LOGIC DESIGN

Contents

Chapter One: INTRODUCTION

NUMBER SYSTEMS AND CONVERSION.

1.1 Number systems and Conversion.

1.2 Binary Arithmetic.

1.3 Binary codes.

Chapter Two: BOOLEAN ALGEBRA

2.1 Basic Gates and operations.

2.2 Boolean Expressions and Truth Tables.

2.3 Basic Theorems.

2.4 Exclusive OR and Equivalence Operations.

Chapter Three: WORD PROBLEMS

MINITERM AND MAXTERM EXPANSIONS.

3.1 Combinational Network Design Using a Truth Table.

3.2 Minterm and Maxterm Expansions.

3.3 Incomletely Specified Function.

Chapter Four : SIMPLIFICATION METHODS

4.1 Minimum Forms of Logic Functions.

4.2 Two- and Three-Variable Karnaugh Maps.

4.3 Four-Variable Karnaugh Maps.

4.4 Five- and Six-Variable Karnaugh Maps.

4.5 Quine-McCluskey method.

Chapter Five: MULTIPLE-OUTPUT NETWORKS

5.1 Multi-Output NAND and NOR Networks.

5.2 Multiplexers.

5.3 Decoders.

5.4 Read-Only Memory.

5.5 Programmable Logic Arrays.

Chapter Six: FLIP-FLOPS

6.1 The Set-Reset Flip-Flops.

6.2 The J-K Flip-Flops.

6.3 The Trigger Flip-Flops.

6.4 The Delay Flip-Flops.

6.5 Counters.

Chapter One

INTRODUCTION: NUMBER SYSTEMS AND CONVERSION.

1.1 Number systems and Conversion.

When we write decimal (base 10) numbers, we use a positional notation; each digit is multiplied by an appropriated power of 10 depending on its position in the number. For example:

$$(953.78)_{10}=9x10^{2}+5x10^{1}+3x10^{0}+7x10^{-1}+8x10^{-2}$$

Similarly, for binary (base 2) numbers, each binary digit is multiplied by the appropriate power of two:

$$(1011.11)_{2}=1x2^{3}+0x2^{2}+1x2^{1}+1x2^{2}+1x2^{-1}+1x2^{-2}$$

$$=8+0+2+1+1/2+1/4$$

$$=(11.75)_{10}$$

Note that the binary point separates the positive and negative powers of two just as the decimal point separates the positive and negative power of ten for decimal numbers.

Any positive integer R(R≥1) can be chosen as the radix or base of a number system. If the base is R then R digits (0,1,2,…,R-1) are used. For example, if R=8 , then the required digits are (0,1,2,3,4,5,6 and 7). A number

written in positional notation can be expanded in power series in R.

For example :

$N=(a_4\ a_3\ a_2\ a_1\ a_0.\ a_{-1}\ a_{-2})_R$

$= a_4xR^4+ a_3xR^3+ a_2xR^2+ a_1xR^1+ a_0xR^0+ a_{-1}xR^{-1}+ a_{-2}xR^{-2}$

where a_i is the coefficient of R^i and $0\leq a_i\leq R-1$. If the arithmetic indicated in the power series expansion is done in base 10, then the result is the decimal equivalent of N.

For example:$(147.3)_8=1x8^2+ 4x8^1+7x8^0+3\ x8^{-1}$

$=64+32+7+3/8$

$=(103.375)_{10}$

For bases grater than 10, more than 10 symbols are needed to represent the digits. In this case, letters are usually used to represent digits greater than 9. For example in hexadecimal (base 16), *A* represent $(10)_{10}$, *B* represent $(11)_{10}$, *C* represent $(12)_{10}$, *D* represent $(13)_{10}$, *E* represent $(14)_{10}$, and *F* represent $(15)_{10}$.

Thus, $(A2F)_{16}=10x16^2+2x16^1+15x16^0$

$= 2560+32+15$

$=(2607)_{10}$

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

Next, we will discuss conversion of a decimal ***integer*** to base R using the division method.

Example : Convert $(53)_{10}$ to binary.

2 | 53		
2 | 26	rem.=1=a_0	
2 | 13	rem.=0=a_1	
2 | 6	rem.=1=a_2	THUS $(53)_{10}=(110101)_2$
2 | 3	rem.=0=a_3	
2 | 1	rem.=1=a_4	

0 rem.=1=a_5

Now, Conversion of a decimal ***fraction*** to base R can be done using successive multiplication by R.

Example : Convert $(.625)_{10}$ to binary.

F=.625 F_1=.250 F_1=.500

X 2	X 2	X 2	THUS $(.625)_{10}=(.101)_2$
1.250	0.500	1.000	
(a_{-1}=1)	(a_{-2}=0)	(a_{-3}=1)	

This process does not always terminate, but if it does not terminate the result is a repeating fraction.

Example : Convert $(.7)_{10}$ to binary.

.7

2

(1).4

2

(0).8

2

(1).6

2

(1).2

2

(0).4 ← process start repeating here since .4 was previously

2 obtained above

(0).8 $(0.7)_{10}=(0.1\ \overline{0110}\ \overline{0110}\ \overline{0110}\ \ldots)_2$

Conversion between two bases other than decimal can be done directly by using the procedures given; however, the arithmetic operations would have to be carried out using a base other than ten. It is generally easier to convert to decimal first and then the decimal to the new base.

Example : Convert $(231.3)_4$ to base 7.

$(231.3)_4=2x16+3x4+1x1+3/4$

$=(45.75)_{10}$

7 | 45		.75
7 6	rem. 3	7
0	rem. 6	(5).25
		7
		(1).75
		7
THUS $(45.75)_{10}=(63.5151..)_7$		(5).25

Conversion from binary to octal (and conversely) can be done by inspection since each octal digit corresponds to exactly 3 binary digits (bits). Starting at the binary point, the bits are divided into groups of 3 and each group is replaced by an octal digit:

$(11010111110.0011)_2 = \overline{011}\ \overline{010}\ \overline{111}\ \overline{110} . \overline{001}\ \overline{100}$

3 2 7 6 1 4

$= (3276.14)_8$

Similarly, binary to hexadecimal conversion is accomplished by dividing the binary number into groups of 4 bits and replacing each group by a hexadecimal digit:

$(1001101.010111)_2 = \overline{0100}\ \overline{1101} . \overline{0101}\ \overline{1100}$

4 D 5 C

$= (4D.5C)_{16}$

1.2 Binary Arithmetic.

Arithmetic operation in digital systems are usually done in binary because design of logic networks to perform binary arithmetic is much easier than for decimal. Binary arithmetic is carried out in much the same manner as decimal, except the addition and multiplication tables are much simpler.

The addition table for binary numbers is

0 + 0 = 0

0 + 1 = 1

1 + 0 = 1

1 + 1 = 0 and carry 1 to the next column

Example : add $(13)_{10}$ and $(11)_{10}$ in binary.

$(13)_{10}$= 1101

$(11)_{10}$= 1011

11000=$(24)_{10}$

The subtraction table for binary numbers is

0 - 0 = 0

0 - 1 = 1 and borrow 1 from the next column

1 - 0 = 1

1 - 1 = 0

Borrowing 1 from a column is equivalent to subtracting 1 from that column.

Example : sub $(11)_{10}$ from $(13)_{10}$ in binary.

1 ← (Indicate a borrow from the 3rd column)

$(13)_{10}$= 1101

$(11)_{10}$= ~~1011~~

0010=$(2)_{10}$

Example : sub $(10000)_2$ and $(11)_2$ in binary.

1111 ← (Indicate a borrow)

10000

~~11~~

1101

Note how the borrow propagate from column to column in the second example.

The multiplication table for binary numbers is

0 x 0 = 0

0 x 1 = 0

1 x 0 = 0

1 x 1 = 1

Example : multiply $(13)_{10}$ from $(11)_{10}$ in binary.

$(13)_{10}$= 1101

$(11)_{10}$= ~~1011~~

1101

1101

0000

~~1101~~

10001111 = $(143)_{10}$

The division table for binary numbers is

$0 \div 0 =$ undefined

$0 \div 1 = 0$

$1 \div 0 =$ infinite

$1 \div 1 = 1$

Example : divide $(72)_{10}$ by $(12)_{10}$ in binary.

$(72)_{10} = (1001000)_2$

$(12)_{10} = (1100)_2$

110

1100 1001000

~~1100~~

01100

~~1100~~

000000

1.3 Binary codes.

Although most large computers work internally with binary numbers, the input-output equipment generally uses decimal numbers. Since most logic circuits only accept two-valued signals, the decimal numbers must be coded in terms of binary signals.

Decimal digit	8-4-2-1 code (BCD)	6-3-1-1 code	Excess-3 code	Gray code
0	0000	0000	0011	0000
1	0001	0001	0100	0001
2	0010	0011	0101	0011
3	0011	0100	0110	0010
4	0100	0101	0111	0110
5	0101	0111	1000	0111
6	0110	1000	1001	0101
7	0111	1001	1010	0100
8	1000	1011	1011	1100
9	1001	1100	1100	1101

In the simplest form of binary code, referred to as binary-coded-decimal (BCD), each decimal digit is replaced by its binary equivalent.

For *example*, 937.25 is represented by

9 3 7 . 2 5

$(1001\ \ 0011\ \ 0111\ .\ \ 0010\ \ 0101)_{BCD}$

Note that the result is quite different than that obtained by converting the number as a whole into binary.

Table shown above shows several possible sets of binary codes for the ten decimal digits and many other possibilities exist.

The excess-3 code is obtained from the 8-4-2-1 code by adding 3 (0011) to each of the code. To translate a decimal number to excess-3 coded form, each decimal digit is replaced by its corresponding code. Thus 937 expressed in excess-3 code is 1100 0110 1010.

The table shows one example of a Gray code. A Gray code has the property that the codes for successive decimal digits differ in exactly one bit. For example, the cods for 6 and 7 differ only in the first bit.

Many application of computers require processing of data which contains numbers, letters and other symbols such as punctuation marks. In order to transmit such alphanumeric data to or from a computer, or store it internally in a computer, each symbol must be represented by a binary code. One common alphanumeric code is the ASCII code (American Standard Code for

Information Interchange). This is a 7-bit code, so 2^7 (128) different code combinations are available to represent letters, numbers and other symbols. Table below shows a portion of ASCII code; the word "Start" is represented in ASCII code as follows :

1010011 1110100 1100001 1110010 1110100

S t a r t

Table ASCII code

Character	ASCII code $A_6A_5A_4A_3A_2A_1A_0$	Character	ASCII code $A_6A_5A_4A_3A_2A_1A_0$	Character	ASCII code $A_6A_5A_4A_3A_2A_1A_0$
Space	0 1 0 0 0 0 0	**@**	1 0 0 0 0 0 0	**‘**	1 1 0 0 0 0 0
!	0 1 0 0 0 0 1	**A**	1 0 0 0 0 0 1	**a**	1 1 0 0 0 0 1
“	0 1 0 0 0 1 0	**B**	1 0 0 0 0 1 0	**b**	1 1 0 0 0 1 0
#	0 1 0 0 0 1 1	**C**	1 0 0 0 0 1 1	**c**	1 1 0 0 0 1 1
$	0 1 0 0 1 0 0	**:**	: :	**:**	: :
:	: :	**Z**	1 0 1 1 0 1 0	**z**	1 1 1 1 0 1 0
0	0 1 1 0 0 0 0	**[**	1 0 1 1 0 1 1	**{**	1 1 1 1 0 1 1

1	0 1 1 0 0 0 1	\	1 0 1 1 1 0 0	|	1 1 1 1 1 0 0
2	0 1 1 0 0 1 0	]	1 0 1 1 1 0 1	}	1 1 1 1 1 0 1
:	: :	:	: :	:	: :
?	0 1 1 1 1 1 1	-	1 0 1 1 1 1 1	**delete**	1 1 1 1 1 1 1

Problems :

1.1 Convert to octal then to binary:

a. $(757.25)_{10}$ b. $(123.17)_{10}$ c. $(356.89)_{10}$ d. $(1063.5)_{10}$

1.2 Convert to octal and then to decimal:

a.$(10111011.1)_2$ b.$(1101101.011)_2$ c.$(10000011.11)_2$

1.3 Add, subtract and multiply in binary:

a. 1111 and 1011 b. 1001001and 111010 c. 110100 and 11011

1.4 Convert to base 5: $(165.2)_7$

(do all of the arithmetic in decimal)

1.5 a. Convert to hexadecimal: $(701.12)_{10}$

b. Convert to decimal: $(ABC.D)_{16}$

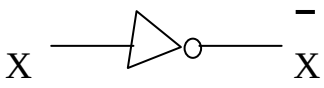
c. Device a scheme for converting hexadecimal directly to base 4 and convert the previous hexadecimal number to base 4.

Chapter Two

BOOLEAN ALGEBRA

2.1 Basic Gates and operations.

The basic mathematics needed for the study of logic design of digital system is Boolean algebra. The basic operation of Boolean algebra are AND, OR and complement (or inversion). The complement of 0 is 1, and the complement of 1 is 0. Symbolically, we write X to denote the complementation of X. we represent an inverter by:



Where the circle at the output indicates inversion. Complementation is sometimes referred to as the Not operation since *X*=1 if *X* is *not* equal to 0

The AND operation can be defined as follows:

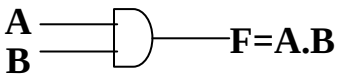
0 . 0=0 , 0 . 1=0 , 1 . 0=0, 1 . 1=1

where "." Denotes AND. If we write the Boolean expression F=A.B, then given values of A and B, we can determine F from the following table:

A	B	F=A.B
0	0	0

0	1	0
1	0	0
1	1	1

A logic gate which performs the AND operation is represented by:



The "." symbol is frequently omitted in a Boolean expression, and we will usually write AB instead of A.B , the AND operation is also referred to as logical (or Boolean) multiplication.

The OR operation can be defined as follows:

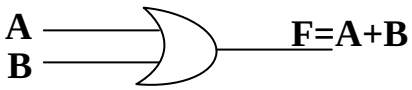
$0 + 0=0, \quad 0 + 1=1, \quad 1 + 0=1, \quad 1 + 1=1$

where "+" denotes OR. If we write F=A+B, then given the values of A and B we can determine F from the following table:

A	B	F=A.B
0	0	0
0	1	1

1	0	1
1	1	1

A logic gate which performs the OR operation is represented by:



The OR operation also referred to as logical (or Boolean) addition.

2.2 Truth Tables

The operation of the AND, OR, and NOT logic operators can be formally described by using a **truth table** as shown in Figure 2.5. A truth table is a two-dimensional array where there is one column for each input and one column for each output (a circuit may have more than one output). Since we are dealing with binary values, each input can be either a 0 or a 1. We simply enumerate all possible combinations of 0's and 1's for all the inputs.

Usually, we want to write these input values in the normal binary counting order. With two inputs, there are 2^2 combinations giving us the four rows in the table. The values in the output column are determined from applying the corresponding input values to the functional operator.

For the AND truth table in Figure 2.1 (a), $F = 1$ only when x and y are both 1, otherwise, $F = 0$. For the OR truth table (b), $F = 1$ when either x or y or both is a 1, otherwise $F = 0$. For the NOT truth table, the output F is just the inverted value of the input x.

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

(a)

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

(b)

x	F
0	1
1	0

(c)

Figure 2.1. Truth tables for the three basic logical operators: (a) AND; (b) OR; (c) NOT.

Using a truth table is one method to formally describe the operation of a circuit or function. The truth table for any given logic expression (no matter how complex it is) can always be derived. Examples on the use of truth tables to describe digital circuits are given in the following sections. Another method to formally describe the operation of a circuit is by using Boolean expressions or Boolean functions.

2.3 Boolean Algebra

George Boole, in 1854, developed a system of mathematical logic, which we now call *Boolean algebra*. Based on Boole's idea, Claude Shannon, in 1938, showed that circuits built with binary switches can easily be described using Boolean algebra. The abstraction from switches being on and off to the use of Boolean algebra is as follows.

Let $B = \{0, 1\}$ be the Boolean algebra whose elements are one of the two values, 0 and 1. We define the operations AND (**.**), OR (+), and NOT ($'$) for the elements of B by the axioms in Figure 2.2 (a). These axioms are simply the definitions for the AND, OR, and NOT operators.

2.4 Duality Principle

Notice in Figure 2.2 that we have listed the axioms and theorems in pairs. Specifically, we define the **dual** of a logic expression as one that is obtained by changing all + operators with . operators, and vice versa, and by changing all 0's with 1's, and vice versa. For example, the dual of the logic expression $(x.y'.z) + (x.y.z') + (y.z)$ is $(x'+y+z') .(x'+y'+z) .(y'+z')$

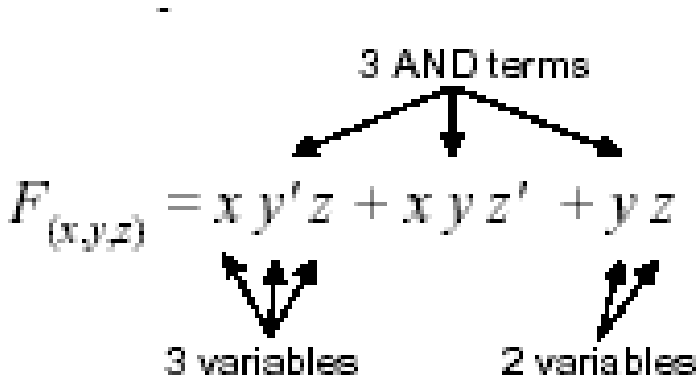
The **duality principle** states that if a Boolean expression is true , then its

dual is also true. Be careful in that it does not say that a Boolean expression is equivalent to its dual. For example , Theorem 5a in Figure 2.2 says that $x \cdot 0 = 0$ is true , thus by the duality principle , its dual , $x + 1 = 1$ is also true. However, $x \cdot 0 = 0$ is not equal to $x + 1 = 1$, since 0 is definitely not equal to 1.We will see in later sections that the duality principle is used extensively in digital logic design. Whereas an expression might be complex to implement, its dual might be simpler. In this case, implementing its dual and converting it back to the original expression will result in a smaller circuit.

2.5 Boolean Function and the Inverse

As we have seen, any digital circuit can be described by a logical expression, also known as a *Boolean function*.

Any Boolean functions can be formed from binary variables and the Boolean operators . , +, and ′ (for AND, OR, and NOT respectively). For example, the following Boolean function uses the three variables or literals *x*, *y*, and *z*. It has three **AND terms** (also referred to as **product terms**), and these AND terms are ORed (summed) together. The first two AND terms contain all three variables each, while the last AND term contains only two variables. By definition, an AND (or product) term is either a single variable, or two or more



variables ANDed together. Quite often, we refer to functions that are in this

format as a **sum-of-products** or **OR-of-ANDs**.

The value of a function evaluates to either a 0 or a 1 depending on the given set of values for the variables. For example, the function above evaluates to a 1 when any one of the three AND terms evaluate to a 1, since 1 OR *x* is 1. The first AND term, *xy'z*, equals to a 1 if

$x = 1, y = 0,$ and $z = 1$

because if we substitute these values for *x*, *y*, and *z* into the first AND term *xy'z*, we get a 1 Similarly, the second AND term, *xyz'*, equals to 1 if

$x = 1, y = 1,$ and $z = 0$

The last AND term, *yz*, has only two variables. What this means is that the value of this term is not dependent on the missing variable *x*. In other words *x* can be either a 0 or a 1, but as long as $y = 1$ and $z = 1$, this term will equal to a 1.

Thus, we can summarize by saying that *F* evaluates to a 1 if

$x = 1, y = 0,$ and $z = 1$

or $x = 1, y = 1,$ and $z = 0$

or $x = 0, y = 1,$ and $z = 1$

or $x = 1, y = 1,$ and $z = 1$ Otherwise, *F* evaluates to a 0.

It is often more convenient to summarize the above verbal description of a function with a truth table as shown in Figure 2.7 under the column labeled *F*. Notice that the four rows in the table where *F* = 1 match the four cases in the description above.

x	*y*	*z*	*F*	*F'*
0	0	0	0	1
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	0

Figure 2.3. Truth table for the function $F = xy'z + xyz' + yz$

The inverse of a function, denoted by *F'*, can be easily obtained from the truth table for *F* by simply changing all the 0's to 1's and 1's to 0's as shown in the truth table in Figure 2.7 under the column labeled *F'*. Therefore, we can write the Boolean function for *F'* in the sum-of-products format, where the AND terms are obtained from those rows where *F'* = 1. Thus, we get

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

To deduce *F'* algebraically from *F* requires the use of DeMorgan's Theorem (Theorem 15a) twice . For example , using the same function $F = xy'z + xyz' + yz$ we obtain *F'* as follows

$F' = (xy'z + xyz' + yz)'$

$= (xy'z)' \cdot (xyz')' \cdot (yz)'$

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

There are three things to notice about this equation for *F'*. First, *F'* is just the dual of *F* as defined in Section 2.5.2. Second, instead of being in a sum-of-products format, it is in a **product-of-sums** (**and-of-ors**) format where three OR terms (also referred to as sum terms) are ANDed together. Third, from the same original function *F*, we obtained two different equations for *F'*.

From the truth table, we obtained

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

and from applying DeMorgan's theorem to *F*, we obtained

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

Hence, we must conclude that these two expressions, where one is in the sum-of-products format, and the other is in the product-of-sums format, are equivalent. In general, all functions can be expressed in either the sum-of products or product-of-sums format.

Thus , we should also be able to express the same function $F = xy'z + xyz' + yz$ in the product-of-sums format. We can derive it using one of two methods. For method one, we can start with *F'* and apply DeMorgan's Theorem to it just like how we obtained *F'* from *F*.

$$F = F''$$

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

$= (x'y'z')' \cdot (x'y'z)' \cdot (x'yz')' \cdot (xy'z')'$

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

For the second method, we start with the original *F* and convert it to the product-of-sums format using the Boolean theorems.

$F = xy'z + xyz' + yz$

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

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

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

$(z+y).(z+y)$ step 2

$=(x+y) .(x+z).(x+y+z) .(x+y+z') .(x+y'+z) .(z+y)$ step 3

$=(x+y+zz').(x+yy'+z).(x+y+z).(x+y+z').(x+y'+z).(xx'+y+z)$ step 4

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

$(x+y+z) .(x'+y+z)$ step 5

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

In the first step, we apply Theorem 12b (Distributive) to get every possible combination of sum terms. For example, the first sum term $(x+x+y)$ is obtained from getting the first *x* from *xy'z*, the second *x* from *xyz'*, and the *y* from *yz*. The second sum term $(x+x+z)$ is obtained from getting the first *x* from *xy'z*, the second *x* from *xyz'*, and the *z* from *yz*. This is repeated for all combinations. In

this step, the sum terms, such as $(x+z'+z)$, where it contains variables of the form $v + v'$ can be eliminated since $v + v' = 1$, and $1 .x = x$.

In the second and third steps, duplicate variables and terms are eliminated. For example, the term $(x+x+y)$ is equal to just $(x+y+y)$, which is just $(x+y)$. The term $(x+z'+z)$ is equal to $(x+1)$, which is equal to just 1, and therefore, can be eliminated completely from the expression.

In the fourth step, every sum term with a missing variable will have that variable added back in by using Theorems 6b and 9a, which says that $x + 0 = x$ and $yy' = 0$, therefore, $x + yy' = x$.

Step five uses the Distributive Theorem, and the resulting duplicate terms are again eliminated to give us the format that we want.

Functions that are in the product-of-sums format (such as the one shown below) are more difficult to deduce when they evaluate to a 1. For example, using

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

F' evaluates to a 1 when all three terms evaluate to a 1. For the first term to evaluate to a 1, x can be 0, or y can be 1, or z can be 0. For the second term to evaluate to a 1, x can be 0, or y can be 0, or z can be 1. Finally, for the last term, y can be 0, or z can be 0, or x can be either a 0 or a 1. As a result, we end up with many more combinations to consider, even though many of the combinations are duplicates.

However, it is easier to determine when a product-of-sums format expression evaluates to a 0. For example, using the same expression

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

F' evaluates to 0 when any one of the three OR terms is 0, since 0 AND x is 0; and this happens when

$x = 1$, $y = 0$, and $z = 1$ for the first OR term,

Or $x = 1$, $y = 1$, and $z = 0$ for the second OR term,

Or $y = 1$, $z = 1$, and x can be either 0 or 1 for the last or term.

Similarly, for a sum-of-products format expression, it is easy to evaluate when it is a 1, but difficult to evaluate when it is a 0.

These four conditions in which F' evaluates to a 0 match exactly those rows in the table shown in Figure 2.7 where $F' = 0$.

Therefore, we see that in general, the unique algebraic expression for any Boolean function can be specified by either (1) selecting the rows from the truth table where the function is a 1 and use the sum-of-products format, or (2) selecting the rows from the truth table where the function is a 0 and use the product-of-sums format. Whatever format we decide to use, the one thing to remember is that we are always interested in only when the function (or its inverse) is equal to a 1 .

Figure 2.8 summarizes these two formats for the function $F = xy'z + xyz' + yz$ and its inverse. Notice that the sum-of-products format for F is the dual (i.e. by applying the duality principle) of the product-of-sums format for F'. Similarly, the product-of-sums format for F is the dual of the sum-of-products format for

F'.

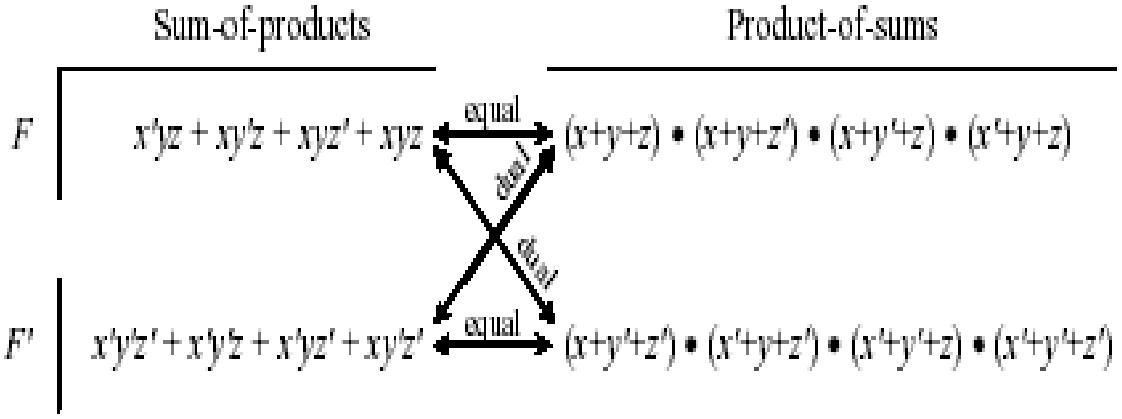


Figure 2.4. Relationships between the function *F = xy'z + xyz' + yz* and its inverse, and the sum-of-products and product-of-sums formats.

2.6 Minterms and Maxterms

As you recall, a product term is a term with either a single variable, or two or more variables ANDed together, and a sum term is a term with either a single variable, or two or more variables ORed together. To differentiate between a term that contains any number of variables with a term that contains *all* the variables used in the function, we use the words minterm and maxterm.

2.6.1 Minterms

A **minterm** is a product term that contains all the variables used in a function. For a function with *n* variables, the notation *mi* where 0< □*i* < 2*n*, is used to denote the minterm whose index *i* is the binary value of the *n* variables such

that the variable is complemented if the value assigned to it is a 0, and uncomplemented if it is a 1.

For example, for a function with three variables *x*, *y*, and *z*, the notation *M3* is used to represent the term in which the values for the variables *xyz* are 011 (for the subscript 3). Since we want to complement the variable whose value is a 0, and uncomplement it if it is a 1. Hence *M3* is for the minterm *x'yz*. Figure 2.9 (a) shows the eight minterms and their notations for $n = 3$ using the three variables *x*, *y*, and *z*.

x	y	z	Minterm	Notation
0	0	0	$x'y'z'$	m_0
0	0	1	$x'y'z$	m_1
0	1	0	$x'yz'$	m_2
0	1	1	$x'yz$	m_3
1	0	0	$xy'z'$	m_4
1	0	1	$xy'z$	m_5
1	1	0	xyz'	m_6
1	1	1	xyz	m_7

(a)

x	y	z	Maxterm	Notation
0	0	0	$x+y+z$	M_0
0	0	1	$x+y+z'$	M_1
0	1	0	$x+y'+z$	M_2
0	1	1	$x+y'+z'$	M_3
1	0	0	$x'+y+z$	M_4
1	0	1	$x'+y+z'$	M_5
1	1	0	$x'+y'+z$	M_6
1	1	1	$X'+y'+z'$	M_7

(b)

When specifying a function, we usually start with product terms that contain all the variables used in the function. In other words, we want the **sum of minterms**, and more specifically the sum of the one-minterms, that is the minterms for which the function is a 1 (as opposed to the zero-minterms, that is the minterms for which the function is a 0). We use the notation **1-minterm** to denote one-minterm, and **0-minterm** to denote zero-minterm.

Figure 2.5. (a) Minterms for three variables. (b) Maxterms for three variables.

The function from the previous section

$$F = xy'z + xyz' + yz$$

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

and repeated in the following truth table has the 1-minterms *M3*, m_5, m_6, and m_7.

x	y	z	F	F'	Minterm	Notation
0	0	0	0	1	$x'y'z'$	m_0
0	0	1	0	1	$x'y'z$	m_1
0	1	0	0	1	$x'yz'$	m_2
0	1	1	1	0	$x'yz$	m_3
1	0	0	0	1	$xy'z'$	m_4
1	0	1	1	0	$xy'z$	m_5
1	1	0	1	0	xyz'	m_6
1	1	1	1	0	xyz	m_7

Thus, a shorthand notation for the function is

$$F(x, y, z) = M3 + m_5 + m_6 + m_7$$

By just using the minterm notations, we do not know how many variables are in the original function.

Consequently, we need to explicitly specify the variables used by the function as in $F(x, y, z)$. We can further simplify the notation by using the standard algebraic symbol $\sum$ for summation. Therefore, we have

$F(x, y, z) = \sum (3, 5, 6, 7)$

These are just different ways of expressing the same function. Since a function is obtained from the sum of the 1-minterms, the inverse of the function, therefore, must be the sum of the 0-minterms. This can be easily obtained by replacing the set of indices with those that were excluded from the original set.

Example 2.5: Given the Boolean function $F(x, y, z) = y + x'z$, use Boolean algebra to convert the function to the sum-of-minterms format.

Solution:

This function has three variables. In a sum-of-minterms format, all product terms must have all variables. To do so, we need to expand each product term by ANDing it with $(v + v')$ for every missing variable v in that term. Since $(v + v') = 1$, therefore, ANDing a product term with $(v + v')$ does not change the value of the term.

$F = y + x'z$

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

expand 1st term by ANDing it with $(x+x')(z+z')$,and 2nd term with $(y+y')$

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

$= m_7 + m_6 + M3 + m_2 + m_1$

$=\sum (1, 2, 3, 6, 7)$ sum of 1-minterms

Example 2.6: Given the Boolean function $F(x, y, z) = y + x'z$, use Boolean algebra to convert the inverse of the function to the sum-of-minterms format.

Solution:

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

$= y' .(x'z)'$ use DeMorgan

$= y' .(x+z')$ use DeMorgan

$= y'x + y'z'$ use Distributive Theorem to change to SoP

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

expand 1st term by ANDing it with $(z+z')$, and 2nd term with $(x+x')$

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

$= m_5 + m_4 + m_0$

$= \sum (0, 4, 5)$ sum of 0-minterms

2.6.2 Maxterms

Analogous to a minterm, a **maxterm** is a sum term that contains all the variables used in the function. For a function with *n* variables, the notation *Mi* where $0 \leq i \leq 2n$, is used to denote the maxterm whose index *i* is the binary value of the *n* variables such that the variable is complemented if the value assigned to it is a 1, and uncomplemented if it is a 0.

For example, for a function with three variables *x*, *y*, and *z*, the notation M_3 is used to represent the term in which the values for the variables *xyz* are 011. For maxterms, we want to complement the variable whose value is a 1, and uncomplement it if it is a 0. Hence M_3 is for the maxterm $x + y' + z'$.

Figure 2.9 (b) shows the eight maxterms and their notations for $n = 3$ using the three variables *x*, *y*, and *z*. We have seen that a function can also be specified as a product of sums, or more specifically, a **product of 0- maxterms**, that is, the maxterms for which the function is a 0. Just like the minterms, we use the notation **1-maxterm** to denote one-maxterm, and **0-maxterm** to denote zero-maxterm. Thus, the function

$$F(x, y, z) = xy'z + xyz' + yz$$

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

x	y	z	F	F'	Maxterm	Notation
0	0	0	0	1	$x+y+z$	M_0
0	0	1	0	1	$x+y+z'$	M_1
0	1	0	0	1	$x+y'+z$	M_2
0	1	1	1	0	$x+y'+z'$	M_3
1	0	0	0	1	$x'+y+z$	M_4
1	0	1	1	0	$x'+y+z'$	M_5
1	1	0	1	0	$x'+y'+z$	M_6
1	1	1	1	0	$x'+y'+z'$	M_7

which is shown in the following table can be specified as the product of the 0-maxterms M_0, M_1, M_2, and M_4. The shorthand notation for the function is

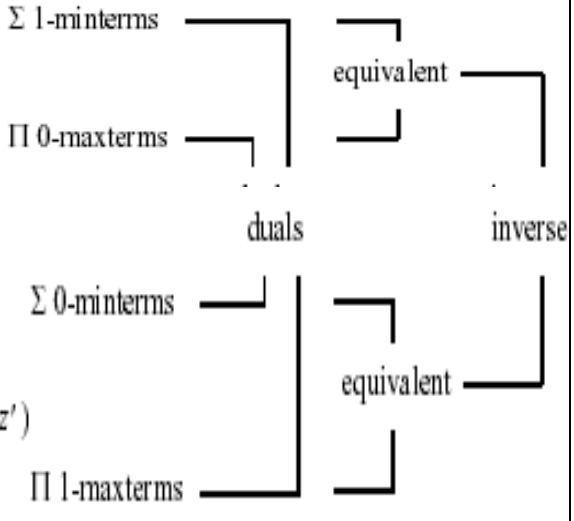
$F(x, y, z) = M_0 .M_1 .M_2 .M_4$ Again, by using the standard algebraic symbol for

$$F(x, y, z) = x'yz + xy'z + xyz' + xyz$$
$$= m_3 + m_5 + m_6 + m_7$$
$$= \Sigma(3, 5, 6, 7)$$

$$= (x+y+z) \cdot (x+y+z') \cdot (x+y'+z) \cdot (x'+y+z)$$
$$= M_0 \cdot M_1 \cdot M_2 \cdot M_4$$
$$= \Pi(0, 1, 2, 4)$$

$$F'(x, y, z) = x'y'z' + x'y'z + x'yz' + xy'z'$$
$$= m_0 + m_1 + m_2 + m_4$$
$$= \Sigma(0, 1, 2, 4)$$

$$= (x+y'+z') \cdot (x'+y+z') \cdot (x'+y'+z) \cdot (x'+y'+z')$$
$$= M_3 \cdot M_5 \cdot M_6 \cdot M_7$$
$$= \Pi(3, 5, 6, 7)$$



Notice that it is always the Σ of minterms and Π of maxterms; you never have Σ of maxterms or Π of minterms.

product, the notation is further simplified to F(x, y, z) =∑ (0, 1, 2, 4) The

following summarizes these relationships for the functionF = xy'z + xyz' + yz and its inverse. Comparing these equations with those in Figure 2.8, we see that they are identical.

Example 2.7: Given the Boolean function $F(x, y, z) = y + x'z$, use Boolean algebra to convert the function to the product-of-maxterms format.

Solution:

To change a sum term to a maxterm, we expand each term by ORing it with (vv') for every missing variable v in that term. Since $(vv') = 0$, therefore, ORing a sum term with (vv') does not change the value of the term.

$F = y + x'z$

$= y + (x'z)$

$=(y+x')(y+z)$ use Distributive Theorem to change to product of sum format

$=(y+x'+zz')(y+z+xx')$ expand1st term by Oring it with zz',and 2nd with xx'

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

$= M_4. M_5 . M_0 = \prod (0, 4, 5)$ product of 0-maxterms

Example 2.8: Given the Boolean function $F(x, y, z) = y + x'z$, use Boolean algebra to convert the inverse of the function to the product-of-maxterms format.

Solution:

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

$= y' .(x' z)'$ use DeMorgan

$= y' .(x+z')$ use DeMorgan

$=(y'+xx'+zz').(x+z'+yy')$ expand 1st term by ORing with $xx' +zz'$, and 2nd term with yy'

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

$= M_2 .M_3 .M_6 .M_7 .M_1$

$=\prod (1, 2, 3, 6, 7)$ product of 1-maxterms

2.7 Canonical, Standard, and non-Standard Forms

Any Boolean function that is expressed as a sum of minterms, or as a product of maxterms is said to be in its **canonical form**. For example, the following two expressions are in their canonical forms

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

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

As noted from the previous section, to convert a Boolean function from one canonical form to its other equivalent canonical form, simply interchange the symbols . with +, and list the index numbers that were excluded from the original form.

For example, the following two expressions are equivalent

$F1(x, y, z) = \sum (3, 5, 6, 7)$

$F2(x, y, z) = \prod (0, 1, 2, 4)$

To convert a Boolean function from one canonical form to its dual (inverse), simply interchange the symbols $\sum$ with $\prod$, and list the same index numbers from the original form. For example, the following two expressions are duals

$F1(x, y, z) = \sum (3, 5, 6, 7)$

$F2(x, y, z) = \prod (3, 5, 6, 7)$

A Boolean function is said to be in a **standard form** if a sum-of-products expression or a product-of-sums expression has at least one term that is not a minterm or a maxterms respectively. In other words, at least one term in the expression is missing at least one variable. For example, the following expression is in a standard form because the last term is missing the variable *x*.

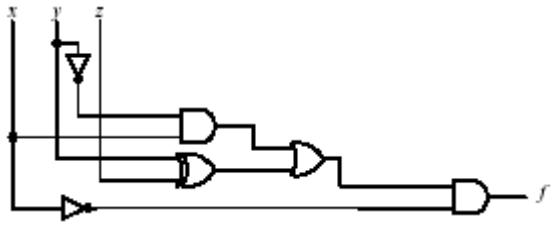
$$F = xy'z + xyz' + yz$$

Sometimes, common variables in a standard form expression can be factored out. The resulting expression is no longer in a sum-of-products or product-of-sums format. These expressions are in a **non-standard form**. For example, starting with the previous expression, if we factor out the common variable *x* from the first two terms, we get the following expression, which is in a non-standard form.

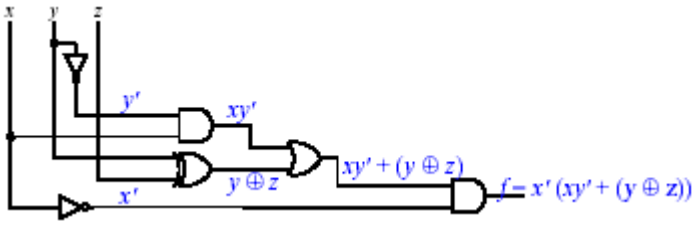
$$F = x(y'z + yz') + yz$$

Example 3.1

Consider the combinational circuit below,



Starting from the primary inputs x, y, and z, we annotate the outputs of each



logic gate with the resulting logical expression. Hence, we obtain the annotated circuit below

The output of the circuit is the final function f = x' (xy' + (y (+) z)).

3.2 Minimization of Combinational Circuits

When constructing digital circuits, in addition to obtaining a functionally correct circuit, we like to optimize it in terms of circuit size, speed, and power consumption. In this section, we will focus on the reduction of circuit size.

Usually, by reducing the circuit size, we will also improve on speed and power consumption. We have seen in the previous sections that any combinational circuit can be represented using a Boolean function. The size of the circuit is directly proportional to the size or complexity of the functional expression. In fact, it is a one-to-one correspondence between the functional expression and the circuit size. In Section 2.5.1, we saw how we can transform a Boolean

function to another equivalent function by using the Boolean algebra theorems. If the resulting function is simpler than the original, then we want to implement the circuit based on the simpler function, since that will give us a smaller circuit size.

Using Boolean algebra to transform a function to one that is simpler is not an easy task, especially for the computer. There is no formula that says which is the next theorem to use. Luckily, there are easier methods for reducing Boolean functions. The Karnaugh map method is an easy way for reducing an equation manually, and is discussed in Section 3.4.1. The Quine-McCluskey or tabulation method for reducing an equation is ideal for programming the computer, and is discussed in Section 3.4.3.

3.4.1 Karnaugh Maps

To minimize a Boolean equation in the sum-of-products form, we need to reduce the number of product terms by applying the combining Boolean Theorem (Theorem 14) from Section 2.5.1. In so doing, we will also have

reduced the number of variables used in the product terms. For example, given the following 3-variable function $F = xy'z' + xyz'$ we can reduce it to

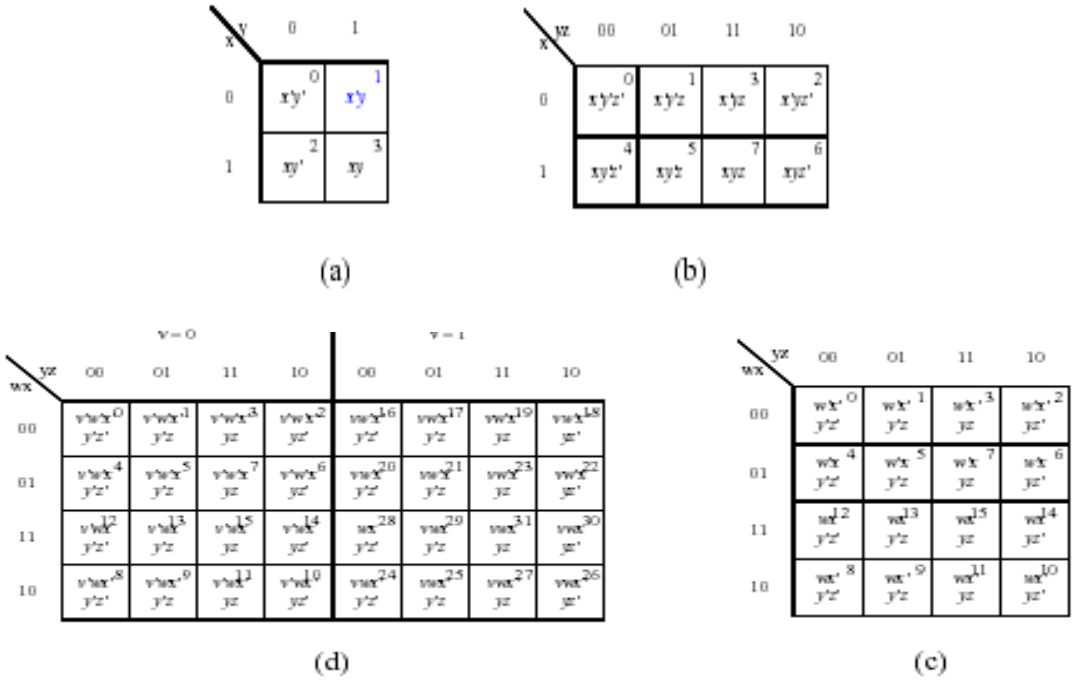
$F = xz' (y' + y) = xz' 1 = xz'$

In other words, two product terms that differ by only one variable whose value is a 0 (primed) in one term, and a 1 (unprimed) in the other term, can be combined together to form just one term with that variable omitted as shown in the example above. Thus, we have reduced the number of product terms and the resulting product term has one less variable. By reducing the number of product terms, we reduce the number of OR operators required, and by reducing the number of variables in a product term, we reduce the number of AND operators

required. Looking at a logic function's truth table, it is sometimes difficult to see how the product terms can be combined and minimized. A Karnaugh map, or K-map for short, provides a simple and straightforward procedure for combining these product terms. A K-map is just a graphical representation of a logic function's truth table where the minterms are grouped in such a way that it allows one to easily see which of the minterms can be combined. It is a 2-dimensional array of squares, each of which represents one minterm in the Boolean function. Thus, the map for an *n*-variable function is an array with 2*n* squares.

Figure 3.5 shows the K-maps for functions with 2, 3, 4, and 5 variables. Notice the labeling of the columns and rows are such that any two adjacent columns or rows differ in only one bit change. This condition is required because we want minterms in adjacent squares to differ in the value of only one variable or one bit, and so these minterms can be combined together. This is why the labeling for the third and fourth columns, and the third and fourth rows are always interchanged. When we read K-maps, we need to visualize it as such that the two end columns or rows wrap around so that the first and last columns, and the first and last rows are really adjacent to each other because they differ in only one bit also.

In Figure 3.5, the K-map squares are annotated with its minterm and its minterm number for easy reference only. For example, in Figure 3.5 (a) for a 2-variable K-map, the entry in the first row and second column is labeled *x'y*, and annotated with the number 1. This is because the first row is when the variable *x* is a 0, and the second column is when the variable *y* is a 1. Since for minterms, we need to prime a variable whose value is a 0, and not prime it if its value is a 1, therefore, this entry represents the minterm *x'y*, which is minterm



number 1.

Figure 3.5. Karnaugh maps for:

(a) 2 variables; (b) 3 variables; (c) 4 variables; (d) 5 variables.

Be careful that if we label the rows and columns differently, the minterms and the minterm numbers will be in different locations. When we are actually using K-maps to minimize an equation, we will not write these in the squares. Instead, we will be putting 0's and 1's in the squares.

Given a Boolean function, we set the value for each K-map square to either a 0 or a 1 depending on whether that minterm for the function is a 0-minterm or a 1-minterm respectively. However, since we are only interested in the 1-minterms for a function, the 0's are sometimes not written in the 0-minterm squares.