Chapter 1
Introduction

In this chapter, we will discuss basic digital system principles starting with digital signals and numbers. Then, digital systems building elements, the logic gates, and how they are realized will be discussed. Later in this chapter, we will learn how to use logic gates to design logic circuits and how to model and analyze logic circuits using Boolean Algebra.

1. Digital Signals and Systems

Logic circuits are used to build computer hardware, as well as many other types of products. Actually, almost, all modern consumer electronics are digital systems that use analog circuitry only to interface with the surrounding environment through sensors and actuators. All such products are classified as digital hardware. Digital hardware deals with digital signals with only two values: a high voltage and a low voltage. But what is a signal?

A signal is a description of how one parameter varies with another parameter. For instance, voltage changing over time in an electronic circuit, or brightness varying with distance in an image.

Signals carry information and are defined as any physical quantity that varies with time, space, or any other independent variable. Signals are ubiquitous in science and engineering. Examples include:

- Electrical signals: currents and voltages in AC circuits, radio communications signals, audio and video signals.
- Mechanical signals: sound or pressure waves, vibrations in a structure, earthquakes.
- Financial Signal: time variations of a stock value or a market index.

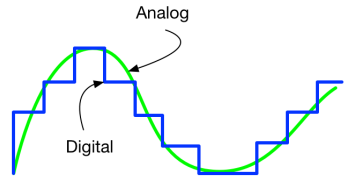


Figure 1-1. Analog vs. Digital Signals

In electrical systems, signals represent information that is transmitted between devices using an electrical quantity (voltage or current). Signals could be analog or digital (Figure 1-1):

- An Analog signal is any continuous (in time and in values) signal for which the time varying feature (variable) of the signal is a representation of some other time varying quantity, i.e., analogous to another time varying signal. An analog signal has a theoretically infinite resolution. In practice, an analog signal is subject to electronic noise and distortion introduced by communication channels.
- A Digital signal uses discrete (discontinuous) values. Digital signals have a finite resolution. They can in general be processed or transmitted without introducing additional noise or distortion.

A digital signal can be produced from an analog signal through the process of analog-to-digital conversion. The conversion process can be thought of as occurring in two steps as outlined by Figure 1-2:

- Sampling, which produces a continuous-valued discrete-time signal. The generated signal is usually called Pulse Amplitude Modulated signals (PAM),
- Quantization, which replaces each sample value by an approximation selected from a given discrete set (for example by truncating or rounding), and
- Encoding, which represents the quantized samples by numbers.

Internally, digital systems represent each discrete value as a string of 0's and 1's (also, called ON/OFF or TRUE/FALSE). 0's and 1's are easy to present and store. The signal that is presented by the 2 values: 0 and 1 are called Binary signals because it has only 2 valid states. In that sense, a Digital signal is actually a collection Binary signals.

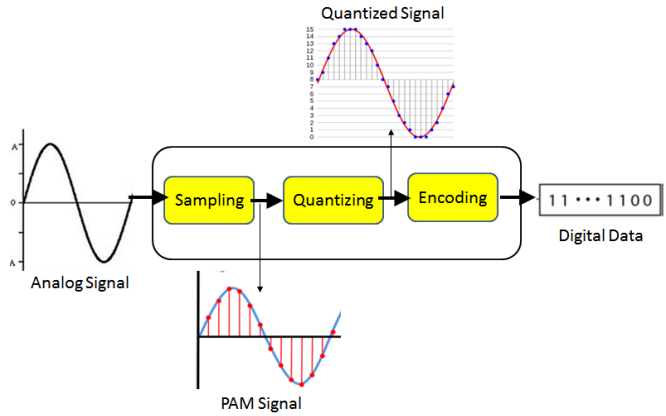


Figure 1-2. Analog to Digital Conversion

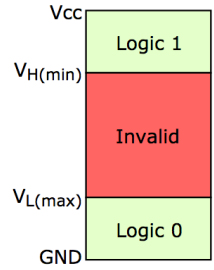


Figure 1-3. Using Voltage Levels to represent Logic "0" and Logic "1"

Digital Systems, usually, encode "0" and "1" using voltage levels; one level for "0" and another level for "1" (e.g., 0 volt for "0" and 5 volts for "1"). But as world is not ideal, we must ensure that a "0" will never be mistaken for anything and the same for "1". For that, digital systems encode "0" and "1" into voltage ranges, instead of a single value, as shown in Figure 1-3. Any voltage in the ranges labeled "0" and "1" represents a "0" or "1" symbol respectively. This way, a "0" or "1" transmitted over a noisy channel (that adds a small random voltage value to the transmitted signal) will not be distorted. Voltages between the two ranges for "0" and "1", the region labeled "Invalid" in Figure 2, are undefined and represent neither symbol. Voltages outside the ranges, below the "0" range or above the "1" range are not allowed and may damage the system if they occur.

2. Numbers

In digital signals, discrete values are represented by numbers. To process digital signals, binary numbers made of strings of 0's and 1's are used. Digital systems are built to process these 0's and 1's. In this section, you will learn about different number systems with a focus on binary number systems. Also, we will show how addition and subtraction are performed in these systems.

As humans, numbers are very important to us. We keep track of many numbers in our lives. We are always keeping score, measuring, recording and counting. We use numbers to communicate, to quantify, to measure, label, etc. In common usage, number may refer to a symbol, a word, or a mathematical abstraction. Numbers have become symbols of the present era. A notational symbol that represents a number is called a numeral.

2.1 Positional Numeral Systems

Number systems provide the basis for conveying and quantifying information. For this purpose, we find the decimal (or Hindu–Arabic numeral system) number system to be reliable and easy to use.

A number is represented by a string of symbols that are called digits. The possible set of digits that can be used to express a number is defined by the numbering system. In decimal system, there are 10 digits which is no surprise:

we have 10 fingers! The decimal numbering system is a positional system. In a positional numbering system, numbers are represented by an ordered set of digits and the value of each digit depends on its position. For example, the right digit of the decimal number 22 has the value 2 while the left digit has the value 20.

In general, every positional numbering system is defined by a base (b) that determines the number of possible digits as well as the weight of every digit in a number. The first known positional numeral system was the Mesopotamian base 60 system (3400 BC) and the earliest known base 10 system dates to 3100 BC in Egypt. An *n-bit* number in base b is represented by the string of digits:

$$(d_{n-1} \ldots . d_3 d_2 d_1 d_0)_b$$

The weight of any digit is b^p where p is the digit position ($p = 0$ for the right most digit). The value of an n-digit number in base b is $\sum_{i=0}^{n-1} d_i \times b^i$. For example, the value of the decimal ($b = 10$) number 5302_{10} has the value: $2 \times 10^0 + 0 \times 10^1 + 3 \times 10^2 + 5 \times 10^3$. Please notes that if the base is omitted from the number, base-10 is assumed; as decimal is the most commonly used numbering system. Another Example:

$$72.021 = 7 \times 10^1 + 2 \times 10^0 + 0 \times 10^{-1} + 2 \times 10^{-2} + 1 \times 10^{-3}$$

In the following subsections, we, briefly, review the concept of positional number systems, with a focus on the binary system (using the digits 0 and 1). We, also, discuss number systems that are variations on the binary system: octal and hexadecimal. In addition to that, we will learn how to convert between different positional number systems, and the basics binary addition and subtraction.

2.2 Binary, Octal, and Hexadecimal Numbers

Digital hardware systems, almost universally, use the binary number system (base-2) rather than the decimal number system (base-10). However, the basic concepts of positional number systems still apply. A binary number can be represented only by using the two digits 0 and 1. These are called binary digits, or simply *bits*. As the number is written down, each bit has twice the weight of its neighbor to its immediate right. For example, consider the 5-bit binary number 11001_2; what is its value? Let's rewrite it in positional notation:

$$11001_2 = 1 \times 2^0 + 0 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 + 1 \times 2^4 = 25_{10}$$

The simplest way to convert a decimal number to binary is to repeatedly divide the number by two and keep track of the quotient and remainder. The quotient gives the next number to divide by two and the remainder, always either 1 or 0, is one binary digit of the result. If the numbers are not too big, dividing by two is easy enough to do without paper and pencil. For example, to convert 30 to binary we do the following:

$30 \div 2$	$= 15$	remainder 0	0_2
$15 \div 2$	$= 7$	remainder 1	10_2
$7 \div 2$	$= 3$	remainder 1	110_2
$3 \div 2$	$= 1$	remainder 1	1110_2
$1 \div 2$	$= 0$	remainder 1	11110_2

Hence, $31 = 11110_2$.

Table 1. Hexadecimal digits and their Binary Equivalents

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

It is common to find binary numbers consisting of 8, 16, 32 and even 64 digits which makes it difficult to both read or write without producing errors. To make life easier, designers have introduced the hexadecimal number systems (base-16). The 16 hexadecimal digits are: 0, 1, 2, …., 9, A, B, C, D, E and F. It is easy to convert between binary and the hexadecimal system, because the base is a power of 2. So, every hexadecimal digit can be replaced by 4 bits according to table 1. For example, $A0B5_{16} = 1010\ 0000\ 1011\ 0101_2$. In the hexadecimal number system, we represent the binary digits as a set of 4 digits ($2^4 = 16$), in octal numbering system we represent the binary numbers as a set of 3 digits ($2^3 = 8$). The octal number system uses 8 digits from 0 to 7. For example, the binary number $100\ 011\ 010_2$ is equivalent to 432_8. Converting from base 16 to decimal is very similar to converting Binary to Decimal:

$$A0B5_{16} = 5 \times 16^0 + 11 \times 16^1 + 0 \times 16^2 + 10 \times 16^3 = 41141$$

2.3 Addition in Positional Notation

In positional number systems, the numbers are added column by column, one position at a time. Should the column sum exceed the largest digit (1 in binary, 9 in decimal, 7 in octal and F in hexadecimal), we must generate a carry-out to the next higher-order position. The following examples illustrate this idea:

- $125 + 478$

Carry out ↓				
0	1	1	←	Carries
	1	2	5	
	4	7	8	
	6	0	3	← Sum

- $2A5_{16} + 767_{16}$

0	1	0	←	Carries
	2	A	5	
	7	6	7	
	A	0	C	← Sum

- $101_2 + 011_2$

1	1	1	←	Carries
	0	1	1	
	1	0	1	
	0	0	0	← Sum

2.4 Subtraction in Positional Notation

Subtraction can be performed using addition using the method of complements. The method of complements is a technique used to subtract one number from another using only addition of positive numbers. For a number system that uses the base b, each number has 2 complements, the b's complement and (b-1)'s complement. In decimal system (b=10), each number has a 10's complement and a 9's complement. The 9's complement of a number is formed by replacing each digit with nine minus that digit. The 10's complement is the 9's complement plus 1. For example, the 9's complement of the number 121 is $999 - 121 = 878$ and the 10's complement is 879. On other words, the 9's complement is $10^3 - 1 - 121 = 878$ and the 10's complement is $10^3 - 1 - 121 + 1 = 10^3 - 121 = 879$ (3 is the number of digits). To generalize, for an n-digit number y in base b:

- the b's complement is $b^n - y$ and
- the (b-1)'s is $b^n - 1 - y$.

The following steps summary the subtraction in decimal system using the method of complements:

- Make the both numbers having same numbers of digits by adding 0's to the left of the number that has less digits.
- Determine the 10's complement of the number to be subtracted.
- Add the 10's complement to the given number from which we subtract (minuend).
- If the carry out is 1, remove it from the result and the remaining digits form the final result. If the carry out is 0, then determine the 10's complement of the result and prefix it by the negative sign.

For example, to subtract 255 from 1239

- $1239 - 255$
- 10's complement of 0255 = 9's complement of $0255 + 1 = 9999 - 0255 + 1 = 9744 + 1 = 9745$
- $1239 + 9745 = 0984$; carry out = 1
- $1239 - 0255 = 984$

Another example but in binary, 11101_2 - 1101_2

- $11101_2 - 01101_2$
- 2's complement of 01101_2 = 1's complement of $011012 + 12 = 111112 - 011012 + 12 = 100112$
- $111012 + 100112 = 10000_2$; carry out = 1
- $11101_2 - 01101_2 = 10000_2$

From the second example, it is obvious that finding the 1's complement of any binary number is as easy as complementing every bit, every 0 becomes 1 and every 1 becomes 0. For the 2's complement, scan the number from the right and flip all the bits to the left of the first occurrence of 1. For example, the 2's complement of 110010_2 is 001110_2.

2.5 Signed Numbers

All the numbers referred to so far are unsigned numbers. As negative numbers are often involved in scientific computing, the representation of signed numbers is discussed here. An n-bit base b signed number is represented as follows:

$$(d_{n-1} \ldots . d_3 d_2 d_1 d_0)_b$$

Where the most significant digit (MSD), dn-1, is used as a sign bit such that $d_{n-1} = 1$ if the number is negative (less than zero) and $d_{n-1} = 0$ if the number is not negative (zero or positive). The remaining digits are used to represent the magnitude of the number. In general, there are three main methods for representation of signed integers. They are: sign-magnitude, radix complement, and diminished radix complement. In all of them, the most significant digit (msd) represents the sign (o: positive; 1: negative). Table 2, shows how a signed 4-bit binary number may be represented using the three methods.

Sign-Magnitude: According to this representation, the msd of n-digit number represents the sign and the remaining (n-1) digits are used to represent the magnitude of the number. For example, the negative number (-20) is represented using 6 bits, base 2 in the sign-magnitude format, as follows (110100_2), while a (+20) is represented as (010100_2). Although simple, the sign-magnitude representation is complicated when performing arithmetic operations.

Radix Complement: According to this system, a positive number is represented the same way as in the sign-magnitude. However, a negative number is represented using the b's complement (for base b numbers). In decimal, the radix complement is the 10's complement which is obtained by subtracting the n-bit number from 10^n. In binary, it is called the 2's complement and obtained

by subtracting the number from 2^n, or inverting all the digits (1's complement; see below) then add one to the result. Consider, for example, the representation of the number (-19) using 2's complement. In this case, the number 19 is first represented as (010011_2). Then the 2's complement is calculated (101101_2). The Radix complement is the preferred method for presenting negative numbers in digital systems. Subtracting b from a is equivalent to adding a to the 2's complement of b. Formally, the Radix Complement of the number N is $b^n - N$

Table 2. 4-bit signed values using different representations

Decimal	2's Complement	1's Complement	Signed Magnitude
-8	1000	—	—
-7	1001	1000	1111
-6	1010	1001	1110
-5	1011	1010	1101
-4	1100	1011	1100
-3	1101	1100	1011
-2	1110	1101	1010
-1	1111	1110	1001
0	0000	1111 or 0000	1000 or 0000
1	0001	0001	0001
2	0010	0010	0010
3	0011	0011	0011
4	0100	0100	0100
5	0101	0101	0101
6	0110	0110	0110
7	0111	0111	0111

Diminished Radix Complement: This representation is similar to the radix complement except that we use the (b-1)'s complement instead of the b's complement. In the decimal number system, the diminished radix complement is 9's complement. This is obtained by subtracting each digit of the number from 9. For binary, it is called the 1's complement and is obtained by subtract each digit from 1. This is equivalent to complementing every bit. According to this, (-19) is represented in binary as (101100_2), while a (+18) is represented as (010010_2). The main disadvantage of the diminished radix representation is that it cannot be used to implement subtraction using addition directly as with the Radix Complement. A correction factor is needed whenever a carry is obtained from the most significant bit while performing arithmetic operations. Moreover, the number "0" has 2 representations (see Figure 3). Formally, the Radix Complement of the number N is $b^n - N - 1$

2.6 Binary Coded Decimal

binary-coded decimal (BCD), which encodes the digits 0 through 9 by their 4-bit unsigned binary representations, 0000 through 1001. The code words 1010 through 1111 are not used. Conversions between BCD and decimal representation is trivial, a direct substitution of four bits for each decimal digit. For example, the decimal number 25 can be encoded into 8 bits as follows: 0010 0101. The main advantage of binary coded decimal is that it allows easy conversion between decimal (base-10) and binary (base-2) form. However, the disadvantage is that BCD code is wasteful as the states between 1010 (decimal 10), and 1111 (decimal 15) are not used.

Old computers, such as the IBM System/360, used to support BCD and placed two BCD digits in one 8-bit byte in packed-BCD representation; thus, one byte may represent the values from 0 to 99 as opposed to 0 to 255 for a normal unsigned 8-bit binary number. Also, they included instructions that can perform arithmetic directly on packed BCD data and convert between packed BCD data and other integer representations. Modern computers dropped the support for BCD; BCD capabilities are almost always implemented in software rather than the CPU's instruction set. However, BCD is still used in some applications specially in digital displays.

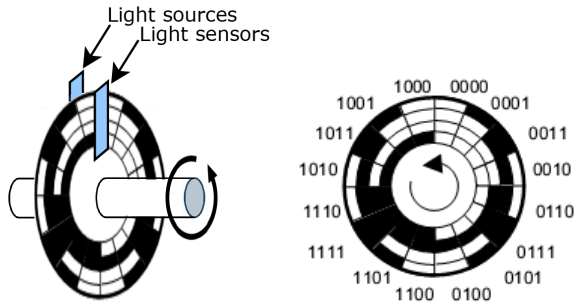


Figure 1-4. Gray Code in Rotary Encoders

Start	Mirror	Prefix	Mirror	Prefix
0	0	00	00	000
1	1	01	01	001
	1	11	11	011
	0	10	10	010
			00	100
			01	101
			11	111
			10	110
1-bit		2-bit		3-bit

Figure 1-5. 3-bit Gray Code generation

2.7 Gray Code

Is a reflected binary code (RBC) invented by Frank Gray of Bell Labs (1887-1969). Gray code is an ordering of the binary numeral system such that two successive values differ in only one bit. Gray code is sometimes used to refer to any single-distance code that is, one in which adjacent code words (perhaps representing integers differing by 1) differ by 1 in one-digit position only. Gray code was originally designed to prevent spurious output from electromechanical switches. Today, Gray codes are widely used to facilitate error correction in digital communications and in rotary encoders used to measure the angular displacement (Figure 1-4). In rotary encoders, Gray code is used to avoid the possibility that, when several bits change in the binary representation of an angle, a misread will result from some of the bits changing before others. A Gray code can be generated by the following procedure which is illustrated on Figure 1-5:

1. Start with the simplest Gray code possible; that is, for a single bit.
2. Create a mirror image of the existing Gray code below the original values.
3. Prefix the original values with 0s and the mirrored values with 1s.
4. Repeat steps (2) and (3) until the desired width is achieved.

3. Logic Gates

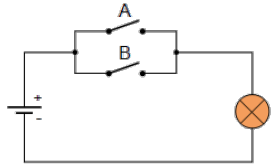
Digital Systems building blocks are categorized as one of two types, depending on whether they contain memory. Blocks without memory are called **combinational**; the output of a combinational block depends only on the current input. In blocks with memory, the outputs can depend on both the inputs and the value stored in memory, which is called the **state** of the logic block. Such blocks are called **sequential**.

The basic building blocks in digital systems are called the logic gates. Logic gates are combinational and operate with 0's and 1's. All digital systems can be constructed by using only three basic logic gates. These basic gates are called the AND gate, the OR gate, and the NOT gate. A logic gate can be realized using a switching circuit built out switches. Switches could be mechanical (e.g., push button), electro-mechanical (e.g., relays) or electronic (e.g., transistors). Relays were used to realize early logic circuits. Modern logic circuits are realized using different transistor-based implementation technologies discussed in Section 6.

3.1 OR Gate

An OR gate will give a high output (Logic 1) if any of the inputs is high, that is why it is sometimes called "any or all" gate. The schematic in Figure 1-6.a shows the idea of the OR gate. The lamp will glow when either switch A or switch B is closed. The lamp will also glow when both switches A and B are closed. The lamp will not glow when both switches (A and B) are open. The OR gate function can be expressed using the following algebraic expression: $Y = A + B$

and/or the table in Figure 1-6.c which is called the truth table. A truth table shows how the inputs of a combinational logic function relate to its output(s). The truth table has 2n rows for n-input function; each row represents a unique combination of the inputs. Figure 1-6.b shows the symbol used for 2-input OR gate.



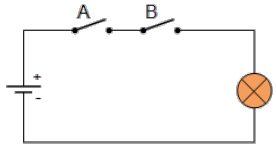
(a)

(b)

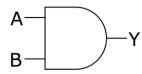
A	B	$Y = A + B$
0	0	0
0	1	1
1	0	1
1	1	1

(c)

Figure 1-6. The OR Gate (a) Using Switches, (b) OR Gate Symbol, (c) Truth Table



(a)

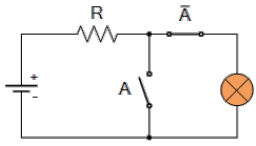


(b)

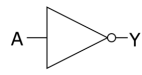
A	B	$Y = A \cdot B$
0	0	0
0	1	0
1	0	0
1	1	1

(c)

Figure 1-7. The AND Gate (a) Using Switches, (b) AND Gate Symbol, (c) Truth Table



(a)



(b)

A	$Y = \bar{A}$
0	1
1	0

(c)

Figure 1-8. The NOT Gate (a) Using Switches, (b) NOT Gate Symbol, (c) Truth Table

3.2 AND Gate

An AND gate will give a high output if all of the inputs are high. For example, in a simple lighting circuit (Figure 1-7.a) with two switches in series; the lamp will glow if both switches are closed. The AND gate function can be expressed using the following algebraic expression: $Y = A \cdot B$. The truth table of the AND function is given by Figure 1-7.c. Figure 1-7.b shows the symbol used for the AND gate.

3.3 NOT Gate (Inverter)

One of the most basic operations in a digital system is inversion, or negation. The logic gate that performs such operation is called the inverter, or sometimes the NOT gate. The NOT gate has only one input and one output. Figure 1-8.a shows the switch representation of the inverter. If switch A is closed, the light is

off (due to the established short circuit across the lamp terminals). When switch A is open there will be current flow through the lamp and the light will be on. The inversion can be expressed using the expression $Y = \bar{X}$. Figure 1-8.b shows the logic symbol for the inverter and Figure 1-8.c shows its truth table.

4. Boolean Algebra

Boolean algebra is the branch of algebra in which the values of the variables (Boolean Variables) are the truth values **true** and **false**, usually denoted 1 and 0 respectively. Main operations of Boolean algebra are the conjunction **and** denoted as ·, the disjunction **or** denoted as +, and the negation not denoted as ¯ or `. Boolean algebra was introduced by George Boole in his first book The Mathematical Analysis of Logic (1847), and set forth more fully in his An Investigation of the Laws of Thought (1854). Boolean algebra has been fundamental in the development of digital electronics, and is provided for in all modern programming languages. In 1938, Shannon proved that a two-valued Boolean algebra (whose members are most commonly denoted 0 and 1, or false and true) can describe the operation of two-valued electrical switching circuits.

Table 3. Basic Identities of Boolean Algebra

	Identity Name	AND Form	OR Form
1	Identity Law	$\mathbf{1} \cdot x = x$	$\mathbf{0} + x = x$
2	Null Law	$\mathbf{0} \cdot x = \mathbf{0}$	$\mathbf{1} + x = \mathbf{1}$
3	Idempotent Law	$x \cdot x = x$	$x + x = x$
4	Inverse Law	$x \cdot \bar{x} = \mathbf{0}$	$x + \bar{x} = \mathbf{1}$
5	Commutative Law	$x \cdot y = y \cdot x$	$x + y = y + x$
6	Associative Law	$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	$(x + y) + z = x + (y + z)$
7	Distributive Law	$x + y \cdot z = (x + y) \cdot (x + z)$	$x \cdot (y + z) = x \cdot y + x \cdot z$
8	Absorption Law	$x \cdot (x + z) = x$	$x + (x \cdot z) = x$

Combining the variables and operators yields Boolean expressions. A Boolean function typically has one or more input values and yields a result, based on these input values, in the range {0,1}. A Boolean expression is used to express Boolean functions, For example:

$$f(x, y, z) = x + y \cdot z$$

Frequently, a Boolean expression is not in its simplest form. Recall from algebra that an expression such as $2x + x$ is not in its simplest form; it can be reduced (represented by fewer or simpler terms) to $3x$. Boolean expressions can also be simplified, but we need new identities, or laws, that apply to Boolean algebra instead of regular algebra. These identities, which apply to single Boolean variables as well as Boolean expressions, are listed in Table 3. Please notice the following:

- 1 is the neutral value for the AND operation (.) and 0 is neutral value for the OR (+) operation
- It is legal to distribute the OR operation on a product

- The AND operator can be interpreted as having an enabling or disabling function (sometimes called a 'gating function"; Null and Identity Laws)

Note that each relationship, given by Table 3, has both an AND (product) form and an OR (sum) form. This is known as the duality principle. Every Boolean expression has a dual which is can be obtained by interchanging every 0 with 1, every 1 with 0, as well as interchanging the operators i.e., every (+) with (.) and every (.) with (+). The principle of duality is an important concept in Boolean algebra, particularly in proving various theorems. Once an identity is proved, by the principle of duality, its dual is also valid. Hence, our effort in providing various theorems is reduced to half.

x	y	$\overline{x+y}$	$\bar{x}\cdot\bar{y}$
0	0	1	1
0	1	0	0
1	0	0	0
1	1	0	0

Figure 1-9. Proof of DeMorgan's theorem

4.1 DeMorgan's Theorem

The theorem is named after Augustus De Morgan (1806–1871), who introduced a formal version of the theorem to classical propositional logic. The theorems play important parts in Boolean algebra. The theorem states that the complement of a product is equal to the sum of the complements. That is, if the variables are A and B, then $\overline{(A\cdot B)} = \bar{A} + \bar{B}$. Using duality, the complement of a sum is equal to the product of the complements. In equation form, this can be expressed as $\overline{(A+B)} = \bar{A}\cdot\bar{B}$. DeMorgan's theorem applications will be discussed later when NAND and NOR gates are introduced.

4.2 Two-Input Boolean Functions

The number of different combinations of n Boolean variables is $m = 2^n$. Each combination represents a unique input which can generate either "true" or "false" as the output. Hence, the number of combinations for the output, the number of Boolean functions, is 2^m. Table 3 shows the 16 possible functions for 2 variables ($n = 2$).

Table 4. List of all possible Boolean functions of two variables

Function		0	$\bar{x}\bar{y}$	$\bar{x}y$	$\bar{x}$	$x\bar{y}$	$\bar{y}$	$\bar{x}y + x\bar{y}$	$\overline{xy}$	xy	$\bar{x}\bar{y} + xy$	y	$\bar{x} + y$	x	$x + \bar{y}$	$x + y$	1
X	y																
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

5. Logic Circuits

A logic circuit is the physical realization of a logic function. Hence, a logic function can be expressed using a logical expression, a truth table or a logic circuit. Digital logic circuits use several logic gates. To realize the logic function.

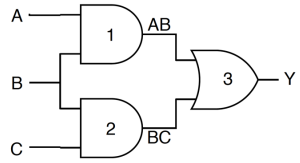


Figure 1-10. Logic circuit for the logic function $Y = f(A,B,C) = A \cdot B + B \cdot C$

The most common pattern of gates is shown in Figure 1-10. This pattern is called the AND-OR (Product-Sum) pattern. Also, called the Sum of Products (or SoP). The outputs of the AND gates (1 and 2) are feeding the inputs of the OR gate (3). You will note that this logic circuit has three inputs (A, B, and C). The output of the entire circuit is labeled Y. The expression that describes this logic circuit is:

$$Y = f(A,B,C) = A \cdot B + B \cdot C \quad (1)$$

The expression tells us that if both variables A and B are 1, the output will be 1. Same goes for B and C; if both are 1, the output is 1. $A \cdot B$ and $B \cdot C$ are called product terms; so, Y is the sum of these 2 product terms. The truth table of the logic function is given in Figure 1-11.

A	B	C	Y	
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	1	← $\bar{A} \cdot B \cdot C$
1	0	0	0	
1	0	1	0	
1	1	0	1	← $A \cdot B \cdot \bar{C}$
1	1	1	1	← $A \cdot B \cdot C$

Figure 1-11. Truth Table for the logic function $Y = f(A,B,C) = A \cdot B + B \cdot C$

6. Minterms and Maxterms

As discussed earlier, a logic function/circuit can be represented as SoP or PoS. If the product term has all the variables (or their inversions), the product term is called a minterm. Hence, $A \cdot B$ and $B \cdot C$ are not minterms because the product term $A \cdot B$ does not have C or $\bar{C}$; the same goes for $B \cdot C$ as it does not have A or $\bar{A}$.

For a boolean function of n variables $f(x_0, x_1, x_2, \ldots, x_{n-1})$, a product term in which each of the n variables appear once (in either its complemented or

uncomplemented form) is called a minterm. We define the variable or its complement as a **Literal**. Thus, a minterm is a logical expression of n variables employs only the complement operator and the AND operator.

- 1- minterms = minterms for which the function $f = 1$.
- 0-minterms = minterms for which the function $f = 0$.

If a function is expressed as the sum of 1-minterms then this sum is called Canonical Sum of Products (SoP). The canonical SoP is unique for any given logic function. The canonical SoP of the logic function given in Figure 9 is

$$f(A,B,C) = \bar{A} \cdot B \cdot C + A \cdot B \cdot \bar{C} + A \cdot B \cdot C \quad (2)$$

$$f(A,B,C) = m_3 + m_6 + m_7 = \Sigma m(3,6,7) \quad (3)$$

The principle of duality suggests that if it is possible to synthesize a function f by considering the rows in the truth table for which $f = 1$, then it should also be possible to synthesize f by considering the rows for which $f = 0$.

$$\overline{f(A,B,C)} = m_0 + m_1 + m_2 + m_4 + m_5 = \Sigma m(0,1,2,3,5) \quad (4)$$

$$f(A,B,C) = \overline{m_0 + m_1 + m_2 + m_4 + m_5} \quad (5)$$

$$f(A,B,C) = \overline{m_0} \cdot \overline{m_1} \cdot \overline{m_2} \cdot \overline{m_4} \cdot \overline{m_5} = M_0 \cdot M_1 \cdot M_2 \cdot M_4 \cdot M_5 \quad (6)$$

So, $\bar{f}$ is the sum of 0-minterms $(m_0, m_1, m_2, m_4, m_5)$. Applying DeMorgan's theorem, we end up with the sum of the complements of the 0-minterms. The complement of a minterm is a sum term (using DeMorgan's) that contains all the function literals. We call the complements of a minterm a maxterm.

Expression (6) represents f as a product of maxterms. A logic expression consisting of sum (OR) terms that are the factors of a logical product (AND) is said to be of the product-of-sums (PoS) form. If each sum term is a maxterm, then the expression is called a canonical PoS for the given function.

A logic function can be expressed using several expressions; some of them are simpler than the others. For example, the expressions of (1) and (2) represent the same logic function. However, (1) is simpler than (2). Finding the simplest expression is an important design step to minimize the number of logic gates of the circuit that implements such function. Boolean Algebra can be used to minimizing the number of gates of a circuit by simplifying Boolean expressions through Boolean manipulation. In fact, Boolean Algebra laid down the foundations for designing, optimizing and analyzing the logic circuits, can be used to.

7. Don't Care and Incompletely Specified Functions

Don't care condition is a condition (often represented by X) that indicates the actual signal value or values have no impact on the outcome a circuit. Don't care values can be inputs that do not affect the output of a circuit or outputs that do not matter for a specific combination of inputs. When the output of a circuit is not defined for all possible input combinations, the Boolean function that

represents such circuit is called an Incompletely specified function. An example of an incompletely specified function, a BCD increment by 1 function. BCD (Binary Coded Decimal) is a system for coding a number in which each digit of a decimal number is represented individually by its binary equivalent. Hence, 0 is represented by 0000, 1 by 0001, …. And 9 by 1001. The binary codes 1010, 1011, 1100, 1101, 1110, 1111 are not used. Figure 1-12 shows the truth table of the BCD increment by 1 function ($Y = X + 1$). We don't care about the last 7 rows of the truth table as they correspond to non-BCD digits.

X				Y			
x_3	x_2	x_1	x_0	y_3	y_2	y_1	y_0
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	1
0	1	1	1	0	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	X	X	X	X
1	1	1	0	X	X	X	X
1	1	1	1	X	X	X	X
1	0	0	0	X	X	X	X
1	0	0	1	X	X	X	X
1	1	1	0	X	X	X	X
1	1	1	1	X	X	X	X

Figure 1-12. BCD Increment by 1 Truth Table

8. Other Logic Gates

The most complex digital systems, such as large computers, are constructed from basic logic gates. The AND, OR, and NOT gates are the three fundamental gates. Four other useful logic gates can be made from these fundamental devices. The other gates are called the NAND gate, the NOR gate, the exclusive-OR (XOR) gate, and the exclusive-NOR (XNOR) gate.

x	y	$\overline{x.y}$
0	0	1
0	1	1
1	0	1
1	1	0

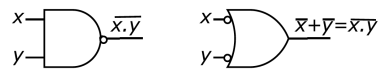


Figure 1-13. The Truth Table and Logic Symbols for NAND Gate

x	y	$\overline{x+y}$
0	0	1
0	1	0
1	0	0
1	1	0

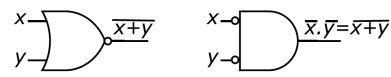


Figure 1-14. The Truth Table and Logic Symbols for NOR Gate

8.1 NAND and NOR Gates

The NAND and NOR gates produce complementary output to AND and OR, respectively. Each gate has two different logic symbols that can be used for gate representation. Figure 1-13 and Figure 1-14 depict the logic diagrams for NAND and NOR along with the truth tables to explain the functional behavior of each gate. NAND and NOR gates are attractive as they are cheaper to implement compared to the AND and OR gates as we will see later.

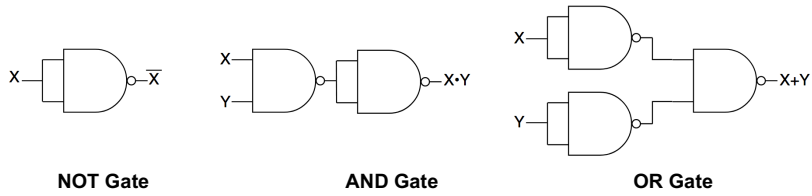


Figure 1-15. Implementing NOT, OR and AND using NAND gates only

The NAND gate is commonly referred to as a universal gate, because any electronic circuit can be constructed using only NAND gates. To prove this, Figure 1-15 depicts an AND gate, an OR gate, and a NOT gate using only NAND gates. Why not simply use the AND, OR, and NOT gates we already know exist? There are two reasons to investigate using only NAND gates to build any given circuit. First, NAND gates are cheaper to build than the other gates. Second, complex integrated circuits (which are discussed in the following sections) are often much easier to build using the same building block (i.e., several NAND gates) rather than a collection of the basic building blocks (i.e., a combination of AND, OR, and NOT gates). Please note that the duality principle applies to universality as well. One can build any circuit using only NOR gates.

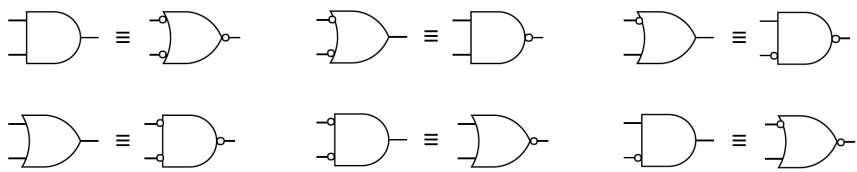


Figure 1-16. Bubble pushing

A shortcut method for forming equivalent logic circuits, based on DeMorgan's theorem, is to use what's called bubble pushing. Bubble pushing involves the flowing tricks: First, change an AND gate to an OR gate or change an OR gate to an AND gate. Second, add inversion bubbles to the inputs and outputs where there were none, while removing the original bubbles. That's it. You can prove to yourself that this works by examining the corresponding truth

tables for the original gate and the bubble-pushed gate, or you can work out the Boolean expressions using DeMorgan's theorem. Figure 1-16 shows examples of bubble pushing.

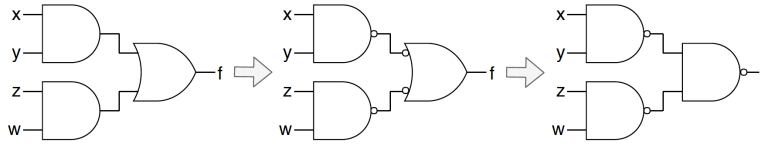


Figure 1-17. Using NAND gates to implement a SoP

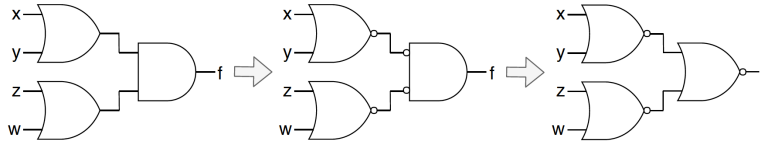


Figure 1-18. Using NOR gates to implement a PoS

8.2 Implementing Logic Circuits using NAND only or NOR only

In Section 6 we explained how a logic function can be implemented either in SoP or PoS form, which leads to logic networks that have either an AND-OR or an OR-AND structure, respectively. In this section, we show how such networks can be implemented using only NAND gates or only NOR gates.

Consider the network in Figure 1-17 as a representative of general AND-OR networks. This network can be transformed into a network of NAND gates as shown in the figure. First, each connection between the AND gate and an OR gate is replaced by a connection that includes two inversions of the signal: one inversion at the output of the AND gate and the other at the input of the OR gate. Such double inversion has no effect on the behavior of the network. According to Figure 1-16, the OR gate with inversions at its inputs is equivalent to a NAND gate. Thus, we can redraw the network using only NAND gates, as shown in Figure 1-17. This example shows that any AND-OR network can be implemented as a NAND-NAND network having the same topology. Figure 1-18 shows how to transform OR-AND logic network to NOR only logic network using a similar procedure to that used in Figure 1-17.

8.3 XOR and XNOR Gates

The exclusive-OR gate is referred to as the "any but not all" gate. The exclusive-OR term is often shortened to read as XOR. A truth table for the XOR function is shown in Figure 1-19. The truth table of the XOR is similar to the OR truth table except that, when both inputs are 1, the XOR gate generates a 0. The XOR gate is enabled only when an odd number of 1s appear at the inputs. Lines 2 and 3 of the truth table have odd numbers of Is, and therefore the output

is enabled with a 1. Lines 1 and 4 of the truth table contain even numbers (0,2) of Is, and therefore the XOR gate is disabled and a 0 appears at the output. The XOR gate could be referred to as an odd-bits checker.

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

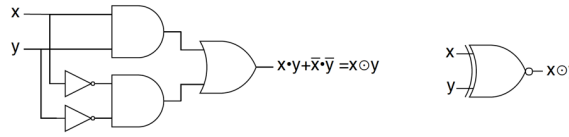


Figure 1-19. The XOR Gate

x	y	$x \odot y$
0	0	1
0	1	0
1	0	0
1	1	1

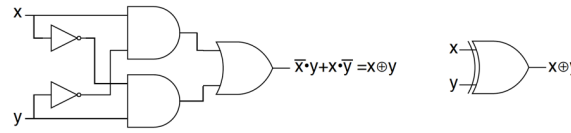


Figure 1-20. XNOR Gate

The $\oplus$ symbol is used for the XOR operator in Boolean algebra. The Boolean expression for XOR is

$$Y = \mathrm{A} \oplus \mathrm{B} = A \cdot \bar{B} + \bar{A} \cdot B$$

If the output of the XOR gate is inverted the result is the XNOR gate. The XNOR gate produces "1" if all the inputs are equivalent (sometimes we call the XNOR gate the EQV gate). The symbol $\odot$ is used as XNOR operator in Boolean algebra. Figure 1-20 shows the symbol and the equivalent circuit of the XNOR gate. The Boolean expression for the XNOR is:

$$Y = \overline{\mathrm{A} \oplus \mathrm{B}} = A \cdot B + \bar{A} \cdot \bar{B} = A \odot B$$

By examining the truth tables of the XOR and XNOR gates (Figure 1-19 and Figure 1-20), we can come up with the following set of equalities (Laws of XOR and XNOR):

$$\begin{array}{lcl} A \oplus 0 = A & & A \odot 1 = A \\ A \oplus 1 = \bar{A} & \overset{\text{By Duality}}{\longleftrightarrow} & A \odot 0 = \bar{A} \\ A \oplus A = 0 & & A \odot A = 1 \\ A \oplus A = 1 & & A \odot A = 0 \end{array}$$

Here, the dual relationship between the XOR and XNOR laws is established by interchanging the 1 's and O's while simultaneously interchanging the XOR and XNOR operators, as indicated by the double arrow.

9. Realizing Logic Circuits

As discussed earlier, logic circuits are built out of logic gates which are, mostly, built out of transistors. Transistors are semiconductor devices used to amplify or switch electronic. They have several distinct modes that make them useful in

a variety of applications. However, in digital systems we use them as a voltage-controlled switch. There are two main types of transistors: Metal Oxide Semiconductor Field-Effect Transistor (MOSFET or just MOS) and Bipolar Junction Transistor (BJT). Also, there are different ways for using them to build logic gates. Since the 1990s, most logic gates are made in CMOS (Complementary Metal Oxide Semiconductor) technology that uses both NMOS and PMOS transistors. For example, Figure 1 shows the CMOS implementation of the three basic gates (NOT, AND and OR) using MOSFET.

So, does that mean we need to build transistor circuits to realize logic circuits. Well, we do that if we want to build a custom integrated circuit[1] (IC), usually referred to as Application Specific Integrated Circuit (ASIC), from scratch which takes lots of engineering effort but yields the best performance in terms of area, delay and power. The book does not focus on ASIC design; instead it focuses on Field Programmable Gate Array (FPGA). However, many of the topics covered by the book are applicable for ASIC design.

9.1 CMOS Technology

As digital circuits are, mostly, implemented using MOS transistor utilizing the CMOS technology, being aware of their construct and, more specifically, their functionality can help aid the designer in making better overall design decisions (for example, low power design decisions).

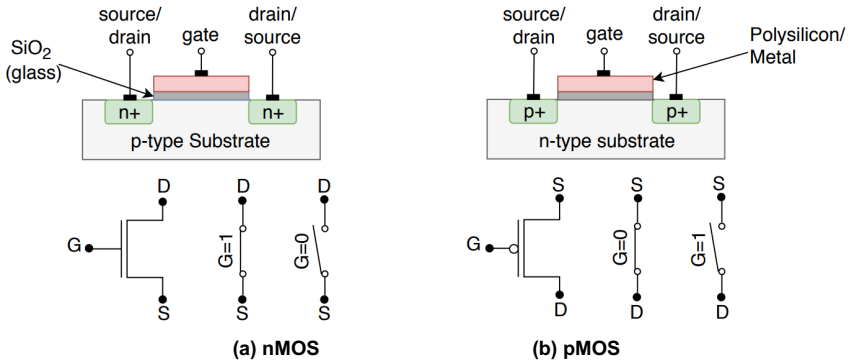


Figure 1-21. nMOS and pMOS Transistors

A MOS transistor is formed on a semiconductor substrate and has three terminals: the gate, source, and drain. The source and drain are identical terminals formed by diffusing an impurity into the substrate. The gate terminal is formed from polycrystalline silicon (called polysilicon) or a metal and is insulated from the substrate by a thin layer of oxide. The name MOS refers to the layering of the gate (metal), gate oxide (oxide) and substrate (semiconductor). There are 2 types of MOS transistors: n-channel MOSFET transistor (nMOS)

[1] circuit formed on a small piece of semiconducting material.

and p-channel MOSFET transistor (pMOS). Figure 1-21 shows the nMOS and pMOS transistors structures and their symbols. In an nMOS the source and drain are n-type semiconductor in a p-type substrate and the charge carriers are electrons. In the other hand, in a pMOS the types are reversed – the source and drain are p-type in a n-type substrate and the carriers are holes. To construct logic circuits, MOS transistors are used as electrically-controlled switches An electrical connection is created between the source and the drain of nMOS transistor by connecting its gate to the supply voltage (Logic "1"). The same happens for the pMOS when its gate is connected to the ground (Logic "0"). This is illustrated in Figure 1-21. A terminal other than the gate terminal, could be labeled a drain or source based on how a MOS transistor is used in a circuit. The source terminal of a pMOS transistor has a higher positive potential compared to the drain terminal. It is the other way around for the nMOS transistor.

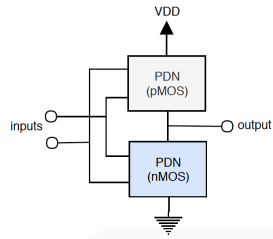
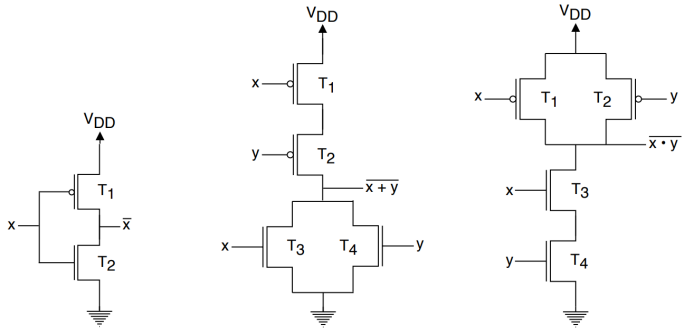


Figure 1-22. CMOS Circuit using both pMOS and nMOS



(a) Not Gate **(b) NAND Gate** **(c) NOR Gate**

Figure 1-23. CMOS implementation of basic logic gates

Using both nMOS and pMOS transistors, we can construct a CMOS circuit to implement any given logic function as shown by Figure 1-22. The circuit is made of a pMOS network of transistors connecting the output to the voltage supply (Pull Up Network or PUN) and a network on nMOS transistors connecting the output to the ground (Pull Down Network or PDN). The PDN and PUN networks are complements of each other. However, logic circuits can

be implemented using pMOS only or nMOS only transistors, CMOS is preferred technology for building logic circuits. CMOS offers low power dissipation, relatively high speed, high noise margins in both states, and will operate over a wide range of source and input voltages (provided the source voltage is fixed)

The simplest example of a CMOS circuit, a NOT gate, is shown in Figure 1-23.a. When x=0V, transistor T2 is off and transistor T1 is on. This makes $y=V_{DD}$, and since T2 is off, no current flows through the transistors. When $x=V_{DD}$, T2 is on and T1 is off. Thus x=0V, and no current flows because T1 is off.

2-input NAND gate (shown in Figure 1-23.b) PDN is made of 2 nMOS transistors in series and the PUN is made of 2 parallel pMOS transistors. To produce 0V on y, both x and y must be at V_{DD}; otherwise, the PUN connects the output to V_{DD}. For the 2-input NOR gate (Figure 1-23.c), x and y must be at 0V for the output to be at VDD; otherwise the PDN connects the output to the ground (0V).

In CMOS, the AND gate is constructed by attaching a CMOS inverter to the output of a CMOS NAND gate. Same for the OR gate, a NOR followed by an inverter. That means, non-inverting CMOS gates (AND and OR) need more transistors when compared to inverting CMOS gates (NAND and NOR). For CMOS, the number of transistors needed for a NAND or a NOR gate is $2n$; where n is the number of inputs (fan-in). For a CMOS NAND or NOR gate, it is $2n + 2$.

CMOS circuits are, just, one way to implement logic gates using MOS transistors. One may construct a logic gate using nMOS or pMOS only transistors. Using nMOS or pMOS only to construct logic gates yields to simpler circuits compared to CMOS alternative. For example, 2-input NAND gate requires 4 MOS transistors (2 nMOS and 2 pMOS transistors) in CMOS. The same gate needs only 2 nMOS transistors and a resistor when built using nMOS technology. However, CMOS circuits are superb when compared to the nMOS counterparts. They are more noise immune and consume less power.

Usually, several very small MOS transistors are constructed on a single small piece of silicon (or chip of silicon) and then interconnected with equally small metal wires. These microscopic MOS transistors typically have dimensions measured in nanometers (10^{-9}m). Since a silicon chip might measure several millimeters on a side, several billions of MOS transistors can be constructed on a single chip. Circuits assembled in this fashion are said to form "integrated circuits" (or IC's), because all circuit components are constructed and integrated on the same piece of silicon.

9.2 Standard Chips

An approach used widely until the mid-1980s was to connect together multiple chips, each containing only a few logic gates. A wide variety of chips, with different types of logic gates, is available for this purpose. There are several

families of logic chips but the most popular one is the Transistor Transistor Logic (TTL) family. Also, it is referred to as 7400-series because the chip part numbers always begin with the digits 74. The 7400-series chips are produced in standard forms by a number of integrated circuit manufacturers, using agreed-upon specifications. The logic gates inside these chips are implemented using either Bipolar Junction Transistors (e.g., 74LS sub-family) or MOS transistors (as CMOS circuit; e.g., 74HC sub-family). An example is the 7404 chip, which comprises six NOT gates. Figure 1-24 shows 7408 in the Dual Inline Package (DIP). This chip has four 2-input OR gates.

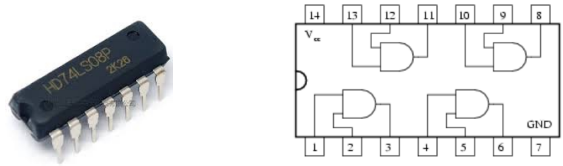


Figure 1-24. 7408 Chip in DIP Package

Building a small practical logic circuit using TTL usually requires many chips. As an example of how a logic circuit can be implemented using 7400-series chips, consider the function $f = A \cdot B \cdot C$. Figure 1-25 shows how to wire a single 7408 chip to implement f. This example makes use of only a portion of the gates available in the chip, hence the remaining gates can be used to realize other functions.

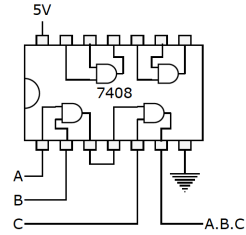


Figure 1-25. Implementing a logic circuit using TTL chips

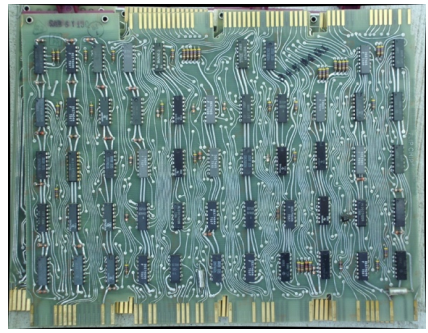


Figure 1-26. PDP-8 CPU Control Unit

Figure 1-26 shows the control unit of the CPU used in the PDP-8 minicomputer introduced in 1965. This computer was built out of standard chips. Because of their low logic capacity, the standard chips are rarely used in practice today, with one exception. Today, surface-mounted CMOS versions of the 7400 series are used, mainly, for glue logic[2] in computers and industrial electronics.

9.3 Field Programmable Gate Array (FPGA)

There are a wide variety of ICs that can have their logic function "programmed" into them after they are manufactured. Most of these devices use technology that also allows the function to be reprogrammed, which means that if you find a bug in your design, you may be able to fix it without physically replacing or rewiring the device.

Historically, programmable logic arrays (PLAs) were the first programmable logic devices. PLA structure was enhanced and PLA costs were reduced with the introduction of programmable array logic (PAL) devices. PLA and PAL devices are generically called programmable logic devices (PLDs). The ever-increasing capacity of integrated circuits created an opportunity for IC manufacturers to design larger PLDs for larger digital-design applications. Complex PLD (CPLD) and Field Programmable Gate Arrays (FPGA) were introduced to address that need. FPGA and CPLD devices were introduced around the same time by Xilinx and Altera respectively. They have different architectures and they achieve the programmability using different means. In general, FPGA is suitable for more complex digital designs compared to CPLD.

In short, FPGAs are off-the-shelf silicon chips whose function can be programmed by design engineers. This means that one chip can be re-programmed to perform many different functions. Also, the chip can be re-programmed to fix bugs or add new functionality in field. A feature is not possible for custom ASIC. the programmability is based on either SRAM (volatile) or FLASH (non-volatile). There are a large variety of FPGAs from different vendors, each with different capabilities, advantages, and limitations. FPGA will be visited in great details in Chapter 4.

9.4 A Brief History of Digital Hardware

The technology used to build digital hardware has evolved dramatically over the past few decades. There are two key inventions that have driven the digital revolution. The first was the invention of the transistor in the late 1940s, and the second was the invention of the integrated circuit in the late 1950s. Until the 1960s, logic circuits were constructed with bulky components, such as transistors and resistors that came as individual parts. The advent of integrated circuits made

[2] custom logic circuitry used to interface a number of off-the-shelf integrated circuits.

it possible to place a large number of transistors, and thus an entire circuit, on a single chip. In the beginning these circuits had only a few transistors, but as the technology improved they became more complex. Integrated circuit chips are manufactured on a silicon wafer, such as the one shown on Figure 1-27. The process of implementing logic circuit into a silicon wafer is a complex time consuming one. It involves hundreds of steps performed by expensive precise equipment in a clean-room environment. The fabrication takes place in a fabrication plant (commonly called a fab; sometimes foundry). Fabs are extremely expensive to build. Nowadays, it cost more than twenty billion US dollars to build a state of the art fab.

The manufactured wafer is cut to produce the individual chips (or dies), which are then placed inside a special type of chip package. By 1970 it was possible to implement all circuitry needed to realize a microprocessor on a single chip (Intel 4004) which laid the foundations for the microcomputer revolution that began in the 1970s. Although early microprocessors were small in size (Intel 4004 had 2,300 transistors only) and had modest computing capability by today's standards, they opened the door for the information processing revolution by providing the means for implementation of affordable personal computers.

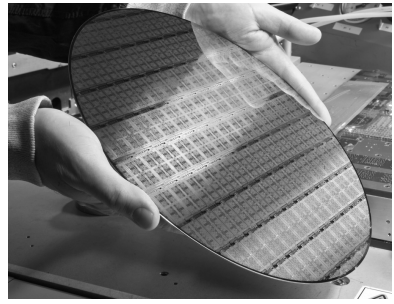


Figure 1-27. Silicon Wafer

In 1965, Gordon Moore, while working for Fairchild, observed that integrated circuit technology was progressing at an astounding rate, approximately doubling the number of transistors that could be placed on a chip every 12 months. This figure was revised in 1975, after co-founding Intel, to doubling every 2 years. This phenomenon, informally known as Moore's law, proved accurate for several decades, and has been used in the semiconductor industry to guide long-term planning and to set targets for research and development. Doubling the number of transistors is backed by shrinking the transistor. The first microprocessor chip, Intel 4004, was built using 10-micron transistors and had 2300 transistors. Nowadays, digital hardware can be built using transistors as small as 5nm (a strand of human DNA is 2.5 nm in diameter, a carbon atom is 0.3 nm) and tens of billions of them can be integrated on the same chip. This trend is illustrated in Figure 1-28.

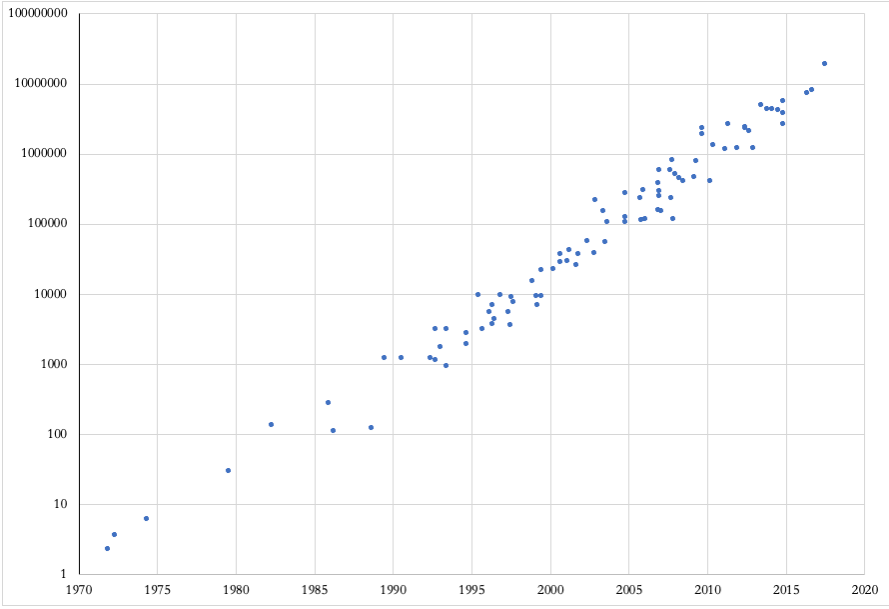


Figure 1-28. Number of Transistors per microprocessor from 1971 to 2018[3]

Moore acknowledges that the trend cannot last forever, and he gave a presentation at an international conference in 2003, entitled "No exponential is forever, but we can delay 'forever'". Currently, the pace of advancement has slowed down, so that Brian Krzanich, CEO of Intel, announced in 2015, "Our cadence today is closer to two and a half years than two." In 2015 report, International Technology Roadmap for Semiconductors (ITRS) predicted that the transistor could stop shrinking in just five years. The smallest transistor size is predicted to be 3nm. After 2021, the report forecasts, it will no longer be economically desirable for companies to continue traditional transistor miniaturization in microprocessors. Instead, chip manufacturers will turn to other means of boosting density, namely turning the transistor geometry from horizontal to vertical and building multiple layers of circuitry, one on top of another.

10. Summary

11. Questions

[3] Source: https://github.com/karlrupp/microprocessor-trend-data

Chapter 2
Combinational Logic

In the previous chapter we discussed the basic principles of digital systems. The goal of this chapter is to build on these principles to show how to optimize and analyze combinational logic circuits. Also, we will discuss the combinational logic building blocks and how to use them to build complex combinational logic. Combinational logic is a digital circuit that has output that depends on current value of inputs only, and it is a memoryless circuit. In the following chapter, we will discuss sequential circuits that contain memory elements; hence, can keep the circuit history.

1. Combinational Logic Optimization

As discussed in Chapter 1, every Boolean function can be expressed as a sum of minterms (Canonical SoP) or a product of maxterms (Canonical PoS). Since the number of terms in such expressions are usually high, and the complexity of the digital logic gates that implement a Boolean function is directly related to the complexity of the Boolean expression from which the function is implemented, it is preferable to have the most simplified form of the Boolean expression. Implementing the most simplified Boolean expression of a function results in the smallest logic circuit. One of the objectives of the digital designer when using is to keep the number of gates to a minimum when implementing a Boolean function. The smaller the number of gates used, the lower the cost of the circuit. The process of simplifying the Boolean expression of a Boolean function is called minimization. Minimization is important since it reduces the cost and complexity of the associated circuit.

Simplification could be achieved by a purely algebraic manipulation, but this can be tedious, and the designer is not always sure that the simplest solution has been found at the end of the process. A much easier method of simplification is to plot the function on a Karnaugh map (K-Map) and with the help of a number of simple rules to reduce the Boolean function to its minimal form. This particular method is very straightforward up to and including six variables. Above four variables it becomes tricky. For number of variables above six, it is better to use a tabulation method such as the Quine-McCluskey (QM) method which can be converted into a computer program. K-Map and QM methods

have as their starting point a truth table or, equivalently, a minterm list or maxterm list. If we are given a logic function that is not expressed in this form, then we must convert it to an appropriate form before using these methods. All minimization methods reduce the cost of a two-level AND-OR, OR-AND, NAND-NAND, or NOR-NOR circuit in three ways:

1. By minimizing the number of first-level gates.
2. By minimizing the number of inputs on each first-level gate.
3. By minimizing the number of inputs on the second-level gate. This is actually a side effect of the first reduction.

1.1 Minimization using Algebraic Manipulation

From Chapter 1, we know that a Boolean function can be represented using different Boolean expressions. Also, it is obvious that the complexity of logical implementation of a Boolean function is directly related to the complexity of algebraic expression from which it is implemented. However, Boolean algebraic manipulation can be used to simplify complex expressions. This method is the simplest of all methods used for minimization. It is suitable for medium sized expressions involving 4 or 5 variables. Algebraic manipulation is a manual method; hence it is prone to human error. It should be noted that there are no fixed rules that can be used to minimize a given expression. It is left to an individual's ability to apply Boolean laws in order to minimize a function. Hence, the simplified expression is not guaranteed to be the simplest one for the given function. For example, the function $f = (x + \bar{y} + \bar{z})(x + \bar{y}z)$ can be simplified as follows:

$f = xx + x\bar{y}\bar{z} + x\bar{y} + \bar{y}\bar{y}z + +x\bar{z} + \bar{y}z\bar{z}$ Distributive Law
$f = x + x\bar{y}\bar{z} + x\bar{y} + x\bar{z} + \bar{y}z$ Idempotent and Inverse Laws
$f = x(1 + \bar{y}\bar{z} + \bar{y} + \bar{z}) + \bar{y}z$ Distributive Law
$f = x + \bar{y}z$ Null Law

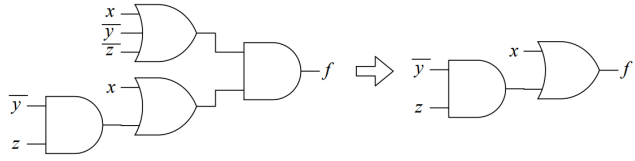


Figure 2-1. Logic diagram before and after minimization

The implementation cost of the original expression is one 3-input OR gate (8 MOS transistors), one 2-input OR gate (6 MOS transistors), two 2-input AND gate (2x6=12 MOS transistors) and three inverters (3x2=6 MOS transistors). Hence, the total cost is 32 MOS transistors. After minimization, the simplified expression cost is: One 2-input AND gate, one 2-input OR gate and one inverter which is equivalent to 14 MOS transistors (less than 44% of the original expression cost).

A seen from the example, the Algebraic manipulation method is tedious and cumbersome and does not suite complex functions. In the following sections, we will discuss other methods that provide systematic way to minimize Boolean functions which can be used with more complex Boolean functions.

1.2 Karnaugh Map

In 1953 Maurice Karnaugh developed K-Map in his paper titled 'The map method for synthesis of combinational logic circuits. The Karnaugh Map (or just K-Map) method is faster, as it takes advantage of humans' pattern-recognition capability, and can be used to solve Boolean functions of up to six variables. A K-Map is, just, another representation of the truth table. In a truth table, every row represents a minterm which is a product term that has all the function literals (a variable or its inversion). If a Boolean function has two minterms that differ only in one literal, these two minterms can be reduced into a single product term by eliminating the different literal. For example, $xyz + xy\bar{z}$ can be reduced into $xy(z + \bar{z}) = xy$. We refer to these two minterms xyz and $xy\bar{z}$ as being **logically adjacent** or have a logic distance of 1. Using a truth table to spot adjacent minterms is confusing as some logically adjacent minterms are not represented by physically adjacent rows as shown by Figure 13.a. Rows 1 and 3 represent 2 logically adjacent minterms but the 2 rows are not physically adjacent on the table. Same goes for rows 4 and 6. To solve this problem, the K-Map was introduced.

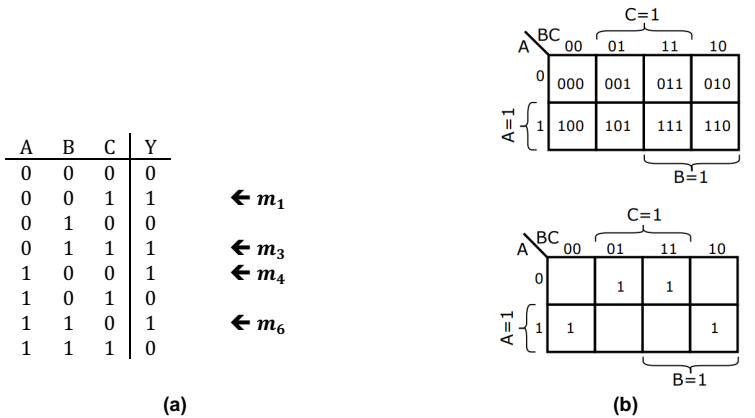


Figure 2-2. 3-Variable Truth Table and K-Map

A K-Map is a kind of truth table arranged so that adjacent entries represent minterms that differ by one literal. A K-Map makes it easy to spot such adjacent minterms to combine them. Figure 13.b shows a 3-variable K-Map to represent the truth table of Figure 13.a. Now minterms 1 (001) and 3 (011) are adjacent on the K-Map. Please note that on the 3-variable k-map the left most column and the right most column are adjacent. Figure 14 shows the M-Maps for 4 variables.

In the 4-variable K-Map the left most column and the right most column are adjacent; same goes for the top row and the bottom row. Also, the four corners are adjacent (Fold up the corners of the map below like it is a napkin).

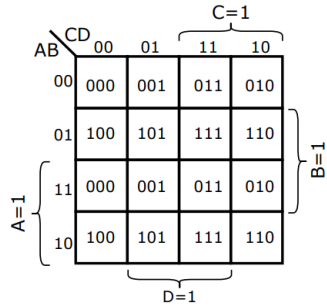


Figure 2-3. 4-Variable K-Map

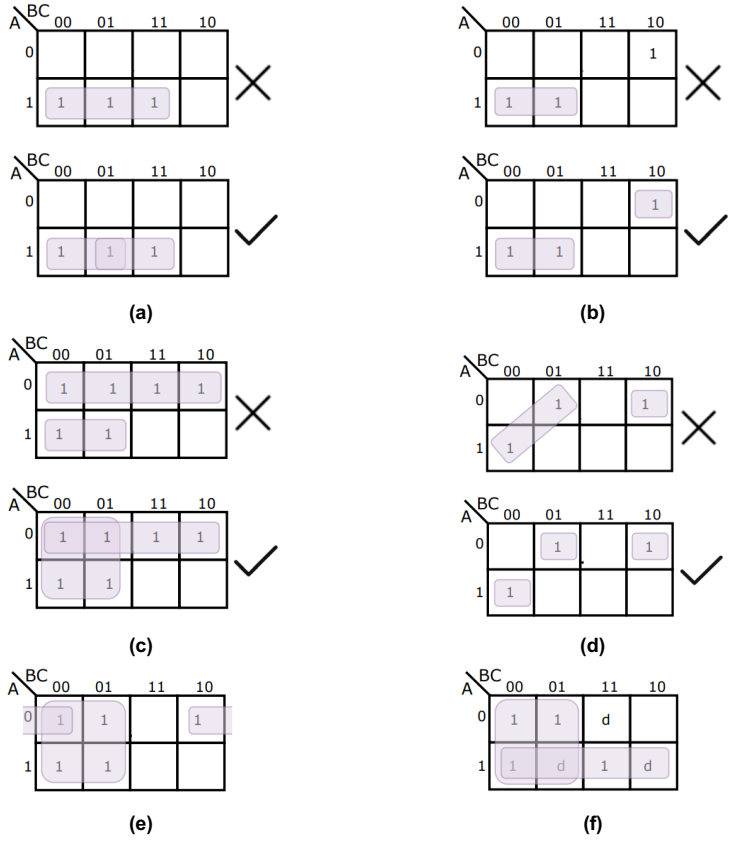


Figure 2-4. K-Map Solution Guidelines

Generating simplified Boolean expression for a K-Map involves two simple steps:

1) Finding largest groups of adjacent 1s (Covers). The group size must be a power of 2 (1, 2, 4, …).
2) Generating an equation from the identified groups

When grouping the cells together, there is a set of rules that must be followed:

- Groups are made of power of 2 number of cells; e.g. 1, 2, 4, 8, 16, … (Figure 15.a)
- Groups consists of one or more cells of 1s only - i.e. no cells of 0s and every cell of 1 must be in at least one group (Figure 15.b)
- Groups should be as large as possible (Figure 15.c).
- Overlapping groups are allowed (Figure 15.c)
- Groups can be made of cells that are adjacent to one another; i.e., cells must be alongside, above or below one another; but groups may not go diagonally (Figure 15.d)
- Wrapping around is allowed (Figure 15.e)

If grouping don't care cells results in larger groups, then more variables may be eliminated resulting in smaller Boolean expressions (Figure 15.d).

1.2.2 Simplifying Products of Sums

All the minimized Boolean functions derived from the maps in all previous examples were expressed in sum of products (SOP) form.Using the principle of duality, we can minimize logic functions by covering the cells containing 0s on a Karnaugh map. Each 0 on the K-Map corresponds to a maxterm in the canonical product of the logic function. By covering 0s, we are minimizing the complement of the function represented by the K-Map. Since the resulting SoP expression represents the complement of the function, the actual function is the complement of this expression. Hence, we end up with the minimum PoS representation of the function.

For an example, consider the Boolean function $f = \sum m(0,2,5,7)$ which represented by the K-Map given by Figure 2-5. If we cover the cells containing 0s then:

$$\bar{f} = \bar{A}C + A\bar{C}$$

$$\bar{\bar{f}} = f = \overline{\bar{A}C + A\bar{C}} = (A + \bar{C})(\bar{A} + C)$$

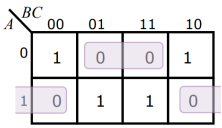


Figure 2-5. Simplifying the PoS

1.2.3 Prime Implicant

Solving the K-Map depends on the skills of designer and it is done visually. In many cases, finding the best solution is not obvious. In this section, we will try to devise and algorithm to systemically solve a K-Map. But before doing so, let's start with some definitions:

- Implicant: Any product term. In other words, a valid power of 2 cover on a K-Map. By definition, a single cell containing 1 (minterm) is an implicant.
- Prime Implicant (PI): An implicant that cannot be combined with any other adjacent implicants. A prime implicant is a product term which cannot be combined with other product terms to create a new product term with fewer number of literals.
- Essential Prime Implicant (EPI): The prime implicant is called essential if it is the only PI covering a minterm.

Using these definitions, solving a K-Map involves

1) Identifying all PIs and the EPIs.
2) Expressing the minimum form of the function by logically ORing the EPIs obtained in 1) along with other PIs that may be required to cover any remaining minterms not covered by the EPIs.

For example, consider the K-Map of Figure 2-6.

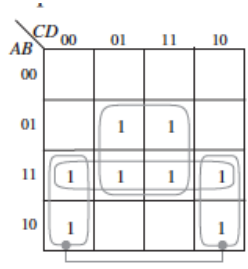


Figure 2-6.

There are 3 PIs ($A\overline{D}, AB, BD$) out of them 2 are EPIs ($A\overline{D}, BD$). As the EPIs cover all the minterms, the solution does not include the non-essential EPI.

$$f = A\overline{D} + BD$$

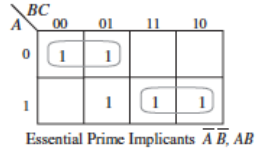


Figure 2-7.

Another example is shown on Figure 2-7. In this example, there are 4 PIs ($\bar{A}\bar{B}, AB, \bar{B}C, AC$). Two of them are EPIs ($\bar{A}\bar{B}, AB$). The solution includes the 2 EPIs and one of the non-essential PIs.

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

or

$$f = \bar{A}\bar{B} + AB + AC$$

1.2.4 Reed Muller Logic

Some digital functions can be difficult to optimize if they are represented in the conventional SoP or PoS forms. For these functions, it may be more appropriate to implement them in a form known as Reed-Müller logic, which is based on XORs and XNORs. The classical form of Reed-Muller logic is based on SoP-style representations, in which XOR gates replace OR gates. You can also implement functions using only XOR and XNOR gates. One indication as to whether a function is suitable for the Reed-Müller form of implementation is if that function's K-Map displays a checkerboard pattern of 0s and 1s. Consider a familiar two-input function as shown in Figure 2-8. The Boolean function represented by the K-Map of Figure 2-8(a) can be expressed as:

$$f_a = \bar{A}(\bar{B}C + B\bar{C}) + A(\bar{B}\bar{C} + BC) = \bar{A}(B \oplus C) + A(\overline{B \oplus C}) = A \oplus B \oplus C$$

And that of Figure 2-8(b) can be expressed as:

$$f_b = \bar{A}(\bar{B}\bar{C} + BC) + A(\bar{B}C + B\bar{C}) = \bar{A}(\overline{B \oplus C}) + A(B \oplus C) = \overline{A \oplus B \oplus C}$$

Larger checkerboard patterns involving groups of 0s and 1s also indicate functions suitable for a Reed-Müller implementation. For example, the Boolean function represented by the K-Map of Figure 2-8(c) can be expressed as:

$$f_c = \bar{B}\bar{C} + BC = \overline{A \oplus B}$$

And that of Figure 2-8(d) can be expressed as:

$$f_d = \bar{B}C + B\bar{C} = A \oplus B$$

It is obvious that implementing the functions of Figure 2-8 using XOR/XNOR gates yields less umber of transistors compared to the standard PoS/SoP implementations. For example, the cost of implementing f_a as PoS/SoP is 4x3-input AND gates and 1x4-input OR gate (42 transistors) versus 1x3-input XOR gate (20 transistors).

(a) (b)

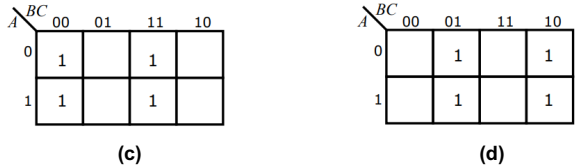


Figure 2-8. Examples of Boolean Functions suitable for Reed Muller Implementation

1.3 Minimizing Complex Logic Functions

Minimizing Boolean functions by hand using algebraic manipulation or K-maps is a laborious, tedious and error prone process. They are not suited for function with large number of variable (six or more) and practical only for up to four variables. Moreover, these methods don't lend themselves to be automated in the form of a computer program. As practical logic functions are usually dealing with large number of variables, relying on computers to perform logic optimizations is a must. The first alternative method to become popular was the tabular method developed by Willard Quine and Edward McCluskey. Starting with the truth table for a logic function, by combining the minterms for which the function is active (the ON-cover) or for which the function value is irrelevant (Don't-Care-cover or DC-cover) a set of prime implicants is composed. Finally, a systematic procedure is followed to find the smallest set of prime implicants to realize the logic functions.

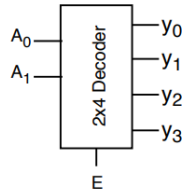
2. Combinational Logic Building Blocks

In this section, we will introduce a number of combinational circuit components that are used as building blocks in larger digital systems. While these components can, themselves, be constructed from gates, it is generally not useful to do so. Instead, we will work at a higher level of abstraction. We will think of these components as basic blocks that, together with gates, are used to construct complex combinational circuits.

2.1 Binary Decoders

A popular combinational building block which used in building larger digital circuits. A binary decoder has an n-bit input and 2^n outputs, where only one output is active (Logic 1) for each input combination. For that, sometimes, we refer to the output of a binary decoder as one-hot-output. Figure 14 shows the circuit diagram for 2x4 binary decoder. The decoder output can be globally enabled or disabled using ad optional enable control "E". When enabled (E=1), every decoder output (D0-D3) is set to 0 except of one output. This output is determined by the inputs A_0 and A_1. If disabled (E=0), all the outputs D0-D3 will be to '0'. The circuit can be designed using NAND gates instead of AND

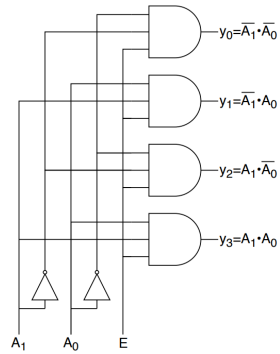
gates. In this case every output (y0-y3) is set to logic '1' except of one output, which is determined by the inputs A_0 and A_1.



(a) Blackbox

E	A_0	A_1	y3	y2	y1	y0
0	X	X	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

(b) Truth Table



(c) Logic Circuit

Figure 2-9. Binary Decoder

A binary decoder is also called a minterm generator. A minterm generator is constructed using AND and NOT gates. The appropriate output is indicated by logic 1 (positive logic). Using the duality, we can define the maxterm (minterm dual). A maxterm is a sum term that makes the function to be false (0); hence the maxterms are the complement of the minterms. We use M_0 to denote maxterm number 0 ($M_0 = \overline{m_0}$). A Boolean function can be represented as a product of maxterms (Canonical PoS). A binary decoder built out of NAND gates can be used as maxterms generator. In general, for n inputs, the n-to-2^n decoder when enabled selects one of 2^n minterms or maxterms at the output based on the input combinations. Any n-variable logic function, in canonical SoP form can be implemented using a single n-to-2n binary decoder to generate the minterms, and an OR gate to form the sum. **Figure 2-10** shows how to implement the function $f(x, y, z) = \Sigma m(1, 6, 7)$ using a 3x8 decoder.

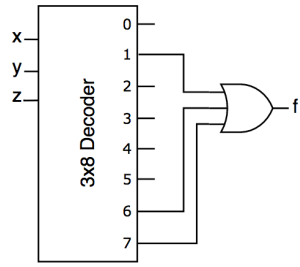


Figure 2-10. Implementing a logic function using a binary decoder

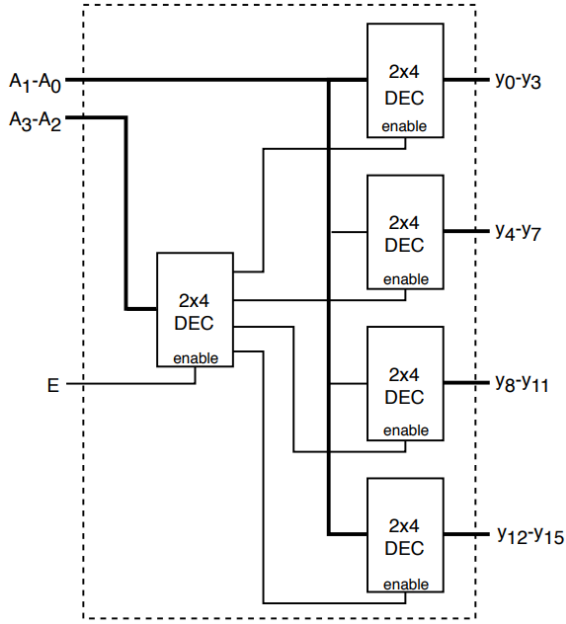


Figure 2-11. Implementation of a 4-to-16 decoder using 2-to-4 decoders.

Large decoders can be designed using small decoders as the building blocks. For example, a 4-to-16 decoder can be designed using five 2-to-4 decoders as shown in Figure 2-11.

2.2 Multiplexers

A multiplexer (MUX) is a popular combinational building block. A MUX is a digital switch, which connects data from one of n sources to a single output. A number of select inputs determine which data source is connected to the output. The MUX acts like a digitally controlled multi-position switch (Figure 17) where the binary code applied to the select inputs controls the input source that will be switched on to the output. Figure 18 shows the logic circuit and the black box of 2x1 multiplexer. The 2x1 multiplexer circuit can be divided into 2 sub-circuits. A 1x2 binary decoder (1st sub-circuit) that decodes the selection line into two

lines used to enable one of the 2 inputs using 2 AND gates (2nd sub-circuit). The outputs of the AND gates are connected to an OR gate to generate the multiplexer output. Using the same idea, a 4x1 multiplexer can be built out of a 2x4 binary decoder, 4 AND gates and an OR gate as outlined by Figure 19.

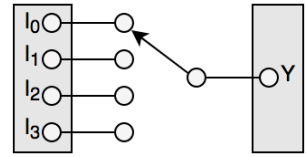


Figure 2-12. The multiplexer as a digitally controlled multi-position switch

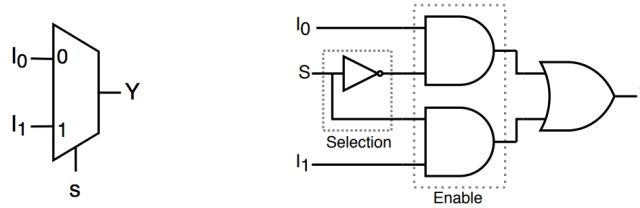


Figure 2-13. 2x1 Multiplexer

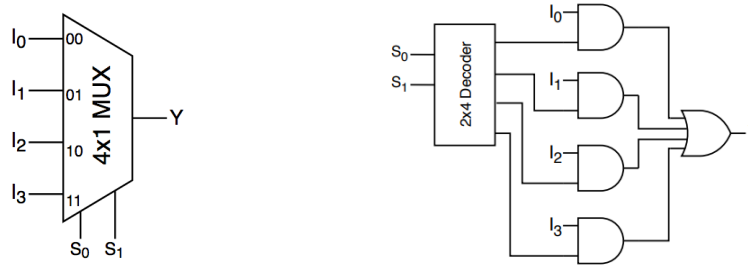


Figure 2-14. 4x1 Multiplexer

While Multiplexers are primarily thought of as "data selectors" because they select one of several inputs to be logically connected to the output, they can also be used to implement Boolean functions. Multiplexer sometimes is called universal logic circuit because a 2^n-to-l multiplexer can be used to implement a logic function of n variables directly from its truth table by applying 0' s and 1' s to the appropriate data inputs (Figure 20). Indeed, by changing the data inputs, the multiplexer can be reprogrammed to perform a different function.

x	y	f
0	0	0
0	1	1
1	0	1
1	1	0

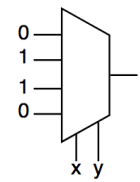
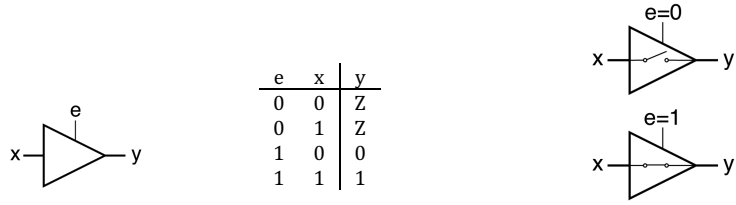


Figure 2-15. Using a 4x1 Multiplexer to Realize a Boolean Function

2.2.1 Tri-State Buffers and Inverters

Outputs of two ordinary logic gates cannot be connected together, because a short circuit would result if one gate forces the output value 1 while the other gate forces the value 0. Therefore, some special gate is needed if the outputs of two, or more, gates are to be connected to a common wire. A commonly used circuit element for this purpose is the tristate buffer. Unlike ordinary component outputs, which always drive either a low or high logic level, the output of a tristate buffer can be turned off by placing it in a high-impedance, or hi-Z, state. ("Z" is commonly used as the symbol for impedance in a circuit.) Tristate buffers enable multiple devices to share a common output wire by cooperatively agreeing to have only one device drive the wire (usually called the bus) at any one time, during which all other devices remain in hi-Z.



e	x	y
0	0	Z
0	1	Z
1	0	0
1	1	1

(a) Symbol **(b) Truth Table** **(c) Equivalent Circuit**

Figure 2-16. Tristate Buffer

As shown in Figure 21, a tristate buffer has a data input x, an output y, and an enable input e. Its operation is illustrated by the equivalent circuit in 21.b. The triangular symbol in the figure represents a buffer. The buffer performs no logic operation. It simply passes its input, unchanged, to its output ($f(x) = x$). Its purpose is to provide additional electrical driving capability, strengthen a weak signal4 or introduce delay to the circuit. The truth table of the tristate buffer is given by Figure 21.c.

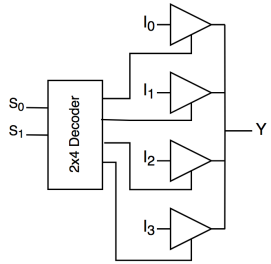


Figure 30. 4x1 Multiplexer using Tri-State Buffers

[4] A weak "1" is a logic 1 signal represented by a voltage very close to V_{Hmin}. A weak "0" is a logic 0 signal represented by a voltage very close to V_{Lmax}

Tristate buffers can be used to implement 4x1 multiplexers. Specifically, they are used to implement the enable part of a multiplexer as shown by Figure 22. Please note that the enables of the 4 tristate buffers are **mutual exclusive** (only one can be hot at a time) which is enforced through the usage of a 2x4 binary decoder.

2.2.2 Shannon's Expansion

Boole's expansion theorem, often referred to as the Shannon expansion or decomposition, is the identity:

$$f = x \cdot f_x + \bar{x} \cdot f_{\bar{x}}$$

Where f is any Boolean function, x is a variable, x' is the complement of x, and f_x and $f_{\bar{x}}$ are f with the argument x equal to 1 and to 0 respectively. The terms f_x and $f_{\bar{x}}$ are, sometimes, called the positive and negative Shannon cofactors, respectively, of f with respect to x. In the above identity, f is expanded on x. f can be expanded further on its other variables.

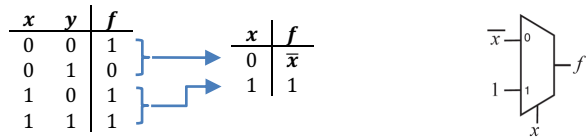


Figure 31. Expanding f on y to implement y using 2x1 multiplexor

We can implement any Boolean function of x variables using a 2^x input Multiplexer as we saw before. However, multiplexers can be used more effectively by using some forms of functional decomposition. For example, we can implement a 2-variable function using, only, 2x1 multiplexer instead of using 4x1 multiplexer as we did before. This example is illustrated in figure 23 where the function f is expanded on y.

Give example

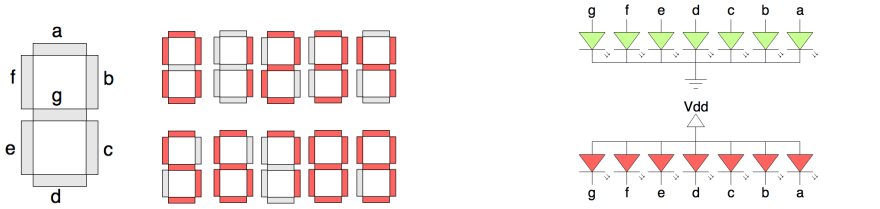
2.3 BCD to 7-Segment Converter/Decoder

The purpose of the decoder circuit is to convert from one type of input encoding to a different output encoding. For example, a 3-to-8 binary decoder converts from a binary number on the input to a one-hot encoding at the output. One common example is a BCD-to-7-segment decoder, which is used to convert Binary Coded Decimal (a 4-bit binary code for decimal digits) number into appropriate outputs for the segments to display the input decimal digit.

A seven-segment display is an electronic display device for displaying decimal numerals. Seven-segment displays are widely used in digital clocks, electronic meters and other electronic devices that display numerical information. A 7-Segment display digit has 7 Light Emitting Diodes (LED) that

can be lit in different combinations to represent the Arabic numerals as shown by Figure 24.a. A seven-segment display generally has 8 input connections, one for each LED segment and one that acts as a common terminal. There are 2 types of 7 Segment LED digital display as shown by Figure 24.b:

- **Common Cathode (CC) Display** – all the cathode connections of the LEDs are connected to ground. A logic '1' applied to the anode terminal of the individual segment illuminates it.
- **Common Anode (CA) Display** – all the anode connections of the LEDs are connected to VCC. A logic '0' applied to the cathode terminal of the individual segment illuminates it.



(a) 7-Segment Digit **(b) CA (bottom) and CC (top) Configurations**

Figure 32. 7-Segment Display

x_3	x_2	x_1	x_0	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1

Figure 33. Truth Table for BCD to CC 7-Segment Decoder

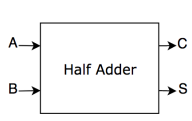
Figure 25 shows the black box of the BCD to CA 7-Segment decoder as well as the truth table of the decoder function. Please note that the outputs a to g must be inverted to use the decoder with a CC 7-Segment digit.

2.4 Priority Encoder

2.5 Full Adder

Another common and very useful combinational logic circuit which allows the addition of two or more binary numbers is the Binary Adder. There are several types of binary adders and all of them are built out of a simple building block called the Full Adder (FA). But before discussing the full adder, let's discuss a

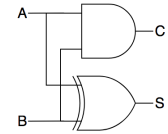
simpler building block called the Half Adder (HA). The HA is a logic circuit that adds just two bits to generate the sum and the carry out as shown by Figure 24. The HA is made out of only 2 gates, an XOR gate to generate the sum and AND gate to generate the carry out.



(a) Black Box

A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

(b) Truth Table



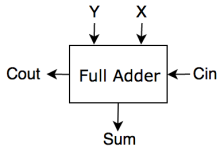
(c) Logic Circuit

Figure 34. The Half Adder

c_4	c_3	c_2	c_1		0	1	1	1	←	Carries
	x_3	x_2	x_1	x_0		0	0	1	1	
	y_3	y_2	y_1	y_0		0	1	0	1	
	s_3	s_2	s_1	s_0		1	0	0	0	← Sum

Figure 34. Adding Binary Numbers

To add two n-bit binary numbers, we use the normal everyday addition (decimal addition), except that it carries on a value of 2 instead of a value of 10. This creates a problem in the sense that, once we have added the first two bits, there may be a carry to contend with from the previous bit addition. Therefore, the HA cannot be used to realize n-bit binary adder. We need a combinational logic circuit that will not only add two bits together and produce a sum and a carry, but also can accept a carry from a previous operation and deal with it in such a way as to still produce a correct answer. This circuit is called the Full Adder (FA).



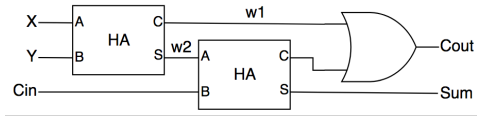


Figure 35. The Full Adder

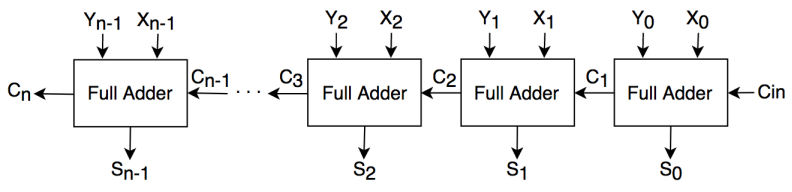


Figure 36. The Ripple Carry Adder (RCA)

The full-adder circuit, shown by Figure 35, adds three one-bit binary numbers (A, B and Cin) and outputs two one-bit binary numbers, a sum (S) and a carry out (Cout). Figure 35, shows the implementation of the FA using the HA

as a building block. This way, the FA can be used to realize n-bit binary adder that implements our everyday addition algorithm. As shown on Figure 26, n FAs are cascaded to build a parallel adder circuit that adds n-bit binary numbers. In such parallel adder, each full adder gets a Cin, which is the Cout of the previous full adder. This kind of adder is called a Ripple-Carry Adder (RCA), since each carry bit "ripples" to the next full adder.

3. Timing of Combinational Logic Circuits

As discussed in Chapter 1, a logic gate is implemented as a CMOS circuit out of PMOS and NMOS transistors. MOS transistor gates inherently have capacitance due to the dielectric insulator that separate gate polysilicon/metal from the transistor body. In addition to the gate capacitance, the transistor channel can be treated as a switch in series with a resistor. We call this the MOS RC delay model and it is illustrated in Figure 2-17.

Figure 2-17. MOSFET RC Models

To understand the effect of the gate capacitance and the channel resistance, let's examine the circuit consists of two inverters in series as illustrated in Figure 2-18. When x changes from "1" to "0", T1 switches ON and T2 switches OFF causing y to change from "0" to "1" (T1 pulls up y to VDD). When x changes form "0" to "1", T1 switches OFF and T2 switches ON causing y to switch from "1" to "0" (T2 pulls down y to GND).

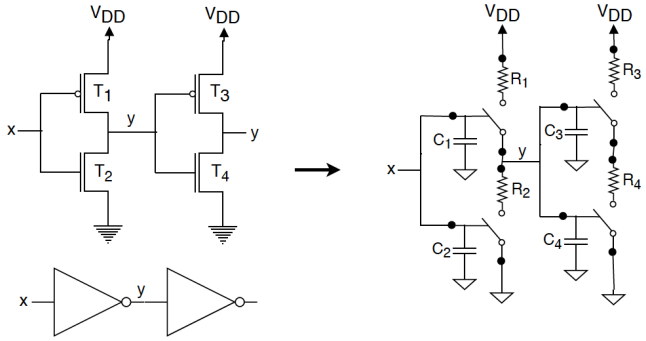


Figure 2-18. A CMOS inverter driving another inverter

Figure 2-19shows the equivalent circuits when y is "1" and when it is "0". As you can see, when T1 pulls up y (the output of the first inverter) to VDD; hence, the second inverter equivalent pin capacitance (also called the load capacitance for the first inverter or C_L) is charged through the channel resistance

of T1 (R1). Because of that, y changes from "0" (GND) to "1" (VDD) exponentially according to the equation:

$$V_{DD} \cdot (1 - e^{-t/(R_{PU} \cdot C_L)})$$

When x changes from "0" to "1", T2 pulls down y to GND; hence, the pin capacitance of the second inverter is discharged through the channel resistance of T2 (R2). Because of that, y changes from "1" (VDD) to "0" (GND) exponentially according to the equation:

$$V_{DD} \cdot e^{-t/(R_{PD} \cdot C_L)}$$

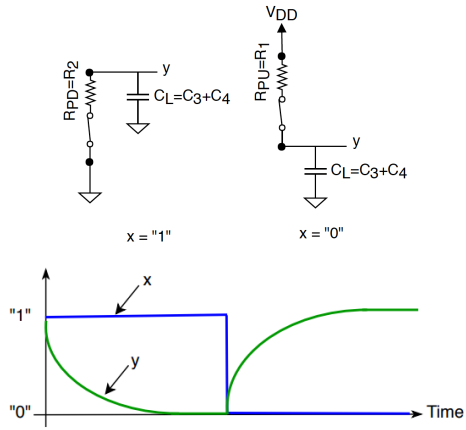


Figure 2-19. The Inverter Output Waveform

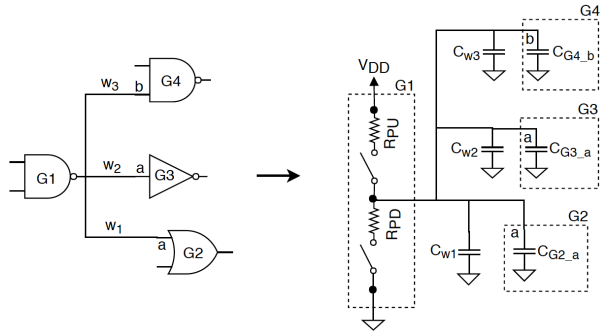


Figure 2-20. RC Delay model for a CMOS gate

To generalize, the above two equations can be used to describe the behavior of the output of any CMOS gate in a circuit where R_{PU} is the equivalent resistance of the gate pullup network, R_{PD} is the equivalent resistance of the gate pulldown network and C_L is the sum of the capacitances of all input pins connected to this output pin as well as the wire capacitances as shown on Figure 2-20. Please note that as the input of any gate is the output of another gate, the

input cannot be a perfect step. Instead it is an exponential signal which can be approximated as a ramp.

As shown by Figure 30, the delay is measured as the delay from the 50% point of the input signal to the 50% point of the output signal. The 50% point is the point at which the signal is half-way (50%) between its LOW and HIGH values as it transitions. There are two values of the delay, t_{PHL} and t_{PLH}. t_{PHL} measures the delay when the output changes from High to Low due to the change of the input. t_{PLH} reflects the delay when the output goes from Low to High. For simplification, we use a single value, t_{pd}, which is the average of t_{PHL} and t_{PLH}. Although the timing diagram on Figure 30 is very close to the reality, we usually simplify it by using sharp transitions instead of the exponential transition

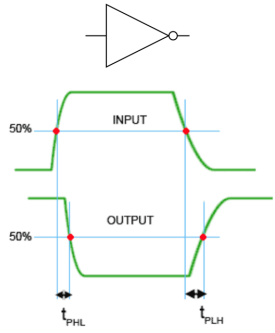


Figure 37. The Inverter Propagation Delay

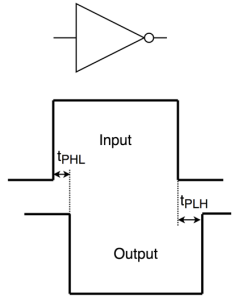


Figure 38. Approximated Timing Diagram for the Inverter

The delay of a gate is not constant and varies with several parameters such as the temperature, the fan out (number of gates connected to the output of the gate), the fan in (the number of inputs) and the supply voltage. The propagation delay increases as the temperature goes up, the fan in increase and the fan out increases. The gate becomes faster (less propagation delay) when the supply voltage increases.

3.1 Logic Circuit Delay

Calculating the propagation delay of a circuit is a tricky because the delay is not the same for all input-to-output paths. Some paths have higher delay than the others. So, for a logic circuit, the propagation delay, t_{pd}, is defined as the maximum time from when an input changes until the output or outputs reach their final value(s). Also, for a circuit, we define the contamination delay, t_{cd}, as the minimum time from when an input changes until any output starts to change its value. In other words, the propagation delay of a combinational circuit is the sum of the propagation delays through each element on the longest path, also called the critical path. The contamination delay is the sum of the contamination delays through each element on the short path. For example, for the circuit shown on Figure 31, if we assume that the t_{pd} is the same for all gates (1 delay unit) then t_{pd} of the circuit is 3 delay units. The propagation delay has to do with the longest path: a to y or b to y. The contamination delay of the circuit is 2 delay units and has to do with the shortest path: d to y or e to y.

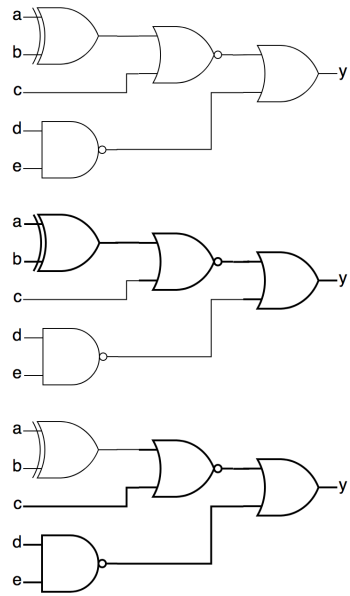


Figure 39. Propagation and Contamination Delays Example

4. Summary

5. Questions

Chapter 3
Sequential Logic

Chapter 1 In this chapter, we will learn how to design and analyze digital circuits that can store binary data and use the stored data to produce future outputs. Such circuits are called *Sequential Logic Circuits*. Sequential logic is a type of logic circuit whose output depends not only on the present value of its input signals but on the sequence of past inputs. That is, sequential logic has state (memory) and the its current output depends on the state and the current input. This is different from the *Combinational Circuits* we visited in the first two chapters. The output of a combinational circuit depends only on the current input. That is, if an input changes state, output may also change state. Hence, a combinational circuit does not need to store something.

1. Basic Storage Elements

In many applications, there are requirements for digital circuits whose outputs will remain unchanged, once set, even if there is a change in input level(s). Such circuits could be used to store a binary number. A flip-flop is one such circuit, and the characteristics of the most common types of flip-flops used in digital systems are considered in this chapter. Flip-flops are used in the construction of registers and counters, and in numerous other applications.

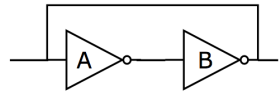
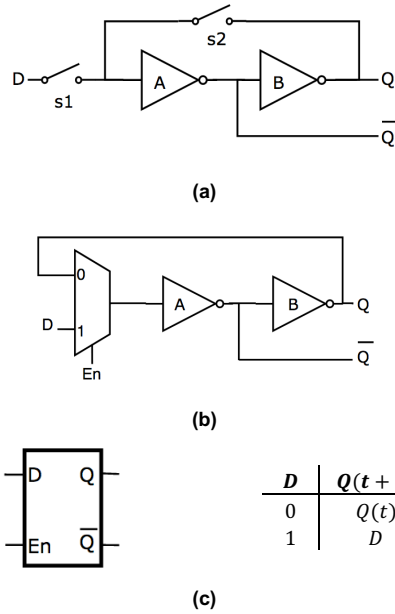


Figure 3-1. Basic Storage Element

One of the easiest ways to construct a flip-flop is to connect two inverters in series as shown in Figure 32. The line connecting the output of inverter B (INV B) back to the input of inverter A (INV A) is referred to as the feedback line. The problem of this simple circuit is that we cannot control what is stored in this circuit. This can be improved by adding 2 switches as shown by Figure 33.a. When s1 is closed and s2 is open the output, Q, follows the input, D. When we reverse the state of the 2 switches, the circuit becomes similar to that of Figure 32 and the last value on Q stays till we change the state of the switches

again. So, to write a new value s1 has to be ON and s2 has to be OFF; to store the value on Q, s1 has to be OFF and s2 must be ON. But, how to implement s1 and s2? This can be done using a 2x1 multiplexer as shown in Figure 33.b. When En is "1" (write) the output, Q, follows the input, D. When En is "0" (store), the feedback between the 2 inverters is enabled and the last data on Q is stored. This storage element is called the D latch. Sometimes, it is called the transparent latch because when En is "1" the latch becomes transparent, the output follows the input. The D latch is also known as a level-sensitive latch because the state of the output is dependent on the level of the enable signal (En).



D	$Q(t+1)$
0	$Q(t)$
1	D

Figure 3-2. The D Latch

The behavior of any storage element can be expressed using a table called the characteristic table. The characteristic table and the symbol for the D latch are given by Figure 33.c which shows, also, its graphical symbol.

1.1 Flip Flops

In the level-sensitive latches, the state of the latch keeps changing according to the values of input signals during the period when the enable signal is active. As we will see later, there is also a need for storage elements that can change their states with the edge of the enable signal, either the rising edge or the falling edge. One way to achieve this is by combining two level-sensitive latches, one negative-sensitive and one positive-sensitive, as shown on Figure 34. The first latch is called the master and the second is called the salve. For that, we call this

flip-flop the Master Slave D flip flop. Please notice the small triangle associated with the clock input. This triangle indicates that the flip flop is edge triggered. This particular flip flop is positive (rising) edge triggered. If the master latch is triggered by the clock and the slave by the inversion of the clock, the flip flop becomes a negative (falling) edge triggered flip flop. The symbol of the negative edge triggered flip flop is given by Figure 36. Please notice the little bubble (circle) next to the triangle at the clock input.

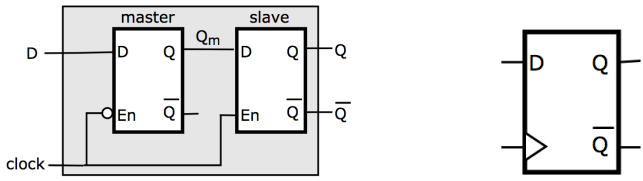


Figure 44. Master-Slave Positive Edge Triggered D Flip Flop

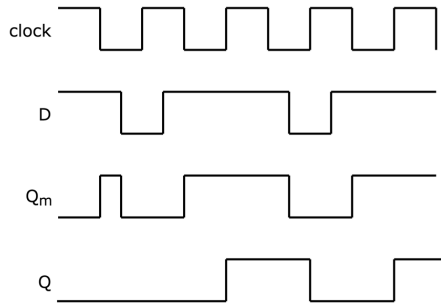


Figure 45. Timing Diagram for a positive edge triggered D Flip Flop

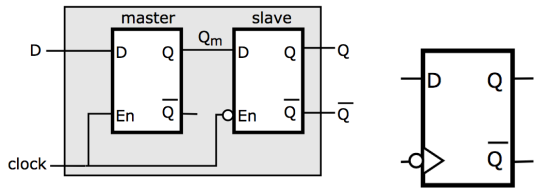
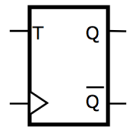


Figure 46. Negative Edge Triggered D Flip Flop Symbol

There are other types of flip flops such as T flip flop (toggle flip flop) and the *JK* flip flop. The output of the Toggle (T) flip flop, shown on Figure 37, toggles with every triggering edge if the input T is 1. If T is 0 then it does not toggle. It is useful for constructing binary counters, and frequency dividers.

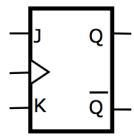
The JK flip flop, shown on Figure 38, is the most widely used of all the flip-flop designs and is considered to be a universal flip-flop circuit. The

operation of the JK flip flop is given by the characteristic table shown by Figure 38. Please note that when both the J and the K inputs are at logic level "1" at the same time, and the triggering clock edge is observed the output (Q) toggles. That means if *J* and *K* are tied together and labeled "T", we end up with a T flip flop. Also, a D flip flop can be constructed by connecting D to *J* and $\overline{D}$ to *K*.



T	$Q(t+1)$
0	$Q(t)$
1	$\overline{Q(t)}$

Figure 37. The Toggle (T) Flip Flop



J	K	$Q(t+1)$
0	0	$Q(t)$
0	1	0
1	0	1
1	1	$\overline{Q(t)}$

Figure 47. The JK Flip Flop

1.2 Asynchronous Reset and Preset

When a flip flop is first powered up, its state is not known. Thus, it is typical to add extra inputs that can force the flip flop state to '1' or '0' independent of the clock (Figure 48). Thus, they are called Asynchronous "reset" (rst) and "preset" (pre); they override all other inputs to drive the stored value to a '0' or '1' respectively without the need for the clock edge. These signals are most useful when a flip flop is first initialized after power-on, but they can be used at any time to force the output low or high regardless of the state of the clock or D signals.

The primary purpose of a reset is to force the digital system into a known state for stable operations. This would avoid the digital system to power on to a random state and get hanged. Initializing the digital system is discussed in Chapter 4.

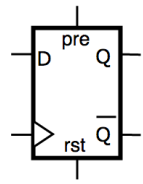


Figure 48. D Flip Flop with Asynchronous Reset and Preset Inputs

1.3 Flip-Flop Timing Parameters

Because a flip-flop changes state only on the active edge of the clock, the propagation delay of a flip flop, as illustrated by Figure 41, is the time between the active edge of the clock and the resulting change in the output. We call this tCQ (CQ: Clock to Q). However, there are also timing issues associated with the D input. To function properly, the D input to an edge-triggered flip-flop must be held at a constant value for a period of time before and after the active edge of the clock, shaded time window on Figure 39. If D changes within this time window, the behavior is unpredictable. The amount of time that D must be stable before the active edge is called the setup time (tsu), and the amount of time that D must hold the same value after the active edge is the hold time (th). The setup time allows a change in D to propagate through the first latch before the rising edge of Clock. The hold time is required so that D gets stored in the first latch before D changes.

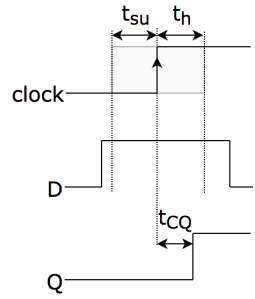


Figure 49. Flip Flop Timing Parameters

2. Sequential Circuits

In the previous section, we studied combinational logic. We have approached our study of Boolean functions by examining the variables, the values for those variables, and the function outputs that depend solely on the values of the inputs to the functions. If we change an input value, this has a direct and immediate impact on the value of the output. The major weakness of combinational circuits is that there is no concept of storage—they are memoryless. This presents us with a bit of a dilemma. We know that computers must have a way to remember values.

A sequential circuit defines its output as a function of both its current inputs and its previous inputs. Therefore, the output depends on past inputs. To remember previous inputs, sequential circuits must have some sort of storage element. We typically refer to this storage element as a flip-flop. The state of this flip-flop is a function of the previous inputs to the circuit. Therefore, pending output depends on both the current inputs and the current state of the circuit.

In the same way that combinational circuits are generalizations of gates, sequential circuits are generalizations of flip-flops.

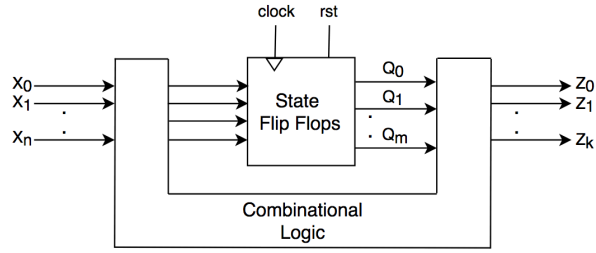


Figure 3-3. Synchronous Sequential System Model

Figure 50 is a conceptual view of a synchronous sequential system. A sequential system consists of a set of flip flops and combinational logic. This diagram depicts a system with n inputs (X_1 … X_n), in addition to the clock, k outputs (Z_1 … Z_k), and m Flip Flops.

A synchronous sequential circuit state can be affected only at discrete instants of time. The synchronization is achieved by using a timing device, termed as System Clock Generator, which generates a periodic train of clock pulses.

3. Clocks

The fact that a sequential circuit uses past inputs to determine present outputs indicates we must have event ordering. Before we discuss sequential logic, we must first introduce a way to order events. Some sequential circuits are asynchronous, which means they become active the moment any input value changes. On the other hand, synchronous sequential circuits use clock signals to order events. A clock signal (Figure 3-4) is a series of pulses with a precise pulse width and a precise interval between consecutive pulses. This interval is called the clock cycle time (or the clock period). The clock period determines how fast is the clock signal. Clock speed is, generally, called frequency and measured in Hertz (Hz), or one pulse per second. Common clock frequencies are from one to tens of hundred MHz.

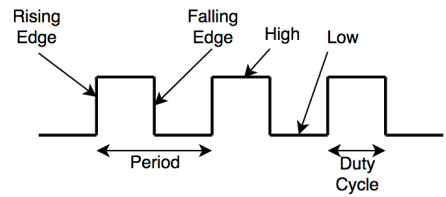


Figure 3-4. A Clock Signal

A clock is used by a sequential circuit to decide when to update the state of the circuit (when do "present" inputs become "past" inputs?). This means that inputs to the circuit can only affect the storage element at given, discrete instances of time. In this chapter, we examine synchronous sequential circuits because they are easier to understand than their asynchronous counterparts. From this point, when we refer to "sequential circuit," we are implying "synchronous sequential circuit." Most sequential circuits are edge-triggered (as opposed to being level-triggered). This means they are allowed to change their states on either the rising or falling edge of the clock signal, as seen on Figure 3-4

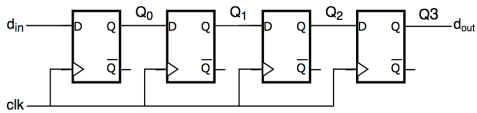
4. Registers

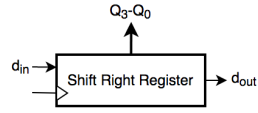
Registers are groups of flip flops, where each flip flop is capable of storing one bit of information. An n-bit register is a group of *n* flipflops. The basic function of a register is to store information in a digital system and make it available to the logic elements for the computing process. n-bit register is capable of storing one of 2n 0-1 combinations. Each of those combinations is known as state or content of the register. Data can be loaded into a register serially or parallelly.

4.1 Shift Registers

Serial-load registers are called shift registers. As shown on Figure 52, a shift register has one serial input (d_{in}). Every clock cycle one bit is loaded from serial in into the first flip-flop of the register while all the actual flip-flop contents are shifted to the next flip-flop, dropping the last bit. The shift register shown on Figure 52 is called shift right register as the content of the register is shifted one bit to the right every clock cycle. A shift register, also, features a serial output (d_{out}) so that the last bit that gets shifted out of the register can be processed further. For that, this register is called Serial In – Serial Out (SISO). With SISO register, it is possible to build up a chain of shift registers by connecting each serial out to another shift register's serial in, effectively creating a single big shift register. It is also possible to create a Cyclic register by connecting the serial out to the same register's serial in. If the output of all flip-flops (and therefore the register's complete content) are read from the lines Q_1 to Q_n the register is used as Serial In – Parallel Out (SIPO). A typical purpose for such a SIPO register is to collect data that is delivered bitwise and that is needed in *n*-bit data words which is case for serial communication (used by your computer serial port or USB) as shown by Figure 53. The SIPO is used by the USB receiver to receive the data from the communication channel one bit every clock cycle and convert them into one word (deserialization). For that the SIPO is usually called deserializer.

Shifting bits are important for mathematical operations: if the content of the register is interpreted as a binary number, shifting by one bit corresponds to multiplying or dividing by *2*. Shifting the number bits from left to right (most to least) by one bit is equivalent to dividing the number by *2*. This is operation is called shift right. Hence, shifting a number *n*-bits to the right is equivalent to dividing the number by 2^n. If the shift is done from least to most (shift left) for n bits then the number is multiplied by 2^n. For example, shifting 00110_2 (6) to the left by 2 bits yields 11000_2 (6x4=24). Shifting it to the right by one bit yields 00011_2 (6/2=3).





Cycle	Q3	Q2	Q1	Q0	d_{in}
0	0	0	0	0	1
1	0	0	0	1	0
2	0	0	1	0	1
3	0	1	0	1	1
4	1	0	1	1	1
5	0	1	1	1	1
6	1	1	1	1	0
7	1	1	1	0	1
8	1	1	0	1	0

Figure 52. Shift Right Register

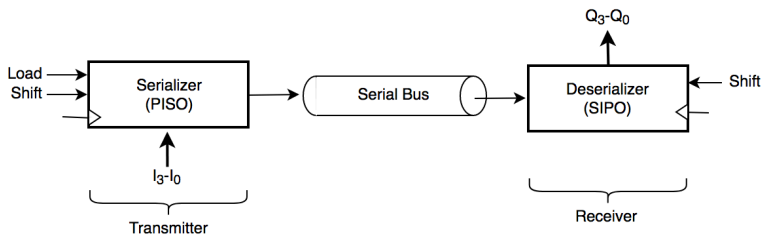


Figure 53. Serialization and Deserialization

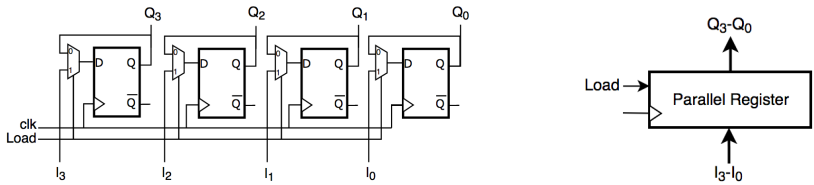


Figure 54. Parallel Load Register

4.2 Parallel Load Register

It is n-bit register that can be loaded with *n* bits in one clock cycle. This is faster than the shift register that needs *n* clock cycles to load *n* bits. Figure 54 shows a

4-bit parallel load register which can be loaded into the 4 flip flops by the input bits (I_3-I_0) when Load is active and the positive edge is observed. When Load is active the 4 multiplexers connect I_3-I_0 to the flip flops inputs. When Load is inactive the multiplexers connect the outputs of the flip flops to their inputs. Hence, the flip flops keep their contents. This register is called Parallel In – Parallel Out (PIPO) register. The design on Figure 54 can be updated to enable shifting the content of the register by using a larger multiplexer and connecting the inputs and the outputs of the flip flops as we did in the shift register (Figure 52). This resultant register is capable of loading the data in parallel and the data can be read out serially. This register is called Parallel In – Serial Out (PISO) register. PISO register is used by the USB transmitter to serialize the data to be sent over the serial bus. For that, PISO is called serializer (Figure 53).

4.3 Counters

A counter is simply a register with combinational logic to implement counting, that is it is possible to retrieve the contents, add or subtract one to the contents, and then store it back into the register in one operation. However, this description implies the usage of an adder or a subtractor to realize an up or down counter, we will see that adding 1 or subtracting 1 can be done using only few gates.

There are different types of counters. The simplest is the binary counter. N-bit binary counter counts 2^n counts. The binary counter could be an up counter or a down counter. Figure 55 shows the contents of a three-bit binary up-counter for eight consecutive clock cycles, assuming that the count is initially 0. Observing the pattern of bits in each row of the table, it is apparent that bit Q_0 changes on each clock cycle. Bit Q_1 changes only when Q is equal to 1. Bit Q_2 changes only when both Q_1 and Q_0 are equal to 1. This idea is used to implement a 4-bit binary up counter shown on Figure 56. It is obvious that implementing a down counter is possible by using $\overline{Q}$ instead of Q. So, Q_1 toggles only when $Q_0 = 0$ or $\overline{Q_0} = 1$.

A binary counter circuit can be constructed to support up counting, down counting and parallel loading. Figure 46 shows the symbol used for such counter. It up counts when $count$ is 1 and $Up/\overline{Down}$ is 1. When $count$ is 1 and $Up/\overline{Down}$ is 0 it down counts. If $count$ is 0, it does not count. This counter can be used as a building block for other types of counters. For example, a modulo-6 counter, a counter that counts 6 counts 0, 1, 2, 3, 4 and 5 only, can be constructed using the binary counter of Figure 57. Figure 58 shows the modulo-6 counter. The AND gate is used to detect the terminal count (5); once detected, the initial count (0) is loaded into the counter. Please note that the AND gate detects any count that has Q_0=Q_2=1 (5, 7, 13, and 15). As the counter starts from 0 the first count it hits out of these 4 counts is 5.

Cycle	Q2	Q1	Q0	
0	0	0	0	
1	0	0	1	
2	0	1	0	Q1 changes
3	0	1	1	
4	1	0	0	Q1 and Q2 change
5	1	0	1	
6	1	1	0	Q1 changes
7	1	1	1	
8	0	0	0	Q1 and Q2 change

Figure 55. Counting Sequence for 3-bit Binary Up Counter

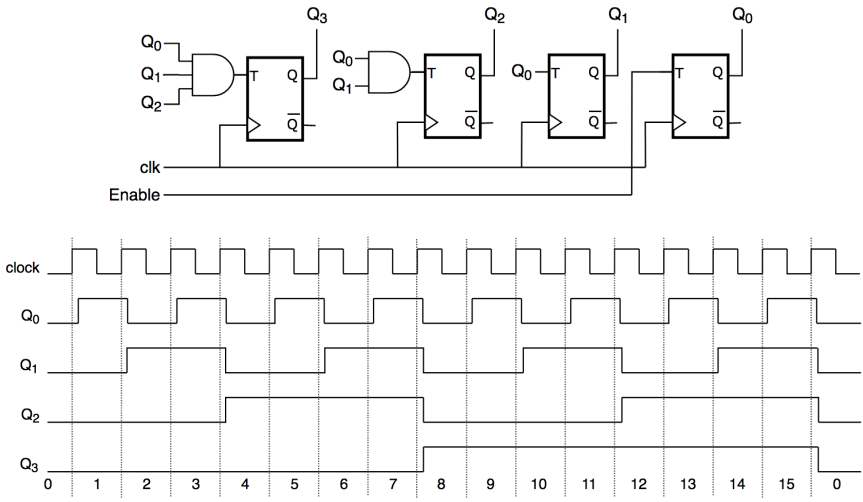


Figure 56. Binary Up Counter

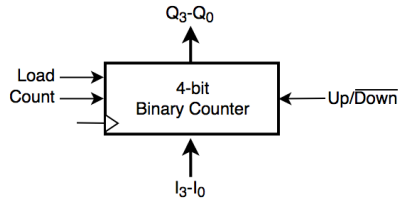


Figure 57. Symbol for Binary Counter with Parallel Load

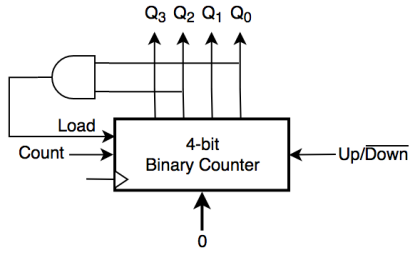


Figure 58. Modulo-6 Counter

5. Timing of Sequential Circuits

Selecting the clock frequency for a sequential circuit is a very important design activity. For some designs the clock frequency is given and cannot be changed because it has to do with a standard or a hard requirement. For example, USB 2.0 transceiver uses 480MHz clock in the high-speed mode to achieve 480Mbps. This clock frequency is fixed and cannot be reduced or increased. Otherwise, the transceiver will not be able to communicate with USB 2.0 devices. In other designs, such as microprocessors, it is desirable to use the highest possible clock frequency. In both cases, the designer should analyze the circuit to ensure that for a given clock frequency, the flip flops will never be metastable by satisfying their timing constraints.

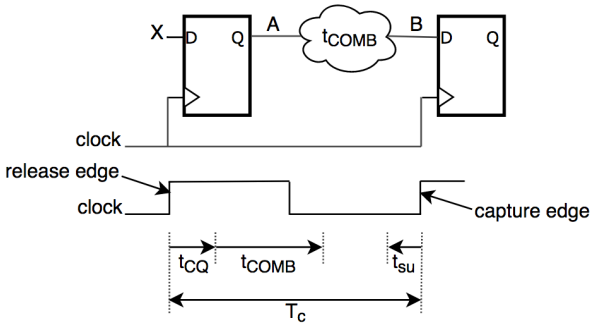


Figure 59. Setup Time Analysis without Clock Skewing

5.1 Setup Time and Hold Time Requirements Analysis

Figure 59 shows a section of a sequential circuit where a combinational logic block with a propagation delay of t_{COMB} is sandwiched between two flipflops. Before the rising edge of the clock (release edge), the valid data that meets the setup and hold time requirements is introduced at the X terminal. After t_{CQ} delay (of the left flip flop), the data emerges at the node A and propagates through the combinational logic block as shown in the timing diagram. The data must arrive at node B before the next clock edge (capture

edge) by at least the setup time (t_{su}) of the capturing (right) flip flop. If the data arrives at the node B too late and violates the allocated setup time of flip-flop. This is called the setup violation. Hence in order not to have a setup violation at the right flip flop, the clock period (T_c) has to satisfy the following:

$$T_c \geq t_{CQ} + t_{COMB} + t_{su}$$

For maximum clock frequency ($f_c = {}^{1}/_{T_c}$), both sides must be equal. The difference between the left-hand side and the right side is called the slack. A negative slack means a setup violation at the capturing flip flop.

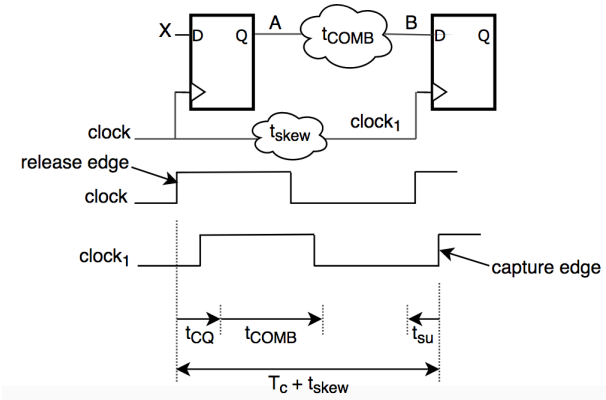


Figure 3-5. Setup Time Analysis with Clock Skewing

In real digital systems, it is very hard to deliver the clock to all flip flops at the same time due to several reasons. In this case, the clock is said to be skewed. Figure 3-5 shows the same example outlined by Figure 59 but after introducing the clock skewing. The skew is represented by a delay on the clock line (t_{skew}) that causes the clock to arrive at the releasing flip flop before the capturing flip flop. The condition for meeting the setup time requirement for the capturing flip flop has to incorporate t_{skew} as follows:

$$T_c + t_{skew} \geq t_{CQ} + t_{COMB} + t_{su}$$
$$T_c \geq t_{CQ} + t_{COMB} + t_{su} - t_{skew}$$

This positive skew is called a useful skew and it allows the system to use a shorter clock period which means a faster clock frequency. In other cases, the clock edge may arrive at the capturing flip flop before arriving at the release flip flop. In this case, t_{skew} is negative and the system requires a longer clock period to meet the setup time requirement of the capturing flip flop.

The positive skew is useful for meeting the setup time requirement as discussed earlier; however, it may cause a violation of the hold time (t_h) requirement. If $t_{CQ} + t_{COMB} < t_h + t_{skew}$ new data released by a clock edge overwrites the data released by the previous clock edge before satisfying the hold

time constraint. Hence, to avoid hold time violation, the following condition has to be satisfied:

$$t_{CQ} + t_{COMB} \geq t_h + t_{skew}$$

Hold time requirement violation may occur for timing paths with no combinational delay (t_{COMB}=0) and/or the clock is positively skewed (t_{skew}>0). The first condition is satisfied in many sequential circuits (e.g., a shift register). For such circuits, t_{skew} must be zero or negative to avoid hold violations. Satisfying the hold time and setup time requirements is tricky specially when the clock is skewed. Clock skew has to do with the clock distribution network that we call the *Clock Tree*. Usually, we design the clock tree to be balanced with zero skew which is very hard task and usually not achieved. Hence, it is the job of the digital designer to check every timing path in the design to make sure that all timing requirements are satisfied. This task is called the *Static Timing Analysis* (STA).

5.2 Timing Paths

In the previous section, we used the phrase *Timing Path* to refer to the path taken by the data released by one flipflop (release FF), due to the clock edge, till it reaches the input of the flipflop that captures it with the following clock edge. This is just one type of timing paths. In general, a timing path is defined as the data path between a start point and an end point in a logic circuit:

- **Start Point:** All input ports or Q pins of a sequential elements are considered as valid start point.
- **End Point:** All output port or D pins of sequential elements is considered as end point.

For that, there are four types of timing paths:

- Input to Register (In to Reg)
- Input to Output (In to Out)
- Register to Register (Reg to Reg)
- Register to Output (Reg to Out)

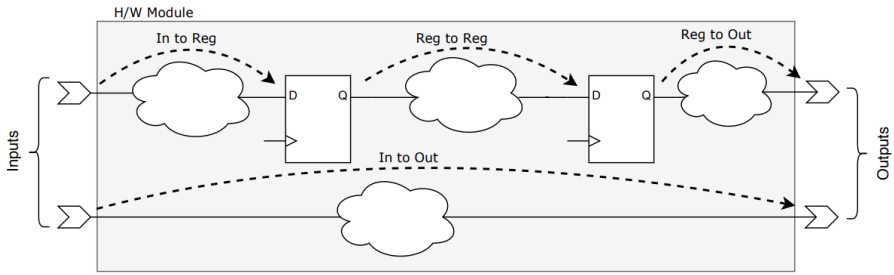


Figure 3-6. Types of Timing Paths

In order to analyze timing paths other than Register to Register, the arrival times of the data at the input ports (relative to the clock edge that releases the data; also called the input delay) and the required time for the data at the output ports (relative to the clock edge that captures that data; also called the output delay) must be known.

6. Finite State Machines

As Sequential circuits output depends not only on current inputs, but also previous inputs. Thus, sequential circuits require memory elements. Finite State Machines (FSMs) are mathematical abstractions of sequential circuits. FSM is a system comprising states, inputs and outputs. FSM models time as discrete instants (clock edges) at which input or output can change. The FSM can change from one state to another in response to some external inputs; the change from one state to another is called a transition. An FSM is defined by a list of its states, its initial state, and the conditions for each transition. A FSM can generate output based on a given input and/or a state. There are two types of FSMs based on how the output is generated:

- Moore FSM: the current output is a function of the current state only.
- Mealy FSM: the current output is a function of the current state and the current input.

The model for a FSM is given by Figure 3-7. The current state/present state of the circuit is stored in the state register. The next state and the output are generated using combinational logic. The next state of the system is determined by the present state (current state) and by the inputs. The output of the FSM is determined by the present state of the machine and/or by the inputs.

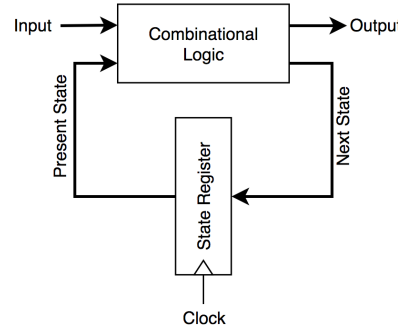
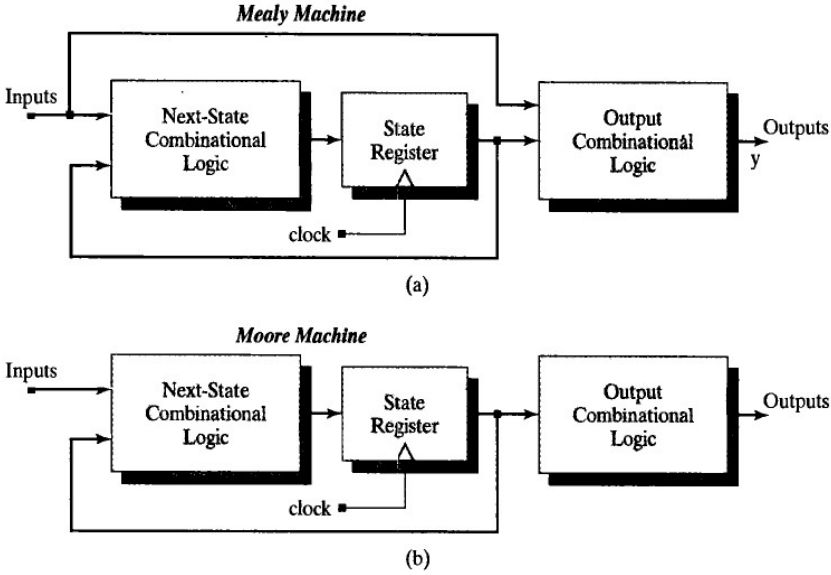


Figure 3-7. FSM Building Blocks

In general, a FSM is defined by:

- Set of p states: $S = \{S_0, S_1, S_2, \dots, S_{p-1}\}$
- Set of n inputs: $X = \{X_0, X_1, X_2, \dots, X_{n-1}\}$

- Set of m outputs: $Z = \{Z_0, Z_1, Z_2, \dots, Z_{m-1}\}$
- A function that determines the next state: $S(t+1) = f(S(t), X(t))$
- A function that determines the output:
  - $Z(t) = g(S(t))$: Moore
  - $Z(t) = g(S(t), X(t))$: Mealy



6.1 State Diagram and State Table

A state transition diagram (or just state diagram) is an abstract diagrammatic representation of a finite-state machine. It uses a circle, or "bubble," to represent each state (figure 62). Directed arcs between state bubbles represent transitions from one state to another. The reset/initial state is identified by using a double wall. An arc may be labeled with a combination of input values that allow the transition to occur. The output of the FSM is either associated with the state (Moore machine) or the transition (Mealy machine).

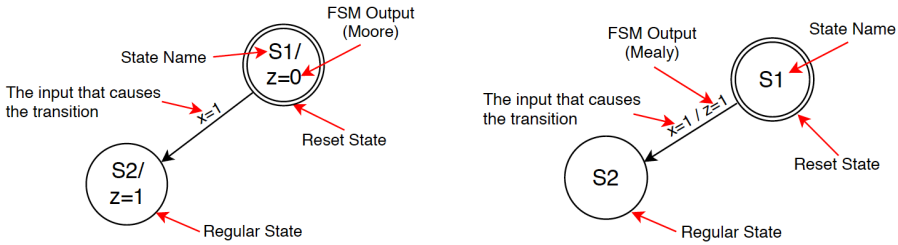


Figure 3-8. The elements of the state diagram for Mealy and Moore FSM

Figure 3-9 shows a complete state transition diagram for a Mealy FSM, with a single input (x) and a single output (y), that detects the sequence 0101 (from left to right) on the input. Once detected the output is 1 for one clock cycle. Also, the FSM allows overlapping. The following shows a sample sequence and the corresponding output.

X	0	0	1	0	1	0	1	1	0
Z	0	0	0	0	1	0	1	0	0

The FSM has 4 states A, B, C and D. Every state has two outgoing arcs for two transitions to two next states. Each arc is labeled with the value of the input (x) that causes the transition and the associated output (z). For example, if the input is 1 while in state B, the next state is C and the output is 0. Figure 3-10 shows the state diagram for the Moore machine for the same sequence detector.

State table is a tabular way to represent the behavior of an FSM. It is similar to a truth table, but it also includes the FSM current state as an input and its next state as an output.

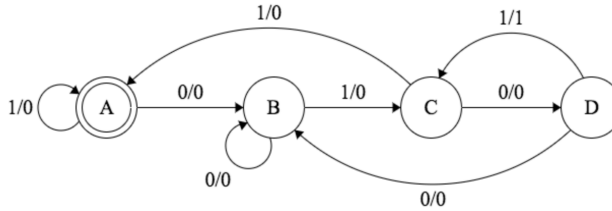


Figure 3-9. State Diagram for 0101 Mealy Sequence Detector

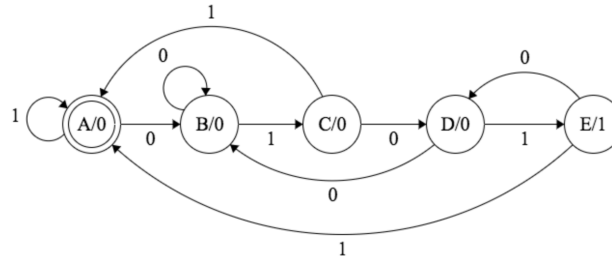


Figure 3-10. State Diagram for 0101 Moore Sequence Detector

Table 5.

Present State	Next State		Output	
	x=0	x=1	x=0	x=1
A	B	A	0	0
B	B	C	0	0
C	D	A	0	0
D	B	C	0	1

Table 6.

Present State	Next State		Output
	x=0	x=1	
A	B	A	0
B	B	C	0
C	D	A	0
D	B	E	0
E	D	A	1

6.2 State Encoding

If a state machine has n states, we need a minimum of log2(n) bits to hold the current state. This is called binary encoding. Another way to encode the FSM states is the One-Hot-Encoding (OHE). OHE uses one state flip-flop for each state bit. A state machine with 16 states requires 16 state flip-flops for one- hot encoding. One-hot encoded FSM is larger compared to binary encoding as it requires more state flip-flops. However, one-hot encoding is faster compared to binary encoding, as only one state bit is used to describe state, thus a simpler logic as compared to binary encoding that requires larger logic to decode the states.

6.3 State Assignment

State assignment is the process of assigning a unique bit string to each state represented in a description of a controller circuit. An example of state assignment was given in the rightmost state diagram of Fig. 2.10 where each state was labeled by a unique bit string rather than a symbolic symbol as is shown in the ASM chart example in Fig. 2.8, the Mealy state diagram in Fig. 2.9, and the leftmost Moore machine state diagram in Fig. 2.10. Before a controller can be implemented, the symbolic symbols representing each state must be transformed into a unique set of identifying bit strings. For each bit in these binary-valued state names, a memory element is synthesized.

While any arbitrary assignment can be made to a state machine as long as each state has a unique bit string, the exact assignment will affect the resulting synthesized logic in terms of both area and maximum clock frequency. The theory of optimal state assignment has been studied for many years and this problem has been proven to be intractable, which is to say that no existing method is known for finding the best assignment that does not require runtime proportional to trying all possibilities. Furthermore, the state assignment that yields the least amount of circuitry is likely not the same as the state assignment that allows the circuit to be clocked as fast as possible, so tradeoffs are present. The minimum number of bits possible in a state encoding for a controller with N states is _log2(N)_;

Adjacent state assignment is the name for a class of heuristics that assumes two-level logic in sum-of-products form. The idea behind all such heuristics

essentially is to lower the number of product terms and the number of literals by assigning states that have similar entries in the state table codes that differ in a single bit. Put differently, they make similar states adjacent in the Karnaugh map. Somewhat surprisingly, this approach has been found to have beneficial effects on multi-level logic implementations too and is, therefore, quite popular.

6.4 State Reduction Techniques

In design of sequential logic circuit state reduction techniques play an important role, more so for complex problems. By reducing or minimizing the total number of states, the number of flip-flops required for a design is also reduced. For example, if a finite state machine drops from 8 states to 4 states, only two flip-flops are required rather than three when using binary encoding (4 FFs instead of 8 FFs in case of OHE). The total number of states is reduced by eliminating the equivalent states. Two states are equivalent if they have the same output for all inputs, and if they transition to equivalent states on all inputs.

In this section, we explain two state reduction techniques, row elimination method and implication table method.

7. Summary

This chapter has concentrated on the optimization of finite state machines. We have emphasized the methods for state reduction, state assignment, choice of flip-flops, and state machine partitioning. For state reduction, we introduced the row matching and implication chart methods. These can be used to identify and eliminate redundant states, thus reducing the number of flip-flops needed to implement a particular finite state machine.

We then examined heuristic methods for state assignment, aimed at reducing the number of product terms or literals needed to implement the next-state and output functions. Since paper-and-pencil methods are not particularly effective, we introduced computer-aided design tools for state assignment that do a much better job in a fraction of the time: ***nova***, ***mustang***, and ***jedi***.

The latter part of the chapter focused on choosing flip-flops for implementing the state registers of the finite state machine. *J-K* flip-flops tend to be most effective in reducing the logic, but they require logical remapping of the next-state functions and more wires than the simpler *D* flip-flops.

Finally, we discussed state machine partitioning methods, in particular partitioning based on inputs and outputs and partitioning by introducing idle states. These techniques are needed when we cannot implement a finite state machine with a single programmable logic component.

In the next chapter, we will examine implementation strategies in more detail. In particular, we will look at the methods for implementing finite state machines

based on structured logic methods, such as ROM, programmable logic, and approaches based on MSI components.