The equation for the 3-input XOR gate is derived as follows

$$
\begin{aligned}
& x \oplus y \oplus z \\
&= (x \oplus y) \oplus z \\
&= (x'y + xy') \oplus z \\
&= (x'y + xy')z' + (x'y + xy')'z \\
&= x'yz' + xy'z' + (x'y)'(xy')'z \\
&= x'yz' + xy'z' + (x+y')(x'+y)z \\
&= x'yz' + xy'z' + \cancel{xx'z} + xyz + x'y'z + \cancel{y'yz} \\
&= x'y'z + x'yz' + xy'z' + xyz
\end{aligned}
$$

The last four product terms in the above derivation are the four 1-minterms in the 3-input XOR truth table. For 3 or more inputs, the XOR gate has a value of 1when there is an odd number of 1's in the inputs, otherwise, it is a 0.

Notice also that the truth tables for the 3-input XOR and XNOR gates are identical. It turns out that for an even number of inputs, XOR is the inverse of XNOR, but for an odd number of inputs, XOR is equal to XNOR.

All these gates can be interconnected together to form large complex circuits which we call **networks**. These networks can be described graphically using **circuit diagrams**, with Boolean expressions or with truth tables.

3.2 Describing Logic Circuits Algebraically

Any logic circuit, no matter how complex, may be completely described using the Boolean operations previously defined, because of the OR gate, AND gate, and NOT circuit are the basic building blocks of digital systems. For example consider the circuit shown in Figure 1.3(c). The circuit has three inputs, A, B, and C, and a single output, *x***.** Utilizing the Boolean expression for each gate, we can easily determine the expression for the output.

The expression for the AND gate output is written $A \cdot B$. This AND output is connected as an input to the OR gate along with C, another input. The OR gate operates on its inputs such that its output is the OR sum of the inputs. Thus, we can express the OR output as $x = A \cdot B + C$**.** (This final expression can also be written as $x = C + A \cdot B$, since it does not matter which term of the OR sum is written first.)

Occasionally, there may be confusion as to which operation in an expression is performed first. The expression $A \cdot B + C$ can be interpreted in two different ways: (1) $A \cdot B$ is ORed with *C***,** or (2) *A* is ANDed with the term $B + C$. To avoid this confusion, it will be understood that if an expression contains both AND and OR operations, the AND operations are performed first, unless there are parentheses in the expression, in which case the operation inside the parentheses is to be performed first. This is the same rule in ordinary algebra to determine the order of operations.

To illustrate further, consider the circuit in Figure 1.3(d) . The expression for the OR gate output is simply $A + B$. This output serves as an input to the AND gate along with another input, *C*. Thus, we express the output of the AND gate as $x = (A + B) \cdot C$. Note the use of parentheses here to indicate that A and B are ORed first, before their sum is ANDed with C. Without parentheses it would be interpreted incorrectly, since $A + B \cdot C$ means *A* is Ored with the product $B \cdot C$.

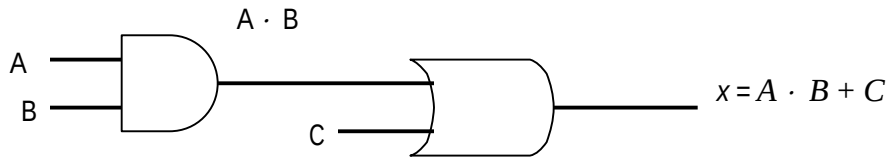


Figure 1.3(c): Logic Circuit with its Boolean expression

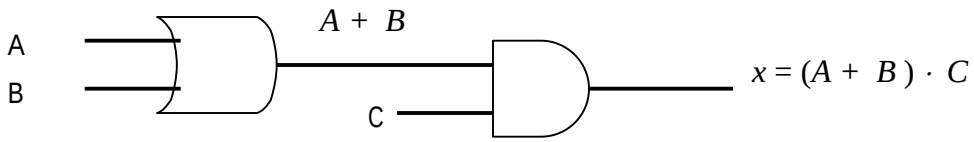


Figure 1.3(d): Logic Circuit whose expression requires parentheses

3.2.1 Circuits containing INVERTERS

Whenever an INVERTER is present in a logic-circuit diagram, its output expression is simply equal to the input expression with a bar over it. Figure 1.3(e) shows examples using INVERTERS. In Figure 1.3(e), the input A is fed to an OR gate together with B, so the OR output is equal to $A' + B$. The INVERTER output is fed to an OR gate together with B, so the OR output is equal to $A' + B$. Note that the bar is only over the A, indicating that A is first inverted and then ORed with B.

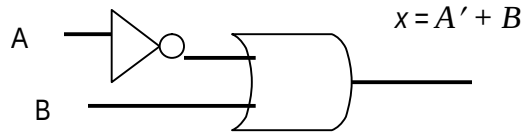


Figure 1.3(e): Circuit using INVERTERS

3.3 Constructing Circuits from Boolean expressions

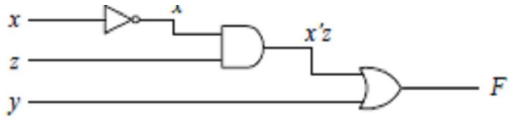
If the operation of a circuit is defined by a Boolean expression, a logic-circuit diagram can be implemented directly from that expression.

Any logic circuit, no matter how complex, may be completely described using the Boolean operations previously defined , because the OR gate, AND gate, and NOT circuit are the basic building blocks of digital system. For example, consider the circuit in Example 1.3(a). This circuit has 3 inputs x, z, y and a single output , F. Utilizing the Boolean expression for each gate, we can easily determine the expression for the output. The next example illustrates the construction of logic circuits from an expression.

Example 1.3(a): Draw the circuit diagram for the equation

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

In the equation, we need to first invert *x*, and then AND it with *z*. Finally, we need to OR *y* with the output of the AND. The resulting circuit is shown below. For easy reference, the internal nodes in the circuit are annotated with the two intermediate values *x'* and *x'z*.

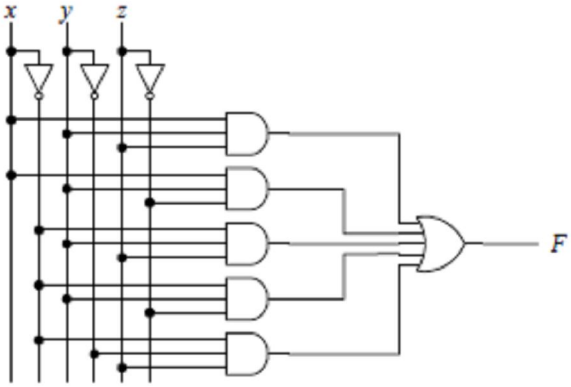


Example 1.3(b): Draw the circuit diagram for the equation:

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

The equation consists of five AND terms that are ORed together. Each AND term requires three inputs for the three variables. Hence, the circuit shown below has five 3-input AND gates, whose outputs are connected to a 5-input OR gate. The inputs to the AND gates come directly from the three variables *x, y,* and *z,* or their inverted values.

Notice that in the equation, there are six inverted variables. However, in the circuit, we do not need six inverters, rather, only three inverters are used; one for each variable.



4.0 Conclusion

A logic gate is an electronic circuit/device which makes the logical decisions. There are three basic logic gates each of which performs a basic logic function, they are called NOT, AND and OR. All other logic functions can ultimately be derived from combinations of these three. The NAND and NOR gates are called universal gates. The exclusive-OR gate is another logic gate which can be constructed using AND, OR and NOT gate. For each of the three basic logic gates a summary is given including the *logic symbol*, the corresponding *truth table* and the *Boolean expression*.

Self Assessment Exercise

Implement a circuit having the output expression Z = (AB)′C using only a NOR gate and an INVERTER.

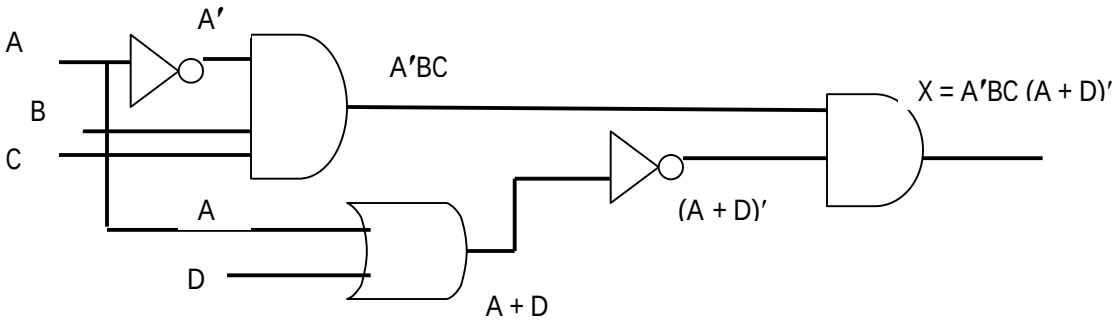
5.0 Summary

In this unit, you learnt about:

- How to construct the truth tables for the AND, NAND, OR and NOR gates.
- Implementation of logical operators with Logic gates
- The construction of Circuit diagrams

6.0 Tutor Marked Assignment

1) In the figure below, change each AND gate to an OR gate, and change the OR gate to an AND gate. Then write the expression for output *x*.



2) Draw a logic circuit for the function F = (A + B)(B + C)(A + C), using NAND gates only.

3) Draw the circuit diagram for the expression $x = AB + B'C$

4) Implement a circuit having the output expression $Z = A' + B' + C$ using a NAND gate and an INVERTER.

7.0 Further Reading and Other Resources

1. Ronald J. Tocci (1988). "Digital Systems: Priciples and Applications", 4th Edition Prentice-Hall International edition.
2. http://en.wikipedia.org/wiki/Logic_gate
3. http://www.discovercircuits.com/D/digital.htm
4. http://www.encyclopedia.com/doc/1G1-168332407.html
5. http://www.logiccircuit.org/

MODULE 1 - Basic Logic Operators and Logic Expressions
UNIT 4 - Combinatorial Circuit

Contents **Pages**

1.0 Introduction

In the last unit, we studied the operation of all the basic logic gates and we used Boolean algebra to describe and analyze circuits that were made up of combinations of logic gates. These circuits can be classified as combinatorial logic circuits because, at any time, the logic level at the output depends on the combination of logic levels present at the inputs. A combinatorial circuit has no memory characteristics and so its output depends only on the current value of its inputs. In this unit, you will learn more on combinatorial logic circuits and a description of the operation of the circuits.

2.0 Objectives

Upon completion of this unit, you will be able to:

- derive a Boolean function from combinatorial circuits.
- Synthesizing Combinatorial Circuits from truth table.
- Minimize combinatorial circuits.

3.0 Combinational Circuits

The digital system consists of two types of circuits, namely

(*i*) Combinational circuits and

(*ii*) Sequential circuits

A combinational circuit consists of logic gates, where outputs are at any instant and are determined only by the present combination of inputs without regard to previous inputs or previous state of outputs.

3.1 Analysis of Combinational Circuits

Digital circuits, regardless of whether they are part of the control unit or the data path, are classified as either one of two types: combinational or sequential. **Combinational circuits** are the class of digital circuits where the outputs of the circuit are dependent only on the current inputs. In other words, a combinational circuit is able to produce an output simply from knowing what the current input values are. **Sequential circuits**, on the other hand, are circuits whose outputs are dependent on not only the current inputs, but also on all of the past inputs. Therefore, in order for a sequential circuit to produce an output, it must know the current input and all past inputs. Because of their dependency on past inputs, sequential circuits must contain memory elements in order to remember the history of past input values. Combinational circuits do not need to know the history of past inputs, and therefore, do not require any memory elements. A "large" digital circuit may contain both combinational circuits and sequential circuits. However, regardless of whether it is a combinational circuit or a sequential circuit, it is nevertheless a digital circuit, and so they use the same basic building blocks – the AND, OR, and NOT gates. What makes them different is in the way the gates are connected.

Very often, we are given a digital logic circuit, and we would like to know the operation of the circuit. The analysis of combinational circuits is the process in which we are given a combinational circuit, and we want to derive a precise description of the operation of the circuit. In general, a combinational circuit can be described precisely either with a truth table or with a Boolean function.

3.1.1 Using a Truth Table

For example, given the combinational circuit of Figure 1.4(a), we want to derive the truth table that describes the circuit. We create the truth table by first listing all of the inputs found in the circuit, one input per column, followed by all of the outputs found in the circuit. Hence, we start with a table with four columns: three columns (*x*, *y*, *z*) for the inputs, and one column (*f*) for the output, as shown in (a) of Table 1.4(a).

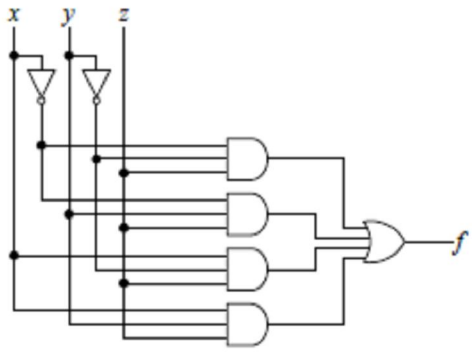


Figure 1.4(a): Combinational circuit truth table

In deriving the truth table for the sample circuit in Figure 1.4(a), the following steps are taken:

Table 1.4(a): (a) listing the input and output columns; (b) enumerating all possible combinations of the three input values;

x	*y*	*z*	*f*

(a)

x	*y*	*z*	*f*
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

(b)

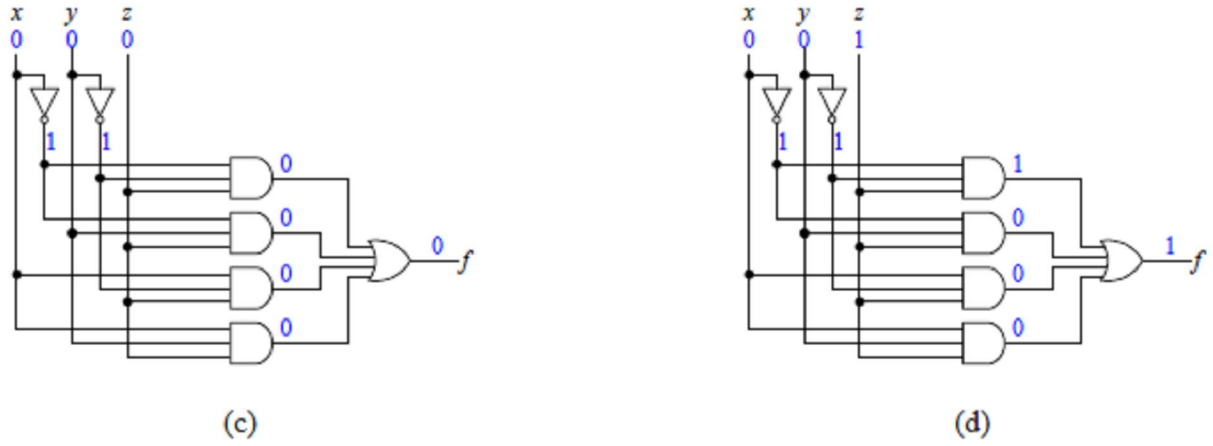


Figure 1.4(b): (c) circuit annotated with the input values *xyz* =000; (d) circuit annotated with the input values *xyz* = 001

Table 1.4(b): Complete truth table for the circuit.

x	*y*	*z*	*f*
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

(e)

The next step is to enumerate all possible combinations of 0's and 1's for all of the input variables. In general, for a circuit with *n* inputs, there are 2*n* combinations, from 0 to 2*n* – 1. Continuing on with the example, the table in (b) of Table 1.4(a) lists the eight combinations for the three variables in order.

Now, for each row in the table (that is, for each combination of input values) we need to determine what the output value is. This is done by substituting the values for the input variables and tracing through the circuit to the output. For example, using *xyz* = 000, the outputs for all of the AND gates are 0, and ORing all the zeros gives a zero, therefore, *f* = 0 for this set of values for *x*, *y*, and *z*. This is shown in the annotated circuit in (c) of Table 1.4(a).

For *xyz* = 001, the output of the top AND gate gives a 1, and 1 OR with anything gives a 1, therefore, *f* = 1, as shown in the annotated circuit in (d) of Figure 1.4(b).

Continuing in this fashion for all of the input combinations, we can complete the final truth table for the circuit, as shown in Table 1.4(b).

A faster method for evaluating the values for the output signals is to work backwards, that is, to trace the circuit from the output back to the inputs. You want to ask the question: When is the

output a 1 (or a 0)? Then trace back to the inputs to see what the input values ought to be in order to get the 1 output. For example, using the circuit in Figure 1.4(a), *f* is a 1 when any one of the four OR gate inputs is a 1. For the first input of the OR gate to be a 1, the inputs to the top AND gate must be all 1's. This means that the values for *x*, *y*, and *z* must be 0, 0, and 1, respectively. Repeat this analysis with the remaining three inputs to the OR gate. What you will end up with are the four input combinations for which *f* is a 1. The remaining input combinations, of course, will produce a 0 for *f*.

Example 1.4(a): Deriving a truth table from a circuit diagram
Derive the truth table for the following circuit with three inputs, *A*, *B* and *C*, and two outputs, *P* and *Q*:

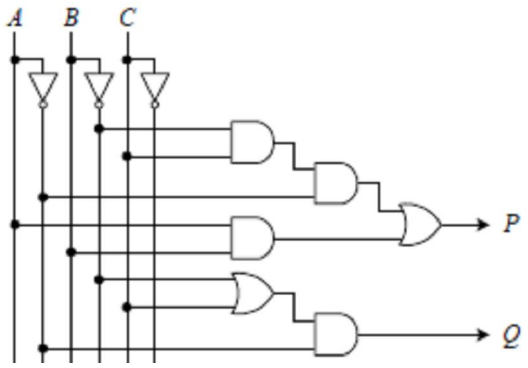


Figure 1.4(b): A circuit diagram

The truth table will have three columns for the three inputs and two columns for the two outputs. Enumerating all possible combinations of the three input values gives eight rows in the table. For each combination of input values, we need to evaluate the output values for both *P* and *Q*. For *P* to be a 1, either of the OR gate inputs must be a 1. The first input to this OR gate is a 1 if *ABC* = 001. The second input to this OR gate is a 1 if *AB* = 11. Since *C* is not specified in this case, it means that *C* can be either a 0 or a 1. Hence, we get the three input combinations for which *P* is a 1, as shown in the following truth table under the *P* column. The rest of the input combinations will produce a 0 for *P*. For *Q* to be a 1, both inputs of the AND gate must be a 1. Hence, *A* must be a 0, and either *B* is a 0 or *C* is a 1. This gives three input combinations for which *Q* is a 1, as shown in the truth table below under the *Q* column.

A	*B*	*C*	*P*	*Q*
0	0	0	0	1
0	0	1	1	1
0	1	0	0	0
0	1	1	0	1
1	0	0	0	0
1	0	1	0	0
1	1	0	1	0
1	1	1	1	0

3.1.2 Using a Boolean Function

To derive a Boolean function that describes a combinational circuit, we simply write down the Boolean logical expressions at the output of each gate (instead of substituting actual values of 0's and 1's for the inputs) as we trace through the circuit from the primary input to the primary output. Using the sample combinational circuit of Figure 1.4(a), we note that the logical expression for the output of the top AND gate is *x'y'z*. The logical expressions for the following AND gates are, respectively *x'yz*, *xy'z*, and *xyz*. Finally, the outputs from these AND gates are all ORed together. Hence, we get the final expression

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

To help keep track of the expressions at the output of each logic gate, we can annotate the outputs of each logic gate with the resulting logical expression as shown here.

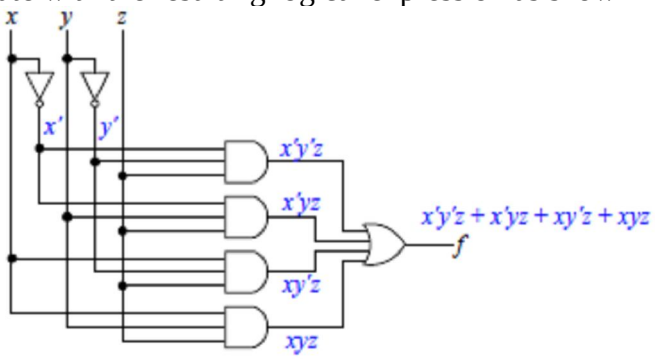
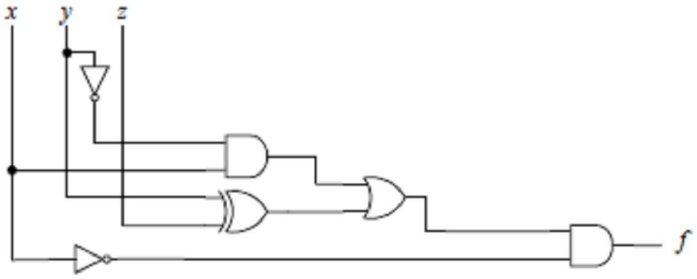


Figure 1.4(d): The annotated circuit for expression *f*

If we substitute all possible combinations of values for all of the variables in the final equation, we should obtain the same truth table as before.

Example 1.4(b): Deriving a Boolean function from a circuit diagram

Derive the Boolean function for the following circuit with three inputs, *x*, *y*, and *z*, and one output, *f*.



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:

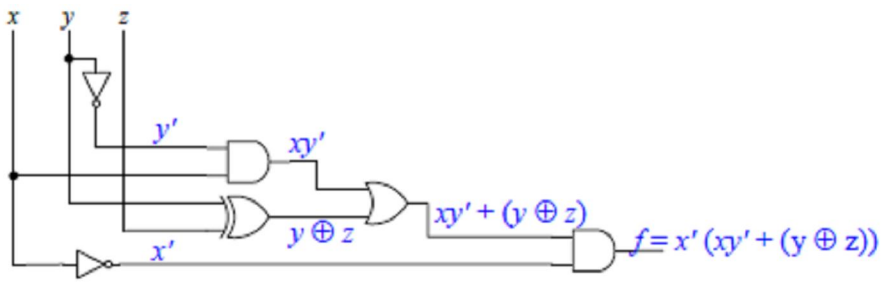


Figure 1.4(e): The annotated circuit for Example 1.4(b)

The Boolean function for the circuit is the final equation, $f = x'(xy' + (y \square z))$, at the output of the circuit. If a circuit has two or more outputs, then there must be one equation for each of the outputs. All the equations are then derived totally independent of each other.

3.2 Synthesis of Combinational Circuits

Synthesis of combinational circuits is just the reverse procedure of the analysis of combinational circuits. In synthesis, we start with a description of the operation of the circuit. From this description, we derive either the truth table or the Boolean logical function that precisely describes the operation of the circuit. Once we have either the truth table or the logical function, we can easily translate that into a circuit diagram.

For example, let us construct a 3-bit comparator circuit that outputs a 1 if the number is greater than or equal to 5 and outputs a 0 otherwise. In other words, construct a circuit that outputs a 0 if the input is a number between 0 and 4 inclusive and outputs a 1 if the input is a number between 5 and 7 inclusive. The reason why the maximum number is 7 is because the range for an unsigned 3-bit binary number is from 0 to 7. Hence, we can use the three bits, x_2, x_1, and x_0, to represent the 3-bit input value to the comparator. From the description, we obtain the following truth table:

Decimal number	Binary number			Output
	x_2	x_1	x_0	f
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	0
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	1

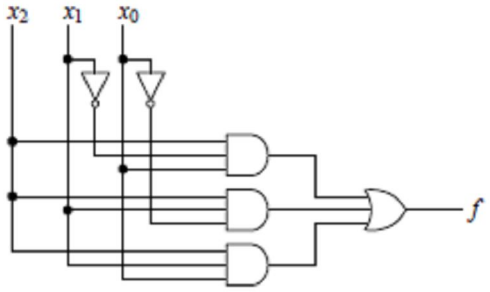
In constructing the circuit, we are interested only in when the output is a 1 (i.e., when the function f is a 1).

Thus, we only need to consider the rows where the output function $f = 1$. From the previous truth table, we see that there are three rows where $f = 1$, which give the three AND terms $x_2x_1'x_0$, $x_2x_1x_0'$, and $x_2x_1x_0$. Notice that the variables in the AND terms are such that it is inverted if its

value is a 0, and not inverted if its value is a 1. In the case of the first AND term, we want $f = 1$ when $x_2 = 1$ and $x_1 = 0$ and $x_0 = 1$, and this is satisfied in the expression $x_2x_1'x_0$.
Similarly, the second and third AND terms are satisfied in the expressions $x_2x_1x_0'$ and $x_2x_1x_0$ respectively. Finally, we want $f = 1$ when either one of these three AND terms is equal to 1. So we ORed the three AND terms together giving us our final expression:

$$f = x_2x_1'x_0 + x_2x_1x_0' + x_2x_1x_0 \qquad (equation\ 1.0)$$

In drawing the schematic diagram, we simply convert the AND operators to AND gates and OR operators to OR ates. The equation is in the sum-of-products format, meaning that it is summing (ORing) the product (AND) terms.
A sum-of-products equation translates to a two-level circuit with the first level being made up of AND gates and the second level made up of OR gates. Each of the three AND terms contains three variables, so we use a 3-input AND gate for each of the three AND terms. The three AND terms are ORed together, so we use a 3-input OR gate to connect the output of the three AND gates. For each inverted variable, we need an inverter. The schematic diagram derived from Equation 1.0 is shown here.



From this discussion, we see that any combinational circuit can be constructed using only AND, OR, and NOT gates from either a truth table or a Boolean equation.

Example 1.4(c): Synthesizing a combinational circuit from a truth table

Synthesize a combinational circuit from the following truth table. The three variables, *a*, *b*, and *c*, are input signals, and the two variables, *x*, and *y*, are output signals.

a	*b*	*c*	*x*	*y*
0	0	0	1	0
0	0	1	0	0
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	1
1	1	0	1	0
1	1	1	0	0

We can first derive the Boolean equation from the truth table, and then derive the circuit from the equation, or we can derive the circuit directly from the truth table. For this example, we will first

derive the Boolean equation. Since there are two output signals, there will be two equations; one for each output signal.

From the previous unit, we saw that a function is formed by summing its 1-minterms. For output *x*, there are five 1-minterms: m_0, m_2, m_3, m_5, and m_6. These five minterms represent the five AND terms, *a'b'c'*, *a'bc'*, *a'bc*, *ab'c*, and *abc'*.

Hence, the equation for *x* is:

$$x = a'b'c' + a'bc' + a'bc + ab'c + abc'$$

Similarly, the output signal *y* has three 1-minterms, and they are *a'bc'*, *ab'c'*, and *ab'c*. Hence, the equation for *y* is

$$y = a'bc' + ab'c' + ab'c$$

The combinational circuit constructed from these two equations is shown in (a) of Figure 1.4(f).

. Each 3-variable AND term is replaced by a 3-input AND gate. The three inputs to these AND gates are connected to the three input variables *a*, *b*, and *c*, either directly if the variable is not primed or through a NOT gate if the variable is primed. For output *x*, a 5-input OR gate is used to connect the outputs of the five AND gates for the corresponding five AND terms. For output *y*, a 3-input OR gate is used to connect the outputs of the three AND gates.

Notice that the two AND terms, *a'bc'*, and *ab'c*, appear in both the *x* and the *y* equations. As a result, we do not need to generate these two signals twice. Hence, we can reduce the size of the circuit by not duplicating these two AND gates, as shown in (b) of Figure 1.4(f).

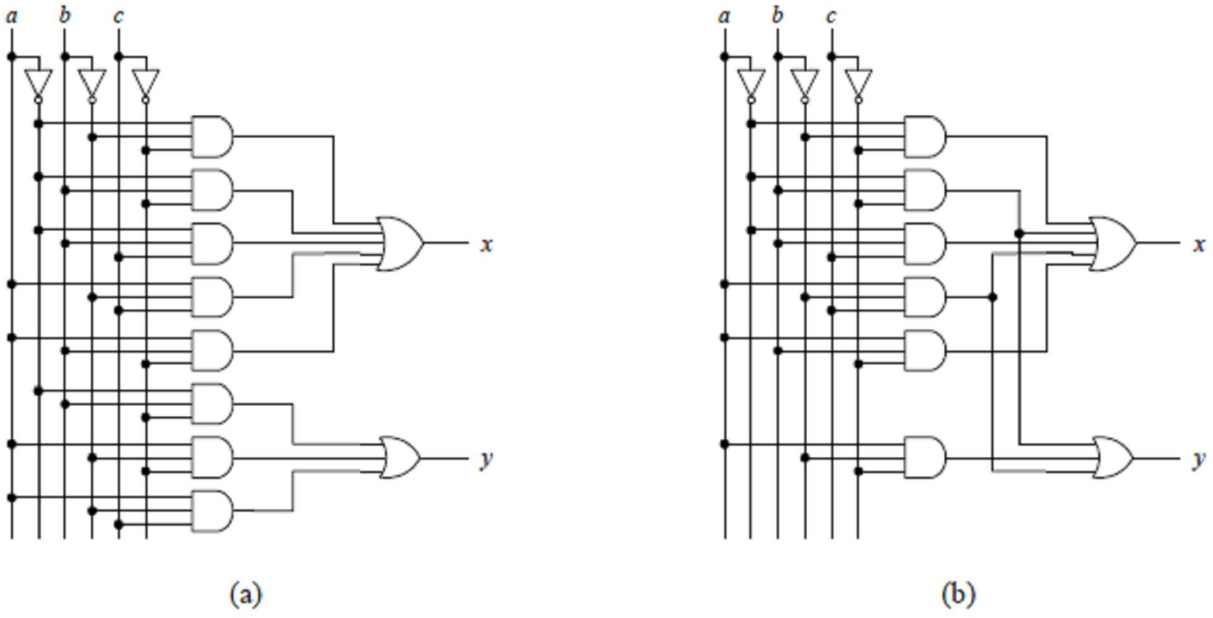


Figure 1.4(f): Combinational circuit for Example 1.4(c): (a) no reduction; (b) with reduction.

3.3 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 previous unit, 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 unit 5 of this module. The **Quine-McCluskey** or **tabulation** method for reducing an equation is ideal for programming the computer.

4.0 Conclusion

The logic circuits considered in this unit are combinatorial circuits whose output levels at any instant of time are independent of the levels present at the inputs at that time. Any prior input-level conditions have no effect on the present outputs because combinatorial logic circuits have no memory. Most digital systems are made up of both combinatorial circuits and memory elements.

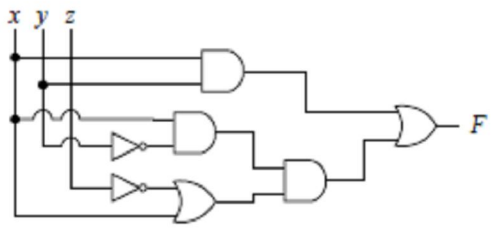
5.0 Summary

In this unit, you should be able to differentiate between combinatorial circuit and sequential circuit. Also to derive a Boolean function from combinatorial circuits. You should also have learnt how to synthesizing combinatorial circuits from truth table. Minimizing combinatorial circuits shouldn't be a problem.

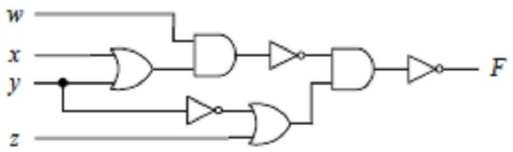
6.0 Tutor Marked Assignment

1) Derive the truth table for the following circuits:

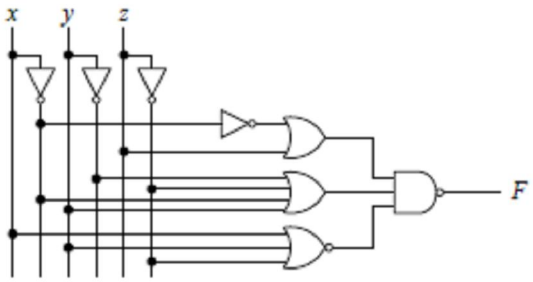
a)



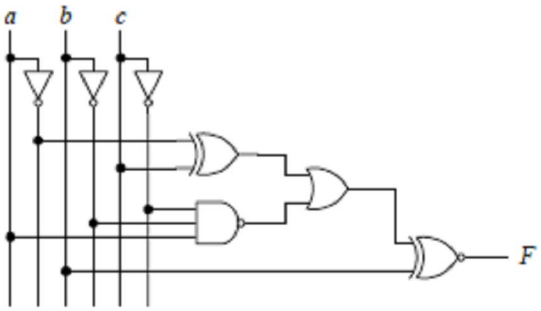
b)



c)



d)



2) Design a circuit that inputs a 4-bit number. The circuit outputs a 1 if the input number is greater than or equal to 5. Otherwise, it outputs a 0.

7.0 Further Reading and Other Resources

1. Ronald J. Tocci (1988). "Digital Systems: Priciples and Applications", 4th Edition Prentice-Hall International edition.
2. http://en.wikipedia.org/wiki/Logic_gate
3. http://www.discovercircuits.com/D/digital.htm
4. http://www.encyclopedia.com/doc/1G1-168332407.html
5. http://www.logiccircuit.org/

MODULE 1 - Basic Logic Operators and Logic Expressions

UNIT 5: Karnaugh Maps

Contents **Pages**

1.0 Introduction

The complexity of digital logic gates to implement a Boolean function is directly related to the complexity of algebraic expression. Also, an increase in the number of variables results in an increase of complexity. Although the truth table representation of a Boolean function is unique, its algebraic expression may be of many different forms. Boolean functions may be simplified or minimized by algebraic means as described in previous unit. However, this minimization procedure is not unique because it lacks specific rules to predict the succeeding step in the manipulative process. The map method, first proposed by Veitch and slightly improvised by Karnaugh, provides a simple, straightforward procedure for the simplification of Boolean functions. The method is called Veitch diagram or Karnaugh map, which may be regarded either as a pictorial representation of a truth table or as an extension of the Venn diagram. This unit gives a detailed discussion on Karnaugh maps.

2.0 Objectives

Upon completion of this unit, you will be able to:

- Modify a logic expression into a sum-of-products expression.
- Perform the necessary steps to derive a sum-of-products expression in order to design a combinatorial logic circuit in its simplest form.
- Use Karnaugh map as a tool to simplify and design logic circuits.
- Understand the role of don't cares in logic systems.
- Design logic circuit with and without the help of truth table.

3.0 What is Karnaugh Map?

A Karnaugh Map (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. The K-map 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.

3.1 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. 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 factor out the two common variables xz' and 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 previous equations. 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, sometimes it is difficult to see how the product terms can be combined and minimized. A **Karnaugh map** (**K-map** for short) provides a simple and straightforward procedure for combining these product terms.

Figure 1.5(a) 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 for the third and fourth rows are always interchanged. When we read K-maps, we need to visualize them 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 also differ in only one bit.
In Figure 1.5(a), the K-map squares are annotated with their minterms and minterm numbers for easy reference only. For example, in (a) of Figure 1.5(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. Be careful that, if we label the rows and columns differently, the minterms and the minterm numbers will be in different locations. When we use 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.
For a 5-variable K-map, as shown in (d) of Figure 1.5(a), we need to visualize the right half of the array (where $v = 1$) to be on top of the left half (where $v = 0$). In other words, we need to view the map as three-dimensional. Hence, although the squares for minterms 2 and 16 are located next to each other, they are not considered to be adjacent to each other. On the other hand, minterms 0 and 16 are adjacent to each other, because one is on top of the other.

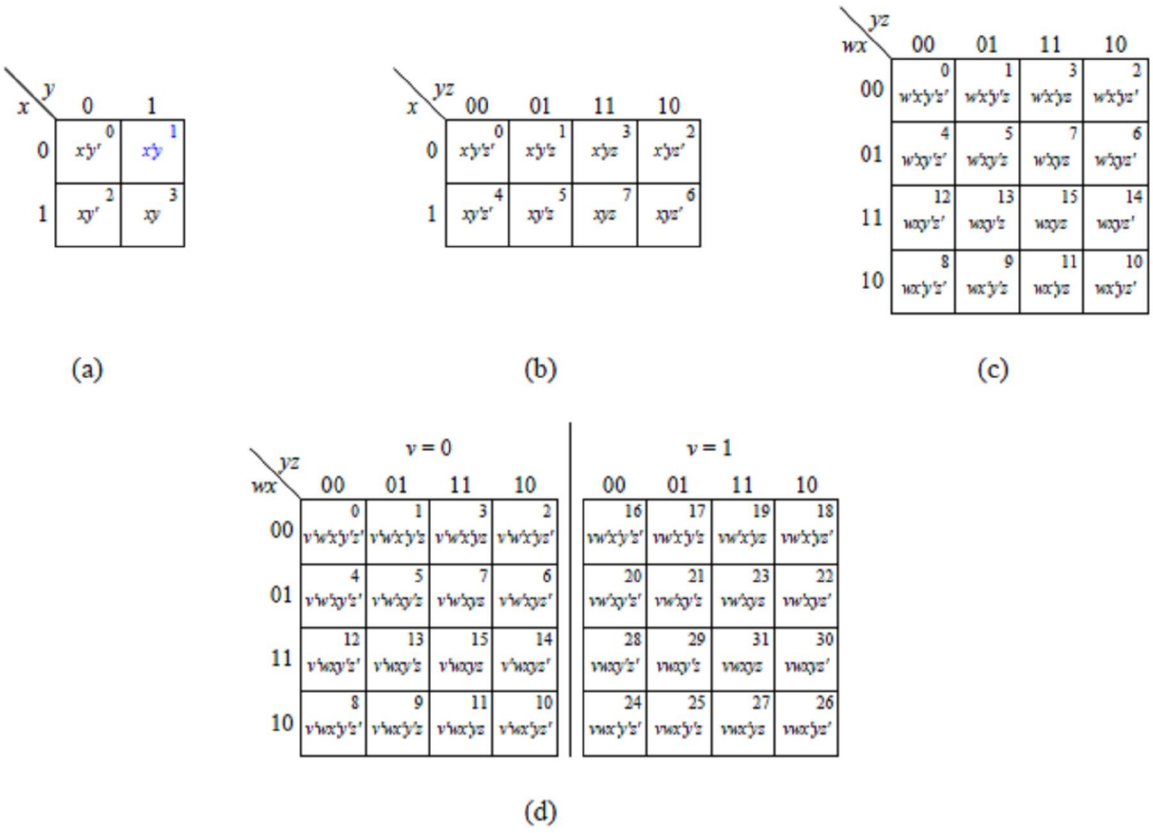
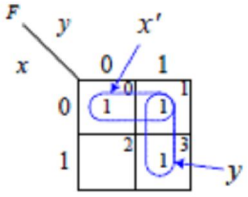


Figure 1.5(a): Karnaugh maps for: (a) 2 variables; (b) 3 variables; (c) 4 variables; (d) 5 variables.

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 using the 1-minterms for a function, the 0's are sometimes not written in the 0-minterm squares.
For example, the K-map for the 2-variable function:

$F = x'y' + x'y + xy$
is



The 1-minterms, m_0 ($x'y'$) and m_1 ($x'y$), are adjacent to each other, which means that they differ in the value of only one variable. In this case, x is 0 for both minterms, but for y, it is a 0 for one minterm and a 1 for the other minterm. Thus, variable y can be dropped, and the two terms are

combined together giving just x'. The prime in x' is because x is 0 for both minterms. This reasoning corresponds with the expression:

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

Similarly, the 1-minterms m_1 ($x'y$) and m_3 (xy) are also adjacent and y is the variable having the same value for both minterms, and so they can be combined to give

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

We use the term **subcube** to refer to a rectangle of adjacent 1-minterms. These subcubes must be rectangular in shape and can only have sizes that are powers of two. Formally, for an n-variable K-map, an *m-subcube* is defined as that set of 2m minterms in which $n - m$ of the variables will have the same value in every minterm, while the remaining variables will take on the 2m possible combinations of 0's and 1's. Thus, a 1-minterm all by itself is called a 0-subcube, two adjacent 1-minterms is called a 1-subcube, and so on. In the previous 2-variable K-map, there are two 1-subcubes: one labeled with x' and one labeled with y.

A 2-subcube will have four adjacent 1-minterms and can be in the shape of any one of those shown in (a) through (e) of Figure 1.5(b). Notice that (d) and (e) in Figure 1.5(b) also form 2-subcubes, even though the four 1-minterms are not physically adjacent to each other. They are considered to be adjacent because the first and last rows and the first and last columns wrap around in a K-map. In (f) of Figure 1.5(b) the four 1-minterms cannot form a 2-subcube, because even though they are physically adjacent to each other, they do not form a rectangle. However, they can form three 1-subcubes – $y'z$, $x'y'$ and $x'z$.

We say that a subcube is *characterized* by the variables having the same values for all of the minterms in that subcube. In general, an m-subcube for an n-variable K-map will be characterized by $n - m$ variables. If the value that is similar for all of the variables is a 1, that variable is unprimed; whereas, if the value that is similar for all of the variables is a 0, that variable is primed. In an expression, this is equivalent to the resulting smaller product term
when the minterms are combined together. For example, the 2-subcube in (d) of Figure 1.4(b) is characterized by z', since the value of z is 0 for all of the minterms, whereas the values for x and y are not all the same for all of the minterms.

Similarly, the 2-subcube in (e) of Figure 1.5(b) is characterized by $x'z'$.

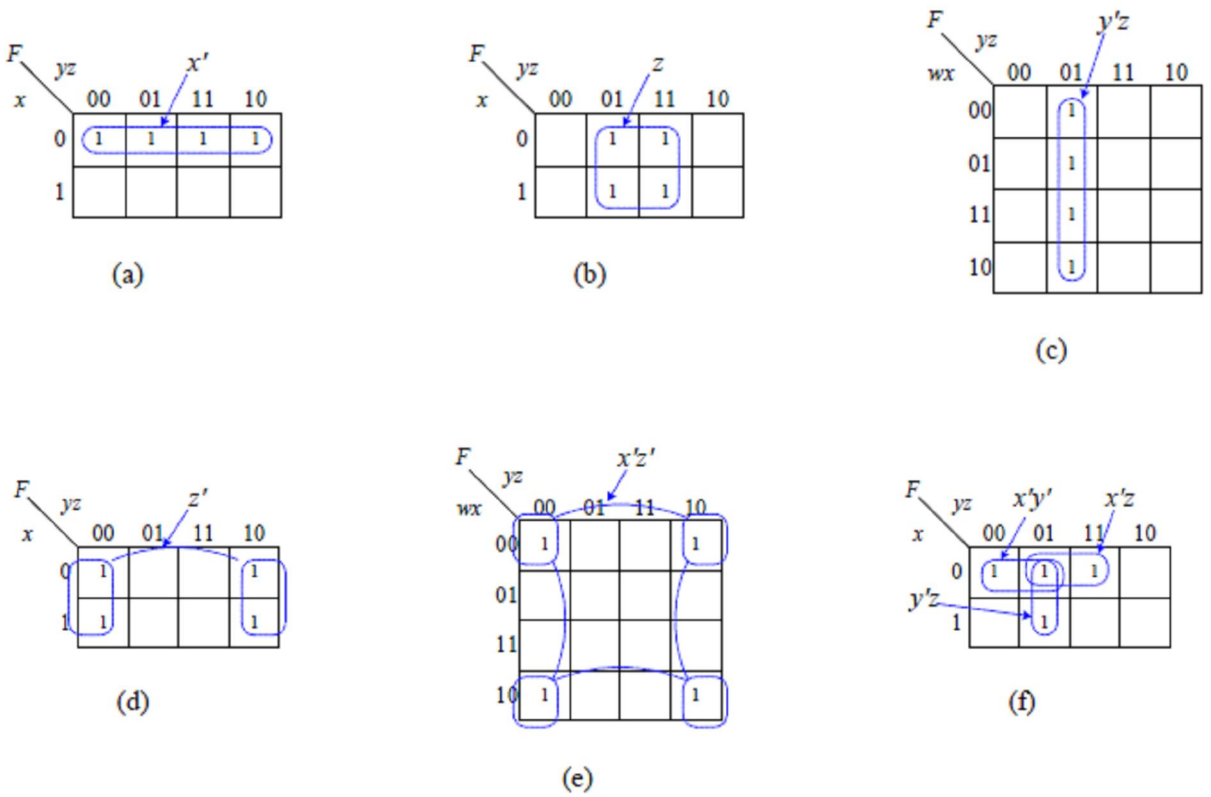


Figure 1.5(b): Examples of K-maps with 2-subcubes: (a) and (b) 3-variable; (c) 4-variable; (d) 3-variable with wrap around subcube; (e) 4-variable with wrap around subcube; (f) four adjacent minterms that cannot form one 2-subcube.

For a 5-variable K-map, as shown in Figure 1.5(c), we need to visualize the right half of the array (where $v = 1$) to be on top of the left half (where $v = 0$). Thus, for example, minterm 20 is adjacent to minterm 4 since one is on top of the other, and they form the 1-subcube $w'xy'z'$. Even though minterm 6 is physically adjacent to minterm 20 on the map, they cannot be combined together, because when you visualize the right half as being on top of the left half, then they really are not on top of each other. Instead, minterm 6 is adjacent to minterm 4 because the columns wrap around, and they form the subcube $v'w'xz'$. Minterms 9, 11, 13, 15, 25, 27, 29, and 31 are all adjacent, and together they form the subcube wz. Now that we are viewing this 5-variable K-map in three dimensions, we also need to change the condition of the subcube shape to be a three-dimensional rectangle.

You can see that this visualization becomes almost impossible to work with very quickly as we increase the number of variables. In more realistic designs with many more variables, tabular methods (instead of K-maps) are used for reducing the size of equations.

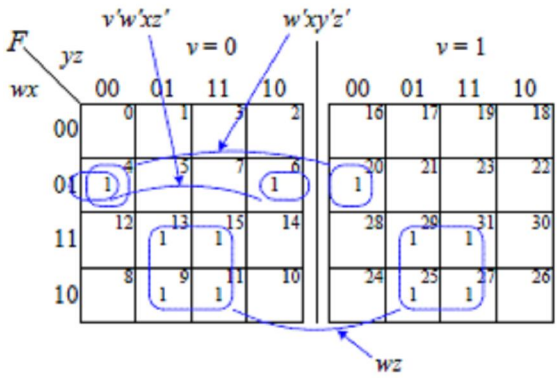


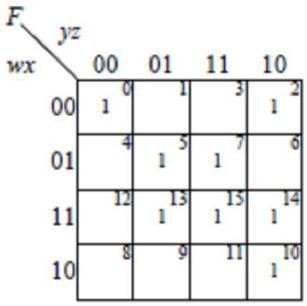
Figure 1.5(c): A 5-variable K-map with wrap-around subcubes.

The K-map method reduces a Boolean function from its canonical form to its standard form. The goal for the Kmap method is to find as few subcubes as possible to cover all of the 1-minterms in the given function. This naturally implies that the size of the subcube should be as big as possible. The reasoning for this is that each subcube corresponds to a product term, and all of the subcubes (or product terms) must be ORed together to get the function.
Larger subcubes require fewer AND gates because of fewer variables in the product term, and fewer subcubes will require fewer inputs to the OR gate.
The procedure for using the K-map method is as follows:

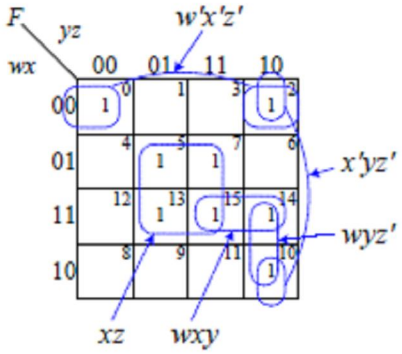
1. Draw the appropriate K-map for the given function and place a 1 in the squares that correspond to the function's 1-minterms.
2. For each 1-minterm, find the largest subcube that covers this 1-minterm. This largest subcube is known as a prime implicant (PI). By definition, a **prime implicant** is a subcube that is not contained within any other subcube. If there is more than one subcube that is of the same size as the largest subcube, then they are all prime implicants.
3. Look for 1-minterms that are covered by only one prime implicant. Since this prime implicant is the only subcube that covers this particular 1-minterm, this prime implicant must be in the final solution. This prime implicant is referred to as an *essential* prime implicant (EPI). By definition, an **essential prime implicant** is a prime implicant that includes a 1-minterm that is not included in any other prime implicant.
4. Create a minimal cover list by selecting the smallest possible number of prime implicants such that every 1-minterm is contained in at least one prime implicant. This cover list must include all of the essential prime implicants plus zero or more of the remaining prime implicants. It is acceptable that a particular 1-minterm is covered in more than one prime implicant, but all 1-minterms must be covered.
5. The final minimized function is obtained by ORing all of the prime implicants from the minimal cover list. Note that the final minimized function obtained by the K-map method may not be in its most reduced form. It is only in its most reduced *standard* form. Sometimes, it is possible to reduce the standard form further into a nonstandard form.

Example 1.5(a): Using K-map to minimize a 4-variable function.
Use the K-map method to minimize a 4-variable (*w*, *x*, *y*, and *z*) function *F* with the 1-minterms: m_0, m_2, m_5, m_7, m_{10}, m_{13}, m_{14}, and m_{15}.

We start with the following 4-variable K-map with a 1 placed in each of the eight minterm squares:



The prime implicants for each of the 1-minterms are shown in the following K-map and table:



1-minterm	Prime Implicant
m_0	$w'x'z'$
m_2	$w'x'z'$, $x'yz'$
m_5	xz
m_7	xz
m_{10}	$x'yz'$, wyz'
m_{13}	xz
m_{14}	wyz', wxy
m_{15}	xz

For minterm m_0, there is only one prime implicant $w'x'z'$. For minterm m_2, there are two 1-subcubes that cover it, and they are the largest. Therefore, m_2 has two prime implicants, $w'x'z'$ and $x'yz'$. When we consider m_{14}, again there are two 1-subcubes that cover it, and they are the largest. So m_{14} also has two prime implicants. Minterm m_{15}, however, has only one prime implicant xz. Although the 1-subcube wxy also covers m_{15}, it is not a prime implicant for m_{15} because it is smaller than the 2-subcube xz.

From the K-map, we see that there are five prime implicants: $w'x'z'$, $x'yz'$, xz, wyz', and wxy. Of these five prime implicants, $w'x'z'$ and xz are essential prime implicants, since m_0 is covered only by $w'x'z'$, and m_5, m_7, and m_{13} are covered only by xz.

We start the cover list by including the two essential prime implicants $w'x'z'$ and xz. These two subcubes will have covered the minterms m_0, m_2, m_5, m_7, m_{13}, and m_{15}. To cover the remaining two uncovered minterms, m_{10} and m_{14}, we want to use as few prime implicants as possible. Hence, we select the prime implicant wyz', which covers both of them.

Finally, our reduced standard form equation is obtained by ORing the two essential prime implicants and one prime implicant in the cover list:

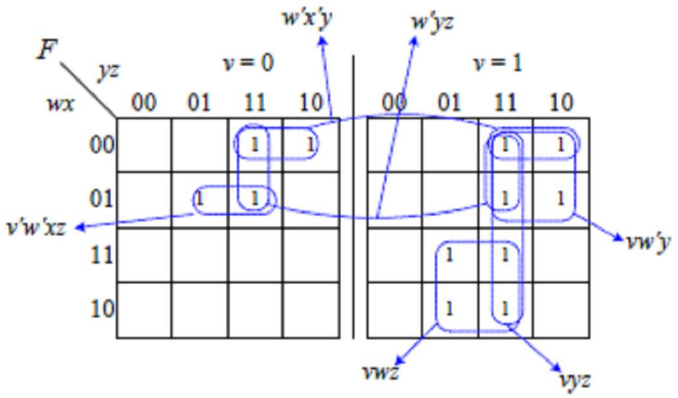
$$F = w'x'z' + xz + wyz'$$

Notice that we can reduce this standard form equation even further by factoring out the z' from the first and last term to get the non-standard form equation

$F = z'(w'x' + wy) + xz$

Example 1.5(b): Using K-map to minimize a 5-variable function

Use the K-map method to minimize a 5-variable function F (v, w, x, y and z) with the 1-minterms: $v'w'x'yz'$, $v'w'x'yz$, $v'w'xy'z$, $v'w'xyz$, $vw'x'yz'$, $vw'x'yz$, $vw'xyz'$, $vw'xyz$, $vwx'y'z$, $vwx'yz$, $vwxy'z$, and $vwxyz$.



The list of prime implicants is: $v'w'xz$, $w'x'y$, $w'yz$, $vw'y$, vyz, and vwz. From this list of prime implicants, $w'yz$ and vyz are not essential. The four remaining essential prime implicants are able to cover all of the 1-minterms. Hence, the solution in standard form is

$F = v'w'xz + w'x'y + vw'y + vwz$

3.2 Don't-cares

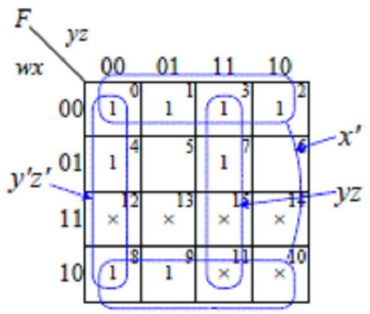
There are times when a function is not specified fully. In other words, there are some minterms for the function where we do not care whether their values are a 0 or a 1. When drawing the K-map for these "**don't-care**" minterms, we assign an "×" in that square instead of a 0 or a 1. Usually, a function can be reduced even further if we remember that these ×'s can be either a 0 or a 1. As you recall when drawing K-maps, enlarging a subcube reduces the number of variables for that term. Thus, in drawing subcubes, some of them may be enlarged if we treat some of these ×'s as 1's. On the other hand, if some of these ×'s will not enlarge a subcube, then we want to treat them as 0's so that we do not need to cover them. It is not necessary to treat all ×'s to be all 1's or all 0's. We can assign some ×'s to be 0's and some to be 1's.

For example, given a function having the following 1-minterms and don't-care minterms:

1-minterms: m_0, m_1, m_2, m_3, m_4, m_7, m_8, and m_9

×-minterms: m_{10}, m_{11}, m_{12}, m_{13}, m_{14}, and m_{15}

we obtain the following K-map with the prime implicants x', yz, and $y'z'$.



Notice that, in order to get the 4-subcube characterized by x', the two don't-care minterms, $m10$ and $m11$, are taken to have the value 1. Similarly, the don't-care minterms, $m12$ and $m15$, are assigned a 1 for the subcubes $y'z'$ and yz, respectively. On the other hand, the don't-care minterms, $m13$ and $m14$, are taken to have the value 0, so that they do not need to be covered in the solution. The reduced standard form function as obtained from the K-map is, therefore,

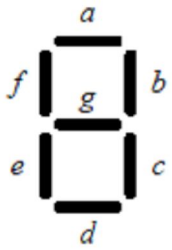
$$F = x' + yz + y'z'$$

Again, this equation can be reduced further by recognizing that $yz + y'z' = y \bigcirc z$. Thus,

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

3.3 BCD to 7-Segment Decoder

We will now synthesize the circuit for a BCD to 7-segment decoder for driving a 7-segment LED display. The decoder converts a 4-bit binary coded decimal (BCD) input to seven output signals for turning on the seven lights in a 7-segment LED display. The 4-bit input encodes the binary representation of a decimal digit. Given the decimal digit input, the seven output lines are turned on in such a way so that the LED displays the corresponding digit. The 7-segment LED display schematic with the names of each segment labeled is shown here



The operation of the BCD to 7-segment decoder is specified in the truth table in Table 1.5(a). The four inputs to the decoder are i_3, i_2, i_1, and i_0, and the seven outputs for each of the seven LEDs are labeled a, b, c, d, e, f, and g.

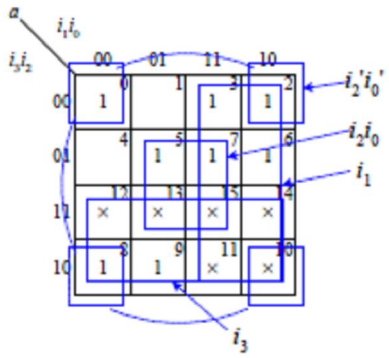
For each input combination, the corresponding digit to display on the 7-segment LED is shown in the "Display" column. The segments that need to be turned on for that digit will have a 1 while the segments that need to be turned off for that digit will have a 0. For example, for the 4-bit input 0000, which corresponds to the digit 0, segments a, b, c, d, e, and f need to be turned on, while segment g needs to be turned off.

Notice that the input combinations 1010 to 1111 are not used and so don't-care values are assigned to all of the segments for these six combinations.

Table 1.5(a): Truth table for the BCD to 7-segment decoder.

Inputs i_3	i_2	i_1	i_0	Decimal Digit	Display	a	b	c	d	e	f	g
0	0	0	0	0		1	1	1	1	1	1	0
0	0	0	1	1		0	1	1	0	0	0	0
0	0	1	0	2		1	1	0	1	1	0	1
0	0	1	1	3		1	1	1	1	0	0	1
0	1	0	0	4		0	1	1	0	0	1	1
0	1	0	1	5		1	0	1	1	0	1	1
0	1	1	0	6		1	0	1	1	1	1	1
0	1	1	1	7		1	1	1	0	0	0	0
1	0	0	0	8		1	1	1	1	1	1	1
1	0	0	1	9		1	1	1	0	0	1	1
rest of the combinations						×	×	×	×	×	×	×

From the truth table in Table 1.5(a), we are able to specify seven equations that are dependent on the four inputs for each of the seven segments. For example, the canonical form equation for segment a is $a = i_3'i_2'i_1'i_0' + i_3'i_2'i_1i_0' + i_3'i_2'i_1i_0 + i_3'i_2i_1'i_0 + i_3'i_2i_1i_0' + i_3'i_2i_1i_0 + i_3i_2'i_1'i_0' + i_3i_2'i_1'i_0$
Before implementing this equation directly in a circuit, we want to simplify it first using the K-map method.
The K-map for the equation for segment a is



From evaluating the K-map, we derive the simpler equation for segment a as

$$a = i_3 + i_1 + i_2'i_0' + i_2i_0 = i_3 + i_1 + (i_2 \odot i_0)$$

Proceeding in a similar manner, we get the following remaining six equations

$$b = i_2' + (i_1 \odot i_0)$$
$$c = i_2 + i_1' + i_0$$
$$d = i_1i_0' + i_2'i_0' + i_2'i_1 + i_2i_1'i_0$$
$$e = i_1i_0' + i_2'i_0'$$
$$f = i_3 + i_2i_1' + i_2i_0' + i_1'i_0'$$
$$g = i_3 + (i_2 \oplus i_1) + i_1i_0'$$

From these seven simplified equations, we can now implement the circuit, as shown in Figure 1.5(d).

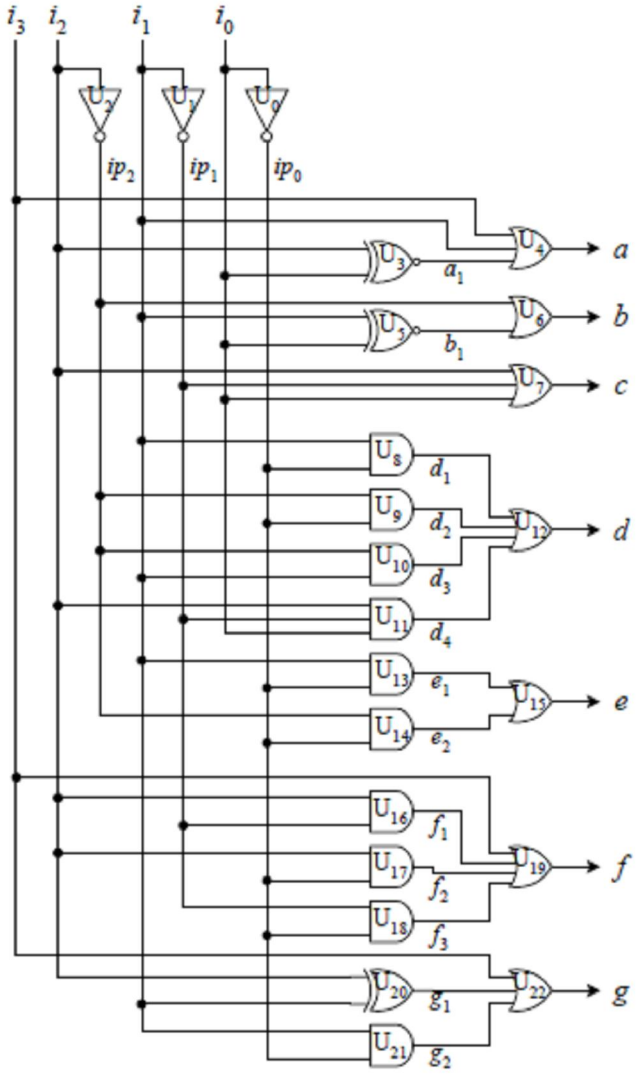


Figure 1.5(d): Circuit for the BCD to 7-segment decoder.

Self Assessment Exercise

Use the K-Map to simplify the expression x = A′B′C′ + B′C + A′B

4.0 Conclusion

Looking at a logic function's truth table, sometimes it is difficult to see how the product terms can be combined and minimized. A Karnaugh map (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. The K-map 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

5.0 Summary

In this unit, you learnt about:

- The use of Karnaugh maps to minimize Boolean equation.
- The don't cares.
- How to synthesize the circuit for a BCD to 7-segment.

6.0 Tutor Marked Assignment

1) Use K-maps to reduce the Boolean functions for the following expressions
   a) $F(x, y, z) = \sum(0, 1, 6)$
   b) $F(w,x, y, z) = \sum(0, 1, 6)$
   c) $F(w,x, y, z) = \sum(2, 6, 10, 11, 14, 15)$
   d) $F(x, y, z) = \sum(0, 1, 6)$
   e) $F(w,x, y, z) = \sum(0, 1, 6)$
   f) $F(w,x, y, z) = \sum(2, 6, 10, 11, 14, 15)$

2) Use K-maps to reduce the Boolean functions
   a) $F = xy' + x'y'z + xyz'$
   b) $F = w'z' + w'xy + wx'z + wxyz$
   c) $F = w'xy'z + w'xyz + wxy'z + wxyz$

3) What is meant by “don’t care” conditions?

7.0 Further Reading and Other Resources

1. Ronald J. Tocci (1988). “Digital Systems: Priciples and Applications”, 4th Edition Prentice-Hall International edition.
2. http://en.wikipedia.org/wiki/Logic_gate
3. http://www.discovercircuits.com/D/digital.htm
4. http://www.encyclopedia.com/doc/1G1-168332407.html
5. http://www.logiccircuit.org/

MODULE 2 - Latches and Flip-Flops

UNIT 1: Sequential Circuits

Contents **Pages**

1.0 Introduction

So far, we have been looking at the design of combinational circuits. We will now turn our attention to the design of **sequential circuits**. Recall that the outputs of sequential circuits are dependent on not only their current inputs (as in combinational circuits), but also on all their past inputs. Because of this necessity to remember the history of inputs, sequential circuits must contain memory elements.

2.0 Objectives

Upon completion of this unit, you will be able to:

- Understand Latches and Flip-Flops
- Understand Bistable element
- Understand SR Latch
- Understand D Latch

3.0 Latches and Flip-Flops

In order to remember the history of inputs, sequential circuits must have memory elements. Memory elements, however, are just like combinational circuits in the sense that they are made up of the same basic logic gates. What makes them different is in the way these logic gates are connected together. In order for a circuit to "remember" its current value, we have to connect the output of a logic gate directly or indirectly back to the input of that same gate.

We call this a **feedback loop** circuit, and it forms the basis for all memory elements. Combinational circuits do not have any feedback loops.

Latches and **flip-flops** are the basic memory elements for storing information. Hence, they are the fundamental building blocks for all sequential circuits. A single latch or flip-flop can store only one bit of information. This bit of information that is stored in a latch or flip-flop is referred to as the **state** of the latch or flip-flop. Hence, a single latch or flip-flop can be in either one of two states: 0 or 1. We say that a latch or a flip-flop changes state when its content changes from a 0 to a 1 or vice versa. This state value is always available at the output. Consequently, the content of a latch or a flip-flop is the state value, and is always equal to its output value.

The main difference between a latch and a flip-flop is that for a latch, its state or output is constantly affected by its input as long as its enable signal is asserted. In other words, when a latch is enabled, its state changes immediately when its input changes. When a latch is disabled, its state remains constant, thereby, remembering its previous value. On the other hand, a flip-flop changes state only at the active edge of its enable signal, i.e., at precisely the moment when either its enable signal rises from a 0 to a 1 (referred to as the rising edge of the signal), or from a 1 to a 0 (the falling edge). However, after the rising or falling edge of the enable signal, and during the time when the enable signal is at a constant 1 or 0, the flip-flop's state remains constant even if the input changes. In a microprocessor system, we usually want changes to occur at precisely the same moment. Hence, flip-flops are used more often than latches, since they can all be synchronized to change only at the active edge of the enable signal. This enable signal for the flip-flops is usually the global controlling clock signal.