Discussion and Experimental Procedure

You saw in Assignment 1 that binary numbers comprise a collection of binary digits or bits, each of which can be either 0 or 1 and that these binary values can be represented in hardware by various means such as toggle switch positions and lamp states.

What we are now about to study are the operations that may be performed on one, two or more of these 'binary variables' which, although they were introduced to you as a means of representing binary numbers, actually have an independent existence and may be used to represent quite different things.

For instance 1 and 0 can represent any of the pairs of opposites shown in the following table.

'1'	'0'
TRUE	FALSE
IN	OUT
UP	DOWN
YES	NO
WET	DRY

In general binary variables can represent any pair of concepts in which the existence of one implies the non-existence of the other. Notice carefully the idea of 'opposite-ness' or 'NOT-ness'

TRUE is NOT FALSE
UP is NOT DOWN
1 is NOT 0
0 is NOT 1
etc., etc.

Practical 2.1

Plug in inverter (7404) module into any convenient position on the CK353 deck, aligning it vertically with the guide lines provided and connect it up to a switch output and a lamp input as shown in fig 2.1. Switch on the power and operate the toggle switch.

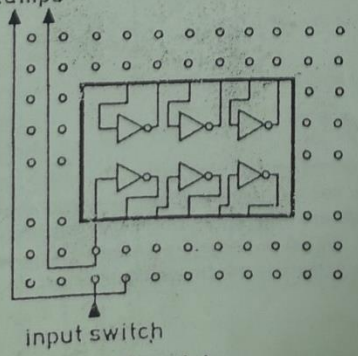


Fig. 2.1

Observe the lamp indications in relation to the switch position and complete the table:-

INPUT	OUTPUT
0 (down)	
1 (up)	

The inverter is performing a NOT operation on the input and the simple table representing it is called a 'truth table'. The ideas of true and false have a particularly close association historically with the development of algebraic methods of dealing with binary variables, the earliest application being the study of formal logic. This is why we refer to computer circuits as logic elements and why we use the term 'truth table'.

The algebra that we shall use to describe the various operations we are going to study is called 'Boolean Algebra' after its originator and in it, just as in ordinary algebra, different variables are denoted by letters e.g A, B, C etc or X, Y, Z. The difference is that whereas in ordinary algebra A could be any value at all, in Boolean algebra it can only be either 0 or 1.

$$A = 0 \quad \text{or} \quad A = 1$$

The operation of NOT that we have just studied can be applied to a single variable like this:-

NOT A

This is called the 'complement' or 'negation' of A and it is usually symbolized by a bar over the variable and said-- 'A bar'.

$$\text{NOT } A = \overline{A}$$

We could now write the truth table for NOT in a more general fashion thus

A	$\overline{A}$
0	1
1	0

● *Q2.1 If $\overline{A}$ is the same as NOT A, what do you think $\overline{\overline{A}}$ is?*

Now it is time to turn attention upon operations that can be applied to more than one variable and we shall start, for simplicity, with two variables, let us call them A and B.

Practical 2.2

Switch off the power to CK353, remove the 7404 module and replace it with a 7400 module, connecting it as shown in fig 2.2.

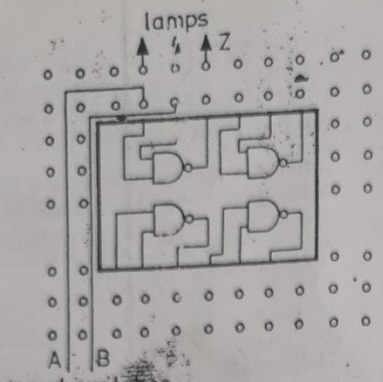


Fig. 2.2

A truth table gives the result of an operation for all possible values of the variables. We shall call the result Z. Switch on the power and successively set the input switches to the combinations shown in the table below, noting the binary value of Z (lamp on = 1, off = 0) that results.

A	B	Z
0	0	
0	1	
1	1	
1	0	

Truth table for NAND

You should have found that Z was 1 in all cases except where A and B were both 1. If we associate 1 with true and 0 with false, which is the usual convention, we can see that Z could be described as being true when it was not true that A and B were both 1.

Or, more simply

$$Z = \text{NOT (A AND B)}$$

NOT-AND is usually abbreviated to NAND so we have

$$Z = \text{A NAND B}$$

Even this abbreviated notation is not very convenient and so the notation A.B is adopted to mean A AND B.

This in turn gives us

$$Z = \text{NOT } A.B = \overline{A.B}$$

using the bar notation (say this 'AB bar'). When no ambiguity is possible the dot is usually dropped to give just AB.

Practical 2.3

On the INTIKIT plug in a 7404 module and connect Z to one of the inverters, taking its output to a fourth lamp. This will be $\overline{Z}$. Add another column for $\overline{Z}$ to the truth table you already have and complete it by setting A and B once again to all the possible values.

The operation you have now set up on INTIKIT is the AND operation because

$$\overline{Z} = \overline{\overline{AB}} = AB = \text{A AND B}$$

The AND operation is one of three fundamental operators in Boolean algebra, the other two being NOT, which we have studied already and OR, which we will meet soon.

You might ask why it is that AND was introduced in such a roundabout way via the NAND operation. The reason is that in integrated circuit logic NAND is the easiest and cheapest operation to achieve and it is therefore more widely used in practice as a logic element than any other.

The circuit symbols for NOT, AND and NAND are shown in fig 2.3.

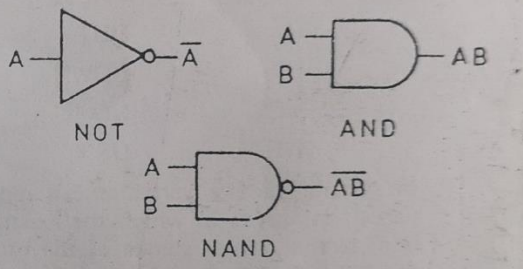


Fig. 2.3

Exercise 2.1

Draw the logic diagram for AND using a NAND followed by a NOT. Repeat for NAND using AND followed by NOT. Construct a truth table for AND.

The truth table for AND gives us some important basic identities that will later on assist in the manipulation of Boolean expressions.

0.0	= 0
0.1	= 0
1.0	= 0
1.1	= 1

Also

$$0.1 = 1.0 = 0$$

or in other words the order in which variables appear has no significance. More generally, AB and BA are the same thing. An expression like this is called a 'logical product' because of its similarity to arithmetic multiplication.

The third fundamental operation referred to earlier was that of OR. Applied to two variables A, B this says that the operation

$$Z = A \text{ OR } B$$

will have a true (1) result when either of A or B is true. The truth table is

A	B	Z
0	0	0
0	1	1
1	1	1
1	0	1

Truth table for OR

The operation is expressed symbolically as

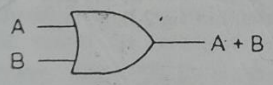
$$Z = A + B$$

the choice of the plus sign suggesting, as is true, that it has similarity to ordinary arithmetic addition. It is called a 'logical sum' for this reason.

● *Q2.2 How does the OR operation differ from ordinary addition?*

To find out put another column alongside Z in the truth table for OR, head it 'A plus B' and fill in the correct arithmetic values for all cases.

The circuit symbol for OR is shown in fig 2.4.



OR

Fig. 2.4

Take a look through the modules you have in the storage tray of INTIKIT. *Do you see this symbol anywhere?*

You probably will not since it is not a common function amongst digital integrated circuits. *How then are we to perform this operation when we wish to?*

To find the answer, carry out the following exercise.

Exercise 2.2

Complete the following truth table, filling in the columns from the left.

A	B	$\overline{A}$	B	$\overline{A}.\overline{B}$	$\overline{\overline{A}.\overline{B}}$	A + B
0	0					
0	1					
1	1					
1	0					

What do you notice about the contents of the last two columns? You should find that they are identical. Since all possible conditions are described in the table we can write

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

But the left-hand side of this equation is actually

$$\overline{A} \text{ NAND } \overline{B}$$

so that, by first negating both inputs and applying $\overline{A}$ and $\overline{B}$ to a NAND gate (gate is the usual term for a logic element), the OR operation is obtained. This is shown in fig 2.5

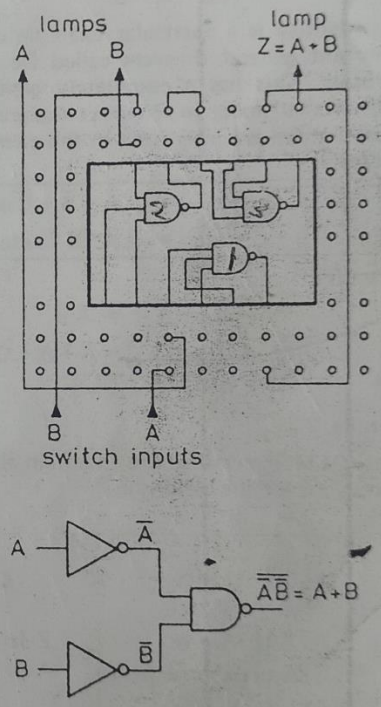


Fig. 2.5.

Practical 2.4

You will see that fig 2.5 shows a 7410 module and 3-input NAND gates have been used instead of inverters and a 2-input gate, as shown in the schematic. Before considering why this is acceptable, satisfy yourself that it does give the correct result by connecting up as shown and checking the output Z against the truth table for OR as you vary A and B.

It is a feature of the circuit used for the NAND gates that if an input is left unconnected it has the same effect as applying binary 1 to that input. So if a 3-input gate with inputs A, B, C has 2 inputs, B and C, unconnected

$$Z = \overline{A.B.C} = \overline{A.1.1}$$

But we found above that

$$0.1 = 0$$

and

$$1.1 = 1$$

Therefore $A.1.1 = A.1 = A$

so that $Z = \overline{\overline{A}}$

Similarly if only input C is unconnected

$$Z = \overline{A.B.C} = \overline{A.B.1} = \overline{AB}$$

Hence you can always leave unwanted inputs disconnected without affecting the results.

In the derivation of the OR operation above we saw that

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

This identity is a particular example of a very important general theorem called De Morgan's Theorem. This has a completely general form that we shall meet in the next Assignment but at present we are interested in the two forms as follows

$\overline{A.\ B.\ C}$ etc	=	$\overline{A} + \overline{B} + \overline{C}$ etc....
$\overline{A + B + C + \text{etc}}$	=	$\overline{A}.\ \overline{B}.\ \overline{C}.$ etc....

Examples:

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

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

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

Exercise 2.3

Apply De Morgan's theorem to obtain alternative expressions for the following

(a) $\overline{A\ \overline{B}}$

(b) $\overline{\overline{A} + \overline{B}}$

(c) $A + \overline{B}$

(d) $\overline{A + \overline{B}}$

The result of exercise (d) introduces the last operation of this Assignment. If A + B means A OR B then $\overline{A + B}$ must be NOT (A OR B) or alternatively A NOR B. The circuit of fig 2.5 showed how OR was achieved and NOR is simply the complement or negation of OR.

Exercise 2.4

Draw the schematic for NOR using three inverters and one NAND gate.

Practical 2.5

Set this up on INTIKIT using the 7410 module. You already have connected plus one inverter from a 7404 module. Construct the truth table for NOR.

Exercise 2.5

From the truth table for OR write down the four basic identities in the form of equations e.g

$$0 + 0 = 0 \text{ etc.}$$

Summary of this Assignment

You should now be familiar with negation or complementation and with the four operations AND, OR, NAND, NOR. These four are summarized in fig 2.6.

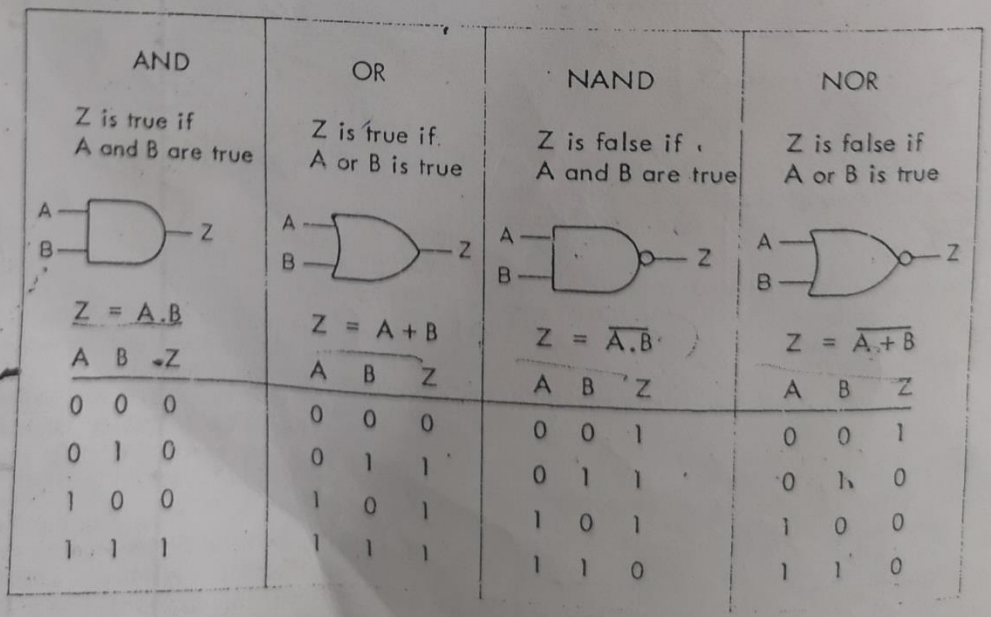


Fig. 2.6. The basic logical functions

You have also encountered the very important

De Morgan's Theorem

which relates operations to their complements.

Practical considerations & applications

As already mentioned the NAND gate is almost universally used in integrated circuit logic although all types of operation can be obtained, usually at greater cost. However, in designing logic systems, as we shall see later, the design tools and techniques are mostly better adapted to the use of the three fundamental operations of AND, OR, NOT so methods of converting the resulting expressions to the exclusive use of NAND have to be used.

Some formal methods do exist but in most practical cases these are heavy-handed and cumbersome and have little advantage over simple experience.

The fact that multi-input gates can be used for operations requiring fewer inputs than are available by leaving inputs disconnected means that when designing a large logic system it is easier to ensure that the number of modules (or packages) can be kept to a minimum. This is usually the most important economic factor nowadays. For example, suppose a circuit required three 2-input NAND gates and one NOT. This could be achieved using either

(a) 1 7410 - triple 3-input NAND

plus 1 7404 - hex inverter

or

(b) 1 7400 - quadruple 2-input NAND

Obviously (b) would be better.

Further Reading

Discussion and Experimental Procedure

You should now be aware of the fundamental Boolean operations of AND, OR, NOT and should understand the basic identities of logical sums and products:-

Logical Sums	Logical Products
0 + 0 = 0	0.0 = 0
0 + 1 = 1	0.1 = 0
1 + 0 = 1	1.0 = 0
1 + 1 = 1	1.1 = 1

Before we can go on to the design and construction of more elaborate logic circuits than have so far been met we have to learn some more theorems which will help in the manipulation of general expressions.

The method of establishing all Boolean theorems is to subject them to a truth table, setting all the variables in turn to both possible values so as to set up all possible conditions.

For example, what value has Z in the identity

$$Z = 0 + A?$$

Putting A = 0 and 1 in turn we find

If A = 0 Z = 0 + 0 = 0

If A = 1 Z = 0 + 1 = 1

Therefore Z = A and 0 + A = A.

Exercise 3.1

Using the truth table method prove the following groups of theorems

(a) $1 + A = 1$, $0.A = 0$, $1.A = ?$

(b) $A + A = A$, $A.A = A$

(c) $A + \overline{A} = ?$, $A.\overline{A} = 0$

Look now at the following expression

$$Z = AB + AC$$

This differs from anything we have met so far because it contains both sum and product operations in the same expression.

The expression above is called a 'sum of products'. In ordinary algebra we know that we could take out A as a common factor to give

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

but can this be done in Boolean algebra?

To find out we will construct a truth table for the factored and unfactored forms and compare them for identity. This table will have eight rows because three variables are involved.

A	B	C	AB	AC	AB+AC	B+C	A(B+C)
0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0
0	1	0	0	0	0	1	0
0	1	1	0	0	0	1	0
1	0	0	0	0	0	0	0
1	0	1	0	1	1	1	1
1	1	0	1	0	1	1	1
1	1	1	1	1	1	1	1

Looking at the two columns headed AB+AC and A(B+C) we see that they are identical, which proves that factoring is allowable.

In other words

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

This is all very well but how can an expression of this sort be set up using only NAND gates and inverters? Let's start by trying to realize the left hand side. We know how to form AB and AC separately because this is a simple AND operation. Fig 3.1 shows the circuits.

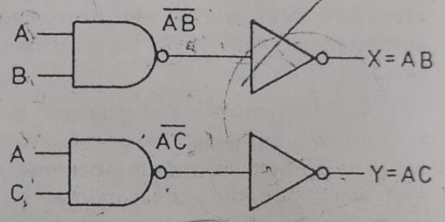


Fig. 3.1

Now to get X + Y = AB + AC we need an OR operation as in fig 3.2.

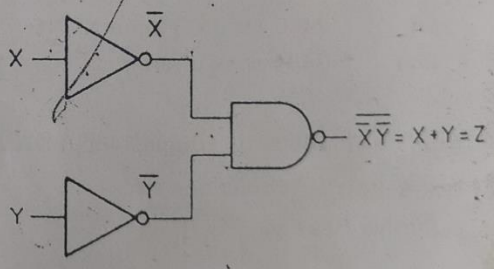


Fig. 3.2

Exercise 3.2

Put the two circuits of figs 3.1 and 3.2 together to get a logic circuit for Z = AB + AC. What do you notice about the inverters?

Practical 3.1

Connect up the circuit on INTIKIT and check out the truth table for all combinations of A,B,C. You should have had to use only one module 7400 to set up this function. If you seem to

need more look again at the inverters. Ask your instructor if you still cannot see how to use one module only.

Leave the previous circuit connected and look now at the right hand side of the identity

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

Exercise 3.3

Draw a logic circuit for X = B+C and another for Z = AX; put the two together to get a circuit for A(B+C).

Practical 3.2

Set up the result on INTIKIT, using the same input switches for A,B,C as you are already using but a different lamp to indicate the output. The two Z outputs should be identical for all settings of A,B,C.

● *Q3.1 How many modules did you use in this exercise? Is this solution more or less economical than the first?*

Now consider another expression

$$Z = A + BC$$

Unlike the previous case it is not evident from analogy with ordinary algebra that this can be factorized but in Boolean algebra it can, the result being

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

We can easily prove this by 'multiplying out' the right hand side and then applying the identities we learnt in Assignment 2

$$(A+B)\,(A+C) = AA + AC + BA + BC$$

But AA = A so that

$$AA + AC + BA = A + AC + BA = A(1 + C + B)$$

Remembering that 1 + X = 1

$$A\,(1 + C + B) = A.1 = A$$

Therefore $(A + B)\,(A + C) = A + BC$

Exercise 3.4

Draw logic circuits for both sides of this identity.

Practical 3.3

Set up your circuits for both sides as before and check that the outputs are equal for all settings of A,B,C. The left hand expression needs no more than one 7400 and one 7404. What does the right hand expression need and which is simpler?

The exercises and experiments you have just performed have demonstrated an important theorem of Boolean algebra called the 'distributive' theorem.

$$(A + B + \ldots\ldots)\,(C + D + \ldots\ldots) =$$
$$AC + AD + BC + BD + \ldots\ldots$$
$$AB + CD + \ldots\ldots = (A+C)\,(A+D)\,(B+C)\,(B+D)\,\ldots\ldots$$

They also showed that the 'sum of products' form of an expression is generally the one most readily and cheaply realized by NAND logic. In fact you should take special note of the configuration shown in fig 3.3 which provides the general 'sum of products' function.

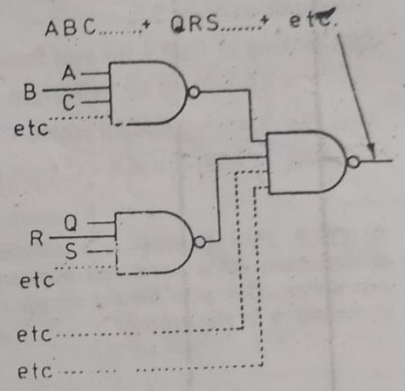
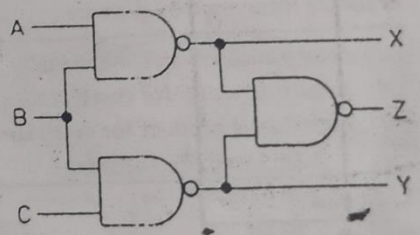


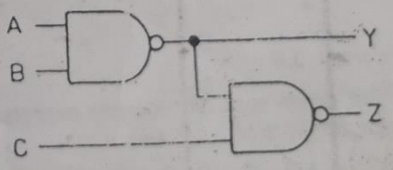
Fig. 3.3

Exercise 3.5

Write expressions for the outputs X, Y, Z in the circuits of fig 3.4 a) and (b)



(a)



(b)

Fig. 3.4

De Morgan's Theorem

In Assignment 2 we discussed De Morgan's theorem for negating a function but only as applied to simple sums like (A+B) or products like A B. We need to know how to apply it to expressions such as

$$Z = AB + \overline{C}$$

which combine sums and products. You should recognise this expression as one of the solutions to Exercise 3.5 (b) above and you should have been able to write this down directly by application of the general sum of products configuration of fig 3.3.

Still looking at fig 3.4 (b) you should have also

$$Y = \overline{AB} = \overline{A} + \overline{B} \text{ by De Morgan}$$

Now $Z = Y \text{ NAND } C = \overline{AB} \text{ NAND } C$

$= (\overline{A} + \overline{B})\,\overline{C} = A\,B + \overline{C}$ from above

● *Q3.2 This identity is a solution for the complement of a combined sum and product expression. Can you see the simple rules which convert one to the other?*

Practical 3.4

Set up logic circuits on INTIKIT for $(\overline{A} + \overline{B})\cdot C$ and for $A\,B + \overline{C}$ and display their outputs on adjacent lamps. For all settings of A, B, C you should find the outputs to be complementary. Before drawing the circuit for $(\overline{A} + \overline{B})C$ apply the distribution theorem to put it into sum of products form.

The simple rules for negation which you have probably recognised by now are

(a) negate each variable individually
(b) substitute a sum for every product
(c) substitute a product for every sum

De Morgan's Theorem

Examples

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

(b) $\overline{A + B(C + \overline{D}\,E)} = \overline{A}\,[\overline{B} + \overline{C}(D + \overline{E})]$

Exercise 3.6

Write down sum of products expressions for the complements of

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

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

simplifying the results as much as possible.

Exercise 3.7

Simplify the following

(a) $A + AB$

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

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

Hint: The usual way to a simple result is to expand the expression using the distribution theorem and then look for terms that either cancel to 0 or have common factors.

In case you had difficulty with this exercise, here is a worked solution to (b)

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

$A\,A + A\overline{B} + BA + B\overline{B} + CA + C\overline{B}$ — distribution theorem

$= A(1 + \overline{B} + B + C) + B\,\overline{B} + C\overline{B}$

$= A.1 + 0 + C\overline{B}$ (since $B\overline{B} = 0$)

$= A + C\overline{B}$

Here also is a solution to Exercise 3.6 (b)

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

$= \overline{AB + \overline{A}\,\overline{B} + C + \overline{C}} = \overline{A\,B + \overline{A}\,\overline{B} + 1}$ (since $C + \overline{C} = 1$)

$= \overline{1} = 0$

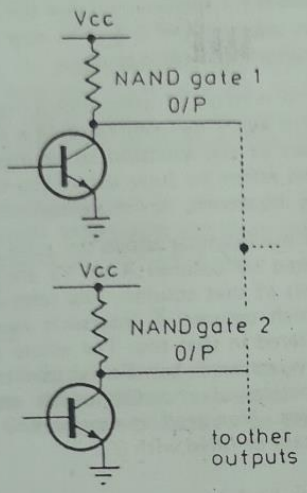
Practical considerations & applications

The sum of products form for Boolean functions is certainly the most useful for NAND logic but if the number of variables is large or if the number of product terms is large, either could exceed the number of inputs available on a standard integrated circuit module.

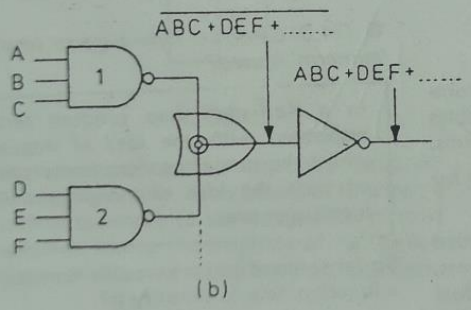
Up to eight inputs are available on some types (e.g 7430) and in other inputs expanders can be connected to increase the number of available inputs.

Yet another solution if the large number of inputs is needed at the second gating level (that is, there are many individual product terms to be summed) is to use what is called a WIRED-OR connection.

The WIRED-OR principle is shown in fig 3.5



(a)



(b)

Fig 3.5 Wired OR connection

When all inputs to a NAND gate are '1' its output is near ground since the output transistor conducts. The output is thus at '0'. If the outputs of two (or more) gates are simply connected together as in (a) the common output will be '0' when any one of the transistors conducts so the effect is to generate the function shown in (b), which also shows the symbol for a wired-OR connection. A simple negation then produces the required OR function.

A practical difficulty with the circuit of fig 3.5 (a) is that, if each gate has its own load resistor internally, all the load resistors are put in parallel by the wired-OR connection so that the current which any one transistor must conduct becomes very large. In practice such a current could not be passed and the output voltage would not go to near zero as desired. Therefore, for the wired-OR connection to be possible a type of gate not fitted with its own load resistor must be used and this is usually called 'open-collector'. A single load resistor is then fitted externally. Usually up to about ten gates can be coupled in this way.

Apart from the various 'circuit' solutions just outlined it is of course possible to provide for any number of inputs by purely logical means. Fig 3.6 shows how eight product terms can be dealt with by two four-input NAND gates.

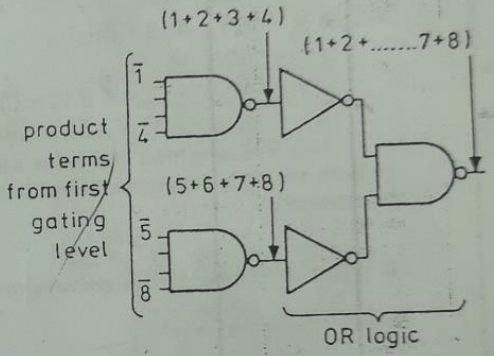


Fig. 3.6

To generate an eight-variable product term using two four-input NAND gates the logic of fig 3.7 could be used.

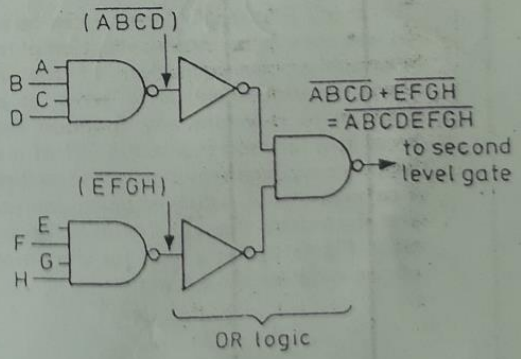


Fig. 3.7

Further Reading

Discussion and Experimental Procedure

If we look at a Boolean expression such as

$$Z = A + BC$$

one thing that might be noted about it is that neither term contains all of the three variables A,B,C. Can we rewrite it in a form such that all terms do contain all variables?

Yes, this is possible by the application of some basic theorems, but in reverse. For instance

$$BC = BC.1 = BC(A + \bar{A})$$
$$= ABC + \bar{A}BC$$
$$\text{Also } A = A.1 = A(B + \bar{B})$$
$$= AB + A\bar{B}$$
$$= AB(C + \bar{C}) + A\bar{B}(C + \bar{C})$$
$$= ABC + AB\bar{C} + A\bar{B}C + A\bar{B}\bar{C}$$

Noting that the term ABC appears in both the above expressions and thus need not be repeated.

● *Q4.1 Which basic theorem says this?*

We can therefore write

$$Z = ABC + \bar{A}BC + AB\bar{C} + A\bar{B}C + A\bar{B}\bar{C}$$

Now all terms contain all variables—in this form they are called 'minterms' and for three variables there can be up to $2^3 = 8$ of such minterms.

● *Q4.2 How many minterms can there be for (a) 2, (b) 4, (c) 5 variables?*

Since any Boolean expression can be expanded as was done above into a collection of minterms, a graphical representation of the individual terms should enable us to arrive at a unique diagram to represent any function we choose since any function is just the OR of a selection of minterms from amongst those available. The Karnaugh map is such a graphical representation and it contains one cell for each possible minterm. Fig 4.1 shows a map for the simplest case of two variables and a function.

$$F = A + \bar{A}B$$

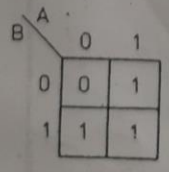


Fig 4.1

The map has 4 cells as you would expect and the columns and rows are labelled with 0's and 1's to represent the possible states of each variable. Thus column A = 0 represents $\bar{A}$ whilst row B = 1 represents B and the cell which is both in column A = 0 and in row B = 1 represents minterm $\bar{A}B$, and so on.

The method of using such a map is to enter a '1' into every cell representing a minterm which is part of the function to be mapped. We can do this either by fully expanding the function into its minterms, or more intuitively, as follows.

In the example above the term A will be represented by column A so 1's are entered in both cells of that column. The term $\bar{A}B$ is the cell in which column $\bar{A}$ intersects row B so a '1' is entered in that too. The whole area of the map now covered by 1's represents the complete function as the OR of its separate product terms. Non-used minterm cells are either left vacant or filled with 0's.

Exercise 4.1

Expand $F = A + \bar{A}B$ into its minterms and verify that the same map is produced.

● *Q4.3 Can you now suggest why the word 'minterm' is used?*

> In a Karnaugh map product terms are associated with the idea of intersections whilst the sum of product terms is associated with the idea of merging small areas into bigger ones.

So far so good but how can a Karnaugh map of a function help to simplify it?

The answer to this lies in the grouping of adjacent cells into pairs (or 4's or 8's in larger maps) so that each pair can be described by the simplest possible combination of variables (i.e with as few variables as possible). Fig 4.2 shows the application of this idea to the map of fig 4.1.

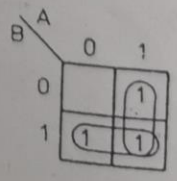


Fig 4.2

The two cells linked horizontally represent B whilse those linked vertically represent A, the OR of these two representing the whole function.

Thus F = A + B

Of course, in this example, you could have taken this easy step from (A + $\bar{A}$B) to (A + B) without the aid of a map but in more involved examples the simplification would not always be so obvious.

Notice that the cell AB in the above example was used twice in drawing the loops but this is permissible because AB = AB + AB.

Exercise 4.2

In the map of fig 4.3 a function has been plotted. Write down the algebraic expression for this function. Can any simplification be possible in this case?

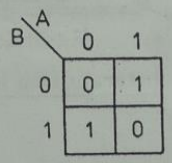


Fig. 4.3

Let us now study a map for three variables, containing eight cells. It is essential to adopt a methodical approach to the designation of rows and columns and to the mapping of functions— the three variable map of fig 4.4 illustrates both.

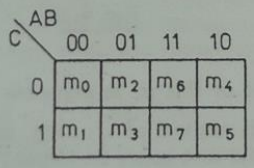


Fig. 4.4

The columns representing the various combinations of A, B are labelled so that only one variable at a time changes as we proceed from left to right. This ensures the greatest ease in the construction of loops when simplifying a function.

The labelling of separate cells in this map has been done in accordance with a natural binary progression as this provides an easy way of plotting a function when it is given in the form of a sum of minterms.

m_0 represents ABC = 000
m_1 " " = 001
m_6 " " = 110
m_7 " " = 111
and so on.

What kind of simplifying loops can be drawn on this map? Fig 4.5(a), (b) and (c) show all the possible loops of size 2, 4 and 8 which represent simple product terms.

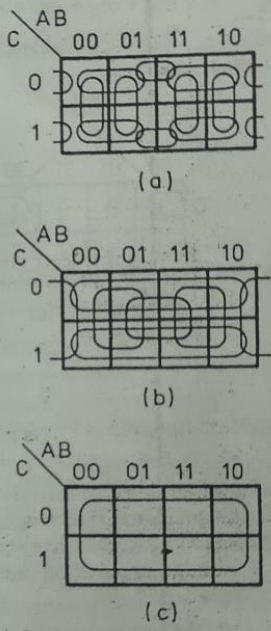


Fig. 4.5

In the next diagram fig 4.6, some only of the above loops have been drawn and labelled with their algebraic expressions.

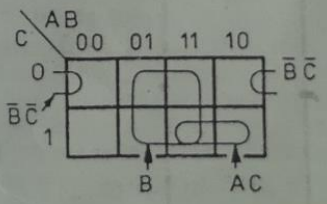


Fig. 4.6

Notice especially the loop $\bar{B}\bar{C}$, which is in two separate parts. For the purposes of simplification the map has to be regarded as being continuous from the right-hand edge to the left-hand edge.

Exercise 4.3

Redraw the maps of fig 4.5(a) and (b) on a larger scale and label all the loops in the manner of fig 4.6.

Q4.4 What is special about the loop of fig 4.5(c)?

Now let us apply the three-variable map to the simplification of some functions

(a) $F = A\overline{B}C + \overline{A}\overline{B}\overline{C} + ABC + A\overline{B}\overline{C}$

This is a sum of minterms

$$F = m_5 + m_0 + m_7 + m_4$$

and the map, with loops added is as in fig 4.7.

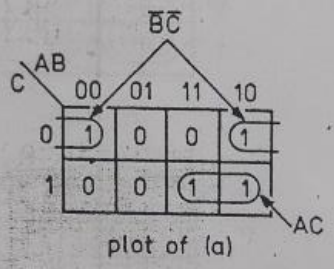


Fig. 4.7

The simplified function is $F = AC + \overline{B}\overline{C}$

(b) Take another function $F = A + BC + \overline{A}B\overline{C}$, which can be mapped intuitively by filling 1's in all cells covered by A (last two columns on the right) then adding any extra covered by BC (just m_3) and finally adding minterm m_2 for $\overline{A}B\overline{C}$. The result is fig 4.8.

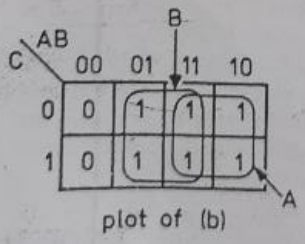


Fig. 4.8

The simplified function is $F = A + B$.

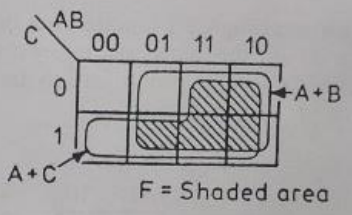
(c) $F = (A + B)(A + C)$

How are we to plot this since it is not in sum of products form? There are three ways—first by expanding it algebraically until it is a sum of products, not necessarily minterms.

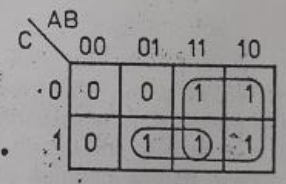
$$F = A + AC + AB + BC$$
$$= A + BC$$

This method rather nullifies the point of the map as an aid to simplification because the algebra must be carried out anyway before plotting the map and in this case led directly to the simplest form. However in more complex examples it is a reasonable method.

The second way is to draw loops on the map for (A+B) and (A+C) separately and then fill in with 1's all cells that are included in the intersection (overlap) of the loops thus drawn. Finally the normal simplifying loops are drawn and the function written down. Fig 4.9 shows these two steps.



Intersecting loops



Simplifying loops

Fig. 4.9

The result of this is, as expected $F = A + BC$.

The third method of plotting, which is sometimes useful is to apply De Morgan's Theorem to find the complement of the function and use the result, which will be in sum of products form, to plot the 0's into the map instead of the 1's.

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

Exercise 4.4

Plot the 0's from this expression and confirm that they occupy the part of the map not containing 1's in fig 4.9 above.

CD \ AB	00	01	11	10
00	m_0	m_4	m_{12}	m_8
01	m_1	m_5	m_{13}	m_9
11	m_3	m_7	m_{15}	m_{11}
10	m_2	m_6	m_{14}	m_{10}

(a) 4 Variables

DE \ ABC	000	001	011	010	110	111	101	100
00	m_0	m_4	m_{12}	m_8	m_{24}	m_{28}	m_{20}	m_{16}
01	m_1	m_5	m_{13}	m_9	m_{25}	m_{29}	m_{21}	m_{17}
11	m_3	m_7	m_{15}	m_{11}	m_{27}	m_{31}	m_{23}	m_{19}
10	m_2	m_6	m_{14}	m_{10}	m_{26}	m_{30}	m_{22}	m_{18}

(b) 5 Variables

Fig. 4.10

Maps for more than three variables

Fig 4.10 shows layouts for 4 and 5 variable maps with the minterm numbers inserted on the same principle explained above for the 3 variable map.

Satisfy yourself of the correctness of the minterm numbering in these two maps.

In 4 and 5 variable maps the number of possible loops of 2, 4, 8 etc minterms which can be drawn is quite large and a diagram showing them all would be confusing, but by now you have probably grasped the basic idea of adjacency between cells or groups of cells whose expressions differ in one variable only.

For example in fig 4.10(a) above the two minterms m_4 and m_{12} are horizontally adjacent and the loop enclosing them represents the function.

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

The two terms differed only in A, which disappears from the description of the loop. Similarly minterms m_6 and m_{14} are described by $B\,C\,\bar{D}$. Now if the four minterms m_4, m_{12}, m_6, m_{14} are enclosed by a loop the resulting expression for that loop is

$$B\,\bar{C}\,\bar{D} + B\,C\,\bar{D}$$
$$= B\,\bar{D}$$

That is, C is not needed in the description. In this case the adjacency of the two loops of 2 is across the top and bottom boundaries, which must therefore be assumed adjacent.

Exercise 4.5

Trace through a similar argument in each case to show that the following groups of minterms form simplifying loops.

(a) 4-variable (0, 2, 8, 10)
(b) 4-variable (0, 1, 2, 3, 8, 9, 10, 11)
(c) 5-variable (13, 15, 29, 31)
(d) 5-variable (0, 2, 8, 10, 16, 18, 24, 26)

Redundant States

It frequently occurs in a practical application of logic design that one or both of the following conditions arise.

(a) Certain combinations of the input variables can never occur. An obvious case of this sort is in the set of binary variables representing the ten states of a decade counter, in which only those binary numbers from 0000 (zero) through 1001 (nine) are used. The other six combinations can never occur.

Such input states as cannot occur are often called 'can't happen' (or CH for short) conditions.

(b) Sometimes the output of a logic circuit will be immaterial for certain of the input combinations, or in other words it will not matter what output is produced by these combinations. Examples of this are more difficult to find but they do occur. One we shall meet with in Assignment 10 concerns the logic for control of a JK flip-flop. We often call the input states in such cases 'don't care' (DC for short) conditions.

So far as the design of combinational logic circuits is concerned CH and DC conditions can be treated as the same thing (this is not true for

sequential logic, but that need not concern us at this stage). They are both examples of redundancy and can often help towards a simpler solution than would otherwise be possible.

What we do is to enter X in the Karnaugh map for any input combination which is either DC or CH. Then, when looking for simplifying loops, we can choose to make any given X a '0' or a '1', whichever gives the simplest solution.

Here is an example:

A binary-decimal counter counts in natural binary code from zero to nine repeating. An output is required when the count is two, three, seven or eight but not otherwise.

The counter states from ten to fifteen will never occur so we have the following requirements for the various minterms

$m_0, m_1, m_4, m_5, m_6, m_9 = 0$

$m_2, m_3, m_7, m_8 = 1$

$m_{10}, m_{11}, m_{12}, m_{13}, m_{14}, m_{15} = X$

These are mapped in fig 4.11.

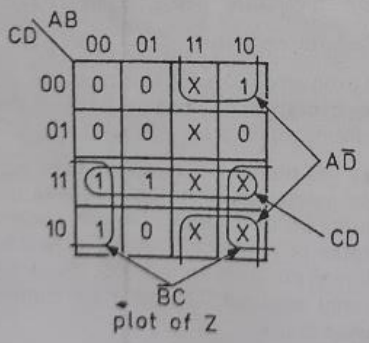


Fig. 4.11

The loops drawn in fig 4.11 make full use of the redundant (X) conditions to eliminate variables, giving

$$\text{Output } Z = A\bar{D} + \bar{B}C + CD$$

The logic diagram is in fig 4.12 and would need 1–7400 plus 1–7410 (triple 3-input NAND) using spare gates as inverters.

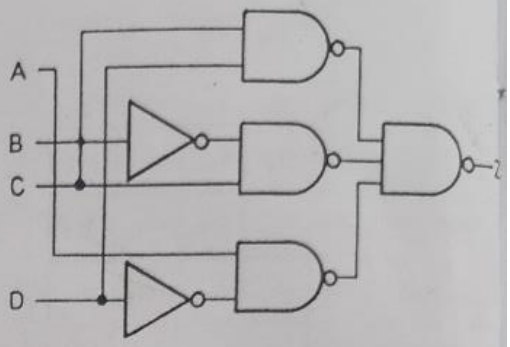


Fig. 4.12

Exercise 4.6

Write down the simplest expression for Z in the above example which could have been obtained without the use of X conditions. How many more logic modules would it have needed?

Exercise 4.7

A circuit with three inputs A, B, C must have a '1' output for minterms m_4 and m_7. Minterms m_0 and m_6 are CH conditions. Map the function and write down the simplest expression for the output Z.

In the last exercise you should have found that there were two possible solutions, both apparently simple, namely

$$Z = AB + \bar{B}\bar{C}$$

and

$$Z = AB + A\bar{C}$$

If your solution was

$$Z = AB + \bar{B}\bar{C} + A\bar{C}$$

you did not notice that you were including minterm m_4 twice unnecessarily. It frequently happens that alternative solutions exist for the same problem and in very complex cases it may even be difficult to ensure that you are not covering a minterm twice. There are formal tabular procedures for dealing with this difficulty but the problems you are likely to encounter will rarely need them. If you are especially interested your instructor will give you an idea of how to ensure that you only end up with minimal solutions.

Take another look at the two minimal solutions to exercise 4.7

$$Z = AB + \overline{B\bar{C}}$$

$$Z = AB + \overline{A\bar{C}}$$

Are they really equally simple?

To find out, let us draw their logic diagrams in fig 4.13.

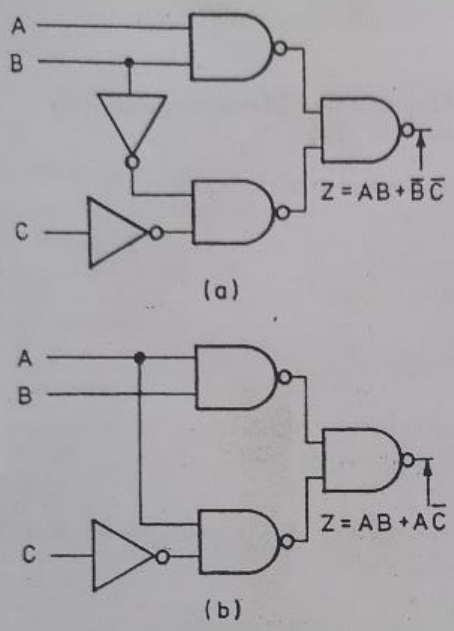


Fig.4.13

Solution (b) needs one only 7400 module but solution (a) needs one 7400 plus an extra inverter, assuming that only A, B, C are available as inputs.

This tells you that formal methods of logic simplification such as algebraic manipulation and Karnaugh Maps are only an aid to design. In the end you must use your technical judgment and commonsense to arrive at a final circuit which will cost the least.

Practical considerations & applications

Karnaugh maps for up to five or six variables offer a useful and, with practice in their application, a quick method of arriving at a first approximation to the cheapest solution to a logic design problem. After that experience and judgment must be applied to achieve the most practical configuration.

Above six variables the number of possible loops becomes very large and the advantage of a graphical representation is lost. Then tabular methods, possibly supported by computer processing, becomes necessary.

Probably the biggest single virtue of Karnaugh maps, even for relatively simple functions, is their ability to show clearly how redundant states (CH and DC conditions) can be used to best advantage.

The most important area of application you are likely to encounter is in the design of logic circuits for the excitation of inputs to flip-flops, particularly the JK type which you will meet in Assignment 6.

Further Reading