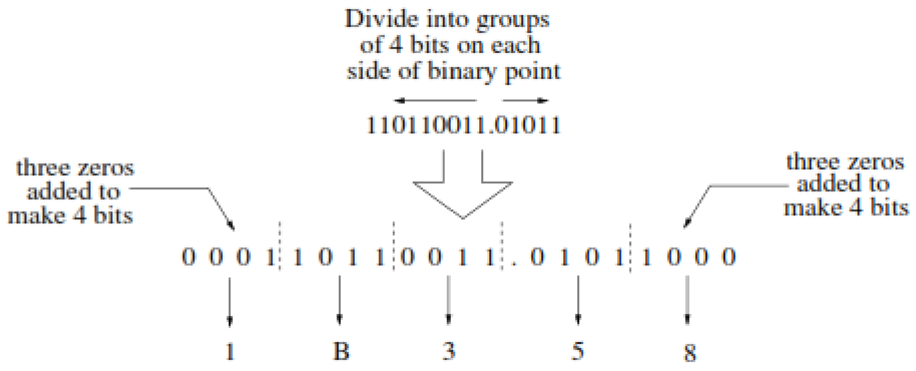

Divide into groups of 4 bits on each side of binary point
110110011.01011
three zeros added to make 4 bits
three zeros added to make 4 bits
0 0 0 1 1 0 1 1 0 0 1 1 . 0 1 0 1 1 0 0 0
1
B
3
5
8

Fig. 2.5: Binary to Hexadecimal conversion

Hence

$$110110011.01011_2 = 1B3.58_{16}$$

2.3 Compliments

Complements are used in digital computers for simplifying the subtraction operation and for logical manipulation. There are two types of complements for each base-r system: the *radix complement* and the *diminished radix complement*.

2.3.1 Diminished radix complement

This is also called the $(r-1)'$s compliment where r is the base of the number. For example for base 10, it is called the 9's compliment, for base 2 it is called the 1's compliment.

The compliment is obtained by subtracting each of the digits from $r-1$. Example;

- the 9's complement of 546700 is 999999 - 546700 $=$ 453299, and
- the 9's complement of 012398 is 999999 - 012398 $=$ 987601.

The l's complement of a binary number is obtained by subtracting each digit from 1. When subtracting binary digits from 1, we can have either $1-0=1$ or $1-1=0$, which causes the bit to change from 0 to 1 or from 1 to O. Therefore, the 1's complement of a binary number is formed by changing 1's to 0's and 0's to 1's. Example;

- the 1's complement of 1011000 is 0100111, and
- the 1's complement of 0101101 is 1010010.

The $(r-1)$'s complement of octal or hexadecimal numbers is obtained by subtracting each digit from 7 or F (decimal 15), respectively.

2.3.2 Radix compliment

Also called the r's compliment. It is obtained by adding 1 to the $(r-1)$'s compliment. Example;

- The l0's complement of decimal 2389 is $7610 + 1 = 7611$ and is obtained by adding 1 to the 9's-complement value.
- The 2's complement of binary 101100 is $010011 + 1 = 010100$ and is obtained by adding 1 to the 1's-complement value.

2.4 Signed Binary Numbers

Since digital computers and calculators handle negative as well as positive numbers, some means is required for representing the sign of the number ($+$ or $-$). This is done by the use of a sign bit. The most significant bit of a binary number is used to denote the sign of the number.

There are three notations that are commonly used representing signed numbers. These are:

1. Sign-Magnitude Notation
2. Ones (1's) Complement Notation
3. Twos (2's) Complement Notation

In all these notations, positive numbers have the Most Significant Bit (MSB) as zero, while negative numbers have MSB as 1.

2.4.1 Sign-Magnitude Notation

To obtain the sign-magnitude notation of a given number, we first obtain its unsigned binary equivalent using the methods described in the previous sections. If the number is positive, we then add a zero (0) to become the MSB, and if the number is negative, we add a one (1) to become the MSB.

Exercise 14:

Convert $+53$ and -53 to binary in sign-magnitude notation.

Solution 14:

The unsigned binary code for 53 can be obtained by successive division by 2 as 110101. For +53, we add a '0' sign bit as MSB to give the binary sign-magnitude code for +53 as 0110101. For -53, we add a '1' for a sign bit to get 1110101.

2.4.2 Ones (1's) Complement Notation

To get the ones complement notation for a positive number, the unsigned binary notation of the number is obtained, after which a zero (0) is added to the number as the MSB (This is similar to the Sign-Magnitude notation).

The ones complement notation of a negative number is obtained from the corresponding positive binary number by changing each zero in the digit to a 1, and each 1 in the positive binary number to a zero.

As an example, we saw in the previous section that $53 = 110101$ in unsigned binary. +53 would be represented by 0110101 in Ones Complement Notation (OCN). To represent -53 in OCN, we simply complement all the bits in +53 to get 1001010.

2.4.3 Twos Complement Notation

The procedure for obtaining the Twos Complement Notation (TCN) of a positive number is similar to that of obtaining OCN for a positive number.

For a negative number, you add 1 to the Least Significant Bit (LSB) position of the ones complement notation of the number.

Exercise 15:

Obtain the TCN of +53 and -53.

Solution 15:

53 = 110101 in unsigned binary.

Adding a sign bit gives +53 = 0110101 in TCN.

To obtain the code for -53, we first obtain the ones complement notation of the code 0110101, which is 1001010. We then add 1 to the LSB position to get 1001011, which is the TCN of -53.

This can be verified as follows

$$(-1 \times 2^6) + (1 \times 2^3) + (1 \times 2^1) + (1 \times 2^0)$$

$$= -64 + 8 + 2 + 1 = -53$$

Exercise 16:

Convert -29.625 into Twos Complement Binary.

Solution 16:

$$29.625 = 11101.101$$

$$+29.625 = 011101.101$$

$$-29.625 = 100010.010 \quad \textbf{OCN}$$

$$-29.625 = 100010.011 \quad \textbf{TCN}$$

$$(-1 \times 2^5) + (1 \times 2^1) + (1 \times 2^{-2}) + (1 \times 2^{-3}) = -29.625$$

2.5 Range of signed integer numbers

For signed binary numbers, the range of integer numbers that can be represented is determined by the number of bits used to represent the number. If n bits (inclusive of the sign bit) are used to represent integers, the range of integers that can be represented is given by

$$\text{Range} = (-2^{n-1}) \text{ to } (+2^{n-1} - 1). \tag{2.1}$$

Exercise 17:

(a) What is the range of signed integers that can be represented by $n = 16$ bits?

(b) What is the total number of unique integers that are represented in the range calculated in (a)?

(c) What is the range of unsigned integers that can be represented by $n = 16$ bits?

(d) What is the total number of unique integers represented in the range calculated in (c)?

(e) Comment on the results obtained in (a) and (c).

Solution 17:

(a) $\text{Range} = -2^{15}$ to $2^{15} - 1$, which is $-32,768$ to $+32,767$.

(b) Number of integers $= 32,767 + 32,768 + 1 = 2^{16} = 65,536$.

(c) $\text{Range} = 0$ to $2^{16} - 1$, which is 0 to $65,535$.

(d) Number of integers $= 65,535 - 0 + 1 = 2^{16} = 65,536$.

(e) The results are equal. Both signed and unsigned numbers represent a similar **total** number of unique integers.

Exercise 18:

Which signed number systems can represent $+0$ and -0?

Solution 18:

Try this by yourself.

2.6 Subtraction using compliments

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

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

Exercise 19:

Using 10's complement, subtract $72532 - 3250$.

Solution 19:

$$\begin{aligned} \text{M} = &\quad 72532 \\ \text{10' s complement of } N = &\quad +96750 \\ \text{Sum} = &\quad 169282 \\ \text{Discard end carry } 10^5 = &\quad -100000 \\ \text{Answer} = &\quad 69282 \end{aligned}$$

Exercise 20:

Using l0's complement, subtract $3250 - 72532$.

Solution 20:

$$\begin{aligned}
M = {} & 03250 \\
\text{10's complement of } N = {} & +27468 \\
\text{Sum} = {} & 30718 \\
\text{There is no end carry.} & \\
\text{Answer: -(10's complement of 30718)=} {} & -69282
\end{aligned}$$

Exercise 21:

Given the two binary numbers $X = 1010100$ and $Y = 1000011$, perform the following subtraction using 2's complements.

i. $X - Y$

ii. $Y - X$

Solution 21:

i. $X - Y$

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

ii. $Y - X$

$$\begin{aligned}
Y = {} & 1000011 \\
\text{2's complement of } X = {} & +0101100 \\
\text{Sum} = {} & 1101111 \\
\text{There is no end carry.} & \\
\text{Answer: } Y - X = -\text{(2's complement of 1101111)} = {} & -0010001
\end{aligned}$$

Subtraction of unsigned numbers can be done also by means of the $(r - 1)$'s compliment. The $(r - l)$'s complement is one less than the r's complement.

Because of this, the result of adding the minuend to the complement of the subtrahend produces a sum that is 1 less than the correct difference when an end carry occurs. Removing the end carry and adding 1 to the sum is referred to as an end-around carry.

Exercise 22:

Given the two binary numbers $X = 1010100$ and $Y = 1000011$, perform the following subtraction using 1's complements.

i. $X - Y$

ii. $Y - X$

Solution 22:

i. $X - Y$

X=	1010100
1's complement of Y =	+ 0111100
Sum =	10010000
end carry round	0010000
	+1
Answer: X - Y =	0010001

ii. $Y - X$

Y=	1000011
1's complement of X =	+ 0101011
Sum =	1101110

There is no end carry.

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

2.7 Floating-point numbers

Floating-point numbers contain both the integer part and the fraction part. They are *real* numbers. Floating point numbers are typically represented using a *mantissa* and an *exponent*. The *mantissa* represents the magnitude of the number and the *exponent* represents the number of places that the decimal point is to be moved.

For binary numbers, there are standards used to represent floating numbers. One such standard is defined by ANSI/IEEE Standard 754-1985 (you can look this up online). There are three forms of representation of complex numbers:

i. single-precision (using 32 bits),

ii. double-precision (using 64 bits), and

iii. extended-precision (using 80 bits).

The format of representation is the same with the difference only coming in the number of bits used for representation of floating-point numbers. The format is represented in Table 2.1. The format for m

Table 2.1: Binary floating-point format (ANSI/IEEE Standard 754-1985)

Sign (s)	Exponent (e)	Mantissa (m)

when the number $\mathbf{1.XXXX} \times 2^n$ is $\mathbf{XXXX}$. The 1 to the left of the decimal point (the integer) is assumed to be present. The exponent is writen biased (i.e. $e =$ actual exponent $+ 2^{n_\mathrm{e}-1} - 1$, where n_e are the number of exponent bits).

The sign, exponent, and mantissa bits are allocated as shown in Table 2.2

The value of a floating-point number is obtained using

$$(-1)^s \times m \times 2^{e-b}, \tag{2.2}$$

where $b = 2^{n_\mathrm{e}-1}$ is the bias.

Table 2.2: Allocation of bits (ANSI/IEEE Standard 754-1985)

	s bits, n_s	e bits, n_e	m bits n_m
single-precision	1	8	23
double-precision	1	11	52
extended-precision	1	15	64

Exercise 23:

Represent -10110100.10001_2 in single-precision floating-point format.

Solution 23:

The components are:

$$\begin{aligned} \mathrm{s} &: 1_2, \\ \mathrm{m} &: 1.011010010001_2, \text{and} \\ \mathrm{e} &: (7 + 2^7 - 1)_{10} = 134_{10} = 10000110_2. \end{aligned}$$

Therefore:

s	e	m
1	10000110	01101001000100000000000

Exercise 24:

Represent -407688_{10} in binary single-precision floating-point format.

Solution 24:

Try this by yourself.

Exercise 25:

Convert the following single-precision floating-point number to its equivalent decimal.

s	e	m
1	10010001	10001110001000000000000

Solution 25:

Try this by yourself.

Chapter 3

Number codes

3.1 Binary Codes

3.1.1 Binary Coded Decimal (BCD) Code

BCD code represents each digit of a decimal number by a 4-bit binary number. The codes used are tabulated below:

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

Note that BCD code uses binary codes 0000 to 1001 to represent decimal digits, it does not use codes 1010, 1011, 1100, 1101, 1110 and 1111. It is a weighted code.

To convert a decimal number to BCD code, we simply write out the BCD code for each digit e.g. to convert the decimal number 137 to BCD, we can see from the above table that the BCD code for 1 is 0001, for 3 is 0011 and for 7 is 0111, so the BCD code for 137 is 000100110111.

To convert a BCD code number to decimal, we simply group the bits in groups of 4 bits each and write out the decimal digit corresponding to each decimal digit.

The main advantage of BCD code is the relative ease of converting to and from decimal. BCD code is used in digital machines whenever decimal information is either applied as inputs or displayed as outputs e.g. digital voltmeters, digital clocks, e.t.c. use BCD because they display information in decimal. Electronic calculators use BCD because the input numbers are entered in decimal via the keypad and the output numbers displayed in decimal. However, BCD code is not used in modern high-speed digital computers because:

1. it requires more storage space, and
2. the arithmetic with BCD is more complicated.

Can you explain each of the points enumerated above?

3.1.2 Excess-3 Code

The excess-3 code (also known as Xs-3 code) for a decimal number is obtained in the same manner as for BCD, except that 3 is added to each digit before encoding it in binary

The table below shows the codes used by Xs-3 code, and these are listed alongside BCD codes.

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

The Xs-3 code does not use codes 0000, 0001, 0010, 1101, 1110 and 1111. The advantage of this code is that at least one 1 is present in all codes, providing an error-detection ability.

3.1.3 Gray Code

In gray code, only one bit changes in going from one number to the next. It is a *non-weighted code* - bit positions in the code do not have any specific weights attached to them.

The table below shows the Gray codes for the first 16 decimal digits.

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

Binary-to-Gray code conversion

The following rules are followed to convert a binary code to Gray code.

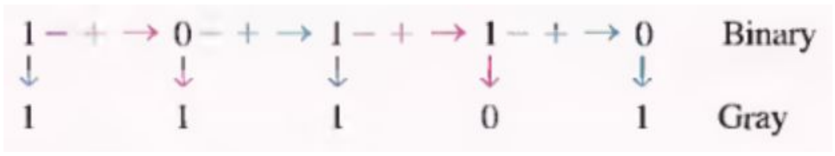


Fig. 3.1: Binary-to-Gray code conversion.

1. The MSB is the same as the corresponding MSB in the binary number.
2. Going from the MSB to the LSB, add each adjacent pair of binary code bits to get the next Gray code bit. Discard any carries.

Exercise 26:

Convert 10110 to Gray code.

Solution 26:

Following the procedure, the results are obtained as follows: The Gray code of the binary number 10110 is 11101.

Gray-to-Binary conversion

The following rules are followed to convert a Gray code to binary code.

1. The MSB is the same as the corresponding MSB in the Gray code.
2. Going from the MSB to the LSB, add each binary code generated to the next Gray code bits to get the next Gray code bit. Discard any carries.

Exercise 27:

Convert 11011 to binary code.

Solution 27:

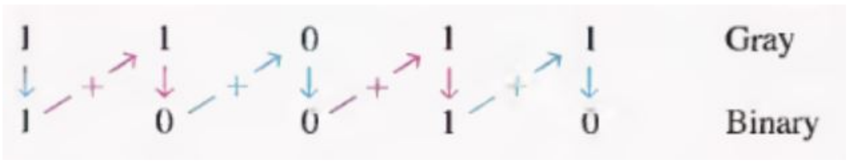


Fig. 3.2: Gray-to-Binary conversion.

Following the procedure, the results are obtained as follows: The binary number of the Gray code 11011 is 10010.

Applications of Gray codes

Observe that only one bit position changes in the binary code in moving from one number to the next. Gray code is often used in situations where other codes might produce erroneous or ambiguous results during those transitions in which more than 1 bit of the code is changing e.g. in the transition from 7 to 8 in binary, all bit positions change, and in a practical circuit, these bit positions may not change at exactly the same time and this could cause problems in some circuits. Another application of Gray code is in *Karnaugh maps* (a tool used in the process of digital design), as we will see later in this unit.

3.1.4 Alphanumeric codes

Communication requires more than just numbers. You also have to convey alphabetic characters and symbols. You only need to look at your computer's keboard (or smartphone's keypad) to see that there are many symbols and characters that also need some digital representation.

At the minimum, alphaneumeric codes have to represent letters of the alphabet and the 10 decimal numbers. If the alphabet to be represented is the English alphabet, a total number of 36 items have to be represented (0 to 9 and a to z). A minimum of 6 bits are required to represent each of the 36 items uniquely. In practice, in addition to the 36 items, you also need a way to represent symbols and other necessary instructions. A 6-bit code has room for $2^6 - 36 = 64 - 36 = 28$ symbols to be represented alongside the alphanumeric codes.

ASCII

The most common of the alphanumeric codes is the *American Standard Code for Information Interchange (ASCII)* (pronouced '***askee***'). It has 128 characters and symbols represented by a 7-bit binary code. It is normally presented as an 8-bit code, with the MSB always set to 0. The code can also be represented in the hexadecimal format. For instance, 01110000 is represented as $7F$.

You can search an ASCII table online. Generally, the code is grouped as follows:

1. the first 32 ASCII characters represent non-graphic commands that are never printed, such as *CONTROL*, *ESCAPE*, e.t.c.
2. The other characters are used to represent the decimal digits, the alphabet (lowercase and uppercase), punctuation signs and other commonly used symbols.

Extended ASCII

These codes make use of the remaining 128 combinations of the 8-bit code (from 80_{16} to FF_{16}), to have a total of 256 combinations. Extended ASCII characteros fall into the following categories.

- Non-English alphabetic characters
- Currency symbols
- Greek letters
- Mathematical symbols
- Drawing characters
- Bar graphing characters
- Shading characters

3.2 Error detection and correction codes

In communications systems, one of the objectives is reliable transfer of information. While it is impossible to avoid errors because of *imperfect channels*, communication systems can be designed so that transfer of information is affected by an acceptable amount of errors. To achieve acceptable reliablility, a method of for detecting and/or detecting errors in transmissions at the receiver side is normally implemented. Desingning codes for reliable (low transmission errors) and efficient (optimal user of resources) transmission is the subject of *Information Theory*, which most of you are likely to meet in some form or another in the course of your studies.

To detect and/or correct errors, redundant bits are added to binary codes. Error correction requires the error to be detected first. The design of the code determines how many erroneous bit errors can be detected in a code. Some of the methods used to do this are enumerated here for illustration.

i. Parity check for *error detection*.
ii. Hamming codes for *error correction*.

3.2.1 Parity Check

In the parity method, a parity bit is added to a code to tell if *the number of 1's is odd or even*. There are two parity methods:

i. even parity, or
ii. odd parity.

A system may only choose to use one of them, not both.

In **even parity**, a parity bit is added to a code to make the total number of 1's even.

In **odd parity**, a parity bit is added to a chode to make the total number of 1's odd.

Exercise 28:

Add parity bits to the following BCD codes for an *even* parity system.

i. 0110

ii. 0000

iii. 1000

Solution 28:

i. There are two 1's in 0110, the parity bit is 0 to make total 1's even. The code with parity bit is: **00110**.

ii. There are zero 1's in 0000, the parity bit is 0 to make total 1's even. The code with parity bit is: **00000**.

iii. There is one 1's in 1000, the parity bit is 1 to make total 1's even. The code with parity bit is: **11000**.

Exercise 29:

Add parity bits to the following BCD codes for an *odd* parity system.

i. 0110

ii. 0000

iii. 1000

Solution 29:

i. There are two 1's in 0110, the parity bit is 1 to make total 1's odd. The code with parity bit is: **10110**.

ii. There are zero 1's in 0000, the parity bit is 1 to make total 1's odd. The code with parity bit is: **10000**.

iii. There is one 1's in 1000, the parity bit is 0 to make total 1's odd. The code with parity bit is: **01000**.

Error detection using parity bit

Parity check method can only detect an error occuring in an odd number of bits in a group. For instance, assuming the **1001** is to be transmitted in an *even parity* system, the transmitted group is **01001**. Suppose that on the reception side, **01101** is received, with one bit corrupted by the *transmission channel*. Parity check at the receiver detects that the number of 1's are odd and therefore an error is detected.

Other examples are given in Table 3.1 for even parity.

Table 3.1: Parity check: even parity. Errors are indicated using red.

code	parity bit	parity bit + code	Received	Parity check	reason
0101	0	00101	01100	no error	2 errors (even)
0001	1	10001	01101	error	3 errors (odd)
0110	0	00110	10110	erorr	1 error (odd)

Similarly, for odd parity, even errors are not detected by parity check. In Table 3.2, assuming the errors are in the same locations as the illustration for even parity in Table 3.1.

Table 3.2: Parity check: odd parity. Errors are indicated using red.

code	parity bit	parity bit + code	Received	Parity check	reason
0101	1	10101	01100	no error	2 errors (even)
0001	0	00001	11101	error	3 errors (odd)
0110	1	10110	00110	erorr	1 error (odd)

3.2.2 Hamming code

Parity check can detect odd errors. In practice, it is used to detect a single error. In order to correct the error, the location of the error has to be identified. Now more bits can be added to the transmitted code in order to identify the location of the error and to correct it. Here, an illustration of a Hanning code that uses 3 parity bits for a 4-bide code.

Constructing Hamming codes

Let the number of parity bits be $p = 3$ and the number of bits in the code be $d = 4$. Let the parity bits be represented by P_i where $i = 0, \ldots, p - 1$. Also, let the data bits be represented by D_j where $j = 0, \ldots, d - 1$. In a (7,4) Hamming code (total of 7 bits, with 4 data bits and 3 parity bits), the bits are placed as shown in Table 3.3.

Table 3.3: (7,4) Hamming code construction

bit 6 (MSB)	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0 (LSB)
P_0	P_1	D_3	P_2	D_2	D_1	D_0

In Table 3.3, parity is used as follows:

i. bit P_2 checks P_2, D_2, D_1 and D_0,

ii. bit P_1 checks P_1, D_3, D_1 and D_0, and

iii. bit P_0 checks P_0, D_3, D_2 and D_0.

Exercise 30:

Create a (7,3) Hamming code for 1001. Use even parity.

Solution 30:

Table 3.4: (7,4) Hamming code using even parity.

Designation	P_0	P_1	D_3	P_2	D_2	D_1	D_0
Data			1		0	0	1
Parity	0	0		1			
Hamming code	0	0	1	1	0	0	1

Error detection and correction using Hamming codes

To detect and correct single errors, construct a code from the results of parity checks with P_2, P_1, and P_0. You can use the following convention. When the parity check indicates there is no error, put a 1 and when the parity check indicates the presense of an error, put a 0. The resulting 3-bit code indicates the error position in the 7-bit Hamming code.

Here is the procedure.

i. Check parity using P_2. If the check is correct, set $C_2 = 1$, else set $C_2 = 0$.

ii. Check parity using P_1. If the check is correct, set $C_1 = 1$, else set $C_1 = 0$.

iii. Check parity using P_0. If the check is correct, set $C_0 = 1$, else set $C_0 = 0$.

iv. Construct code $C_2C_1C_0$ to point to the location of bit error. 111 represents no error, and errors at bit 0 (LSB) to bit 6 (MSB) are represented by 000 to 110, respectively.

Exercise 31:

For the data code 1001, the corresponding Hamming code using even parity is 0011001. Suppose the code 1011001 is received after transmission. Show how the error can be corrected.

Solution 31:

i. $P_2D_2D_1D_0 = 1001$, even parity check is correct, $C_2 = 1$

ii. $P_1D_3D_1D_0 = 0101$, even parity check is correct, $C_1 = 1$

iii. $P_0D_3D_2D_0 = 1101$, even parity check is false, $C_0 = 0$

Therefore, the code is 011, representing an error at bit position 6. We can go ahead and flip the bit to correct the error in 1011001 to 0011001.

Exercise 32:

i. Create a (7,4) Hamming code for 0111. Use odd parity.

ii. After transmission, there is an error in bit position 0. Show how you would detect and correct it.

Solution 32:

Try this on your own.

Chapter 4

Boolean algebra and logic gates

4.1 Introduction

4.1.1 Boolean Algebra

In Boolean algebra, the variables (called *Boolean variables*) are allowed to have only two possible values, 0 and 1. For example in the expression

$$y = f(A, B) \tag{4.1}$$

y is a function of variables A and B. A and B are Boolean variables and can only take the possible values of 0 or 1. A more general form of a Boolean expression for n (X_i where $i = 0, \ldots, n-1$) variables is written

$$y = f\left(X_0, X_1, \ldots, X_{n-1}\right) \tag{4.2}$$

Basic operations of Boolean algebra

Boolean algebra has only 3 basic operations;

1. *Logical addition* (the **OR** operation), symbol '$+$',
2. *Logical Multiplication* (the **AND** operation), symbol '.', and
3. *Logical Complementation* (the **NOT** operation), symbol '$^{-}$', *, or $'$.

Any Boolean function, however complex, can be written as a combination of these three operations.

4.1.2 Logic gates

A logic gate is an electronic device used to make logic decisions. It is the basic building block for all digital systems. There are three basic logic gates

1. Inverter or NOT gate,
2. OR gate, and
3. AND gate.

4.1.3 Truth Table

It has inputs and outputs. The output usually depends on different combinations of the inputs.

Example 1
$F = \overline{A}$

A	F
0	1
1	0

Example 2
$F = A + B$

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

4.2 Logic gates

Other than the basic gates, there are other common gates used to implement Boolean expressions. The input-output operation of the basic gates and other commonly used gates are described section that follows.

4.2.1 Basic gates

NOT gate

The NOT gate has a single input and a single output. Its symbol is shown in Fig. 4.1. The small circle at the ouput side denotes the NOT operation. Without the small circle, the symbol represents a signal *buffer* (amplifying the current, maintaining the voltage).

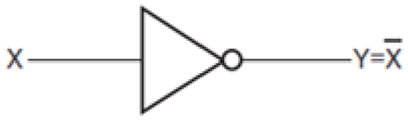


Fig. 4.1: NOT gate

The truth table for the NOT gate is

X	Y
0	1
1	0

OR gate

It is used to implement the OR operation. The operation operates on **two** or **more** variables. The OR function operates on the following rules

0+0=0

0+1=1

1+0=1

1+1=1

The logic symbol for the OR gate is

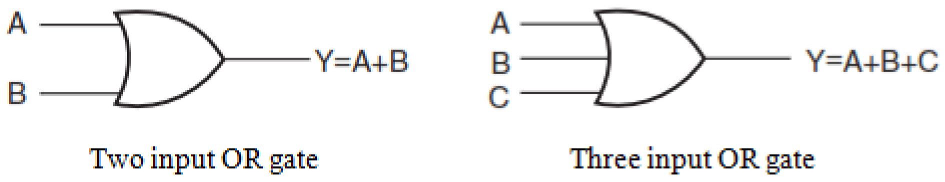


Fig. 4.2: OR gate

The truth table of two input and three input OR gates are shown below.

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

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

In general, the OR operation produces a result of 1 when **any** of the input variables is 1.

AND gate

It is used to implement the AND operation. The AND operation operates on two or more variables. The AND function operates on the following rules

0.0=0

0.1=0

1.0=0

1.1=1

The logic symbol for the AND gate is

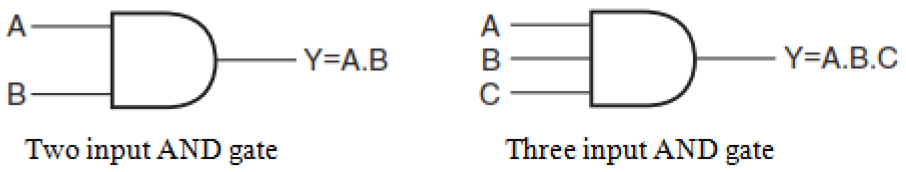


Fig. 4.3: AND gate

The truth table of two input and three input AND gates are

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

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

The output is equal to 1 only for the single case when all the inputs are 1

4.2.2 Other common logic gates

XOR (Exclusive OR) gate

This is a gate that gives a high only when either of the inputs is high.The logic symbol for the XOR gate is

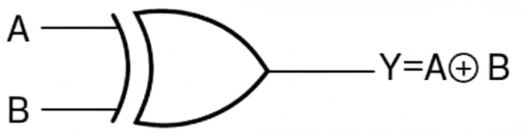


Fig. 4.4: XOR gate

The truth table is

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

$$Y = A \oplus B$$

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

XNOR (Exclusive-NOR gate)

This gate is used to complement the XOR operation

The logic symbol is

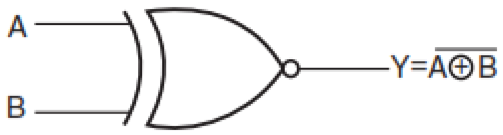


Fig. 4.5: XNOR gate

The truth table is

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

$$Y = \overline{A \oplus B}$$

$$Y = \overline{AB} + AB$$

4.2.3 Universal gates

Universal gates can be used to implement all the other gates. That is, one type of universal gates can implement AND, OR, and NOT operations. There are two universal gates

i. NAND gates

ii. NOR gates

NOR gate

This is a gate that gives an output of 1 only when all inputs are 0. It is used to complement the OR gate. The logic symbol is

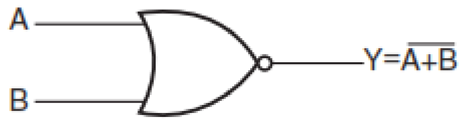


Fig. 4.6: NOR gate

The truth table is

A	B	Y
0	0	1
0	1	0
1	0	01
1	1	0

$Y = \overline{A+B}$ $Y = \overline{AB}$

NAND gate

This is a gate that gives a low output when all inputs are high. It complements the AND gate. The logic symbol is

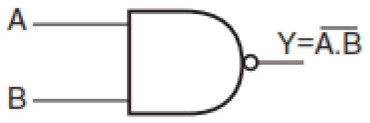


Fig. 4.7: NAND gate

The truth table is

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

$$Y = \overline{AB}$$

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

Exercise 33:

i. Implement all the other gates using NOR gates

ii. Implement all the other gates using NAND gates

iii. Implement the following using NOR and NAND gates

- $F = AB + \overline{B}C + CD$
- $F = ABC + \overline{ABC}$

Solution 33:

Try those by yourself. We will revisit the process later.

4.3 Laws of Boolean Algebra

Laws of Boolean algebra can be put in the following categories

i. OR laws

ii. AND laws

iii. Commutative laws

iv. Associated laws

v. Distributive laws

vi. Absorptive laws

Let us look at the laws contained in each of the categories now.

4.3.1 OR laws

$$A + 0 = A$$

$$A + 1 = 1$$

$$A + A = A$$

$$A + \overline{A} = 1$$

4.3.2 AND laws

$$A.0 = 0$$

$$A.1 = A$$

$$A.A = A$$

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

4.3.3 Commutative laws

Laws that allow change of position of variables in OR and AND functions or expressions

$$A + B = B + A$$

$$A.B = B.A$$

4.3.4 Associative laws

Laws that allow removal of brackets from logic expressions and regrouping of variables

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

$$A.(B.C) = (A.B).C$$

4.3.5 Distributive laws

Laws that permit factoring or multiplying out of an expression

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

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

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

4.3.6 Absorptive laws

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

$$A + AB = A$$

$$A(\overline{A} + B) = AB$$

4.4 DeMorgan's theorems

DeMorgan's theorems relate the NAND and NOR logic. There are two theorems and they can be stated as follows.

1. The complement of ANDed variables is equal to the OR of the complements of the variables.
2. The complement of ORed variables is equal to the AND of the complements of the individual variables.

For two variables A and B, the laws are written mathematically as

1. $\overline{XY} = \overline{X} + \overline{Y}$
2. $\overline{(X+Y)} = \overline{X}\overline{Y}$

4.5 Proofing Boolean expressions

Boolean expressions may be proved by

1. truth table, or
2. algebraic means.

4.5.1 Proof by truth table

You can use a truth table to prove that the left hand side of a Boolean expression equals the right hand side. Let us illustrate this using an example.

Exercise 34:

Use a truth table to prove that $AB + \overline{A}C + BC = AB + \overline{A}C$

Solution 34:

A	B	C	AB	$\overline{A}B$	BC	$AB + \overline{A}B + BC$	$AB + \overline{A}C$
0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0
0	1	0	0	1	0	1	1
0	1	1	0	1	1	1	1
1	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0
1	1	0	1	0	0	1	1
1	1	1	1	0	1	1	1

The columns $AB + \overline{A}C + BC$ and $AB + \overline{A}C$ are identical so the two expressions are identical. Proof by truth table is known as proof by perfect induction

4.5.2 Proof by algebraic means

This requires a mastery of the laws of Boolean algebra. As with many exercises, this approach may seem challenging at first. However, the more exercises you do, the better you get at mastering the laws.

Again, let us illustrate by way of an example.

Exercise 35:

Use algebraic means to show that

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

Solution 35:

$$\begin{aligned}(A+B)(A+\overline{B})(\overline{A}+C) &= (AA+AB+B\overline{B}+A\overline{B})(\overline{A}+C)\\ &= (A+AB+A\overline{B})(\overline{A}+C)\\ &= (A+A)(\overline{A}+C)\\ &= A(\overline{A}+C)\\ &= AC\end{aligned}$$

Exercise 36:

Prove that

1. $AB + A\overline{B}C + AB\overline{C} = A(B+C)$
2. $\overline{\overline{AB} + \overline{A} + AB} = 0$
3. $AB + \overline{AC} + A\overline{B}C(AB+C) = 1$

Solution 36:

Try these by yourself.

Chapter 5

Minimization of Boolean expressions

5.1 Standard Forms of Boolean Expressions

There are two standard forms of Boolean expressions

1. Standard Sum of Products (SSOP)
2. Standard Product of Sums (SPOS)

5.1.1 Standard Sum of Products

Given the expression $F = AB + BC + ABC + AC$ The terms AB, BC, ABC and AC are products and are all combined with an OR operation. The expression is said to be in Sum of Products form.

Literals in every product appear in compliment or non compliment form. No single bar can cover more than one literal e.g $\overline{ABC}$

In the expression $F = AB + BC + ABC + AC$, not all the product terms contain all the literals. To express the expression in standard sum of products form, we must add the missing literals to all the product terms.

To do this we use the Boolean algebra law $(A + \bar{A}) = 1$ and $(A.1) = A$

$$\begin{aligned} F &= AB + BC + ABC + AC \\ &= AB(C + \bar{C}) + BC(A + \bar{A}) + ABC + AC(B + \bar{B}) \\ &= ABC + AB\bar{C} + \bar{A}BC + A\bar{B}C \end{aligned}$$

Each individual term in the standard sum of products expression is known as a ***minterm***, and it is denoted by m_i where i is used to distinguish the minterms. Each minterm will have a logical value of 1 only when all the terms have a logical value of 1. An illustration is provided for 3 literals A, B, and C.

A	B	C	minterms	symbol
0	0	0	$\bar{A}\bar{B}\bar{C}$	m_0
0	0	1	$\bar{A}\bar{B}C$	m_1
0	1	0	$\bar{A}B\bar{C}$	m_2
0	1	1	$\bar{A}BC$	m_3
1	0	0	$A\bar{B}\bar{C}$	m_4
1	0	1	$A\bar{B}C$	m_5
1	1	0	$AB\bar{C}$	m_6
1	1	1	ABC	m_7

The expression can be written as

$$\begin{aligned} F &= m_7 + m_6 + m_3 + m_5 \\ &= \textstyle\sum_m(3,5,6,7) \end{aligned}$$

5.1.2 Standard product of sums

Given the expression

$$F = (AB + C)(B + AC)$$

Using the distributive law

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

The expression is said to be in product of sums form. Each sum gives a 1 when any of the literals within it is a 1.

To express the expression in standard product of sums form, we must add the missing literals to all the sum terms. To do this we use the Boolean algebra law $(A.\bar{A}) = 1$ and $(A + 0) = 0$.

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

Table 5.1: Maxterms for 3 variables.

A	B	C	maxterm	symbol
0	0	0	$A+B+C$	M_0
0	0	1	$A+B+\bar{C}$	M_1
0	1	0	$A+\bar{B}+C$	M_2
0	1	1	$A+\bar{B}+\bar{C}$	M_3
1	0	0	$\bar{A}+B+C$	M_4
1	0	1	$\bar{A}+B+\bar{C}$	M_5
1	1	0	$\bar{A}+\bar{B}+C$	M_6
1	1	1	$\bar{A}+\bar{B}+\bar{C}$	M_7

Using distributive law

$$\begin{aligned} F &= (A+B+C)(A+B+\bar{C})(A+B+C)(A+\bar{B}+C)(A+B+C)(\bar{A}+B+C) \\ &= (A+B+C)(A+B+\bar{C})(A+\bar{B}+C)(\bar{A}+B+C) \end{aligned}$$

Each term in a standard product of sums expression is called a ***maxterm***, and is denoted by M_i, where i is used to distinguish between different combinations of literals. Each Max-term will have a logical value of 0 only when all the terms have a logical value of 0. Therefore, in the truth table, the literals are inverted. Example, for variable A, 0 represents literal A and 1 represents literal $\bar{A}$. This is the opposite of what happens in a SOP truth table. An example of a POS truth table is shown in Figure 5.1.

$$\begin{aligned} F &= M_0.\, M_1.\, M_2.\, M_4 \\ &= \prod_M (1,2,3,4) \end{aligned}$$

5.2 Karnaugh maps (K-maps)

This is a graphical method used to simplify Boolean expressions. Each combination of inputs is represented by a square/cell in the map. The number of squares is equal to 2^n where n is the number of input variables.

The binary equivalent of the input variables is represented using Gray code format to get the adjacent squares.

Once a K-map has been filled with zeros and ones, the sum of product expression can be obtained by ORing together those cells that contain ones.

The product of sums expression can be obtained by ANDing together those cells that contain zeros and inverting the literals (i.e. A becomes $\bar{A}$ and so forth).

As indicated before, the number of squares in a K-map depends on the number of variables in the Boolean expression. Here, we will illustrate the design and use of 2-, 3-, and 4-variable K-maps. As the number of variables grow, their construction and use for minimization become a little more tedious.

5.2.1 Constructing K-maps

Two Variables K-map

Consider the 2 variables truth table shown below

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

The truth table can be converted to a K-map with 4 cells as follows

The K-map can be drawn with the variable A on the vertical side and B on the horizontal side or vice versa.

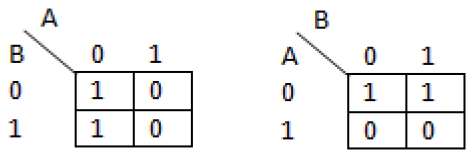


Fig. 5.1: 2-Variables K-map

Three Variables K-map

Consider the 3-input truth table below

A	B	C	D
0	0	0	1
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	0

This can be represented using K-map as shown below

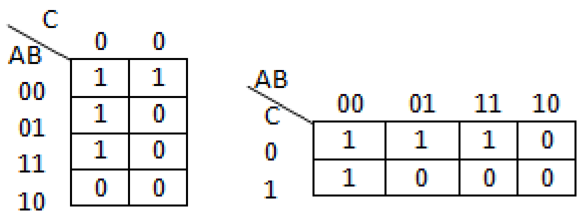


Fig. 5.2: 3-Variables K-map

NB: The K-map cells are labelled in such a way that adjacent cells differ only in one variable

Four Variables K-map

Consider the four variables truth table below

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

This may be represented using K-map as shown below

AB / CD	00	01	11	10
00	0	0	0	0
01	1	1	1	0
11	0	0	1	0
10	0	0	0	0

CD / AB	00	01	11	10
00	0	1	0	0
01	0	1	0	0
11	0	1	1	0
10	0	0	0	0

Fig. 5.3: 4-Variables K-map

5.2.2 Loopings in a K-map

Logic functions can be simplified by properly combining those squares that contain 1s (or 0s). This process is called looping. Variables that are the same for all the cells of the loop must appear in the final expression.

The number of 1s (or 0s) that can be looped together should be a integer power of 2. For instantce, $2^0 = 1$, $2^1 = 2$, $2^2 = 4$, $2^3 = 8$, etc can be looped. The condition is that the looped items should mutually neighbour each other. Figure **??** uses a 4-variable K-map to show how various cells neighbor each other (using blue arrows).

We illustrate how groups of 2, 4, and 8 mutually neighboring 1s (or 0s) can be looped.

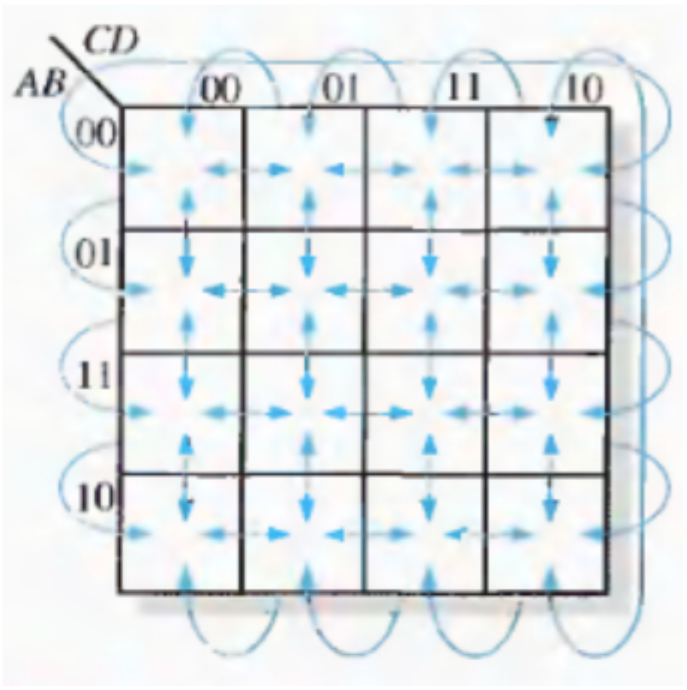


Fig. 5.4: How cells neighbor each other in a K-map. Blue arrows are used to show adjacent cells that neighbor each other.

Groups of 2 (pairs)

Two 1s (or 0s) can be looped together if they are horizontally or vertically adjacent. When two 1s (or 0s) are looped, 1 $(= \log_2 2)$ variable is eliminated.

Two 1s next to each other diagonally are not adjacent.

See figure for illustration.

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

$x = B\bar{C}$

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

$x = \bar{B}\bar{C}$

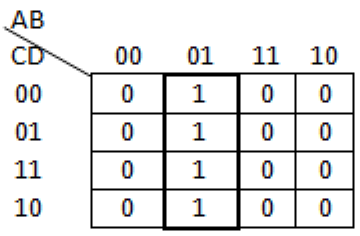
C \ AB	00	01	11	10
0	0	1	0	0
1	0	1	0	0

$x = \bar{A}B$

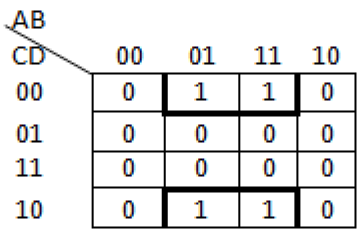
Groups of 4 (quads)

Four 1s (or 0s) can be looped together if they are horizontally or vertically adjacent (mutually) or form a square. A loop of four 1s (or 0s) eliminates 2 ($= \log_2 4$)variables.

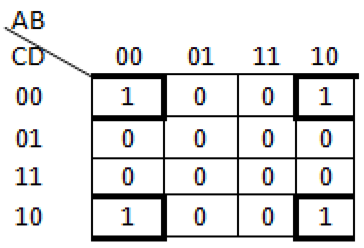
See the figure for illustration.



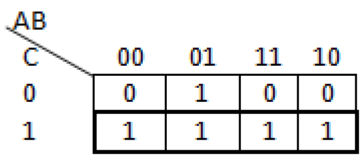
$x = A\bar{B}$



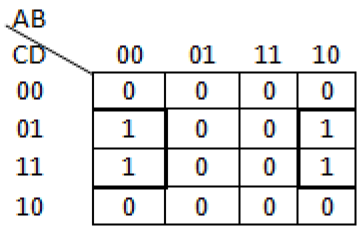
$x = B\bar{D}$



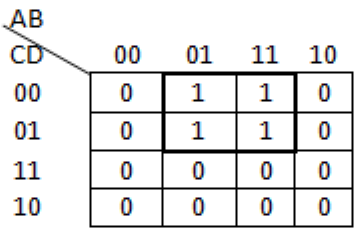
$x = \bar{B}\bar{D}$



$x = C$



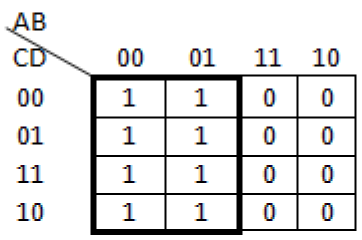
$x = \bar{B}D$



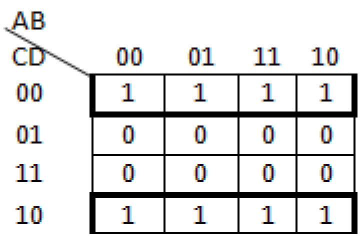
$x = B\bar{C}$

Groups of 8 (octets)

Eight 1s (or 0s) may be looped together if they are adjacent. A loop of eight 1s (or zeros) eliminates 3 ($= \log_2 8$) variables. Figure illustrates.



$x = \bar{A}$



$x = \bar{D}$

5.2.3 Minimized expression

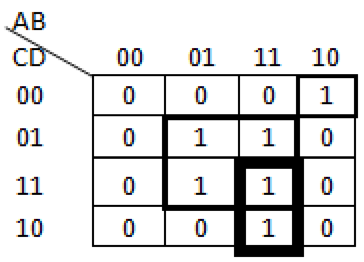
The minimized expression can be obtained in one of two ways:

1. an SOP format, or
2. a POS format.

The two formats can be obtained using the following methods.

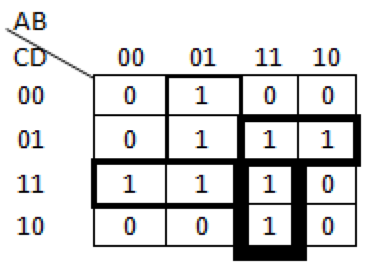
1. For an SOP expression, an OR sum of all the product terms generated by each loop of 1s is used.
2. For a POS expression, a product of all the sum terms generated by each loop of 0s is used.

A further illustration of minimization using 1s is provided in the K-maps in the figure.



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

$x = A\bar{B}\bar{C}\bar{D} + ABC + BD$



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

$x = \bar{A}B\bar{C} + A\bar{C}D + ABC + \bar{A}CD$

5.2.4 The simiplification procedure

The procedure for simplification can be carried out using the following steps. The illustration is provided using 1s (for SOP) but a similar looping process applies for 0s (for POS).

1. Construct the K-map placing 1s and 0s in their appropriate cells.
2. Identify cells with isolated 1s and loop them.
3. Identify cells with 1s that neighbor only a single other 1 and loop them to form pairs.

4. Identify cells with 1s that can be combined with *three* other 1s in only 1 way and loop them to form quads. In so doing, do not completely consume loops that have already been created.

5. Repeat the process for groups of 8, 16, 32, and so on

6. Looking at the remaining unlooped 1s, use them to create the largest permited groups (groups of 1, 2, 4, 8, 16, e.t.c) without completely consuming previously created loops. This step should create groups of 2 and larger, since isolated 1s were already exhausted in step 2.

7. Form the OR sum of all the product terms generated by each loop.

When looping 0s, step 7 changes to: *Form the product of all the sum terms generated by each loop.*. Also remember to invert the literals when looping zeros, i.e. for a variable A, 1 represents literal $\bar{A}$ and 0 represents literal A. The approach is similar to how the maxterms are formed. See the formation of maxterms in Table 5.1.

After going through the looping process and minimization, if the result you obtained can be minimized further, that indicates that your loops were not constructed appropriately. There are likely to be loops that are smaller than they should be. Examine your looping once more to optimize the looping.

5.2.5 Don't care conditions

In some design problems, some input combinations are not allowed. A code example is the BDC code, where the codes from 1010 to 1111 are not used to code decimal numbers. These combinations are called ***don't care*** because whether the outputs that result from them is a 1 or a 0 does not matter. In practice, you do not get a BCD code with combinations 1010, 1011, 1100, 1101, 1110, and 1111.

When designing using K-maps, the don't care terms can be used to make the loops for 1s (or 0s) larger, depending on the convenience. An example of a truth table with don't care conditions is given in Table 5.2. We will the table to illustrate the use of don't care terms during minimization with K-maps.

Exercise 37:

Obtain a minimized SOP expression for Table using a K-map.

Solution 37:

Table 5.2: A truth table with don't care terms.

A	B	C	D	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	X
1	0	1	1	X
1	1	0	0	X
1	1	0	1	X
1	1	1	0	X
1	1	1	1	X

The terms are arranged in a K-map, and the don't care terms are used to reduce the expression from $Y = ABCD + A\bar{B}\bar{C}$ to $Y = A + BCD$. In illustration is given in Figure 5.5.

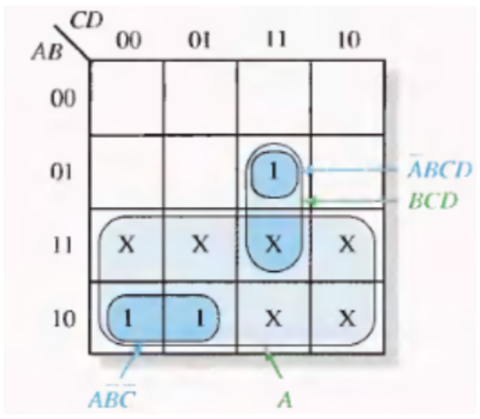


Fig. 5.5: Looping using don't care terms.

Exercise 38:

Draw logic circuits using AND-OR-NOT logic (using AND, OR, and NOT gates only) to implement the outcomes of minimization with and without don't care terms in the previous exercise.

Solution 38:

Try this by yourself.

5.2.6 Further examples of SOP and POS minimizations

Further examples of looping to obtain SOP are shown in Fig. 5.6.

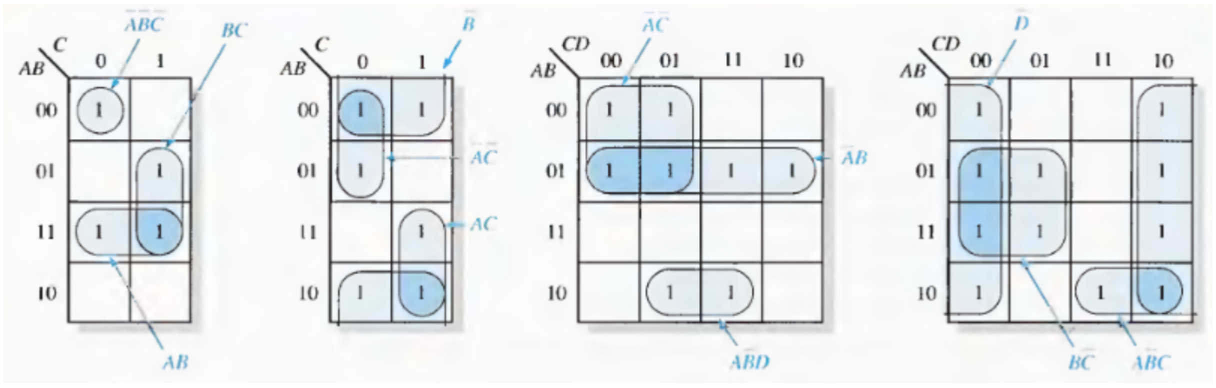


Fig. 5.6: Examples of looping minterms

Some examples of looping to obtain POS are shown in Fit. 5.7.

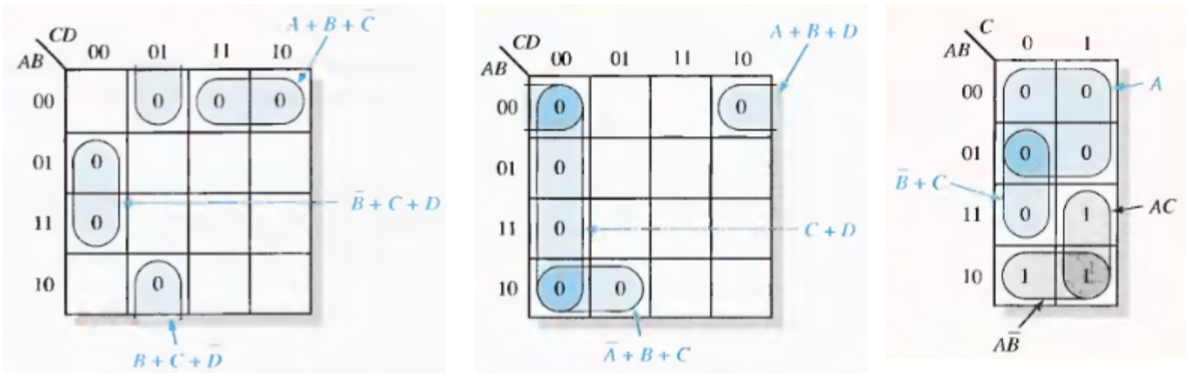


Fig. 5.7: Looping maxterms

Exercise 39:

Obtain SOP and POS minimizations from the K-maps in Fig. 5.8. The empty cells are occupied by 0s.

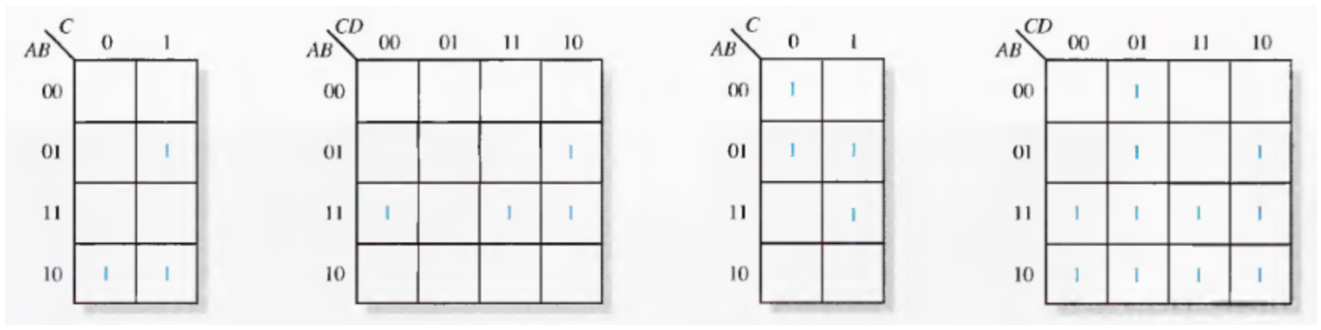


Fig. 5.8: Figure for K-map minimization exercises.

Solution 39:

5.2.7 A note on minimization

It is possible to also construct K-maps for more than 4 variables. However, the maps can become tedious to work with. In such cases, other minimization algorithms such such the *Quine-McCluskey* method can be used. All in all, minimization by manual means is only convenient when the inputs are few. For a large number of inputs, computer-based logic synthesis tools are used for fast and efficient minimization. Nonetheless, it is important for an engineer to have the mathematical background required to perform minimizations. That is the reason why solving simple logic design problems using manual minimization methods is important.