NATIONAL OPEN UNIVERSITY OF NIGERIA

FACULTY OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

COURSE CODE:CIT 292

COURSE TITLE:COMPUTER LABORATORY I

Course Code	CSC 292
Course Title	COMPUTER LABORATORY I
Course Developer/Writer	ONASHOGA, S. A. (Mrs.) DEPT. OF COMPUTER SCIENCE UNIVERSITY OF AGRICULTURE, ABEOKUTA, OGUN STATE NIGERIA. (2020)
Course Editor	
Programme Leader	
Course Coordinator	

NATIONAL OPEN UNIVERSITY OF NIGERIA

Introduction

With the advancement of technology, digital logic systems became inevitable and became the integral part of digital circuit design. Digital logic is concerned with the interconnection of digital components and modules, and is a term used to denote the design and analysis of digital systems. Recent technology advancements have led to enhanced usage of digital systems in all disciplines of engineering and have also created the need of in-depth knowledge about digital circuits among the students as well as the instructors. It has been felt that a single textbook dealing with the basic concepts of digital technology with design aspects and applications is the standard requirement. This course is designed to fulfill such a requirement by presenting the basic concepts used in the design and analysis of digital systems, and also providing various methods and techniques suitable for a variety of digital system design applications.

Digital circuits can be designed at any one of several abstraction levels. When designing a circuit at the **transistor level**, which is the lowest level, you are dealing with discrete transistors and connecting them together to form the circuit. The next level up in the abstraction is the **gate level**. At this level, you are working with logic gates to build the circuit. At the gate level, you also can specify the circuit using either a truth table or a Boolean equation.
In using logic gates, a designer usually creates standard combinational and sequential components for building larger circuits. In this way, a very large circuit, such as a microprocessor, can be built in a hierarchical fashion.
Design methodologies have shown that solving a problem hierarchically is always easier than trying to solve the entire problem as a whole from the ground up. These combinational and sequential components are used at the register-transfer level in building the datapath and the control unit in the microprocessor. At the register-transfer level, we are concerned with how the data is transferred between the various registers and functional units to realize or solve the problem at hand. Finally, at the highest level, which is the behavioral level, we construct the circuit by describing the behavior or operation of the circuit using a hardware description language. This is very similar to writing a computer program using a programming language.

What you will learn in this Course

The course consists of units and a course guide. This course guide tells you briefly what the course is about. What course materials you will be using and how you can work your way with these materials. In addition, it advocates some guidelines for the amount of time you are likely to spend on each unit of the course in order to complete it successfully.

It gives you guidance in respect of your Tutor-Marked Assignment which will be made available in the assignment file. There will be regular tutorial classes that are related to the course. It is advisable for you to attend these tutorial sessions. The course will prepare you for the challenges you will meet as you explore logic circuit and digital design.

Course Aims

This course provides an introduction to digital logic design. For an introductory course with no previous background in logic, Module 1 is intended to provide the fundamental concepts in designing combinational circuits, and Module 2 is intended to cover the basic sequential circuits. Module 3 aims to cover counter design.

Course Objectives

To achieve the aims set out, the course has a set of objectives. Each unit has a specific objectives which are included at the beginning of the unit. You should read these objectives before you study the unit. You may wish to refer to them during your study to check on your progress. You should always look at the unit objectives after completion of each unit. By doing so, you would have followed the instruction in the unit.

Below are the comprehensive objectives of the course as a whole. By meeting these objectives, you should have achieved the aims above.

Working through this Course

To complete this course you are required to read each study unit, read the textbooks and read other materials which may be provided by the National Open University of Nigeria.

Each unit contains an assignment which you would be required to submit for assessment purpose. At the end of the course, there is a final examination. The course should take about 17 weeks to complete. Below you will find listed all the components of the course, what you have to do and how you should allocate your time to each unit in order to complete the course on time and successfully.

This course entails that you spend a lot of time to read. I would advice that you avail yourself the opportunity of attending the tutorial sessions where you will have the opportunity of comparing your knowledge with that other people.

The Course Materials

The main components of the course are:

1. The Course Guide
2. Study Units
3. Assignments
4. Further Readings

Study Unit

The study units in this course are as follows:

Module 1 Basic Logic Operators and Logic Expressions

Unit 1 Basic Logic Operations

Unit 2 Boolean Algebra and Functions

Unit 3 Logic Gates and Circuit Diagrams

Unit 4 Combinatorial Circuit

Unit 5 Karnaugh Maps

Module 1 Latches and Flip-Flops

Unit 1 Sequential Circuits

Unit 2 Flip-Flops

Unit 3 Clocked Flip-Flops

Unit 4 J-K Flip-Flops

Module 3 Counters

Unit 1 Introduction to Counters

Unit 2 Asynchronous Counter

Unit 3 Synchronous Counter

Assessment

There are two assessment of this course. First is made up of the tutor-marked assignments and the second is the written examination/end of course examination.

You are advised to do the exercises. In tackling the assignments, you are expected to apply information, knowledge and techniques you gathered during the course. The assignments must be submitted to your facilitator for formal assessment in accordance with the deadlines stated in the assignment file. The work you submitted to your tutor for assessment will count for 30% of your total course work. At the end of the course you will need to sit for a final or end of course examination of about three hour duration. This examination will count for 70% of your total course mark.

Tutor-Marked Assignment (TMA)

The TMA is a continuous assessment component of your course. It accounts for 30% of the total score. You will be given four (4) TMAs to answer. Three of these must be answered before you are allowed to sit for the end of course examination. The TMAs would be given to you by your facilitator and returned after you have done the assignment. Assignment question for the units in this course are contained in the assignment file. You will be able to complete your assignment from the information and material contained in your reading, references and study units. However, it is desirable in all degree level of education to demonstrate that you have read and researched more into your references which will give you a wider view point and may provide you with a deeper understanding of the subject.

Make sure that each assignment reaches your facilitator on or before the deadline given in the assignment file. If for any reason you cannot complete your work on time, contact your facilitator before the assignment is due to discuss the possibility of an extension, Extension will not be granted after the due date unless there are exceptional circumstances.

Final Examination and Grading

The duration for the final exam is about 3 hours and it has a value of 70% of the total course work. The examination will consist of questions, which will reflect the type of self-testing, practice exercise and tutor-marked assignment problems you have previously encountered. All areas of the course will be assessed.

Make use the time between finishing the last unit and sitting for the examination to revise the whole course. You might find it useful to review your self-test, TMAs and comment on them before the examination. The end of course examination covers information from all parts of the course.

Course Marking Scheme

Assignment	Marks
Assignment 1-4	Four Assignments, best three marks of the four count at 10% each -30% of course marks.
End of Course Examination	70% of overall course marks.
Total	100% of course materials

Facilitators/Tutors and Tutorials

There are 16 hours of tutorials provided in support of this course. You will be notified of the dates, times and locations of these tutorials as well as the name and phone number of your facilitator so soon as you are allocated a tutorial group.

Your facilitator will mark and comment on your assignment, keep a close watch on your progress and any difficulties you might face and provide assistance to you during the course. You are expected to mail your Tutor Marked Assignment to your facilitator before the schedule date (at least two working days are required). They will be marked by your tutor and returned to you as soon as possible.

Do not delay to contact your facilitator by telephone or e-mail if you need assistance.

The following might be circumstances in which you would find assistance necessary, hence you would have to contact your facilitator if:

- You do not understand any part of the study or the assigned readings.
- You have difficulty with the self-tests.
- You have a question or problem with an assignment or with the grading of an assignment.

You should endeavor to attend the tutorials. This is the only chance to have face-to-face contact with your course facilitator and to ask questions which are answered instantly. You can raise any problem encountered in the course of your study.

Summary

This course intends to provide a background to computer electronics by looking into designs of digital circuits using logic gates. Upon completing the course, you will be equipped with the

basic knowledge of the principles and techniques that are common to all digital systems from the simplest on/off switch to the most complex computers. In addition, you will be able to answer the following questions:

3 Of what importance is the binary numbering system to digital systems?

4 What operations can we perform with binary numbers?

5 What is a Logic gate?

6 How can we build a digital circuit with logic gates?

7 How can we use Karnaugh map to simplify and design logic circuits?

8 Can you construct and analyze the operation of a latch flip-flop made from NAND and NOR gates?

9 Can you describe the difference between synchronous and asynchronous systems?

The questions to be answered in this course are not limited to the above-listed. To gain the most from this course, you should check out the texts from the list of references for further studies.

As you go through this class, I wish you success in this course and I hope you will find it both interesting and useful.

Best of Luck!

MODULE 1 - Basic Logic Operations and Logic Expressions
UNIT 1: Basic Logic Operations

Contents **Pages**

1.0 Introduction

Our world is an analog world. Measurements that we make of the physical objects around us are never in discrete units, but rather in a continuous range. We talk about physical constants such as 2.718281828… or 3.141592…. To build analog devices that can process these values accurately is next to impossible. Even building a simple analog radio requires very accurate adjustments of frequencies, voltages, and currents at each part of the circuit. If we were to use voltages to represent the constant 3.14, we would have to build a component that will give us exactly 3.14 volts every time. This is again impossible; due to the imperfect manufacturing process, each component produced is slightly different from the others. Even if the manufacturing process can be made as perfect as we can get, we still would not be able to get 3.14 volts from this component every time we use it. The reason being that the physical elements used in producing the component behave differently in different environments, such as temperature, pressure, and gravitational force, just to name a few. Therefore, even if the manufacturing process is perfect, using this component in different environments will not give us exactly 3.14 volts every time.

To make things simpler, we work with a digital abstraction of our analog world. Instead of working with an infinite continuous range of values, we use just two values: 1 and 0, on and off, high and low, true and false, black and white, or however you want to call it. It is certainly much easier to control and work with two values rather than an infinite range. We call these two values a binary value for the reason that there are only two of them. A single 0 or a single 1 is then a **binary digit** or **bit**. This sounds great, but we have to remember that the underlining building block for our digital circuits is still based on an analog world. This unit explores the world of binary system with the different operations attached to it.

2.0 Objectives

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

- Provide theoretical foundation for building digital logic circuits.
- Understand the binary number system and conversion
- Understand the use of Truth Tables
- Understand the Boolean algebras
- Understand the duality principle

3.0 Binary Numbers

Since digital circuits deal with binary values, we will begin with a quick introduction to binary numbers. A bit, having either the value of 0 or 1 can represent only two things or two pieces of information. It is, therefore, necessary to group many bits together to represent more pieces of information. A string of n bits can represent $2n$ different pieces of information. For example, a string of two bits results in the four combinations 00, 01, 10, and 11.

By using different encoding techniques, a group of bits can be used to represent different information, such as a number, a letter of the alphabet, a character symbol, or a command for the microprocessor to execute.

The use of decimal numbers is quite familiar to us. However, since the binary digit is used to represent information within the computer, we also need to be familiar with **binary numbers**. Note that the use of binary numbers is just a form of representation for a string of bits. We can

just as well use octal, decimal, Binary Coded Decimal (BCD) or hexadecimal numbers to represent the string of bits. In fact, you will find that hexadecimal numbers are often used as a shorthand notation for binary numbers.

The decimal number system is a positional system. In other words, the value of the digit is dependent on the position of the digit within the number. For example, in the decimal number 48, the decimal digit 4 has a greater value than the decimal digit 8 because it is in the tenth position, whereas the digit 8 is in the unit position. The value of the number is calculated as

$$4\times10^1 + 8\times10^0.$$

Like the decimal number system, the binary number system is also a positional system. The only difference between the two is that the binary system is a base-2 system, and so it uses only two digits, 0 and 1, instead of ten.

The binary numbers from 0 to 15 (decimal) are shown in Table 1.1(a). . The range from 0 to 15 has 16 different combinations, we need a 4-bit binary number, i.e., a string of four bits, to represent this range.

When we count in decimal, we count from 0 to 9. After 9, we go back to 0, and have a carry of a 1 to the next digit. When we count in binary, we do the same thing except that we only count from 0 to 1. After 1, we go back to 0, and have a carry of a 1 to the next bit.

The decimal value of a binary number can be found just like for a decimal number except that we raise the base number 2 to a power rather than the base number 10 to a power. For example, the value for the decimal number 658 is

$$658_{10} = 6\times10^2 + 5\times10^1 + 8\times10^0 = 600 + 50 + 8 = 658_{10}$$

Similarly, the decimal value for the binary number 1011011_2 is

$$1011011_2 = 1\times2^6 + 0\times2^5 + 1\times2^4 + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$$
$$= 64 + 16 + 8 + 2 + 1 = 91_{10}$$

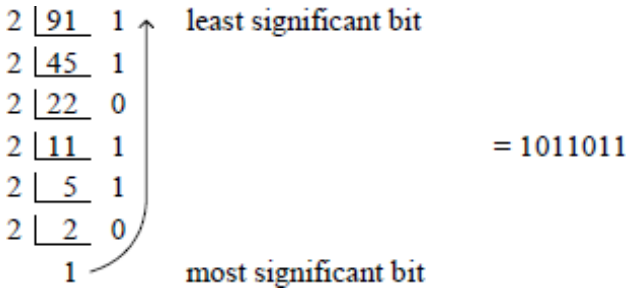
To get the decimal value, the least significant bit (in this case, the rightmost 1) is multiplied with 2^0. The next bit to the left is multiplied with 2^1, and so on. Finally, they are all added together to give the value 91_{10}.

Notice the subscript 10 in the decimal number 658_{10}, and the 2 in the binary number 1011011_2. This subscript is used to denote the base of the number whenever there might be confusion as to what base the number is in.

Table 1.1(a) - Numbers from 0 to 15 in binary, octal, and hexadecimal.

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

Converting a decimal number to its binary equivalent can be done by successively dividing the decimal number by 2 and keeping track of the remainder at each step. Combining the remainders together (starting with the last one) forms the equivalent binary number. For example, using the decimal number 91, we divide it by 2 to get 45 with a remainder of 1. Then we divide 45 by 2 to get 22 with a remainder of 1. We continue in this fashion until the end as shown below.



Concatenating the remainders together starting with the last one results in the binary number 1011011_2.
Binary numbers usually consist of a long string of bits. A shorthand notation for writing out this lengthy string of bits is to use either the octal or hexadecimal numbers. Since octal is base-8 and hexadecimal is base-16, both of which are a power of 2, a binary number can be easily converted to an octal or hexadecimal number, or vice versa.

Octal numbers only use the digits from 0 to 7 for the eight different combinations. When counting in octal, the number after 7 is 10 as shown in Table 1.1(a). . To convert a binary number to octal, we simply group the bits into groups of threes starting from the right. The

reason for this is because $8 = 2^3$. For each group of three bits, we write the equivalent octal digit for it. For example, the conversion of the binary number $1\ 110\ 011_2$ to the octal number 163_8 is shown below:

$$\begin{array}{c} \underline{001\ 110\ 011} \\ 1\quad 6\quad 3 \end{array}$$

Since the original binary number has seven bits, we need to extend it with two leading zeros to get three bits for the leftmost group. Note that when we are dealing with negative numbers, we may require extending the number with leading ones instead of zeros.

Converting an octal number to its binary equivalent is just as easy. For each octal number, we write down the equivalent three bits. These groups of three bits are concatenated together to form the final binary number. For example, the conversion of the octal number 5724_8 to the binary number $101\ 111\ 010\ 100_2$ is shown below.

$$\begin{array}{c} 5\quad 7\quad 2\quad 4 \\ 101\ 111\ 010\ 100 \end{array}$$

The decimal value of an octal number can be found just like for a binary or decimal number except that we raise the base number 8 to a power instead. For example, the octal number 5724_8 has the value

$$5724_8 = 5\times8^3 + 7\times8^2 + 2\times8^1 + 4\times8^0 = 2560 + 448 + 16 + 4 = 3028_{10}$$

Hexadecimal numbers are treated basically the same way as octal numbers except with the appropriate changes to the base. Hexadecimal (or hex for short) numbers use base-16, and thus require 16 different digit symbols as shown in Table 1.1(a). Converting binary numbers to hexadecimal numbers involve grouping the bits into groups of fours since $16 = 2^4$. For example, the conversion of the binary number $110\ 1101\ 1011_2$ to the hexadecimal number $6DB_{16}$ is shown below. Again, we need to extend it with a leading zero to get four bits for the leftmost group.

$$\begin{array}{c} \underline{0110\ 1101\ 1011} \\ 6\quad D\quad B \end{array}$$

To convert a hex number to a binary number, we write down the equivalent four bits for each hex digit, and then concatenate them together to form the final binary number. For example, the conversion of the hexadecimal number $5C4A_{16}$ to the binary number $0101\ 1100\ 0100\ 1010_2$ is shown below.

$$\begin{array}{c} 5\quad C\quad 4\quad A \\ 0101\ 1100\ 0100\ 1010 \end{array}$$

The following example shows how the decimal value of the hexadecimal number $C4A_{16}$ is evaluated.

$$C4A_{16} = C\times16^2 + 4\times16^1 + A\times16^0 = 12\times16^2 + 4\times16^1 + 10\times16^0 = 3072 + 64 + 10 = 3146_{10}$$

Binary Coded Decimal (BCD**)** is a means of encoding decimal numbers. If each digit of a decimal number is represented by its binary equivalent, this produces a code called BCD. Since a decimal digit can be as large as 9, 4 bits are also required to code each digit. To illustrate the BCD code, take a decimal number such as 874. Each digit is changed to its binary equivalent as follows

8 7 4 (decimal)
1000 0111 0100 (BCD)

The BCD code, then represents each digit of the decimal number by a 4-bit binary number. Clearly, only the 4-bit binary numbers from 0000 through 1001 are used. The BCD does not use the numbers 1010, 1011, 1100, 1101, 1110 and 1111. In other words, only 10 of the 16 possible 4-bit binary code groups are used. If any of these "forbidden" 4-bit numbers ever occur in a machine using the BCD code, it is usually an indication that an error has occurred.

Note: It is important to realize that BCD is not another number system like binary, octal, decimal and hexadecimal. It is in fact, the decimal system with each digit encoded in its binary equivalent.

3.1 Binary Addition

The addition of two binary numbers is performed in exactly the same manner as the addition of decimal numbers. In fact, binary addition is simpler, since there are fewer cases to learn. The general steps in performing decimal addition are also followed in binary addition. However, only four cases can occur in adding the two binary digits (bits) in any position. They are:

0 + 0 = 0
1 + 0 = 1
1 + 1 = 10 = 0 + carry of 1 into next position
1 + 1 + = 11 = 1 + carry of 1 into next position

The last case occurs when the two bits in a certain position are 1 and there is a carry from the previous position. Here are several examples of the addition of two binary numbers:

011 (3)	1001 (9)	11.011 (3.375)
+110 (6)	+1111 (15)	+10.110 (2.750)
1001 (9)	11000 (24)	110.001 (6.125)

It is not necessary to consider the addition of more than two binary numbers at a time, because in all digital systems the circuitry that actually performs the addition can handle only two numbers at a time. When more than two numbers are added, the first two are added together and then their sum is added to the third number; and so on. This is not a serious drawback, since modern digital computers can typically perform an addition operation in a few microseconds.

Try the following:

Add the following pairs of binary numbers: (a) 10110 + 00111 (b) 011.101 + 010.010

3.2 Binary Switch

Besides the fact that we are working only with binary values, digital circuits are easy to understand because they are based on one simple idea of turning a switch on or off to obtain either one of the two binary values. Since the switch can be in either one of two states (on or off), we call it a **binary switch**, or just a **switch** for short. The switch has three connections: an input, an output, and a control for turning the switch on or off as shown in Figure 1.1(a). When the switch is opened as in (a), it is turned off and nothing gets through from the input to the output. When the switch is closed as in (b), it is turned on, and whatever is presented at the input is allowed to pass through to the output.

Figure 1.1(a): Binary switch: (a) opened or off; (b) closed or on.

Uses of the binary switch idea can be found in many real world devices. For example, the switch can be an electrical switch with the input connected to a power source and the output connected to a siren *S* as shown in Figure 1.1(b)

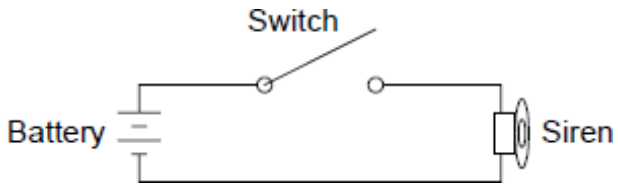


Figure 1.1(b): A siren controlled by a switch.

When the switch is closed, the siren turns on. The usual convention is to use a 1 to mean "on" and a 0 to mean "off." Therefore, when the switch is closed, the output is a 1 and the siren will turn on. We can also use a variable, *x*, to denote the state of the switch. We can let $x = 1$ to mean the switch is closed and $x = 0$ to mean the switch is opened. Using this convention, we can describe the state of the siren *S* in terms of the variable *x* using a simple logic expression. Since

$S = 1$ if $x = 1$ and $S = 0$ if $x = 0$,

we can write $S = x$

This logic expression describes the output *S* in terms of the input variable *x*.

Two binary switches can be connected together either in series or in parallel as shown in Figure 1.1(c) .

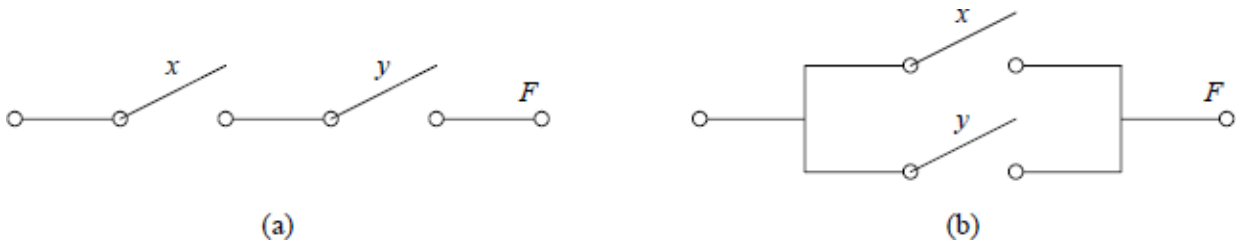


Figure 1.1(c): Connection of two binary switches: (a) in series; (b) in parallel.

If two switches are connected in series as in (a), then both switches have to be on in order for the output *F* to be a 1. In other words,

$F = 1$ if $x = 1$ AND $y = 1$. If either *x* or *y* is off, or both are off, then $F = 0$.

Translating this into a logic expression, we get

$F = x$ AND y

Hence, two switches connected in series give rise to the logical **AND** operator. In a Boolean function the AND operator is either denoted with a dot (**.**) or no symbol at all. Thus we can rewrite the above expression as

$F = x \bullet y$

Or simply as

$F=xy$

If we connect two switches in parallel as in (b), then only one switch needs to be on in order for the output F to be a 1. In other words, $F = 1$ if either $x = 1$, or $y = 1$, or both x and y are 1's. This means that $F = 0$ only if both x and y are 0's. Translating this into a logic expression, we get

$F = x$ OR y

and this gives rise to the logical **OR** operator. In a Boolean function, the OR operator is denoted with a plus symbol (+).

Thus we can rewrite the above expression as

$F = x + y$

In addition to the AND and OR operators, there is another basic logic operator ¯ the **NOT** operator, also known as the **INVERTER**. Whereas, the AND and OR operators have multiple inputs, the NOT operator has only one input and one output. The NOT operator simply inverts its input, so a 0 input will produce a 1 output, and a 1 becomes a 0.

In a Boolean function, the NOT operator is either denoted with an apostrophe symbol (′) or a bar on top (¯) as in

$F = x'$

When several operators are used in the same expression, the precedence given to the operators are, from highest to lowest, NOT, AND, and OR. The order of evaluation can be changed by means of using parenthesis. For example, the expression

$F = xy + z'$

means (x and y) or (not z), and the expression

$F = x(y + z)'$

3.3 Truth Tables

The operation of the AND, OR, and NOT logic operators can be formally described by using a **truth table** as shown in Table 1.1(b). A truth table is a two-dimensional array where there is one column for each input and one column for each output (a circuit may have more than one output). Since we are dealing with binary values, each input can be either a 0 or a 1. We simply enumerate all possible combinations of 0's and 1's for all the inputs.

Usually, we want to write these input values in the normal binary counting order. With two inputs, there are 2^2 combinations giving us the four rows in the table. The values in the output column are determined from applying the corresponding input values to the functional operator. For the AND truth table in (a) of Table 1.1(b), $F = 1$ only when x and y are both 1, otherwise, $F = 0$. For the OR truth table (b), $F = 1$ when either x or y or both is a 1, otherwise $F = 0$.

For the NOT truth table, the output F is just the inverted value of the input x.

Table 1.1(b): Truth tables for the three basic logical operators: (a) AND; (b) OR; (c) NOT.

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

(a)

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

(b)

x	F
0	1
1	0

(c)

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

3.4 Boolean Algebra

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

Let $B = \{0, 1\}$ be the Boolean algebra whose elements are one of the two values, 0 and 1. We define the operations AND (•), OR (+), and NOT (′) for the elements of B by the axioms in (a) of Table 1.1(c). These axioms are simply the definitions for the AND, OR, and NOT operators. A variable x is called a *Boolean variable* if x takes on only values in B, i.e. either 0 or 1. Consequently, we obtain the theorems in (b) of Table 1.1(c) for single variable and (c) for two and three variables.

Theorems in (b) of Table 1.1(c) can be proved easily by substituting the binary values into the expressions and using the axioms. For example, to show that Theorem 6a is true, we substitute 0 into x to get axiom 3a, and substitute 1 into x to get axiom 2a.

To prove the theorems in Table 1.1(c), we can use either one of two methods: 1) use a truth table, or 2) use axioms and theorems that have already been proven. We show these two methods in the following two examples.

Table 1.1(c): Boolean algebra axioms and theorems: (a) Axioms; (b) Single variable theorems; (c) two and three variable theorems.

1a.	$0 \bullet 0 = 0$	1b.	$1 + 1 = 1$
2a.	$1 \bullet 1 = 1$	2b.	$0 + 0 = 0$
3a.	$0 \bullet 1 = 1 \bullet 0 = 0$	3b.	$1 + 0 = 0 + 1 = 1$
4a.	$0' = 1$	4b.	$1' = 0$

(a)

5a.	$x \bullet 0 = 0$	5b.	$x + 1 = 1$	Null element
6a.	$x \bullet 1 = 1 \bullet x = x$	6b.	$x + 0 = 0 + x = x$	Identity
7a.	$x \bullet x = x$	7b.	$x + x = x$	Idempotent
8a.	$(x')' = x$			Double complement
9a.	$x \bullet x' = 0$	9b.	$x + x' = 1$	Inverse

(b)

10a.	$x \bullet y = y \bullet x$	10b.	$x + y = y + x$	Commutative
11a.	$(x \bullet y) \bullet z = x \bullet (y \bullet z)$	11b.	$(x + y) + z = x + (y + z)$	Associative
12a.	$x \bullet (y + z) = (x \bullet y) + (x \bullet z)$	12b.	$x + (y \bullet z) = (x + y) \bullet (x + z)$	Distributive
13a.	$x \bullet (x + y) = x$	13b.	$x + (x \bullet y) = x$	Absorption
14a.	$(x \bullet y) + (x \bullet y') = x$	14b.	$(x + y) \bullet (x + y') = x$	Combining
15a.	$(x \bullet y)' = x' + y'$	15b.	$(x + y)' = x' \bullet y'$	DeMorgan's

(c)

Example 1.1(a): Proof of theorem using a truth table.
Theorem 12a states that $x \bullet (y + z) = (x \bullet y) + (x \bullet z)$. To prove that Theorem 12a is true using a truth table, we need to show that for every combination of values for the three variables *x*, *y*, and *z*, the left-hand side of the expression is equal to the right-hand side. The truth table below is constructed as follows:

x	y	z	$(y + z)$	$(x \bullet y)$	$(x \bullet z)$	$x \bullet (y + z)$	$(x \bullet y) + (x \bullet z)$
0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	0	1	1	1
1	1	0	1	1	0	1	1
1	1	1	1	1	1	1	1

We start with the first three columns labeled *x*, *y*, and *z*, and enumerate all possible combinations of values for these three variables. For each combination (row), we evaluate the intermediate expressions *y*+*z*, *x*•*y*, and *x*•*z* by substituting the values of *x*, *y*, and *z* into the expression. Finally, we obtain the values for the last two columns, which correspond to the left-hand side and right-hand side of Theorem 12a. The values in these two columns are identical for every combination of *x*, *y*, and *z*, therefore, we can say that Theorem 12a is true.

Example 1.1(b): Proof of theorem using axioms and theorems.

Theorem 13b states that $x + (x \bullet y) = x$
To prove that Theorem 13b is true using axioms and theorems, we can argue as follows:

$$\begin{aligned} x + (x \bullet y) &= (x \bullet 1) + (x \bullet y) && \text{by Identity Theorem 6a} \\ &= x \bullet (1 + y) && \text{by Distributive Theorem 12a} \\ &= x \bullet (1) && \text{by Null element Theorem 5b} \\ &= x && \text{by Identity Theorem 6a} \end{aligned}$$

Example 1.1(b) shows that some theorems can be derived from others that have already been proven with the truth table. Full treatment of Boolean algebra is beyond the scope of this book and can be found in the references. For our purposes, we simply assume that all the theorems are true and will just use them to show that two circuits are equivalent as depicted in the next two examples.

Example 1.1(c): Use Boolean algebra to reduce the equation $F(x,y,z) = (x' + y' + x'y' + xy)\ (x' + yz)$ as much as possible.

$$\begin{aligned} F &= (x' + y' + x'y' + xy)\ (x' + yz) \\ &= (x' \bullet 1 + y' \bullet 1 + x'y' + xy)\ (x' + yz) \\ &= (x'(y + y') + y'(x + x') + x'y' + xy)\ (x' + yz) \\ &= (x'y + x'y' + y'x + y'x' + x'y' + xy)\ (x' + yz) \\ &= (x'y + x'y' + y'x + \cancel{y'x'} + \cancel{x'y'} + xy)\ (x' + yz) \\ &= (x'(y + y') + x(y + y'))\ (x' + yz) \\ &= (x' \bullet 1 + x \bullet 1)\ (x' + yz) \\ &= (x' + x)\ (x' + yz) \\ &= 1\ (x' + yz) \\ &= (x' + yz) \end{aligned}$$

Since the expression $(x' + y' + x'y' + xy)\ (x' + yz)$ reduces down to $(x' + yz)$, therefore, we do want to implement the circuit for the latter expression rather than the former because the circuit size for the latter is much smaller.

Example 1.1(d): Show, using Boolean algebra, that the two equations $F_1 = (xy' + x'y + x' + y' + z')\ (x + y' + z)$ and $F_2 = y' + x'z + xz'$ are equivalent.

$$\begin{aligned} F_1 &= (xy' + x'y + x' + y' + z')\ (x + y' + z) \\ &= xy'x + xy'y' + xy'z + x'yx + x'yy' + x'yz + x'x + x'y' + x'z + y'x + y'y' + y'z + z'x + z'y' + z'z \\ &= xy' + xy' + xy'z + 0 + 0 + x'yz + 0 + x'y' + x'z + xy' + y' + y'z + xz' + y'z' + 0 \\ &= xy' + xy'z + x'yz + x'y' + x'z + y' + y'z + xz' + y'z' \\ &= y'(x + xz + x' + 1 + z + z') + x'z(y + 1) + xz' \\ &= y' + x'z + xz' \\ &= F_2 \end{aligned}$$

3.5 Duality Principle

Notice in Table 1.1(c) that we have listed the axioms and theorems in pairs. Specifically, we define the **dual** of a logic expression as one that is obtained by changing all + operators with • operators, and vice versa, and by changing all 0's with 1's, and vice versa. For example, the dual of the logic expression

$(x \bullet y' \bullet z) + (x \bullet y \bullet z') + (y \bullet z) + 0$

is

$(x + y' + z) \bullet (x + y + z') \bullet (y + z) \bullet 1$

The **duality principle** states that if a Boolean expression is true, then its dual is also true. Be careful in that it does not say that a Boolean expression is equivalent to its dual. For example, Theorem 5a in Table 1.1(c) says that $x \bullet 0 = 0$ is true, thus by the duality principle, its dual, $x + 1 = 1$ is also true.

However, $x \bullet 0 = 0$ is not equal to $x + 1 = 1$, since 0 is definitely not equal to 1.

We will see later that the inverse of a Boolean expression can be obtained by first taking the dual of that expression, and then complementing each Boolean variable in the resulting dual expression. In this respect, the duality principle is often used in digital logic design. Whereas an expression might be complex to implement, its inverse might be simpler, thus resulting in a smaller circuit, and inverting the final output of this circuit will produce the same result as from the original expression.

4.0 Conclusion

Boolean Expressions are equivalent expressions of the logic state of gates. Truth tables are tables which are set to list the possible inputs and find their corresponding outputs. By looking at a truth table, one is able to know the output of **any** possible combination of inputs. The **NOT** gate, the **OR** gate and the **AND** gate are three main types of logic gates. The rules for these operations may be summarized as follows:

OR	**AND**	**NOT**
0 + 0 = 0	0 . 0 = 0	0′ = 1
0 + 1 = 1	0 . 1 = 0	1′ = 0
1 + 0 = 1	1 . 0 = 0	
1 + 0 = 1	1 . 1 = 1	

Self Assessment Exercises

1. Convert 100011011011_2 to its decimal and octal equivalent

2. Convert 614_8 to decimal

3. Convert $24CE_{16}$ to decimal

5.0 Summary

In this unit, you learnt about

- The theoretical foundation for building digital logic circuits.
- The binary number system and conversion, also its addition
- How Truth Tables are used to show how circuits output responds to the various combination of logic levels at the inputs.
- Boolean algebras differ in a way from ordinary algebra in that Boolean constants and variables are allowed to have only two possible values, 0 or 1.
- The duality principle

6.0 Tutor Marked Assignment

1) Add this pair of binary numbers 10001111 + 000000001

2) For the function F = AB′C′ + AB, find the logic value of F under the conditions—
(*a*) A = 1, B = 0, C = 1; (*b*) A = 0, B = 1, C = 1;
(*c*) A = 0, B = 0, C = 0

3) Draw truth tables for the following expressions:
(a) F = AC + AB (*b*) F = AB (B + C + D′)
(*c*) Y= A (B′ + C′) (*d*) Y = (A + B + C) AB′
(*e*) F = ABC (C + D′) (*f*) F = AB + BA + C (A + B)

4) Use a truth table to show that the following expressions are true:
a) w'z' + w'xy + wx'z + wxyz = w'z' + xyz + wx'y'z + wyz
b) *z + y' + yz'* = 1
c) xy'z' + x' + xyz' = x' + z'
d) xy + x'z + yz = xy + x'z

5) Simplify the following Boolean expressions using Boolean technique:
(a) AB + A (B + C) + B (B + C) (*b*) AB(C + BD′) (AB)′
(*c*) A + AB + AB′C (*d*) (A′ + B)C + ABC
(*e*) AB′C (BD + CDE) + AC′ (*f*) BD + B (D + E) + D′ (D + F)

7.0 Further Reading and Other Resources

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

MODULE 1 - Basic Logic Operators and Logic Expressions
UNIT 2: BOOLEAN ALGEBRA AND FUNCTIONS

Contents **Pages**

1.0 Introduction

Boolean algebra is a tool for the analysis and design of digital system. Boolean algebra differs in a way from ordinary algebra in that Boolean constants and variables are allowed to have only two possible values, 0 or 1. A Boolean variable is a quantity that may, at different times, be equal to either 0 or 1.

2.0 Objectives

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

- Simplify complex logic expressions by applying the various Boolean algebra laws and rules.
- Simplify intricate Boolean equations by applying the DeMorgan's theorem.
- Understand minterms and maxterms.
- Understand the canonical, standard and non-standard forms of Boolean functions.

3.0 Boolean Theorem

Boolean theorem are rules that can help us to simplify logic expressions and logic circuits as shown in Table 1.1(c). When this is done, the reduced expression will produce a circuit that is less complex than the one which the original expression would have produced. The next section discusses in detail the DeMorgan's theorem

3.1 DeMorgan's Theorem

Two of the most important theorems of Boolean algebra were contributed by a great mathematician named DeMorgan. DeMorgan's theorems are extremely useful in simplifying expressios in which a product or sum of variables is inverted. The two theorems as in Theorem 15a and 15b are:

(i) $(x + y)' = x' \,.\, y'$

(ii) $(x \,.\, y)' = x' + y'$

The theorem in (i) says that when the OR sum of two variables is inverted, this is the same as inverting each variable individually and then ANDing these inverted variables. The (ii) says that when the AND product of two variables is inverted, this is the same as inverting each variable individually and then ORing them. Each of DeMorgan's theorems can be readily proven by checking for all possible combinations of x and y. Although these theorems have been stated in terms of single variables x and y, they are equally valid for situations where x and/or y are expressions that contain more than one variable. For example, let's apply them to the expression

$(AB + C)' = (AB)' \,.\, C'$

Note that here we treated $(AB)'$ as x and C as y. The result can be further simplified since we have a product $(AB)'$ that is inverted.

Using the theorem in (ii), the expression becomes

$$(AB')' \cdot C' = (A' + (B')') \cdot C'$$

Notice that we can replace $(B')'$ by B, so we finally have

$$(A' + B) \cdot C' = A'C' \cdot BC'$$

The final result contains only inverter signs that invert a single variable.

Class work:

Simplify the expression $z = ((A' + C) \cdot (B + D'))'$

Using (ii), we can write this as

$$z = ((A' + C)' + (B + D')'$$

We can think of this as breaking the large inverter sign down the middle and changing the AND sign (.) to an OR sign (+). Now the term $((A' + C)'$ can be simplified by applying the theorem in (i). Likewise, $(B + D')'$ can be simplified.

$$z = ((A' + C)' + (B + D')'$$

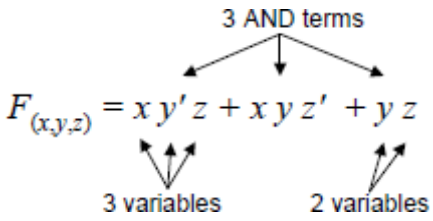
$$= ((A')' \cdot C') + (B' \cdot (D')')$$

here, we have broken the larger inverter signs down the middle and replaced the (+) with a (.). Canceling out the double inversions, we have finally

$$z = AC' + B'D$$

3.2 Boolean Function and the Inverse

As we have seen, any digital circuit can be described by a logical expression, also known as a *Boolean function*. Any Boolean functions can be formed from binary variables and the Boolean operators •, +, and ′ (for AND, OR, and NOT respectively). For example, the following Boolean function uses the three variables or literals *x*, *y*, and *z*. It has three **AND terms** (also referred to as **product terms**), and these AND terms are ORed (summed) together. The first two AND terms contain all three variables each, while the last AND term contains only two variables. By definition, an AND (or product) term is either a single variable, or two or more variables ANDed together. Quite often, we refer to functions that are in this format as a **sum-of-products** or **or-of-ands**.



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

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

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

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

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

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

or

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

or

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

or

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

Otherwise, F evaluates to a 0.

It is often more convenient to summarize the above verbal description of a function with a truth table as shown in Table 1.2(a) under the column labeled F. Notice that the four rows in the table where $F = 1$ match the four cases in the description above.

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

Table 1.2(a): Truth table for the function $F = xy'z + xyz' + yz$

The inverse of a function, denoted by F', can be easily obtained from the truth table for F by simply changing all the 0's to 1's and 1's to 0's as shown in the truth table in Table 1.2(a) under the column labeled F'. Therefore, we can write the Boolean function for F' in the sum-of-products format, where the AND terms are obtained from those rows where $F' = 1$. Thus, we get

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

To deduce F' algebraically from F requires the use of DeMorgan's Theorem twice. For example, using the same function

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

we obtain F' as follows

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

There are three things to notice about this equation for *F'*. First, *F'* is just the dual of *F* and then having all the variables inverted. Second, instead of being in a sum-of-products format, it is in a **product-of-sums** (**and-of-ors**) format where three OR terms (also referred to as sum terms) are ANDed together.

Third, from the same original function *F*, we obtained two different equations for *F'*. From the truth table, we obtained

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

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

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

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

Thus, we should also be able to express the same function $F = xy'z + xyz' + yz$ in the product-of-sums format.

We can derive it using one of two methods. For method one, we can start with *F'* and apply DeMorgan's Theorem to it just like how we obtained *F'* from *F*.

$F = F''$
$= (x'y'z' + x'y'z + x'yz' + xy'z')'$
$= (x'y'z')' \bullet (x'y'z)' \bullet (x'yz')' \bullet (xy'z')'$
$= (x+y+z) \bullet (x+y+z') \bullet (x+y'+z) \bullet (x'+y+z)$

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

$F = xy'z + xyz' + yz$
$= (x+x+y) \bullet (x+x+z) \bullet (x+y+y) \bullet (x+y+z) \bullet (x+z'+y) \bullet \sout{(x+z'+z)} \bullet$ (step 1)
$\sout{(y'+x+y)} \bullet (y'+x+z) \bullet \sout{(y'+y+y)} \bullet \sout{(y'+y+z)} \bullet \sout{(y'+z'+y)} \bullet \sout{(y'+z'+z)} \bullet$
$(z+x+y) \bullet (z+x+z) \bullet (z+y+y) \bullet (z+y+z) \bullet \sout{(z+z'+y)} \bullet \sout{(z+z'+z)}$
$= (x+y) \bullet (x+z) \bullet \sout{(x+y)} \bullet (x+y+z) \bullet (x+z'+y) \bullet (y'+x+z) \bullet \sout{(z+x+y)} \bullet \sout{(z+x)} \bullet (z+y) \bullet \sout{(z+y)}$ (step 2)
$= (x+y) \bullet (x+z) \bullet (x+y+z) \bullet (x+y+z') \bullet (x+y'+z) \bullet (z+y)$ (step 3)
$= (x+y+zz') \bullet (x+yy'+z) \bullet (x+y+z) \bullet (x+y+z') \bullet (x+y'+z) \bullet (xx'+y+z)$ (step 4)
$= (x+y+z) \bullet (x+y+z') \bullet \sout{(x+y+z)} \bullet (x+y'+z) \bullet \sout{(x+y+z)} \bullet \sout{(x+y+z')} \bullet \sout{(x+y'+z)} \bullet \sout{(x+y+z)} \bullet (x'+y+z)$ (step 5)
$= (x+y+z) \bullet (x+y+z') \bullet (x+y'+z) \bullet (x'+y+z)$

In the first step, we apply Theorem 12b (Distributive) in Unit 1 to get every possible combination of sum terms. For example, the first sum term $(x+x+y)$ is obtained from getting the first *x* from *xy'z*, the second *x* from *xyz'*, and the *y* from *yz*. The second sum term $(x+x+z)$ is obtained from getting the first *x* from *xy'z*, the second *x* from *xyz'*, and the *z* from *yz*. This is repeated for all combinations. In this step, the sum terms, such as $(x+z'+z)$, where it contains variables of the form $v + v'$ can be eliminated since $v + v' = 1$, and $1 \bullet x = x$.

In the second and third steps, duplicate variables and terms are eliminated. For example, the term (x+x+y) is equal to just (x+y+y), which is just (x+y). The term (x+z'+z) is equal to (x+1), which is equal to just 1, and therefore, can be eliminated completely from the expression.
In the fourth step, every sum term with a missing variable will have that variable added back in by using Theorems 6b and 9a, which says that $x + 0 = x$ and $yy' = 0$, therefore, $x + yy' = x$.
Step five uses the Distributive Theorem, and the resulting duplicate terms are again eliminated to give us the format that we want.
Functions that are in the product-of-sums format (such as the one shown below) are more difficult to deduce when they evaluate to a 1. For example, using

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

F' evaluates to a 1 when all three terms evaluate to a 1. For the first term to evaluate to a 1, x can be 0, or y can be 1, or z can be 0. For the second term to evaluate to a 1, x can be 0, or y can be 0, or z can be 1. Finally, for the last term, y can be 0, or z can be 0, or x can be either a 0 or a 1. As a result, we end up with many more combinations to consider, even though many of the combinations are duplicates.
However, it is easier to determine when a product-of-sums format expression evaluates to a 0. For example, using the same expression

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

F' evaluates to 0 when any one of the three OR terms is 0, since 0 AND x is 0; and this happens when $x = 1$, $y = 0$, and $z = 1$ for the first OR term,
or

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

or

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

Similarly, for a sum-of-products format expression, it is easy to evaluate when it is a 1, but difficult to evaluate when it is a 0.
These four conditions in which F' evaluates to a 0 match exactly those rows in the table where $F' = 0$. Therefore, we see that in general, the unique algebraic expression for any Boolean function can be specified by either (1) selecting the rows from the truth table where the function is a 1 and use the sum-of-products format, or (2) selecting the rows from the truth table where the function is a 0 and use the product-of-sums format.
Whatever format we decide to use, the one thing to remember is that we are always interested in only when the function (or its inverse) is equal to a 1.
Figure 1.2(a) summarizes these two formats for the function $F = xy'z + xyz' + yz$ and its inverse F'. Notice that the sum-of-products format for F is the dual with its variables inverted of the product-of-sums format for F'. Similarly, the product-of-sums format for F is the dual with its variables inverted of the sum-of-products format for F'.

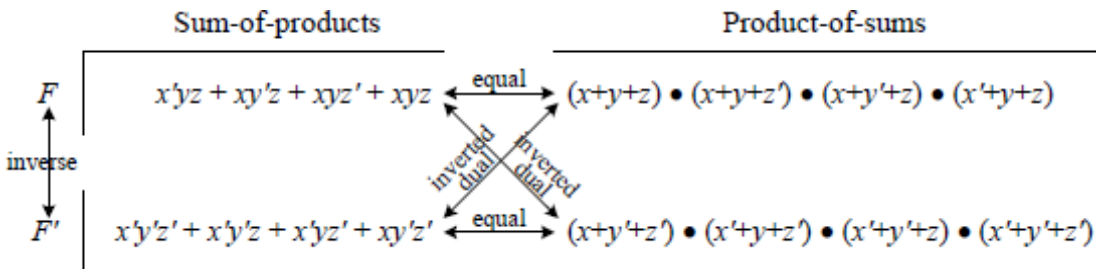


Figure 1.2(a): Relationships between the function $F = xy'z + xyz' + yz$ and its inverse F', and the sum-of-products and product-of-sums formats. The label "inverted dual" means applying the duality principle and then inverting the variables.

3.3 Minterms and Maxterms

As you recall, a product term is a term with either a single variable, or two or more variables ANDed together, and a sum term is a term with either a single variable, or two or more variables ORed together. To differentiate between a term that contains any number of variables with a term that contains *all* the variables used in the function, we use the words minterm and maxterm. We are not introducing new ideas here, rather, we are just introducing two new words and notations for defining what we have already learned.

3.3.1 Minterms

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

For example, for a function with three variables x, y, and z, the notation m_3 is used to represent the term in which the values for the variables xyz are 011 (for the subscript 3). Since we want to complement the variable whose value is a 0, and uncomplement it if it is a 1. Hence m_3 is for the minterm $x'yz$. The (a) of **Table 1.2(b)** shows the eight minterms and their notations for $n = 3$ using the three variables x, y, and z.

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

Table 1.2(b): (a) Minterms for three variables. (b) Maxterms for three variables.

x	y	z	Minterm	Notation
0	0	0	$x'y'z'$	m_0
0	0	1	$x'y'z$	m_1
0	1	0	$x'y\,z'$	m_2
0	1	1	$x'y\,z$	m_3
1	0	0	$x\,y'z'$	m_4
1	0	1	$x\,y'z$	m_5
1	1	0	$x\,y\,z'$	m_6
1	1	1	$x\,y\,z$	m_7

(a)

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

(b)

The function from the previous section

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

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

and repeated in the following truth table has the 1-minterms m_3, m_5, m_6, and m_7.

x	y	z	F	F'	Minterm	Notation
0	0	0	0	1	$x'y'z'$	m_0
0	0	1	0	1	$x'y'z$	m_1
0	1	0	0	1	$x'y\,z'$	m_2
0	1	1	1	0	$x'y\,z$	m_3
1	0	0	0	1	$x\,y'z'$	m_4
1	0	1	1	0	$x\,y'z$	m_5
1	1	0	1	0	$x\,y\,z'$	m_6
1	1	1	1	0	$x\,y\,z$	m_7

Thus, a shorthand notation for the function is

$F(x, y, z) = m_3 + m_5 + m_6 + m_7$

By just using the minterm notations, we do not know how many variables are in the original function. Consequently, we need to explicitly specify the variables used by the function as in $F(x, y, z)$. We can further simplify the notation by using the standard algebraic symbol Σ for summation. Therefore, we have

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

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

Example 1.2(a): Given the Boolean function $F(x, y, z) = y + x'z$, use Boolean algebra to convert the function to the sum-of-minterms format.

This function has three variables. In a sum-of-minterms format, all product terms must have all variables. To do so, we need to expand each product term by ANDing it with $(v + v')$ for every

missing variable *v* in that term. Since (*v* + *v'*) = 1, therefore, ANDing a product term with (*v* + *v'*) does not change the value of the term.

$F = y + x'z$
$= y(x+x')(z+z') + x'z(y+y')$ expand 1st term by ANDing it with $(x+x')(z+z')$, and 2nd term with $(y+y')$
$= xyz + xyz' + x'yz + x'yz' + \cancel{x'yz} + x'y'z$
$= m_7 + m_6 + m_3 + m_2 + m_1$
$= \Sigma(1, 2, 3, 6, 7)$ sum of 1-minterms

Example 1.2(b): Given the Boolean function $F(x, y, z) = y + x'z$, use Boolean algebra to convert the inverse of the function to the sum-of-minterms format.

$F' = (y + x'z)'$ inverse
$= y' \bullet (x'z)'$ use DeMorgan
$= y' \bullet (x+z')$ use DeMorgan
$= y'x + y'z'$ use Distributive Theorem to change to sum of products format
$= y'x(z+z') + y'z'(x+x')$ expand 1st term by ANDing it with $(z+z')$, and 2nd term with $(x+x')$
$= xy'z + xy'z' + \cancel{xy'z'} + x'y'z'$
$= m_5 + m_4 + m_0$
$= \Sigma(0, 4, 5)$ sum of 0-minterms

3.3.2 Maxterms

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

For example, for a function with three variables *x*, *y*, and *z*, the notation M_3 is used to represent the term in which the values for the variables *xyz* are 011. For maxterms, we want to complement the variable whose value is a 1, and uncomplement it if it is a 0. Hence M_3 is for the maxterm $x + y' + z'$. (b) of Figure 1.2(c) shows the eight maxterms and their notations for $n = 3$ using the three variables *x*, *y*, and *z*. We have seen that a function can also be specified as a product of sums, or more specifically, a **product of 0-maxterms**, that is, the maxterms for which the function is a 0. Just like the minterms, we use the notation **1-maxterm** to denote one-maxterm, and **0-maxterm** to denote zero-maxterm. Thus, the function

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

which is shown in the following table

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

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

$F(x, y, z) = M_0 \bullet M_1 \bullet M_2 \bullet M_4$

Again, by using the standard algebraic symbol Π for product, the notation is further simplified to

$F(x, y, z) = \Pi\,(0, 1, 2, 4)$

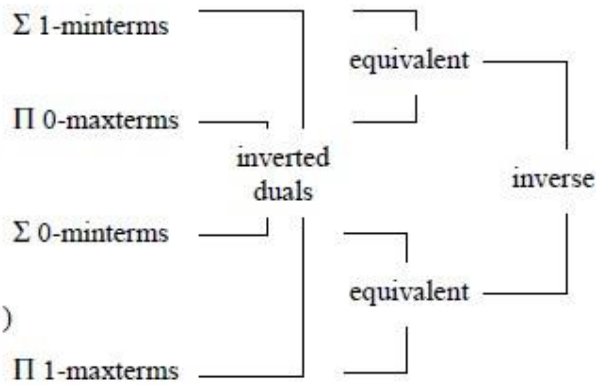
The following summarizes these relationships for the function $F = xy'z + xyz' + yz$ and its inverse. Comparing these equations with those in table 1.2(a), we see that they are identical.

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

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

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

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



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

Example 1.2(c): Given the Boolean function $F(x, y, z) = y + x'z$, use Boolean algebra to convert the function to the product-of-maxterms format.

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

$F = y + x'z$
$= y + (x'z)$
$= (y+x')(y+z)$ use Distributive Theorem to change to product of sums format
$= (y+x'+zz')(y+z+xx')$ expand 1st term by ORing it with zz', and 2nd term with xx'
$= (x'+y+z)\ (x'+y+z')\ (x+y+z)$ ~~$(x'+y+z)$~~
$= M_4 \bullet M_5 \bullet M_0$
$= \Pi(0, 4, 5)$ product of 0-maxterms

Example 1.2(d): Given the Boolean function $F(x, y, z) = y + x'z$, use Boolean algebra to convert the inverse of the function to the product-of-maxterms format.

$F' = (y + x'z)'$ inverse
$= y' \bullet (x'z)'$ use DeMorgan
$= y' \bullet (x+z')$ use DeMorgan
$= (y'+xx'+zz') \bullet (x+z'+yy')$ expand 1st term by ORing it with $xx'+zz'$, and 2nd term with yy'
$= (x+y'+z)\ (x+y'+z')\ (x'+y'+z)\ (x'+y'+z')\ (x+y+z')$ ~~$(x+y'+z')$~~
$= M_2 \bullet M_3 \bullet M_6 \bullet M_7 \bullet M_1$
$= \Pi(1, 2, 3, 6, 7)$ product of 1-maxterms

3.4 Canonical, Standard, and non-Standard Forms

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

$F = x'\,y\,z + x\,y'\,z + x\,y\,z' + x\,y\,z$
$F' = (x+y'+z') \bullet (x'+y+z') \bullet (x'+y'+z) \bullet (x'+y'+z')$

As noted from the previous section, to convert a Boolean function from one canonical form to its other equivalent canonical form, simply interchange the symbols Σ with Π, and list the index numbers that were excluded from the original form. For example, the following two expressions are equivalent

$F1(x, y, z) = \sum 3, 5, 6, 7)$
$F2(x, y, z) = \boldsymbol{\pi}\ (0, 1, 2, 4)$

To convert a Boolean function from one canonical form to its inverse, simply interchange the symbols $\sum$ with $\boldsymbol{\pi}$, and list the same index numbers from the original form. For example, the following two expressions are inverses

$F1(x, y, z) = \sum 3, 5, 6, 7)$
$F2(x, y, z) = \boldsymbol{\pi}\ (3, 5, 6, 7)$

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

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

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

common variable *x* from the first two terms, we get the following expression, which is in a non-standard form.

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

4.0 Conclusion

The switching functions can be expressed with Boolean equations. Complex Boolean equations can be simplified by a new kind of algebra, which is popularly called Switching Algebra or Boolean algebra. When a Boolean expression is implemented with logic gates, each literal in the function is designated as input to the gate. The literal may be a primed or unprimed variable.
Minimization of the number of literals and the number of terms leads to less complex circuits as well as less number of gates, which should be a designer's aim. There are several methods to minimize the Boolean function.

Self Assessment Exercise

Use DeMorgan's theorems to convert the expression $z = (A + B) \cdot (C')'$ to one that has only single-variable inversions.

5.0 Summary

In this unit, you learnt about:

- How to simplify Boolean equations by applying the DeMorgan's theorem.
- The minterms and maxterms.
- The canonical, standard and non-standard forms of Boolean functions.

6.0 Tutor Marked Assignment

1. What is meant by duality in Boolean algebra?
2. State DeMorgan's theorem.
3. Prove the following using Boolean theorems:
   (a) (A + C)(A + D)(B + C)(B + D) = AB + CD
   (*b*) (A′ + B′ + D′) (A′ + B + D′) (B + C + D) (A + C′) (A + C′ + D) = A′C′D + ACD′+ BC′D′
4. Find the canonical sum of products and product of sums expression for the function
   F = X1X2X3 + X1X3X4 + X1X2X4.
5. (*a*) Convert Y = ABCD + A′BC + B′C′ into a sum of minterms by algebraic method.
   (b) Convert Y = AB + B′CD into a product of maxterms by algebraic method.

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 3 - Logic Gates and Circuit Diagrams

Contents **Pages**

1.0 Introduction

As Boolean functions are expressed in terms of AND, OR, and NOT operations, it is easier to implement the Boolean functions with some basic types of gates. However, for all practical purposes, it is possible to construct other types of logic gates. **Logic gates** are the actual physical implementations of the logical operators discussed in the previous units.

2.0 Objectives

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

- Describe the operation of and construct the truth tables for the AND, NAND, OR and NOR gates.
- Discuss the implementation of logical operators with Logic gates
- Understand the construction of Circuit diagrams

3.0 Logic Gates

Logic gates are the actual physical implementations of the logical operators discussed in the previous units. Transistors, acting as tiny electronic binary switches are connected together to form these gates.

9.1 Types of Logic Gates

There are basically the AND gate, the OR gate, and the NOT gate (also called the INVERTER) for the corresponding AND, OR, and NOT logical operators. These gates form the basic building blocks for all digital logic circuits. The name “gate” comes from the fact that these devices operate like a door or gate to let or not to let things (in our case, current) through.
In drawing digital circuit diagrams, also called **schematic diagrams**, or just **schematics**, we use special **logic symbols** to denote these gates as shown in Figure 1.3(a).

3.1.1 AND gate

The AND gate, or specifically, the 2-input AND gate, in (a) of Figure 1.3(a) has two input connections coming in from the left and one output connection going out on the right. The AND gate output is equal to the AND product of the logic inputs. In other words, the AND gate is a circuit that operates such that its output is high only when all its inputs are high. For all other cases the AND gate output is low.

3.1.2 OR gate

Similarly, the 2-input OR gate in (b) of Figure 1.3(a) has two input connections and one output connection. In digital circuitry, an OR gate is a circuit that has two or more inputs and whose output is equal to the OR sum of the inputs. The OR gate operates such thatits output is high (logic 1), if either input or both are at alogic 1 level. The OR gate output will be low (logic 0) only if all its inputs are at logic 0. This same idea can be extended to more than two inputs.

3.1.3 NOT circuit (INVERTER)

The INVERTER in (c) of Figure 1.3(a) has one input coming from the left and one output going to the right. In other words, it always has only a single input, and its output logic level is always opposite to the logic level of this input.

The outputs from these gates, of course, are dependent on their inputs, and are defined by their logical functions.

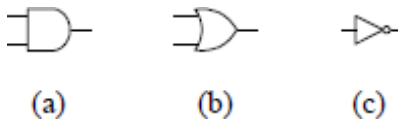


Figure 1.3(a) Logic symbols for the three basic logic gates: (a) 2-input AND; (b) 2-input OR; (c) NOT.

Sometimes, an AND gate or an OR gate with more than two inputs are needed. Hence, in addition to the 2-input AND and OR gates, there are 3-input, 4-input, or as many inputs as are needed of the AND and OR gates. In practice, however, the number of inputs is limited to a small number, such as five. The logic symbols for some of these gates are shown in (a) to (d) of Figure 1.3(b).

There are several other gates that are variants of the three basic gates that are also often used in digital circuits.
They are the NAND gate, the NOR gate, the XOR gate, and the XNOR gate. The NAND gate is derived from an AND gate and the INVERTER connected in series so that the output of the AND gate is inverted. The name "NAND" comes from the description "Not AND." Similarly, the NOR gate is the OR gate with its output inverted. The XOR, or eXclusive OR gate is like the OR gate except that when both inputs are 1, the output is a 0 instead. The XNOR, or eXclusive NOR gate is just the inverse of the XOR gate for when there are an even number of inputs (like 2 inputs). When there are an odd number of inputs (like 3 inputs), the XOR is the same as the XNOR. The logic symbols and their truth tables for some of these gates are shown in Figure 1.3(b) and Table 1.3(a) respectively.
Notice, in Table 1.3(a), the use of the little circle or bubble at the output of some of the logic symbols. This bubble is used to denote the inverted value of a signal. For example, the NAND gate is the inverse of the AND gate, thus the NAND gate logic symbol is the same as the AND gate logic symbol except that it has the extra bubble at the output.

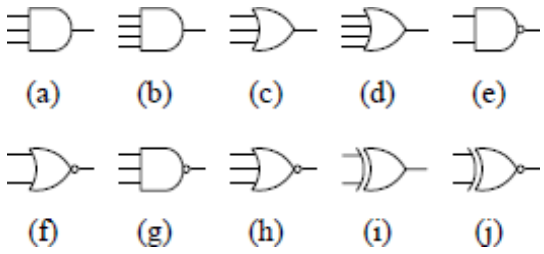


Figure 1.3(b) Logic symbols for: (a) 3-input AND; (b) 4-input AND; (c) 3-input OR; (d) 4-input OR; (e) 2-input NAND; (f) 2-input NOR; (g) 3-input NAND; (h) 3-input NOR; (i) 2-input XOR; (j) 2-input XNOR.

Table 1.3(a): Truth tables for: 2-input NAND; 2-input NOR; 2-input XOR; 2-input XNOR; 3-input AND; 3-input OR; 3-input NAND; 3-input NOR; 3-input XOR; 3-input XNOR.

x	y	2-NAND $(x \bullet y)'$	2-NOR $(x+y)'$	2-XOR $x \oplus y$	2-XNOR $x \odot y$
0	0	1	1	0	1
0	1	1	0	1	0
1	0	1	0	1	0
1	1	0	0	0	1

x	y	z	3-AND $x \bullet y \bullet z$	3-OR $x+y+z$	3-NAND $(x \bullet y \bullet z)'$	3-NOR $(x+y+z)'$	3-XOR $x \oplus y \oplus z$	3-XNOR $x \odot y \odot z$
0	0	0	0	0	1	1	0	0
0	0	1	0	1	1	0	1	1
0	1	0	0	1	1	0	1	1
0	1	1	0	1	1	0	0	0
1	0	0	0	1	1	0	1	1
1	0	1	0	1	1	0	0	0
1	1	0	0	1	1	0	0	0
1	1	1	1	1	0	0	1	1

The notations used for these gates in a logical expression are $(xy)'$ for the 2-input NAND gate, $(x+y)'$ for the 2-input NOR gate, $x \oplus y$ for the XOR gate, and $x \cdot \odot$ for the XNOR gate.

Looking at the truth table for the 2-input XOR gate, we can derive the equation for the 2-XOR gate as

$$x \oplus y = x'y + xy'$$

Similarly, the equation for the 2-input XNOR gate as derived from the 2-XNOR truth table is

$$x \odot y = x'y' + xy$$

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 $B \cdot C$.

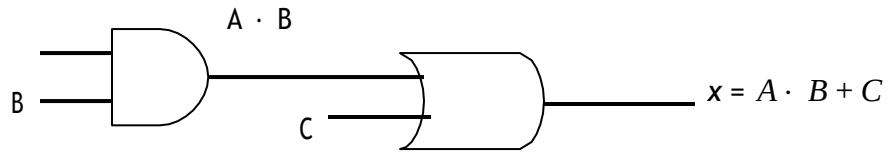


Figure 1.3(c): Logic Circuit with its Boolean expression

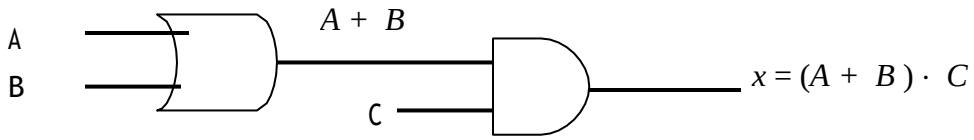


Figure 1.3(d): Logic Circuit whose expression requires parentheses

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.

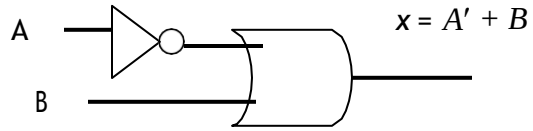


Figure 1.3(e): Circuit using INVERTERS

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.

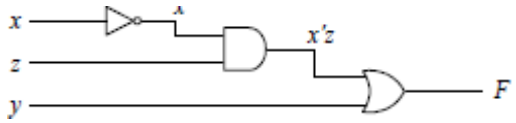
Any logic circuit, no matter how complex, may be completely described using the Boolean operations previously defined , because the OR gate, AND gate, and NOT circuit are the basic building blocks of digital system. For example, consider the circuit in Example 1.3(a). This circuit has 3 inputs x, z, 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*.

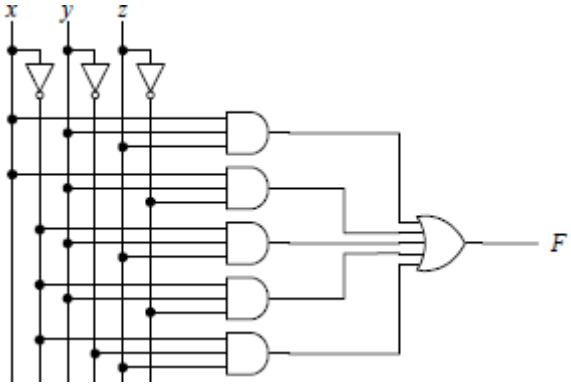


Example 1.3(b): Draw the circuit diagram for the equation:

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

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.

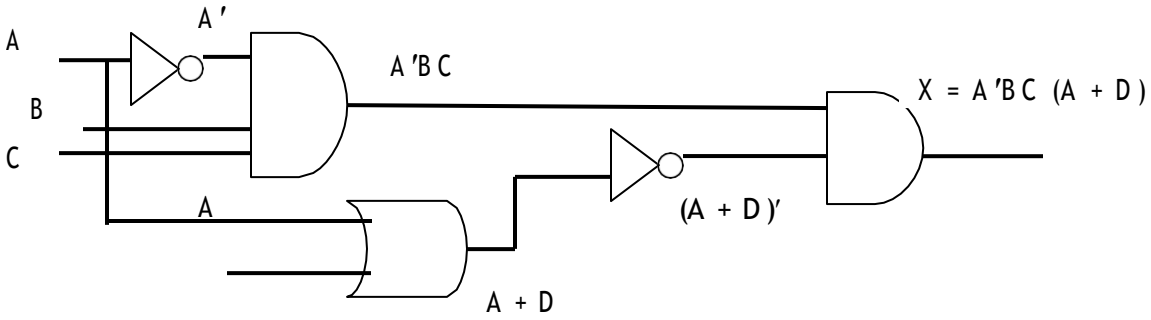
5.0 Summary

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).

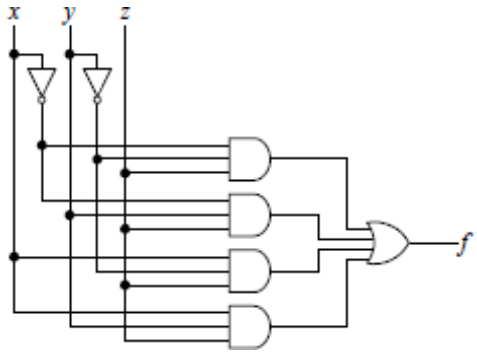


Figure 1.4(a): Combinational circuit truth table

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)

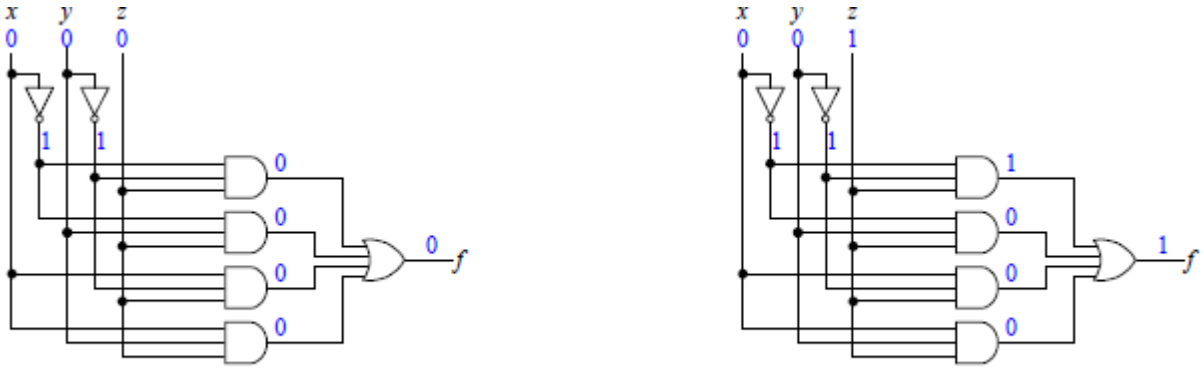


Figure 1.4(b): (c) circuit annotated with the input values *xyz* =000; (d) circuit annotated with the input values *xyz* = 001

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*:

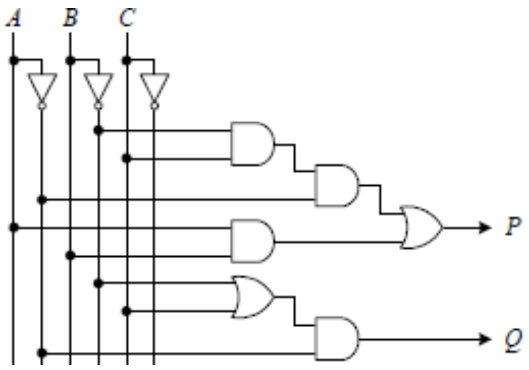


Figure 1.4(b): A circuit diagram

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.

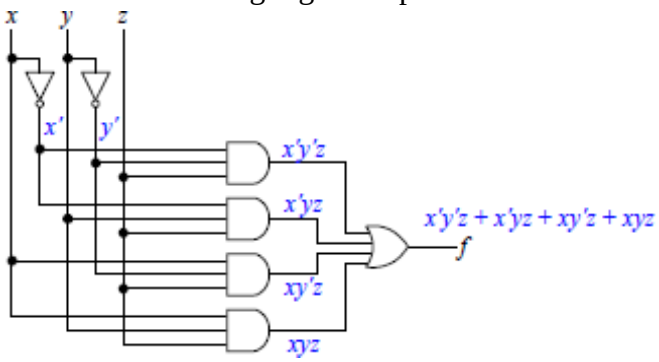
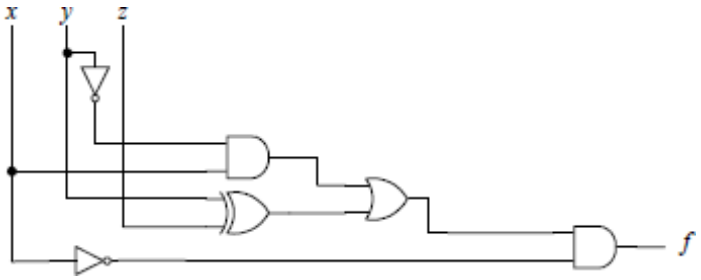


Figure 1.4(d): The annotated circuit for expression *f*

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:

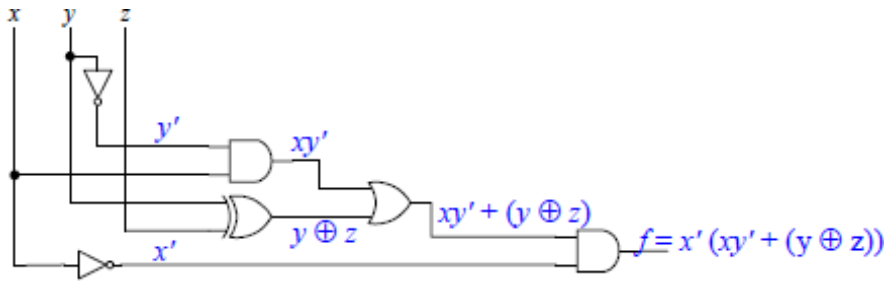


Figure 1.4(e): The annotated circuit for Example 1.4(b)

The Boolean function for the circuit is the final equation, $f = x'(xy' + (y \oplus z))$, 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

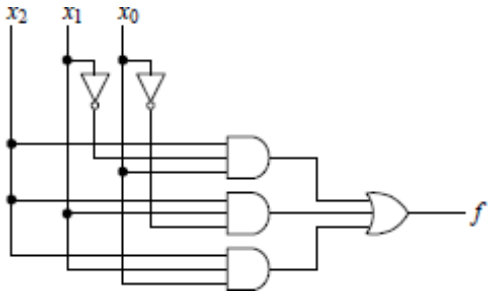
In constructing the circuit, we are interested only in when the output is a 1 (i.e., when the function f is a 1).

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).

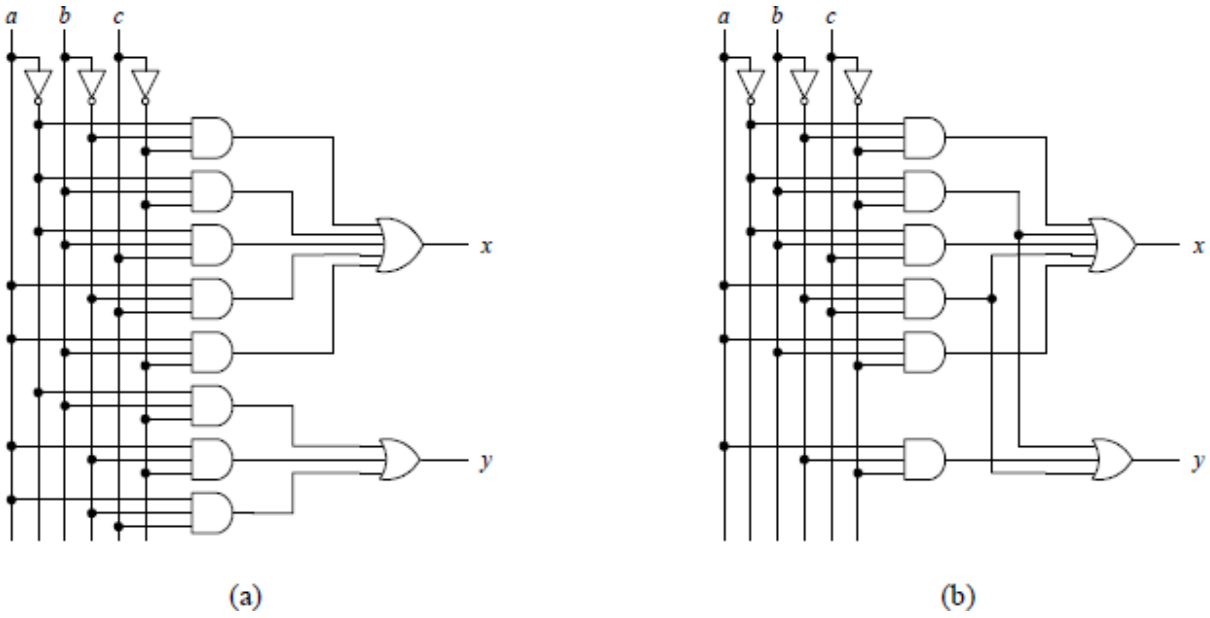


Figure 1.4(f): Combinational circuit for Example 1.4(c): (a) no reduction; (b) with reduction.

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.

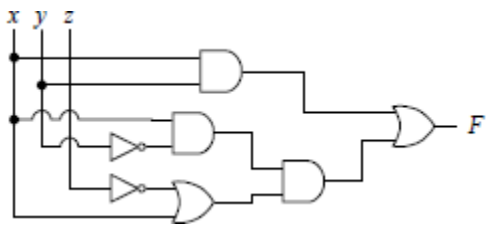
5.0 Summary

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.

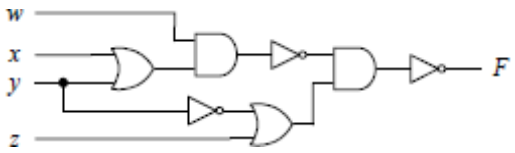
6.0 Tutor Marked Assignment

1) Derive the truth table for the following circuits:

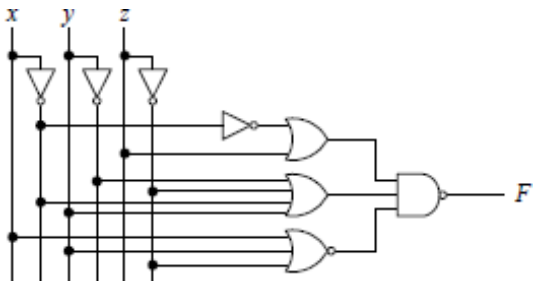
a)



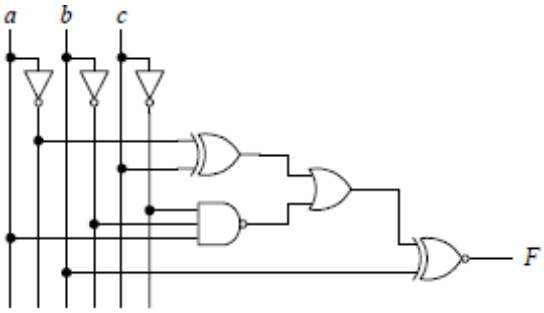
b)



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.

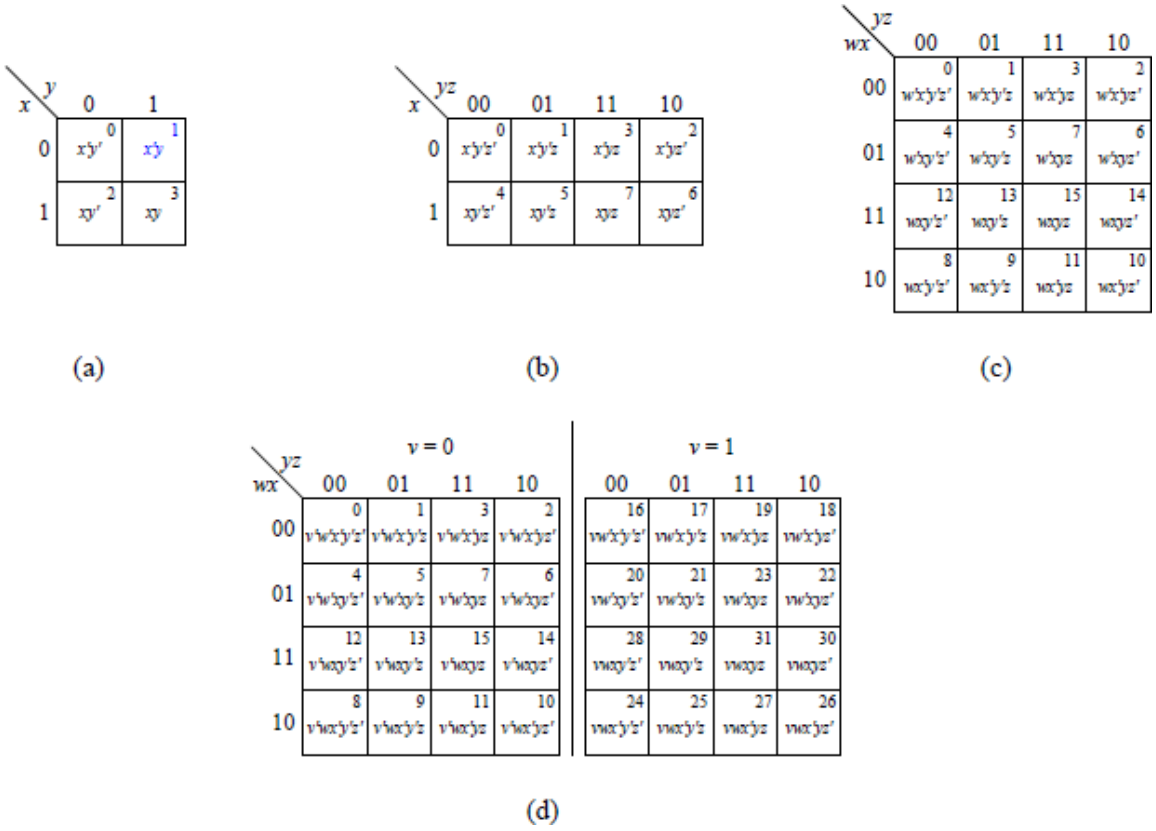
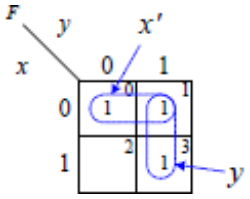


Figure 1.5(a): Karnaugh maps for: (a) 2 variables; (b) 3 variables; (c) 4 variables; (d) 5 variables.

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 $x'z'$.

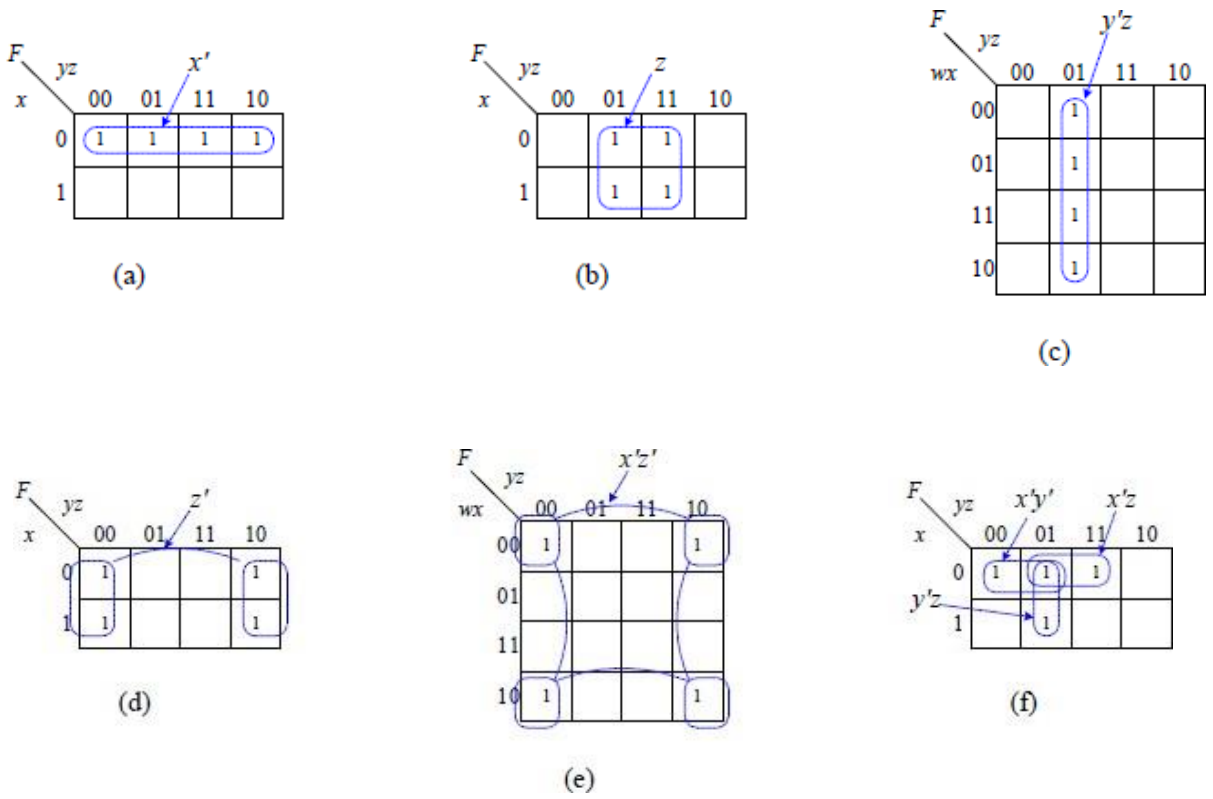


Figure 1.5(b): Examples of K-maps with 2-subcubes: (a) and (b) 3-variable; (c) 4-variable; (d) 3-variable with wrap around subcube; (e) 4-variable with wrap around subcube; (f) four adjacent minterms that cannot form one 2-subcube.

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.

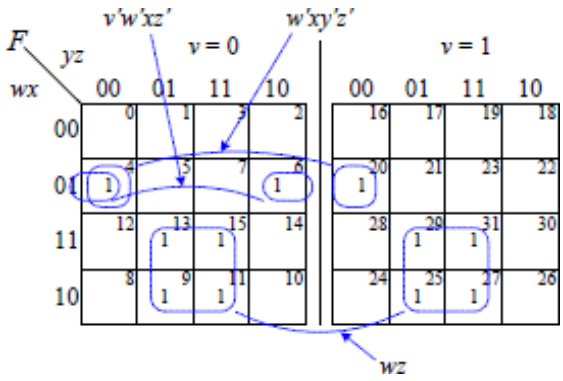


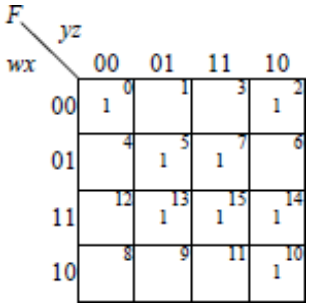
Figure 1.5(c): A 5-variable K-map with wrap-around subcubes.

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:

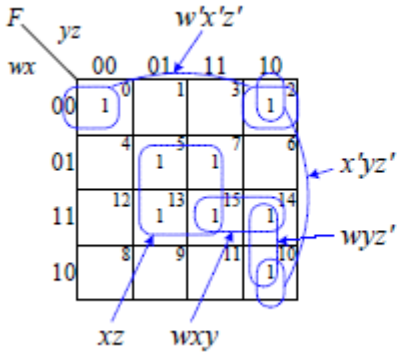
1. Draw the appropriate K-map for the given function and place a 1 in the squares that correspond to the function's 1-minterms.
2. For each 1-minterm, find the largest subcube that covers this 1-minterm. This largest subcube is known as a prime implicant (PI). By definition, a **prime implicant** is 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:

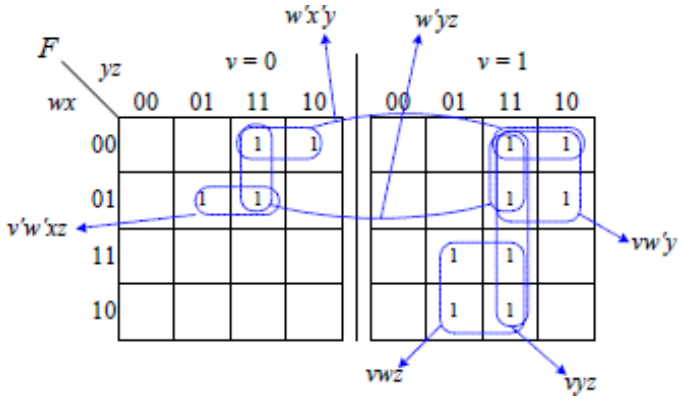
$$F = w'x'z' + xz + wyz'$$

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

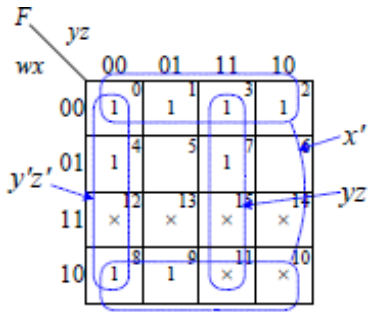
There are times when a function is not specified fully. In other words, there are some minterms for the function where we do not care whether their values are a 0 or a 1. When drawing the K-map for these "**don't-care**" minterms, we assign an "×" in that square instead of a 0 or a 1. Usually, a function 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,

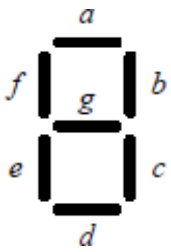
$$F = x' + yz + y'z'$$

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

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

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.

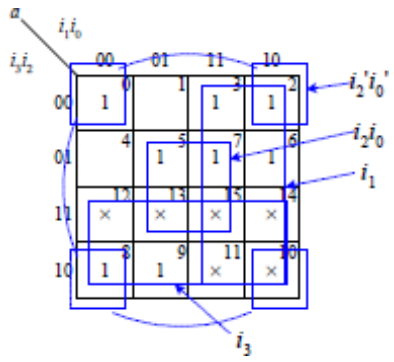
For each input combination, the corresponding digit to display on the 7-segment LED is shown in the "Display" column. The segments that need to be turned on for that digit will have a 1 while the segments that need to be turned off for that digit will have a 0. For example, for the 4-bit input 0000, 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).

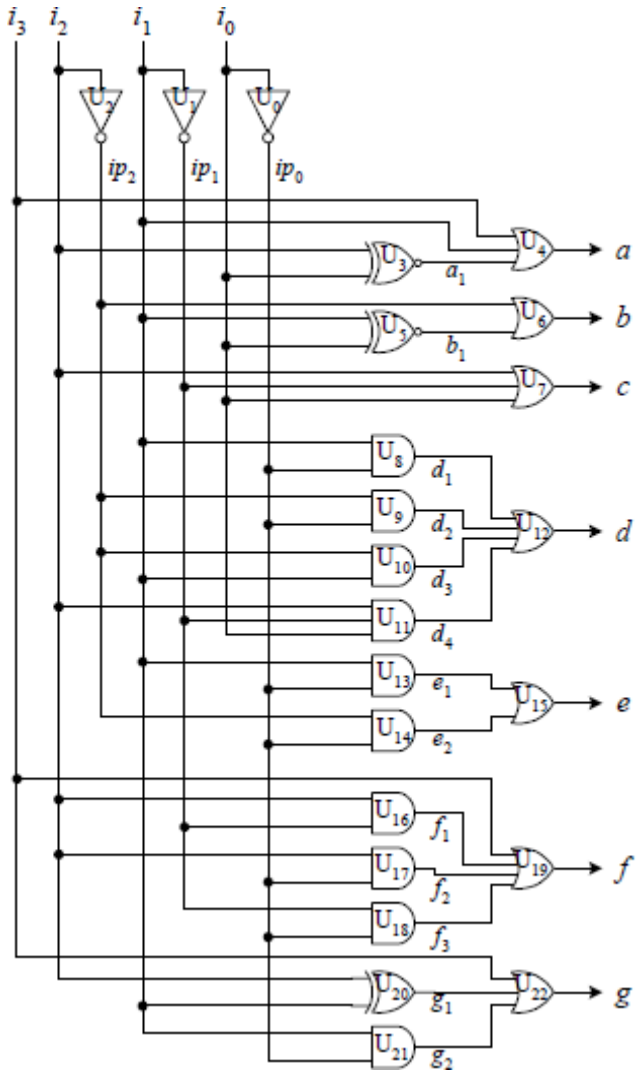


Figure 1.5(d): Circuit for the BCD to 7-segment decoder.

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.

3.1 Bistable Element

Let us look at the inverter. If you provide the inverter input with a 1, the inverter will output a 0. If you do not provide the inverter with an input (that is neither a 0 nor a 1), the inverter will not have a value to output. If you want to construct a memory circuit using the inverter, you would want the inverter to continue to output the 0 even after you remove the 1 input. In order for the inverter to continue to output a 0, you need the inverter to self-provide its own input. In other words, you want the output to feed back the 0 to the input. However, you cannot connect the output of the inverter directly to its input, because you will have a 0 connected to a 1 and so creating a short circuit. The solution is to connect two inverters in series, as shown in Figure 2.1(a). This circuit is called a **bistable element**, and it is the simplest memory circuit. The bistable element has two symmetrical nodes labeled Q and Q', both of which can be viewed as either an input or an output signal. Since Q and Q' are symmetrical, we can arbitrarily use Q as the state variable, so that the state of the circuit is the value at Q. Let us assume that Q originally has the value 0 when power is first applied to the circuit. Since Q is the input to the bottom inverter, therefore, Q' is a 1. A 1 going to the input of the top inverter will produce a 0 at the output Q, which is what we started off with.

Hence, the value at Q will remain at a 0 indefinitely. Similarly, if we power-up the circuit with $Q = 1$, we will get $Q' = 0$, and again, we get a stable situation with Q remaining at a 1 indefinitely. Thus, the circuit has two stable states: $Q = 0$ and $Q = 1$; hence, the name "bistable."

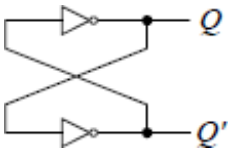


Figure 2.1(a) Bistable element circuit.

We say that the bistable element has memory because it can remember its state (i.e., keep the value at Q constant) indefinitely. Unfortunately, we cannot change its state (i.e., cannot change the value at Q). We cannot just input a different value to Q, because it will create a short circuit by connecting a 0 to a 1. For example, let us assume that Q is currently 0. If we want to change the state, we need to set Q to a 1, but in so doing we will be connecting a 1 to a 0, thus creating a short. Another way of looking at this problem is that we can think of both Q and Q' as being the primary outputs, which means that the circuit does not have any external inputs. Therefore, there is no way for us to input a different value.

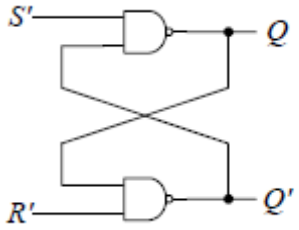
3.2 S-R Latch

In order to change the state for the bistable element, we need to add external inputs to the circuit. The simplest way to add extra inputs is to replace the two inverters with two NAND gates, as shown in (a) of Figure 2.1(b). This circuit is called an **SR latch**. In addition to the two outputs Q and Q', there are two inputs S' and R' for *set* and *reset*, respectively. Just like the bistable element, the SR latch can be in one of two states: a set state when $Q = 1$, or a reset state when $Q = 0$. Following the convention, the primes in S and R denote that these inputs are active-low (i.e., a 0 asserts them, and a 1 de-asserts them).

To make the SR latch go to the set state, we simply assert the S' input by setting it to 0 (and de-asserting R'). It doesn't matter what the other NAND gate input is, because 0 NAND anything gives a 1, hence $Q = 1$, and the latch is set. If S' remains at 0 so that Q (which is connected to one input of the bottom NAND gate) remains at 1, and if we now de-assert R' (i.e., set R' to a 1) then the output of the bottom NAND gate will be 0, and so, $Q' = 0$. This situation is shown in (d) of Figure 2.1(b) at time t_0. From this current situation, if we now de-assert S' so that $S' = R' = 1$, the latch will remain in the set state because Q' (the second input to the top NAND gate) is 0, which will keep $Q = 1$, as shown at time t_1. At time t_2, we reset the latch by making $R' = 0$ (and $S' = 1$). With R' being a 0, Q' will go to a 1. At the top NAND gate, 1 NAND 1 is 0, thus forcing Q to go to 0. If we de-assert R' next so that, again, we have $S' = R' = 1$, this time the latch will remain in the reset state, as shown at time t_3.

Notice the two times (at t_1 and t_3) when both S' and R' are de-asserted (i.e., $S' = R' = 1$). At t_1, Q is at a 1; whereas, at t_3, Q is at a 0. Why is this so? What is different between these two times? The difference is in the value of Q immediately before those times. The value of Q right before t_1 is 1; whereas, the value of Q right before t_3 is 0.

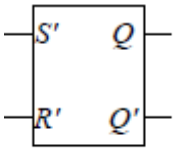
When both inputs are de-asserted, the SR latch remembers its previous state. Previous to t_1, Q has the value 1, so at t_1, Q remains at a 1. Similarly, previous to t_3, Q has the value 0, so at t_3, Q remains at a 0.



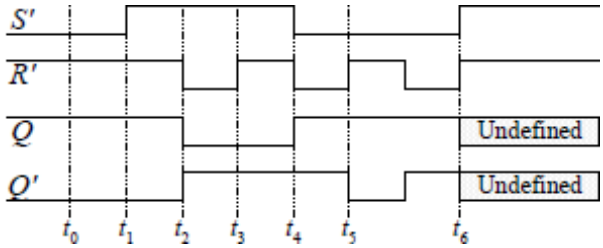
(a)

S'	R'	Q	Q_{next}	Q_{next}'
0	0	×	1	1
0	1	×	1	0
1	0	×	0	1
1	1	0	0	1
1	1	1	1	0

(b)



(c)



(d)

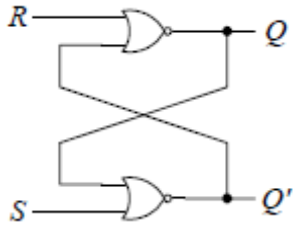
Figure 2.1(b): SR latch: (a) circuit using NAND gates; (b) truth table; (c) logic symbol; (d) sample trace.

If both S' and R' are asserted (i.e., $S' = R' = 0$), then both Q and Q' are equal to a 1, as shown at time t_4, since 0 NAND anything gives a 1. Note that there is nothing wrong with having Q equal

to Q'. It is just because we named these two points Q and Q' that we don't like them to be equal. However, we could have used another name say, P instead of Q'.
If one of the input signals is de-asserted earlier than the other, the latch will end up in the state forced by the signal that is de-asserted later, as shown at time t_5. At t_5, R' is de-asserted first, so the latch goes into the set state with $Q = 1$, and $Q' = 0$.
A problem exists if both S' and R' are de-asserted at *exactly* the same time, as shown at time t_6. Let us assume for a moment that both gates have exactly the same delay and that the two wire connections between the output of one gate to the input of the other gate also have exactly the same delay. Currently, both Q and Q' are at a 1. If we set S' and R' to a 1 at exactly the same time, then both NAND gates will perform a 1 NAND 1 and will both output a 0 at exactly the same time. The two 0's will be fed back to the two gate inputs at exactly the same time, because the two wire connections have the same delay. This time around, the two NAND gates will perform a 1 NAND 0 and will both produce a 1 again at exactly the same time. This time, two 1's will be fed back to the inputs, which again will produce a 0 at the outputs, and so on and on. This oscillating behavior, called the *critical race*, will continue indefinitely until one outpaces the other. If the two gates do not have exactly the same delay then, the situation is similar to de-asserting one input before the other, and so, the latch will go into one state or the other. However, since we do not know which the faster gate is, therefore, we do not know which state the latch will end up in. Thus, the latch's next state is undefined.

Of course, in practice, it is next to impossible to manufacture two gates and make the two connections with precisely the same delay. In addition, both S' and R' need to be de-asserted at exactly the same time. Nevertheless, if this circuit is used in controlling some mission-critical device, we don't want even this slim chance to happen.
In order to avoid this non-deterministic behavior, we must make sure that the two inputs are never de-asserted at the same time. Note that we do want the situation when both of them are de-asserted, as in times t_1 and t_3, so that the circuit can remember its current content. We want to de-assert one input after de-asserting the other, but just not de-asserting both of them at *exactly* the same time. In practice, it is very difficult to guarantee that these two signals are never de-asserted at the same time, so we relax the condition slightly by not having both of them asserted together. In other words, if one is asserted, then the other one cannot be asserted. Therefore, if both of them are never asserted simultaneously, then they cannot be de-asserted at the same time. A minor side benefit for not having both of them asserted together is that Q and Q' are never equal to each other. Recall that, from the names that we have given these two nodes, we do want them to be inverses of each other.
From the above analysis, we obtain the truth table in (b) of Figure 2.1(b) for the NAND implementation of the SR latch. In the truth table, Q and Q_{next} actually represent the same point in the circuit. The difference is that Q is the current value at that point, while Q_{next} is the new value to be updated in the next time period. Another way of looking at it is that Q is the input to a gate, and Q_{next} is the output from a gate. In other words, the signal Q goes into a gate, propagates through the two gates, and arrives back at Q as the new signal Q_{next}. The (c) of Figure 2.1(b) shows the logic symbol for the SR latch.
The SR latch can also be implemented using NOR gates, as shown in (a) of Figure 2.1(c). The truth table for this implementation is shown in (b) of Figure 2.1(c). From the truth table, we see that the main difference between this implementation and the NAND implementation is that for the NOR implementation, the S and R inputs are active-high, so that setting S to 1 will set the

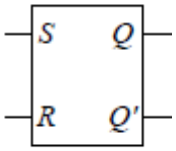
latch, and setting R to 1 will reset the latch. However, just like the NAND implementation, the latch is set when $Q = 1$, and reset when $Q = 0$. The latch remembers its previous state when $S = R = 0$. When $S = R = 1$, both Q and Q' are 0. The logic symbol for the SR latch using NOR implementation is shown in (c) of Figure 2.1(d).



(a)

S	R	Q	Q_{next}	Q_{next}'
0	0	0	0	1
0	0	1	1	0
0	1	×	0	1
1	0	×	1	0
1	1	×	0	0

(b)



(c)

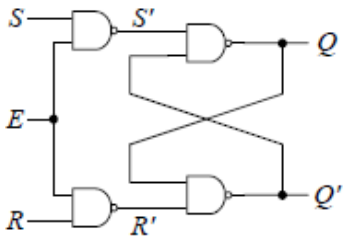
Figure 2.1(c): SR latch: (a) circuit using NOR gates; (b) truth table; (c) logic symbol.

3.2.1 S-R Latch with Enable

The SR latch is sensitive to its inputs all the time. In other words, Q will always change when either S or R is asserted. It is sometimes useful to be able to disable the inputs so that asserting them will not cause the latch to change state but to keep its current state. Of course, this is achieved by de-asserting both S and R. Hence, what we want is just one enable signal that will de-assert both S and R. The **SR latch with enable** (also known as a **gated SR latch**) shown in (a) of Figure 2.1(d) accomplishes this by adding two extra NAND gates to the original NAND gate

implementation of the latch. These two new NAND gates are controlled by the enable input, E, which determines whether the latch is enabled or disabled. When $E = 1$, the circuit behaves like the normal NAND implementation of the SR latch except that the new S and R inputs are active-high rather than active-low. When $E = 0$, then $S' = R' = 1$, and the latch will remain in its previous state regardless of the S and R inputs. The truth table and the logic symbol for the SR latch with enable is shown in (b) and (c) of Figure 2.1(d), respectively.

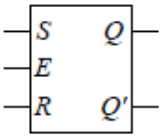
A typical operation of the latch is shown in the sample trace in (d) of Figure 2.1(d). Between t_0 and t_1, $E = 0$, so changing the S and R inputs do not affect the output, between t_1 and t_2, $E = 1$ and the trace is similar to the trace of Figure 2.1(b) except that the input signals are inverted.



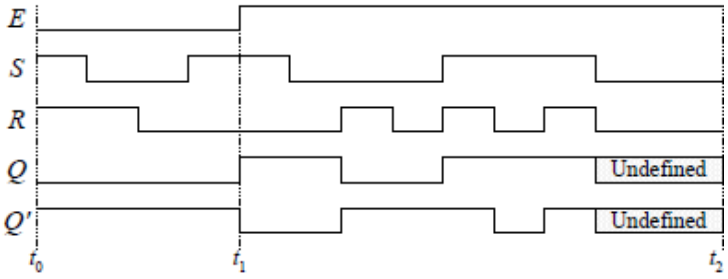
(a)

E	S	R	Q	Q_{next}	Q_{next}'
0	×	×	0	0	1
0	×	×	1	1	0
1	0	0	0	0	1
1	0	0	1	1	0
1	0	1	×	0	1
1	1	0	×	1	0
1	1	1	×	1	1

(b)



(c)



(d)

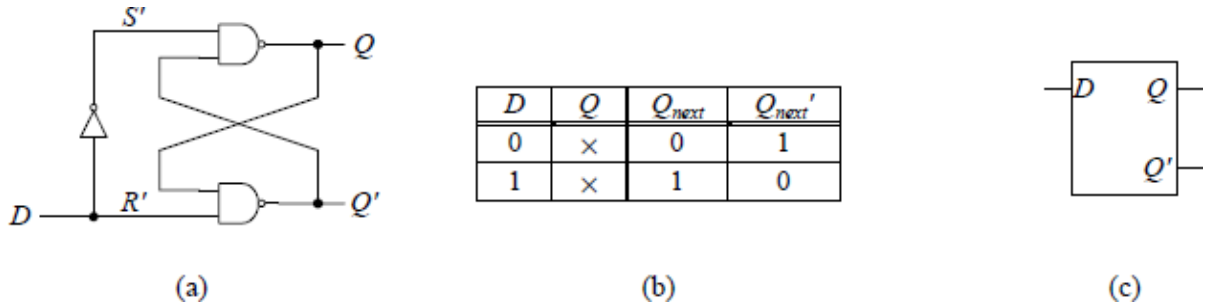
Figure 2.1(d): SR latch with enable: (a) circuit using NAND gates; (b) truth table; (c) logic symbol; (d) sample trace.

3.3 D Latch

Recall from Section 3.2 that the disadvantage with the SR latch is that we need to ensure that the two inputs, *S* and *R*, are never de-asserted at exactly the same time, and we said that we can guarantee this by not having both of them asserted. This situation is prevented in the **D latch** by adding an inverter between the original *S′* and *R′* inputs.

This way, *S′* and *R′* will always be inverses of each other, and so, they will never be asserted together. The circuit using NAND gates and the inverter is shown in Figure 2.1(e). There is now only one input *D* (for *data*). When *D* = 0, then *S′* = 1 and *R′* = 0, so this is similar to resetting the SR latch by making *Q* = 0. Similarly, when *D* = 1, then *S′* = 0 and *R′* = 1, and *Q* will be set to 1. From this observation, we see that *Qnext* always gets the same value as the input *D*, and is independent of the current value of *Q*. Hence, we obtain the truth table for the D latch, as shown in (b) of Figure 2.1(e).

Comparing the truth table for the D latch shown in (b) of Figure 2.1(e) with the truth table for the SR latch shown in (b) of Figure 2.1(b), it is obvious that we have eliminated not just one, but three rows, where *S′* = *R′*. The reason for adding the inverter to the SR latch circuit was to eliminate the row where *S′* = *R′* = 0. However, we still need to have the other two rows where *S′* = *R′* = 1 in order for the circuit to remember its current value. By not being able to set both *S′* and *R′* to 1, this D latch circuit has now lost its ability to remember. *Qnext* cannot remember the current value of *Q*, instead it will always follow *D*. The end result is like having a piece of wire where the output is the same as the input!



D	Q	Q_{next}	Q_{next}'
0	×	0	1
1	×	1	0

Figure 2.1(e): D latch: (a) circuit using NAND gates; (b) truth table; (c) logic symbol.

3.3.1 D Latch with Enable

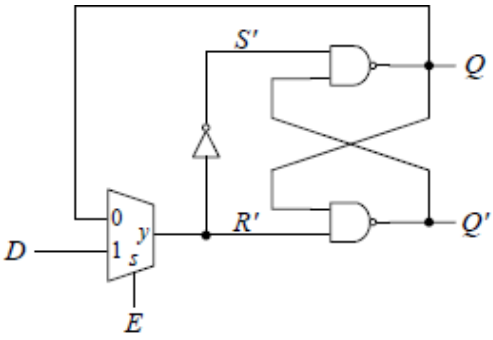
In order to make the D latch remember the current value, we need to connect *Q* (the current state value) back to the input *D*, thus creating another feedback loop. Furthermore, we need to be able to select whether to loop *Q* back to *D* or input a new value for *D*. Otherwise, like the bistable element, we will not be able to change the state of the circuit. One way to achieve this is to use a 2-input multiplexer to select whether to feedback the current value of *Q* or pass an external input back to *D*. The circuit for the **D latch with enable** (also known as a **gated D latch**) is shown in (a) of Figure 2.1(f). The external input becomes the new *D* input, the output of the multiplexer is connected to the original *D* input, and the select line of the multiplexer is the enable signal *E*.

When the enable signal *E* is asserted (*E* = 1), the external *D* input passes through the multiplexer, and so *Qnext* (i.e., the output *Q*) follows the *D* input. On the other hand, when *E* is de-asserted (*E* = 0), the current value of *Q* loops back as the input to the circuit, and so *Qnext* retains its last value independent of the *D* input.

When the latch is enabled, the latch is said to be open, and the path from the input *D* to the output *Q* is transparent. In other words, *Q* follows *D*. Because of this characteristic, the D latch with enable circuit is often referred to as a **transparent latch**. When the latch is disabled, it is closed, and the latch remembers its current state.

The truth table and the logic symbol for the D latch with enable are shown in (b) and (c) of Figure 2.1(f). A sample trace for the operation of the D latch with enable is shown in (d) of Figure 2.1(f). Between t_0 and t_1, the latch is enabled with *E* = 1, so the output *Q* follows the input *D*. Between $t1$ and t_2, the latch is disabled, so *Q* remains stable even when *D* changes.

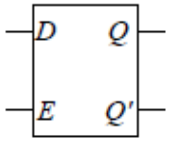
An alternative way to construct the D latch with enable circuit is shown in Figure 2.1(g). Instead of using the 2-input multiplexer, as shown in (a) of Figure 2.1(f), we start with the SR latch with enable circuit of (a) of Figure 2.1(f), and connect the *S* and *R* inputs together with an inverter. The functional operations of these two circuits are identical.



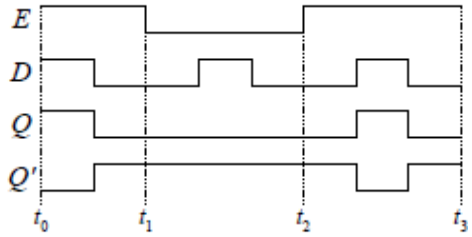
(a)

E	D	Q	Q_{next}	Q_{next}'
0	×	0	0	1
0	×	1	1	0
1	0	×	0	1
1	1	×	1	0

(b)



(c)



(d)

Figure 2.1(f): D latch with enable: (a) circuit; (b) truth table; (c) logic symbol; (d) sample trace.

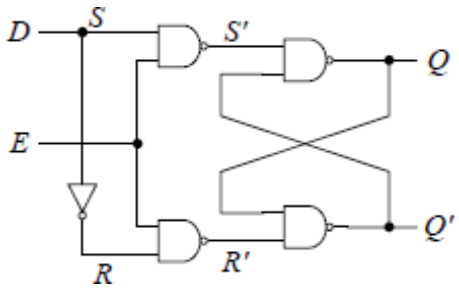


Figure 2.1(g): D latch with enable circuit using four NAND gates.

3.4 Analyzing Sequential Circuits

Many logic circuits contain flip-flops, one-shots, and logic gates that are connected to perform a specific operation. Very often, a master clock signal is used to cause the logic levels in the circuit to go through a particular sequence of states. We can generally analyze these sequential circuits by following this step-by-step procedure.

1. Examine the circuit diagram and look for circuit structures such as counters or shift registers that you are familiar with. This can help to simplify the analysis.

2. Determine the logic levels that are present at the inputs of each flip-flop prior to the occurrence of the first clock pulse.

3. Use these levels to determine how each flip-flop output will change in response to the first clock pulse.

4. Repeat steps 2 and 3 for each successive clock pulse.

4.0 Conclusion

The other major aspect of a digital system is the analysis and design of sequential digital circuits. However, sequential circuit design depends, greatly, on the combinational circuit design. The logic circuits whose outputs at any instant of time depend only on the input signals present at that time are known as combinational circuits. On the other hand, the logic circuits whose outputs at any instant of time depend on the present inputs as well as on the past outputs are called sequential circuits. In sequential circuits, the output signals are fed back to the input side.

5.0 Summary

In this unit, you learnt that:

- Latches and Flip-flops are the building blocks of sequential circuits.
- Bistable Elements
- SR Latch
- D Latch is a circuit that contains a NAND gate latch and two steering NAND gates.

6.0 Tutor Marked Assignment

1) Draw the SR latch with enable but using NOR gates to implement the SR latch. Derive the truth table for this circuit.

2) Draw the D latch using NOR gates

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 2: Flip Flops

Contents **Pages**

1.0 Introduction

There are basically four main types of flip-flops: D, S-R, J-K, and T. The major differences in these flip-flop types are in the number of inputs they have and how they change states. Like the D flip-flop, each type can also have different variations, such as active-high or low inputs, whether they change state at the rising or falling edge of the clock signal, and whether they have any asynchronous inputs. Any given sequential circuit can be built using any of these types of flip-flops or combinations of them. However, selecting one type of flip-flop over another type to use in a particular circuit can affect the overall size of the circuit. Today, sequential circuits are designed primarily with D flip-flops only because of their simple operation. The action of clearing a Flip-Flop or a latch is also called resetting, and both terms are used interchangeably in the digital field. In fact, a CLEAR input can also be called RESET input, and a SET-CLEAR latch can be called a SET-RESET latch, which will be used as S-R throughout our discussions.

2.0 Objective

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

- Identify the types of flip-flops
- Discuss the various types of S-R Flip-Flop

3.0 Flip-Flops

The most important memory element is the flip-flop, which is made up of an assembly of logic gates. Even though a logic gate by itself has no storage capability, several can be connected together in ways that permit information to be stored. Figure is the general type of symbol used for a flip-flop. It shows two outputs, labeled Q and Q', that are the inverse of each other. Actually, any letter can be used, but Q is the one most often used. The Q output is called the normal flip-flop output, and Q' is the inverted flip-flop output. Whenever we refer to the state of a flip-flop, we are referring to the state of its normal (Q) output; it is understood that its inverted output (Q') is in the opposite state. For example, if we say that a flip-flop is in the HIGH (1) state, we mean that $Q = 1$; if we say that a flip-flop is in the LOW (0) state, we mean that $Q = 0$. Of course, the Q' state will always be the inverse of Q.

A flip-flop, then, has the two allowed operating states indicated in (b) of Figure 2.2(a). Note the different ways that are used to refer to these two states. As the symbol in (a) of Figure 2.2(a) implies, a flip-flop can have one or more inputs. These inputs are used to cause the flip-flop to switch back and forth ("flip-flop") between its possible output states. As we shall see, a Flip-Flop input only has to be pulsed momentarily to cause a change in the Flip-Flop output state, and the output will remain in that new state even after the input pulse is over. This is the Flip-Flop's memory characteristics.

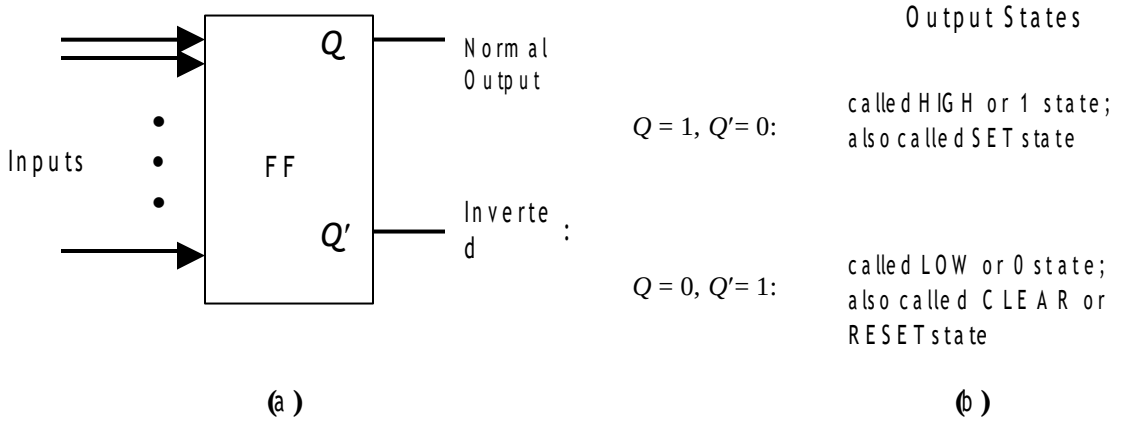


Figure 2.2(a): General flip-flop symbol and definition of two possible output states

There are different types of flip-flops depending on how their inputs and clock pulses cause transition between two states. The most basic FF circuit can be constructed from either two NAND gates or two NOR gates. We will discuss the different types of flip-flops in this unit and the next unit.

3.1 S-R (Set-Reset) Flip-flop

An S-R flip-flop has two inputs named Set (S) and Reset (R), and two outputs Q and Q'. The outputs are complement of each other, *i.e.*, if one of the outputs is 0 then the other should be 1. This can be implemented using NAND or NOR gates. The block diagram of an S-R flip-flop is shown in Figure 2.2(b).

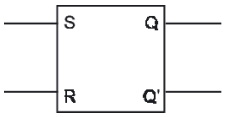


Figure 2.2(b): Block diagram for S-R flip-flop.

3.2 S-R Flip-flop Based on NOR Gates

An S-R flip-flop can be constructed with NOR gates at ease by connecting the NOR gates back to back as shown in Figure 2.2(c). The cross-coupled connections from the output of gate 1 to the input of gate 2 constitute a feedback path. This circuit is not clocked and is classified as an asynchronous sequential circuit. The truth table for the S-R flip-flop based on a NOR gate is shown in Table 2.2 (a).

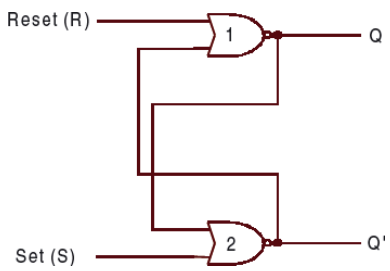


Figure 2.2(c): S-R flip-flop based on a NOR gate

To analyze the circuit shown in Figure 2.2(c), we have to consider the fact that the output of a NOR gate is 0 if any of the inputs are 1, irrespective of the other input. The output is 1 only if all of the inputs are 0. The outputs for all the possible conditions as shown in Table 2.2 (a) are described as follows.

Table 2.2(a): Output of S-R flip-flop based on a NOR gate

Inputs		*Outputs*		*Action*
S	R	Q_{n+1}	Q'_{n+1}	
0	0	Q_n	Q'_n	No change
0	1	0	1	Reset
1	0	1	0	Set
1	1	0	0	Forbidden (Undefined)
0	0	–	–	Indeterminate

Case 1. For S = 0 and R = 0, the flip-flop remains in its present state (Q*n*). It means that the next state of the flip-flop does not change, *i.e.*, Q*n*+1 = 0 if Q*n* = 0 and vice versa. First let us assume that Q*n* = 1 and Q'*n* = 0. Thus the inputs of NOR gate 2 are 1 and 0, and therefore its output Q'*n* +1 = 0. This output Q'*n*+1 = 0 is fed back as the input of NOR gate 1, thereby producing a 1 at the output, as both of the inputs of NOR gate 1 are 0 and 0; so Q*n*+1 = 1 as originally assumed.
Now let us assume the opposite case, *i.e.*, Q*n* = 0 and Q'*n* = 1. Thus the inputs of NOR gate 1 are 1 and 0, and therefore its output Q*n*+1 = 0. This output Q*n*+1 =0 is fed back as the input of NOR gate 2, thereby producing a 1 at the output, as both of the inputs of NOR gate 2 are 0 and 0; so Q'*n*+1 = 1 as originally assumed. Thus we find that the condition S = 0 and R = 0 do not affect the outputs of the flip-flop, which means this is the memory condition of the S-R flip-flop.
Case 2. The second input condition is S = 0 and R = 1. The 1 at R input forces the output of NOR gate 1 to be 0 (*i.e.*, Q*n*+1 = 0). Hence both the inputs of NOR gate 2 are 0 and 0 and so its output Q'*n*+1 = 1. Thus the condition S = 0 and R = 1 will always reset the flip-flop to 0. Now if the R returns to 0 with S = 0, the flip-flop will remain in the same state.
Case 3. The third input condition is S = 1 and R = 0. The 1 at S input forces the output of NOR gate 2 to be 0 (*i.e.*, Q'*n*+1 = 0). Hence both the inputs of NOR gate 1 are 0 and 0 and so its output Q*n*+1 = 1. Thus the condition S = 1 and R = 0 will always set the flip-flop to 1. Now if the S returns to 0 with R = 0, the flip-flop will remain in the same state.
Case 4. The fourth input condition is S = 1 and R = 1. The 1 at R input and 1 at S input forces the output of both NOR gate 1 and NOR gate 2 to be 0. Hence both the outputs of NOR gate 1 and NOR gate 2 are 0 and 0; *i.e.*, Q*n*+1 = 0 and Q'*n*+1 = 0. Hence this condition S = 1 and R = 1

violates the fact that the outputs of a flip-flop will always be the complement of each other. Since the condition violates the basic definition of flip-flop, it is called the *undefined* condition. Generally this condition must be avoided by making sure that 1s are not applied simultaneously to both of the inputs.

Case 5. If case 4 arises at all, then S and R both return to 0 and 0 simultaneously, and then any one of the NOR gates acts faster than the other and assumes the state. For example, if NOR gate 1 is faster than NOR gate 2, then Q*n*+1 will become 1 and this will make Q'*n*+1 = 0. Similarly, if NOR gate 2 is faster than NOR gate 1, then Q'*n*+1 will become 1 and this will make Q*n*+1 = 0. Hence, this condition is determined by the flip-flop itself. Since this condition cannot be controlled and predicted it is called the *indeterminate* condition.

3.3 S'-R' Flip-flop Based on NAND Gates

An S'-R' flip-flop can be constructed with NAND gates by connecting the NAND gates back to back as shown in Figure 2.2(d). The operation of the S'-R' flip-flop can be analyzed in a similar manner as that employed for the NOR-based S-R flip-flop. This circuit is also not clocked and is classified as an asynchronous sequential circuit. The truth table for the S'-R' flip-flop based on a NAND gate is shown in Table 2.2(b)

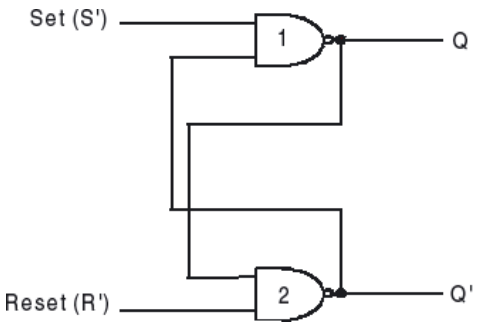


Figure 2.2(d): NAND-based S'-R' flip-flop.

To analyze the circuit shown in Figure 2.2(c), we have to remember that a LOW at any input of a NAND gate forces the output to be HIGH, irrespective of the other input. The output of a NAND gate is 0 only if all of the inputs of the NAND gate are 1. The outputs for all the possible conditions as shown in Table 2.2(b) are described below.

Table 2.2(b): Output of S-R flip-flop based on a NAND gate

Inputs		*Outputs*		*Action*
S'	R'	Q_{n+1}	Q'_{n+1}	
1	1	Q_n	Q_n	No change
1	0	0	1	Reset
0	1	1	0	Set
0	0	1	1	Forbidden (Undefined)
1	1	–	–	Indeterminate

Case 1. For S' = 1 and R' = 1, the flip-flop remains in its present state (Q*n*). It means that the next state of the flip-flop does not change, *i.e.*, Q*n*+1 = 0 if Q*n* = 0 and vice versa. First let us assume that Q*n* =1 and Q'*n* = 0. Thus the inputs of NAND gate 1 are 1 and 0, and therefore its output Q*n*+1 = 1. This output Q*n*+1 = 1 is fed back as the input of NAND gate 2, thereby producing a 0 at the output, as both of the inputs of NAND gate 2 are 1 and 1; so Q'*n*+1 = 0 as originally assumed.

Now let us assume the opposite case, *i.e.*, Q*n* = 0 and Q'*n* = 1. Thus the inputs of NAND gate 2 are 1 and 0, and therefore its output Q'*n*+1 = 1. This output Q'*n*+1 = 1 is fed back as the input of NAND gate 1, thereby producing a 0 at the output, as both of the inputs of NAND gate 1 are 1 and 1; so Q*n*+1 = 0 as originally assumed. Thus we find that the condition S' = 1 and R' = 1 do not affect the outputs of the flip-flop, which means this is the memory condition of the S'-R' flip-flop.

Case 2. The second input condition is S' = 1 and R' = 0. The 0 at R' input forces the output of NAND gate 2 to be 1 (*i.e.*, Q'*n*+1 = 1). Hence both the inputs of NAND gate 1 are 1 and 1 and so its output Q*n*+1 = 0. Thus the condition S' = 1 and R' = 0 will always reset the flip-flop to 0. Now if the R' returns to 1 with S' = 1, the flip-flop will remain in the same state.

Case 3. The third input condition is S' = 0 and R' = 1. The 0 at S' input forces the output of NAND gate 1 to be 1 (*i.e.*, Q*n*+1 = 1). Hence both the inputs of NAND gate 2 are 1 and 1 and so its output Q'*n*+1 = 0. Thus the condition S' = 0 and R' = 1 will always set the flip-flop to 1. Now if the S' returns to 1 with R' = 1, the flip-flop will remain in the same state.

Case 4. The fourth input condition is S' = 0 and R' = 0. The 0 at R' input and 0 at S' input forces the output of both NAND gate 1 and NAND gate 2 to be 1. Hence both the outputs of NAND gate 1 and NAND gate 2 are 1 and 1; *i.e.*, Q*n*+1 = 1 and Q'*n*+1 = 1. Hence this condition S' = 0 and R' = 0 violates the fact that the outputs of a flip-flop will always be the complement of each other. Since the condition violates the basic definition of a flip-flop, it is called the *undefined* condition. Generally, this condition must be avoided by making sure that 0s are not applied simultaneously to both of the inputs.

Case 5. If case 4 arises at all, then S' and R' both return to 1 and 1 simultaneously, and then any one of the NAND gates acts faster than the other and assumes the state. For example, if NAND gate 1 is faster than NAND gate 2, then Q*n*+1 will become 1 and this will make Q'*n*+1 = 0. Similarly, if NAND gate 2 is faster than NAND gate 1, then Q'*n*+1 will become 1 and this will make Q*n*+1 = 0. Hence, this condition is determined by the flip-flop itself. Since this condition cannot be controlled and predicted it is called the *indeterminate* condition.

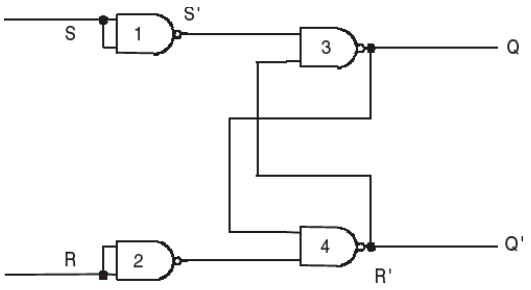


Figure 2.2(e): An S-R flip-flop using NAND gates

Thus, comparing the NOR flip-flop and the NAND flip-flop, we find that they basically operate in just the complement fashion of each other. Hence, to convert a NAND-based S'-R' flip-flop into a NOR-based S-R flip-flop, we have to place an inverter at each input of the flip-flop. The resulting circuit is shown in Figure 2.2(e), which behaves in the same manner as an S-R flip-flop.

4.0 Conclusion

The basic 1-bit digital memory circuit is known as a flip-flop. It can have only two states, either the 1 state or the 0 state. A flip-flop is also known as a bistable multivibrator. Flip-flops can be obtained by using NAND or NOR gates

5.0 Summary

In this unit, you learnt about:

- Types of Flip-flops including:
  - S-R Flip-flop Based on NOR Gates
  - S'-R' Flip-flop Based on NAND Gates

6.0 Tutor Marked Assignment

1) What is the normal resting state of the S-R inputs? What is the active state of each input?
2) Derive the truth table for the SR flip-flop with enable.
3) Draw the logic diagram for the SR flip-flop. Use NOR gates.
4) Draw the logic diagram for the SR flip-flop. Use AND gates.

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 3: Clocked S-R Flip-Flop

Contents **Pages**

1.0 Introduction

Generally, synchronous circuits change their states only when clock pulses are present. The operation of the basic flip-flop can be modified by including an additional input to control the behavior of the circuit.

2.0 Objectives

- Understand the Clocked S-R flip-flop
- Understand D Flip-Flop
- Understand the initial state of circuit with preset and clear input

3.0 Clocked Signals and Clocked Flip-Flops

Digital systems can operate either asynchronously or synchronously. In asynchronous systems, the outputs of logic circuits can change state any time one or more of the inputs change. As asynchronous system is difficult to design and troubleshoot. In synchronous systems, the exact times at which any output can change states is determined by a signal commonly called the clock. The clock signal is generally a rectangular pulse train or squarewave. The clock signal is distributed to all parts of the system, and most (if not all) of the system outputs can change state only when the clock makes a transition. When the clock changes from a 0 to a 1, this is called the positive-going transition (PGT); when the clock goes from 1 to 0, this is the negative-going transition (NGT). Most digital systems are principally synchronous (although there are always some asynchronous parts), since synchronous circuits are easier to design and troubleshoot. They are easier to troubleshoot because the circuit outputs can change only at specific instants of time. In other words, almost everything is synchronized to the clock-signal transitions.

The synchronizing action of the clock signals is accomplished through the use of clocked flip-flops that are designed to change states on one or the other of the clock's transitions. There are types of clocked Flip-flops that are used in a wide range of applications. There are some principal ideas that are common to all of them.

1. Clocked Flip-Flops have a clock input that is typically labeled CLK, CK or CP. We will use CLK. In most clocked Flip-Flops, the CLK input is edge-triggered, which means that it is activated by a signal transition.

2. Clocked Flip-Flops also have one or more control inputs that can have various names, depending on their operation. The control inputs will have no effect on *Q* until the active clock transition occurs. In other words, their effect is synchronized with the signal applied to CLK. For this reason they are called synchronous control inputs.

3. In summary, we can say that the control inputs get the Flip-Flop outputs ready to change, while the active transition at the CLK input actually triggers the change.

3.1 Clocked S-R flip-flop

The circuit shown in Figure 2.3(a) consists of two AND gates. The clock input is connected to both of the AND gates, resulting in LOW outputs when the clock input is LOW. In this situation the changes in S and R inputs will not affect the state (Q) of the flip-flop.

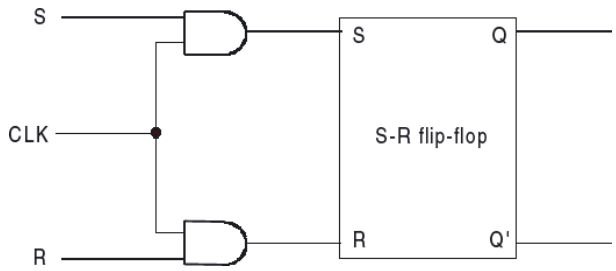


Figure 2.3(a): Block diagram of a clocked S-R flip-flop.

On the other hand, if the clock input is HIGH, the changes in S and R will be passed over by the AND gates and they will cause changes in the output (Q) of the flip-flop. This way, any information, either 1 or 0, can be stored in the flip-flop by applying a HIGH clock input and be retained for any desired period of time by applying a LOW at the clock input. This type of flip-flop is called a *clocked S-R flip-flop*. Such a clocked S-R flip-flop made up of two AND gates and two NOR gates is shown in Figure 2.3(b)

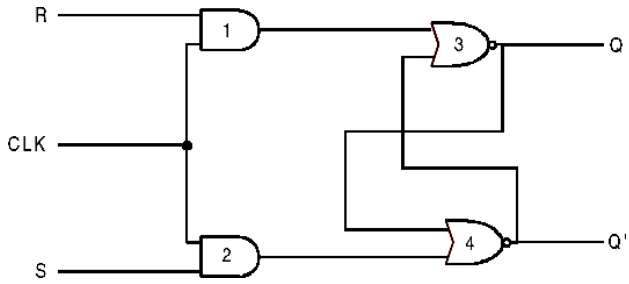


Figure 2.3(b): A clocked NOR-based S-R flip-flop.

Now the same S-R flip-flop can be constructed using the basic NAND latch and two other NAND gates as shown in Figure 2.3(c). The S and R inputs control the states of the flip-flop in the same way as described earlier for the unclocked S-R flip-flop. However, the flip-flop only responds when the clock signal occurs. The clock pulse input acts as an enable signal for the other two inputs. As long as the clock input remains 0 the outputs of NAND gates 1 and 2 stay at logic 1. This 1 level at the inputs of the basic NAND-based S-R flip flop retains the present state.

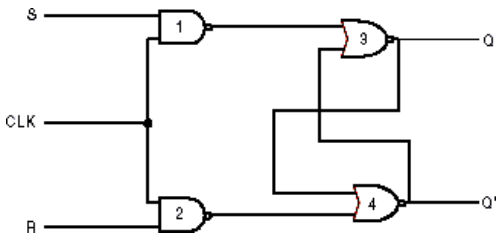


Figure 2.3(c): A clocked NAND-based S-R flip-flop

The logic symbol of the S-R flip-flop is shown in Figure 2.3(d). It has three inputs: S, R, and CLK. The CLK input is marked with a small triangle. The triangle is a symbol that denotes the fact that the circuit responds to an edge or transition at CLK input.
Assuming that the inputs do not change during the presence of the clock pulse, we can express the working of the S-R flip-flop in the form of the truth table in Table 2.3(a). Here, S*n* and R*n* denote the inputs and Q*n* the output during the bit time *n.*
Q*n*+1 denotes the output after the pulse passes, *i.e.*, in the bit time *n* + 1.

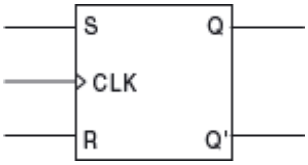


Figure 2.3(d): The logic symbol of the S-R flip-flop

Table 2.3(a): Output of the S-R flip-flop.

Inputs		*Output*
S_n	R_n	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	–

Case 1. If S*n* = R*n* = 0, and the clock pulse is not applied, the output of the flip-flop remains in the present state. Even if S*n* = R*n* = 0, and the clock pulse is applied, the output at the end of the clock pulse is the same as the output before the clock pulse, *i.e.*, Q*n*+1 = Q*n*. The first row of the table indicates that situation.

Case 2. For S*n* = 0 and R*n* = 1, if the clock pulse is applied (*i.e.*, CLK = 1), the output of NAND gate 1 becomes 1; whereas the output of NAND gate 2 will be 0. Now a 0 at the input of NAND gate 4 forces the output to be 1, *i.e.*, Q' = 1. This 1 goes to the input of NAND gate 3 to make both the inputs of NAND gate 3 as 1, which forces the output of NAND gate 3 to be 0, *i.e.*, Q =0.

Case 3. For S*n* = 1 and R*n* = 0, if the clock pulse is applied (*i.e.*, CLK = 1), the output of NAND gate 2 becomes 1; whereas the output of NAND gate 1 will be 0. Now a 0 at the input of NAND gate 3 forces the output to be 1, *i.e.*, Q = 1. This 1 goes to the input of NAND gate 4 to make both the inputs of NAND gate 4 as 1, which forces the output of NAND gate 4 to be 0, *i.e.*, Q'=0.

Case 4. For S*n* = 1 and R*n* = 1, if the clock pulse is applied (*i.e.*, CLK = 1), the outputs of both NAND gate 2 and NAND gate 1 becomes 0. Now a 0 at the input of both NAND gate 3 and NAND gate 4 forces the outputs of both the gates to be 1, *i.e.*, Q = 1 and Q' = 1. When the CLK input goes back to 0 (while S and R remain at 1), it is not possible to determine the next state, as it depends on whether the output of gate 1 or gate 2 goes to 1 first.

3.1.1 Preset and Clear

In the flip-flops shown in Figures 2.2(b) or figure 2.2(d), when the power is switched on, the state of the circuit is uncertain. It may come to reset (Q = 0) or set (Q = 1) state. But in many applications it is required to initially set or reset the flip-flop., *i.e.*, the initial state of the flip-flop is to be assigned. This is done by using the direct or asynchronous inputs. These inputs are referred to as *preset* (Pr) and *clear* (Cr) inputs. These inputs may be applied at any time between clock pulses and is not in synchronism with the clock. Such an S-R flip-flop containing preset and clear inputs is shown in Figure 2.3(e). From Figure 2.3(e), we see that if Pr = Cr = 1, the circuit operates according to Table 2.3(a).

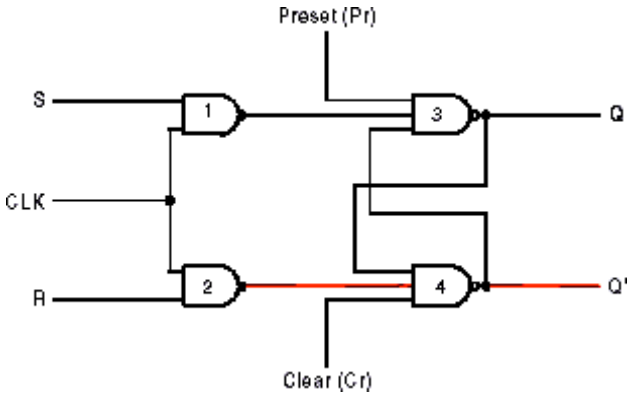


Figure 2.3(e): S-R flip-flop with preset and clear.

If Pr = 1 and Cr = 0, the output of NAND gate 4 is forced to be 1, *i.e.*, Q' = 1 and the flip-flop is reset, overwriting the previous state of the flip-flop.

If Pr = 0 and Cr = 1, the output of NAND gate 3 is forced to be 1, *i.e.*, Q = 1 and the flip-flop is set, overwriting the previous state of the flip-flop. Once the state of the flip-flop is established asynchronously, the inputs Pr and Cr must be connected to logic 1 before the next clock is applied.

The condition Pr = Cr = 0 must not be applied, since this leads to an uncertain state.

The logic symbol of an S-R flip-flop with Pr and Cr inputs is shown in Figure 2.3(f). Here, bubbles are used for Pr and Cr inputs, which indicate these are active low inputs, which means that the intended function is performed if the signal applied to Pr and Cr is LOW. The operation

of Figure 2.3(f) is shown in Table 2.3(b). The circuit can be designed such that the asynchronous inputs override the clock, *i.e.*, the circuit can be set or reset even in the presence of the clock pulse.

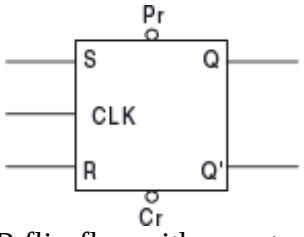


Figure 2.3(f): Logic symbol of an S-R flip-flop with preset and clear.

Table 2.3(b): Output of S-R flip-flop with preset and clear

Inputs			*Output*	*Operation*
CLK	*Cr*	*Pr*	*Q*	*performed*
1	1	1	Q_{n+1} (Figure 7.3)	Normal flip-flop
0	1	0	1	Preset
0	0	1	0	Clear
0	0	0	–	Uncertain

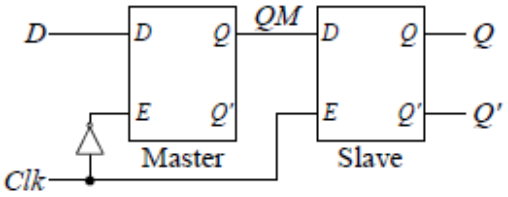
3.2 D Flip-Flop

Unlike the latch, a flip-flop is not level-sensitive, but rather **edge-triggered**. In other words, data gets stored into a flip-flop only at the active edge of the clock. An **edge-triggered D flip-flop** achieves this by combining in series, a pair of D latches. The figure (a) of Figure 2.3(g) shows a **positive-edge-triggered D flip-flop**, where two D latches are connected in series. A clock signal *CLK* is connected to the *E* input of the two latches: one directly, and one through an inverter. The first latch is called the *master* latch. The master latch is enabled when *CLK* = 0 because of the inverter, and so *QM* follows the primary input *D*. However, the signal at *QM* cannot pass over to the primary output *Q*, because the second latch (called the *slave* latch) is disabled when *CLK* = 0. When *CLK* = 1, the master latch is disabled, but the slave latch is enabled so that the output from the master latch, *QM*, is transferred to the primary output *Q*. The slave latch is enabled all the while that *CLK* = 1, but its content changes only at the rising edge of the clock, because once *CLK* is 1, the master latch is disabled, and the input to the slave latch, *QM*, will be constant. Therefore, when *CLK* = 1 and the slave latch is enabled, the primary output *Q* will not change because the input *QM* is not changing.

The circuit shown in (a) of Figure 2.3(g) is called a positive-edge-triggered D flip-flop because the primary output *Q* on the slave latch changes only at the rising edge of the clock. If the slave latch is enabled when the clock is low (i.e., with the inverter output connected to the *E* of the slave latch), then it is referred to as a **negative-edge-triggered** flip-flop. The circuit is also referred to as a **master-slave** D flip-flop because of the two D latches used in the circuit.

(b) of Figure 2.3(g) shows the operation table for the D flip-flop. The ↑ symbol signifies the rising edge of the clock. When *CLK* is either at 0 or 1, the flip-flop retains its current value (i.e.,

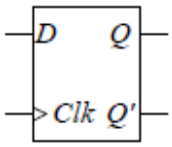
Qnext = *Q*). *Qnext* changes and follows the primary input *D* only at the rising edge of the clock. The logic symbol for the positive-edge-triggered D flip-flop is shown in (c) of Figure 2.3(g). The small triangle at the clock input indicates that the circuit is triggered by the edge of the signal, and so it is a flip-flop. Without the small triangle, the symbol would be that for a latch. If there is a circle in front of the clock line, then the flip-flop is triggered by the falling edge of the clock, making it a negative-edge-triggered flip-flop. (d) of Figure 2.3(g) shows a sample trace for the D flip-flop. Notice that when *CLK* = 0, *QM* follows *D*, and the output of the slave latch, *Q*, remains constant. On the other hand, when *CLK* = 1, *Q* follows *QM*, and the output of the master latch, *QM*, remains constant.



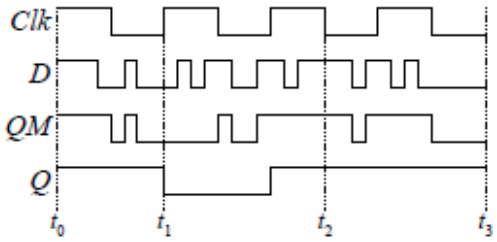
(a)

Clk	*D*	*Q*	Q_{next}	Q_{next}'
0	×	0	0	1
0	×	1	1	0
1	×	0	0	1
1	×	1	1	0
↑	0	×	0	1
↑	1	×	1	0

(b)



(c)



(d)

Figure 2.3(g): Master-slave positive-edge-triggered D flip-flop: (a) circuit using D latches; (b) operation table; (c) logic symbol; (d) sample trace.

Figure 2.3(h) compares the different operations between a latch and a flip-flop. In (a) of Figure 2.3(h), we have a D latch with enable, a positive-edge-triggered D flip-flop, and a negative-edge-triggered D flip-flop, all having the same *D* input and controlled by the same clock signal. (b) of Figure 2.3(h) shows a sample trace of the circuit's operations. Notice that the gated D latch, Q_a, follows the *D* input as long as the clock is high (between times t_0 and t_1 and times t_2 and t_3). The positive-edge-triggered flip-flop, Q_b, follows the *D* input only at the rising edge of the clock at time t_2, while the negative-edge-triggered flip-flop, Q_c, follows the *D* input only at the falling edge of the clock at times t_1 and t_3.

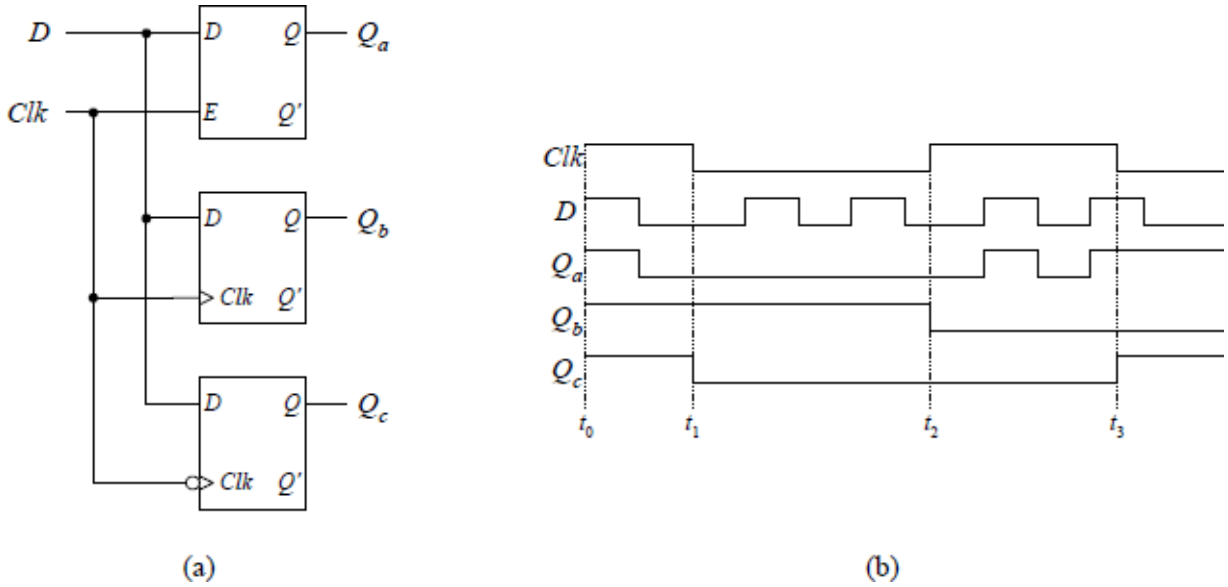


Figure 2.3(h): Comparison of a gated latch (a) a positive-edge-triggered flip-flop, and a negative-edge-triggered flip-flop: circuit; (b) sample trace.

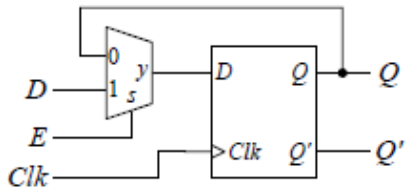
3.3 D Flip-Flop with Enable

So far, with the construction of the different memory elements, it seems like every time we add a new feature we have also lost a feature that we need. The careful reader will have noticed that, in building the D flip-flop, we have again lost the most important property of a memory element—it can no longer remember its current content.

At every active edge of the clock, the D flip-flop will load in a new value. So how do we get it to remember its current value and not load in a new value? The answer, of course, is exactly the same as what we did with the D latch, and that is by adding an enable input, *E*, through a 2-input multiplexer, as shown in Figure 2.3(i).

When $E = 1$, the primary input *D* signal will pass to the *D* input of the flip-flop, thus updating the content of the flip-flop at the active edge. When $E = 0$, the current content of the flip-flop at *Q* is passed back to the *D* input of the flip-flop, thus keeping its current value.

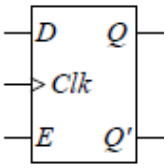
Notice that changes to the flip-flop value occur only at the active edge of the clock. Here, we are using the rising edge as the active edge. The operation table and the logic symbol for the D flip-flop with enable is shown in (b) and (c) of Figure 2.3(i) respectively.



(a)

Clk	E	D	Q	Q_{next}	Q_{next}'
0	×	×	0	0	1
0	×	×	1	1	0
1	×	×	0	0	1
1	×	×	1	1	0
↑	0	×	0	0	1
↑	0	×	1	1	0
↑	1	0	×	0	1
↑	1	1	×	1	0

(b)



(c)

Figure 2.3(i): D flip-flop with enable: (a) circuit; (b) operation table; (c) logic symbol.

4.0 Conclusion

Latches are known as level-sensitive because their outputs are affected by their inputs as long as they are enabled. Their memory state can change during this entire time when the enable signal is asserted. In a computer circuit, however, we do not want the memory state to change at various times when the enable signal is asserted. Instead, we like to synchronize all of the state changes to happen at precisely the same moment and at regular intervals. In order to achieve this, two things are needed: 1) a synchronizing signal, and 2) a memory circuit that is not level-sensitive. The synchronizing signal, of course, is the **clock**, and the non-level-sensitive memory circuit is the flip-flop.

5.0 Summary

In this unit, you learnt about:

- Clocked S-R flip-flop
- Setting the initial state of the flip-flop by using the direct or asynchronous preset (Pr) and clear (Cr) inputs.
- D Flip-flop
- D Flip-Flop with Enable

6.0 Tutor Marked Assignment

1) What is meant by the term “edge-trigggered”?
2) Draw and show the truth table for the clocked S-R flip-flop
3) Show the logic diagram of a clocked D flip-flop with four NAND gates.
4) Draw the logic diagram of master-slave D flip-flop. Use NAND gates.

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 4: J-K Flip-Flop

Contents **Pages**

1.0 Introduction

The operation of the JK flip-flop is very similar to the S-R flip-flop. The *J* input is just like the *S* input in the S-R flip-flop in that, when asserted, it sets the flip-flop. Similarly, the *K* input is like the *R* input where it resets the flip-flop when asserted. The only difference is when both inputs, *J* and *K*, are asserted, for the S-R flip-flop, the next state is undefined; whereas, for the J-K flip-flop, the next state is the inverse of the current state. In other words, the J-K flip-flop toggles its state when both inputs are asserted.

2.0 Objectives

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

- Understand the various design of J-K Flip-Flop
- Understand the Master-Slave J-K Flip-Flop
- Understand T Flip flop

3.0 J-K Flip-Flop

A J-K flip-flop has very similar characteristics to an S-R flip-flop. The only difference is that the undefined condition for an S-R flip-flop, *i.e.*, S*n* = R*n* = 1 condition, is also included in this case. Inputs J and K behave like inputs S and R to set and reset the flip-flop respectively.
When J=K= 1, the flip-flop is said to be in a *toggle state*, which means the output switches to its complementary state every time a clock passes. The data inputs are J and K, which are ANDed with Q' and Q respectively to obtain the inputs for S and R respectively. A J-K flip-flop thus obtained is shown in Figure 2.4(a). The truth table of such a flip-flop is given in Table 2.4(a), which is reduced to Table 2.4(b) for convenience.

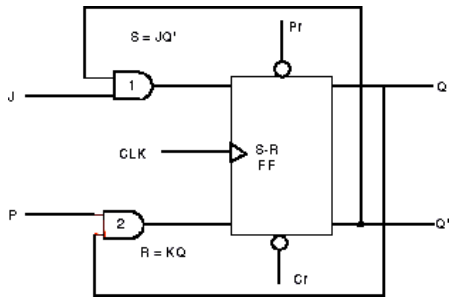


Figure 2.4(a): An S-R flip-flop converted into a J-K flip-flop.

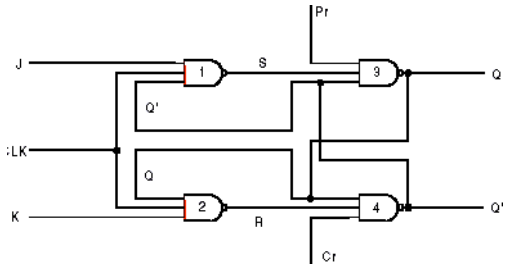


Figure 2.4(b): A J-K flip-flop using NAND gates

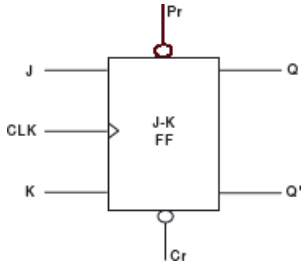


Figure 2.4(c): Logic symbol of a J-K flip-flop.

It is not necessary to use the AND gates of Figure 2.4(a), since the same function can be performed by adding an extra input terminal to each of the NAND gates 1 and 2. With this modification incorporated, we get the J-K flip-flop using NAND gates as shown in Figure 2.4(b). The logic symbol of a J-K flip-flop is shown in Figure 2.4(c).

Table 2.4(a): Output of J-K flip-flop.

Data inputs		*Outputs*		*Inputs to S-R FF*		*Output*
J_n	K_n	Q_n	Q'_n	S_n	R_n	Q_{n+1}
0	0	0	1	0	0	0
0	0	1	0	0	0	1
0	1	0	1	0	0	0
0	1	1	0	0	1	0
1	0	0	1	1	0	1
1	0	1	0	0	0	1
1	1	0	1	1	0	1
1	1	1	0	0	1	0

Case 1. When the clock is applied and J = 0, whatever the value of Q'*n* (0 or 1), the output of NAND gate 1 is 1. Similarly, when K = 0, whatever the value of Q*n* (0 or 1), the output of gate 2 is also 1. Therefore, when J = 0 and K = 0, the inputs to the basic flip-flop are S = 1 and R = 1. This condition forces the flip-flop to remain in the same state.

Table 2.4(b): Reduced Output of J-K flip-flop

Inputs		*Output*
J_n	K_n	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	Q'_n

Case 2. When the clock is applied and J = 0 and K = 1 and the previous state of the flip-flop is reset (*i.e.*, Q*n* = 0 and Q'*n* = 1), then S = 1 and R = 1. Since S = 1 and R = 1, the basic flip-flop does not alter the state and remains in the reset state. But if the flip-flop is in set condition (*i.e.*, Q*n* = 1 and Q'*n* = 0), then S = 1 and R = 0. Since S = 1 and R = 0, the basic flip-flop changes its state and resets.

Case 3. When the clock is applied and J = 1 and K = 0 and the previous state of the flip-flop is reset (*i.e.*, Q*n* = 0 and Q'*n* = 1), then S = 0 and R = 1. Since S = 0 and R = 1, the basic flip-flop changes its state and goes to the set state. But if the flip-flop is already in set condition (*i.e.*, Q*n* = 1 and Q'*n* = 0), then S = 1 and R = 1. Since S = 1 and R = 1, the basic flip-flop does not alter its state and remains in the set state.

Case 4. When the clock is applied and J = 1 and K = 1 and the previous state of the flip-flop is reset (*i.e.*, Q*n* = 0 and Q'*n* = 1), then S = 0 and R = 1. Since S = 0 and R = 1, the basic flip-flop changes its state and goes to the set state. But if the flip-flop is already in set condition (*i.e.*, Q*n* = 1 and Q'*n* = 0), then S = 1 and R = 0. Since S = 1 and R = 0, the basic flip-flop changes its state and goes to the reset state. So we find that for J = 1 and K = 1, the flip-flop toggles its state from *set* to *reset* and vice versa. Toggle means to switch to the opposite state.

3.1 Master-Slave J-K Flip-flop

A master-slave (M-S) flip-flop is shown in Figure 2.4(d). Basically, a master-slave flip-flop is a system of two flip-flops—one being designated as *master* and the other is the *slave*. From the figure we see that a clock pulse is applied to the master and the inverted form of the same clock pulse is applied to the slave.

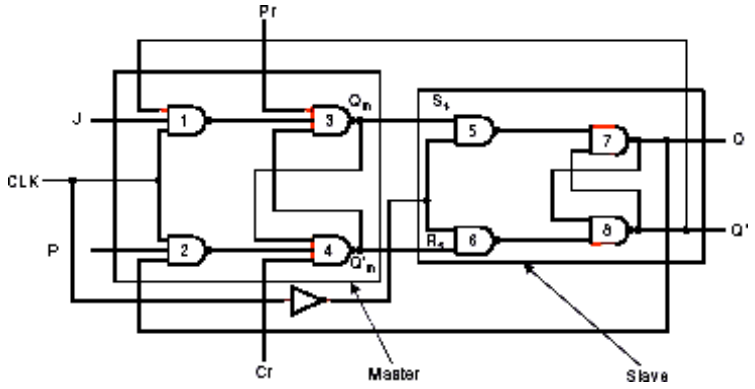


Figure 2.4(d): A master-slave J-K flip-flop.

When *CLK* = 1, the first flip-flop (*i.e.*, the master) is enabled and the outputs Q*m* and Q'*m* respond to the inputs J and K. At this time the second flip-flop (*i.e.*, the slave) is disabled because the *CLK* is LOW to the second flip- flop. Similarly, when CLK becomes LOW, the master becomes disabled and the slave becomes active, since now the CLK to it is HIGH. Therefore, the outputs Q and Q' follow the outputs Q*m* and Q'*m* respectively. Since the second flip-flop just follows the first one, it is referred to as a slave and the first one is called the master. Hence, the configuration is referred to as a master-slave (M-S) flip-flop.

In this type of circuit configuration the inputs to the gates 5 and 6 do not change at the time of application of the clock pulse. Hence the race-around condition does not exist. The state of the master-slave flip-flop, shown in Figure 2.4(d), changes at the negative transition (trailing edge) of the clock pulse. Hence, it becomes negative triggering a master-slave flip-flop. This can be changed to a positive edge triggering flip-flop by adding two inverters to the system—one before the clock pulse is applied to the master and an additional one in between the master and the slave. The logic symbol of a negative edge master-slave is shown in Figure 2.4(e).

The system of master-slave flip-flops is not restricted to J-K master-slave only. There may be an S-R master-slave or a D master-slave, etc., in all of them the slave is an S-R flip-flop, whereas the master changes to J-K or S-R or D flip-flops.

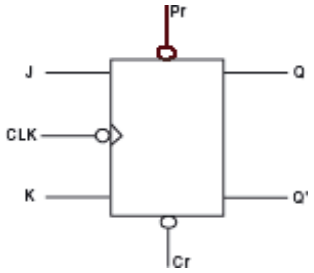


Figure 2.4(e): A negative edge-transition master-slave J-K flip-flop.

3.2 T Flip-Flop

With a slight modification of a J-K flip-flop, we can construct a new flip-flop called a T flip-flop. If the two inputs J and K of a J-K flip-flop are tied together it is referred to as a T flip-flop. Hence, a T flip-flop has only one input T and two outputs Q and Q'. The name T flip-flop actually indicates the fact that the flip-flop has the ability to toggle. It has actually only two states —*toggle state* and *memory state*. Since there are only two states, a T flip-flop is a very good option to use in counter design and in sequential circuits design where switching an operation is required. The truth table of a T flip-flop is given in Table 2.4(c).

Table 2.4(c): The truth table of a T flip-flop

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

If the T input is in 0 state (*i.e.*, J = K = 0) prior to a clock pulse, the Q output will not change with the clock pulse. On the other hand, if the T input is in 1 state (*i.e.*, J = K = 1) prior to a clock pulse, the Q output will change to Q' with the clock pulse. In other words, we may say that, if T = 1 and the device is clocked, then the output toggles its state.

The truth table shows that when T = 0, then Q*n*+1 = Q*n*, *i.e.*, the next state is the same as the present state and no change occurs. When T = 1, then Q*n*+1 = Q'*n*, *i.e.*, the state of the flip-flop is complemented. The circuit diagram of a T flip-flop is shown in Figure 2.4(f) and the block diagram of the flip-flop is shown in Figure 7.41.

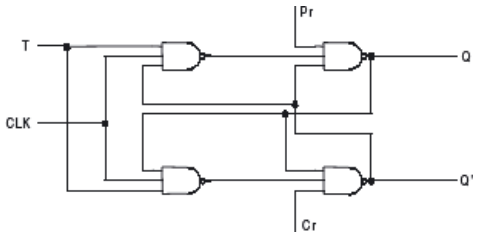


Figure 2.4(f): A T flip flop

The T flip-flop has one input, *T* (which stands for toggle), in addition to the clock. When *T* is asserted (*T* = 1), the flip-flop state toggles back and forth at each active edge of the clock, and when *T* is de-asserted, the flip-flop keeps its current state. The characteristic table, characteristic equation, state diagram, circuit, logic symbol, and excitation table for the T flip-flop are shown in Figure 2.3(f).

4.0 Conclusion

When latches are used for the memory elements in sequential circuits, a serious difficulty arises. Recall that latches have the property of immediate output responses (i.e., transparency). Because of this the output of a latch cannot be applied directly (or through logic) to the input of the same or another latch when all the latches are triggered by a common clock source. Flip-flops are used to overcome this difficulty.

5.0 Summary

In this unit, you learnt about:

- J-K Flip-Flop
- Master-Slave J-K Flip-flop
- T Flip-Flop

6.0 Tutor Marked Assignment

1. What is the difference between a latch and a flip-flop?
2. Draw the circuit diagram for the J-K flip-flop with enable.
3. Draw the circuit diagram for the T flip-flop with enable.
4. *True or False:* The slave will respond to changes in *J* and *K* while *CLK* = 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 3 - Counters

UNIT 1: Introduction to Counters

Contents **Pages**

1.0 Introduction

Counters are one of the simplest types of sequential networks. A counter is usually constructed from one or more flip-flops that change state in a prescribed sequence when input pulses are received. A counter driven by a clock can be used to count the number of clock cycles. Since the clock pulses occur at known intervals, the counter can be used as an instrument for measuring time and therefore period of frequency.

2.0 Objectives

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

- Understand Counters and types of counters
- Discuss the Binary Up Counter
- Discuss the Binary Up-down counter
- Understand the BCD Counter
- Discuss the BCD Up-down counter

3.0 TYPES OF COUNTERS

Counters, as the name suggests, are for counting a sequence of values. However, there are many different types of counters depending on the total number of count values, the sequence of values that it outputs, whether it counts up or down, and so on. The simplest is a modulo-*n* counter that counts the decimal sequence 0, 1, 2, …, up to *n*-1 and back to 0. Some typical counters are described next.

- **Modulo-*n* counter**: Counts from decimal 0 to $n - 1$ and back to 0. For example, a modulo-5 counter sequence in decimal is 0, 1, 2, 3, and 4.
- **Binary coded decimal (BCD) counter**: Just like a modulo-*n* counter, except that *n* is fixed at 10. Thus, the sequence is always from 0 to 9.
- ***n*-bit binary counter**: Similar to modulo-*n* counter, but the range is from 0 to $2n - 1$ and back to 0, where *n* is the number of bits used in the counter. For example, a 3-bit binary counter sequence in decimal is 0, 1, 2, 3,4, 5, 6, and 7.
- **Gray-code counter**: The sequence is coded so that any two consecutive values must differ in only one bit. For example, one possible 3-bit gray-code counter sequence is 000, 001, 011, 010, 110, 111, 101, and 100.
- **Ring counter**: The sequence starts with a string of 0 bits followed by one 1 bit, as in 0001. This counter simply rotates the bits to the left on each count. For example, a 4-bit ring counter sequence is 0001, 0010, 0100, 1000, and back to 0001.

We will now look at the design of several counters.

3.1 Binary Up Counter

An *n*-bit binary counter can be constructed using a modified *n*-bit register where the data inputs for the register come from an incrementer (adder) for an up counter, and a decrementer (subtractor) for a down counter. To get to the next up-count sequence from the value that is

stored in a register, we simply have to add a 1 to it. The full adder adds two operands plus the carry. But what we want is just to add a 1, so the second operand to the full adder is always a 1. Since the 1 can also be added in via the carry-in signal of the adder, we really do not need the second operand input.
This modified adder that only adds one operand with the carry-in is called a **half adder** (HA). Its truth table is shown in (a) of Figure 3.1(a). We have *a* as the only input operand, c_{in} and c_{out} are the carry-in and carry-out signals, respectively, and *s* is the sum of the addition. In the truth table, we are simply adding *a* plus c_{in} to give the sum *s* and possibly a carry-out, c_{out}. From the truth table, we obtain the two equations for c_{out} and *s* shown in (b) of Figure 3.1(a). The HA circuit is shown in (c) of Figure 3.1(a) and its logic symbol in (d).

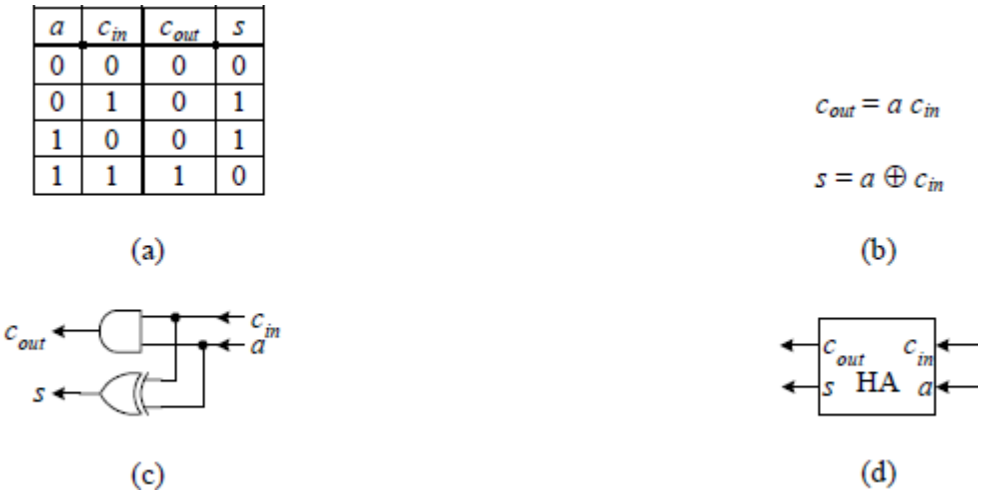
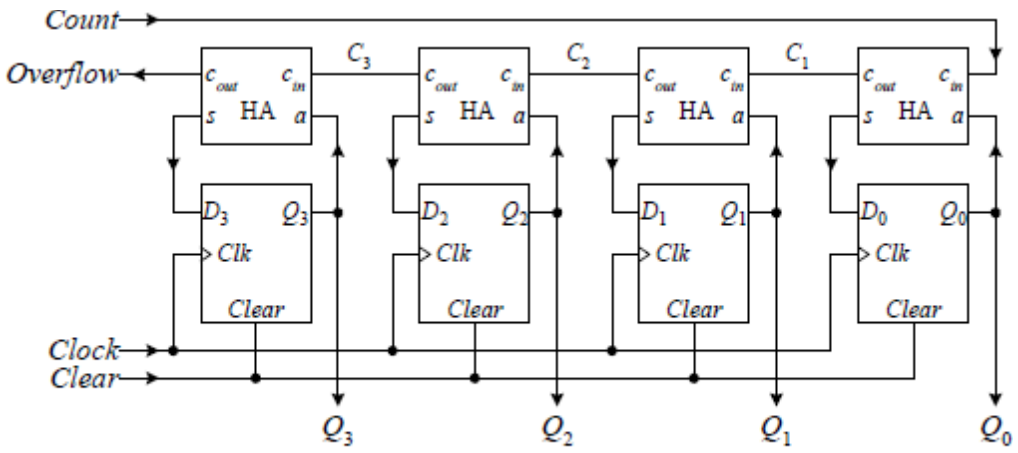


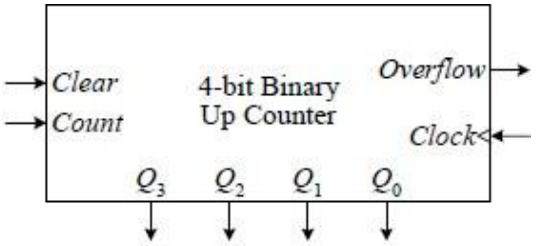
Figure 3.1(a): Half adder: (a) truth table; (b) equations; (c) circuit; (d) logic symbol.

Several half adders can be daisy-chained together, just like with the full adders to form an *n*-bit adder. The single operand input *a* comes from the register. The initial carry-in signal, *c*0, is used as the count enable signal, since a 1 on *c*0 will result in incrementing a 1 to the register value, and a 0 will not. The resulting 4-bit binary up counter circuit is shown in (a) of Figure 3.1(b), along with its operation table and logic symbol in (b) and (c). As long as *Count* is asserted, the counter will increment by 1 on each clock pulse until *Count* is de-asserted. When the count reaches 2*n* – 1 (which is equivalent to the binary number with all 1's) the next count will revert back to 0, because adding a 1 to a binary number with all 1's will result in an overflow on the *Overflow* bit, and all of the original bits will reset to 0. The *Clear* signal allows an asynchronous reset of the counter to 0.



Clear	Count	Operation
1	×	Reset counter to zero
0	0	No change
0	1	Count up

(b)



(c)

Figure 3.1(b): A 4-bit binary up counter with asynchronous clear: (a) circuit; (b) operation table; (c) logic symbol.

3.2 Binary Up-down Counter

We can design an *n*-bit binary up-down counter just like the up counter, except that we need both an adder and a subtractor for the data input to the register. The **half adder-subtractor** (HAS) truth table is shown in (a) of Figure 3.1(c).

The *Down* signal is to select whether we want to count up or down. Asserting *Down* (setting to 1) will count down. The top half of the table is exactly the same as the HA truth table. For the bottom half, we are performing a subtraction of $a - cin$, where s is the difference of the subtraction, and *cout* is a 1 if we need to borrow. For example, for 0 – 1, we need to borrow, so *cout* is a 1. When we borrow, we get a 2; and 2 – 1 = 1, so s is also a 1. The two resulting equations for *cout* and s are shown in (b) of Figure 3.1(c). The circuit and logic symbol for the half adder subtractor are shown in (c) and (d) of Figure 3.1(c).

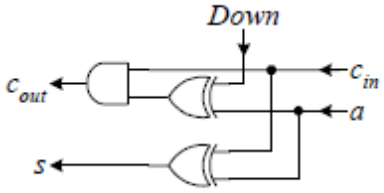
Down	a	c_{in}	c_{out}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	0
1	0	1	1	1
1	1	0	0	1
1	1	1	0	0

(a)

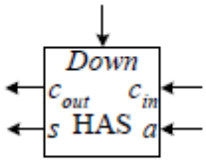
$$c_{out} = Down'\, a\, c_{in} + Down\, a'\, c_{in} = (Down \oplus a)\, c_{in}$$

$$s = Down'\, (a \oplus c_{in}) + Down\, (a \oplus c_{in}) = a \oplus c_{in}$$

(b)



(c)



(d)

Figure 3.1(c): Half Adder-Subtractor (HAS): (a) truth table; (b) equations; (c) circuit; (d) logic symbol.

We can simply replace the HAs with the HASs in the up counter circuit to get the up-down counter circuit, as shown in Figure 8.15(a). Its operation table and logic symbol are shown in Figure 8.15(b) and (c). Again, the *Overflow* signal is asserted each time the counter rolls over from 1111 back to 0000.

3.3 BCD Up Counter

A limitation with the binary up-down counter with parallel load is that it always counts up to $2n - 1$ for an n bit register and then cycles back to zero. If we want the count sequence to end at a number less than $2n - 1$, we need to use an equality comparator to test for this new ending number. The comparator compares the current count value that is in the register with this new ending number. When the counter reaches this new ending number, the comparator asserts its output.

The counter can start from a number that is initially loaded in. However, if we want the count sequence to cycle back to this new starting number each time, we need to assert the *Load* signal at the end of each count sequence and reload this new starting number. The output of the comparator is connected to the *Load* line, so that when the counter reaches the ending number, it will assert the *Load* line and loads in the starting number. Hence, the counter can end at a new ending number and cycles back to a new starting number.

The binary coded decimal (BCD) up counter counts from 0 to 9 and then cycles back to 0. The circuit for it is shown in Figure 3.1(d). The heart of the circuit is just the 4-bit binary up-down

counter with parallel load. A 4-input AND gate is used to compare the count value with the number 9. When the count value is 9, the AND gate comparator outputs a 1 to assert the *Load* line. Once the *Load* line is asserted, the next counter value will be the value loaded in from the counter input *D*. Since *D* is connected to all 0's, the counter will cycle back to 0 at the next rising clock edge. The *Down* line is connected to a 0, since we only want to count up.

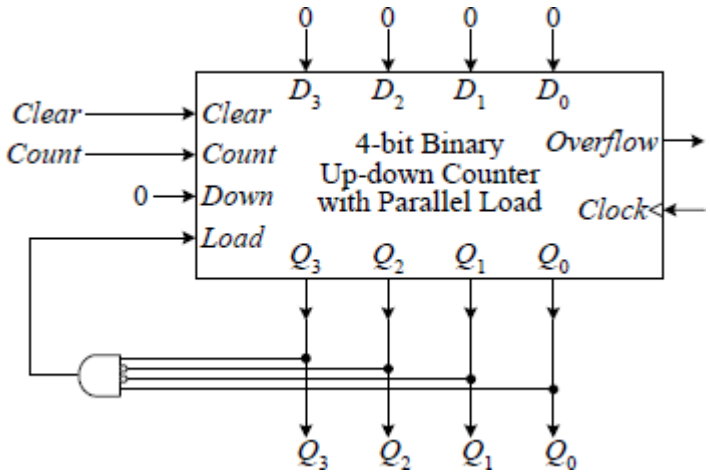


Figure 3.1(d): BCD up counter.

In order for the timing of each count to be the same, we must use the *Load* operation to load in the value 0, rather than using the *Clear* operation. If we connect the output of the AND gate to the *Clear* input instead of the *Load* input, we will still get the correct count sequence. However, when the count reaches 9, it will change to a 0 almost immediately, because when the output of the AND gate asserts the asynchronous *Clear* signal, the counter is reset to 0 right away and not at the next rising clock edge.

Example 3.1(a): Constructing an up counter circuit
This example uses the 4-bit binary up-down counter with parallel load to construct an up counter circuit that counts from 3 to 8 (in decimal), and back to 3.
The circuit for this counter, shown in Figure 3.1(e), is almost identical to the BCD up counter circuit. The only difference is that we need to test for the number 8 instead of 9 as the last number in the sequence, and the first number to load in is a 3 instead of a 0. Hence, the inputs to the AND gate for comparing with the binary counter output is 1000, and the number for loading in is 0011.

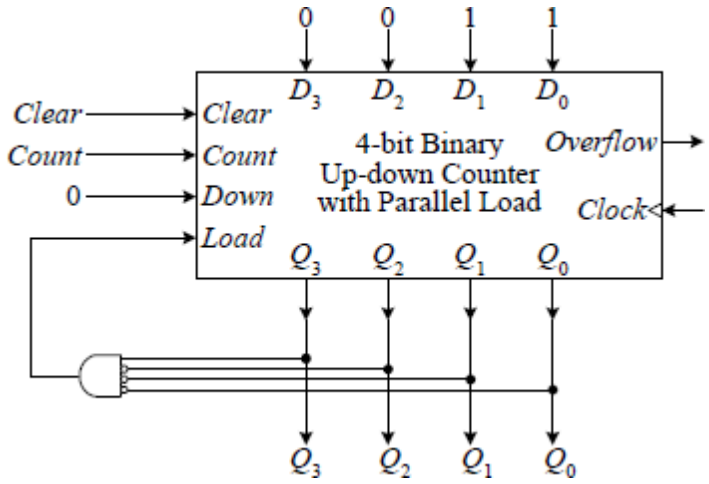


Figure 3.1(e): Counter for Example 3.1(a)

3.4 BCD Up-down Counter

We can get a BCD up-down counter by modifying the BCD up counter circuit slightly. The counter counts from 0 to 9 for the up sequence and 9 down to 0 for the down sequence. For the up sequence, when the count reaches 9, the *Load* line is asserted to load in a 0 (0000 in binary). For the down sequence, when the count reaches 0, the *Load* line is asserted to load in a 9 (1001 in binary).

The BCD up-down counter circuit is shown in Figure 3.1(f). Two 5-input AND gates acting as comparators are used. The one labeled "Up" will output a 1 when *Down* is de-asserted (i.e., counting up), and the count is 9. The one label "Dn" will output a 1 when *Down* is asserted, and the count is 0. The *Load* signal is asserted by either one of these two AND gates. Four 2-to-1 multiplexers are used to select which of the two starting values, 0000 or 1001, is to be loaded in when the *Load* line is asserted. The select lines for these four multiplexers are connected in common to the *Down* signal, so that when the counter is counting up, 0000 is loaded in when the counter wraps around, and 1001 is loaded in when the counter wraps around while counting down. It should be obvious that the two values, 0000 and 1001, can also be loaded in without the use of the four multiplexers.

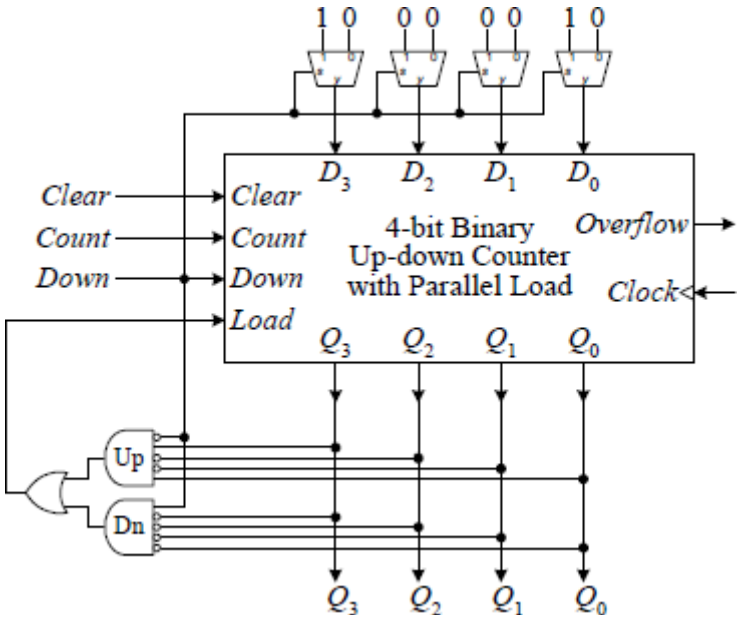


Figure 3.1(f): BCD up-down counter.

Example 3.1(b): Constructing an up-down counter circuit

This example uses the 4-bit binary up-down counter with parallel load to construct an up-down counter circuit that outputs the sequence, 2, 5, 9, 13, and 14, repeatedly.

The 4-bit binary counter can only count numbers consecutively. In order to output numbers that are not consecutive, we need to design an output circuit that maps from one number to another number. The required sequence has five numbers, so we will first design a counter to count from 0 to 4. The output circuit will then map the numbers, 0, 1, 2, 3, and 4 to the required output numbers, 2, 5, 9, 13, and 14, respectively.

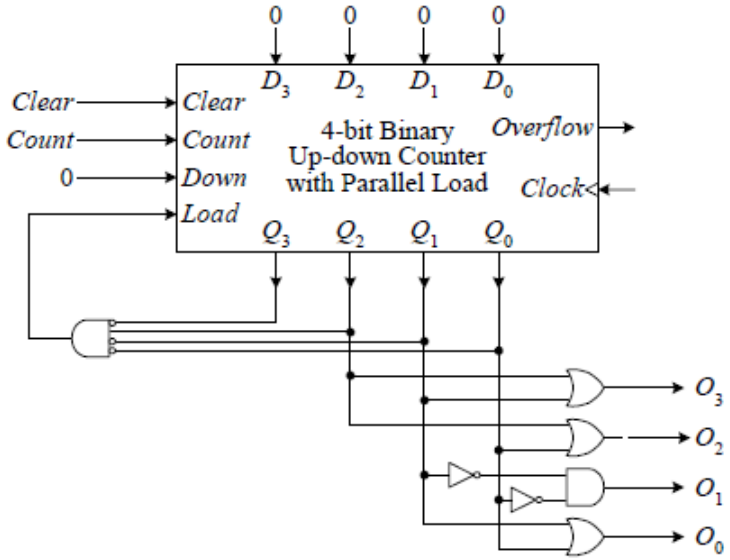
The inputs to the output circuit are the four output bits of the counter, Q_3, Q_2, Q_1, and Q_0. The outputs from this circuit are the modified four bits, O_3, O_2, O_1, and O_0, for representing the five output numbers. The truth table and the resulting output equations for the output circuit are shown in (a) and (b) of Figure 3.1(g), respectively. The easiest way to see how the output equations are obtained is to use a K-map and put in all of the don't-cares. The complete counter circuit is shown in (c) of Figure 3.1(g).

Decimal Input	Q_3	Q_2	Q_1	Q_0	Decimal Output	O_3	O_2	O_1	O_0
0	0	0	0	0	2	0	0	1	0
1	0	0	0	1	5	0	1	0	1
2	0	0	1	0	9	1	0	0	1
3	0	0	1	1	13	1	1	0	1
4	0	1	0	0	14	1	1	1	0
Rest of the Combinations						×	×	×	×

(a)

$O_0 = Q_1 + Q_0$
$O_1 = Q_1'Q_0'$
$O_2 = Q_2 + Q_0$
$O_3 = Q_2 + Q_1$

(b)



(c)

Figure 3.1(g): Counter for Example 3.1

4.0 Conclusion

Counter is a fundamental part of most digital logic applications. It is used in timing units, control circuits, signal generators and numerous other devices. Down counters are not as widely used as up counters. Their major application is in situations where it must be known when a desired number of input pulses has occurred. In these situations, the down counter is preset to the desired number and then allowed to count down as the pulses are applied. When the counter reaches the zero state it is detected by a logic gate whose output then indicates that the preset number of pulses has occurred.

5.0 Summary

In this unit, you learnt about:

- Counters and types of counters including:
  - Binary Up Counter

- Binary Up-down Counter
- BCD Up Counter
- BCD Up-down Counter

6.0 Tutor Marked Assignment

1) How can you convert an up-counter into a down-counter?
2) Which flip-flop is best suited for designing a counter and why?
3) What is meant when we say that a counter is presettable?
4) Design a 4-bit counter with one control signals S. The counter should operate as
   (a) Binary down-counter when S = 0,
   (b) Binary up-counter when S = 1.

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 3 - Counters

UNIT 2: Asynchronous Counter

Contents **Pages**

1.0 Introduction

Asynchronous or *ripple counters* are counter circuits made from cascaded J-K flip-flops where each clock input receives its pulses from the output of the previous flip-flop invariably exhibit a *ripple effect*, where false output counts are generated between some steps of the count sequence.

2.0 Objectives

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

- Understand Asynchronous Counters

3.0 About Asynchronous (Serial or Ripple) Counters

The simplest counter circuit can be built using T flip-flops because the toggle feature is naturally suited for the implementation of the counting operation. J-K flip-flops can also be used with the *toggle* property in hand. Other flip-flops like D or S-R can also be used, but they may lead to more complex designs.

In this counter all the flip-flops are not driven by the same clock pulse. Here, the clock pulse is applied to the first flip-flop; *i.e.*, the least significant bit state of the counter and the successive flip-flop is triggered by the output of the previous flip-flop. Hence the counter has cumulative settling time, which limits its speed of operation. The first stage of the counter changes its state first with the application of the clock pulse to the flip-flop and the successive flip-flops change their states in turn causing a *ripple-through* effect of the clock pluses. As the signal propagates through the counter in a *ripple* fashion, it is called a *ripple counter*.

3.1 Asynchronous (or Ripple) Up-counter

Figure 3.2(a) shows a 3-bit counter capable of counting from 0 to 7. The clock inputs of the three flip-flops are connected in cascade. The T input of each flip-flop is connected to a constant 1, which means that the state of the flip-flop will toggle (reverse) at each negative edge of its clock. We are assuming that the purpose of this circuit is to count the number of pulses that occur on the primary input called CLK (Clock). Thus, the clock input of the first flip-flop is connected to the *Clock* line. The other two flip-flops have their clock inputs driven by the Q output of the preceding flip-flop. Therefore, they toggle their state whenever the preceding flip-flop changes its state from Q = 1 to Q = 0, which results in a negative edge of the Q signal.

Figure 3.2(*b*) shows a timing diagram for the counter. The value of Q0 toggles once each clock cycle. The change takes place shortly after the negative edge of the *Clock* signal.

The delay is caused by the propagation delay through the flip-flop. Since the second flip-flop is clocked by Q_0, the value of Q_1 changes shortly after the negative edge of the Q0 signal.

Similarly, the value of Q_2 changes shortly after the negative edge of the Q_1 signal. If we look at the values Q_2 Q_1 Q_0 as the count, then the timing diagram indicates that the counting sequence is 0, 1, 2, 3, 4, 5, 6, 7, 0, 1, 2, and so on. This circuit is a modulo-8 counter. Since it counts in the upward direction, we call the circuit an *up-counter*.

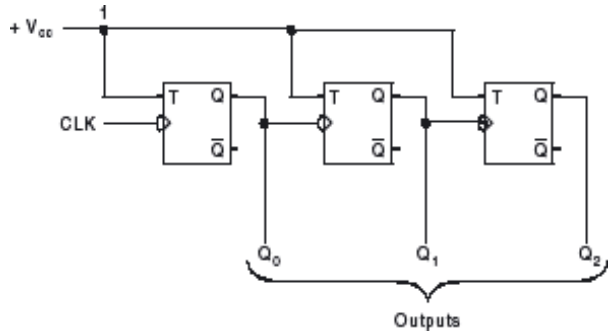


Figure 3.2(a): Logic circuit diagram of a 3-bit asynchronous up-counter.

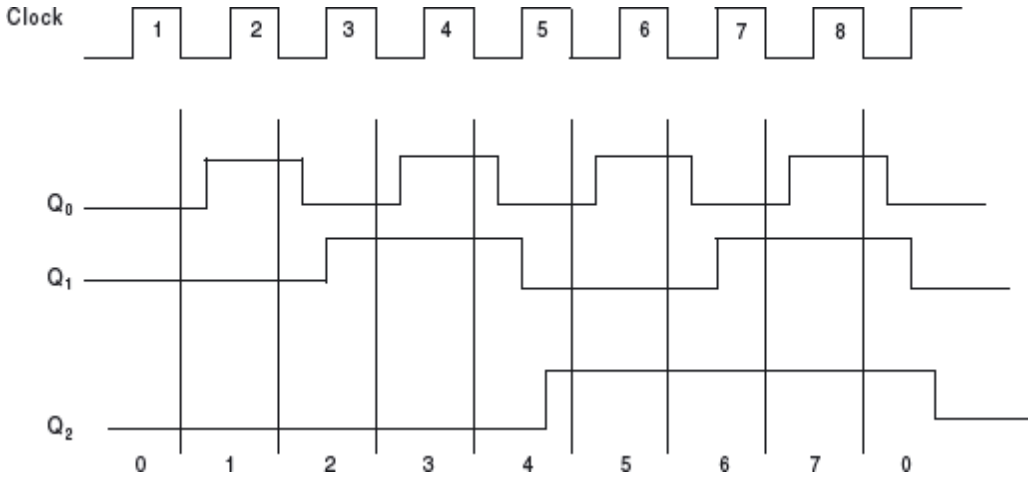


Figure 3.2(b): Timing diagram

The counter in Figure 3.2(a) has three stages, each comprising of a single flip-flop. Only the first stage responds directly to the Clock signal. Hence we may say that this stage is synchronized to the clock. The other two stages respond after an additional delay. For example, when count = 3, the next clock pulse will change the count to 4. Now this change requires all three flip-flops to toggle their states. The change in Q_0 is observed only after a propagation delay from the negative edge of the clock pulse. The Q_1 and Q_2 flip-flops have not changed their states yet. Hence, for a brief period, the count will be $Q_2Q_1Q_0 = 010$.

The change in Q1 appears after a second propagation delay, and at that point the count is $Q_2Q_1Q_0 = 000$. Finally, the change in Q_2 occurs after a third delay, and hence the stable state of the circuit is reached and the count is $Q_2Q_1Q_0 = 100$.

Table 3.2(a) shows the sequence of binary states that the flip-flops will follow as clock pulses are applied continuously. An *n*-bit binary counter repeats the counting sequence for every 2*n* (*n* = number of flip-flops) clock pulses and has discrete states from 0 to 2^n-1.

Table 3.2(a): Count sequence of a 3-bit binary ripple up-counter

Counter State	Q_2	Q_1	Q_0
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Figure 3.2(c) shows the 3-bit binary ripple counter with decoded outputs. It consists of the same circuit as shown in Figure 3.2(a) with additional decoding circuitry. In decoding the states of a ripple counter, pulses of one clock duration will occur at the decoding gate outputs as the flip-flops change their state when the counter content is equal to the given state. For example, a decoding gate Q_6 connected in the circuit will decode state 6 (*i.e.*, $Q_AQ_BQ_C = 110$). Thus the gate output will be high only when $Q_A = 1$, $Q_B = 1$, and $Q_C = 0$. The remaining seven states of the 3-bit counter can be decoded in a similar manner using AND gates as Q_0, Q_1, Q_2, Q_3, Q_4, Q_5, and Q_7.

Now, theoretically each decoding output will be high only when the counter content is equal to a given state, and this state occurs only once during a cycle of 2*n* states of the counter, where *n* is the number of flip-flops in the counter. But practically in an asynchronous counter, the decoding gate produces a high output more than once during the cycle of 2*n* states. Such undesired high or low pulses of short duration, that appear at the decoding gate output at undesired time instants are known as *spikes* or *glitches*. The reason for these spikes is the cumulative propagation delay in the synchronous counter.

As TTL circuits are very fast, they will respond to even glitches of very small duration (a few nanoseconds). Therefore, these glitches should be eliminated. These can be eliminated by using any one of the following methods: (*i*) clock input to strobe the decoding gates, or (*ii*) using synchronous counters.

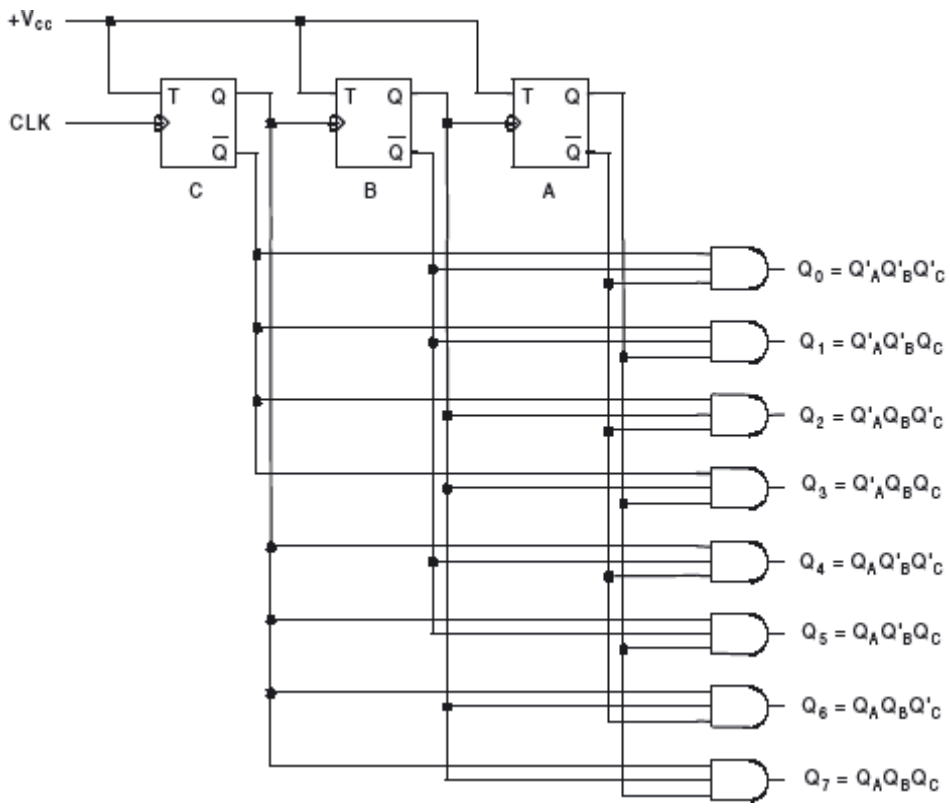


Figure 3.2(c): 3-bit binary asynchronous counter with decoded outputs

3.2 Asynchronous (or Ripple) Down-counter

Figure 3.2(a) is an up-counter because it counts upward from zero. It is a relatively simple matter to construct asynchronous down counters, which will count downward fro a maximum count to zero. Before looking at a ripple down counter, let us examine the count-down sequence for a 3-bit down counter:

	CBA	CBA	CBA
(7)	1 1 1	1 1 1	1 1 1
(6)	1 1 0	1 1 0	1 1 0
(5)	1 0 1	1 0 1	. . .
(4)	1 0 0	1 0 0	. . .
(3)	0 1 1	0 1 1	etc
(2)	0 1 0	0 1 0	etc
(1)	0 0 1	0 0 1	
(0)	0 0 0	0 0 0	etc.

A, B, and *C* represent the flip-flop output states as the counter goes through its sequence.

It can be seen that the A flip-flop (LSB) changes states (toggles) at each in the sequence just as it does in the up counter. The B flip-flop changes states each time *A* goes from LOW to HIGH; *C* changes states each time *B* goes from LOW to HIGH. Thus, in a down counter each flip-flop, except the first, must toggle when the preceding flip-flop goes from LOW to HIGH. If the flip-flops have *CLK* inputs that respond to negative transitions (HIGH to LOW), then an inverter can be placed in front of each *CLK* input; however, the same effect can be accomplished by driving each flip-flop *CLK* input from the inverted output of the preceding flip-flop. This is illustrated in Figure 3.2(d) for a MOD-8 down counter.

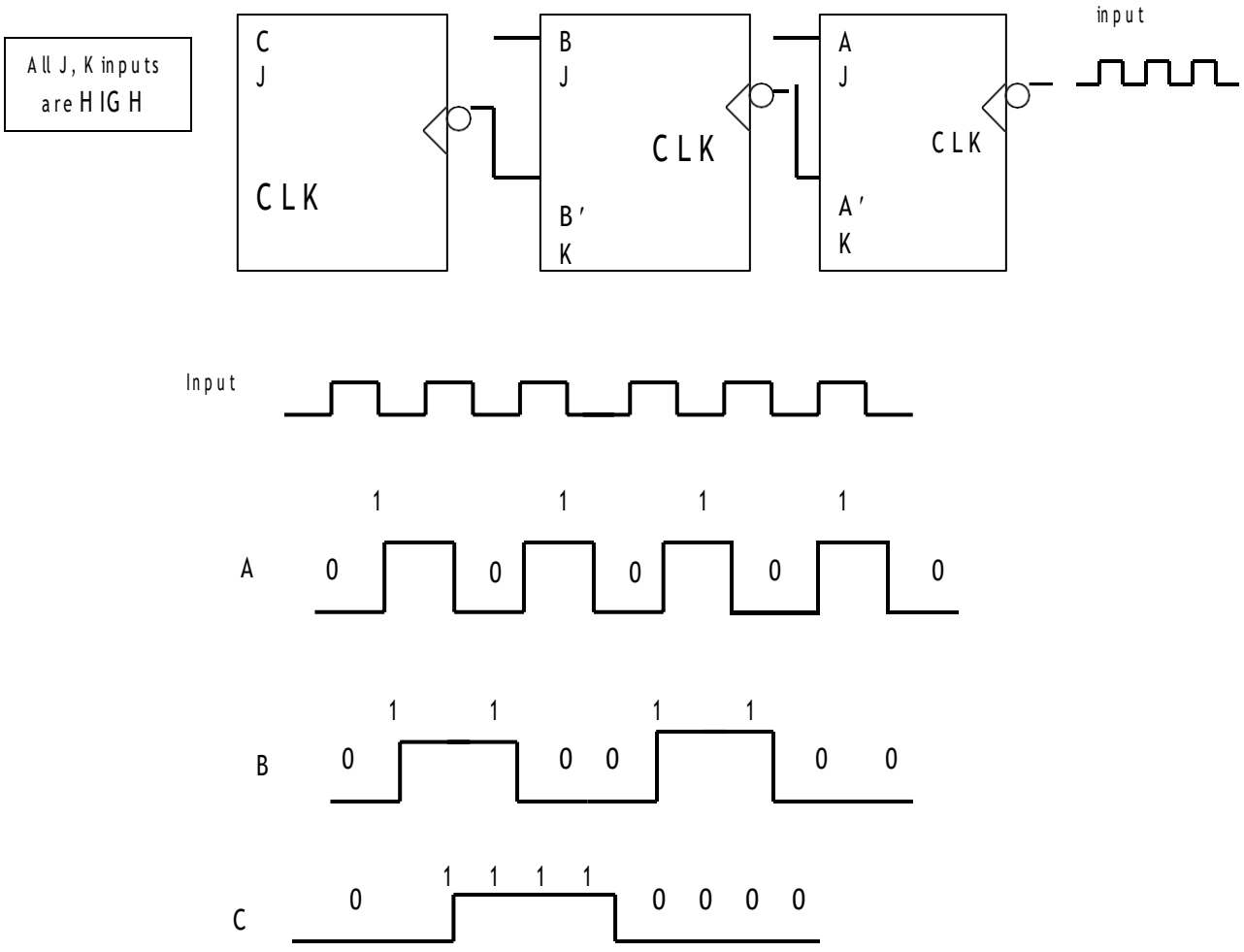


Figure 3.2(d): MOD-8 down counter

The input pulses are applied to the A flip-flop; the A′ output serves as the *CLK* input for the B flip-flop; the B′ output serves as the *CLK* input for the *C* flip-flop. The waveforms at A, B, and C show that B toggles whenever B goes LOW to HIGH. This results in the desired down-counting sequence at the *C, B,* and *A* outputs.

3.3 MOD Number

The counter in Figure 3.2(c) has 16 distinctly different states (0000 through 1111). Thus, it is a MOD-16 ripple counter. Recall that the MOD number is always equal to the number of states which the counter goes through in each complete cycle before it recycles back to its starting state. The MOD number can be increased simply by adding more Flip-Flops to the counter. That is, MOD number = 2^N where N is the number of Flip-Flops connected in the arrangement.

Example 3.1(a): A counter is needed that will count the number of items passing on a conveyor belt. A photocell and light source combination is used to generate a single pulse each time an item crosses its path. The counter has to be able to count as many as one thousand items. How many flip-flops are required?

Solution: It is a simple matter to determine what value of N is needed so that $2^N \geq 1000$. Since $2^9 = 512$, 9 Flip-Flops will not be enough. $2^{10} = 1024$, so 10 flip-flops would produce a counter that could count as high as $1111111111_2 = 1023_{10}$. Therefore, we should use 10 flip-flops. We could use more than 10, but it would be a waste of flip-flops, since any flip-flops past the 10^{th} one will never be toggled.

3.3.1 Counters with MOD numbers $< 2^N$

The basic ripple counter of Figure 3.2(a) is limited to MOD numbers that are equal to 2^N, where N is the number of flip-flops. This value is actually the maximum MOD number that can be obtained using N flip-flops. The basic counter can be modified to produce MOD numbers less than 2^N by allowing the counter to *skip states* that are normally part of the counting sequence. One of the most common methods for doing this is illustrated in Figure 3.2(e) where a 3-bit ripple counter is shown.

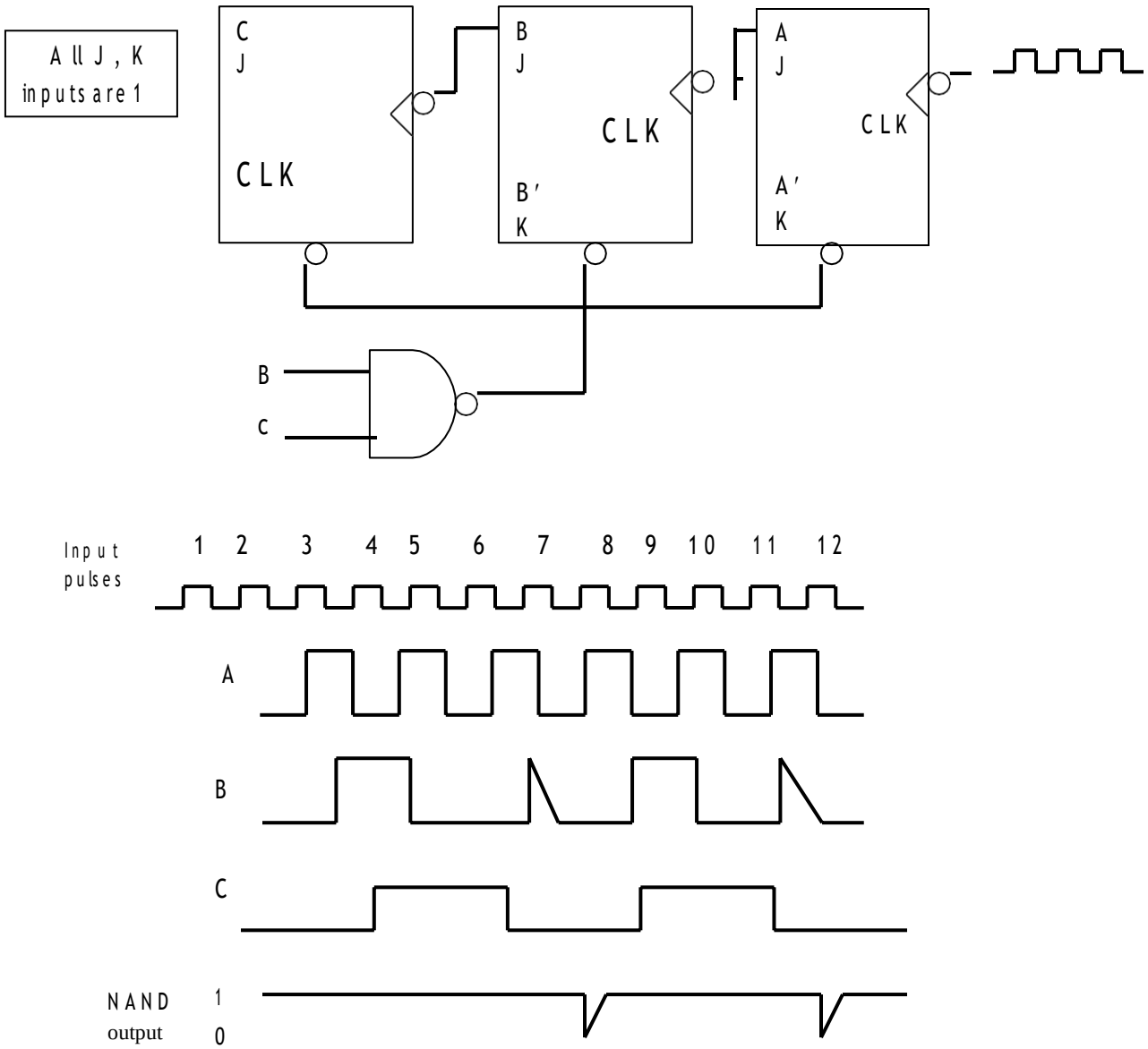


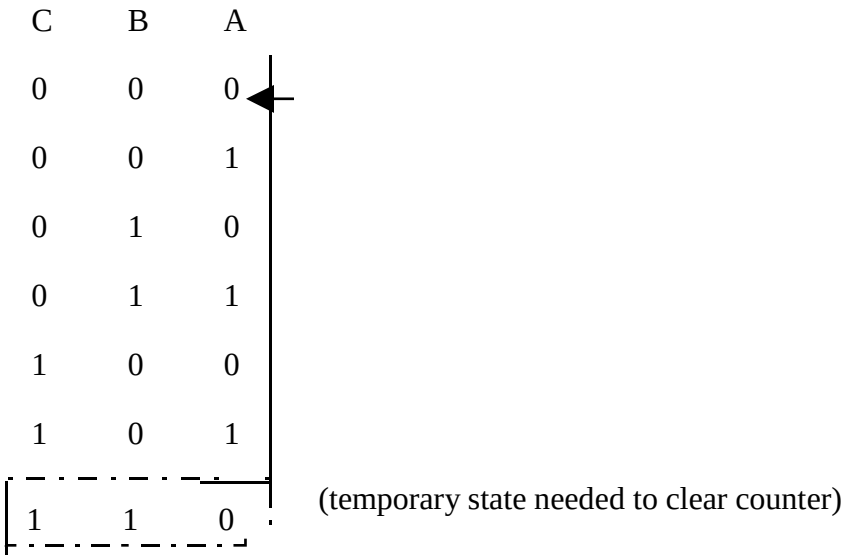
Figure 3.2(e): MOD-6 produced by clearing s MOD-8 counter when count of six (110) occurs

Disregarding the NAND gate for a moment we can see that the counter is a MOD-8 binary counter which will count in sequence from 000 to 111. However, the presence of the NAND gate will alter this sequence as follows:

1. The NAND output is connected to the asynchronous CLEAR inputs of each Flip-flop. As long as the NAND output is HIGH, it will have no effect on the counter. When it goes LOW, however, it will clear all the flip-flops so that the counter immediately goes to the 000 state.

2. The inputs to the NAND gate are the outputs of the *B* and *C* flip-flops, so the NAND output will go LOW whenever $B = C = 1$. This condition will occur when the counter goes from the 101 state to the 110 state (input pulse 6 on waveforms). The LOW at the NAND output will immediately (generally within a few nanoseconds) clear the counter to the 000 state. Once the flip-flops have been cleared, the NAND output goes back HIGH, since the $B = C = 1$ condition no longer exists.

3. The counting sequence is therefore,



Although the counter does go to the 110 state, it remains there for only a few nanoseconds before it recycles to 000. Thus, we can essentially say that this counter counts from 000 (zero) to 101 (five) and then recycles to 000. It essentially skips 110 and 111 so that it only goes through six different states; thus, it is a MOD-6 counter.

3.3.2 Changing the MOD number

The counter of Figure 3.2(d) is a MOD-6 because of the choice of inputs to the NAND gate. Any desired MOD number can be obtained by changing these inputs. For example, using a three-input NAND gate with inputs *A, B,* and *C,* the counter would function normally until the 111 condition was reached, at which point it would immediately reset to the 000 state. Ignoring the temporary excursion into the 111 state, the counter would go from 000 through 110 and then recycle back to 000, resulting in a MOD-7 counter (seven states).

Self Assessment Exercise:

1. How many Flip-flops are required for a counter that will count 0 to 255_{10}?

2. What is the MOD number of this counter?

3. What is the difference between the counting sequence of an up counter and a down counter?

4.0 Conclusion

The ripple or asynchronous counter is simple and straightforward in operation and construction and usually requires a minimum amount of hardware. In asynchronous counters, each flip-flop is triggered by the previous flip-flop, and hence the speed of operation is limited. However, the asynchronous counter highest operating frequency is limited because of ripple action. This problem can be overcome, if all flip-flops are clocked synchronously. The resulting circuit is known as a synchronous counter. Down counters are not as widely used as up counters. Their major application is in situations where it must be known when a desired number of pulses has occurred. In these situations the down counter is preset to the desired number and then allowed to count down as the pulses are applied. When the counter reaches the zero state it is detected by a logic gate whose output then indicates that the preset number of pulses has occurred.

5.0 Summary

In this unit, you learnt about:

- Asynchronous (Serial or Ripple) Counters
- How NAND gate can be used to construct counters

6.0 Tutor Marked Assignment

1) Construct a MOD-10 counter that will count from 0000 through 1001 (decimal 9).
2) Compare between a ripple and a synchronous counter.
3) Discuss the procedure for designing synchronous counters.
4) In an asynchronous counter, all Flip-Flops change states at the same time.

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 3 - Counters

UNIT 3: Synchronous Counter

Contents **Pages**

1.0 Introduction

A *synchronous counter*, in contrast to an *asynchronous counter*, is one whose output bits change state simultaneously, with no ripple. The only way we can build such a counter circuit from J-K flip-flops is to connect all the clock inputs together, so that each and every flip-flop receives the exact same clock pulse at the exact same time.

2.0 Objectives

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

- Understand Synchronous Counter with Ripple Carry
- Understand Synchronous Down-Counter
- Understand Synchronous Up-Down Counter
- Application of Counters

3.0 Why Synchronous (Parallel) Counters?

The problems encountered with ripple counters are caused by the accumulated Flip-flop propagation delays; stated another way, the Flip-flops do not all change states simultaneously in synchronism with the input pulses. These limitations can be overcome with the use of synchronous or parallel counters in which all the Flip-flops are triggered simultaneously (in parallel) by the clock input pulses. Since the input pulses are applied to all the Flip-flops, some means must be used to control when a Flip-flop is to toggle and when it is to remain unaffected by a clock pulse. This is accomplished by using the *J* and *K* inputs.

3.1 Synchronous (Parallel) Counters

The ripple or asynchronous counter is the simplest to build, but its highest operating frequency is limited because of ripple action. Each flip-flop has a delay time. In ripple counters these delay times are additive and the total "settling" time for the counter is approximately the product of the delay time of a single flip-flop and the total number of flip-flops. Again, there is the possibility of glitches occurring at the output of decoding gates used with a ripple counter.

Both of these problems can be overcome, if all the flip-flops are clocked synchronously. The resulting circuit is known as a *synchronous counter*. Synchronous counters can be designed for any count sequence (need not be straight binary). These can be designed following a systematic approach. Before we discuss the formal method of design for such counters, we shall consider an intuitive method.

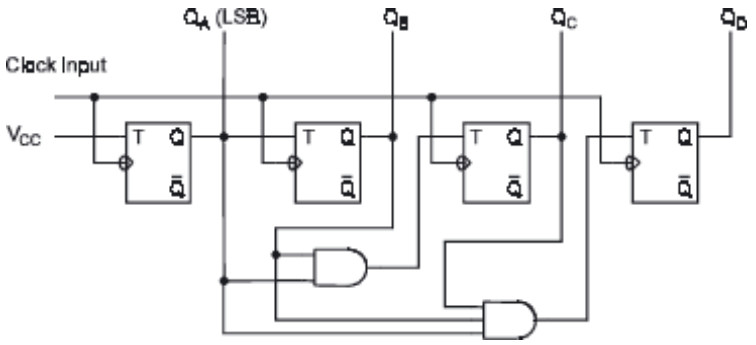


Figure 3.3(a): A 4-bit (MOD-16) synchronous counter.

A 4-bit synchronous counter with parallel carry is shown in Figure 3.3(a). In this circuit the clock inputs of all the flip-flops are tied together so that the input clock signal may be applied simultaneously to each flip-flop. Only the LSB flip-flop A has its T input connected permanently to logic 1 (*i.e.*, VCC), while the T inputs of the other flip-flops are driven by some combination of flip-flop outputs. The T input of flip-flop B is connected to the output Q_A of flip-flop A; the T input of flip-flop C is connected with the AND-operated output of Q_A and Q_B. Similarly, the T input of D flip-flop is connected with the AND-operated output of Q_A, Q_B, and Q_C.

From the circuit, we can see that flip-flop A changes its state with the negative transition of each clock pulse. Flip-flop B changes its state only when the value of QA is 1 and a negative transition of the clock pulse takes place. Similarly, flip-flop C changes its state only when both Q_A and Q_B are 1 and a negative edge transition of the clock pulse takes place. In the same manner, the flip-flop D changes its state when $Q_A = Q_B = Q_C = 1$ and when there is a negative transition at clock input. The count sequence of the counter is given in Table 3.3(a).

Table 3.3(a): Count Sequence of a 4-bit binary synchronous counter

State	Q_D	Q_C	Q_B	Q_A
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1
0	0	0	0	0

3.2 Synchronous Counter with Ripple Carry

The 4-bit synchronous counter discussed in the previous unit is said to be a s*ynchronous counter with parallel carry*. Moreover, in this type of counter, as the number of stages increases, the number of AND gates also increases, along with the number of inputs for each of those AND gates. This is a certain disadvantage for such type of circuits. Now this problem can be eliminated if we use the *synchronous counter with ripple carry* shown in Figure 3.3(b).

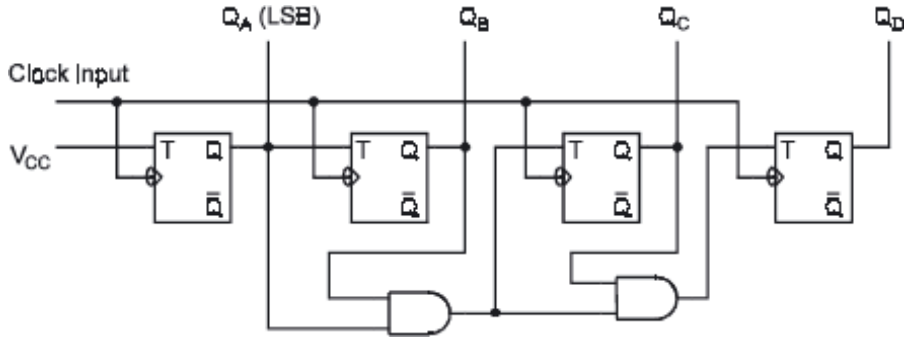


Figure 3.3(b): A 4-bit synchronous counter with ripple carry

But in such circuits the maximum clock frequency of the counter is reduced. This reduction of the maximum clock frequency is due to the delay through control logic which is now 2_{tg} instead of t_g which was achieved with parallel carry. The maximum clock frequency for an *n*-bit synchronous counter with ripple carry is given by

$$f_{\max} = \frac{1}{t_p + (n - 2)t_g}$$

where n = number of flip-flop stages.

3.3 Synchronous Down-Counter

A parallel down-counter can be made to count down by using the inverted outputs of flip-flops to feed the various logic gates. Even the same circuit may be retained and the outputs may be taken from the complement outputs of each flip-flop.

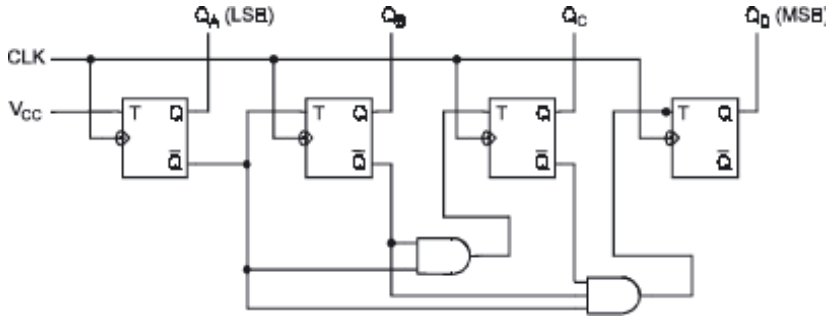


Figure 3.3(c): A 4-bit synchronous down-counter.

The parallel counter shown in Figure 3.3(b) can be converted to a down-counter by connecting the Q′A, Q′B, and Q′C outputs to the AND gates in place of QA, QB, and QC respectively as shown in Figure 3.3(c). In this case the count sequences through which the counter proceeds will be as shown in Table 3.3(b).

Table 3.3(b): Count Sequence of a 4-bit synchronous down-counter

State	Q_D	Q_C	Q_B	Q_A
15	1	1	1	1
14	1	1	1	0
13	1	1	0	1
12	1	1	0	0
11	1	0	1	1
10	1	0	1	0
9	1	0	0	1
8	1	0	0	0
7	0	1	1	1
6	0	1	1	0
5	0	1	0	1
4	0	1	0	0
3	0	0	1	1
2	0	0	1	0
1	0	0	0	1
0	0	0	0	0
15	1	1	1	1

3.4 Synchronous Up-Down Counter

Combining both the functions of up- and down-counting in a single counter, we can make a synchronous up-down counter as shown in Figure 3.3(d).

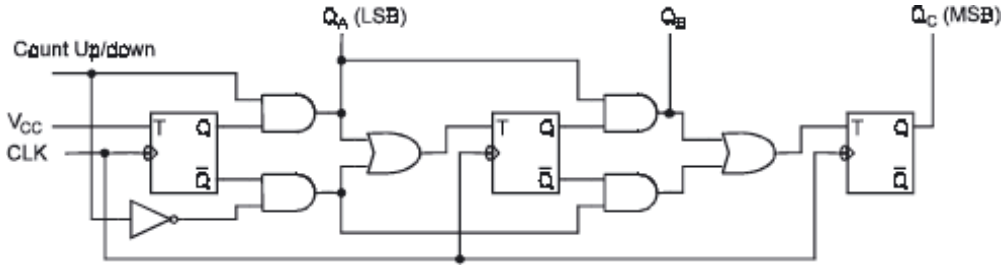


Figure 3.3(d): A MOD-8 synchronous up-down-counter

Here the control input (count-up/down) is used to allow either the normal output or the inverted output of one flip-flop to the T input of the following flip-flop. Two separate control lines (count-Up and count-down) could have been used but in such case we have to be careful that both of the lines cannot be simultaneously in the high state. When the count-up/down line is

high, then the upper AND gates will be active and the lower AND gates will remain inactive and hence the normal output of each flip-flop is carried forward to the following flip-flop. In such case, the counter will count from 000 to 111. On the other hand, if the control line is low, then the upper AND gates remain inactive, while the lower AND gates will become active. So the inverted output comes into operation and the counter counts from 111 to 000.

3.5 Application of Counters Digital Clock

A digital clock, which displays the time of day in hours, minutes, and seconds, is one of the most common applications of counters. To construct an accurate digital clock, a very highly controlled basic clock frequency is required. For battery-operated digital clocks (or watches) the basic frequency can be obtained from a quartz-crystal oscillator. Digital clocks operated from the AC power line can use the 50 Hz power frequency as the basic clock frequency. In either case, the basic frequency has to be divided down to a frequency of 1 Hz or pulse of 1 second (pps). The basic block diagram for a digital clock operating from 50 Hz is shown in Figure 3.3(e).

The 50 Hz signal is sent through a Schmitt trigger circuit to produce square pulses at the rate of 50 pps. The 50 pps waveform is fed into a MOD-50 counter, which is used to divide the 50 pps down to 1 pps. The 1-pps signal is then fed into the SECONDS section.

This section is used to count and display seconds from 0 through 59. The BCD counter advances one count per second. After 9 seconds the BCD counter recycles to 0. This triggers the MOD-6 counter and causes it to advance one count. This continues for 59 seconds. At this point, the BCD counter is at 1001 (9) count and the MOD-6 counter is at 101 (5).

Hence, the display reads 59 seconds. The next pulse recycles the BCD counter to 0. This, in turn, recycles the MOD-6 counter to 0.

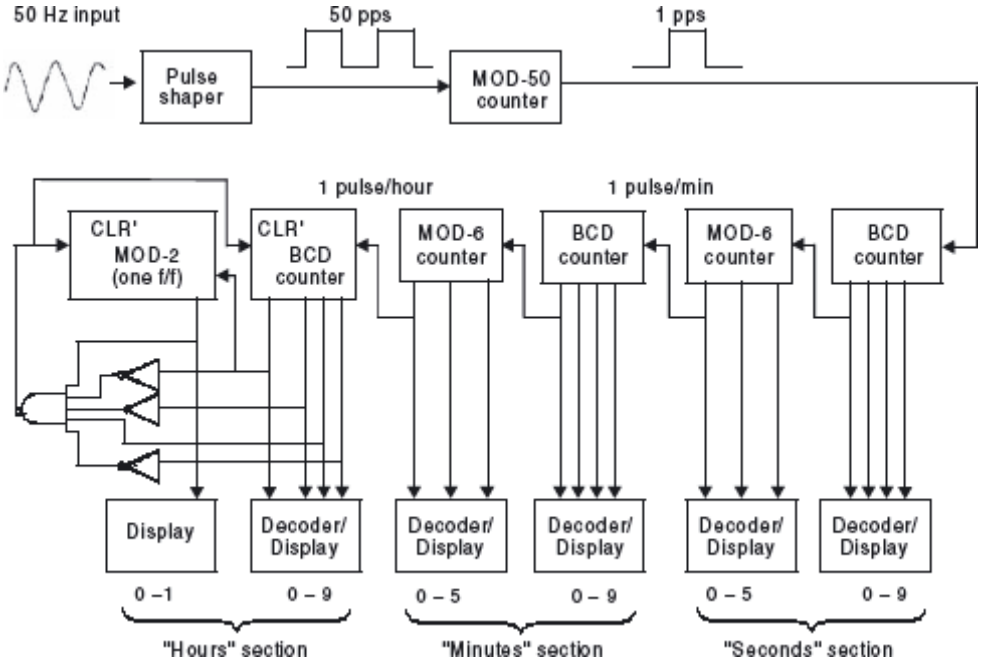


Figure 3.3(e): Block diagram for a digital clock.

The output of the MOD-6 counter in the SECONDS section has a frequency of 1 pulse per minute. This signal is fed to the MINUTES section, which counts and displays minutes from 0 through 59. The MINUTES section is identical to the SECONDS section and operates in exactly the same manner.
The output of the MOD-6 counter in the MINUTES section has a frequency of 1 pulse per hour. This signal is fed to the HOURS section, which counts and displays hours from 1 through 12. The HOURS section is different from the MINUTES and SECONDS section in that it never goes to the zero state. The circuitry in this section is different. When the hours counter reaches 12, it will be reset to zero by the NAND gate.

3.6 Advantage of Synchronous Counters over Asynchronous

In a parallel counter all the Flip-flops will change states simultaneously; that is, they are all synched to the NGTs (see Module 2, Unit 3) of the input clock pulses. Thus, unlike the asynchronous counters, the propagation delays of the flip-flops do not add together to produce the overall delay. Instead, the total response time of a synchronous counter is the time it takes one flip-flop to toggle plus the time for the new logic levels to propagate through a single AND gate to reach the J, K inputs. That is,

$$\text{Total delay} = \text{Flip-Flop } t_{pd} + \text{ AND gate } t_{pd}$$

This means that a synchronous counter can operate at a much higher input frequency than an asynchronous counter with the same number of flip-flops. Of course, the synchronous counter requires more circuitry than the asynchronous counter. This ability to operate at higher frequencies is the major advantage of synchronous counters.

Self Assessment Exercise

1. What is the advantage of a synchronous counter over an asynchronous counter? What is the disadvantage?

4.0 Conclusion

The basic principle of operation of the synchronous counter is this:
The J and K inputs of the flip-flops are connected so that only those flip-flops that are supposed to toggle on a given NGT will have J = K = 1 when that NGT occurs.

5.0 Summary

In this unit, you learnt about:

- Synchronous Counter with Ripple Carry
- Synchronous Down-Counter
- Synchronous Up-Down Counter
- Application of Counters
- The advantages of synchronous counter over asynchronous counter

6.0 Tutor Marked Assignment

1) Design a 4-bit synchronous counter with ripple carry
2) Design a 4-bit synchronous down-counter.
3) Explain two other applications of Counters

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/