Digital Systems Design

Ramaswamy Palaniappan

bookboon
The eBook company

Ramaswamy Palaniappan

Digital Systems Design

Digital Systems Design

ISBN 978-87-7681-806-7

Contents

Preface

The aim of this book is to provide readers with a fundamental understanding of digital system concepts such as logic gates for combinatorial logic circuit design and higher level logic elements such as counters and multiplexers.

First year undergraduates taking a course in computer science or engineering (and related disciplines like information technology) are the main target audience. Foundation year students and those taking pre-university courses (like '*A' levels)* will also benefit from the text.

I have tried to follow a simple approach in writing the text. Mathematics is used only where necessary. There are plenty of illustrations to aid the reader in understanding the concepts.

I hope I have done justice in discussing all the necessary fundamentals related to digital systems in this one volume. But by doing so, I had to skip advanced concepts such as computer hardware and programming and the interested reader can refer to advanced texts after mastering the basic concepts presented in this book.

For over a decade, I have greatly benefited from discussions with students and fellow colleagues who are too many to name here but have all helped in one way or another towards the contents of this book and I must thank them. I must also thank my wife for helping me prepare some of the contents. Many a time, she and my daughter had to put up with my absence to complete this book, so I dedicate this work to them. I am also indebted to Dr. Cota Navin Gupta for his useful comments in the early parts of the book. Finally, I trust that my proofreading is not perfect and some errors would remain in the text and I welcome any feedback or questions from the reader.

Ramaswamy Palaniappan
July 2014

About the author

Dr. Ramaswamy Palaniappan
BE, MEngSc, PhD, SMIEEE, MIET, MBMES
School of Computer Science and Electronic Engineering
University of Essex, United Kingdom

Ramaswamy Palaniappan (better known as *Palani* among friends), received his first degree and MEngSc degree in electrical engineering and PhD degree in microelectronics/biomedical engineering in 1997, 1999 and 2002, respectively from University of Malaya, Kuala Lumpur, Malaysia. He is currently an academic with the School of Computer Science and Electronic Engineering, University of Essex, United Kingdom. Prior to this, he was the Associate Dean and Senior Lecturer at Multimedia University, Malaysia and Research Fellow in the Biomedical Engineering Research Centre-University of Washington Alliance, Nanyang Technological University, Singapore.

He has been teaching in a number of universities worldwide for the past 15 years in both computer science and engineering fields and has received numerous awards for excellence in teaching. He is an expert reviewer for many funding bodies such as Austria, Canada, EU, Russia and Malaysia. He founded and chaired the Bioinformatics division at the Centre for Bioinformatics and Biometrics in Multimedia University, Malaysia. His current research interests include biological signal processing, brain-computer interfaces, biometrics, artificial neural networks, genetic algorithms, and image processing. To date, he has published over 100 papers in peer-reviewed journals, book chapters, and conference proceedings.

Dr. Palaniappan is a senior member of the Institute of Electrical and Electronics Engineers and IEEE Engineering in Medicine and Biology Society, member in Institution of Engineering and Technology, and Biomedical Engineering Society. He also serves as editorial board member for several international journals. His pioneering studies on using brain signals for brain-computer interfaces and biometrics have received international recognition.

Ramaswamy Palaniappan
July 2014

1 Number System Basics

Digital technology has become widespread and encompasses virtually all aspects of our everyday lives. We could see it being used in computers and related gadgets, entertainment, automation (robotics), medical etc. Though physical quantities measured in the real world are analogue, most of these are processed by digital means. In order to do this, we have to convert the measured analogue quantity into digital, process the digital quantity using digital circuitry and then reconvert to analogue.

The contents of this book concentrate on the digital circuit design to enable the processing of the digital quantity. But before we look into the principles of such designs, we need to understand the basics of number systems.

1.1 Decimal Numbers

Decimal number system is the commonly used number system that has ten digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. It is also known as base (or radix) ten system since it has ten digits that can be used to represent any number. Figure 1.1 shows the positional values or weights of the decimal number system for an integer.

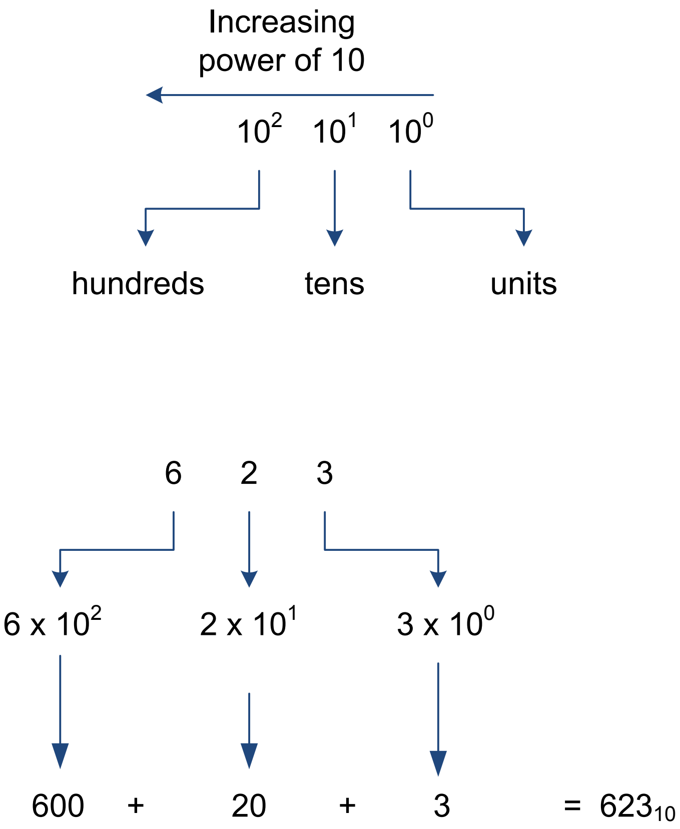


Figure 1.1: Decimal number system for integers.

The digit with least weight (i.e. the one on the foremost right) is known as the least significant digit (LSD) while the highest weight digit is known as the most significant digit (MSD). In the example shown in Figure 1.1, the MSD is digit 6 while the LSD is digit 3. Figure 1.2 shows the case for fractional decimal number.

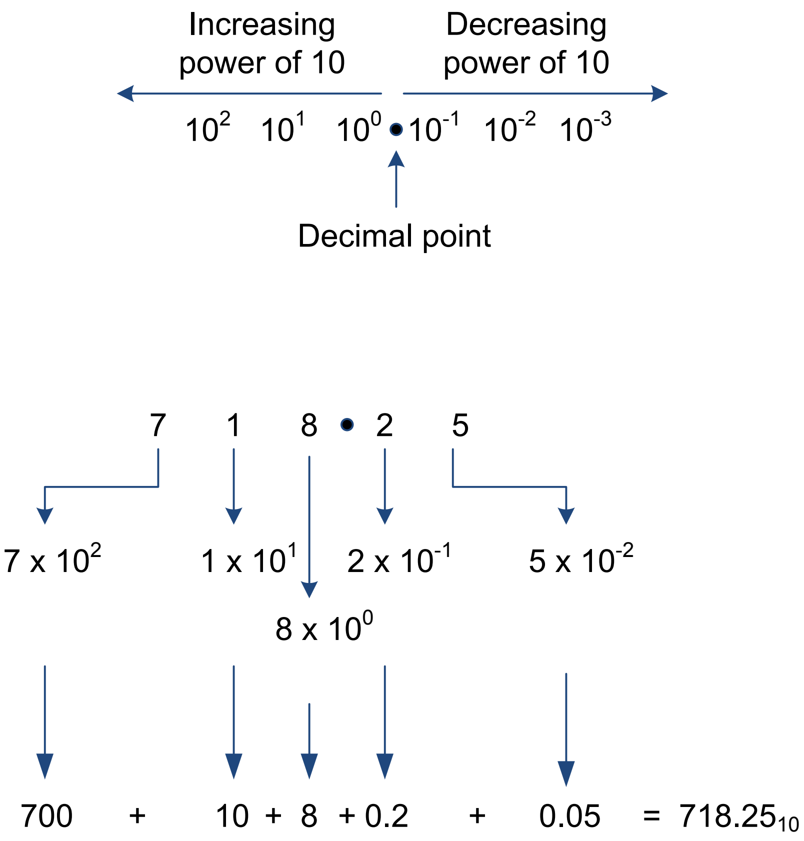
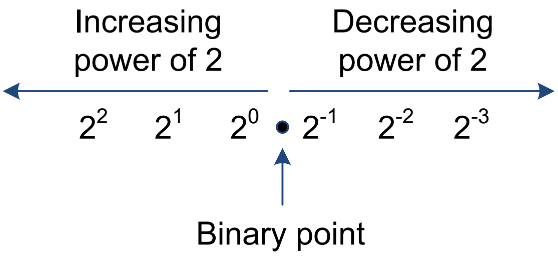


Figure 1.2: Decimal number system for fractional numbers.

1.2 Other Number Systems – Binary, Octal and Hexadecimal

While decimal number system is the commonly used number system in everyday lives, digital devices uses only binary number system that consists of 0 and 1. The base is two for this system and Figure 1.3 show an example of binary number for decimal equivalent of 6.25_{10}



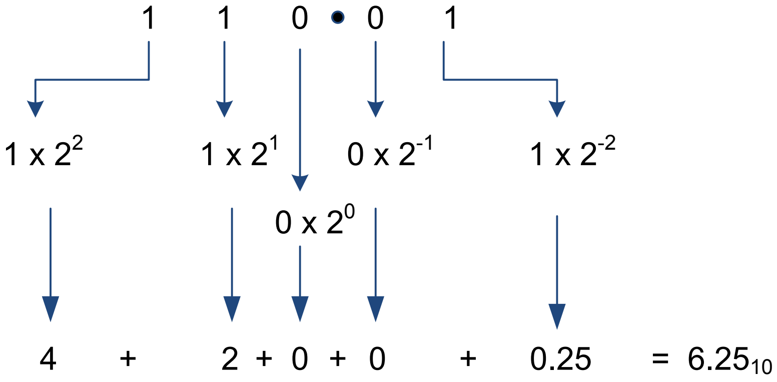
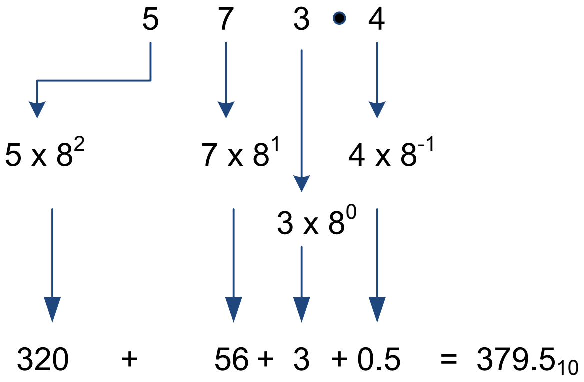
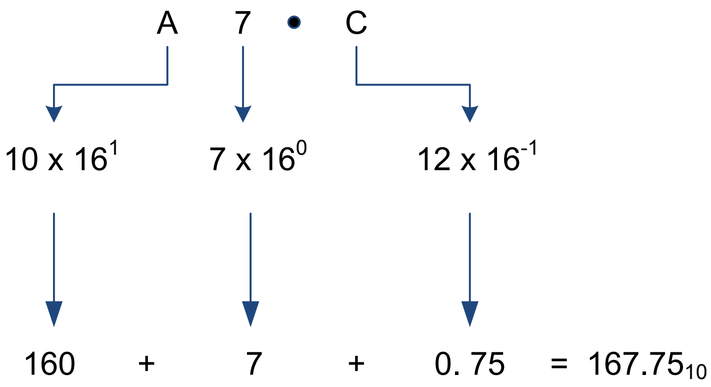


Figure 1.3: Binary number system with an example.

Similarly, octal and hexadecimal (hex in short) number systems have number bases of 8 and 16. For octal number system, the eight digits are 0, 1, 2, 3, 4, 5, 6, and 7 while hexadecimal number system has 16 digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. Figure 1.4 gives examples on these number systems.



(a)



(b)

Figure 1.4: Number system examples (a) octal (b) hex.

1.3 Conversion between different number systems

It is often necessary to convert a number from one base system to another. Converting a number to decimal is rather straightforward as we have seen in the previous examples. The weights or positional values (for the appropriate base) are multiplied with the digit and summed to give the decimal value. In this section, we will look at methods to convert numbers from decimal to binary, octal and hex. Other conversions such as octal to binary (and vice versa), binary to hex, hex to binary, octal to hex and hex to octal are also possible.

1.3.1 Decimal to binary, octal and hex conversions

There are two methods that can be used to achieve decimal to binary conversion. The first method is by presenting the decimal value in units, tens, hundreds etc. For example:

$$26 = 16 + 8 + 2 = 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 11010_{10}$$

The problem with this method is that certain positional values (such as 2^2 and 2^0 in the example above) can easily be forgotten. There is another method called repeated division that is more frequently employed. Figure 1.5 illustrates this method. It works by repeated division with a value of 2 (until the quotient is 0) and the remainder digits from each step represent the binary number (in reverse order).

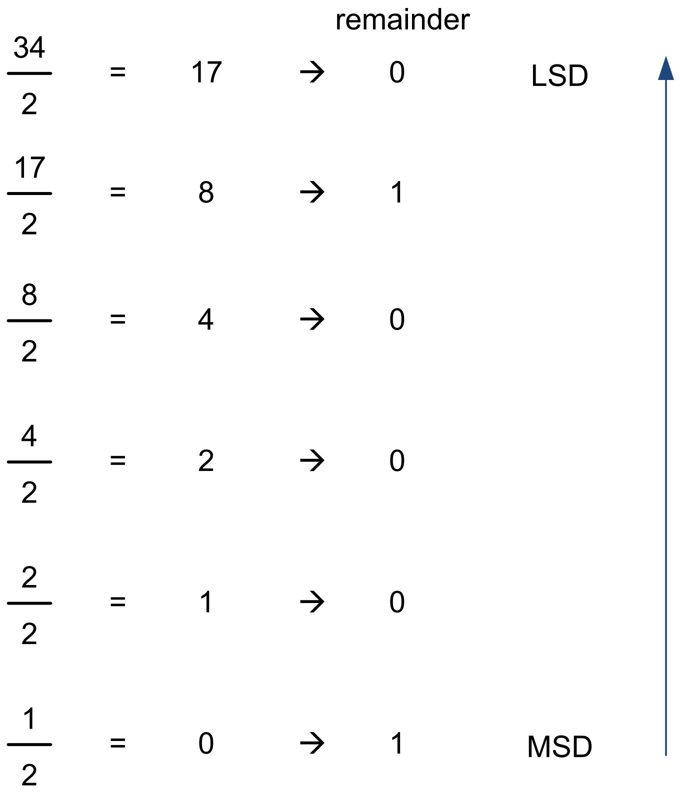


Figure 1.5: Decimal to binary conversion example, $34_{10} = 100010_2$.

Similarly, we can convert a decimal number to octal and hex. Figures 1.6 and 1.7 illustrate the steps for these conversions. Do remember that the final answer is in the reverse order!

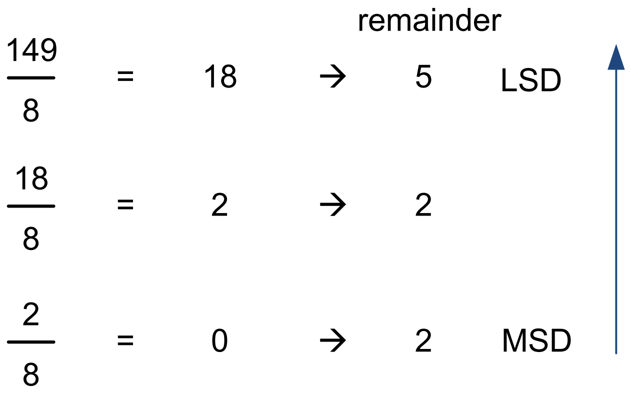


Figure 1.6: Decimal to octal conversion example, $149_{10} = 225_8$.

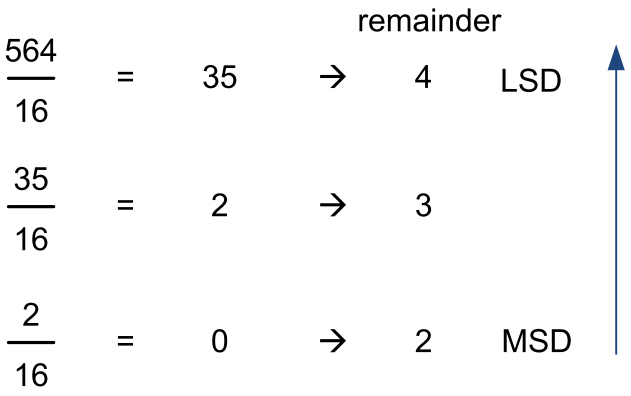


Figure 1.7: Decimal to hex conversion example, $564_{10} = 234_{16}$.

1.3.2 Binary to Octal and vice versa

Any binary number can be converted to octal simply by grouping them in groups of three digits. For example, 100101110_8 can be converted to 456_8 as shown in Figure 1.8 (a). The reverse procedure of converting an octal number to binary can be done by writing three binary digit equivalent for each octal digit. This is shown in Figure 1.8 (b).

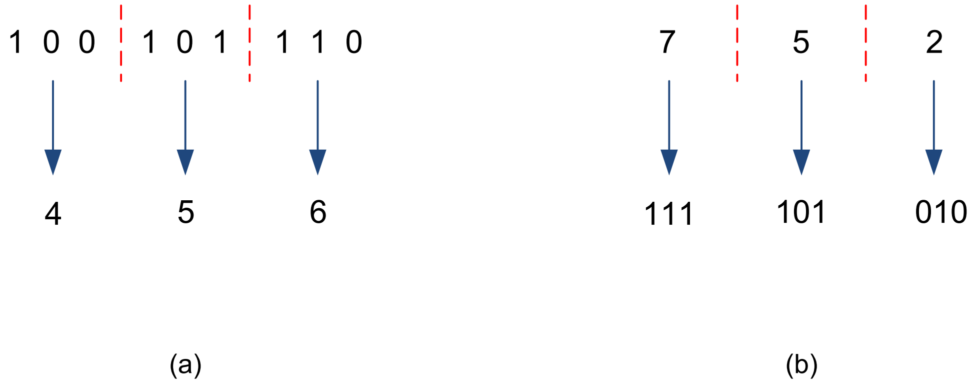


Figure 1.8: Octal to binary conversion example and vice versa: (a) $100101110_2 = 456_8$ (b) $752_8 = 111101010_2$.

1.3.3 Binary to Hex and vice versa

Similar to octal number, binary number can be converted to hex simply by grouping them in groups of four digits. For example, 10010111_2 can be converted to 97_{16} as shown in Figure 1.9 (a). A hex number can be converted to binary by writing four binary digit equivalent for each hex digit. This is shown in Figure 1.9 (b).

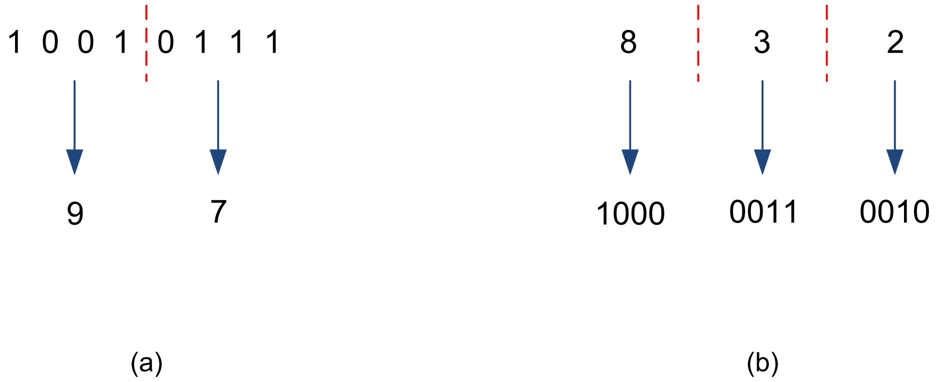


Figure 1.9: Hex to binary conversion example and vice versa: (a) $10010111_2 = 97_{16}$ (b) $832_{16} = 100000110010_2$.

1.4 Other number codes

In this section, several other commonly used codes will be discussed.

1.4.1 ASCII code

ASCII stands for American Standard Code for Information Interchange. Characters such as 'a', 'A', '@', '$' each have a code that is recognised by the computer. Standard ASCII has 128 characters (represented by 7 binary digits; 2^7=128), though the first 32 is no longer used. Extended ASCII has another 128 characters, mostly to represent special characters and mathematical symbols such as 'ÿ', 'ë', 'Σ', and 'σ'. Table 1.1 shows the standard ASCII code.

Table 1.1: Standard ASCII code

D’mal	Hex	B’ary	Char	D’mal	Hex	B’ary	Char	D’mal	Hex	B’ary	Char
32	20	0100000	space	48	30	0110000	0	64	40	1000000	@
33	21	0100001	!	49	31	0110001	1	65	41	1000001	A
34	22	0100010	“	50	32	0110010	2	66	42	1000010	B
35	23	0100011	#	51	33	0110011	3	67	43	1000011	C
36	24	0100100	$	52	34	0110100	4	68	44	1000100	D
37	25	0100101	%	53	35	0110101	5	69	45	1000101	E
38	26	0100110	&	54	36	0110110	6	70	46	1000110	F
39	27	0100111	‘	55	37	0110111	7	71	47	1000111	G
40	28	0101000	(	56	38	0111000	8	72	48	1001000	H
41	29	0101001	)	57	39	0111001	9	73	49	1001001	I
42	2A	0101010	*	58	3A	0111010	:	74	4A	1001010	J
43	2B	0101011	+	59	3B	0111011	;	75	4B	1001011	K
44	2C	0101100	,	60	3C	0111100	<	76	4C	1001100	L
45	2D	0101101	-	61	3D	0111101	=	77	4D	1001101	M
46	2E	0101110	.	62	3E	0111110	>	78	4E	1001110	N
47	2F	0101111	/	63	3F	0111111	?	79	4F	1001111	O
80	50	1010000	P	96	60	1100000	`	112	70	1110000	p
81	51	1010001	Q	97	61	1100001	a	113	71	1110001	q
82	52	1010010	R	98	62	1100010	b	114	72	1110010	r
83	53	1010011	S	99	63	1100011	c	115	73	1110011	s
84	54	1010100	T	100	64	1100100	d	116	74	1110100	t
85	55	1010101	U	101	65	1100101	e	117	75	1110101	u
86	56	1010110	V	102	66	1100110	f	118	76	1110110	v
87	57	1010111	W	103	67	1100111	g	119	77	1110111	w
88	58	1011000	X	104	68	1101000	h	120	78	1111000	x
89	59	1011001	Y	105	69	1101001	i	121	79	1111001	y
90	5A	1011010	Z	106	6A	1101010	j	122	7A	1111010	z
91	5B	1011011	[	107	6B	1101011	k	123	7B	1111011	{
92	5C	1011100	\	108	6C	1101100	l	124	7C	1111100	|
93	5D	1011101	]	109	6D	1101101	m	125	7D	1111101	}
94	5E	1011110	^	110	6E	1101110	n	126	7E	1111110	~
95	5F	1011111	_	111	6F	1101111	o	127	7F	1111111	.

1.4.2 Binary coded decimal (BCD)

BCD is actually a set of binary numbers where a group of four binary numbers represent a decimal digit. As there are 10 basic digits in the decimal number system, four binary digits (bits) are required[1]. Figure 1.10 shows an example, while Table 1.2 gives the BCD code.

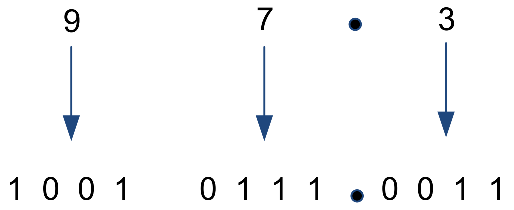


Figure 1.9: Hex to binary conversion example and vice versa: $973_{10} = 10010111.0011_{BCD.}$

[1] Three bits will only give eight representations, which is not enough for a decimal system.

Table 1.2: BCD code

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

1.4.3 Gray code

Gray code is another commonly encountered code system. The main feature of this code is that only one bit changes between two successive values. This system is less prone to errors and is considered very useful for practical applications such as mechanical switches and error correction in digital communication as compared to the standard binary system. Table 1.3 gives the BCD code with 4 bits (i.e. up to decimal value of 15).

Table 1.3: Gray code

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

2 Introduction to Logic Gates

The basic building blocks for digital circuits are logic gates. Most logic gates are binary logic, i.e. have two states of 0 or 1. The input or output of these logic gates can only exist in one of these states, where a positive logic system treats 0 as FALSE value and 1 as TRUE value and conversely for the negative logic system. Figure 2.1 shows a logic waveform that is logic 1 between time *t*1 and *t*2 and is logic 0 at other times. Positive logic will be assumed throughout the book except where denoted otherwise.

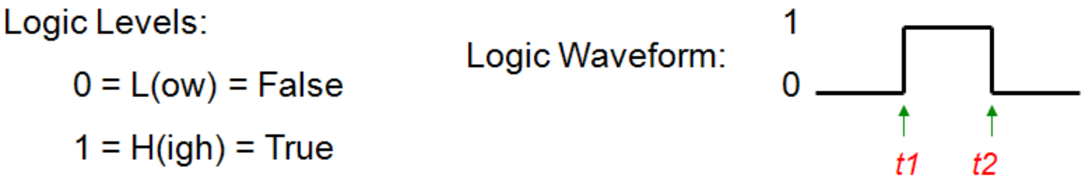


Figure 2.1: Positive logic waveform.

Figures 2.2 and 2.3 show the input and output voltage ranges for logic 0 and 1 for a common logic gate[2] used in digital devices. Transition region is the range where the voltage is not defined and hence, the input or output voltage from the device should not fall in this region as the logic value can be either 0 or 1. The output ranges are smaller as compared to input ranges, which is useful to reduce noise interference. The difference between the input-output ranges is known as noise margin. While it is usual to have a noise margin that is the same for both logical values, this does not have to be the case all the time.

To illustrate the usefulness of this noise margin, consider an example where there is noise interference in between two devices. Suppose the output voltage from the first digital device is 4.6 V (i.e. digital logic 1) and a spike (noise) of -0.5V enters as interference. The value of input voltage to the second device will be 4.1 V and the input digital level will still be 1. Without this noise margin, the digital level input to the second device will be unpredictable as it will fall within the transition region. The difference between input and output ranges for a given logic value is known as guaranteed noise immunity, which is 1 V in this case. It should also be obvious that the transition region for output voltage will be wider than for the input voltage because of this noise margin.

[2] The gate is actually a CMOS type NAND gate. NAND gates will be discussed later in the chapter.

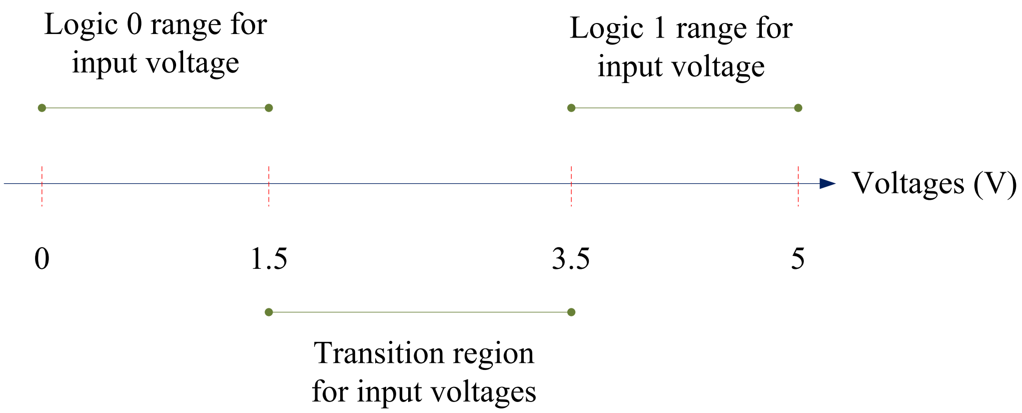

Logic 0 range for input voltage
Logic 1 range for input voltage
Voltages (V)
0
1.5
3.5
5
Transition region for input voltages

Figure 2.2: Input logic related to actual voltages.

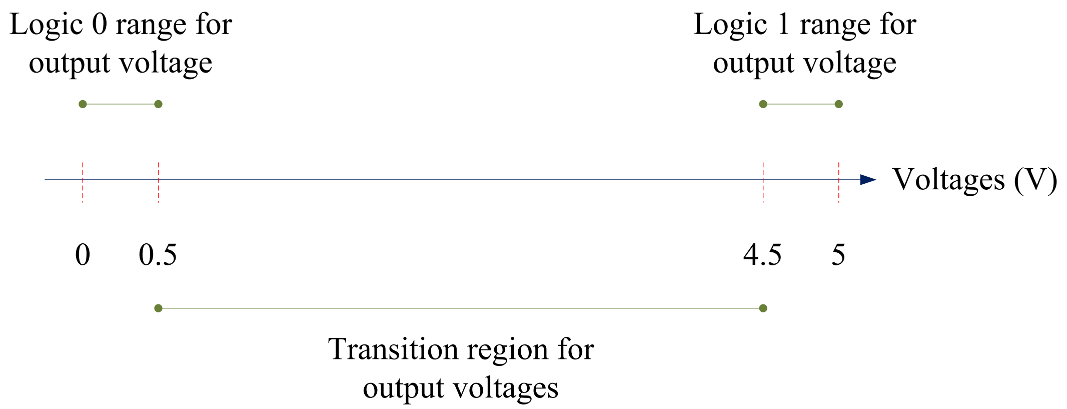

Logic 0 range for output voltage
Logic 1 range for output voltage
Voltages (V)
0
0.5
4.5
5
Transition region for output voltages

Figure 2.3: Output logic related to actual voltages.

Actual pulse waveform does not resemble the form shown in Figure 2.1, but is rather like the one shown in Figure 2.4[3] where there is a period of time required for the pulse to rise and fall and these are known as rise and fall times, respectively. The time taken for the pulse to rise from 10% to 90% of the amplitude is rise time while the fall time is the time taken for the amplitude value to drop to 10% from 90%. The actual rise and fall times for a digital device depends on its specifications; costly devices have smaller times. The pulse width is measured using 50% of the rise and fall amplitude values as shown in the figure.

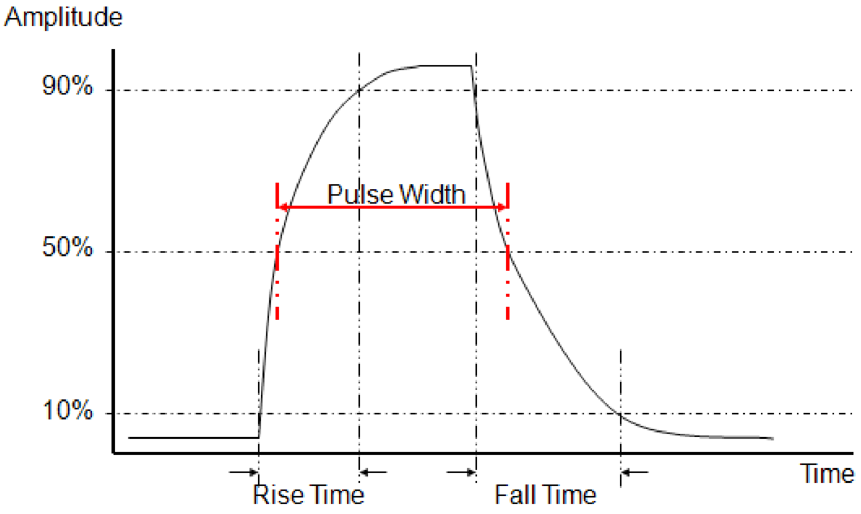


Figure 2.4: An example of actual pulse waveform.

[3] Even this figure is simplified for ease of understanding. Actual waveform will have lots of spikes.

2.1 AND gate

Basically AND gate is composed of two inputs and a single output as shown in Figure 2.5 with algebraic representation[4] $F = A \cdot B$ or simply . $F = AB$ The traditional symbol shown in Figure 2.5(a) is more commonly employed in text books. However, the IEEE/ANSI symbol as shown in Figure 2.5(b) is gaining popularity and has the advantage of containing qualifying symbols inside the logic-symbol that describes the operation of the gate. The truth table that gives the output *F* for inputs *A* and *B* is given in Table 2.1. It can be seen that the output is LOW (FALSE) when any one of the inputs is LOW (FALSE) and the output is only HIGH (TRUE) when all the inputs are HIGH (TRUE).

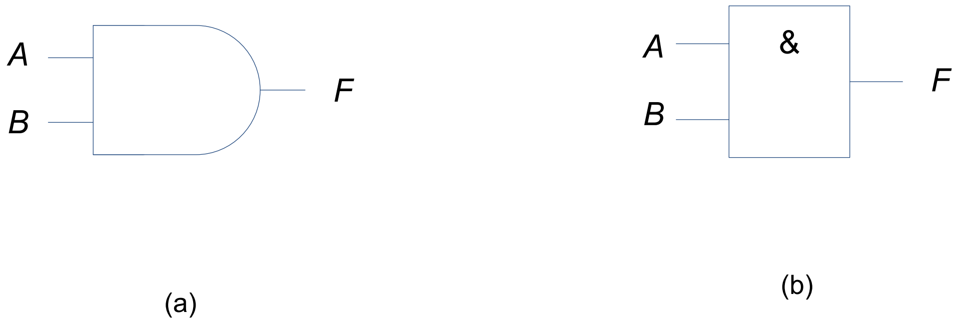


Figure 2.5: AND gate logic symbols (a) traditional (b) IEEE/ANSI standard.

Table 2.1: Truth table for two-input AND gate

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

AND gate inputs do not have to be limited to two; there can be any number of inputs greater than one as shown in Figure 2.6.

[4] Also known as Boolean or logic expression.

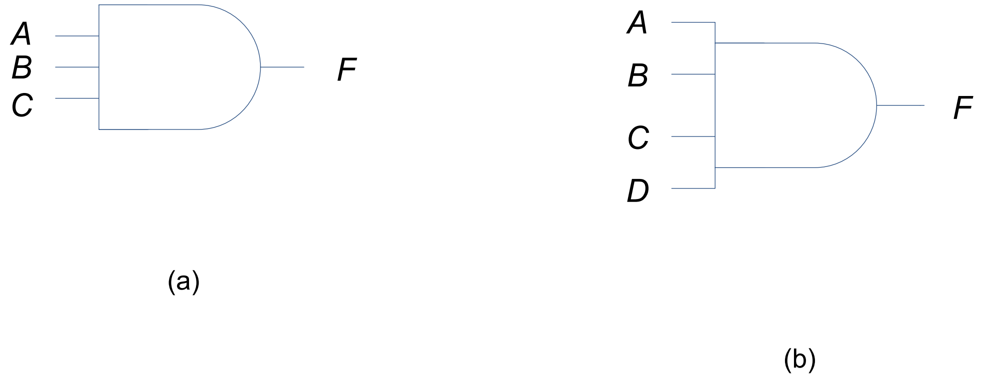
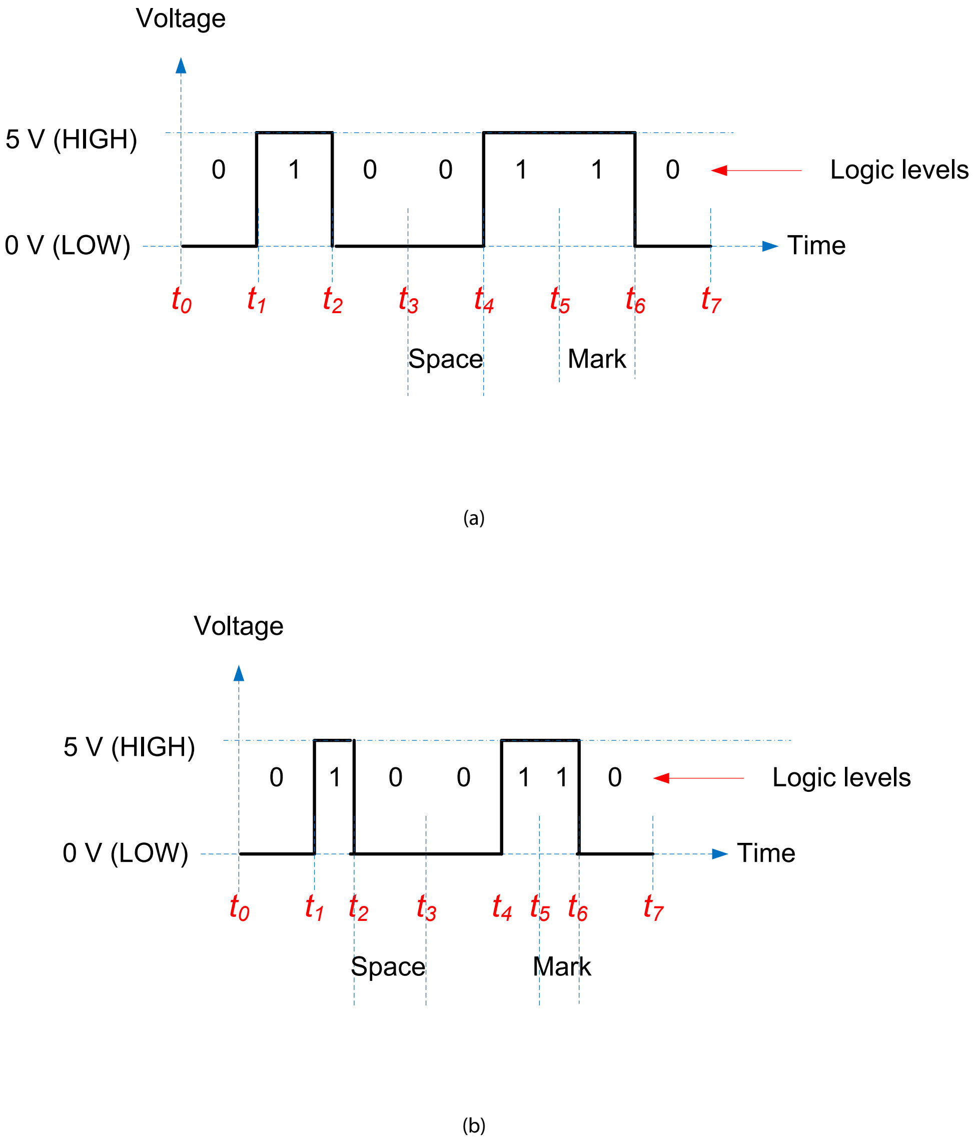


Figure 2.6: Three and four input AND gates: (a) $F = A \cdot B \cdot C$ (b) $F = A \cdot B \cdot C \cdot D$.

2.1.1 Timing diagram

Timing diagram is useful in describing the relationship between the inputs and output of a logic gate. The inputs of a digital logic gate can be shown diagrammatically as a waveform that represents the changing values over time. A waveform corresponding to the changing values of the inputs over time will be generated at the output of the logic gate. Figure 2.7 show examples of timing diagram waveform for equal and unequal mark-space cycles. The mark represents the time for logic level HIGH, while the space represents the time for logic level LOW. Equal mark-space requires periodic clock pulse[5]. All the discussion in this book will be using equal mark-space timing waveforms only.

[5] Clock pulses will be discussed in later chapters.



(a)

(b)

Figure 2.7: Example of timing diagram waveforms: (a) equal mark-space (b) unequal mark-space.

2.1.2 Timing diagram example for AND gate

Figure 2.8 shows an example of a timing diagram for a two-input AND gate. At each time block, the inputs *A* and *B* affect the output *F*. For example, in time block t_0 to t_1, both inputs are LOW, so the output is also LOW. Similarly, the entire timing waveform for the output can be obtained using AND operation of inputs in each time block.

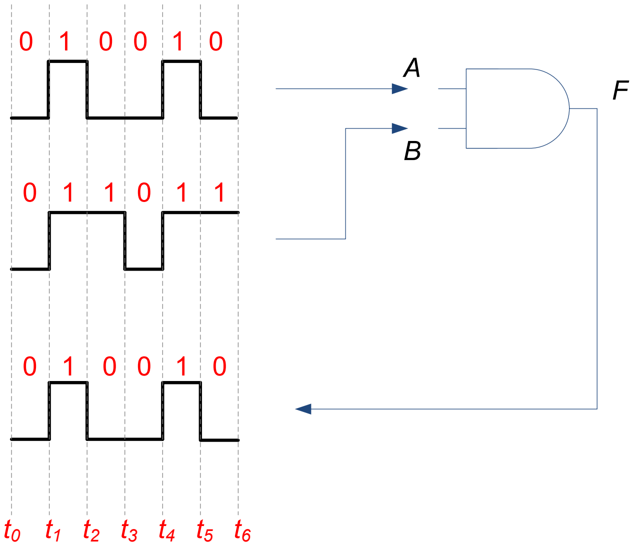


Figure 2.8: Timing diagram waveform for a two-input AND gate.

2.2 OR gate

OR gate as shown in Figure 2.9 has algebraic representation, $F = A + B$. The truth table that gives the output F for inputs A and B is given in Table 2.2. It can be seen that the output is HIGH when any one of the inputs is HIGH and the output is only LOW when all the inputs are LOW.

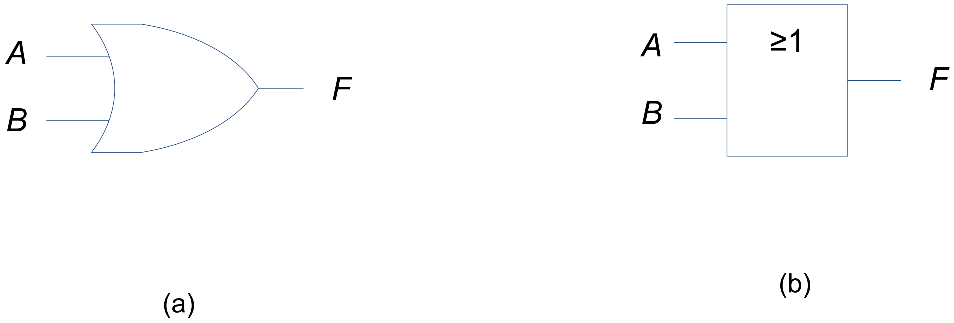


Figure 2.9: OR gate logic symbols: (a) traditional (b) IEEE/ANSI standard.

Table 2.2: Truth table for two-input OR gate

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

Similar to AND gate, there can be any number of inputs greater than one as shown in Figure 2.10.

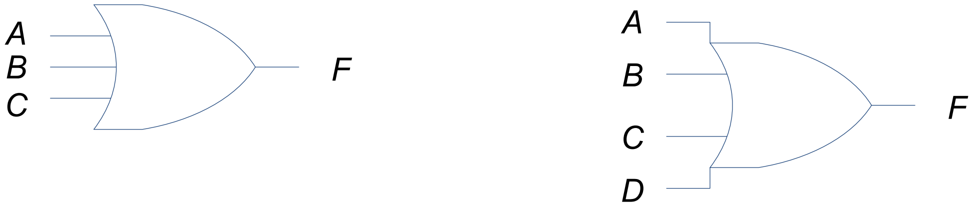


Figure 2.10: Three and four input OR gates: (a) $Y = A + B + C$ (b) $Y = A + B + C + D$.

2.2.1 Timing diagram example for OR gate

Figure 2.11 shows an example of a timing diagram for a two-input OR gate. At each time block, the inputs *A* and *B* affect the output *F*. For example, in time block t_5 to t_6, one input is HIGH, so the output is HIGH. Similarly, the entire timing waveform for the output can be obtained using OR operation of inputs in each time block.

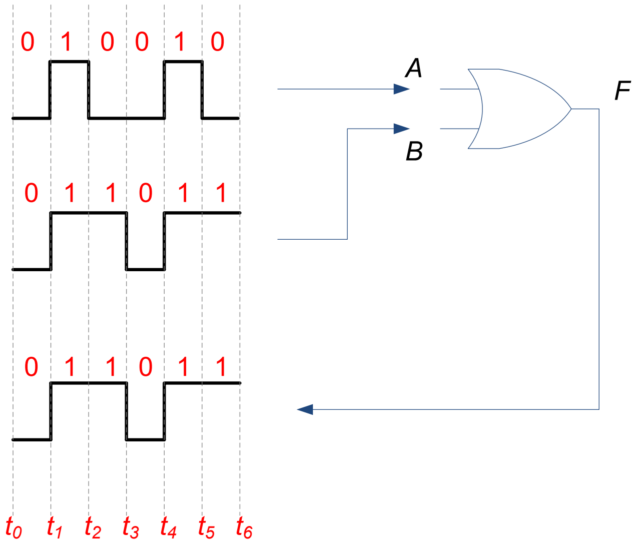
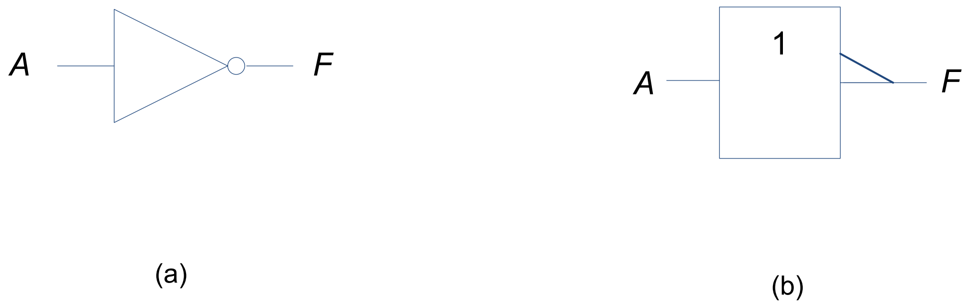


Figure 2.11: Timing diagram waveform for a two-input OR gate.

2.3 NOT gate

NOT gate is also known as INVERTER as it inverts (complements) the input logic level. It is shown in Figure 2.12 and has only one input and one output with algebraic representation of $F = \overline{A}$ or $F = A'$. The bubble in the symbol denotes inversion (without it, the symbol will represent a buffer gate that does not alter the logic level; in IEEE/ANSI standard, the bubble is replaced by a triangle). The truth table for NOT gate is given in Table 2.3.



(a)

(b)

Figure 2.12: NOT gate logic symbols: (a) traditional (b) IEEE/ANSI standard.

Table 2.3: Truth table for NOT gate

A	*F*
0	1
1	0

NOT gate can also be connected in cascade and a few examples are shown in Figure 2.13. It should be obvious that odd number of NOT gate connections give output logic level that is complement to the input logic level and an even number of NOT gates connections give output logic level that is the same as the input logic level.

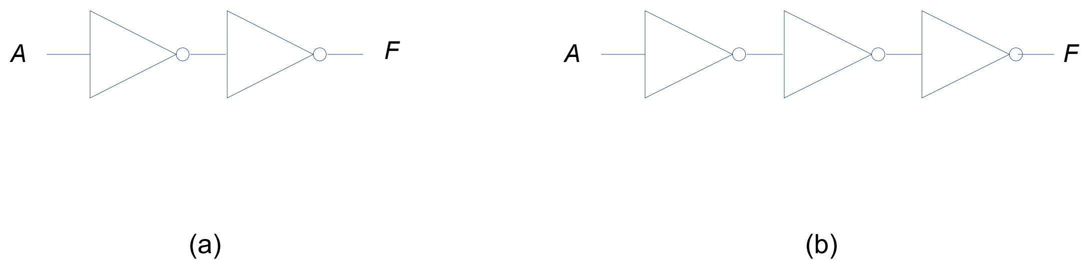


Figure 2.13: Cascade connection of NOT gates: (a) $F = \overline{\overline{A}} = A$ (b) $F = \overline{\overline{\overline{A}}} = \overline{A}$.

2.4 AND implementation with OR gate and vice versa

It is useful to know that AND gate logic can be easily implemented using OR gate and vice versa through a simple process using additional NOT gates. For example, an AND gate equivalent can be constructed with an OR gate with both the inputs and outputs inverted through NOT gates. Figure 2.14 shows an example with equivalent truth table in Table 2.4. This is actually DeMorgan's first theorem, which will be discussed in detail in Chapter Three. It is mentioned here so that the reader is aware that it is possible to implement one gate logic with another gate(s).

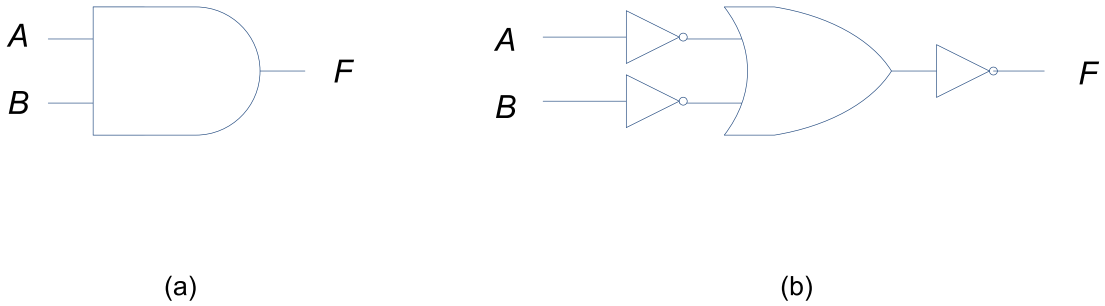


Figure 2.14: AND gate implementation with OR gate: (a) $F = AB$ (b) .$F = \overline{\overline{A} + \overline{B}} = AB$.

Table 2.4: Truth table illustrating AND gate implementation using OR and NOT gates

A	B	$F = AB$	$\overline{A}$	$\overline{B}$	$F = \overline{A} + \overline{B}$	$F = \overline{\overline{A} + \overline{B}}$
0	0	0	1	1	1	0
0	1	0	1	0	1	0
1	0	0	0	1	1	0
1	1	1	0	0	0	1

2.5 NAND gate

NAND and NOR gates that will be discussed in the following section are known as universal gates as combinations of these gates are sufficient to obtain equivalent operation of OR, AND or NOT gates. However, this is different to the implementation discussed in Section 2.4 as either NAND or NOR gates on their own will be sufficient to implement logic function of any of the other gates. NAND gate logic symbol is shown in Figure 2.15 (note the addition of a bubble when compared to AND gate) and its truth table is shown in Table 2.5. A NAND gate operation can also be obtained through cascade operation of AND and NOT gates as shown in Figure 2.16. Algebraically, the operation can be defined as. $F = \overline{AB}$.

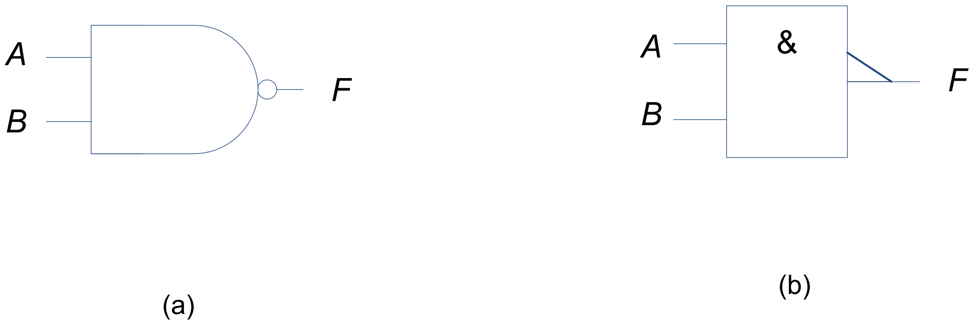


Figure 2.15: NAND gate logic symbols: (a) traditional (b) IEEE/ANSI standard.

Table 2.5: Truth table for NAND gate

A	*B*	*AB*	*F*
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

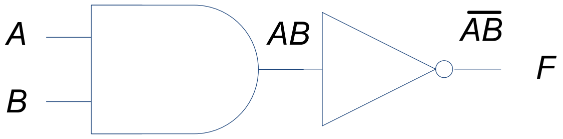


Figure 2.16: NAND gate logic using AND and NOT gates.

Figure 2.17 shows an example for implementing an AND gate using NAND gates only. The blue shaded tiny bubble represents branch-off of the signal and should not be confused with the *empty* bubble that is used to represent inversion operation. Similarly, other gates such as OR and NOT can be implemented using NAND gates and these are left as exercises for the reader.

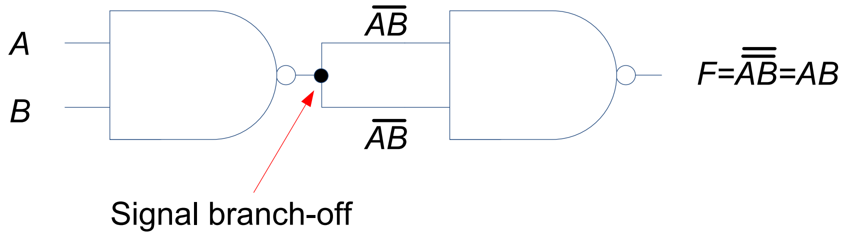


Figure 2.17: AND gate implementation using two NAND gates.

2.6 NOR gate

NOR gate is basically an OR gate with the output inverted. Figure 2.18 shows the logic symbol with truth table shown in Table 2.6. Algebraically, the operation can be defined as $F = \overline{A + B}$. Similar to NAND gate, several NOR gates can be used to implement AND, OR or NOT gates. An example of this is shown in Figure 2.19 and the reader can easily verify through the use of truth tables that $F = AB$.

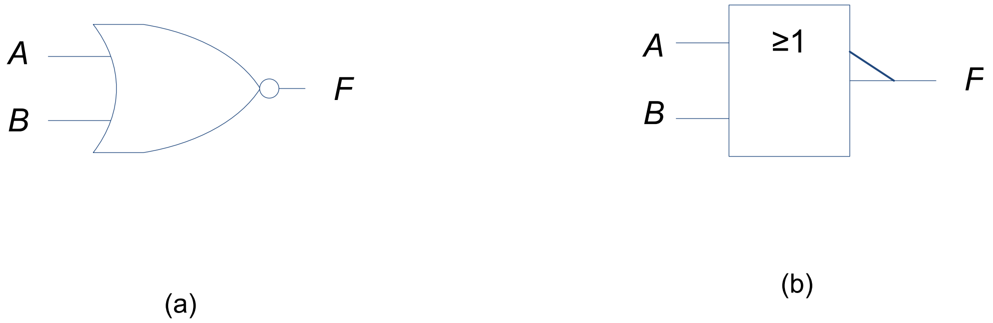


Figure 2.18: NOR gate logic symbols: (a) traditional (b) IEEE/ANSI standard.

Table 2.6: Truth table for NOR gate

A	*B*	*A+B*	*F*
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

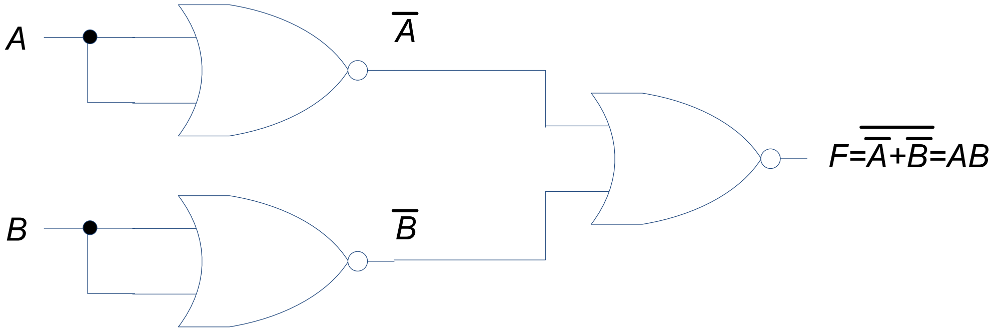


Figure 2.19: AND gate logic implementation using NOR gates.

2.7 Integrated circuits

All the gates that we have discussed in this chapter are manufactured as integrated circuit (IC) with several gates in one IC. For example, 74LS00 is a transistor-transistor logic (TTL) technology based IC that has four (quad) two-input NAND gates. Complementary Metal-Oxide Semiconductor (CMOS) is another technology that is widely used for manufacturing IC but TTL devices are more commonly employed for laboratory experiments as they are more robust to electrostatic noise. Figure 2.20 shows the pin configuration of 74LS00 and Figure 2.21 shows an example of pin configurations to implement NOT operation. Pin 14 is connected to the power supply while pin 7 is the ground pin. It should be obvious that the LED will only light-up (i.e. the output will be HIGH) if switch *A* is turned OFF (i.e. made to logic level LOW) – similar to the input and output values as in the truth table shown in Table 2.3.

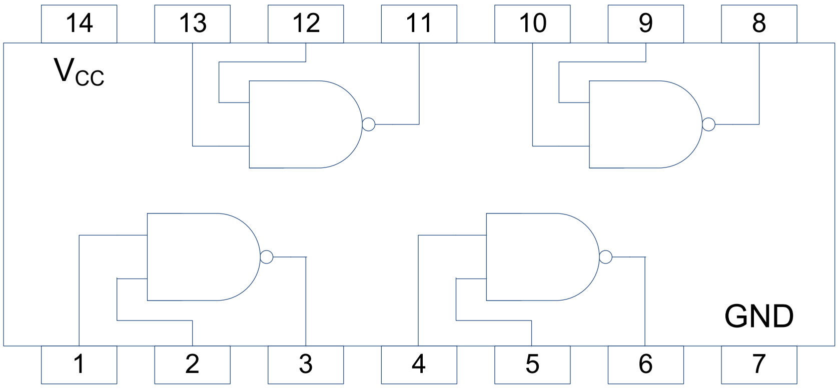


Figure 2.20: 74LS00 - Quad NAND IC.

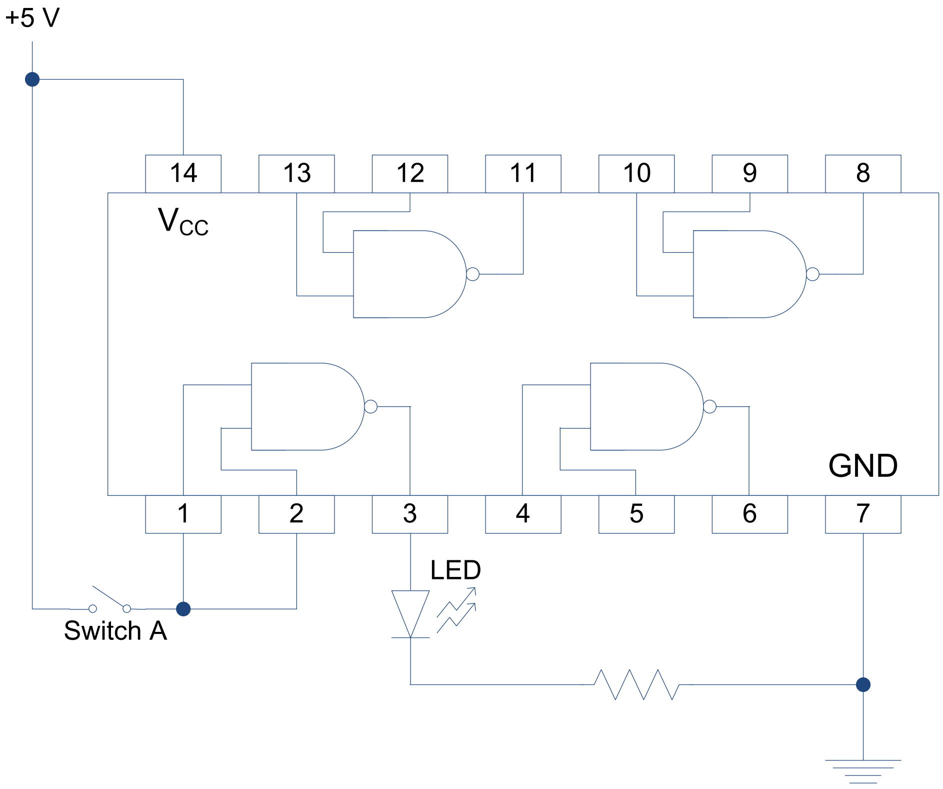


Figure 2.21: NOT gate implementation example using 74LS00.

3 Combinatorial Logic Circuits

In the previous chapter, operation and truth tables of single gates were discussed. However, in practise, single gates are seldom useful and combinations of several gates are employed for a particular application. For example, see Figure 3.1 where different gates are used to obtain the output *F*.

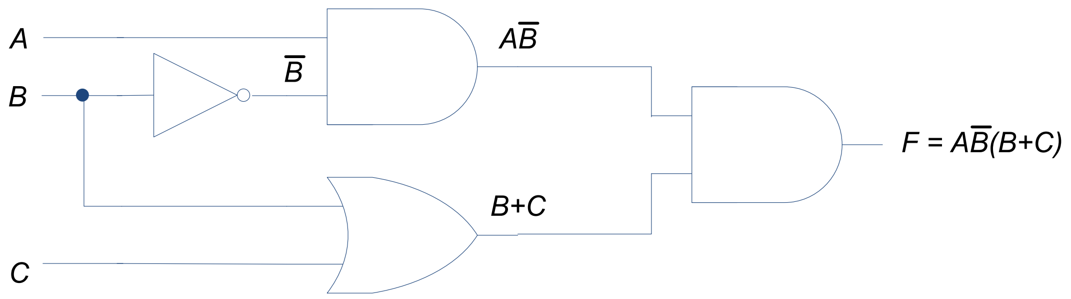


Figure 3.1: Example of combinatorial logic circuit.

3.1 Logic circuit simplification

Very often, there is the need to simplify logic circuits (whenever possible). For example, the circuit shown in Figure 3.1 requires four gates but equivalent logic output can be obtained with just two gates by simplifying the expression as follows:

$$\begin{aligned} F &= A\overline{B}(B+C) \\ &= A\overline{B}B + A\overline{B}C \qquad \text{after expanding} \\ &= A\overline{B}C\,. \end{aligned}$$

$A\overline{B}B$ is zero due to the presence of $\overline{B}B$ as shown in the truth table given in Table 3.1. The simplified circuit is given in Figure 3.2. Table 3.2 gives the truth table and it can be seen that the outputs given by expressions $F = A\overline{B}(B+C)$ and $F = A\overline{B}C$ are the same.

Table 3.1: Truth table for $A\overline{B}B$

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

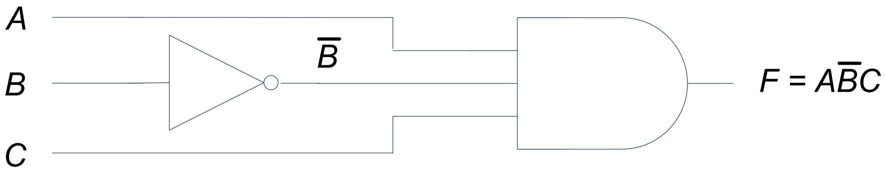


Figure 3.2: Simplified logic circuit.

Table 3.2: Truth table for $F = A\overline{B}(B + C)$ and $F = A\overline{B}C$

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

The above simplification may not be clear at this stage but that will be the purpose of the following sections to look into Boolean algebra that will be useful to simplify logic circuits. Not only will the simplification result in lower cost, smaller and simpler design (since fewer gates will be used), it will also reduce other complications such as overheating and propagation delay.

3.2 Boolean algebra

Basic axioms of Boolean algebra are shown in Table 3.3, while Table 3.4 shows the Boolean theorems for operation of a single variable and a constant (either 0 or1).

Boolean algebra satisfies commutative and associative laws. Therefore, the order of variables in a product or sum does not matter and the order of evaluating sub-expression in brackets does not matter. For example:

Commutative law: $A + B = B + A$ and $A \cdot B = B \cdot A$;

Associative law: $A + (B + C) = (A + B) + C = A + B + C$ and $A \cdot (B \cdot C) = (A \cdot B) \cdot C = A \cdot B \cdot C$.

Boolean algebra also satisfies the distributive law where the expression can be expanded by multiplying out the terms. For example:

Distributive law: $A \cdot (B + C) = A \cdot B + A \cdot C$.

It should be evident by now that when an expression contains AND and OR, AND operator takes precedence over OR operator. For example, $0 \cdot 1 + 1 \cdot 1 = 0 + 1 = 1$ and not $0 \cdot 1 + 1 \cdot 1 = 0 \cdot 1 \cdot 1 = 0$.

Table 3.3: Basic axioms of Boolean algebra

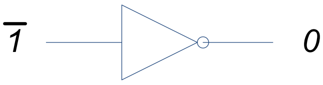
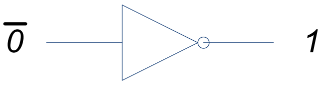
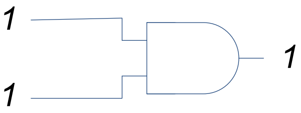
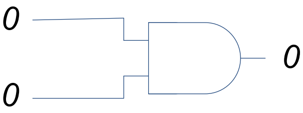
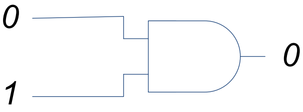
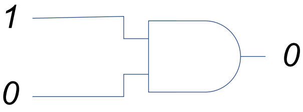
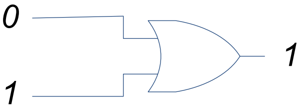
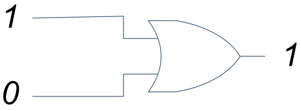
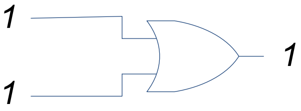
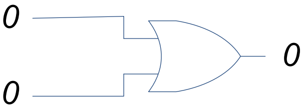
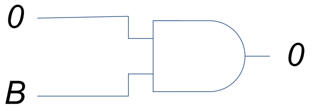
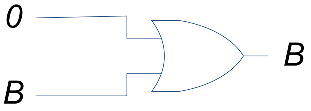
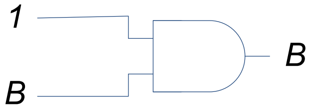
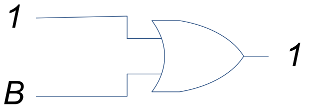
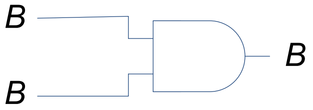
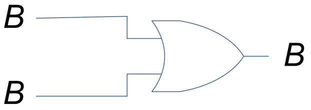
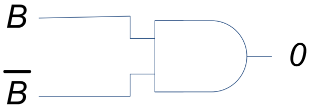
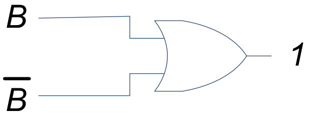
$\overline{1}=0$		$\overline{0}=1$	
$1\cdot 1=1$		$0\cdot 0=0$	
$0\cdot 1=0$		$1\cdot 0=0$	
$0+1=1$		$1+0=1$	
$1+1=1$		$0+0=0$	

Table 3.4: Boolean theorems for operation of a single variable and a constant

$0 \cdot B = 0$		$0 + B = B$	
$1 \cdot B = B$		$1 + B = 1$	
$B \cdot B = B$		$B + B = B$	
$B \cdot \overline{B} = 0$		$B + \overline{B} = 1$	

3.3 DeMorgan's theorem

DeMorgan's theorem is very useful to simplify expressions when they contain a bar (inversion) over more than a single variable. It states that an inverted expression can be replaced by its individual inverted variables but with AND replaced by OR and vice versa. For example:

DeMorgan's theorem: $\overline{A \cdot B} = \overline{A} + \overline{B}$ and $\overline{A + B} = \overline{A} \cdot \overline{B}$

Figure 3.3 shows the circuit equivalence using DeMorgan's theorem.

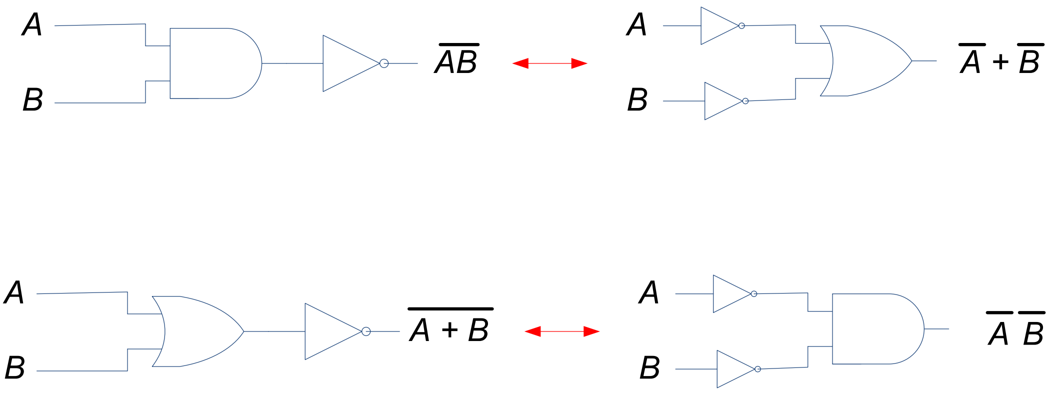


Figure 3.3: Circuit equivalence using DeMorgan's theorem.

3.3.1 Examples illustrating DeMorgan's theorem

The following examples show the usefulness of using DeMorgan's theorem. Note that from now on, the use of AND ($\cdot$) sign in the expression will be dropped for simplicity sake unless noted otherwise, so $F = A \cdot B \cdot C$ will be written as $F = ABC$..

$$\begin{aligned} F &= \overline{A+B+\overline{C}} && \text{apply DeMorgan's theorem} \\ &= \overline{A}\,\overline{B}\,\overline{\overline{C}} \\ &= \overline{A}\,\overline{B}C\,. \end{aligned}$$

$$\begin{aligned} F &= \overline{ABC+D} && \text{apply DeMorgan's theorem} \\ &= \overline{ABC}\,\overline{D} && \text{apply DeMorgan's theorem again} \\ &= \overline{D}(\overline{A}+\overline{B}+\overline{C})\,. \end{aligned}$$

$$\begin{aligned} F &= A+\overline{(B+C)D} && \text{apply DeMorgan's theorem} \\ &= A+\overline{B+C}+\overline{D} && \text{apply DeMorgan's theorem again} \\ &= A+\overline{B}\,\overline{C}+\overline{D}\,. \end{aligned}$$

3.4 More examples

In this section, several examples are given to illustrate simplification using Boolean algebra and DeMorgan's theorem:

$$\begin{aligned} F &= \overline{A+B}+\overline{A}B && \text{apply DeMorgan's theorem} \\ &= \overline{A}\,\overline{B}+\overline{A}B \\ &= \overline{A}(\overline{B}+B) && \text{see Table 3.4, } (\overline{B}+B)=1 \\ &= \overline{A} \end{aligned}$$

$$\begin{aligned} F &= \overline{A+B+C}+A\overline{B}+\overline{B}C && \text{apply DeMorgan's theorem} \\ &= \overline{A}\,\overline{B}\,\overline{C}+\overline{B}(A+C) \\ &= \overline{B}(\overline{A}\,\overline{C}+A+C) \\ &= \overline{B}(\overline{A+C}+A+C) && \text{apply inverse of DeMorgan's theorem} \\ &= \overline{B}\,. && \text{see Table 3.4, since } (\overline{X}+X)=1 \text{ where } X=A+C \end{aligned}$$

$$\begin{aligned}
F &= \overline{AB} + ABC && \text{apply DeMorgan's theorem[6]} \\
&= \overline{A} + \overline{B} + ABC && \\
&= \overline{A} + \overline{B} + C(AB) && \\
&= \overline{A} + \overline{B} + C(\overline{\overline{AB}}) && \text{apply DeMorgan's theorem again} \\
&= \overline{A} + \overline{B} + C(\overline{\overline{A} + \overline{B}}) && \text{let } X = \overline{A} + \overline{B} \\
&= X + C\overline{X} && \\
&= X + C\overline{X} + XC && \text{add } XC \text{ since } X + XC = X(1 + C) = X \\
&= X + C(\overline{X} + X) && \\
&= X + C && \text{as } X + \overline{X} = 1 \\
&= \overline{A} + \overline{B} + C && \text{replace } X = \overline{A} + \overline{B}
\end{aligned}$$

[6] There is a simpler method to obtain the solution by letting X=AB in the first place but the shown procedure illustrates several useful simplifications.

As another example, consider the circuit diagram given in Figure 3.4 which can be simplified as

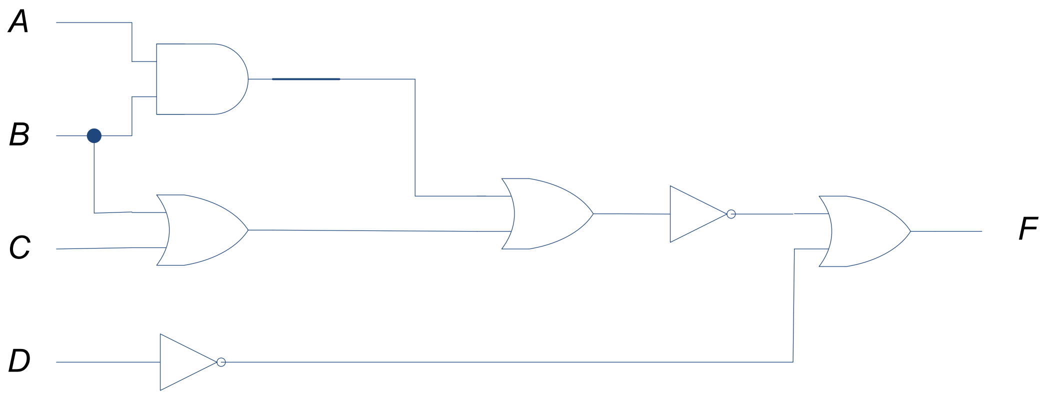


Figure 3.4: Logic circuit example for simplification.

$$\begin{array}{ll}
\overline{AB+(B+C)}+\overline{D} & \text{obtain the expression} \\
\overline{AB}\,\overline{(B+C)}+\overline{D} & \text{using DeMorgan's theorem on the top (outermost) invers} \\
(\overline{A}+\overline{B})(\overline{B}\,\overline{C})+\overline{D} & \text{using DeMorgan's theorem again} \\
\overline{A}\,\overline{B}\,\overline{C}+\overline{B}\,\overline{B}\,\overline{C}+\overline{D} & \text{after expanding} \\
\overline{A}\,\overline{B}\,\overline{C}+\overline{B}\,\overline{C}+\overline{D} & \text{since } \overline{B}\,\overline{B}=\overline{B} \\
\overline{B}\,\overline{C}(\overline{A}+1)+\overline{D} & \text{as } \overline{A}+1=1 \\
\overline{B}\,\overline{C}+\overline{D}\,. &
\end{array}$$

The correctness of the simplified expression can be verified by constructing a truth table and comparing the output from both expressions. The simplified logic circuit diagram is shown in Figure 3.5 where only five gates are required as opposed to six gates in the original circuit. It can be seen that there is no input *A* as its logic value does not affect the output based on the simplified expression.

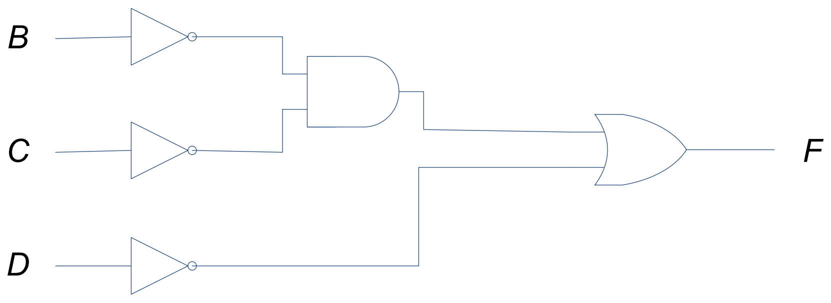


Figure 3.5: Simplified logic circuit of the example shown in Figure 3.4.

While the expression for the logic circuit shown in Figure 3.5 is simplified to single literals, it is interesting to note that another equivalent logic circuit shown in Figure 3.6 only requires four gates as $F = \overline{B}\,\overline{C} + \overline{D} = \overline{B + C} + \overline{D}$.

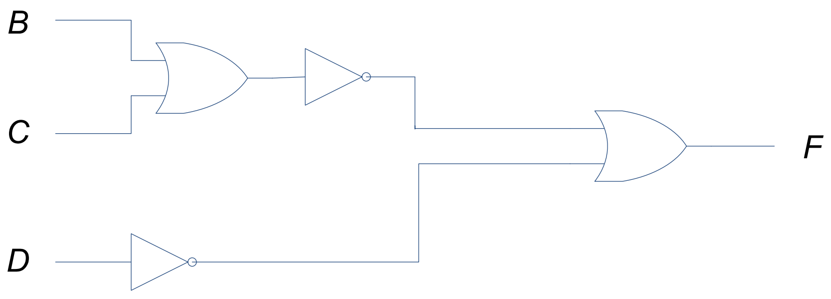


Figure 3.6: Equivalent logic circuit of the example shown in Figures 3.4 and 3.5.

If complement inputs are available, then the simplified circuit shown in Figure 3.5 will only require two gates as shown in Figure 3.7.

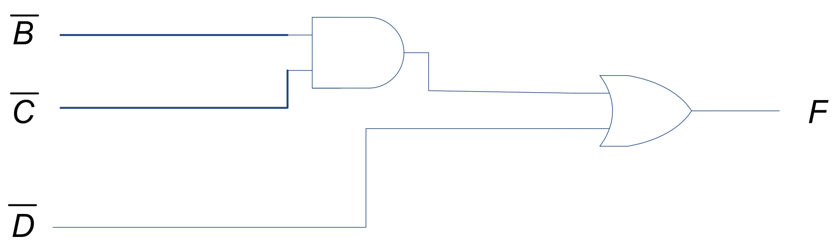


Figure 3.7: Simplified logic circuit when complement inputs are available.

3.5 XOR and XNOR gates

To conclude the chapter, it is useful to look at two more frequently used gates: Exclusive OR (XOR) and Exclusive NOR (XNOR). These gates would be useful when circuitry such as half adders and full adders are discussed in later chapters. XOR gate as shown in Figure 3.8 has algebraic representation, $F = A\overline{B} + \overline{A}B$ or more commonly written as $F = A \oplus B$.

The truth table that gives the output *F* for inputs *A* and *B* is given in Table 3.5. It can be seen that when both inputs have the same logic value, the output is LOW. The output is HIGH when the input logic values are dissimilar, i.e. one LOW and one HIGH.

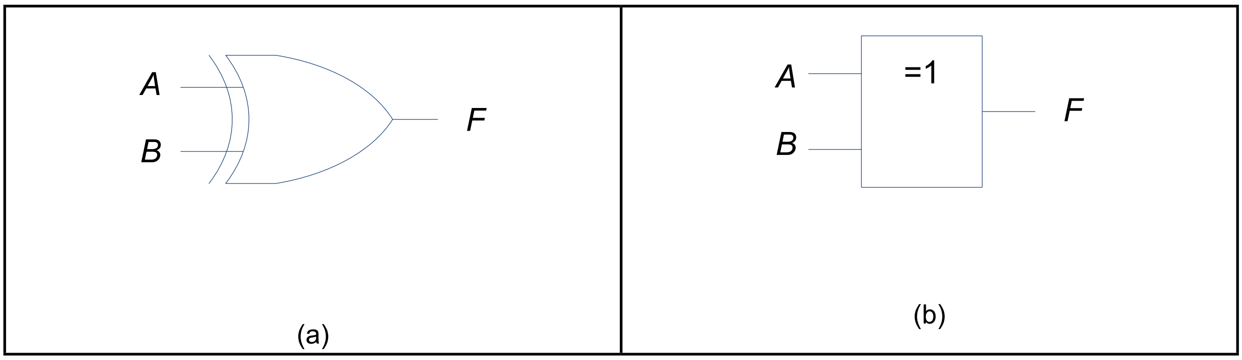


Figure 3.8: NOR gate logic symbols: (a) traditional (b) IEEE/ANSI standard.

Table 3.5: Truth table for two-input XOR gate

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

XNOR gate is simply XOR with an inversion. The gate is shown in Figure 3.9 and has algebraic representation,

$$\begin{aligned}
F &= \overline{A\overline{B} + \overline{A}B} && \\
F &= \overline{A\overline{B}}\,\overline{\overline{A}B} && \text{using DeMorgan's theorem on the top inversion} \\
F &= (\overline{A} + B)(A + \overline{B}) && \text{using DeMorgan's theorem again} \\
F &= \overline{A}A + B\overline{B} + AB + \overline{A}\overline{B} && \text{after expanding} \\
F &= AB + \overline{A}\overline{B} && \text{expression for XNOR}
\end{aligned}$$

or more commonly written as $F = \overline{A \oplus B}$.

The truth table is given in Table 3.6. The output is HIGH when both inputs have the same logic value. The output is LOW when the input logic values are dissimilar, i.e. one LOW and one HIGH.

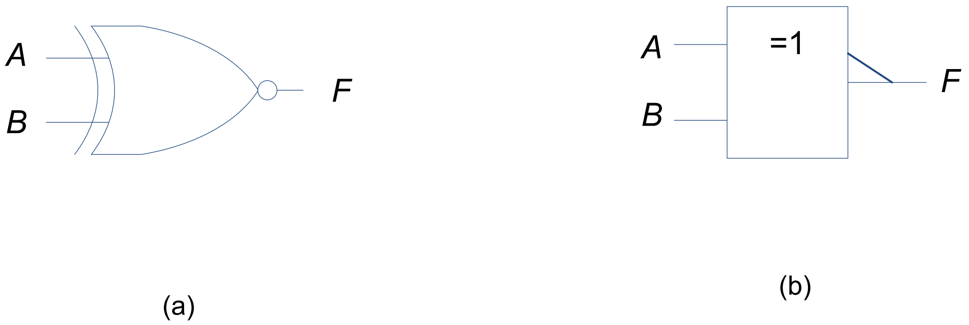


Figure 3.9: XNOR gate logic symbols: (a) traditional (b) IEEE/ANSI standard.

Table 3.6: Truth table for two-input XNOR gate

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

3.5.1 Boolean algebra for XOR operation

Table 3.7 shows the Boolean algebra for XOR operation. XOR operation is also both commutative and associative: $A \oplus B = B \oplus A$ and $A \oplus (B \oplus C) = (A \oplus B) \oplus C = A \oplus B \oplus C$.

Table 3.7: Boolean algebra for XOR operation

$A \oplus 0 = A$	$A \oplus A = 0$	$A \oplus \overline{B} = \overline{A \oplus B}$
$A \oplus 1 = \overline{A}$	$A \oplus \overline{A} = 1$	$\overline{A} \oplus B = \overline{A \oplus B}$

3.5.2 Parity checker

As mentioned earlier, XOR gates are useful when designing more advanced circuitry such as adders, but these are also used in parity checker devices. Parity checker is used to reduce errors from transmitting a binary code across a communication channel. For example, if the seven bit ASCII code for W, 1010111 (see Table 1.1) is to be transmitted, an eight parity bit is appended at the beginning of the code. This parity bit will either be 0 or 1 depending on whether even or odd parity is required. Assuming that it is even parity checker, then the total number of bits will be even. In this case, the parity bit will be 1 and code to be transmitted will be 11010111.

XOR gates can be used as even parity checker. For example, with three inputs, the expression will be $F = A \oplus B \oplus C$ and the output is HIGH if one of the inputs or all three inputs are HIGH. Similarly, for eight inputs, the output is HIGH when odd number of inputs is HIGH.

Figure 3.10 shows the logic circuit using seven two-input XOR gates where the bits representing the code are A_0, A_1,…., A_6 and the parity bit is *P*. The output F will be HIGH when odd number of inputs is HIGH. So if the code is not transmitted correctly (say resulting in odd number of 1s), then the LED will light-up to show that an error has occured. On the other hand, with correct transmission, the number of 1s will be even and the output will be low (i.e. LED will not light-up).

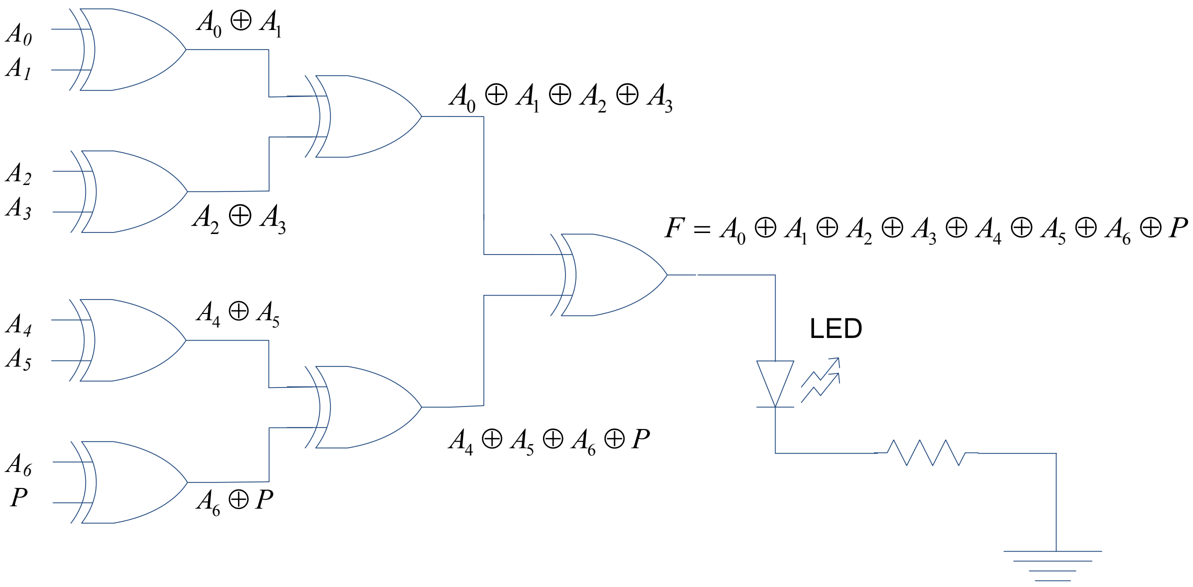

A_0
A_1
$A_0 \oplus A_1$
$A_0 \oplus A_1 \oplus A_2 \oplus A_3$
A_2
A_3
$A_2 \oplus A_3$
$F = A_0 \oplus A_1 \oplus A_2 \oplus A_3 \oplus A_4 \oplus A_5 \oplus A_6 \oplus P$
A_4
A_5
$A_4 \oplus A_5$
LED
$A_4 \oplus A_5 \oplus A_6 \oplus P$
A_6
P
$A_6 \oplus P$

Figure 3.10: XOR gate usage as even parity checker.

It should be obvious that XNOR gates can be used as odd parity checker as the output will be HIGH only when even number of inputs is HIGH.

4 Karnaugh Maps

In the previous chapter, simplification of expressions for combinatorial logic circuits was studied using Boolean algebra and DeMorgan's theorem. In this chapter, a different graphical based method called Karnaugh maps (or K-maps in short) will be studied to simplify the expressions. But before K-maps can be discussed, the two types of methods for writing logic circuit expressions will be discussed.

4.1 Sum of products

Sum of products (SOP) is a method to express the terms in a logic expression as a sum of products. For example:

$F = ABC + A\overline{B}C$ two product terms ABC and $A\overline{B}C$

$F = AB + A\overline{B} + \overline{A}B$ three product terms AB, $A\overline{B}$ and $\overline{A}B$

The logic circuit diagrams for these expressions are shown in Figure 4.1. It can be seen that each product term is connected using an OR gate.

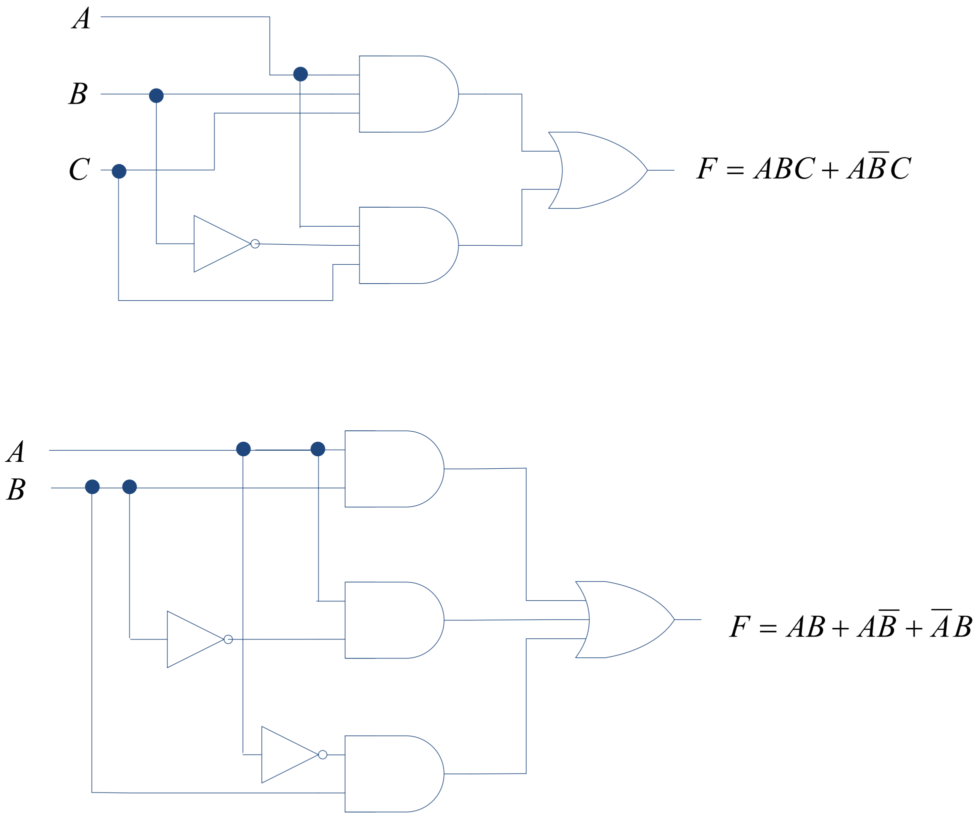

A
B
C
$F = ABC + A\overline{B}C$
A
B
$F = AB + A\overline{B} + \overline{A}B$

Figure 4.1: SOP logic circuit examples.

Tables 4.1 and 4.2 give the truth tables for these expressions. Each product term results in the output $F = 1$. For example, the expression $F = ABC + A\overline{B}C$ gives output of 1 when A=1, B=1 and C=1 for $F = ABC$ and similarly for $F = A\overline{B}C$, the output is 1 when A=1, $\overline{B} = 1$ (i.e. $B = 0$) and C=1.

Table 4.1: Truth table for $F = ABC + A\overline{B}C$

A	B	C	F	
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	0	
1	0	0	0	
1	0	1	1	$\longrightarrow F = A\overline{B}C$
1	1	0	0	
1	1	1	1	$\longrightarrow F = ABC$

Table 4.2: Truth table for $F = AB + A\overline{B} + \overline{A}B$

A	B	F	
0	0	0	
0	1	1	$\longrightarrow F = \overline{A}B$
1	0	1	$\longrightarrow F = A\overline{B}$
1	1	1	$\longrightarrow F = AB$

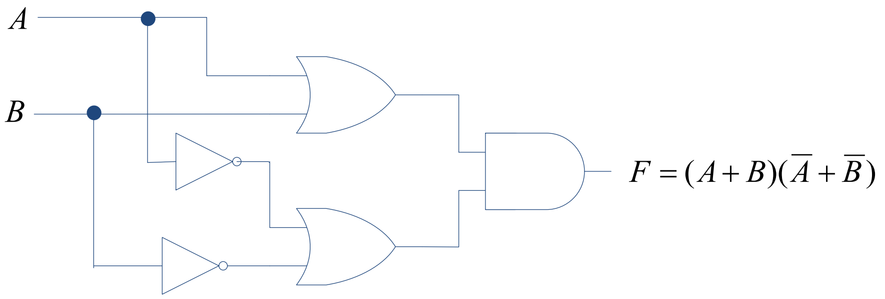
4.2 Product of sums

Products of sums (POS) is another method to express the terms in a logic circuit expression as a product of sums. For example:

$F = (A+B)(\overline{A}+\overline{B})$ two sum terms $(A+B)$ and $(\overline{A}+\overline{B})$

$F = (A+B+C)(A+C)(B+C)$ three sum terms $(A+B+C)$, $(A+C)$ and $(B+C)$

The logic circuit diagrams for these expressions are shown in Figure 4.2. An AND gate connects each of the sum terms.



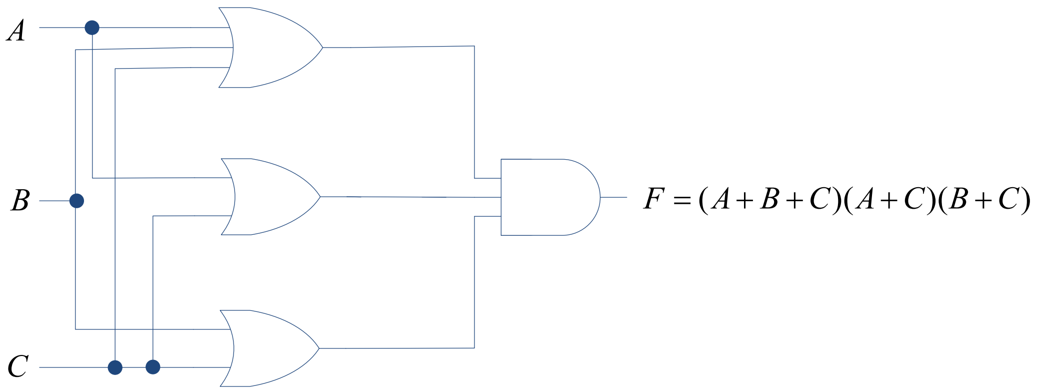


Figure 4.2: POS logic circuit examples.

The truth table for the first POS example, $F=(A+B)(\overline{A}+\overline{B})$ is given in Table 4.3. To understand the table, consider $\overline{F}=\overline{(A+B)(\overline{A}+\overline{B})}$ and using DeMorgan's theorem, we can obtain

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

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

$$\overline{F}=\overline{A}\,\overline{B}+AB$$

So, the truth table for POS terms can be easily completed for each term by giving output *F*=0 with the variables *A* and *B* following negative logic (i.e. complemented variable is logic 1 and uncomplemented variable is logic 0).

Table 4.3: Truth table for $F=(A+B)(\overline{A}+\overline{B})$

A	B	F	
0	0	0	$\longrightarrow$ $\overline{F}=\overline{A}\,\overline{B}$ or $F=(A+B)$
0	1	1	
1	0	1	
1	1	0	$\longrightarrow$ $\overline{F}=AB$ or $F=(\overline{A}+\overline{B})$

Table 4.4 gives the truth table for the second POS example, $F = (A+B+C)(A+C)(B+C)$ . . Following the similar procedure, consider $\overline{F} = \overline{(A+B+C)(A+C)(B+C)}$:

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

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

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

$$\overline{F} = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,\overline{C}(B+\overline{B}) + \overline{B}\,\overline{C}(A+\overline{A}) \qquad \text{since } X + \overline{X} = 1$$

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

$$\overline{F} = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}B\overline{C} + A\overline{B}\,\overline{C} \qquad \text{as } \overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,\overline{B}\,\overline{C} = \overline{A}\,\overline{B}\,\overline{C}$$

Table 4.4: Truth table for $F = (A+B+C)(A+C)(B+C)$.

A	B	C	F	
0	0	0	0	→ $F = (A+B+C)$
0	0	1	1	
0	1	0	0	→ $F = (A+C)$
0	1	1	1	
1	0	0	0	→ $F = (B+C)$
1	0	1	1	
1	1	0	1	
1	1	1	1	

POS expressions are not frequently employed in digital systems but discussed here for the sake of completeness.

4.3 K-maps

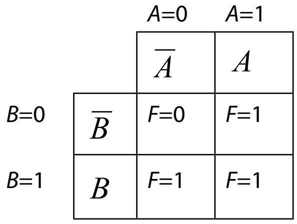
As mentioned earlier, K-map is a graphical method that is useful to simplify logic expressions. While the algebraic methods discussed in Chapter 3 can equally be used to simplify the expression, it is often easier to simplify an expression using K-maps when the number of variables is higher.

4.3.1 Two variable K-map

Consider a truth table as in Table 4.5 with two variables *A* and *B*. Its corresponding K-map is drawn in Figure 4.3. The K-map can be completed for variable combinations that give *F*=1 and *F*=0 as in the figure but it is common practice not to include *F*=0 in K-maps, so we shall only include combinations that give *F*=1 after this example.

Table 4.5: Truth table for two variable K-map example

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



		A=0	*A*=1
		$\overline{A}$	A
B=0	$\overline{B}$	*F*=0	*F*=1
B=1	B	*F*=1	*F*=1

Figure 4.3: K-map example from truth table in Table 4.5.

To simplify the expression, start by creating a loop for $F = A\overline{B} + AB$ (i.e. for adjacent cells) as shown in Figure 4.4(a). This loop is known as pair loop as it involves looping two 1s. Since $F = A\overline{B} + AB = A(B + \overline{B}) = A$, the looping will result in $F = A$, i.e. the variable in complement and uncomplemented form disappears. The process is repeated until all 1s have been looped (note that loops can overlap). Hence, repeat the looping as shown in Figure 4.4(b) where $\overline{A}B + AB = B(A + \overline{A})$. Since all 1s in the K-map have been looped, further simplification is not possible and the simplified expression is a combination of the two looped terms (each loop gives one term): $F = A + B$.

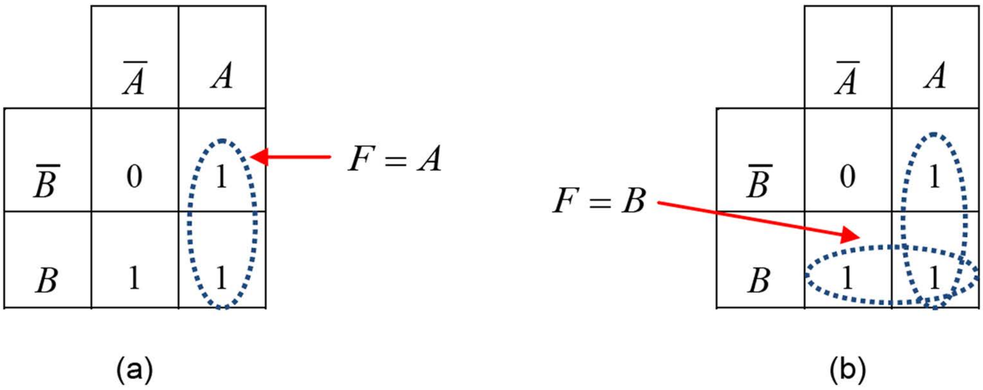


Figure 4.4: Two variable K-map looping: (a) $F = A$, (b) $F = B$. Simplified expression from both loops is $F = A + B$.

Consider solving the example algebraically from the truth table with K-map (each term is a variable combination that gives F=1):

$$F = \overline{A}B + A\overline{B} + AB \qquad \text{from Table 4.5 (see section 4.1 on SOP for more details)}$$
$$F = \overline{A}B + A\overline{B} + AB + AB \qquad \text{since } AB = AB + AB$$
$$F = A(\overline{B} + B) + B(\overline{A} + A)$$
$$F = A + B$$

The answer is obviously the same.

4.3.2 Three variable K-map

In addition to pair loops, we can have quad loops (involving four 1s). Consider a three variable logic expression: $F = ABC + A\overline{B}C + \overline{A}\overline{B}C + AB\overline{C} + A\overline{B}\overline{C}$ . . A truth table can be completed with each term ABC, $A\overline{B}C$, $\overline{A}\overline{B}C$, $AB\overline{C}$, $A\overline{B}\overline{C}$ giving output F=1 as shown in Table 4.6.

Table 4.6: Truth table for $F = ABC + A\overline{B}C + \overline{A}\overline{B}C + AB\overline{C} + A\overline{B}\overline{C}$.

A	B	C	F	
0	0	0	0	
0	0	1	1	→ $F = \overline{A}\overline{B}C$
0	1	0	0	
0	1	1	0	
1	0	0	1	→ $F = A\overline{B}\overline{C}$
1	0	1	1	→ $F = A\overline{B}C$
1	1	0	1	→ $F = AB\overline{C}$
1	1	1	1	→ $F = ABC$

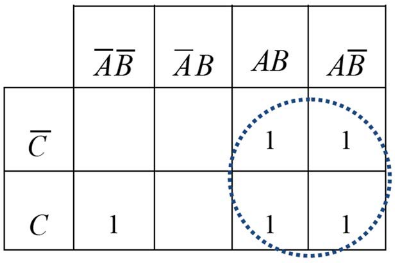
Figure 4.5 gives the completed three variable K-map. Note in particular on the sequence of variables *A* and *B* in the K-map. The sequence (order) follows gray code (00→01→11→10 with $\overline{A}\overline{B}$ → $\overline{A}B$ →AB→ $A\overline{B}$) where only one bit changes in adjacent cells. Figure 4.6(a) shows the quad loop applied for four adjacent 1s. Variables *B* and *C* are in complemented and uncomplemented forms in the quad loop, so these variables will disappear leaving only variable *A*. For this loop, algebraically,

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

However, it is not the end of the simplification as there is one more 1 that is not paired (for $F = \overline{A}\overline{B}C$). Loops in K-maps can wrap around, so create a pair loop as shown in Figure 4.6(b). Variable A is in complemented and uncomplemented forms in the pair loop, so it will disappear leaving only $\overline{B}C$. So the resulting simplified expression will be $F = A + \overline{B}C$.

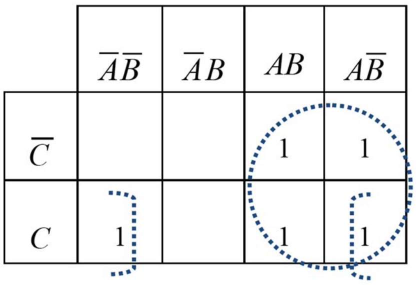
		00	01	11	10
		$\overline{A}\overline{B}$	$\overline{A}B$	AB	$A\overline{B}$
0	$\overline{C}$			1	1
1	C	1		1	1

Figure 4.5: Three variable K-map for . $F = ABC + A\overline{B}C + \overline{A}\overline{B}C + AB\overline{C} + A\overline{B}\overline{C}$.



	$\overline{A}\overline{B}$	$\overline{A}B$	AB	$A\overline{B}$
$\overline{C}$			1	1
C	1		1	1

(a)

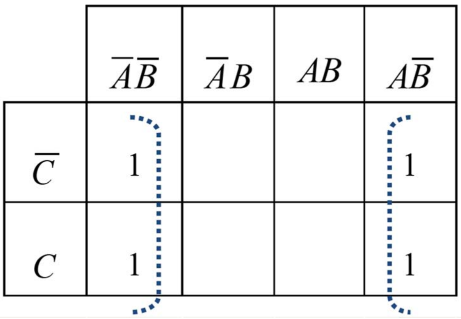


	$\overline{A}\,\overline{B}$	$\overline{A}B$	AB	$A\overline{B}$
$\overline{C}$			1	1
C	1		1	1

(b)

Figure 4.6: Three variable K-map shown in Figure 4.5: (a) quad loop (b) quad with pair loop.

As another example, consider $F = \overline{A}\,\overline{B} + A\overline{B}\,\overline{C} + A\overline{B}C$. Since one of the terms, $\overline{A}\,\overline{B}$ has only two variables, it should be expanded to give $\overline{A}\,\overline{B} = \overline{A}\,\overline{B}(C + \overline{C}) = \overline{A}\,\overline{B}C + \overline{A}\,\overline{B}\,\overline{C}$. So $F = \overline{A}\,\overline{B}C + \overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}C$. Now the K-map can be constructed as shown in Figure 4.7 and quad loop applied to give $F = \overline{B}$.



	$\overline{A}\,\overline{B}$	$\overline{A}B$	AB	$A\overline{B}$
$\overline{C}$	1			1
C	1			1

Figure 4.7: Three variable K-map for $F = \overline{A}\,\overline{B}C + \overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}C$.

It can be verified that algebraic simplification also gives the same result:

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

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

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

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

$$F = \overline{B}$$

4.3.3 Four variable K-map

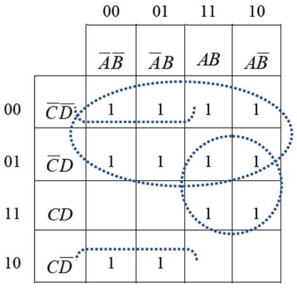
Consider a logic expression with four variables:

$$F = \overline{A}\,\overline{B}\,\overline{C}\,\overline{D} + \overline{A}\,\overline{B}\,\overline{C}D + AB\overline{C}\,\overline{D} + AB\overline{C}D + \overline{A}B\overline{C}\,\overline{D} + \overline{A}B\overline{C}D + A\overline{B}\,\overline{C}\,\overline{D} + A\overline{B}\,\overline{C}D$$
$$+ ABCD + A\overline{B}CD + \overline{A}\,\overline{B}C\overline{D} + \overline{A}BC\overline{D}$$

Figure 4.8 shows the K-map for this expression. With four variables, octet looping (with eight 1s) is possible. Note that loops should be as big as possible, so if there is a choice of two quad loops and one octet loop, then the octet loop should be created.

Only variable $\overline{C}$ remains from the octet loop as the other variables are in both complement and uncomplemented forms and hence disappear. There are two quad loops that give AD and $\overline{A}\,\overline{D}$ (wrapped around loop). The final expression is $F = AD + \overline{A}\,\overline{D} + \overline{C}$ ..

It should be obvious now that a pair loop removes one variable, a quad loop removes two variables while an octet loop removes three variables. In the example above, octet loop removed variables A , B and D .



		00	01	11	10
		$\overline{A}\,\overline{B}$	$\overline{A}B$	AB	$A\overline{B}$
00	$\overline{C}\,\overline{D}$	1	1	1	1
01	$\overline{C}D$	1	1	1	1
11	CD			1	1
10	$C\overline{D}$	1	1		

Figure 4.8: Four variable K-map.

4.3.4 Additional examples

Consider the truth table as in Table 4.7. For this example, let us obtain the simplified logic circuit diagram.

Table 4.7: Truth table for additional example 1

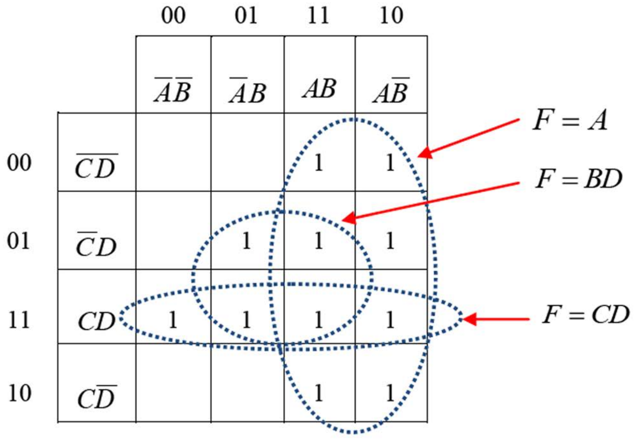
A	B	C	D	F	
0	0	0	0	0	
0	0	0	1	0	
0	0	1	0	0	
0	0	1	1	1	$\longrightarrow F = \overline{A}\overline{B}CD$
0	1	0	0	0	
0	1	0	1	1	$\longrightarrow F = \overline{A}B\overline{C}D$
0	1	1	0	0	
0	1	1	1	1	$\longrightarrow F = \overline{A}BCD$
1	0	0	0	1	$\longrightarrow F = A\overline{BCD}$
1	0	0	1	1	$\longrightarrow F = A\overline{B}\overline{C}D$
1	0	1	0	1	$\longrightarrow F = A\overline{B}C\overline{D}$
1	0	1	1	1	$\longrightarrow F = A\overline{B}CD$
1	1	0	0	1	$\longrightarrow F = AB\overline{CD}$
1	1	0	1	1	$\longrightarrow F = AB\overline{C}D$
1	1	1	0	1	$\longrightarrow F = ABC\overline{D}$
1	1	1	1	1	$\longrightarrow F = ABCD$

First, the logic expression should be obtained from the truth table and using it, K-map drawn (as shown in Figure 4.9). Next, we can obtain the simplified expression and with it draw the simplified logic circuit diagram as shown in Figure 4.10.

Logic expression:

$$F = \overline{A}\,\overline{B}CD + \overline{A}B\overline{C}D + \overline{A}BCD + AB\overline{C}\,\overline{D} + AB\overline{C}D + ABCD +$$
$$ABC\overline{D} + A\overline{B}\,\overline{C}\,\overline{D} + A\overline{B}\,\overline{C}D + A\overline{B}CD + A\overline{B}C\overline{D}$$

K-map:



		00	01	11	10
		$\overline{A}\,\overline{B}$	$\overline{A}B$	AB	$A\overline{B}$
00	$\overline{C}\,\overline{D}$			1	1
01	$\overline{C}D$		1	1	1
11	CD	1	1	1	1
10	$C\overline{D}$			1	1

Figure 4.9: K-map for additional example 1.

Simplified expression: $.F = A + BD + CD$.

Simplified logic circuit diagram:

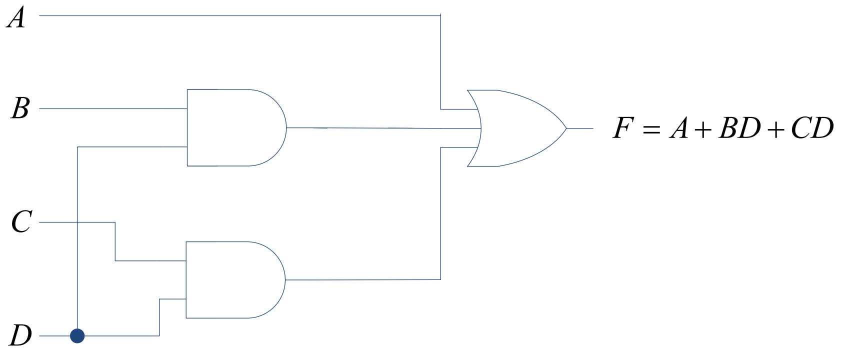


Figure 4.10: Simplified logic circuit diagram additional example 1.

As another example, consider a logic expression, $F = A\overline{B}\,\overline{C}\,\overline{D} + \overline{A}\,\overline{B}\,\overline{C}\,\overline{D} + AB\overline{C}D + \overline{A}\,\overline{B}CD + \overline{A}\,\overline{B}C\overline{D} + A\overline{B}C\overline{D}$ and its corresponding K-map as shown Figure 4.11.

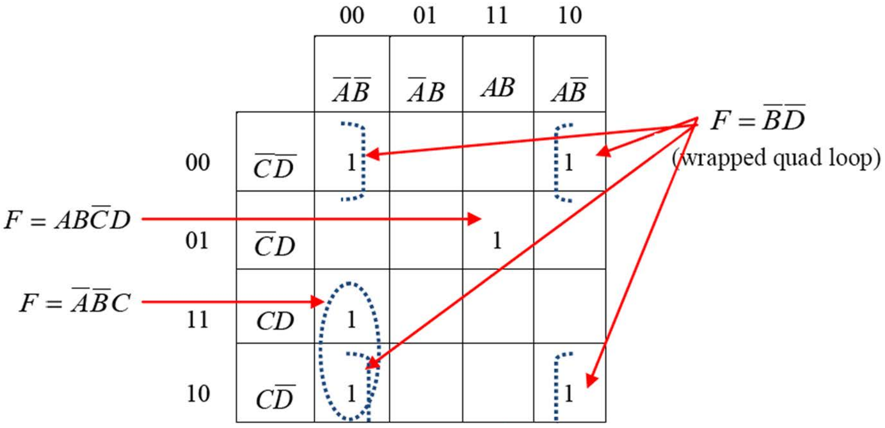


Figure 4.11: K-map for the additional example 2.

The wrapped around quad loop gives $\overline{B}\,\overline{D}$ while the pair loop gives $\overline{A}\,\overline{B}C$. There is a single 1 that can't be looped, so it remains as it is: $AB\overline{C}D$. So, the simplified expression is $F = AB\overline{C}D + \overline{A}\,\overline{B}C + \overline{B}\,\overline{D}$.

4.3.5 Don't care conditions

In digital logic design, we often encounter don't care conditions. These conditions are cases that won't occur in our design and hence the output can be set to any value (either 0 or 1). Don't care conditions are denoted using X in the truth tables and K-maps. For example, consider a seven segment display device as shown in Figure 4.12 that is commonly used to display hexadecimal characters.

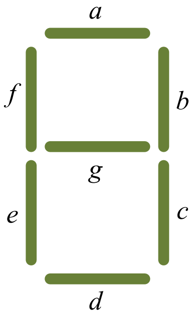


Figure 4.12: Seven segment display.

The device consists of light emitting diodes (LEDs)[7] that light up with different patterns to give the hexadecimal output as shown in Figure 4.13. Note that the hex characters A to F are normally displayed in a mixture of upper and lowercase to avoid ambiguity (for example differentiating D with 0, B with 8 etc).

[7] Newer devices operate using liquid crystal technology.

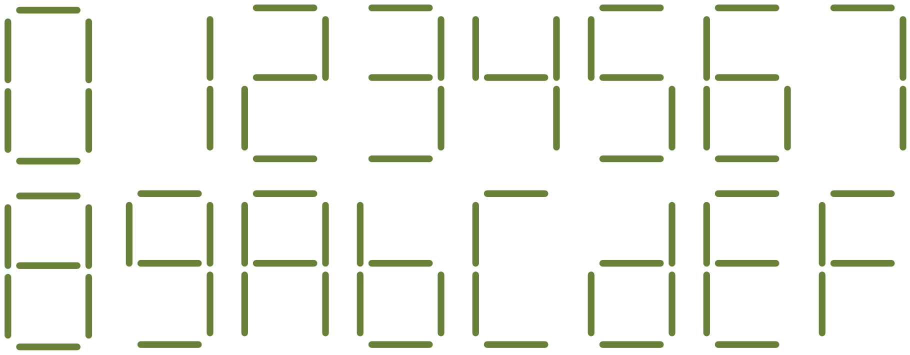


Figure 4.13: Hex characters displayed by the seven segment display.

Table 4.8 gives the character encodings for the seven LEDs (*a, b,* .... ,*g*), where a 1 denotes that the LED will be ON and a 0 denotes that the LED will be OFF. So to display numeral 0, LEDs *a, b, c, d, e,* and *f* will be turned on and LED *g* will be off. Similarly, to display character F, LEDs *a, e, f,* and *g* will be on while LEDs *b, c,* and *d* will be off.

Table 4.7: Character encodings for seven segment display LEDs

	LED						
Digit	*a*	*b*	*c*	*d*	*e*	*f*	*g*
0	1	1	1	1	1	1	0
1	0	1	1	0	0	0	0
2	1	1	0	1	1	0	1
3	1	1	1	1	0	0	1
4	0	1	1	0	0	1	1
5	1	0	1	1	0	1	1
6	1	0	1	1	1	1	1
7	1	1	1	0	0	0	0
8	1	1	1	1	1	1	1
9	1	1	1	1	0	1	1
A	1	1	1	0	1	1	1
b	0	0	1	1	1	1	1
C	1	0	0	1	1	1	0
d	0	1	1	1	1	0	1
E	1	0	0	1	1	1	1
F	1	0	0	0	1	1	1

Now, for the sake of discussing the don't care conditions, consider that we are going to use the seven segment display only to display the decimal numerals (i.e. 0 to 9). So, while designing the necessary wiring for the device, we can now ignore displays for the rest of the characters A to F. This situation will be denoted with X as in Table 4.8. Let us obtain the logic expression for LED *a*. To avoid confusion with the hex characters, we'll denote the variables as *P, Q, R,* and *S* instead of *A, B, C* and *D* as used earlier. Four variables (i.e. four inputs) are required since we have ten possible combinations.

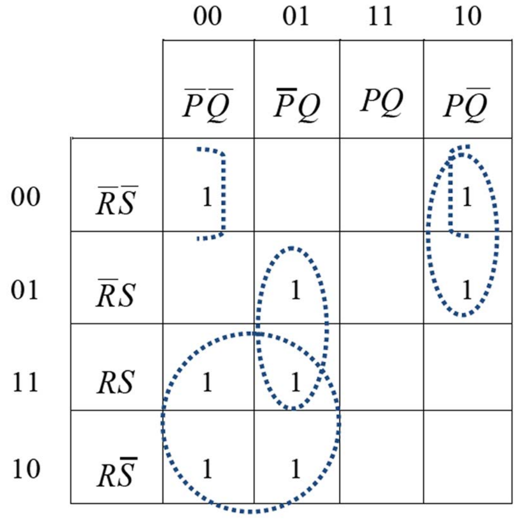
Table 4.8: Seven segment display LED encoding for decimals (showing don't care conditions)

	LED						
Digit	*a*	*b*	*c*	*d*	*e*	*f*	*g*
0	1	1	1	1	1	1	0
1	0	1	1	0	0	0	0
2	1	1	0	1	1	0	1
3	1	1	1	1	0	0	1
4	0	1	1	0	0	1	1
5	1	0	1	1	0	1	1
6	1	0	1	1	1	1	1
7	1	1	1	0	0	0	0
8	1	1	1	1	1	1	1
9	1	1	1	1	0	1	1
A	X	X	X	X	X	X	X
b	X	X	X	X	X	X	X
C	X	X	X	X	X	X	X
d	X	X	X	X	X	X	X
E	X	X	X	X	X	X	X
F	X	X	X	X	X	X	X

Using the truth table, we can now construct the K-map as shown in Figure 4.14 (without considering don't care conditions) and Figure 4.15 (with don't care conditions).

Table 4.9: Truth table for LED *a*

Digit	P	Q	R	S	LED *a*
0	0	0	0	0	1
1	0	0	0	1	0
2	0	0	1	0	1
3	0	0	1	1	1
4	0	1	0	0	0
5	0	1	0	1	1
6	0	1	1	0	1
7	0	1	1	1	1
8	1	0	0	0	1
9	1	0	0	1	1
A	1	0	1	0	X
b	1	0	1	1	X
C	1	1	0	0	X
d	1	1	0	1	X
E	1	1	1	0	X
F	1	1	1	1	X

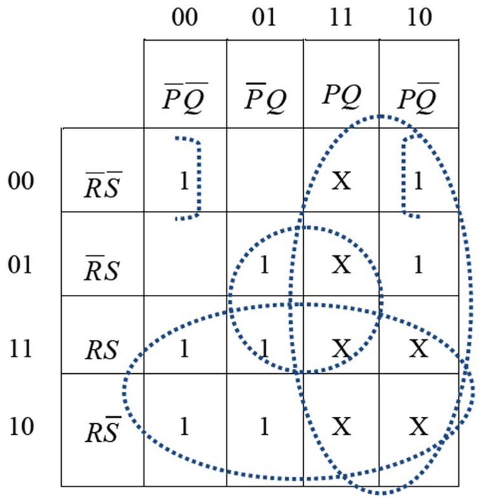


		00	01	11	10
		$\overline{P}\overline{Q}$	$\overline{P}Q$	PQ	$P\overline{Q}$
00	$\overline{R}\overline{S}$	1			1
01	$\overline{R}S$		1		1
11	RS	1	1		
10	$R\overline{S}$	1	1		

Figure 4.14: K-map for LED *a* without considering don't care conditions.

The simplified expression without considering don't care conditions is $LED_a = \overline{P}R + \overline{P}QS + P\overline{Q}\overline{R} + \overline{Q}\overline{R}\overline{S}$. Note that the solution is not unique as the wrapped around pair loop could also be formed for $\overline{P}\overline{Q}\overline{R}\overline{S}$ and $\overline{P}\overline{Q}R\overline{S}$ giving $\overline{P}\overline{Q}\overline{S}$ instead of $\overline{Q}\overline{R}\overline{S}$ as shown for $P\overline{Q}\overline{R}\overline{S}$ and $\overline{P}\overline{Q}R\overline{S}$. With this, the simplified expression will be $LED_a = \overline{P}R + \overline{P}QS + P\overline{Q}\overline{R} + \overline{P}\overline{Q}\overline{S}$.

Now consider Figure 4.15 where the don't care conditions are accounted. Since X is either 0 or 1, we can assume it to be 1 and use in the looping procedures.



		00	01	11	10
		$\overline{P}\overline{Q}$	$\overline{P}Q$	PQ	$P\overline{Q}$
00	$\overline{R}\overline{S}$	1		X	1
01	$\overline{R}S$		1	X	1
11	RS	1	1	X	X
10	$R\overline{S}$	1	1	X	X

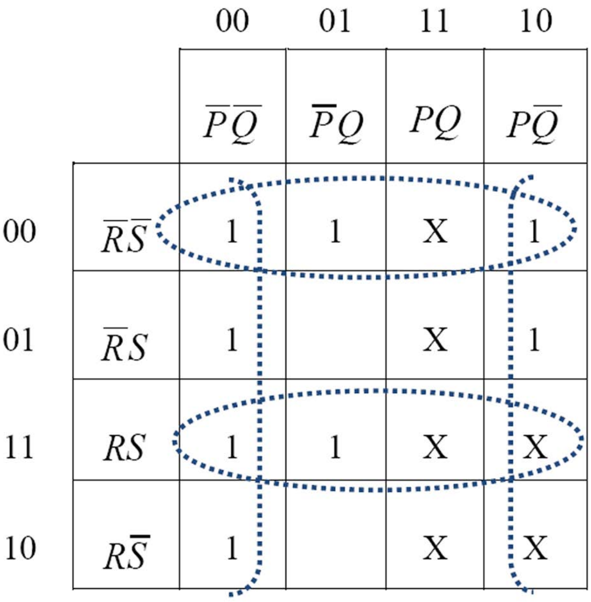
Figure 4.15: K-map for LED *a* (considering don't care conditions).

The simplified expression is now $LED_a = P + R + QS + \overline{Q}\overline{R}\overline{S}$ and it can be seen that the expression is made simpler by considering the don't care conditions.

As a final example for the chapter, let us obtain the logic expression for LED *b*. Table 4.10 gives the truth table and Figure 4.16 shows the K-map with don't care conditions. The simplified logic expression is $LED_b = RS + \overline{R}\overline{S} + \overline{Q} = \overline{R \oplus S} + \overline{Q}$. It should not be forgotten that the loops should be as big as possible.

Table 4.10: Truth table for LED *b*

Digit	*P*	*Q*	*R*	*S*	LED *b*
0	0	0	0	0	1
1	0	0	0	1	1
2	0	0	1	0	1
3	0	0	1	1	1
4	0	1	0	0	1
5	0	1	0	1	0
6	0	1	1	0	0
7	0	1	1	1	1
8	1	0	0	0	1
9	1	0	0	1	1
A	1	0	1	0	X
b	1	0	1	1	X
C	1	1	0	0	X
d	1	1	0	1	X
E	1	1	1	0	X
F	1	1	1	1	X



		00	01	11	10
		$\overline{P}\overline{Q}$	$\overline{P}Q$	PQ	$P\overline{Q}$
00	$\overline{R}\overline{S}$	1	1	X	1
01	$\overline{R}S$	1		X	1
11	RS	1	1	X	X
10	$R\overline{S}$	1		X	X

Figure 4.16: K-map for LED *b* with don't care conditions.

5 Bistable Multivibrator Circuits

In this chapter, circuits that have two stable states (i.e. off and on) will be studied. These circuits are also commonly known as flip-flops. As they have two stable states (i.e. logic 0 or 1), they are useful to store one bit of digital data, i.e. as memory elements. Several types of flip-flops will be studied before we look at other multivibrators to generate single and train of pulses.

Figure 5.1 shows a general flip-flop symbol. Usually, there are one or two inputs to the flip-flop and the output also has a complement. The inputs are either logic 0 or 1 and commonly known as set (or preset) input when equal to 1 (HIGH state) and reset (or clear) input when equal to 0 (LOW state).

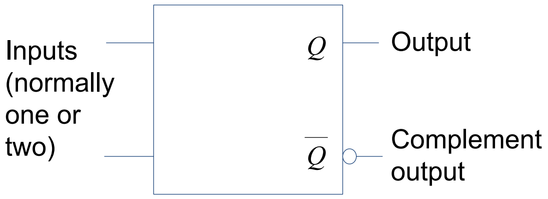
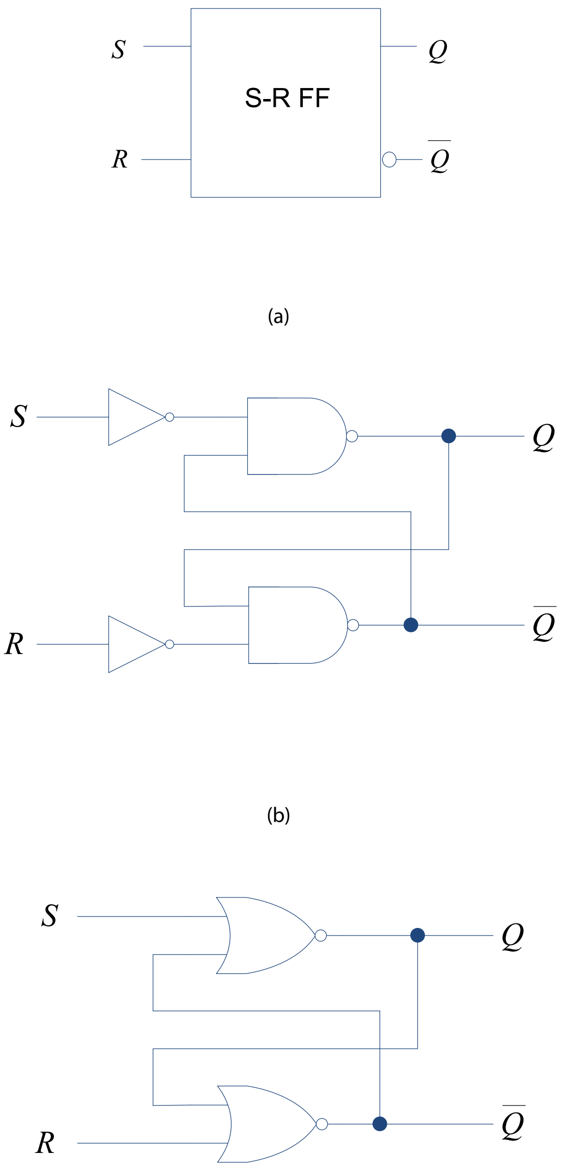


Figure 5.1: General flip-flop symbol.

5.1 S-R flip-flop

S-R flip-flop (also known as set-reset or latch) can be constructed using NOR or NAND gates. Both types of flip-flops are shown in Figure 5.2. The truth table for the S-R flip-flop is shown in Table 5.1. Q^+ here denotes the next state of output Q.



(a)

(b)

(c)

Figure 5.2: S-R flip-flop: (a) general symbol (b) using NAND gates (c) using NOR gates.

Table 5.1: Truth table for S-R flip-flop

S	R	Q^+	
0	0	Q	No change, $Q^+=Q$
1	0	1	Set output $Q^+=1$
0	1	0	Clear output $Q^+=0$
1	1	-	Invalid state

It can be seen that for both NAND and NOR types, there is feedback for the output and complemented output to the inputs. When both S and R inputs are LOW (logic 0), the output of the flip-flop will be the same as its previous state, i.e. no change in the Q state. A HIGH (logic 1) S input to the flip-flop will cause the output Q^+ to change state to HIGH. Similarly, R=1 input will cause the S-R flip-flop's output Q^+=0. It should be obvious that the S input sets the flip-flop to logic 1 while the R input resets the flip-flop to logic 0. S-R flip-flop output is not defined when both inputs are 1, so this situation should be avoided when using the S-R flip-flop. In the above discussion, state of $\overline{Q}$ will be opposite to the state of Q at all times.

5.1.1 S-R flip-flop with Enable input

An enabling input can be used to control the operation of the flip-flops as shown in Figure 5.3. Here the inputs R and S will only have an effect on the output Q^+ if the enable input is 1. When E=1, the NAND gates (in bold) will act as inverters, thereby the circuit behaving exactly like the NAND gate S-R flip-flop in Figure 5.2(b). Table 5.2 gives the truth table values.

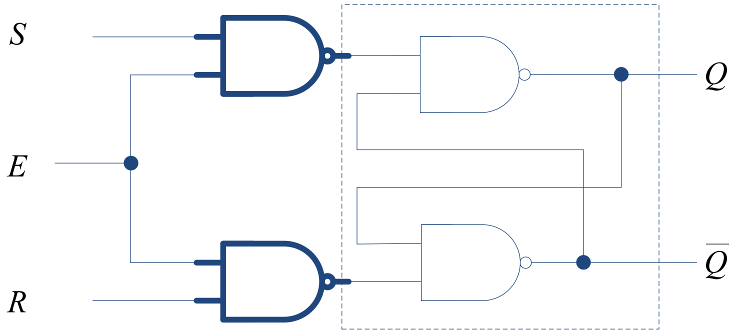


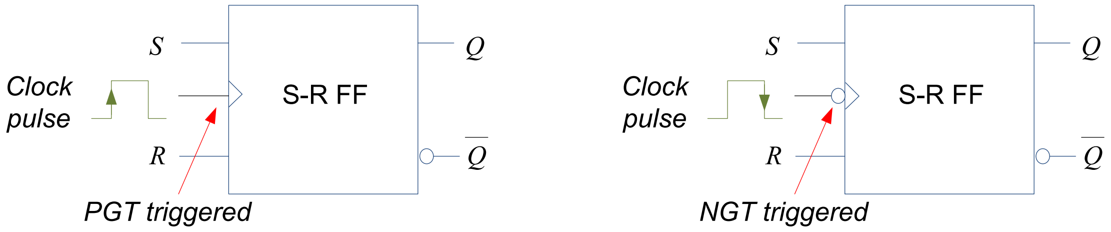
Figure 5.3: S-R flip-flop with Enable input.

Table 5.2: Truth table for S-R flip-flop with Enable input

E	S	R	Q^+	
0	0	0	Q	No change, $Q^+=Q$
0	1	0	Q	No change, $Q^+=Q$
0	0	1	Q	No change, $Q^+=Q$
0	1	1	Q	No change, $Q^+=Q$
1	0	0	Q	No change, $Q^+=Q$
1	1	0	1	Set output $Q^+=1$
1	0	1	0	Clear output $Q^+=0$
1	1	1	-	Invalid state

5.1.2 Clocked S-R flip-flop

Similar to the enable input, there could be a clock (i.e. pulsed) input to the flip-flop. Clocked S-R flip-flop is shown in Figure 5.4 where the edge of the clock (either positive or negative) triggers the change in the flip-flop state. The negative edge of the clock occurs when the clock pulse drops from logic 1 to 0 and is also known as negative going transition (NGT) while the positive going transition (PGT) occurs when the clock pulse goes from logic 0 to 1. An opposite clock edge will not affect the flip-flop output. For example, a negative edge triggered flip-flop will not change state during the positive edge. Table 5.3 shows the truth table for the NGT clocked flip-flop where it can be seen that the flip-flop changes state during the corresponding negative triggering edge of the clock. The PGT clocked flip-flop behaves similarly except that the change (if any) occurs during the positive edge transition of the clock.



(a) (b)

Figure 5.4: Clocked S-R flip-flops: (a) PGT (b) NGT, note the bubble for NGT triggered flip-flop.

Table 5.3: Truth table for NGT clocked S-R flip-flop

Clock	*S*	*R*	Q^+	
↓	0	0	*Q*	No change, $Q^+=Q$
↓	1	0	1	Set output $Q^+=1$
↓	0	1	0	Clear output $Q^+=0$
↓	1	1	-	Invalid state

A few examples using timing diagrams follow to illustrate the behaviour of clocked S-R flip-flops. Figure 5.5 shows an example on how the timing diagram changes for NGT clocked S-R flip-flop. Any change in the output Q will only occur during NGT (shown by t_1, t_2, …, t_5):

- At time t_1, Q goes to logic 1 as S=1, R=0
- At time t_2, Q remains at logic 1 as S=1, R=0
- At time t_3, Q goes to logic 0 as S=0, R=1
- At time t_4, Q remains at logic 0 as S=0, R=1
- At time t_5, Q goes to logic 1 as S=1, R=0

There won't be any changes during t_{PGT} for negative edge triggered flip-flop.

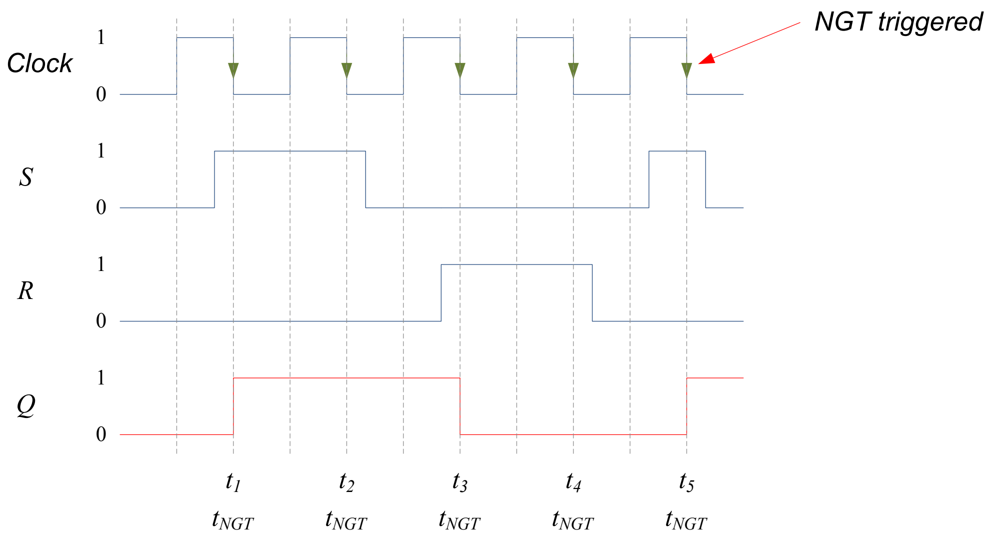


Figure 5.5: Timing diagram for NGT clocked S-R flip-flop example.

A PGT clocked S-R flip-flop timing diagram example is shown in Figure 5.6. Any change in the output Q will only occur during PGT (shown by t_1, t_2, …, t_5):

- At time t_1, Q goes to logic 1 as S=1, R=0
- At time t_2, Q remains at logic 1 as S=0, R=0
- At time t_3, Q goes to logic 0 as S=0, R=1
- At time t_4, Q goes to logic 1 as S=1, R=0
- At time t_5, Q goes to logic 0 as S=0, R=1

There won't be any changes during t_{NGT} for positive edge triggered flip-flop.

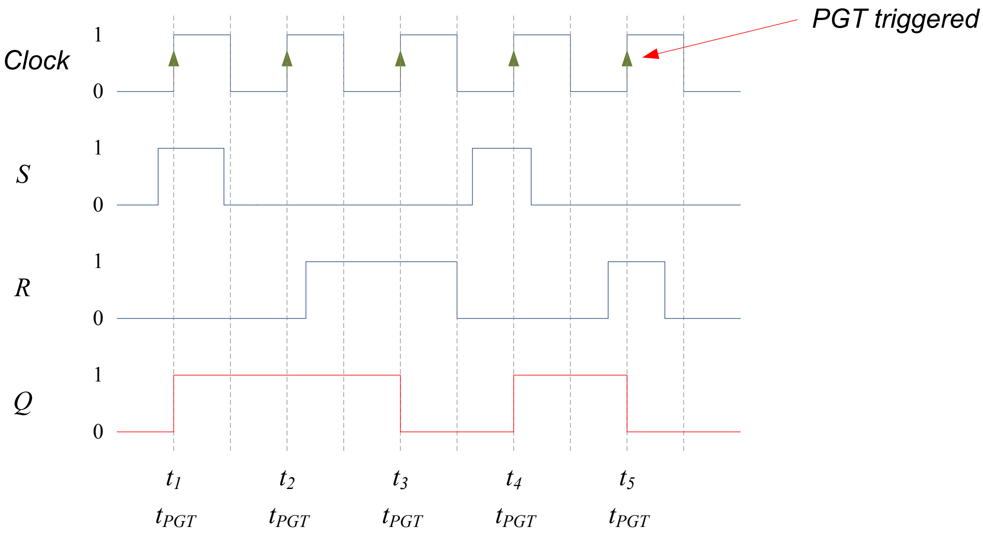


Figure 5.6: Timing diagram for PGT clocked S-R flip-flop example.

5.1.3 Asynchronous flip-flop inputs

The S and R inputs are known as synchronous inputs as their effects are synchronised to the clock input. Flip-flops can also have asynchronous inputs that can affect the output at any time irrespective of the clock pulse. Figure 5.7 shows the NGT S-R flip-flop symbol with two additional pulse inputs: ($\overline{PRE}$) that sets the output to logic 1 and clear ($\overline{CLR}$) that sets the output to logic 0. Both these inputs are ACTIVE LOW[8] (shown with an overbar, also note the existence of the bubble in the figure), which means that a logic 0 input will affect the flip-flop output rather than logic 1. Asynchronous inputs always take precedence over the S and R inputs.

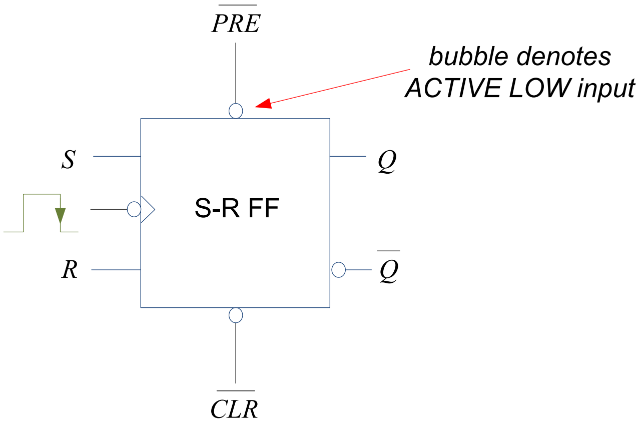


Figure 5.7: NGT S-R flip-flop symbol with asynchronous inputs.

Figure 5.8 illustrates the effect of these asynchronous inputs using a timing diagram. When $\overline{PRE}$ and $\overline{CLR}$ equals logic 1, the flip-flop behaves exactly as an NGT clocked S-R flip-flop. However, when either pulse becomes active (i.e. goes to logic 0), the effect on output Q is immediate (i.e. independent of the clock pulse):

- At time t_1, Q goes to logic 1 as S=1, R=0
- At time t_{12}, Q goes to logic 0 as $\overline{CLR} = 0$
- At time t_2, Q goes to logic 1 as S=1, R=0
- At time t_3, Q goes to logic 0 as S=0, R=1
- At time t_{34}, Q goes to logic 1 as $\overline{PRE} = 0$
- At time t_4, Q goes to logic 0 as S=0, R=1
- At time t_5, Q goes to logic 1 as S=1, R=0

[8] S and R inputs that either sets or resets the flip-flop on logic 1 are examples of ACTIVE HIGH inputs.

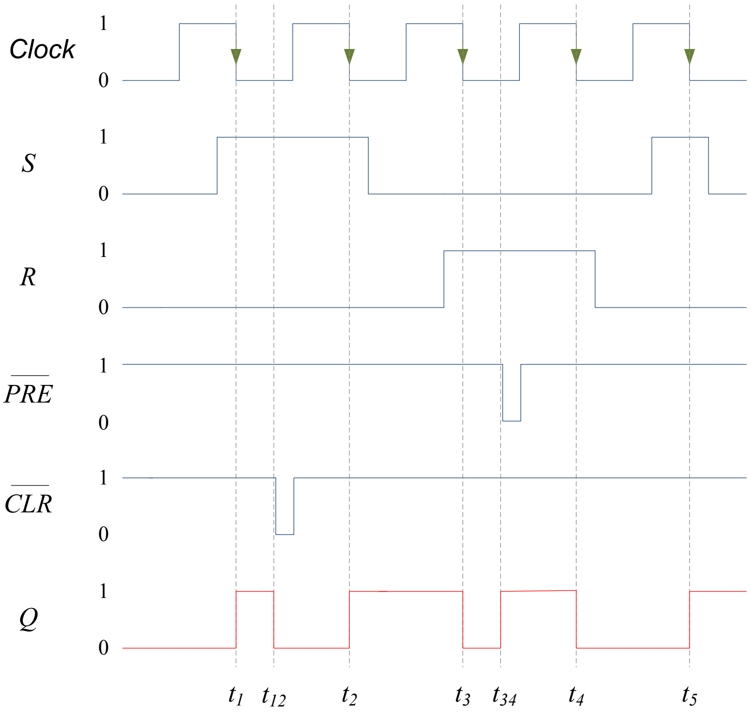


Figure 5.8: NGT S-R flip-flop timing diagram example with asynchronous inputs.

5.2 J-K flip-flop

R-S flip-flop is not very commonly used in digital systems due to the invalid state that can occur when both inputs are logic 1. J-K (named after Jack Kilby) flip-flop overcomes this problem by toggling (i.e. going to opposite state) when inputs J=K=1. Table 5.4 shows the truth table for this flip-flop.

Table 5.4: Truth table for J-K flip-flop

J	K	Q^+	
0	0	Q	No change, Q^+=Q
1	0	1	Set output Q^+=1
0	1	0	Clear output Q^+=0
1	1	$\overline{Q}$	Toggle, $Q^+ = \overline{Q}$

Similar to R-S flip-flop, J-K flip-flop can have enable input, clocked (NGT or PGT) and asynchronous inputs. Figure 5.9 shows the PGT J-K flip-flop symbol.

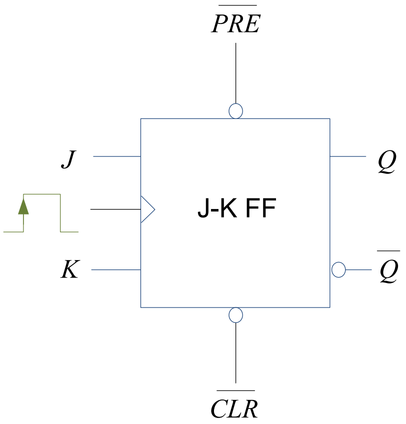


Figure 5.9: PGT J-K flip-flop symbol with asynchronous inputs.

A timing diagram example for J-K flip-flop is given in Figure 5.10. The previous discussions for S-R flip-flop hold for J-K flip-flop except that when *J*=*K*=1, the output toggles from its previous state:

- At time t_1, *Q* goes to logic 1 as *J*=1, *K*=0
- At time t_2, *Q* toggles to logic 0 as *J*=1, *K*=1
- At time t_3, *Q* remains at logic 0 as *J*=0, *K*=0
- At time t_4, *Q* toggles to logic 1 as *J*=1, *K*=1
- At time t_5, *Q* goes to logic 0 as *J*=0, *K*=1

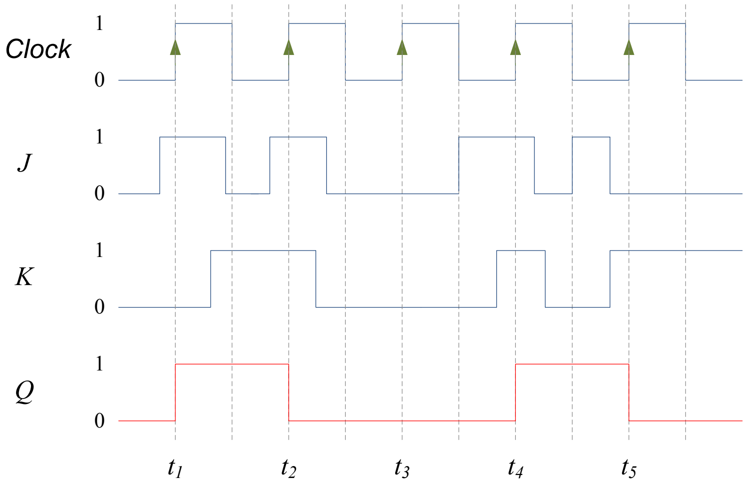


Figure 5.10: PGT J-K flip-flop timing diagram example.

Figure 5.11 gives a timing diagram example of NGT J-K flip-flop with asynchronous inputs:

- At time t_1, *Q* toggles to logic 1 as *J*=1, *K*=1
- At time t_{12}, *Q* goes to logic 0 as $\overline{CLR} = 0$
- At time t_2, *Q* goes to logic 1 as *J*=1, *K*=0
- At time t_3, *Q* goes to logic 0 as *J*=0, *K*=1
- At time t_{34}, *Q* goes to logic 1 as $\overline{PRE} = 0$
- At time t_4, *Q* remains at logic 1 as *J*=0, *K*=0
- At time t_5, *Q* remains at logic 1 as *J*=1, *K*=0

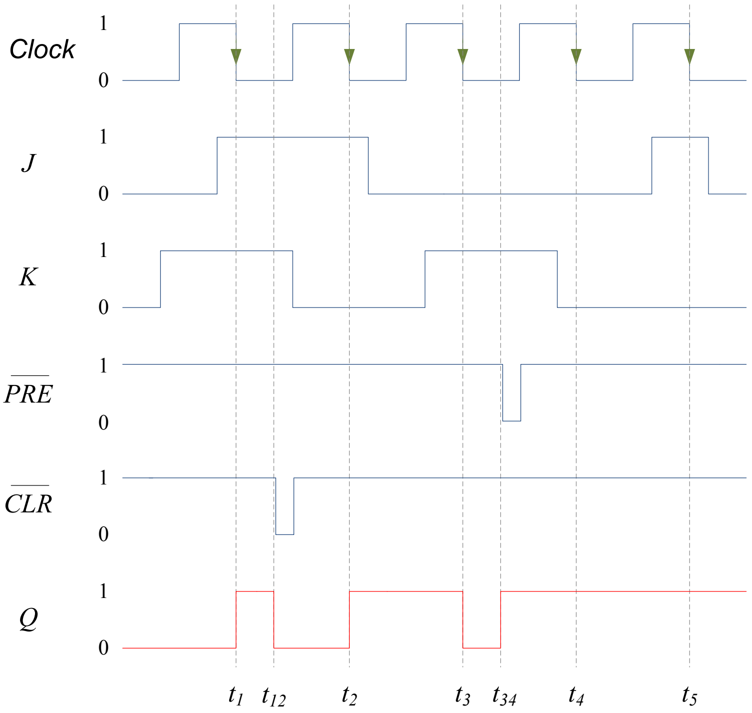


Figure 5.11: NGT J-K flip-flop timing diagram example with asynchronous inputs.

5.2.1 Master-slave flip-flop

As we will see in a later chapter, a sequence of flip-flops are usually connected to each other with a single clock and an example is shown in Figure 5.12. Since there could be a delay in the clock pulse to arrive at FF_2 as compared to FF_1 due to the longer wiring, the output can become unpredictable. To avoid this problem, a master-slave flip-flop can be used where FF_1 is the master and FF_2 is the slave. The inputs to FF_1 are used to determine the output of the master during *CLK*=HIGH and this output is then transferred to the slave when *CLK*=LOW. However, master-slave flip-flops have become obsolete with the design of modern edge-triggered flip-flops that responds with sufficient speed and reliability.

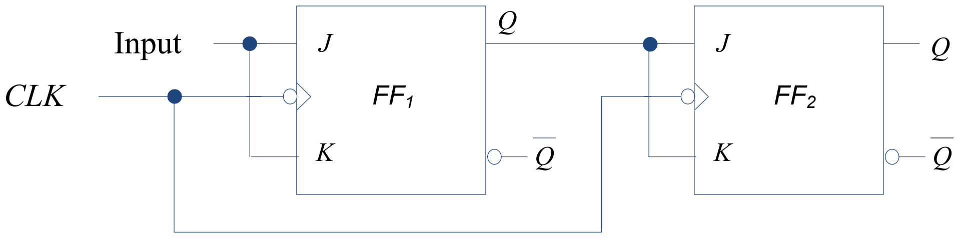


Figure 5.12: Two flip-flops connected with a single clock.

5.3 D flip-flop

D flip-flop is also known as data flip-flop since it can store a single bit of data. The output of the flip-flop Q follows the *single* input D at the respective clock pulses. Figure 5.13 shows the D flip-flop symbol.

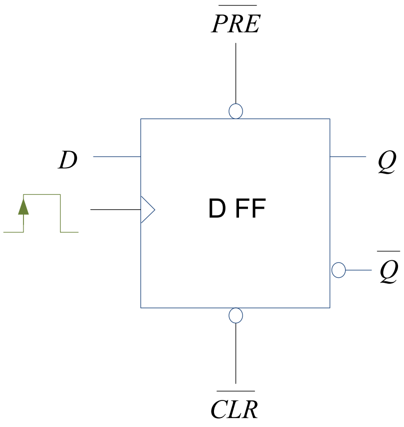


Figure 5.13: PGT D flip-flop general symbol.

Table 5.5 gives the truth table for D flip-flop. The output Q will follow the input D at either NGT or PGT clock depending on whether it is negative or positive edge triggered flip-flop. The D flip-flop can also have asynchronous inputs such as $\overline{PRE}$ and $\overline{CLR}$ that affect the output Q independently of the clock.

Table 5.5: Truth table for D flip-flop

D	Q^+	
0	0	$Q^+=D$
1	1	$Q^+=D$

Figure 5.14 gives an example of the D flip-flop timing diagram:

- At time t_1, Q goes to logic 1 as D=1
- At time t_2, Q goes to logic 0 as D=0
- At time t_{23}, Q goes to logic 1 as $\overline{PRE} = 0$
- At time t_3, Q remains at logic 1 as D=1
- At time t_4, Q remains at logic 1 as D=1
- At time t_{45}, Q goes to logic 0 as $\overline{CLR} = 0$
- At time t_5, Q remains at logic 0 as D=0

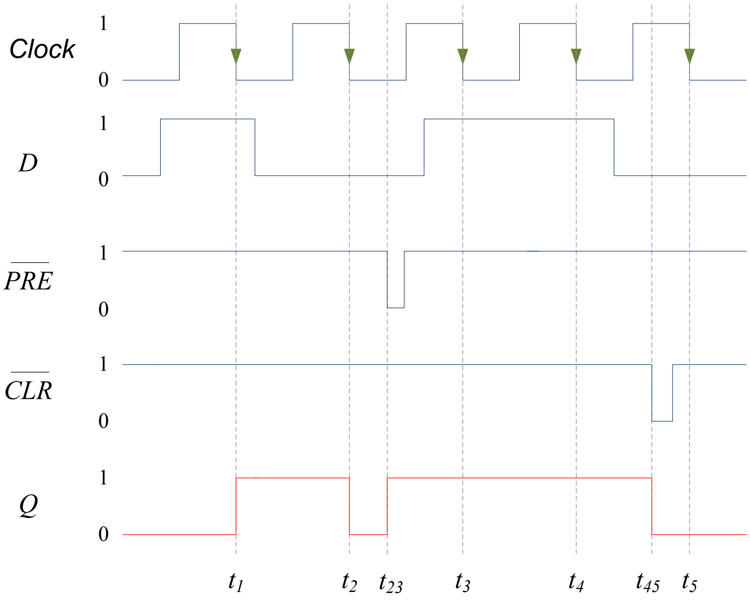


Figure 5.14: NGT D flip-flop timing diagram example.

Figure 5.15 shows how a J-K flip-flop can be used to construct a D flip-flop. When D=1, inputs to J-K flip-flop: $D = J = 1$ and $K = \overline{D} = 0$ and hence, Q = 1. Similarly, when D=0, inputs to J-K flip-flop: $D = J = 0$ and $K = \overline{D} = 1$ and hence, Q = 0. So the output Q follows input D as in the D flip-flop.

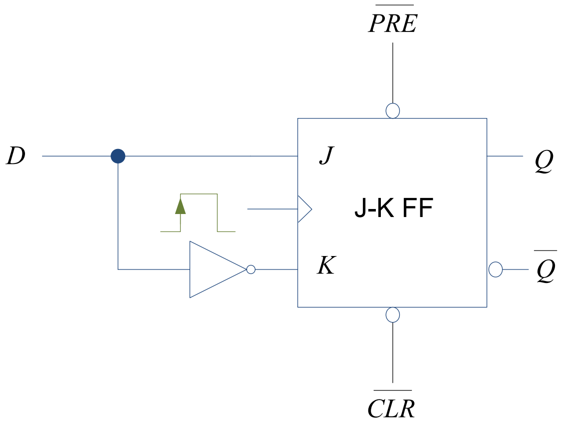
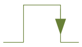


Figure 5.15: PGT D flip-flop constructed using J-K flip-flop.

5.4 T flip-flop

The final flip-flop to be considered in this chapter is T flip-flop. The truth table of the T flip-flop is given in Table 5.6 assuming it is triggered by an NGT clock. The output for T flip-flop toggles at *T*=1 thereby giving a clock like waveform but with half the frequency as shown by the timing diagram in Figure 5.16. When *T*=0, the output *Q* does not change.

Table 5.6: Truth table for NGT T flip-flop

	T	Q^+	
	0	Q	No change, $Q^+ = Q$
	1	$\overline{Q}$	Q^+ toggles, $Q^+ = \overline{Q}$

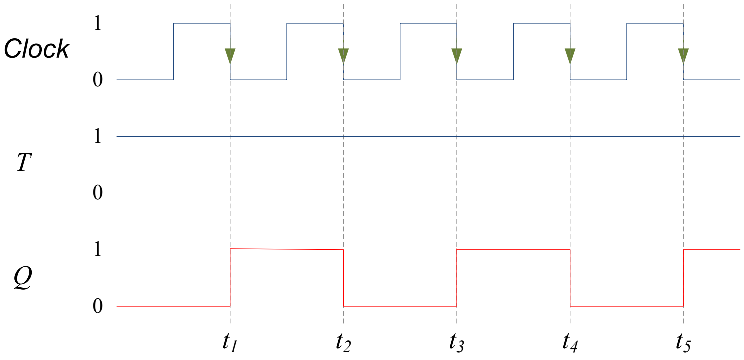


Figure 5.16: NGT T flip-flop where *T*=1, hence flip-flop operates in toggle mode at each clock trigger.

Figure 5.17 shows the general T flip-flop symbol and also how a J-K flip-flop can be used to construct a T flip-flop by tying J-K inputs together. When *J*=*K*=1, the flip-flop output toggles and when *J*=*K*=0, the flip-flop output does not change.

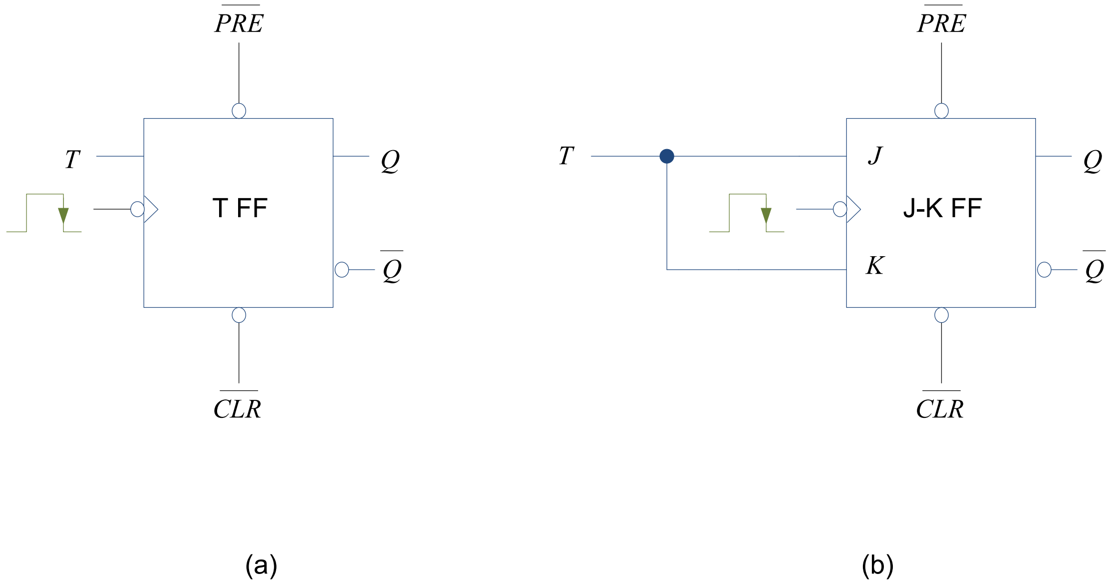


Figure 5.17: NGT T flip-flop: (a) general symbol (b) constructed using J-K flip-flop.

5.5 Monostable and astable multivibrators

So far, we have considered flip-flops that have two stable states. In this section, we will look at two devices, one that give short trigger pulses and another that gives two states that are free running. Monostable multivibrator is also known as one shot as it has one stable state (normally Q=0) and the other state (normally Q=1) occurs for a specific *tp* duration when triggered. Astable multivibrator does not have a stable state but switches continuously between two states (i.e. Q=0 and Q=1) which results in a train of square (or rectangular) wave pulses at a frequency determined by values of connected resistors and capacitors. Square wave pulses (i.e. with a 50% duty cycle) could be used as clock input.

5.5..1 Monostable multivibrator

Monostable multivibrator could be divided into two types: non-retriggerable and retriggerable. Non-retriggerable monostable multivibrator will ignore any trigger request during a *tp* pulse while the retriggerable one will re-trigger the pulse for another *tp* duration. The effects of both multivibrators are illustrated in the examples given in Figure 5.18. For non-retriggerable monostable multivibrator, $trig_2$ has no effect since it is within the duration of the *tp* pulse triggered by $trig_1$. However, for retriggerable monostable multivibrator, *trig2* has the effect of extending the one shot pulse by *tp* duration.

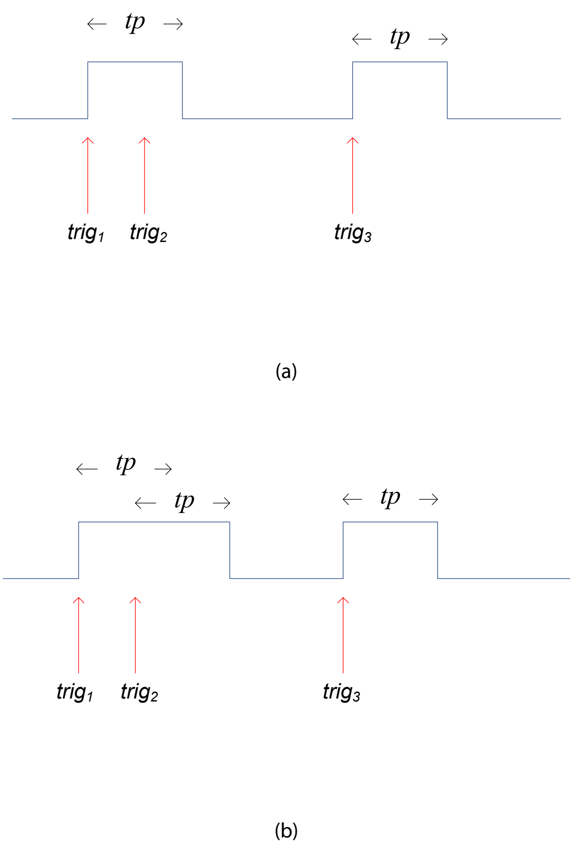


Figure 5.18: Monostable multivibrator: (a) non-retriggerable (b) retriggerable.

5.5.2 Astable multivibrator

Astable multivibrator designed using 555 timer IC is shown in Figure 5.19. It generates rectangular pulses with duration t_A and t_B. Duty cycle is defined as $t_B/(t_A + t_B)$. To generate clock pulses, the duty cycle has to be 50%, i.e. $t_A = t_B$.

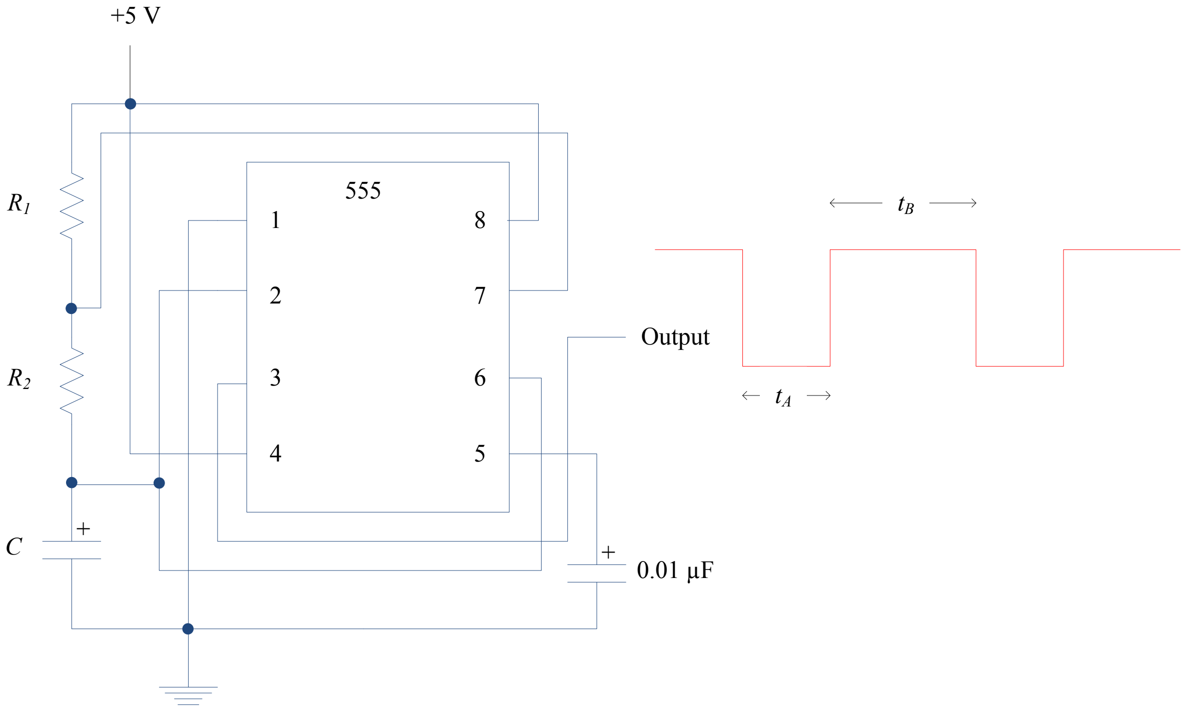


Figure 5.19: Astable multivibrator using 555 timer IC.

The values of the resistors R_1 and R_2 and the capacitor C will affect the durations of t_A and t_B:

$t_A = 0.693\ R_2C$

$t_B = 0.693\ (R_1 + R_2)C$

The frequency of the pulse is given by, *freq*=1/(t_A + t_B).

Consider an example where R_1=4.7 kΩ, R_2 = 10.0 kΩ, and C = 100ųF, we get

t_A = 0.693 R_2C

= 0.693 x (10 kΩ) x 100 μF

= 0.693 x (10000 Ω) x 0.0001 F

= 0.693 s

t_B = 0.693 (R_1 + R_2)C

= 0.693 x (4700 Ω+10000 Ω) x 100 μF

= 1.01871 s

Frequency = 1/(t_B + t_A) = 0.58421 Hz.

6 Arithmetic Circuits

In computers, arithmetic computations such as binary addition and subtraction are done in arithmetic logic unit (ALU) that consists of logic gates and flip-flops. Logic gates perform the arithmetic operation while the flip-flops (i.e. register and accumulator) are used as temporary memory storage (something like a scratch pad that we use to perform mathematical computation). We will look at adder and subtractor circuits in this chapter.

6.1 Half adder

Consider adding two bits, A_0 and B_0 to give sum Σ_0 and carry-out, C_1. Table 6.1 shows the possible combinations that can take place.

Table 6.1: Half adder combination

A_0	B_0		C_1	Σ
0	0		0	0
0	1		0	1
1	0		0	1
1	1		1	0

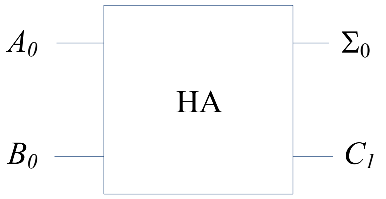


Figure 6.1: Half adder (HA) symbol.

Using K-maps as shown in Figure 6.2, we can obtain the logic expressions for Σ_0 and C_1. It can be seen that for Σ_0, it is not possible to simplify the expression as no looping is possible and the expression is

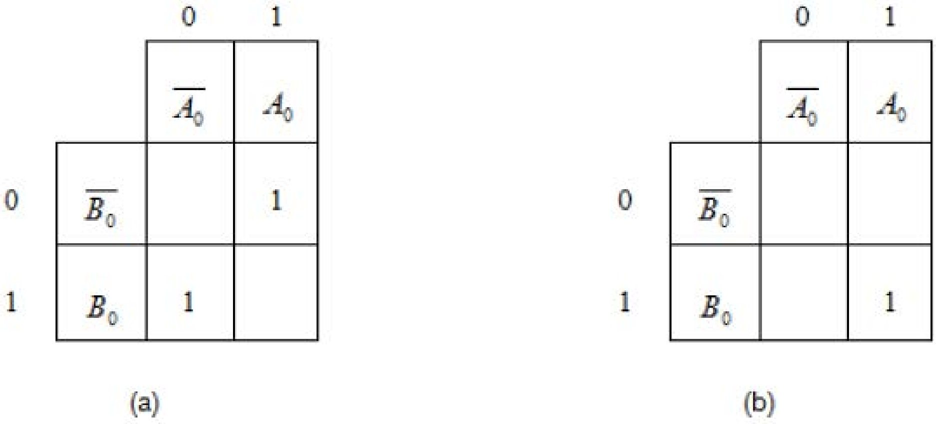
$$\Sigma_0 = A_0\overline{B_0} + \overline{A_0}B_0 .$$

Since this is XOR expression (see Section 3.5), it can also be expressed as

$$\Sigma_0 = A_0 \oplus B_0 .$$

Similarly, the expression for C_1 is

$C_1 = A_0 B_0$.



		0	1
		$\overline{A_0}$	A_0
0	$\overline{B_0}$		1
1	B_0	1	

(a)

		0	1
		$\overline{A_0}$	A_0
0	$\overline{B_0}$		
1	B_0		1

(b)

Figure 6.2: Half adder K-maps for (a) Σ_0 (b) C_1.

The half-adder logic circuit is shown in Figure 6.3.

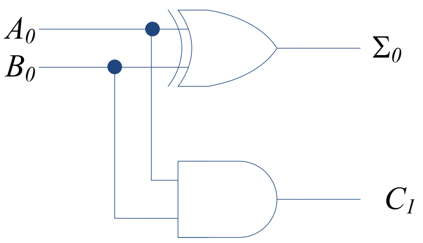


Figure 6.3: Half adder logic circuit.

6.2 Full adder

Very often when adding two bits, A_0 and B_0 to give sum Σ_0 and carry-out C_1, there can be another input, carry-in C_0 resulting from addition of previous bits. The possible combinations for a full adder are shown in Table 6.2 where it can be seen that the three binary inputs, A_0, B_0 and C_0 add to give the two binary outputs, Σ_0 and C_1. Full adder symbol is shown in Figure 6.4.

Table 6.2: Full adder combinations

A_0	B_0	C_0		C_1	Σ
0	0	0		0	0
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	1

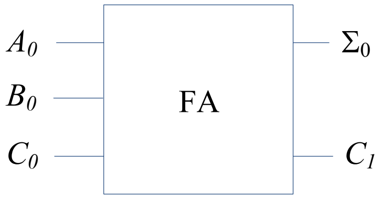


Figure 6.4: Full adder (FA) symbol.

K-maps for the two full adder outputs are shown in Figure 6.5. For Σ_0, no looping is possible and the expression is

$$\Sigma_0 = \overline{A_0}\,\overline{B_0}C_0 + \overline{A_0}B_0\overline{C_0} + A_0B_0C_0 + A_0\overline{B_0}\,\overline{C_0}$$

$$\Sigma_0 = C_0(\overline{A_0}\,\overline{B_0} + A_0B_0) + \overline{C_0}(\overline{A_0}B_0 + A_0\overline{B_0})$$

which can also be expressed in simpler form using XOR and XNOR expressions as $(\overline{A_0}B_0 + A_0\overline{B_0}) = A_0 \oplus B_0$ and $(\overline{A_0}\,\overline{B_0} + A_0B_0) = \overline{A_0 \oplus B_0}$ to give

$$\Sigma_0 = C_o(\overline{A_0 \oplus B_0}) + \overline{C_0}(A_0 \oplus B_0) .$$

We can actually simplify this further by allowing $X = A_0 \oplus B_0$:

$$\Sigma_0 = C_0\overline{X} + \overline{C_0}X .$$

Further simplification can be made using an XOR expression to give

$$\Sigma_0 = X \oplus C_0$$

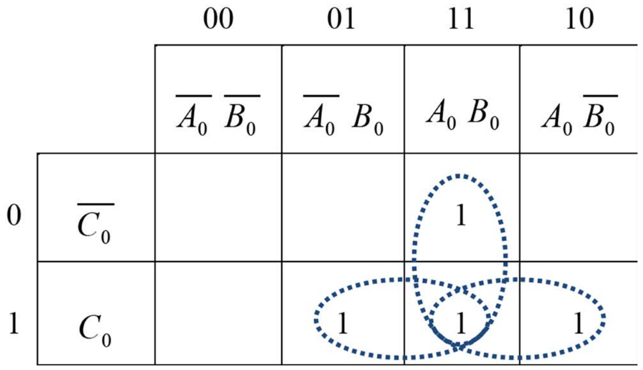
$$\Sigma_0 = A_0 \oplus B_0 \oplus C_0$$

For C_1, three pair loops are possible resulting in

$$C_1 = A_0B_0 + A_0C_0 + B_0C_o .$$

		00	01	11	10
		$\overline{A_0}\ \overline{B_0}$	$\overline{A_0}\ B_0$	$A_0\,B_0$	$A_0\,\overline{B_0}$
0	$\overline{C_0}$		1		1
1	C_0	1		1	

(a)



		00	01	11	10
		$\overline{A_0}\ \overline{B_0}$	$\overline{A_0}\ B_0$	$A_0\ B_0$	$A_0\ \overline{B_0}$
0	$\overline{C_0}$			1	
1	C_0		1	1	1

(b)

Figure 6.5: Full adder K-maps for (a) Σ_0 (b) C_1.

The full adder circuitry is shown in Figure 6.6.

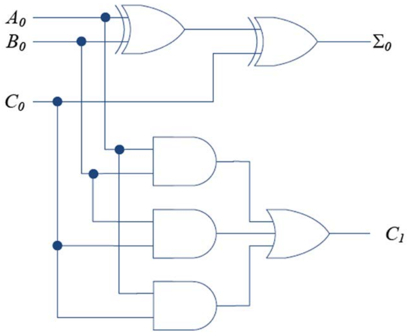


Figure 6.6: Full adder logic circuitry.

It should be obvious that a half-adder can be constructed using a full adder by setting C_0=0. This is illustrated in Figure 6.7.

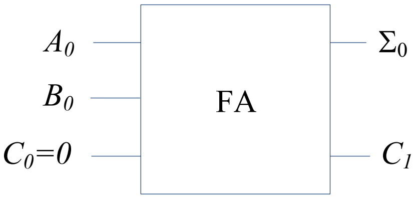


Figure 6.7: Half adder design using full adder.

6.3 Parallel adder

Usually, addition is done on a number of bits using a parallel adder that consists of several full adders as shown in Figure 6.8 for addition of two 3 bit numbers.

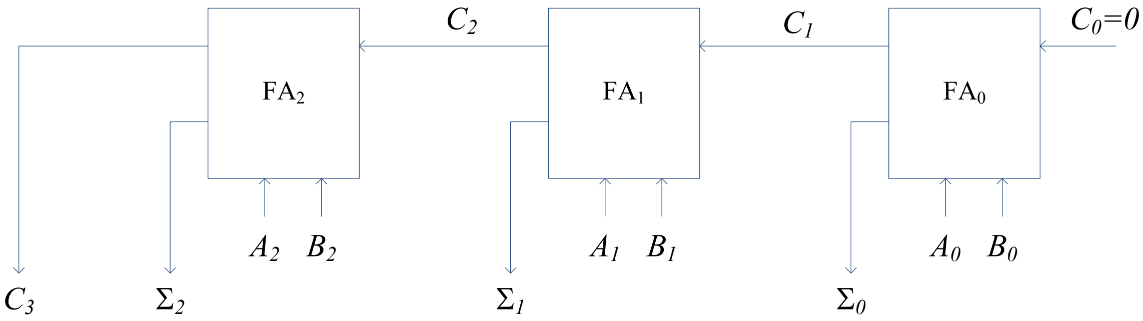


Figure 6.8: Parallel adder layout for addition of two 3 bit numbers.

As an example, consider adding $A = 1\ 1\ 1$ with $B = 1\ 0\ 1$ as depicted in Figure 6.9 to give sum = 1 0 0 and final carry of 1.

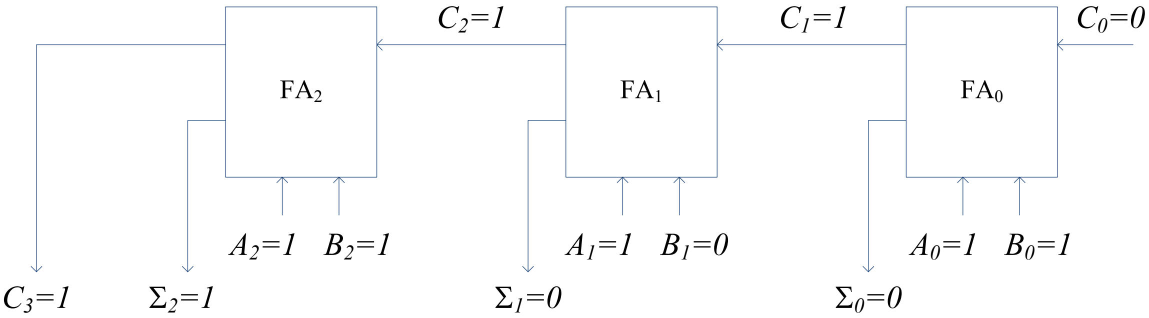


Figure 6.9: Parallel addition example of two 3 bit numbers.

6.4 Parallel addition using integrated circuits

Parallel adders in integrated circuits (IC) form are available such as the four bit TTL 74LS283 as shown in Figure 6.10 (with pin configurations). Such adders can be cascaded to add more bits. For example, two 74LS283 ICs can be used to add two 8 bit numbers as illustrated in Figure 6.11 (pin layout has been modified for ease of understanding, the actual layout is as shown in Figure 6.10). The two numbers: A_0, A_1, A_2, A_3, A_4, A_5, A_7 and B_0, B_1, B_2, B_3, B_4, B_5, B_7 are added together with carry input C_0 to give sum S_0, S_1, S_2, S_3, S_4, S_5, S_7 and carry out C_8. The carry out from the first IC, C_4 is passed as the carry input to the second IC.

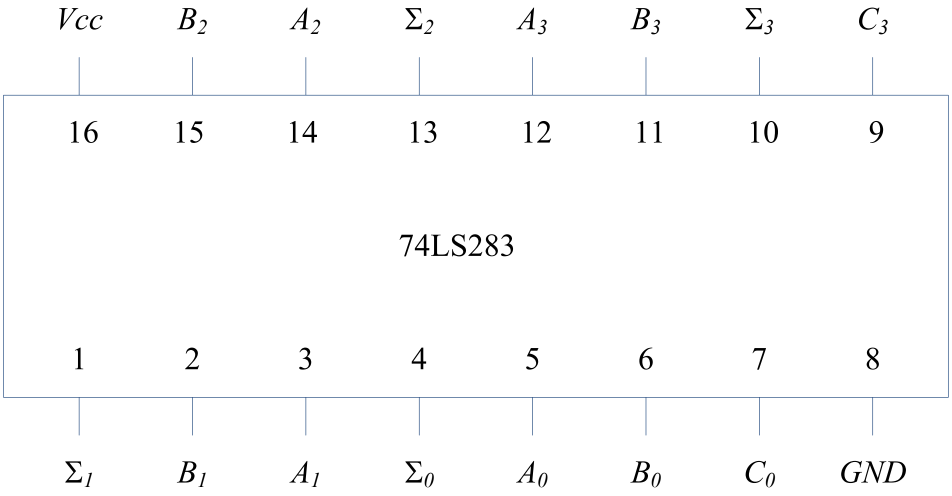


Figure 6.10: Four bit adder IC, 74LS283 showing pin configurations.

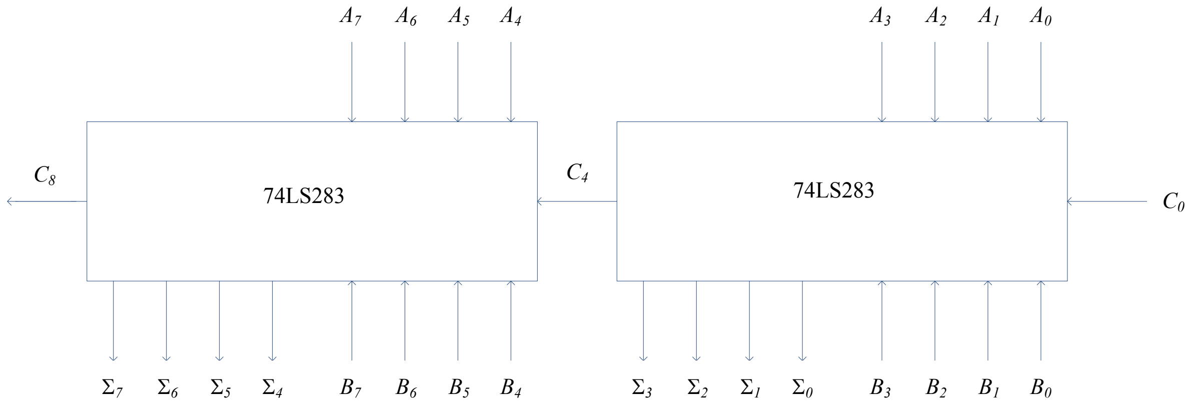


Figure 6.11: Cascading two 74LS283 to add 8 bit numbers.

6.5 Parallel subtraction

Consider a simple subtraction problem: 6 - 4 = 2. In binary, this will be 0110 – 0100 = 0010. Subtraction in binary can be performed through addition by converting the number to be subtracted (i.e. the subtrahend) to 2's complement form and adding to the minued[9].

[9] In the example, 6 – 2 = 4, 6 is the minued and 2 is the subtrahend.

6.5.1 2's complement

A binary number can be converted to 2's complement simply by performing 1's complement (i.e. inverting) each bit and then adding 1 to the inverted bits. Any carry from this operation should be discarded. For example, 2's complement of 4 in binary is

4 in binary → 0100

1's complement of 4 → 1011

2's complement of 4 → 1100

Now, 6 - 4 can be represented in binary as shown in Figure 6.12. The carry is discarded to give the correct answer of 2.

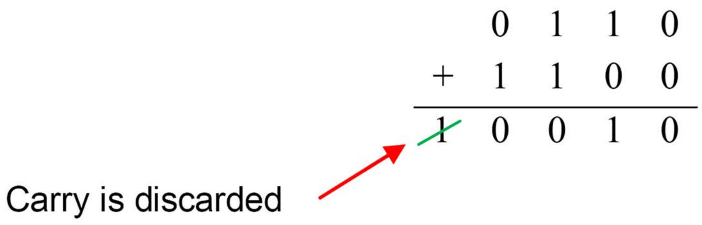


Figure 6.12: Subtracting two numbers using 2's complement method for subtrahend.

It should be obvious that an adder can also function as subtractor with additional gates. For example, the full adder shown in Figure 6.4 can be used to design a subtractor by inverting B_0 and setting C_0=1 (both these actions will result in 2's complement form for B_0) as shown in Figure 6.13. Similar to parallel adders, parallel subtractors can be designed using several full adders as shown in Figure 6.14.

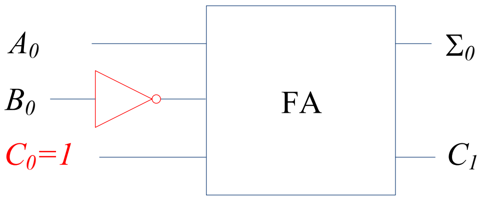


Figure 6.13: A full adder used as subtractor.

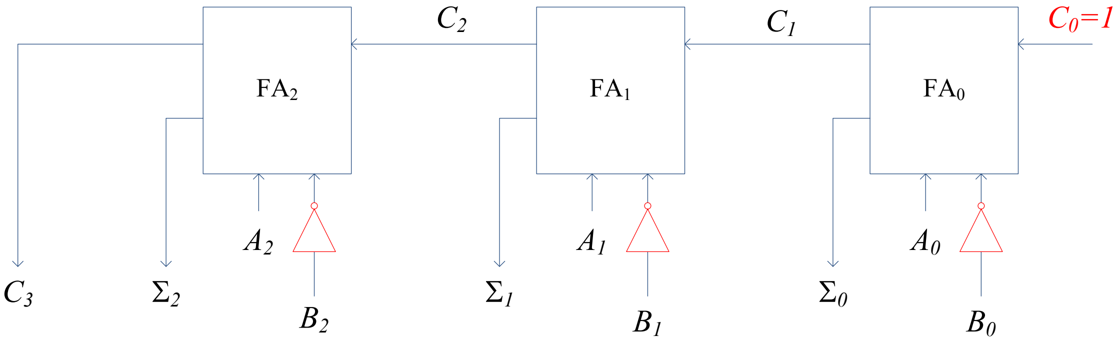


Figure 6.14: Designing a parallel subtractor using several full adders.

Using the example in Figure 6.12, 74LS283 can be modified to act as subtractor as shown in Figure 6.15. The minued is represented by A_0, A_1, A_2, A_3 and the inverters convert the subtrahend (B_0, B_1, B_2, B_3) to 1's complement and C_0 is set to 1 to convert this 1's complement number to 2's complement. The outputs (Σ_0, Σ_1, Σ_2, Σ_3) denote the correct answer as 4 and the carry out, C_4 = 1 is discarded.

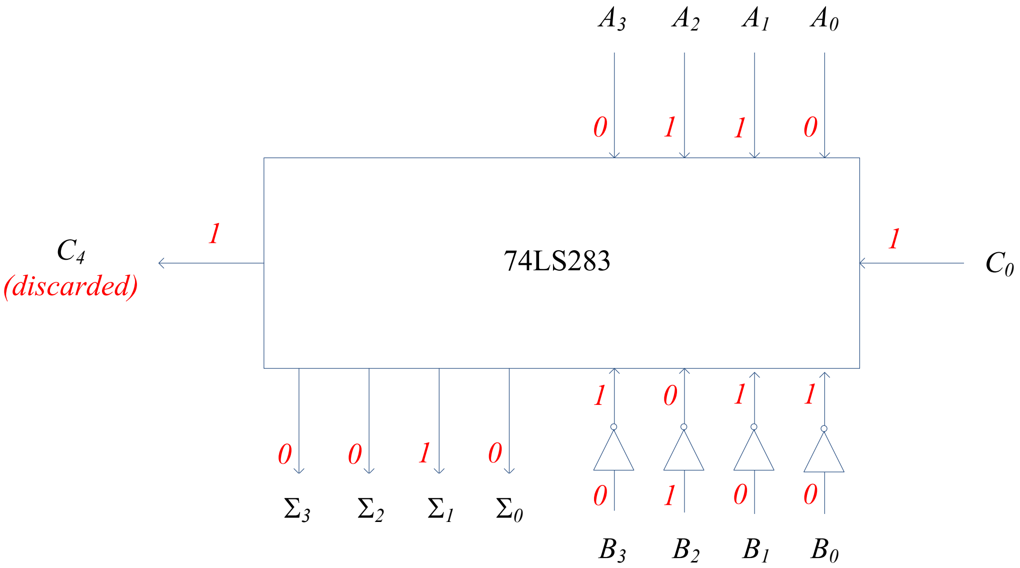


Figure 6.15: Using 74LS283 as subtractor.

6.5.2 Dual adder/subtractor

Replacing the inverters in Figure 6.15 with XOR gates will result in a dual mode adder/subtractor circuit. This is illustrated in Figure 6.16. When the control input is 1, the circuit acts as a subtractor and when the control input is 0, it acts as an adder. For example, when B_0=1 and control input=1 (during subtraction), the output of XOR is 0, i.e. the XOR gate acts as an inverter to give 1's complement and C_0=1 to give 2's complement. When B_0=1 and control input=0 (during addition), C_0=0 and the output of XOR is 1, i.e. the XOR gate acts just as a buffer without changing the logic value.

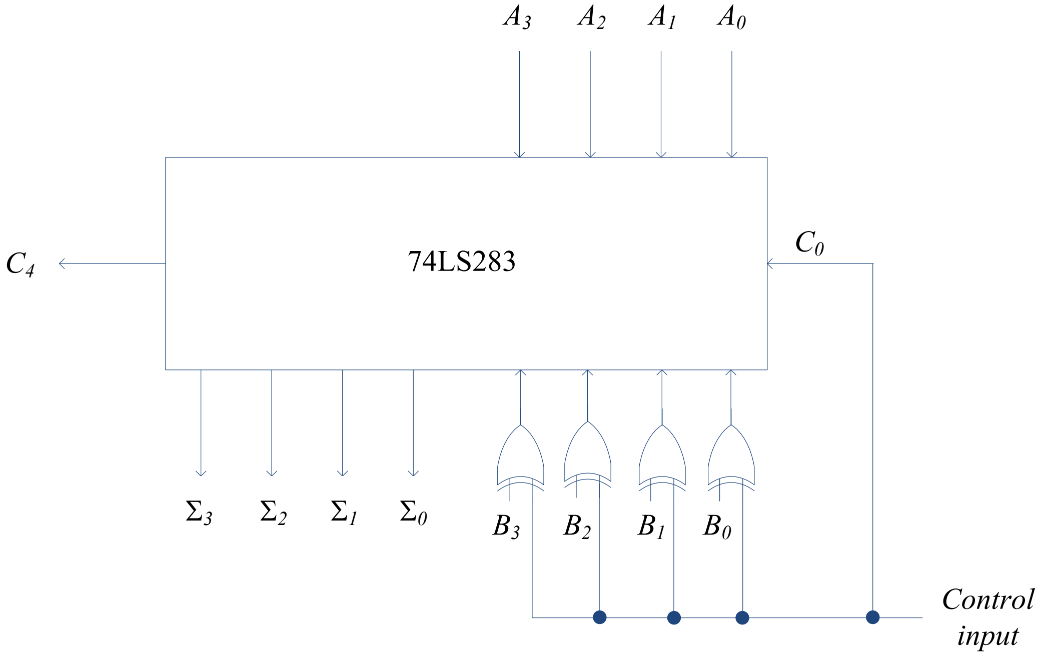


Figure 6.16: Using 74LS283 in dual mode: adder/subtractor.

7 Coders and Multiplexers

In this chapter, we will look two types of operations that are common in digital devices: coding and multiplexing. Coding devices can be categorised as either encoders or decoders and similarly, we have multiplexers and de-multiplexers. Commonly available ICs will be used to illustrate these operations.

7.1 Encoder

An encoder is a device that does some form of coding, for example converting an octal number to binary as shown in Figure 7.1. In general, a *N* bit encoder has 2^N input lines and *N* output lines; in the case of octal to binary encoder, it is 8-to-3, i.e. eight input lines and three output lines. Only one input is active[10] at a time.

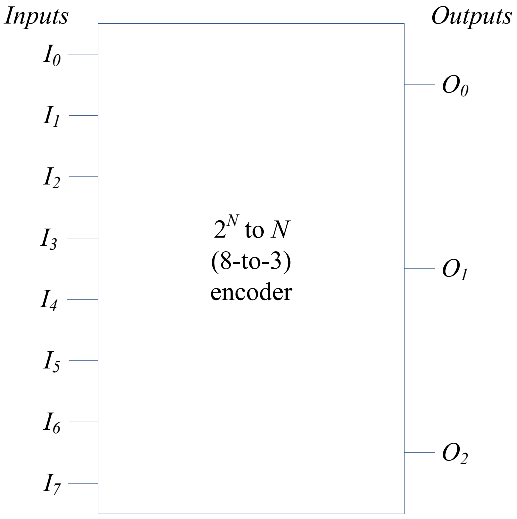
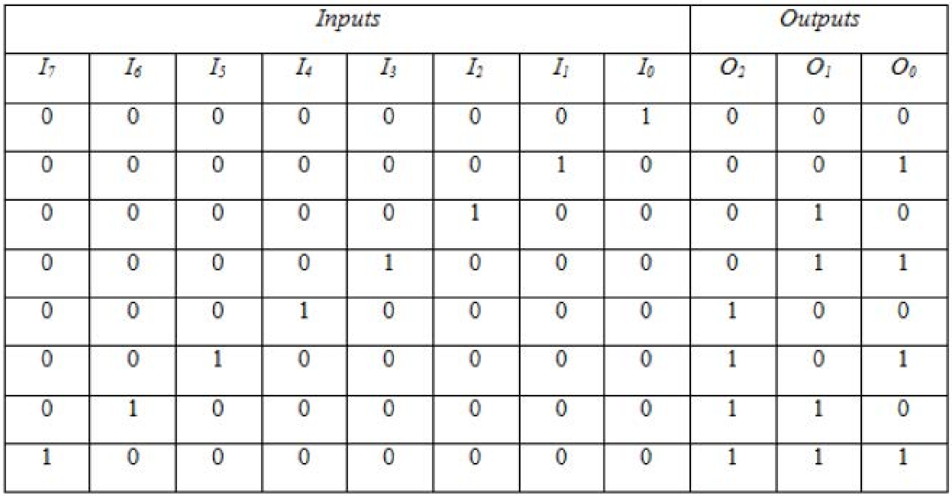


Figure 7.1: A general encoder example: octal to binary.

Table 7.1 gives the truth table for this encoder. It can be seen that only one input line is active. For simplicity of discussion at this stage, we assume that the input and output lines for the decoder are ACTIVE HIGH, though we will see later that most decoders have ACTIVE LOW input and output lines. When one input is activated, the corresponding binary is the output. For example, when $I_6 = 1$, the outputs are $O_2 = 1$, $O_1 = 1$ and $O_0 = 0$, which is binary number for six. Note the ordering of the indexes for the input and output lines in Table 7.1: I_7, I_6,….,I_0 are ordered from left to right while it is O_2,O_1,O_0 for the outputs. This ordering scheme is just chosen to allow easier understanding of the concepts.

[10] Either ACTIVE LOW or ACTIVE HIGH, to be discussed later.

Table 7.1: Truth table for a general 8-to-3 encoder



Inputs								*Outputs*		
I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0	O_2	O_1	O_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

7.1.1 Priority encoding

Though only one input line is supposed to be active at a given time, it is possible to have multiple lines being active perhaps due to noise or error in the logic design. To avoid unpredictable output in such situations, priority encoding can be utilised. Priority encoders allow the higher indexed input lines to take precedence over the lower indexed pins.

Consider a 4-to-2 encoder with the truth table as shown in Table 7.2. Whenever the higher indexed input line is active, the lower indexed lines do not have any effect (irrespective of being active or not). For example, when $I_3 = 1$, the logic values for I_0, I_1 and I_2 do not affect the output (shown by don't care conditions X) and the output will $O_1 = 1$ and $O_0 = 1$.

Table 7.2: Truth table for a general 4-to-2 priority encoder (with don't care conditions)

Inputs				*Outputs*	
I_3	I_2	I_1	I_0	O_1	O_0
0	0	0	1	0	0
0	0	1	X	0	1
0	1	X	X	1	0
1	X	X	X	1	1

The K-maps for outputs O_0 and O_1 are as shown in Figure 7.2. However, the don't care conditions now appear for the inputs, which is different to the don't care conditions for the outputs that was studied in Chapter 4. In order to complete the K-maps, we have to expand Table 7.2 to include both the 0 and 1 cases for the don't care conditions as shown in Table 7.3. From the K-maps, the expressions for the outputs are

$$O_0 = \overline{I_2} I_1 + I_3 ,$$

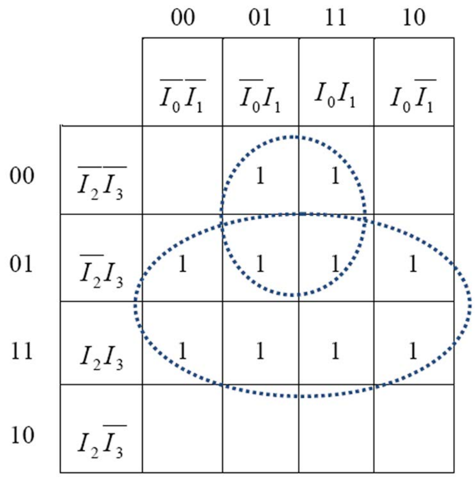
$$O_1 = I_2 + I_3 .$$

From Table 7.3, we can also note that the outputs will be all logic 0 for two cases: when all inputs are 0 and $I_0 = 1$. This ambiguity can be solved by using a special output pin and will be discussed later.

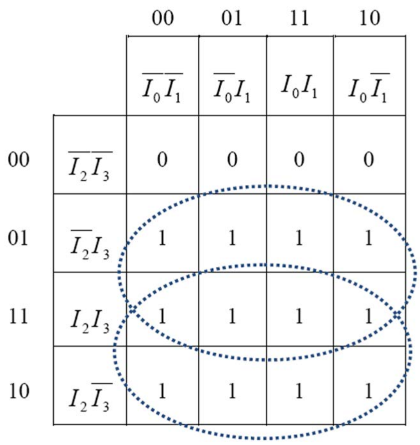
Table 7.3: Truth table for a general 4-to-2 priority encoder (showing full don't care cases)

Inputs				*Outputs*	
I_3	I_2	I_1	I_0	O_1	O_0
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	1
0	0	1	1	0	1
0	1	0	0	1	0
0	1	0	1	1	0
0	1	1	0	1	0
0	1	1	1	1	0
1	0	0	0	1	1
1	0	0	1	1	1
1	0	1	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1
1	1	0	1	1	1
1	1	1	0	1	1
1	1	1	1	1	1

Figure 7.3 shows the logic circuits for the outputs O_0 and O_1.



(a)



		00	01	11	10
		$\overline{I_0}\,\overline{I_1}$	$\overline{I_0}I_1$	I_0I_1	$I_0\overline{I_1}$
00	$\overline{I_2}\,\overline{I_3}$	0	0	0	0
01	$\overline{I_2}I_3$	1	1	1	1
11	I_2I_3	1	1	1	1
10	$I_2\overline{I_3}$	1	1	1	1

(b)

Figure 7.2: K-maps for 4-to-2 priority encoder: (a) O_0 (b) O_1.

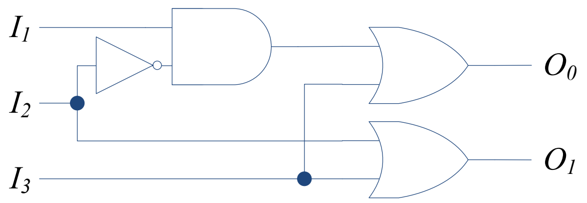


Figure 7.3: Logic circuit for 4-to-2 priority encoder.

7.1.2 Enable inputs

Figure 7.4 shows a 8-to-3 encoder IC, 74xx148 with pin configurations[11]. Both the inputs and outputs are ACTIVE LOW, i.e. enabled/activated on logic 0. It is also a priority encoder, so the higher indexed inputs take priority.

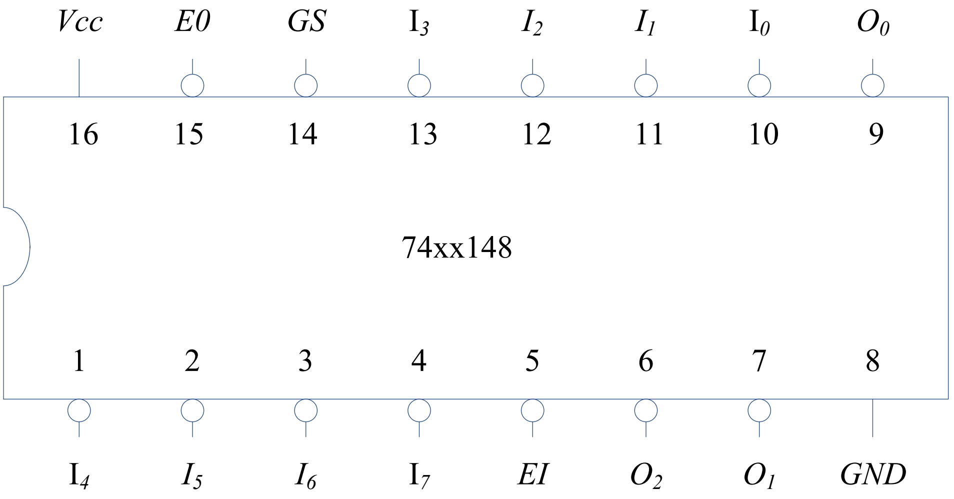


Figure 7.4: 8-to-3 encoder IC, 74xx148 pin configuration.

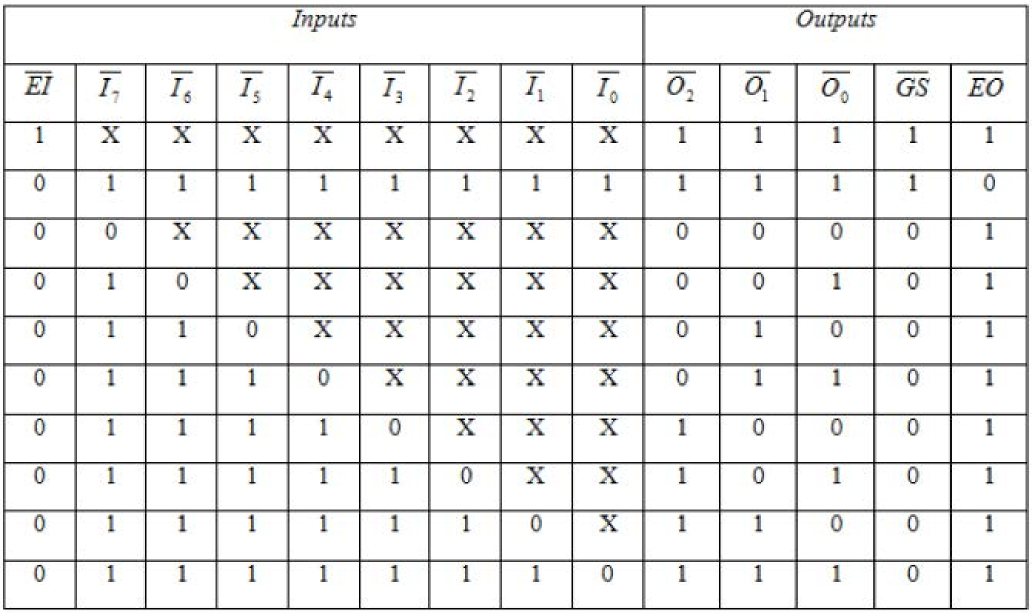
Table 7.4 shows the truth table and it can be seen that there are additional input and outputs: Enable Input ($\overline{EI}$), Enable Output ($\overline{EO}$) and Group Select ($\overline{GS}$). All the enable pins are also ACTIVE LOW as shown by the overbars in Table 7.4 and by the presence of bubbles in Figure 7.4. $\overline{EI}$ enables the device and allows the input values to change the outputs. As shown in Table 7.4, when $\overline{EI} = 1$, the outputs are all inactive (i.e. logic 1). When $\overline{EI} = 0$, the inputs are enabled and affects the outputs. For example, when $\overline{I_7} = 0$, the outputs are $\overline{O_2} = \overline{O_1} = \overline{O_0} = 0$. Similarly when $\overline{I_3} = 0$, the outputs are $\overline{O_2} = 1$ and $\overline{O_1} = \overline{O_0} = 0$ (note that the outputs are active low, so it represents 3 in binary).

[11] xx denotes different types of ICs available such as low power Schottky version, 74LS148 and high speed CMOS version, 74HC148.

When all the inputs are inactive, $\overline{GS}$ is disabled ($\overline{GS}=1$)and when any one input is active, then $\overline{GS}=0$. Hence $\overline{GS}$ is useful to indicate whether the condition $\overline{O_2}=\overline{O_1}=\overline{O_0}=1$ is caused by $\overline{I_0}=0$ or if all inputs are inactive.

$\overline{EO}$ is used when cascading several encoders to form a larger priority encoding system. For this purpose, $\overline{EO}$ output is connected $\overline{EI}$ input of the lower priority encoder.

Table 7.4: Truth table for 74xx148



Inputs									*Outputs*				
$\overline{EI}$	$\overline{I_7}$	$\overline{I_6}$	$\overline{I_5}$	$\overline{I_4}$	$\overline{I_3}$	$\overline{I_2}$	$\overline{I_1}$	$\overline{I_0}$	$\overline{O_2}$	$\overline{O_1}$	$\overline{O_0}$	$\overline{GS}$	$\overline{EO}$
1	X	X	X	X	X	X	X	X	1	1	1	1	1
0	1	1	1	1	1	1	1	1	1	1	1	1	0
0	0	X	X	X	X	X	X	X	0	0	0	0	1
0	1	0	X	X	X	X	X	X	0	0	1	0	1
0	1	1	0	X	X	X	X	X	0	1	0	0	1
0	1	1	1	0	X	X	X	X	0	1	1	0	1
0	1	1	1	1	0	X	X	X	1	0	0	0	1
0	1	1	1	1	1	0	X	X	1	0	1	0	1
0	1	1	1	1	1	1	0	X	1	1	0	0	1
0	1	1	1	1	1	1	1	0	1	1	1	0	1

7.2 Decoder

Decoder is the opposite of encoder, for example a 3-to-8 decoder that accepts three binary inputs and activates the corresponding single output as shown in Figure 7.5. Figure 7.6 shows a 74xx138 IC, which is binary-to-octal (3-to-8) decoder. The three inputs are active HIGH (note that there is no bubble in the figure) but the eight outputs are all ACTIVE LOW. In addition, three enable inputs: two ACTIVE LOW and one ACTIVE HIGH need to be in the asserted mode to enable the IC (i.e. $E_3=1$, $\overline{E_2}=0$ and $\overline{E_1}=0$). If any of these inputs are in an inactive state, then all the outputs will be in inactive state (i.e. logic 1 since these are ACTIVE LOW pins) irrespective of the inputs as shown by the first three rows in Table 7.5. When E_3, $\overline{E_2}$ and $\overline{E_1}$ are enabled, the inputs affect the output. For example when $I_2=I_1=I_0=1$ then pin $\overline{O_7}$ becomes low and when $I_2=I_1=1$ and $I_0=0$ then pin $\overline{O_6}$ becomes low.

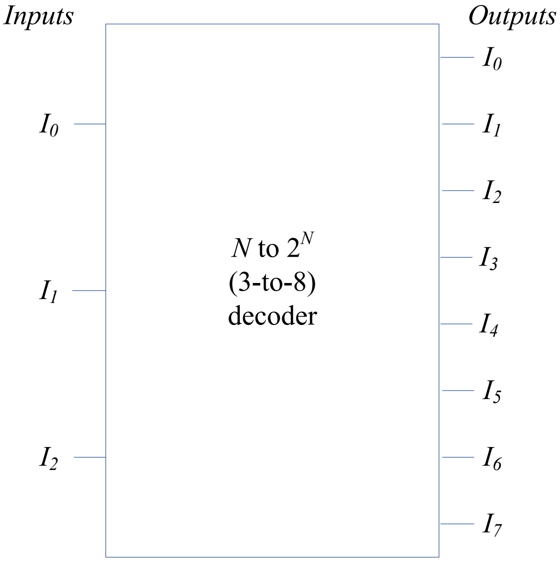


Figure 7.5: 3-to-8 decoder example.

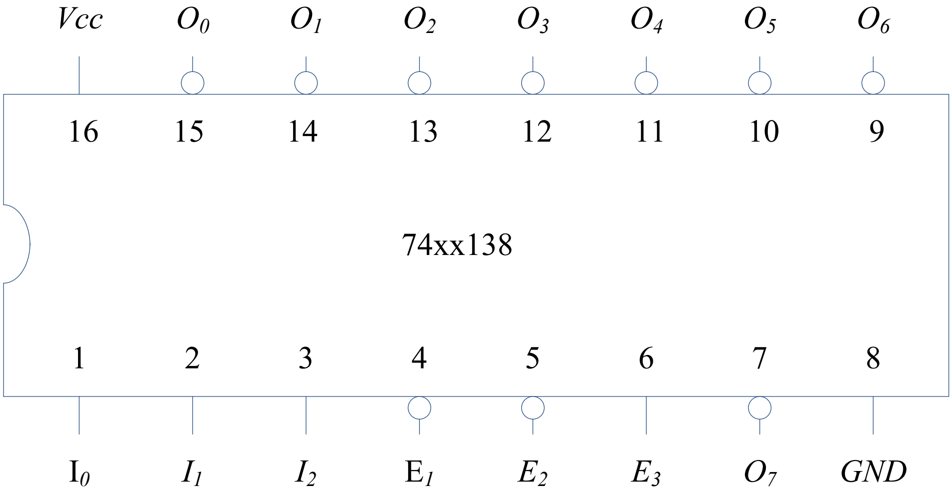
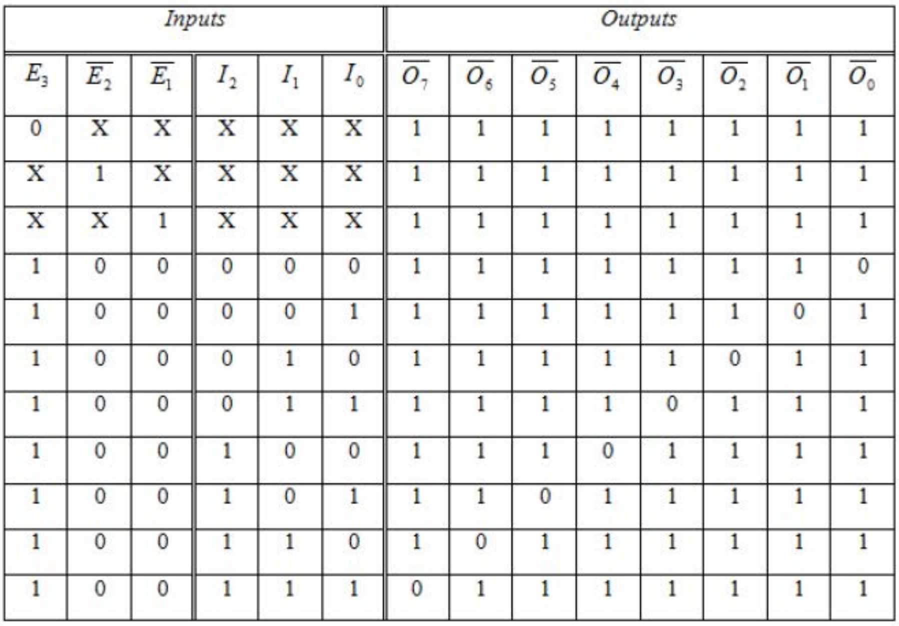


Figure 7.6: 3-to-8 encoder IC, 74xx138 pin configuration.

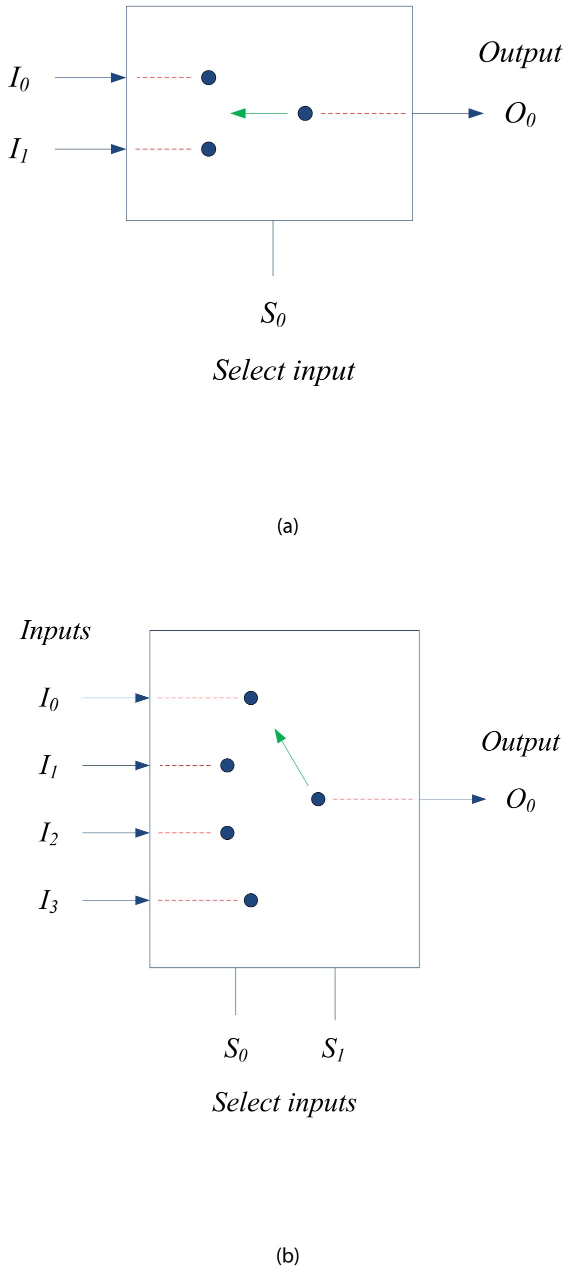
Table 7.5: Truth table for 74xx138



Inputs						Outputs							
E_3	$\overline{E_2}$	$\overline{E_1}$	I_2	I_1	I_0	$\overline{O_7}$	$\overline{O_6}$	$\overline{O_5}$	$\overline{O_4}$	$\overline{O_3}$	$\overline{O_2}$	$\overline{O_1}$	$\overline{O_0}$
0	X	X	X	X	X	1	1	1	1	1	1	1	1
X	1	X	X	X	X	1	1	1	1	1	1	1	1
X	X	1	X	X	X	1	1	1	1	1	1	1	1
1	0	0	0	0	0	1	1	1	1	1	1	1	0
1	0	0	0	0	1	1	1	1	1	1	1	0	1
1	0	0	0	1	0	1	1	1	1	1	0	1	1
1	0	0	0	1	1	1	1	1	1	0	1	1	1
1	0	0	1	0	0	1	1	1	0	1	1	1	1
1	0	0	1	0	1	1	1	0	1	1	1	1	1
1	0	0	1	1	0	1	0	1	1	1	1	1	1
1	0	0	1	1	1	0	1	1	1	1	1	1	1

7.3 Multiplexer

Multiplexer (also known as data selector) is a digital device that acts like a switch taking several inputs and connecting a selected input to the output at a time. Simple two input and four input multiplexers are shown in Figure 7.7. For the two input multiplexer, there are two inputs: I_1 and I_0 with one output, O_0. The selector input, S_0 will decide the route from input to the output. For example, when $S_0 = 1$, I_1 is selected and data at I_1 is routed to output O_0. For the four input multiplexer, there are four inputs: I_3, I_2, I_1 and I_0 with one output, O_0. The selector inputs, S_1 and S_0 will decide which connection is made from the input to the output. For example, when $S_1 = S_0 = 1$, I_3 is selected and data at I_3 is routed to output O_0. Tables 7.6 and 7.7 show the truth table for these multiplexers.



(a)

(b)

Figure 7.7: Simple multiplexers (a) two inputs (b) four inputs.

Table 7.6: Truth table for two input multiplexer

Select input	*Output*
S_0	
0	$O_0 = I_0$
1	$O_0 = I_1$

Table 7.7: Truth table for four input multiplexer

Select inputs		*Output*
S_1	S_0	
0	0	$O_0 = I_0$
0	1	$O_0 = I_1$
1	0	$O_0 = I_2$
1	1	$O_0 = I_3$

To obtain the logic circuit diagram for two input multiplexer, truth table as in Table 7.8 should be constructed. With this, K-map for output O_0 can be obtained as shown in Figure 7.8. Two pair loops can be drawn to give the output as

$$O_0 = \overline{S_0} I_0 + S_0 I_1 .$$

The logic circuit diagram is given in Figure 7.9. Similar approach could be utilised to obtain logic circuit diagrams for higher input multiplexers.

Table 7.8: Full truth table for two input multiplexer

S_0	I_0	I_1	O_0
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

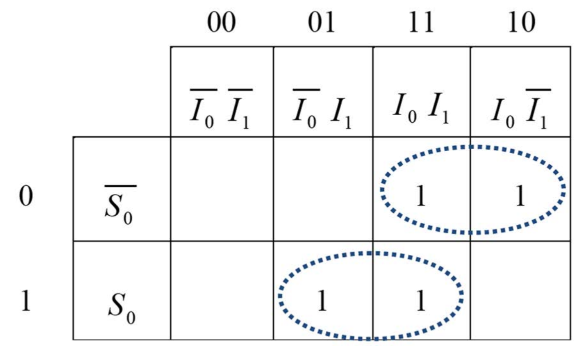


Figure 7.8: K-map for two input multiplexer.

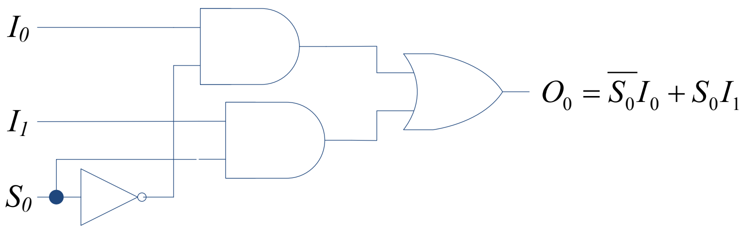


Figure 7.9: Logic circuit diagram for two input multiplexer.

7.3.1 Multiplexer IC example

A quadruple 2-line to 1-line multiplexer (IC 74xx157) is shown in Figure 7.10. The IC contains two sets of four inputs (I_{0a}, I_{1a}, I_{2a}, I_{3a} and I_{0b}, I_{1b}, I_{2b}, I_{3b}) that can be routed to the four outputs (O_0, O_1, O_2, O_3) depending on the select input, S_0. The enable, $\overline{E}$ input must asserted, i.e. it must be logic 0 for the IC to be enabled. Table 7.9 gives the truth table for this IC. When S_0 = 0, the outputs follow I_{0a}, I_{1a}, I_{2a}, I_{3a} inputs and when S_0 = 1, the outputs follow I_{0b}, I_{1b}, I_{2b}, I_{3b} inputs.

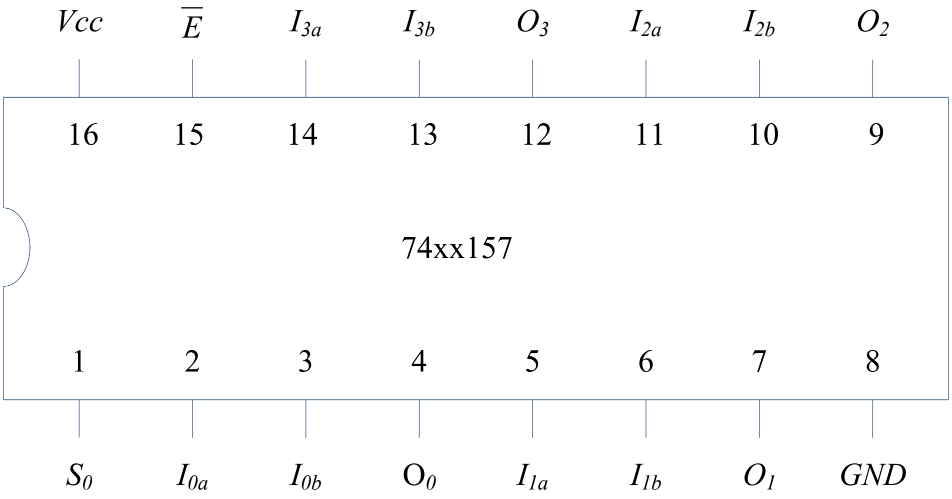


Figure 7.10: Quadruple 2-line to 1-line multiplexer IC, 74xx157 pin configuration.

Table 7.9: Truth table for 74xx157

$\overline{E}$	S_0	O_0	O_1	O_2	O_3
1	X	0	0	0	0
0	0	I_{0a}	I_{1a}	I_{2a}	I_{3a}
0	1	I_{0b}	I_{1b}	I_{2b}	I_{3b}

7.4 De-multiplexer

A de-multiplexer does the opposite of multiplexer in that it takes a single input and distributes it to a selected output. Hence it is also known as data distributor. An example of 1-line to 8-line demultiplexer is shown in Figure 7.11.

IC 74xx138, which is a 3-to-8 decoder (that we discussed earlier) can also be used as a 1-line to 8-line demultiplexer by using $\overline{E_1}$ as data input and the three inputs as selectors. The other two enable pins are asserted to enable the IC by connecting $\overline{E_2}$ to ground (i.e. logic 0) and E_3 is connected to *Vcc* (+5 V) to give logic 1. Using this convention, the pin configuration is as shown in Figure 7.12 and the truth table as in Table 7.10. The select inputs (S_0, S_1, S_2) will select the particular output pin and the input data (I_0) will be distributed to this selected output pin. Due to the dual mode nature of such ICs, these are usually known as *decoder/demultiplexer* ICs.

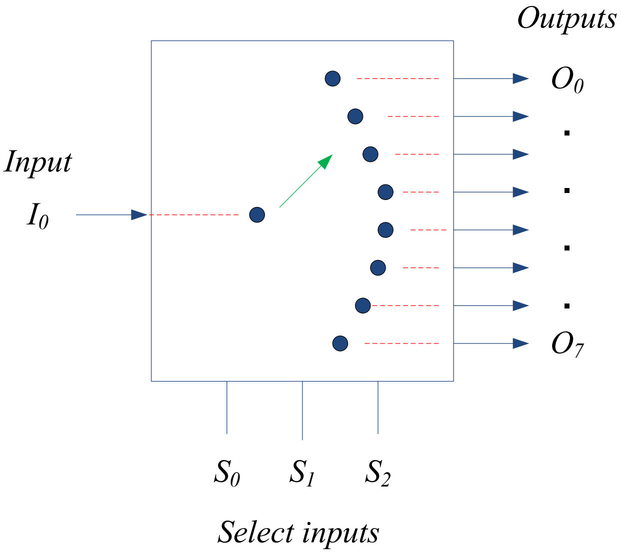


Figure 7.11: 1-line to 8-line demultiplexer.

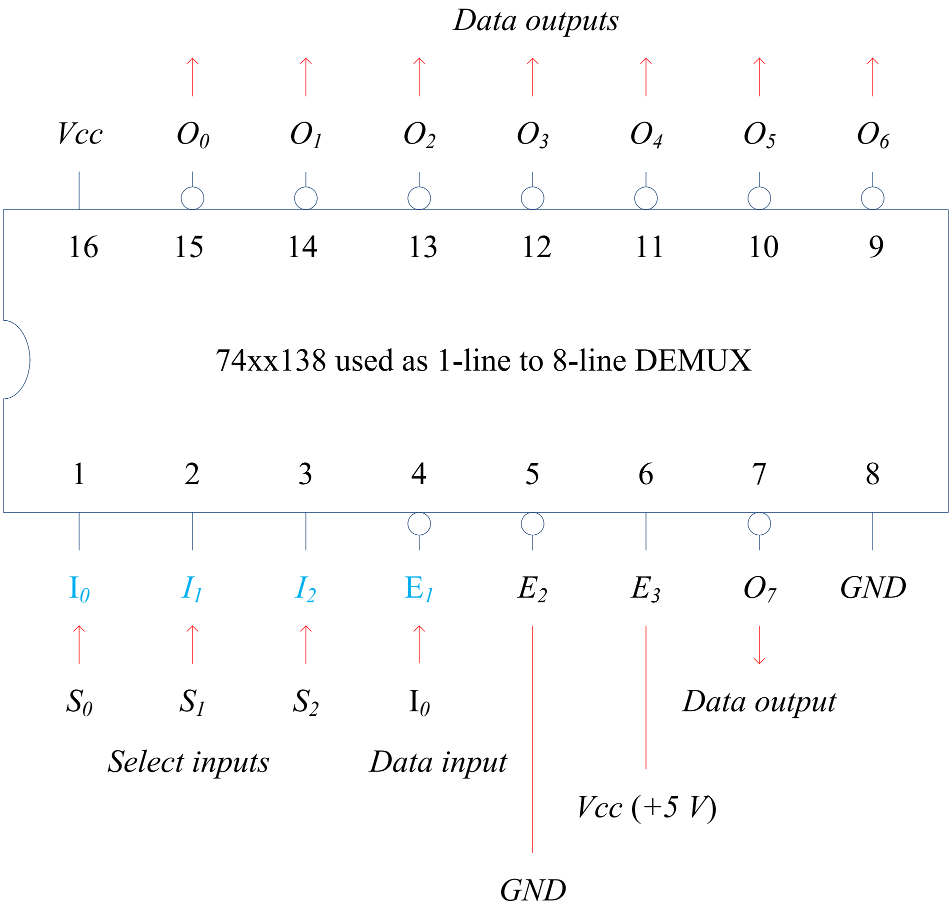
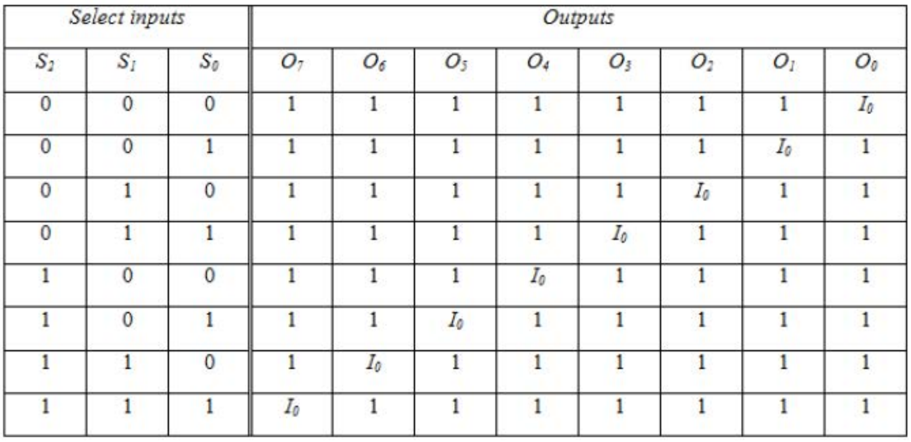


Figure 7.12: 1-line to 8-line demultiplexer using 74xx138 decoder.

Table 7.10: Truth table for 1-line to 8-line demultiplexer (using 74xx138 IC)



Select inputs			*Outputs*							
S_2	S_1	S_0	O_7	O_6	O_5	O_4	O_3	O_2	O_1	O_0
0	0	0	1	1	1	1	1	1	1	I_0
0	0	1	1	1	1	1	1	1	I_0	1
0	1	0	1	1	1	1	1	I_0	1	1
0	1	1	1	1	1	1	I_0	1	1	1
1	0	0	1	1	1	I_0	1	1	1	1
1	0	1	1	1	I_0	1	1	1	1	1
1	1	0	1	I_0	1	1	1	1	1	1
1	1	1	I_0	1	1	1	1	1	1	1

It should be remembered that 74xx138 IC has outputs that are ACTIVE LOW, hence inactive outputs have logic 1 as shown in Table 7.10.

8 Counters

In this chapter, we will look at using flip flops and logic gates to design counters. There are two types of counters: asynchronous and synchronous. Asynchronous counters are also known as ripple counters as the clock pulse ripples from one flip-flop to the next. Incorrect counter output can result if the accumulative ripple delay is longer than the clock pulse. Synchronous counters, on the other hand, have clock pulse input to each flip-flop and hence do not suffer from this ripple effect. However, these counters often require additional circuitry.

8.1 Asynchronous up-counter

Figure 8.1 shows an example of a two bit asynchronous up-counter. J-K flip flops are used here although any flip-flop could be used. Two flip-flops are required in this instance as it is a two bit counter. Figure 8.2 shows the state diagram and state table, i.e. the sequence of the counter output. As there are two bits, the counter cycles through four states[12]: 00, 01, 10, 11 and it is known as up-counter since it counts in increasing order.

As can be seen from the figure, all *J* and *K* inputs are tied to logic level 1. This ensures that all the flip-flops operate in toggle mode only. The output from flip-flop 1, Q_1 is the LSB, while the output from flip-flop 2, Q_2 is MSB. The output Q_1 also acts as the input clock pulse for flip-flop 2.

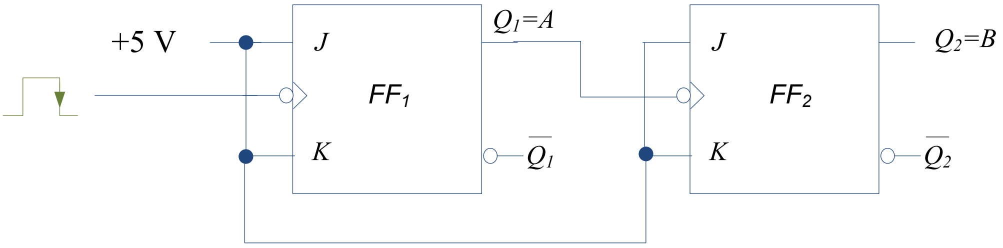


Figure 8.1: Two bit asynchronous up-counter with NGT clock pulse.

00
01
10
11

Figure 8.2: State diagram and table for two bit asynchronous up-counter.

[12] *N* flip flop give 2^N states, sometimes *N* number of flip flop counter is known as modulo *N* counter.

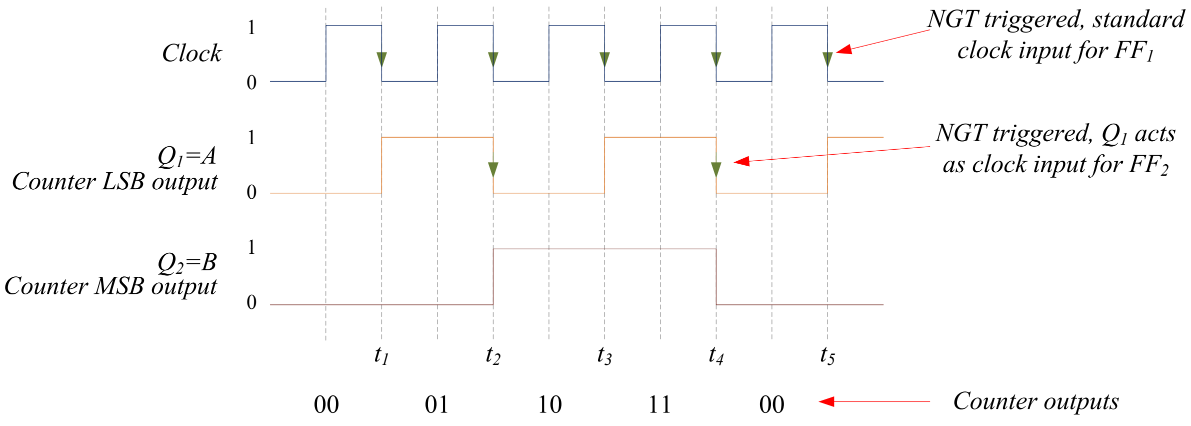


Figure 8.3: Timing diagram for two bit asynchronous up-counter with NGT clock pulse.

Analysing the timing diagram shown in Figure 8.3: at time t_1, NGT clock pulse triggers Q_1 to toggle from logic 0 to logic 1. At time t_2, NGT clock pulse causes Q_1 to change state to logic 0. As output from flip-flop 1 acts as clock input for flip-flop 2, at time t_2, Q_2 toggles to logic level 1. At time t_3, the NGT clock input toggles Q_1 to logic level 1 but there is no change in Q_2 since the clock input to flip-flop 2 at this time is PGT and not NGT. At time t_4, both Q_1 and Q_2 toggles to logic 0. It can be seen that the counter cycles through states 00→01→10→11 and the cycle is repeated.

8.1.1 Asynchronous up-counter – PGT clocked flip-flops

Figure 8.4 shows a two bit asynchronous up-counter but with the clock triggering edge to be positive going. The figure is nearly the same as Figure 8.1 except that the clock input for flip-flop 2 comes from $\overline{Q}_1$ rather than Q_1. The state diagram and state table will be the same as shown in Figure 8.2. The timing diagram is shown in Figure 8.5 where it can be seen that the flip-flop changes at PGT clock edges.

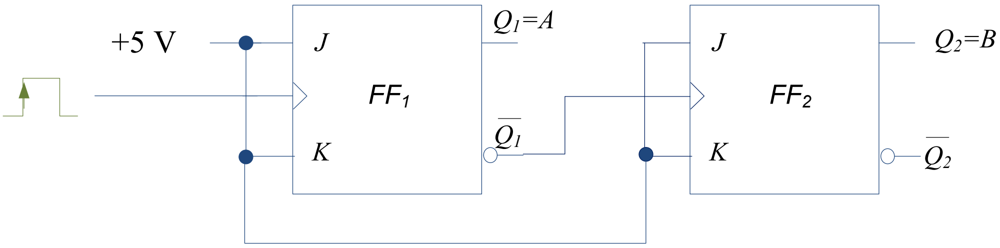


Figure 8.4: Two bit asynchronous up-counter with PGT clock pulse.

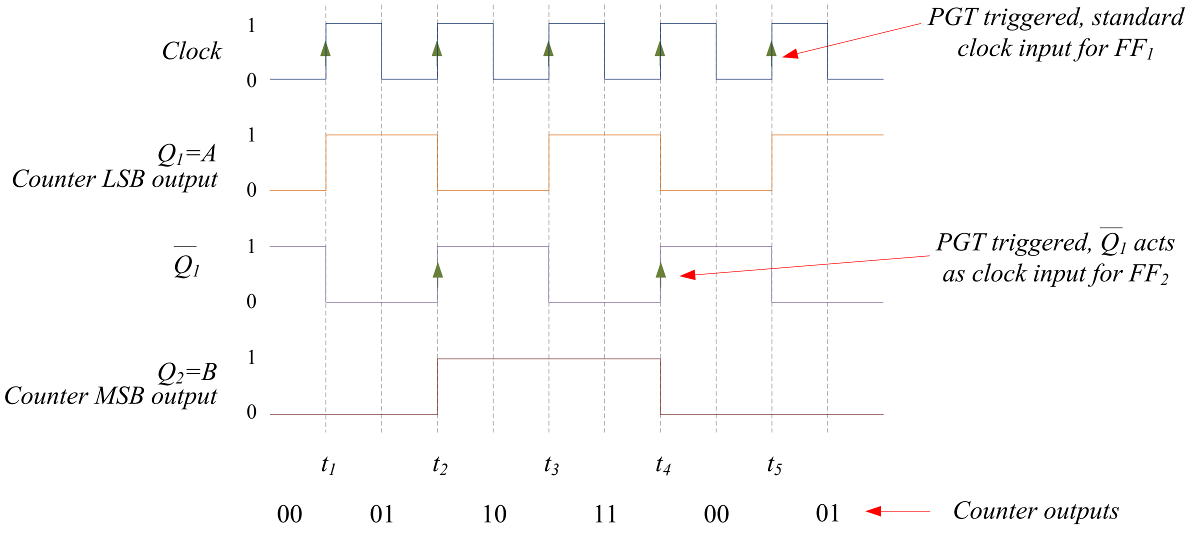


Figure 8.5: Timing diagram for two bit asynchronous up-counter with PGT clock pulse.

The timing diagram in Figure 8.5 can be analysed at follows: at time t_1, PGT clock pulse triggers Q_1 to toggle from logic 0 to logic 1. There is no change in Q_2 and the counter output is 01. At time t_2, PGT clock pulse causes Q_1 to change state to logic 0. As output from $\overline{Q}$ of flip-flop 1 acts as clock input for flip-flop 2, at time t_2, Q_2 toggles to logic level 1 and the counter output is now 10. At time t_3, the PGT clock input toggles Q_1 to logic level 1 but there is no change in Q_2 since the clock input to flip-flop 2 at this time is NGT and not PGT giving counter output of 11. At time t_4, both Q_1 and Q_2 toggles to logic 0 giving counter output of 00. At t_5, Q_1 toggles to logic 1 but there is no change in Q_2. It can be seen that the counter cycles through states 00→01→10→11 and the cycle is repeated.

8.2 Asynchronous down-counter

Figure 8.6 shows an example of a two bit asynchronous down-counter using T flip flops triggered with NGT clock pulse. The clock for the second flip-flop comes from Q_1 (similar to up-counter using PGT as shown in Figure 8.4). Figure 8.7 shows the state diagram and state table, i.e. the sequence of the counter output. The counter cycles through four states: 11à10à01à00 i.e. in decreasing order as it is a down-counter (as shown in Figure 8.8).

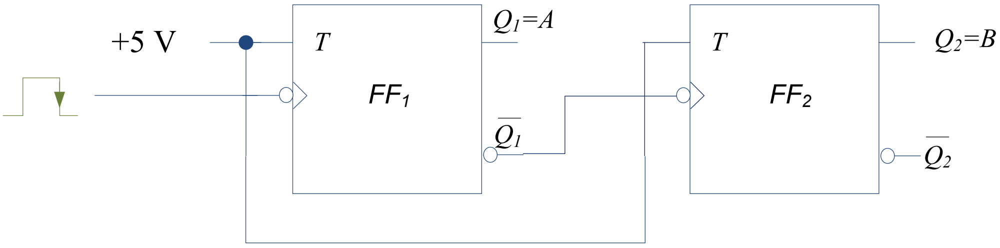
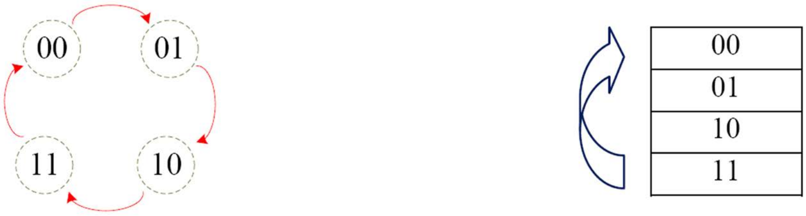


Figure 8.6: Two bit asynchronous down-counter with NGT clock pulse using T flip-flop.



00
01
10
11

Figure 8.7: State diagram and table for two bit asynchronous down-counter.

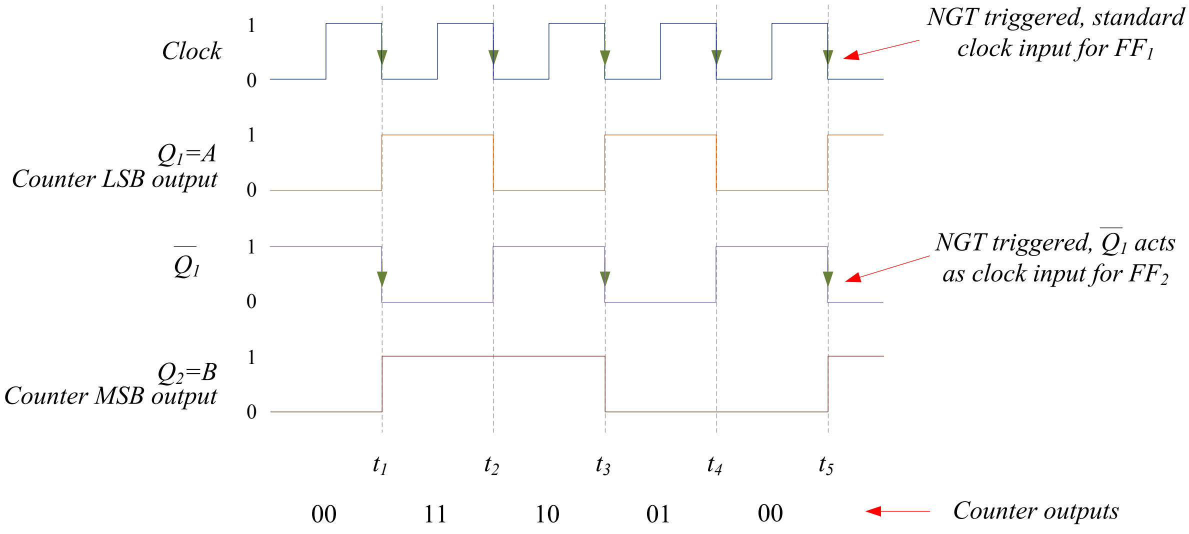


Figure 8.8: Timing diagram for two bit asynchronous down-counter with NGT clock pulse.

At t_1, the NGT clock pulse toggles the flip-flop to logic level 1. As $\overline{Q_1}$ is now the clock input for the second flip-flop, at time t_1, flip-flop 2 output toggles to logic level 1. The counter output is now 11. At time t_2, flip-flop 1 toggles to logic level 0 while there is no change in flip-flop 2 as the clock input for the second flip-flop at this time is PGT. The output is now 10. At time t_3, both flip-flop receive NGT clock inputs and toggles to opposing states as previously giving output as 01. At time t_4, flip-flop 1 toggles to logic level 0 giving counter output as 00. Hence, the counter cycles through 11→10→01→00.

Similarly, a counter with higher number of bits can be constructed. For example, a four bit asynchronous down-counter with PGT clock pulse using J-K flip flops is shown in Figure 8.9. The clock inputs (except for the first flip-flop) come from Q output of the previous flip-flop. The counter will cycle through 1111→1110→1101→1100→1011→1010→1001 →1000→0111→0110→ 0101→0100→ 0011→0010→0001→0000.

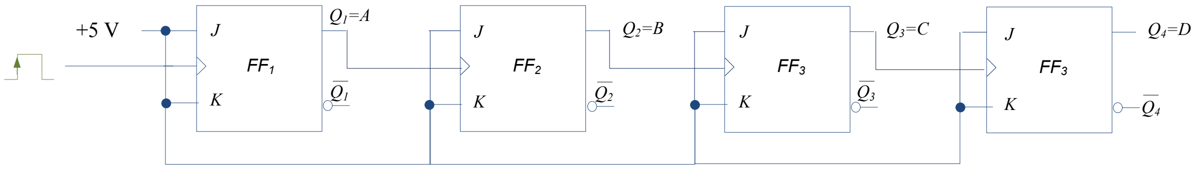


Figure 8.9: Four bit asynchronous down-counter with PGT clock pulse.

Table 8.1 gives a summary of the clock inputs for the second flip-flop onwards against the up/down counter and trigger edge types.

Table 8.1: Clock inputs for the second flip-flop onwards against the up/down counter and trigger edge types

	Clock input
Up-counter NGT	Q
Up-counter PGT	$\overline{Q}$
Down-counter NGT	$\overline{Q}$
Down-counter PGT	Q

8.3 Asynchronous counters with incomplete cycles

So far, we have seen counters that complete the cycle for the specific number of bits, for example a two bit up-counter would have four states: 00→01→10→11 and a three bit down counter would have eight states: 111→110→101→100→011→010→001→000. Consider a case where we need a counter only to count from 00→01→10. Two flip-flops will be required but the counter has to reset to 00 after 10 and not after 11. Therefore additional circuitry will be required to reset the counter after 10. The state diagram is shown in Figure 8.10 where the temporary state of 11 will only occur for a very short period of time and hence will not appear in the counter cycle. For this purpose, the clear asynchronous input of the flip-flops together with a NAND gate could be used to reset both flip-flops instantaneously[13] . This situation is shown in Figure 8.11. As soon as the state Q_2=1 (i.e. B=1) and Q_1=1 (i.e. A=1) occur, the clear inputs reset all the flip-flips to 0 and the counter then resumes its cycle. Figure 8.12 shows the timing diagram for this counter.

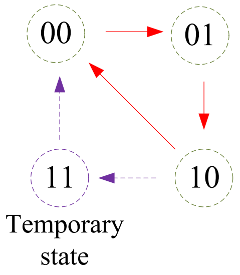


Figure 8.10: Three state up-counter showing a temporary state.

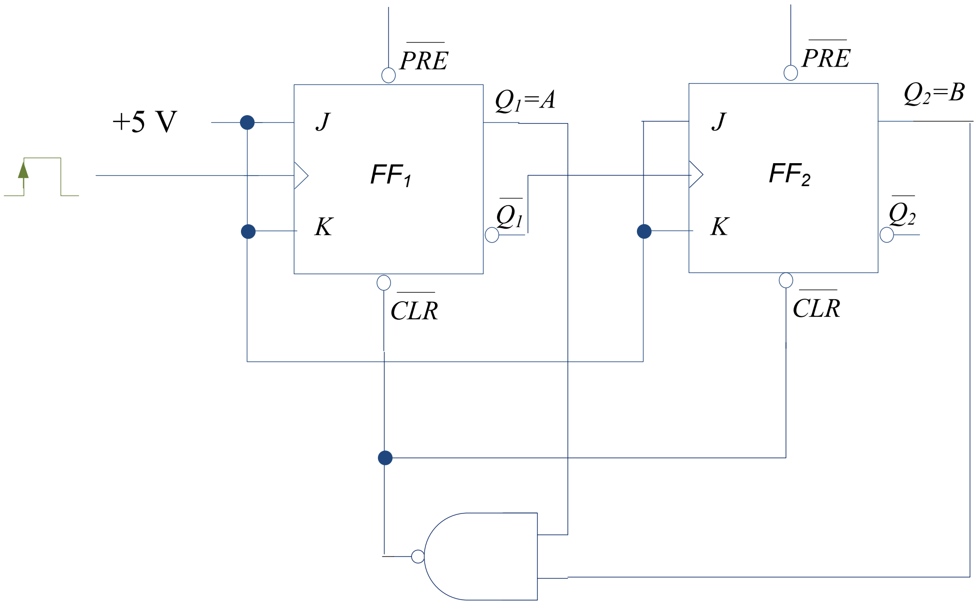


Figure 8.11: Three state asynchronous up-counter with PGT clock pulse (with $\overline{CLR}$ input).

[13] Clear input does not depend on clock edge and hence the change is immediate.

At time t_1, the PGT edge of the clock toggles the first flip-flop output to 1 (i.e. Q_1=1). There is no change in Q_2=0. At time t_2, Q_1 toggles to 0 and Q_2 toggles to 1. At time t_3, Q_2 remains at 1 and Q_1 toggles to 1. However, at this time point, the output of the NAND gate is logic level 0 and hence activates the active low $\overline{CLR}$ inputs, which reset both flip-flops to logic level 0. The counter then resumes its count. The effect of $\overline{CLR}$ is instantaneous and though the timing diagram shows a glitch during time t_3, it occurs only for a very short period of time and does not appear as part of the counter output.

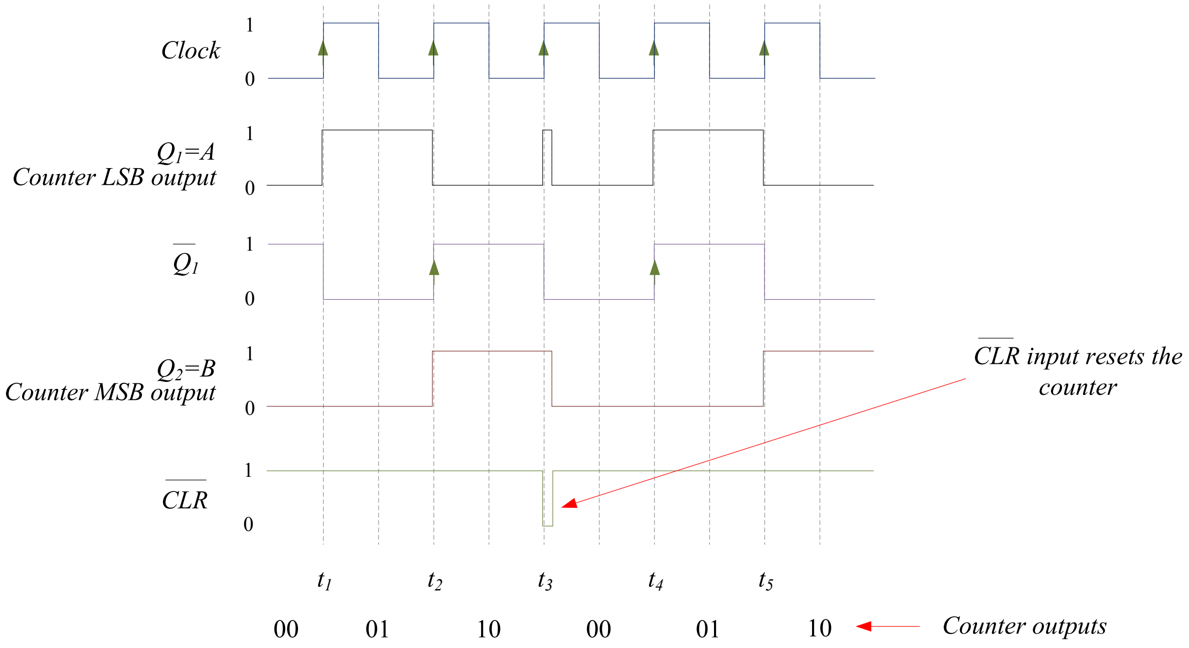


Figure 8.12: Timing diagram for three state asynchronous up-counter with PGT clock pulse.

Let us consider another example: a counter to count 000→001→010→011→100 only. In this situation, we will need three flip-flops and the counter has to stop the cycle at 100 (and skip 101, 110 and 111) and return to 000. In other words, the counter has to reset after 100. The state diagram is shown in Figure 8.13. As mentioned earlier, the temporary state of 101 occurs only for a very short period of time and hence will not appear in the counter cycle. The additional circuitry using NAND gate and $\overline{CLR}$ inputs reset the counter to 000 when the state 101 occurs. The logic circuit is shown in Figure 8.14. As soon as the state Q_3=1 (i.e. C=1) and Q_1=1 (i.e. A=1) occur, the clear inputs reset all the flip-flips to 0 and the counter then resumes its cycle.

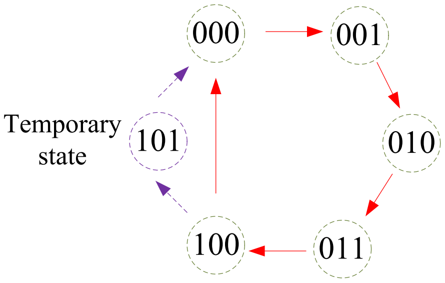


Figure 8.13: Five state up-counter showing a temporary state.

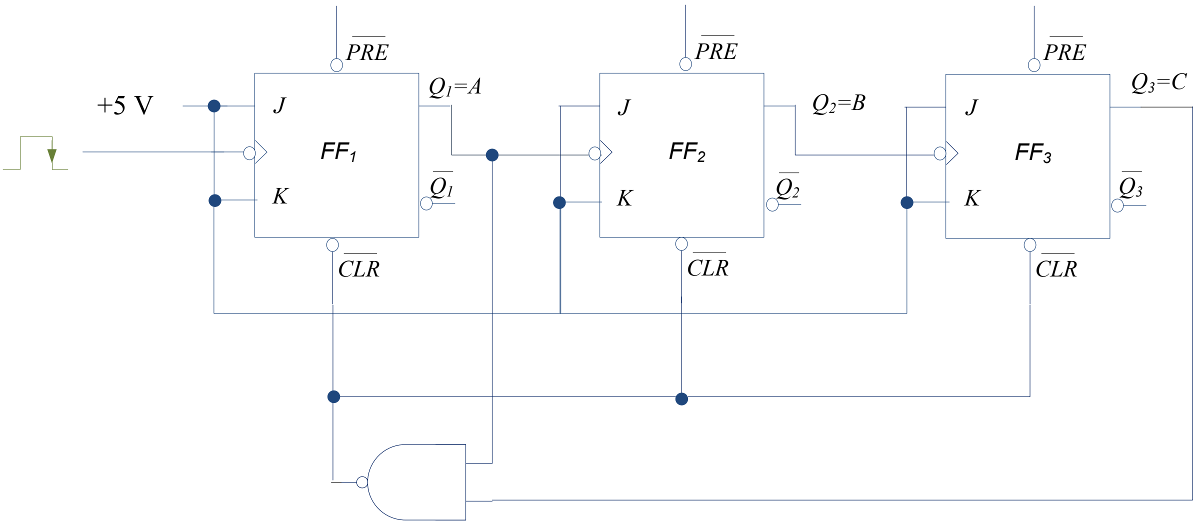


Figure 8.14: Five state asynchronous up-counter with NGT clock pulse (with $\overline{CLR}$ input).

8.4 Synchronous counters

Synchronous counters are advantageous over asynchronous counters as they do not suffer from clock ripple effect due to all flip-flops being clocked at the same time. Also, they allow counter design in any arbitrary sequence. However, synchronous counters often require additional circuitry. In this section, several examples will be used to illustrate the synchronous counter design.

The basic steps in the design are:

1) Obtain the state diagram/table
2) Decide the number of flip-flops and type of flip-flop
3) Derive the state excitation table
4) Obtain the simplified expressions for the flip-flop inputs (for example using K-maps)
5) Draw the logic circuit diagram

8.4.1 Synchronous counter – example 1

Assume that we wish to design a counter that counts 000→010→011→111 and then recycles back to 000. In this counter, there are several unused states: 001, 100, 101 and 110. Though these states should not occur in our design, it is good practice to set the counter to go to 000 if any of these undesired states do occur.

Step 1: State diagram is shown in Figure 8.15.

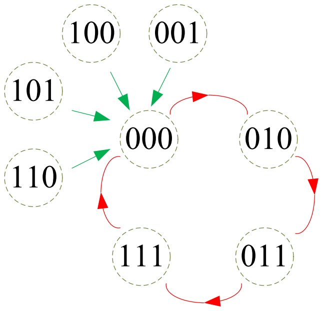


Figure 8.15: State diagram for synchronous counter in example 1.

Step 2: The number of flip-flops is three and let us assume that J-K flip flops are used.

Step 3: The excitation table is basically a truth table that gives the necessary J and K inputs to enable a change in the current output *Q* to next state *Q*$^+$. Table 8.2 shows the general excitation table for a J-K flip-flop (with don't care conditions, X).

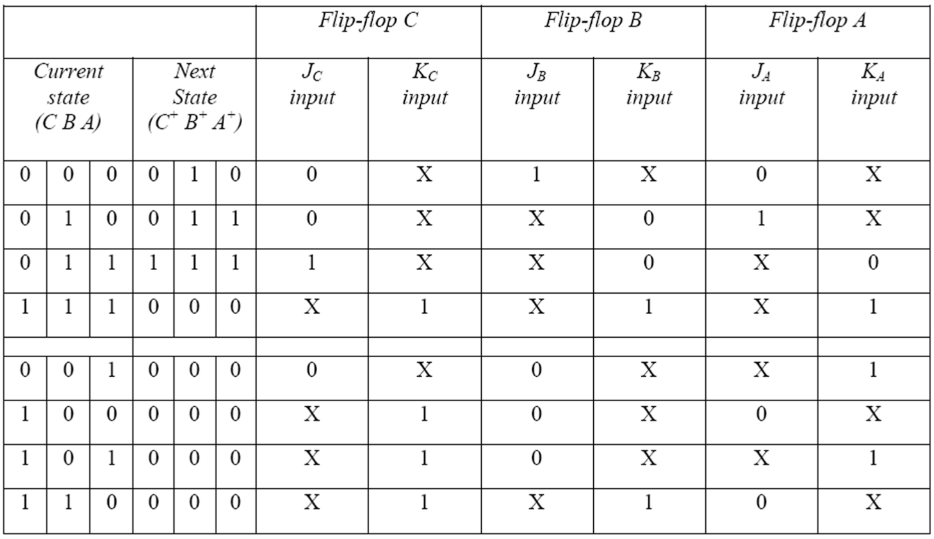
Table 8.2: Excitation table for general J-K flip-flop

J input	*K input*	*Current output, Q*	*Next output, Q*$^+$
0	X	0	0
1	X	0	1
X	1	1	0
X	0	1	1

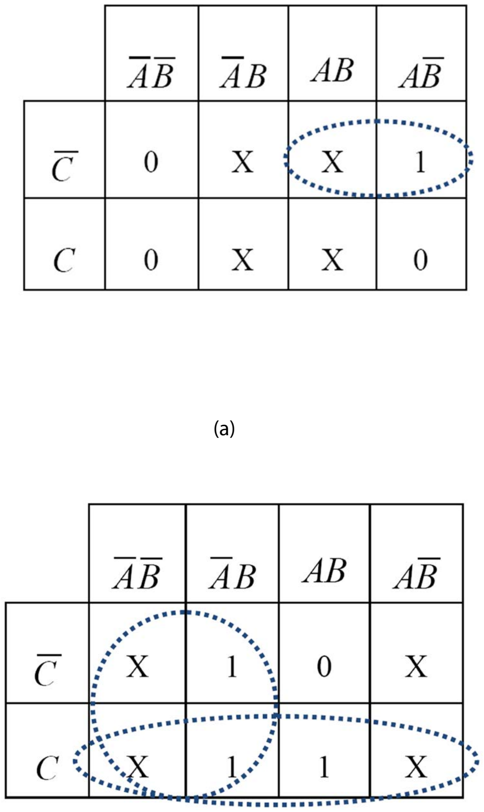
The excitation table for the counter to be designed is shown in Table 8.3.

Step 4: Using the excitation Table 8.3, we can obtain the K-maps for each input as shown in Figures 8.16-8.18 where present states should be used to draw the K-maps.

Table 8.3: Excitation table for the counter in example 1



						Flip-flop C		*Flip-flop B*		*Flip-flop A*	
Current state (C B A)			*Next State (C^+ B^+ A^+)*			*J_C input*	*K_C input*	*J_B input*	*K_B input*	*J_A input*	*K_A input*
0	0	0	0	1	0	0	X	1	X	0	X
0	1	0	0	1	1	0	X	X	0	1	X
0	1	1	1	1	1	1	X	X	0	X	0
1	1	1	0	0	0	X	1	X	1	X	1
0	0	1	0	0	0	0	X	0	X	X	1
1	0	0	0	0	0	X	1	0	X	0	X
1	0	1	0	0	0	X	1	0	X	X	1
1	1	0	0	0	0	X	1	X	1	0	X



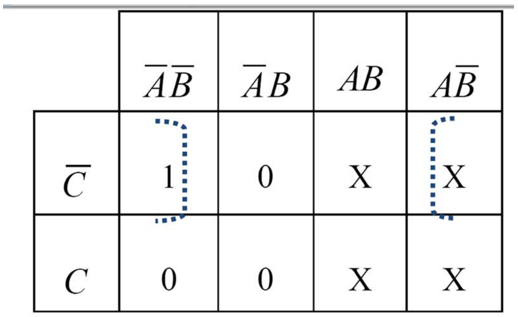
	$\overline{A}\,\overline{B}$	$\overline{A}B$	AB	$A\overline{B}$
$\overline{C}$	0	X	X	1
C	0	X	X	0

(a)

	$\overline{A}\,\overline{B}$	$\overline{A}B$	AB	$A\overline{B}$
$\overline{C}$	X	1	0	X
C	X	1	1	X

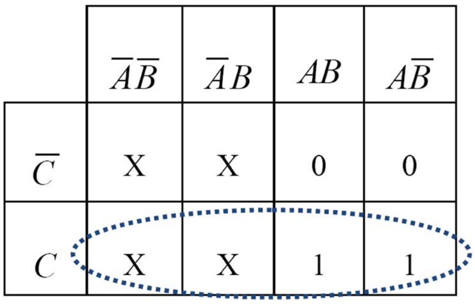
(b)

Figure 8.16: K-maps for inputs (a) J_A and (b) K_A.



	$\overline{A}\,\overline{B}$	$\overline{A}B$	AB	$A\overline{B}$
$\overline{C}$	1	0	X	X
C	0	0	X	X

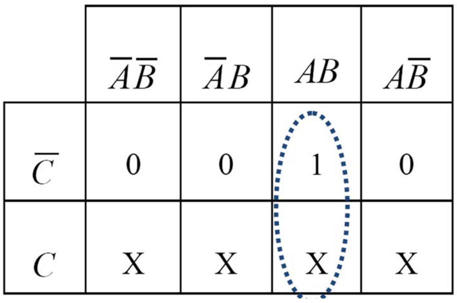
(a)



	$\overline{A}\overline{B}$	$\overline{A}B$	AB	$A\overline{B}$
$\overline{C}$	X	X	0	0
C	X	X	1	1

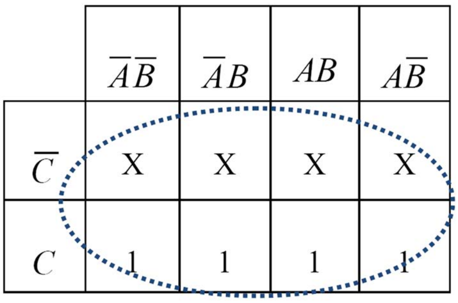
(b)

Figure 8.17: K-maps for inputs (a) J_B and (b) K_B.



	$\overline{A}\overline{B}$	$\overline{A}B$	AB	$A\overline{B}$
$\overline{C}$	0	0	1	0
C	X	X	X	X

(a)



	$\overline{A}\,\overline{B}$	$\overline{A}B$	AB	$A\overline{B}$
$\overline{C}$	X	X	X	X
C	1	1	1	1

(b)

Figure 8.18: K-maps for inputs (a) J_C and (b) K_C.

From the K-maps, the simplified expressions for the inputs are:

$$J_A = A\overline{C} \qquad K_A = C + \overline{A}$$

$$J_B = \overline{B}\,\overline{C} \qquad K_B = C$$

$$J_C = AB \qquad K_C = 1$$

Step 5: The logic circuit diagram is given in Figure 8.19. Notice that all the clock inputs are tied together and hence the flip-flops are clocked simultaneously.

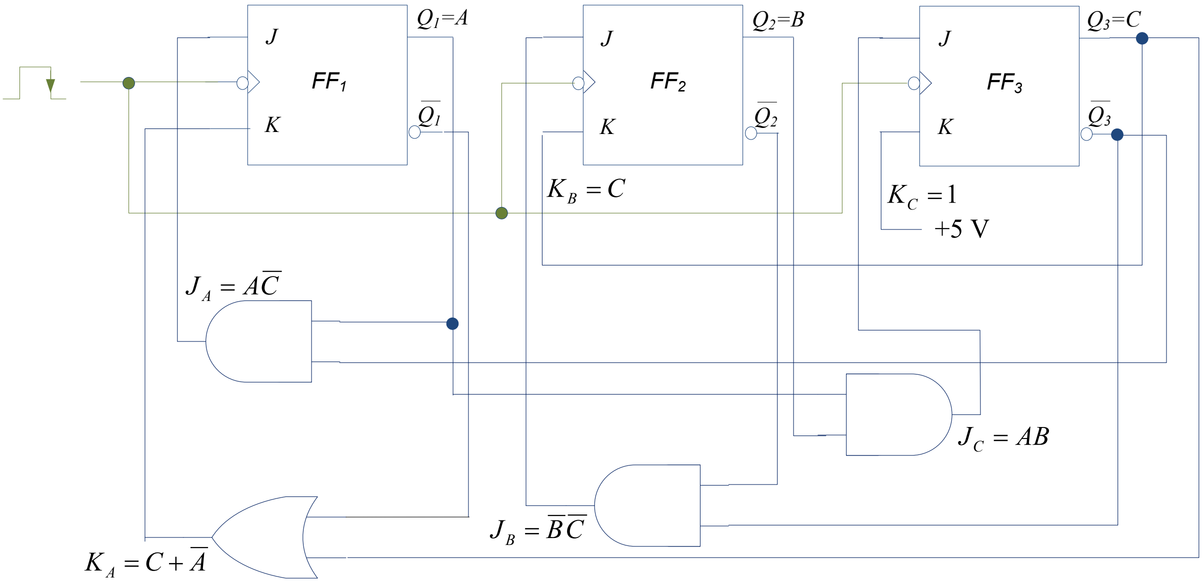


Figure 8.19: Logic circuit diagram for counter in example 1.

8.4.2 Synchronous counter – example 2

Now consider another example using T flip-flops and the counter to be designed cycles through 000→010→100→110 and then resets to 000.

Step 1: Since the LSB of the counter does not change, we need not be concerned about the design for this bit and can set Q_A=0. So, the simplified state diagram is as shown in Figure 8.20.

Step 2: The number of flip-flops is two only and T flip-flops will be used.

Step 3: The general excitation table for T flip-flop is given in Table 8.4, while the excitation table for the counter is given in Table 8.5.

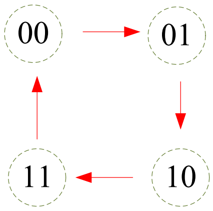


Figure 8.20: Simplified state diagram for example 2.

Table 8.4: General excitation table for T flip-flop

T input	*Current* *output, Q*	*Next* *output, Q^+*
0	0	0
1	0	1
1	1	0
0	1	1

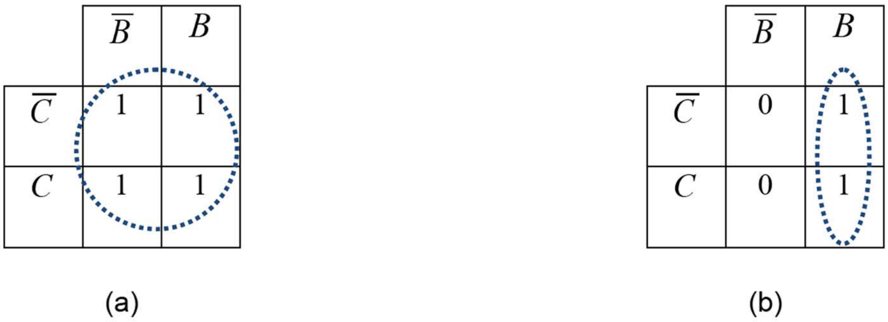
Table 8.5: Excitation table for the counter in example 2

Current *state* *(C B)*		*Next* *State* *(C^+ B^+)*		*Flip-flop C* *T_C*	*Flip-flop B* *T_B*
0	0	0	1	0	1
0	1	1	0	1	1
1	0	1	1	0	1
1	1	0	0	1	1

Step 4: The K-maps are shown in Figure 8.21. The simplified expressions are

$$T_B = 1$$

$$T_C = B$$



	$\overline{B}$	B
$\overline{C}$	1	1
C	1	1

(a)

	$\overline{B}$	B
$\overline{C}$	0	1
C	0	1

(b)

Figure 8.21: K-maps for example 2: (a) T_B (b) T_C.

Step 5: The logic circuit diagram is shown in Figure 8.22.

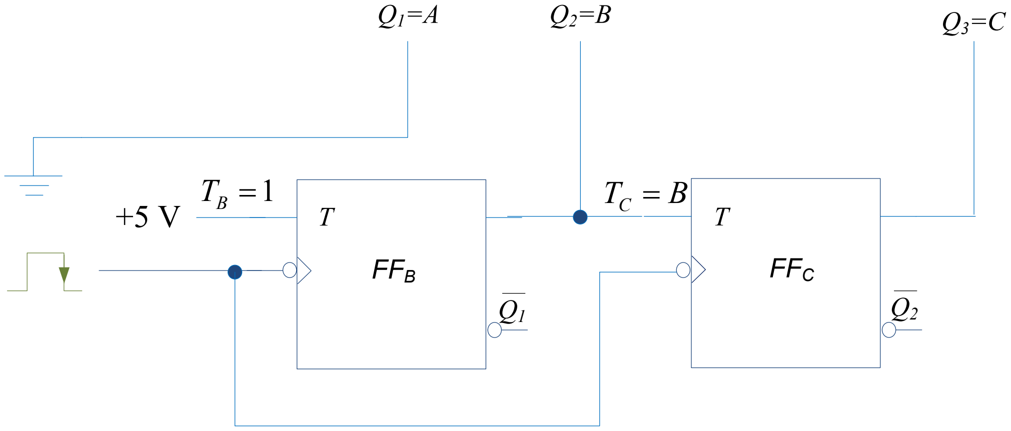


Figure 8.22: Logic circuit diagram for example 2.