Number Systems and Codes

5

OBJECTIVES

- Convert decimal numbers to binary and convert binary numbers to decimal
- Convert binary and decimal numbers to octal and convert octal numbers to binary and decimal
- Convert binary and decimal numbers to hexadecimal and convert hexadecimal numbers to binary and decimal
- Describe the ASCII code, excess-3 code, and Gray code
- Understand Error Detection and Correction Code

5.1 BINARY NUMBER SYSTEM

The *binary number system* is a system that uses only the digits 0 and 1 as codes. All other digits (2 to 9) are thrown away. To represent decimal numbers and letters of the alphabet with the binary code, you have to use different strings of binary digits for each number or letter. The idea is similar to the Morse code, where strings of dots and dashes are used to code all numbers and letters. What follows is a discussion of decimal and binary counting.

Decimal Odometer

To understand how to count with binary numbers, it helps to review how an odometer (miles indicator of a car) counts with decimal numbers. When a car is new, its odometer starts with

00000

After 1 km the reading becomes

00001

Successive kms produce 00002, 00003, and so on, up to

00009

A familiar thing happens at the end of the tenth km. When the units wheel turns from 9 back to 0, a tab on this wheel forces the tens wheel to advance by 1. This is why the numbers change to

00010

Reset-and-Carry

The units wheel has reset to 0 and sent a carry to the tens wheel. Let's call this familiar action *reset and carry*. The other wheels of an odometer also reset and carry. For instance, after 999 kms the odometer shows

00999

What does the next km do? The units wheel resets and carries, the tens wheel resets and carries, the hundreds wheel resets and carries, and the thousands wheel advances by 1, to get

01000

Binary Odometer

Visualize a binary odometer as a device whose wheels have only two digits, 0 and 1. When each wheel turns, it displays 0, then 1, then back to 0, and the cycle repeats. A four-digit binary odometer starts with

0000 (zero)

After 1 mile, it indicates

0001 (one)

The next mile forces the units wheel to reset and carry, so the numbers change to

0010 (two)

The third mile results in

0011 (three)

After 4 miles, the units wheel resets and carries, the second wheel resets and carries, and the third wheel advances by 1:

0100 (four)

Table 5.1 shows all the binary numbers from 0000 to 1111, equivalent to decimal 0 to 15. Study this table carefully and practice counting from 0000 to 1111 until you can do it easily. Why? Because all kinds of logic circuits are based on counting from 0000 to 1111.

The word *bit* is the abbreviation for binary digit. Table 5.1 is a list of 4-bit number from 0000 to 1111. When a binary number has 4 bits, it is sometimes called a *nibble*. Table 5.1 shows 16 nibbles (0000 to 1111). A binary number with 8 bits is known as a *byte*; this has become the basic unit of data used in computers. You will learn

Table 5.1 4-Digit Binary Numbers

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

more about bits, nibbles, and bytes in later chapters. For now memorise these definitions:

$$\text{bit} = X$$
$$\text{nibble} = XXXX$$
$$\text{byte} = XXXXXXXX$$

where the X may be a 0 or a 1.

SELF-TEST

1. What is the binary number for decimal 13?
2. What is the decimal equivalent of binary 1001?
3. How many binary digits (bits) are required to represent decimal 15?

5.2 BINARY-TO-DECIMAL CONVERSION

Table 5.1 lists the binary numbers from 0000 to 1111. But how do you convert larger binary numbers into their decimal values? For instance, what does binary 101001 represent in decimal numbers? This section shows how to convert a binary number quickly and easily into its decimal equivalent.

Positional Notation and Weights

We can express any decimal *integer* (a whole number) in units, tens, hundreds, thousands, and so on. For instance, decimal number 2945 may be written as

$$2945 = 2000 + 900 + 40 + 5$$

In powers of 10, this becomes

$$2945 = 2(10^3) + 9(10^2) + 4(10^1) + 5(10^0)$$

The decimal number system is an example of *positional notation*, each digit position has a *weight* or value. With decimal numbers, the weights are units, tens, hundreds, thousands, and so on. The sum of all the digits multiplied by their weights gives the total amount being represented. In the foregoing example, the 2 is multiplied by a weight of 1000, the 9 by a weight of 100, the 4 by a weight of 10, and the 5 by a weight of 1; the total is

$$2000 + 900 + 40 + 5 = 2945$$

Binary Weights

In a similar way, we can rewrite any binary number in terms of weights. For instance, binary number 111 becomes

$$111 = 100 + 10 + 1 \tag{5.1}$$

In decimal numbers, this may be rewritten as

$$7 = 4 + 2 + 1 \tag{5.2}$$

Writing a binary number as shown in Eq. (5.1) is the same as splitting its decimal equivalent into units, 2s, and 4s as indicated by Eq. (5.2). In other words, each digit position in a binary number has a weight. The least

significant digit (the one on the right) has a weight of 1. The second position from the right has a weight of 2; the next, 4; and then 8, 16, 32, and so forth. These weights are in ascending powers of 2; therefore, we can write the foregoing equation as

$$7 = 1(2^2) + 1(2^1) + 1(2^0)$$

Whenever you look at a binary number, you can find its decimal equivalent as follows:

1. When there is a 1 in a digit position, add the weight of that position.
2. When there is a 0 in a digit position, disregard the weight of that position. For example, binary number 101 has a decimal equivalent of

$$4 + 0 + 1 = 5$$

Table 5.2 Binary System

Bit Position	*Weight*
1 (Right most)	1
2	2
3	4
4	8
5	16
6	32
7	64
8	128

As another example, binary number 1101 is equivalent to

$$8 + 4 + 0 + 1 = 13$$

Still another example is 11001, which is equivalent to

$$16 + 8 + 0 + 0 + 1 = 25$$

Streamlined Method

We can streamline binary-to-decimal conversion by the following procedure:

1. Write the binary number.
2. Directly under the binary number write 1, 2, 4, 8, 16 ... , working from right to left.
3. If a zero appears in a digit position, cross out the decimal weight for that position.
4. Add the remaining weights to obtain the decimal equivalent.

As an example of this approach, let us convert binary 101 to its decimal equivalent:

STEP 1 1 0 1
STEP 2 4 2 1
STEP 3 4 2 1
STEP 4 4 + 1 = 5

As another example, notice how quickly 10101 is converted to its decimal equivalent:

1 0 1 0 1
16 8 4 2 1 → 21

Fractions

So far, we have discussed binary *integers* (whole numbers). How are binary fractions converted into corresponding decimal equivalents? For instance, what is the decimal equivalent of 0.101? In this case, the weights of digit positions to the right of the binary point are given by $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, $\frac{1}{16}$, and so on. In powers of 2, the weights are

$$2^{-1} \quad 2^{-2} \quad 2^{-3} \quad 2^{-4} \quad \text{etc.}$$

or in decimal form:

0.5 0.25 0.125 0.0625 etc.

Here is an example. Binary fraction 0.101 has a decimal equivalent of

0.1 0 1

0.5 + 0 + 0.125 = 0625

Another example, the decimal equivalent of 0.1101 is

0.1 1 0 1

0.5 + 0.25 + 0 + 0.0625 = 0.8125

Mixed Numbers

For *mixed* numbers (numbers with an integer and a fractional part), handle each part according to the rules just developed. The weights for a mixed number are

etc. 2^3 2^2 2^1 2^0 . 2^{-1} 2^{-2} 2^{-3} etc.

↑

Binary point

Table 5.3 Powers of 2

Powers of 2	*Decimal Equivalent*	*Abbreviation*
2^0	1	
2^1	2	
2^2	4	
2^3	8	
2^4	16	
2^5	32	
2^6	64	
2^7	128	
2^8	256	
2^9	512	
2^{10}	1,024	1K
2^{11}	2,048	2K
2^{12}	4,096	4K
2^{13}	8,192	8K
2^{14}	16,384	16K
2^{15}	32,768	32K
2^{16}	65,536	64K
2^{17}	131,072	128K
2^{18}	262,144	256K
2^{19}	524,288	512K
2^{20}	1,048,576	1,024K = 1M
2^{21}	2,097,152	2,048K = 2M
2^{22}	4,194,304	4,096K = 4M

For future reference, Table 5.3 lists powers of 2 and their decimal equivalents and the numbers of K and M. The abbreviation K stands for 1024. Therefore, 1K means 1024. 2K stands for 2048, 4K represents 4096, and so on. The abbreviation M stands for 1,048,576, which is equivalent to 1024K (1024 × 1024 = 1,048,576). A memory chip that stores 4096 bits is called a "4K memory." A digital device might have a memory capacity of 4,194,304 bytes. This would be referred to as a "4-megabyte (Mb) memory."

Example 5.1 Convert binary 110.001 to a decimal number.

Solution

1	1	0	0	0	1
4	2	~~1~~	~~0.5~~	~~0.25~~	0.125 → 6.125

Example 5.2 What is the decimal value of binary 1011.11?

Solution

1	0	1	1	1	1
8	~~4~~	2	1	0.5	0.25 → 11.75

Example 5.3 A computer has a 2 Mb memory. What is the decimal equivalent of 2 Mb?

Solution

$$2 \times 1{,}048{,}576 = 2{,}097{,}152$$

This means that the computer can store 2,097,152 bytes in its memory.

SELF-TEST

4. What is the decimal equivalent of 10010?
5. What is the binary equivalent of 35?
6. A binary number has 9 bits. What is the binary weight of the most significant bit?

5.3 DECIMAL-TO-BINARY CONVERSION

One way to convert a decimal number into its binary equivalent is to reverse the process described in the preceding section. For instance, suppose that you want to convert decimal 9 into the corresponding binary number. All you need to do is express 9 as a sum of powers of 2, and then write 1s and 0s in the appropriate digit positions:

$$9 = 8 + 0 + 0 + 1$$
$$\rightarrow 1001$$

As another example:

$$25 = 16 + 8 + 0 + 0 + 1$$
$$\rightarrow 11001$$

Double Dabble

A popular way to convert decimal numbers to binary numbers is the *double-dabble method.* In the double-dabble method you progressively divide the decimal number by 2, writing down the remainder after each division. The remainders, taken in reverse order, form the binary number. The best way to understand the method is to go through an example step by step. Here is how to convert decimal 13 to its binary equivalent

Step 1 Divide 13 by 2, writing your work like this:

$$\begin{array}{r} 6 \\ 2\overline{)13} \end{array} \qquad 1 \rightarrow \text{(first remainder)}$$

The quotient is 6 with a remainder of 1.

Step 2 Divide 6 by 2 to get

$$\begin{array}{r} 3 \\ 2\overline{)6} \end{array} \qquad 0 \rightarrow \text{(second remainder)}$$
$$2\overline{)13} \qquad 1 \rightarrow \text{(first remainder)}$$

This division gives 3 with a remainder of 0.

Step 3 Again you divide by 2:

$$\begin{array}{ll} 2\overline{)3} & 0 \rightarrow \text{(second remainder)} \\ 2\overline{)6} & 1 \rightarrow \text{(first remainder)} \\ 2\overline{)13} & 1 \rightarrow \text{(first remainder)} \end{array}$$

Here you get a quotient of 1 with a remainder of 1.

Step 4 One more division gives

$$\begin{array}{ll} 2\overline{)1} & 1 \\ 2\overline{)3} & 1 \\ 2\overline{)6} & 0 \\ 2\overline{)13} & 1 \end{array} \quad \begin{array}{l} \mapsto \text{(fourth remainder)} \\ \\ \text{Read down} \\ \downarrow \end{array}$$

In this final division 2 does not divide into 1; thus, the quotient is 0 with a remainder of 1.

Whenever you arrive at a quotient of 0 with a remainder of 1, the conversion is finished. The remainders when read downward give the binary equivalent. In this example, binary 1101 is equivalent to decimal 13.

There is no need to keep writing down 2 before each division because you are always dividing by 2. Here is an efficient way to show the conversion of decimal 13 to its binary equivalent:

$$\begin{array}{ll} 0 & 1 \\ 1 & 1 \\ 3 & 0 \\ 6 & 1 \\ 2\overline{)13} & \end{array} \quad \begin{array}{l} \\ \\ \text{Read down} \\ \downarrow \end{array}$$

Fractions

As far as fractions are concerned, you *multiply* by 2 and record a carry in the integer position. The carries read downward are the binary fraction. As an example, 0.85 converts to binary as follows:

$$\begin{array}{l} 0.85 \times 2 = 1.7 = 0.7 \text{ with a carry of } 1 \\ 0.7 \times 2 = 1.4 = 0.4 \text{ with a carry of } 1 \\ 0.4 \times 2 = 0.8 = 0.8 \text{ with a carry of } 0 \\ 0.8 \times 2 = 1.6 = 0.6 \text{ with a carry of } 1 \\ 0.6 \times 2 = 1.2 = 0.2 \text{ with a carry of } 1 \\ 0.2 \times 2 = 0.4 = 0.4 \text{ with a carry of } 0 \end{array} \quad \begin{array}{l} \\ \\ \text{Read down} \\ \\ \\ \downarrow \end{array}$$

Reading the carries downward gives binary fraction 0.110110. In this case, we stopped the conversion process after getting six binary digits. Because of this, the answer is an approximation. If more accuracy is needed, continue multiplying by 2 until you have as many digits as necessary for your application.

Useful Equivalents

Table 5.4 shows some decimal-binary equivalences. This will be useful in the future. The table has an important property that you should be aware of. Whenever a binary number has all 1s (consists of only 1s), you can find its decimal equivalent with this formula:

$$\text{Decimal} = 2^{n-1}$$

where n is the number of bits. For instance, 1111 has 4 bits; therefore, its decimal equivalent is

$$\text{Decimal} = 2^4 - 1 = 16 - 1 = 15$$

Table 5.4 Decimal-Binary Equivalences

Decimal	*Binary*
1	1
3	11
7	111
15	1111
31	1 1111
63	11 1111
127	111 1111
255	1111 1111
511	1 1111 1111
1,023	11 1111 1111
2,047	111 1111 1111
4,095	1111 1111 1111
8,191	1 1111 1111 1111
16,383	11 1111 1111 1111
32,767	111 1111 1111 1111
65,535	1111 1111 1111 1111

As another example, 1111 1111 has 8 bits, so

$$\text{Decimal} = 2^8 - 1 = 256 - 1 = 255$$

BCD-8421 and BCD-2421 Code

Binary Coded Decimal (BCD) refers to representation of digits 0–9 in decimal system by 4-bit unsigned binary numbers. The usual method is to follow 8421 encoding which employs conventional route of weight placements like 8 representing the weight of the 4th place (as $2^{4-1} = 8$), 4, i.e. 2^{3-1} of the 3rd place, 2, i.e. 2^{2-1} of the 2nd place and 1, i.e. 2^{1-1} of the 1st place. The 2421 code is similar to 8421 code except for the fact that the weight assigned to 4th place is 2 and not 8. The decimal numbers 0–9 in these two codes then can be represented as shown in Table 5.5.

Table 5.5 BCD-8421 and BCD-2421 Code

Decimal	*BCD-8421*	*BCD-2421*
0	0000	0000
1	0001	0001
2	0010	0010
3	0011	0011
4	0100	0100
5	0101	1011
6	0110	1100
7	0111	1101
8	1000	1110
9	1001	1111

As an example, decimal number 29 in BCD-8421 is written as 00101001 (0010 representing 2 and 1001 representing 9) while in BCD-2421, it is written as 00101111 (0010 representing 2 and 1111 representing 9).

Example 5.4 Convert decimal 23.6 to a binary number.

Solution Split decimal 23.6 into an integer of 23 and a fraction of 0.6, and apply double dabble to each part.

$$\begin{array}{r|l} 0 & 1 \\ 1 & 0 \\ 2 & 1 \\ 5 & 1 \\ 11 & 1 \\ 2\overline{)23} & \end{array} \quad \text{Read down}$$

and

$$\begin{array}{l} 0.6 \times 2 = 1.2 = 0.2 \text{ with a carry of } 1 \\ 0.2 \times 2 = 0.4 = 0.4 \text{ with a carry of } 0 \\ 0.4 \times 2 = 0.8 = 0.8 \text{ with a carry of } 0 \\ 0.8 \times 2 = 1.6 = 0.6 \text{ with a carry of } 1 \\ 0.6 \times 2 = 0.2 = 0.2 \text{ with a carry of } 1 \end{array} \quad \text{Read down}$$

The binary number is 10111.10011. This 10-bit number is an approximation of decimal 21.6 because we terminated the conversion of the fractional part after 5 bits.

Example 5.5 A digital computer processes binary numbers that are 32 bits long. If a 32-bit number has all 1s, what is its decimal equivalent?

Solution

$$\begin{aligned} \text{Decimal} &= 2^{32} - 1 = (2^8)(2^8)(2^8)(2^8) - 1 \\ &= (256)(256)(256)(256) - 1 = 4{,}294{,}967{,}295 \end{aligned}$$

SELF-TEST

7. What is double dabble?
8. A binary number is composed of twelve 1s. What is its decimal equivalent?
9. What is the binary number for decimal 255?

5.4 OCTAL NUMBERS

The base of a number system equals the number of digits it uses. The decimal number system has a base of 10 because it uses the digits 0 to 9. The binary number system has a base of 2 because it uses only the digits 0 and 1. The *octal number system* has a base of 8. Although we can use any eight digits, it is customary to use the first eight decimal digits:

0, 1, 2, 3, 4, 5, 6, 7

(There is no 8 or 9 in the octal number code.) These digits, 0 through 7, have exactly the same physical meaning as decimal symbols; that is, 2 stands for ••, 5 symbolizes •••••, and so on.

Octal Odometer

The easiest way to learn how to count in octal numbers is to use an *octal odometer*. This hypothetical device is similar to the odometer of a car, except that each display wheel contains only eight digits, numbered 0 to 7. When a wheel turns from 7 back to 0, it sends a carry to the next-higher wheel.

Initially, an octal odometer shows

0000 (zero)

The next 7 kms produces readings of

0001 (one)
0002 (two)
0003 (three)
0004 (four)
0005 (five)
0006 (six)
0007 (seven)

At this point, the least-significant wheel has run out of digits. Therefore, the next km forces a reset and carry to obtain

0010 (eight)

The next 7 kms produces these readings: 0011, 0012, 0013, 0014, 0015, 0016, and 0017. Once again, the least-significant wheel has run out of digits. So the next km results in a reset and carry:

0020 (sixteen)

Subsequent kms produce readings of 0021, 0022, 0023, 0024, 0025, 0026, 0027, 0030, 0031, and so on.

You should have the idea by now. Each km advances the least-significant wheel by one. When this wheel runs out of octal digits, it resets and carries. And so on for the other wheels. For instance, if the odometer reading is 6377, the next octal number is 6400.

Octal-to-Decimal Conversion

How do we convert octal numbers to decimal numbers? In the octal number system each digit position corresponds to a power of 8 as follows:

$$8^3\ 8^2\ 8^1\ 8^0\ .\ 8^{-1}\ 8^{-2}\ 8^{-3}$$
$$\uparrow$$
Octal point

Therefore, to convert from octal to decimal, multiply each octal digit by its weight and add the resulting products. Note that $8^0 = 1$.

For instance, octal 23 converts to decimal like this:

$$2(8^1) + 3(8^0) = 16 + 3 = 19$$

Here is another example. Octal 257 converts to

$$2(8^1) + 5(8^1) + 7(8^0) = 128 + 40 + 7 = 175$$

Decimal-to-Octal Conversion

How do you convert in the opposite direction, that is, from decimal to octal? *Octal dabble*, a method similar to double dabble, is used with octal numbers. Instead of dividing by 2 (the base of binary numbers), you divide by 8 (the base of octal numbers) writing down the remainders after each division. The remainders in reverse order form the octal number. As an example, convert decimal 175 as follows:

$$\begin{array}{rl} 8\overline{)2} & 2 \rightarrow \text{(third remainder)} \\ 8\overline{)21} & 5 \rightarrow \text{(second remainder)} \\ 8\overline{)175} & 7 \rightarrow \text{(first remainder)} \end{array}$$

You can condense these steps by writing

$$\begin{array}{rl} 0 & 2 \\ 2 & 5 \\ 21 & 7 \\ 8\overline{)175} & \end{array} \quad \text{Read down}$$

Thus decimal 175 is equal to octal 257.

Fractions

With decimal fractions, multiply instead of divide, writing the carry into the integer position. An example of this is to convert decimal 0.23 into an octal fraction.

$$\begin{array}{ll} 0.23 \times 8 = 1.84 = 0.84 & \text{with a carry of 1} \\ 0.84 \times 8 = 6.72 = 0.72 & \text{with a carry of 6} \\ 0.72 \times 8 = 5.76 = 0.76 & \text{with a carry of 5} \\ \text{etc.} & \end{array} \quad \text{Read down}$$

The carries read downward give the octal fraction 0.165. We terminated after three places; for more accuracy, we would continue multiplying to obtain more octal digits.

Octal-to-Binary Conversion

Because 8 (the base of octal numbers) is the third power of 2 (the base of binary numbers), you can convert from octal to binary as follows: change each octal digit to its binary equivalent. For instance, change octal 23 to its binary equivalent as follows:

$$\begin{array}{cc} 2 & 3 \\ \downarrow & \downarrow \\ 010 & 011 \end{array}$$

Here, each octal digit converts to its binary equivalent (2 becomes 010, and 3 becomes 011). The binary equivalent of octal 23 is 010 011, or 010011. Often, a space is left between groups of 3 bits; this makes it easier to read the binary number.

As another example, octal 3574 converts to binary as follows:

3	5	7	4
↓	↓	↓	↓
011	101	111	100

Hence binary 011101111100 is equivalent to octal 3574. Notice how much easier the binary number is to read if we leave a space between groups of 3 bits: 011 101 111 100.

Mixed octal numbers are no problem. Convert each octal digit to its equivalent binary value. Octal 34.562 becomes

3	4	.	5	6	2
↓	↓		↓	↓	↓
011	100	.	101	110	010

Binary-to-Octal Conversion

Conversion from binary to octal is a reversal of the foregoing procedures. Simply remember to group the bits in threes, starting at the binary point; then convert each group of three to its octal equivalent (0s are added at each end, if necessary). For instance, binary number 1011.01101 converts as follows:

1011.01101 → 001		011.	011		010
↓		↓	↓		↓
1		3	3		2

Start at the binary point and, working both ways, separate the bits into groups of three. When necessary, as in this case, add 0s to complete the outside groups. Then convert each group of three into its binary equivalent. Therefore:

$$1011.01101 = 13.32$$

The simplicity of converting octal to binary and vice versa has many advantages in digital work. For one thing, getting information into and out of a digital system requires less circuitry because it is easier to read and print out octal numbers than binary numbers. Another advantage is that large decimal numbers are more easily converted to binary if first converted to octal and then to binary, as shown in Example 5.6.

Example 5.6 What is the binary equivalent of decimal 363?

Solution One approach is double dabble. Another approach is octal dabble, followed by octal-to-binary conversion. Here is how the second method works:

0	5	
5	5	Read down
45	3	
$8\overline{)363}$	↓	

Next, convert octal 553 to its binary equivalent:

5	5	3
↓	↓	↓
101	101	011

The double-dabble approach would produce the same answer, but it is tedious because you have to divide by 2 nine times before the conversion terminates.

SELF-TEST

10. What are the digits used in the octal number system?
11. What is the octal number for binary 111? What is the decimal number for binary 111?

5.5 HEXADECIMAL NUMBERS

Hexadecimal numbers are used extensively in microprocessor work. To begin with, they are much shorter than binary numbers. This makes them easy to write and remember. Furthermore, you can mentally convert them to binary whenever necessary.

The hexadecimal number system has a base of 16. Although any 16 digits may be used, everyone uses 0 to 9 and A to F as shown in Table 5.6. In other words, after reaching 9 in the hexadecimal system, you continue counting as follows:

A, B, C, D, E, F

Table 5.6 Hexadecimal Digits

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

Hexadecimal Odometer

The easiest way to learn how to count in hexadecimal numbers is to use a *hexadecimal odometer*. This hypothetical device is similar to the odometer of a car, except that each display wheel has 16 digits, numbered 0 to F. When a wheel turns from F back to 0, it sends a carry to the next higher wheel.

Initially, a hexadecimal odometer shows

0000 (zero)

The next 9 kms produces readings of

0001 (one)
0002 (two)
0003 (three)
0004 (four)
0005 (five)
0006 (six)
0007 (seven)
0008 (eight)
0009 (nine)

The next 6 kms gives

000A (ten)
000B (eleven)
000C (twelve)
000D (thirteen)
000E (fourteen)
000F (fifteen)

At this point, the least-significant wheel has run out of digits. Therefore, the next km forces a reset and carry to obtain

0010 (sixteen)

The next 15 kms produces these readings: 0011, 0012, 0013, 0014, 0015, 0016, 0017, 0018, 0019, 001A, 001B, 001C, 001D, 001E, and 001F. Once again, the least significant wheel has run out of digits. So, the next km results in a reset and carry:

0020 (thirty-two)

Subsequent kms produce readings of 0021, 0022, 0023, 0024, 0025, 0026, 0027, 0028, 0029, 002A, 002B, 002C, 002D, 002E, and 002F.

You should have the idea by now. Each km advances the least-significant wheel by one. When this wheel runs out of hexadecimal digits, it resets and carries, and so on for the other wheels. For instance, here are three more examples:

Number	Next number
835C	835D
A47F	A480
BFFF	C000

Hexadecimal-to-Binary Conversion

To convert a hexadecimal number to a binary number, convert each hexadecimal digit to its 4-bit equivalent using the code given in Table 5.5. For instance, here's how 9AF converts to binary:

9	A	F
↓	↓	↓
1001	1010	1111

As another example, C5E2 converts like this:

C	5	E	2
↓	↓	↓	↓
1100	0101	1110	0010

Binary-to-Hexadecimal Conversion

To convert in the opposite direction, from binary to hexadecimal, again use the code from Table 5.5. Here are two examples. Binary 1000 1100 converts as follows:

1000	1100
↓	↓
8	C

Binary 1110 1000 1101 0110 converts like this:

1110	1000	1101	0110
↓	↓	↓	↓
E	8	D	6

In both these conversions, we start with a binary number and wind up with the equivalent hexadecimal number.

Hexadecimal-to-Decimal Conversion

How do we convert hexadecimal numbers to decimal numbers? In the hexadecimal number system each digit position corresponds to a power of 16. The weights of the digit positions in a hexadecimal number are as follows

$$16^3 \quad 16^2 \quad 16^1 \quad 16^0 \quad . \quad 16^{-1} \quad 16^{-2} \quad 16^{-3}$$

↑

Hexadecimal point

Therefore, to convert from hexadecimal to decimal, multiply each hexadecimal digit by its weight and add the resulting products. Note that $16^0 = 1$.

Here's an example. Hexadecimal F8E6.39 converts to decimal as follows:

$$\begin{aligned} \text{F8E6} &= \text{F}(16^3) + 8(16^2) + \text{E}(16^1) + 6(16^0) + 3(16^{-1}) + 9(16^{-2}) \\ &= 15(16^3) + 8(16^2) + 14(16^1) + 6(16^0) + 3(16^{-1}) + 9(16^{-2}) \\ &= 61{,}440 + 2048 + 224 + 6 + 0.1875 + 0.0352 \\ &= 63{,}718.2227 \end{aligned}$$

Decimal-to-Hexadecimal Conversion

One way to convert from decimal to hexadecimal is the *hex dabble*. The idea is to divide successively by 16, writing down the remainders. Here's a sample of how it's done. To convert decimal 2479 to hexadecimal, the first division is

$$\begin{array}{rl} 154 & 15 \rightarrow \text{F} \\ 16\overline{)2479} & \end{array}$$

In this first division, we get a quotient of 154 with a remainder of 15 (equivalent to F). The next step is

$$\begin{array}{rl} 9 & 10 \rightarrow \text{A} \\ 154 & 15 \rightarrow \text{F} \\ 16\overline{)2479} & \end{array}$$

Here we obtain a quotient of 9 with a remainder of 10 (same as A). The final step is

$$\begin{array}{rll} 0 & 9 \rightarrow 9 & \Big| \\ 9 & 10 \rightarrow \text{A} & \Big| \quad \text{Read down} \\ 154 & 15 \rightarrow \text{F} & \downarrow \\ 16\overline{)2479} & & \end{array}$$

Therefore, hexadecimal 9AF is equivalent to decimal 2479.

Notice how similar hex dabble is to double dabble. Notice also that remainders greater than 9 have to be changed to hexadecimal digits (10 becomes A, 15 becomes F, etc.).

Using Appendix 1*

A typical microcomputer can store up to 65,535 bytes. The decimal addresses of these bytes are from 0 to 65,535. The equivalent binary addresses are from

0000	0000	0000	0000
1111	1111	1111	1111

The first 8 bits are called the *upper byte*, and the second 8 bits are the *lower byte*.

If you have to do many conversions between binary, hexadecimal, and decimal, learn to use Appendix 1. It has four headings: *binary, hexadecimal, upper byte,* and *lower byte.* For any decimal number between 0 and 255, you would use the binary, hexadecimal, and lower byte columns. Here is the recommended way to use Appendix 1. Suppose you want to convert binary 0001 1000 to its decimal equivalent. First, mentally convert to hexadecimal:

$$0001\ 1000 \rightarrow 18 \quad \text{(mental conversion)}$$

Next, look up hexadecimal 18 in Appendix 1 and read the corresponding decimal value from the lower-byte column:

$$18 \rightarrow 24 \quad \text{(look up in Appendix 1)}$$

For another example, binary 1111 0000 converts like this:

$$1111\ 0000 \rightarrow F0 \rightarrow 240$$

The reason for mentally converting from binary to hexadecimal is that you can more easily locate a hexadecimal number in Appendix 1 than a binary number. Once you have the hexadecimal equivalent, you can read the lower-byte column to find the decimal equivalent.

When the decimal number is greater than 255, you have to use both the upper byte and the lower byte in Appendix 1. For instance, suppose you want to convert this binary number to its decimal equivalent:

$$1110\ 1001\ 0111\ 0100$$

First, convert the upper byte to its decimal equivalent as follows:

$$1110\ 1001 \rightarrow E9 \rightarrow 59{,}648 \quad \text{(upper byte)}$$

Second, convert the lower byte to its decimal equivalent:

$$0111\ 0100 \rightarrow 74 \rightarrow 116 \quad \text{(lower byte)}$$

Finally, add the upper and lower bytes to obtain the total decimal value:

$$59{,}648 + 116 = 59{,}764$$

Therefore, binary 1110 1001 0111 0100 is equivalent to decimal 59,764.

Once you get used to working with Appendix 1, you will find it to be a quick and easy way to convert between the number systems. Because it covers the decimal numbers from 0 to 65,535, Appendix 1 is extremely useful for microprocessors where the typical memory addresses are over the same decimal range.

* A number of hand calculators will convert binary, octal, decimal and hexadecimal numbers.

Example 5.7 A computer memory can store thousands of binary instructions and data. A typical microprocessor has 65,536 addresses, each storing a byte. Suppose that the first 16 addresses contain these bytes:

0011 1100
1100 1101
0101 0111
0010 1000
1111 0001
0010 1010
1101 0100
0100 0000
0111 0111
1100 0011
1000 0100
0010 1000
0010 0001
0011 1010
0011 1110
0001 1111

Convert these bytes to their hexadecimal equivalents.

Solution Here are the stored bytes and their hexadecimal equivalents:

Memory contents	Hexadecimal equivalents
0011 1100	3C
1100 1101	CD
0101 0111	57
0010 1000	28
1111 0001	F1
0010 1010	2A
1101 0100	D4
0100 0000	40
0111 0111	77
1100 0011	C3
1000 0100	84
0010 1000	28
0010 0001	21
0011 1010	3A
0011 1110	3E
0001 1111	1F

What is the point of this example? When discussing the contents of a computer memory, we can use either binary numbers or hexadecimal numbers. For instance, we can say that the first address contains 0011 1100, or we can say

that it contains 3C. Either way, we obtain the same information. But notice how much easier it is to say, write, and think 3C than it is to say, write, and think 0011 1100. In other words, hexadecimal numbers are much easier for people to work with.

Example 5.8 Convert the hexadecimal numbers of the preceding example to their decimal equivalents.

Solution The first address contains 3C, which converts like this:

$$3(16^1) + C(16^0) = 48 + 12 = 60$$

Even easier, look up the decimal equivalent of 3C in Appendix 1, and you get 60. Either by powers of 16 or with reference to Appendix 1, we can convert the other memory contents to get the following:

Memory contents	Hexadecimal equivalents	Decimal equivalents
0011 1100	3C	60
1100 1101	CD	205
0101 0111	57	87
0010 1000	28	40
1111 0001	F1	241
0010 1010	2A	42
1101 0100	D4	212
0100 0000	40	64
0111 0111	77	119
1100 0011	C3	195
1000 0100	84	132
0010 1000	28	40
0010 0001	21	33
0011 1010	3A	58
0011 1110	3E	62
0001 1111	1F	31

Example 5.9 Convert decimal 65,535 to its hexadecimal and binary equivalents.

Solution Use hex dabble as follows:

$$\begin{array}{r|l} 0 & 15 \rightarrow \text{F} \\ 15 & 15 \rightarrow \text{F} \\ 255 & 15 \rightarrow \text{F} \\ 4095 & 15 \rightarrow \text{F} \\ 16\overline{)65{,}535} & \end{array} \quad \text{Read down}$$

Therefore, decimal 65,535 is equivalent to hexadecimal FFFF.

Next, convert from hexadecimal to binary as follows:

F	F	F	F
↓	↓	↓	↓
1111	1111	1111	1111

This means that hexadecimal FFFF is equivalent to binary 1111 1111 1111 1111.

Example 5.10 Show how to use Appendix 1 to convert decimal 56,000 to its hexadecimal and binary equivalents.

Solution The first thing to do is to locate the largest decimal number equal to 56.000 or less in Appendix 1. The number is 55,808, which converts like this:

$$55{,}808 \rightarrow \text{DA} \quad \text{(upper byte)}$$

Next, you need to subtract this upper byte from the original number:

$$56{,}000 - 55{,}808 = 192 \quad \text{(difference)}$$

This difference is always less than 256 and represents the lower byte, which Appendix 1 converts as follows:

$$192 \rightarrow \text{C0}$$

Now, combine the upper and lower byte to obtain

$$\text{DAC0}$$

which you can mentally convert to binary:

$$\text{DAC0} \rightarrow 1101\ 1010\ 1100\ 0000$$

Example 5.11 Convert Table 5.4 into a new table with three column headings: "Decimal," "Binary," and "Hexadecimal."

Solution This is easy. Convert each group of bits to its hexadecimal equivalent as shown in Table 5.7.

Table 5.7 Decimal-Binary-Hexadecimal Equivalences

Decimal	*Binary*	*hexadecimal*
1	1	1
3	11	3
7	111	7
15	1111	F
31	11111	1F
63	111111	3F
127	1111111	7F
255	11111111	FF
511	111111111	1FF
1,023	1111111111	3FF
2,047	11111111111	7FF
4,095	111111111111	FFF
8,191	1111111111111	1FFF
16,383	11111111111111	3FFF
32,767	111111111111111	7FFF
65,535	1111111111111111	FFFF

SELF-TEST

12. What are the symbols used in hexadecimal numbers?
13. What is the binary equivalent of hexadecimal 3C?
14. What is the decimal equivalent of hexadecimal 3C?

5.6 THE ASCII CODE

To get information into and out of a computer, we need to use some kind of *alphanumeric* code (one for letters, numbers, and other symbols). At one time, manufacturers used their own alphanumeric codes, which led to all kinds of confusion. Eventually, industry settled on an input-output code known as the *American Standard Code for Information Interchange* (ASCII, pronounced ask'-ee). This code allows manufacturers to standardize computer hardware such as keyboards, printers, and video displays.

Using the Code

The ASCII code is a 7-bit code whose format is

$$X_6X_5X_4X_3X_2X_1X_0$$

where each X is a 0 or a 1. Use Table 5.8 to find the ASCII code for the uppercase and lowercase letters of the alphabet and some of the most commonly used symbols. For example, the table shows that the capital letter A has an $X_6X_5X_4$ of 100 and an $X_3X_2X_1X_0$ of 0001. The ASCII code for A is, therefore,

1000001

For easier reading, we can leave a space as follows:

100 0001 (A)

The letter a is coded as

110 0001 (a)

More examples are

110 0010 (b)

Table 5.8 ASCII Code

	$X_6X_5X_4$					
$X_3X_2X_1X_0$	*010*	*011*	*100*	*101*	*110*	*111*
0000	SP	0	@	P		p
0001	!	1	A	Q	a	q
0010	"	2	B	R	b	r
0011	#	3	C	S	c	s
0100	$	4	D	T	d	t
0101	%	5	E	U	e	u
0110	&	6	F	V	f	v
0111	'	7	G	W	g	w
1000	(	8	H	X	h	x
1001	)	9	I	Y	i	y
1010	*	:	J	Z	j	z
1011	+	;	K		k	
1100	,	<	L		l	
1101	–	=	M		m	
1110	.	>	N		n	
1111	/	?	O		o	

110 0011 (c)
110 0100 (d)

and so on.

Also, study the punctuation and mathematical symbols. Some examples are

010 0100 ($)
010 1011 (+)
011 1101 (=)

In Table 5.7, SP stands for space (blank). Hitting the space bar of an ASCII keyboard sends this into a microcomputer:

010 0000 (space)

Parity Bit

The ASCII code is used for sending digital data over telephone lines. As mentioned in the preceding chapter, 1-bit errors may occur in transmitted data. To catch these errors, a parity bit is usually transmitted along with the original bits. Then a parity checker at the receiving end can test for even or odd parity, whichever parity has been prearranged between the sender and the receiver. Since ASCII code uses 7 bits, the addition of a parity bit to the transmitted data produces an 8-bit number in this format:

$$X_7X_6X_5X_4 \quad X_3X_2X_1X_0$$
$$\uparrow$$
Parity bit

This is an ideal length because most digital equipment is set up to handle bytes of data.

EBCDIC as Alphanumeric Code

There exists few others but relatively less used alphanumeric codes. The EBCDIC is an abbreviation of Extended Binary Coded Decimal Interchange Code. It is an eight-bit code and primarily used in IBM make devices. Here, the binary codes of letters and numerals come as an extension of BCD code. The bit assignments of EBCDIC are different from the ASCII but the character symbols are the same.

Example 5.12 With an ASCII keyboard, each keystroke produces the ASCII equivalent of the designated character. Suppose that you type PRINT X. What is the output of an ASCII keyboard?

Solution The sequence is as follows: P (101 0000), R (101 0010), I (100 1001), N (100 1110), T (101 0100), space (010 000), X (101 1000).

Example 5.13 A computer sends a message to another computer using an odd-parity bit. Here is the message in ASCII code, plus the parity bit:

1100 1000
0100 0101
0100 1100
0100 1100
0100 1111

What do these numbers mean?

Solution First, notice that each 8-bit number has odd parity, an indication that no 1-bit errors occurred during transmission. Next, use Table 5.7 to translate the ASCII characters. If you do this correctly, you get a message of HELLO.

SELF-TEST

15. What is the ASCII code?
16. What symbol is represented by the ASCII code 100 0000?
17. What ASCII code is used for the percent sign, %?

5.7 THE EXCESS-3 CODE

The *excess-3 code* is an important 4-bit code sometimes used with binary-coded decimal (BCD) numbers. To convert any decimal number into its excess-3 form, add 3 to each decimal digit, and then convert the sum to a BCD number.

For example, here is how to convert 12 to an excess-3 number. First, add 3 to each decimal digit:

1	1
+3	+3
4	5

Second, convert the sum to BCD form:

4	5
↓	↓
0100	0101

So, 0100 0101 in the excess-3 code stands for decimal 12.

Take another example; convert 29 to an excess-3 number:

2	9
+3	+3
5	12
↓	↓
0101	1100

After adding 9 and 3, do *not* carry the 1 into the next column; instead, leave the result intact as 12, and then convert as shown. Therefore, 0101 1100 in the excess-3 code stands for decimal 29.

Table 5.9 shows the excess-3 code. In each case, the excess-3 code number is 3 greater than the BCD equivalent. Such coding helps in BCD arithmetic as 9's complement of any excess-3 coded number can be obtained simply by complementing each bit. Take for example decimal number 2. Its 9's complement is $9 - 2 = 7$. Excess-3 code of 2 is 0101. Complementing each bit we get 1010 and its decimal equivalent is 7. To convert BCD to excess-3 we need an adder and for the reverse we need a subtractor. These circuits are discussed in the next chapter. Incidentally, if you need an integrated circuit (IC) that converts from excess 3 to decimal, look at the data sheet of a 7443. This transistor-transistor logic (TTL) chip has four input lines for the excess-3 input and 10 output lines for the decoded decimal output.

Table 5.9 Excess-3 Code

Decimal	*BCD*	*Excess-3*
0	0000	0011
1	0001	0100
2	0010	0101
3	0011	0110
4	0100	0111
5	0101	1000
6	0110	1001
7	0111	1010
8	1000	1011
9	1001	1100

5.8 THE GRAY CODE

The advantage of such coding will be understood from this example. Let an object move along a track and move from one zone to another. Let the presence of the object in one zone is sensed by sensors *ABC*. If consecutive zones are binary coded then zone-0 is represented by *ABC* = 000, zone-1 by *ABC* = 001, zone-2 by *ABC* = 010 and so on, as shown in Fig. 5.1a. Now consider, the object moves from zone-1 to zone-2. Both *BC* has to change to sense that movement. Suppose, sensor *B* (may be an electro-mechanical switch) reacts slightly late than sensor *C*. Then, initially *ABC* = 000 is sensed as if the object has moved in the other direction from zone-1 to zone-0. This problem can be more prominent if the object moves from zone-3 (*ABC* = 011) to zone-4 (*ABC* = 100) when all three sensors has to change its value. Note that, if zones are gray coded (Fig. 5.1b) such problem does not appear as between two consecutive zones only one sensor changes its value.

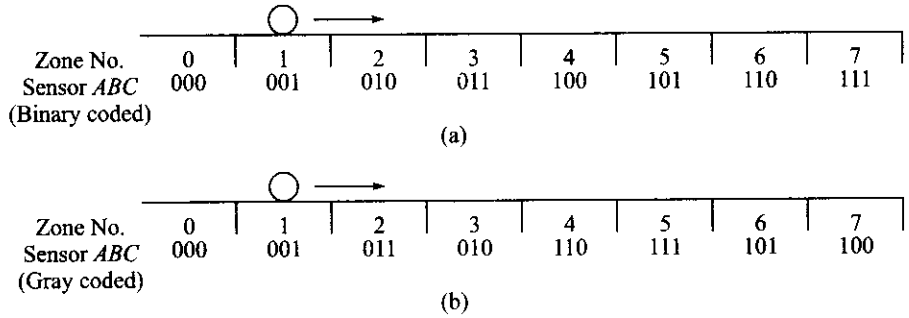


Fig. 5.1 Object moving along a track with sensors: (a) Binary coded, (b) Gray coded

The disadvantage with gray code is that it is not good for arithmetic operation. However, comparing truth tables of binary coded numbers and gray coded numbers (Table 5.18) we can design binary to gray converter as shown in Fig. 5.2a and gray to binary converter as shown in Fig. 5.2b. Let's see how these circuits work by taking one example each.

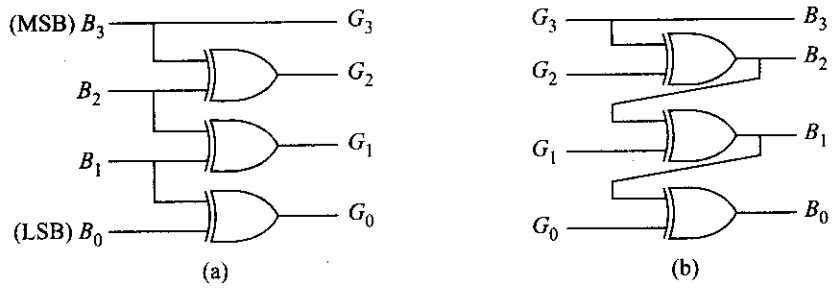


Fig. 5.2 (a) Binary to Gray converter, (b) Gray to Binary converter

Consider, a binary number $B_3B_2B_1B_0 = 1011$. Following the relation shown in Fig. 5.2a we get, $G_3 = B_3 = 1$, $G_2 = B_3 \oplus B_2 = 1 \oplus 0 = 1$, $G_1 = B_2 \oplus B_1 = 0 \oplus 1 = 1$ and $G_0 = B_1 \oplus B_0 = 1 \oplus 1 = 0$, i.e. $G_3G_2G_1G_0 = 1110$ and we can verify the same from truth table.

Similarly, for a gray coded number say, $G_3G_2G_1G_0$ = 0111 from Fig. 5.2b we get, $B_3 = G_3 = 0$, $B_2 = G_3 \oplus G_2 = 0 \oplus 1 = 1$, $B_1 = B_2 \oplus G_1 = 1 \oplus 1 = 0$ and $B_0 = B_1 \oplus G_0 = 0 \oplus 1 = 1$, i.e. $B_3B_2B_1B_0 = 0101$. Again this conversion can be verified from Table 5.10 that shows the *Gray code*, along with the corresponding binary numbers. Each Gray-code number differs from any adjacent number by a single bit. For instance, in going from decimal 7 to 8, the Gray-code numbers change from 0100 to 1100; these numbers differ only in the most significant bit. As another example, decimal numbers 13 and 14 are represented by Gray-code numbers 1011 and 1001; these numbers differ in only one digit position (the second position from the right). So, it is with the entire Gray code; every number differs by only 1 bit from the preceding number.

Table 5.10 Gray Code

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

Besides the excess-3 and Gray codes, there are other binary-type codes. Appendix 5 lists some of these codes for future reference. Incidentally, the BCD code is sometimes referred to as the *8421 code* because the weights of the digit positions from left to right are 8, 4, 2, and 1. As shown in Appendix 5, there are many other weighted codes such as the 7421, 6311, 5421, and so on.

5.9 TROUBLESHOOTING WITH A LOGIC PULSER

Figure 5.3 shows a typical *logic pulser*, a troubleshooting tool that generates a brief voltage pulse when its push-button switch is pressed. Because of its design, the logic pulser (on the left) senses the original state of the node and produces a voltage pulse of the opposite polarity. When this happens, the logic probe (on the right) blinks, indicating a temporary change of output state.

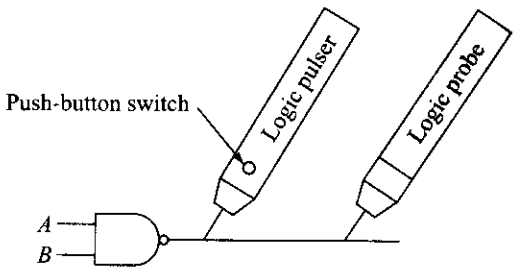


Fig. 5.3 Using a logic pulser and a logic probe

Thévenin Circuit

Figure 5.4a shows the Thévenin equivalent circuit for a typical logic pulser. The Thévenin voltage is a pulse with an amplitude of 5 V; the polarity automatically adjusts to the original state of the test node. As shown, the Thévenin resistance or output impedance is only 2Ω. This Thévenin resistance is representative; the exact value depends on the particular logic pulser being used. Typically, a TTL gate has an output resistance between 12Ω (low state) and 70Ω (high state). When a logic pulser drives the output of a NAND gate, the equivalent circuit appears as shown in Fig. 5.4b. Because of the low output impedance (2Ω) of the logic pulser, most of the voltage pulse appears across the load (12 to 70Ω). Therefore, the output is briefly driven into the opposite voltage state.

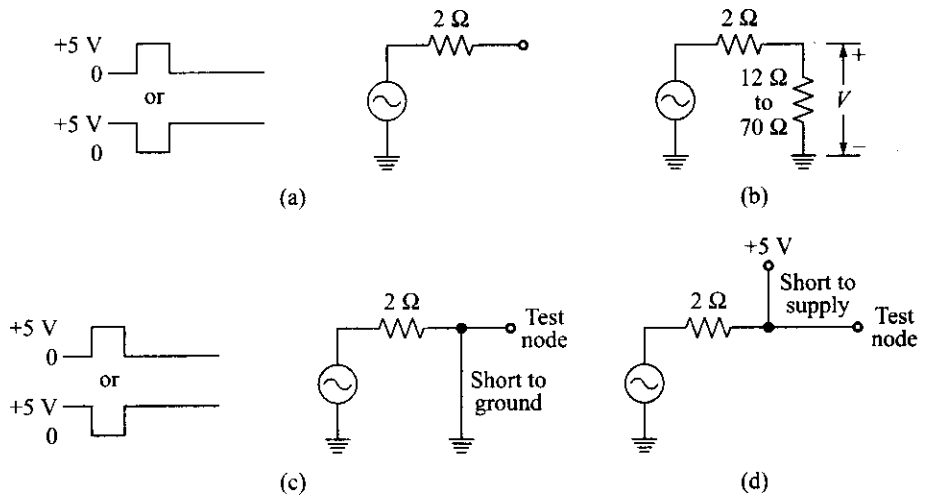


Fig. 5.4 **(a) Thévenin equivalent of logic pulser, (b) Logic pulser driving NAND-gate output, (c) Node stuck in high state**

Testing Any Node

You can use a logic pulser to drive any node in a circuit, whether input or output. Almost always, the load impedance of the node being driven is larger than the output impedance of the logic pulser. For this reason, the logic pulser can usually change the state of any node in a logic circuit. Also, the pulse width is kept very short (fractions of a microsecond) to avoid damaging the circuit being tested. (*Note: Power dissipation* is what damages ICs. A brief voltage pulse produces only a small power dissipation.)

Stuck Nodes

When is a logic pulser unable to change the state of a node? When the test node is shorted to ground or to the supply voltage. For instance, Fig. 5.4c shows the test node shorted to ground. In this case, all the voltage pulse is dropped across the internal impedance of the logic pulser; therefore the test node is stuck at 0 V, the low state.

On the other hand, the test node may be shorted to the supply voltage as shown in Fig. 5.4d. Most power supplies are regulated and have impedances in fractions of 1Ω. For this reason, most of the voltage pulse is again dropped across the output impedance of the logic pulser, which means that the test node is stuck at +5 V.

Finding Stuck Nodes

If a circuit is faulty, you can use a logic pulser and logic probe to locate stuck nodes. Here's how. Touch both the logic pulser and the logic probe to a node as shown in Fig. 5.3. If the node is stuck in either state, the logic pulser will be unable to change the state. So, if the logic probe does not blink, you have a stuck node. Then, you can look for solder bridges on any trace connected to the stuck node, or possibly replace the IC having the stuck node.

5.10 ERROR DETECTION AND CORRECTION

Error Detection and Correction (EDAC) techniques are used to ensure that data is correct and has not been corrupted, either by hardware failures or by noise occurring during transmission or a data read operation from memory. There are many different error correction codes in existence. The reason for different codes being used in different applications has to do with the historical development of data storage, the types of data errors occurring, and the overhead associated with each of the error-detection techniques. We discuss some of the popular techniques here with details of Hamming code.

Parity Code

We have discussed parity generation and checking in detail in Section 4.8. When a word is written into memory, each parity bit is generated from the data bits of the byte it is associated with. This is done by a tree of exclusive-OR gates. When the word is read back from the memory, the same parity computation is done on the data bits read from the memory, and the result is compared to the parity bits that were read. Any computed parity bit that does not match the stored parity bit indicates that there was at least one error in that byte (or in the parity bit itself). However, parity can only detect an odd number of errors. If even number of errors occur, the computed parity will match the read parity, so the error will go undetected. Since memory errors are rare if the system is operating correctly, the vast majority of errors will be single-bit errors, and will be detected.

Unfortunately, while parity allows for the detection of single-bit errors, it does not provide a means of determining which bit is in error, which would be necessary to correct the error. Thus the data needs to be read again if an error is detected. Error Correction Code (ECC) is an extension of the parity concept.

Checksum Code

This is a kind of error detection code used for checking a large block of data. The checksum bits are generated by summing all the codes of a message and are stored with data. Usually the block of data summed is of length 512 or 1024 and the checksum results are stored in 32 bits that allow overflow. When data is read, the summing operation is again done and checksum bits generated are matched with the stored one. If they are unequal, then an error is detected. Obviously, it can fool the detection system if error occurring at one place is compensated by the other.

Cycle Redundancy Code (CRC)

CRC code is a more robust error checking algorithm than the previous two. The code is generated in the following manner. Take a binary message and convert it to a polynomial, then divide it by another predefined polynomial called the *key*. The remainder from this division is the CRC. This is stored with the message. Upon reading the data, memory controller does the same operation, i.e. divides the message by the same key and compares with CRC stored. If they differ, then the data has been wrongly read or stored. Not all keys are equally good. The longer the key, the better is the error checking. On the other hand, the calculations with long keys can get quite complex. Two of the polynomials commonly used are:

$$\text{CRC–16} = x_{16} + x_{15} + x_2 + 1$$

$$\text{CRC–32} = x_{32} + x_{26} + x_{23} + x_{22} + x_{16} + x_{12} + x_{11} + x_{10} + x_8 + x_7 + x_5 + x_4 + x_2 + x + 1$$

Usually, series of exclusive-OR gates are used to generate CRC code. We shall see in the next chapter that the sum term arising out of addition is essentially an exclusive-OR operation.

Hamming Code

Introduced in 1950 by R W Hamming, this scheme allows one bit in the word to be corrected, but is unable to correct events where more than one bit in the word is in error. These multi-bit errors can only be detected, not corrected, and therefore will cause a system to malfunction. Hamming code uses parity bits discussed before but in a different way. For n number of data bits, if number of parity bits required here is m, then

$$2^m \geq m + n + 1$$

In the memory word, (i) all bit positions that are of the form 2^i are used as parity bits (like 1, 2, 4, 8, 16, 32...) and (ii) the remaining positions are used as data bits (like 3, 5, 6, 7, 9, 10, 11, 12, 13, 14, 17, 18...) Thus code will be in the form of

P1 P2 D3 P4 D5 D6 D7 P8 D9 D10 D11 ...

where P1, P2, P4, P8... are parity bits and D3, D5, D6, D7... are data bits.

We discuss Hamming code generation with an example. Consider the 7-bit data to be coded is 0110101. This requires 4 parity bits in position 1, 2, 4 and 8 so that Hamming coded data becomes 11-bit long. To calculate the value of P1, we check parity of zeroth binary locations of data bits. This is shown in 3rd row of Fig. 5.5 for this example. Zeroth locations are the places where address ends with a 1. These are D3, D5, D9 and D11 for 7-bit data. Since we have total odd number of 1s in these 4 positions P1 = 1. This is calculated as done in case of parity generation (refer to Section 4.8) by series of exclusive-OR gates through the equation

$$P1 = D3 \oplus D5 \oplus D9 \oplus D11$$

Similarly for P2, we check locations where we have 1 in address of the 1st bit, i.e. D3, D6, D7, D10 and D11. Since there are even number of 1s, P2 = 1. Proceeding in similar manner and examining parity of 2nd and 3rd position, we get P4 = 0 and P8 = 0.

	0001 P1	0010 P2	0011 D3	0100 P4	0101 D5	0110 D6	0111 D7	1000 P8	1001 D9	1010 D10	1011 D11
Data word (without parity)			0		1	1	0		1	0	1
P1	1		0		1		0		1		1
P2		0	0			1	0			0	1
P4				0	1	1	0				
P8								0	1	0	1
Data with parity	1	0	0	0	1	1	0	0	1	0	1

Fig. 5.5 Calculation of Parity Bits

Next we discuss how error in a Hamming coded data is detected and if it is in single bit, how it is corrected. We continue with the previous example and consider that the data is incorrectly read in position D11 so that 11-bit coded data is 10001100100. Figure 5.6 describes the detection mechanism. First of all, we check the parity of zeroth position and find it to be even. Since P1 = 1, the parity check fails and this is equivalent to generating a parity bit at the output (last column) following the equation

$$\text{Parity P1 check bit} = D3 \oplus D5 \oplus D9 \oplus D11 \oplus P1$$

This is similar to parity checker in Section 4.8. Note that, in addition to data bits, we have also included the corresponding parity bit to the input of exclusive-OR gate tree. Proceeding similarly for other positions, we

find that except for P4 all other parity checks fail. Note that, even a single failure detects an error. However, to correct the error, we use the output of last column 1011 (in the order P8 P4 P2 P1) and find its decimal equivalent which is 11. So the data of location 11, which is D11 needs to be corrected.

	P1	P2	D3	P4	D5	D6	D7	P8	D9	D10	D11	Parity check	Parity bit
Received data word	1	0	0	0	1	1	0	0	1	0	0		
P1	1		0		1		0		1		0	Fail	1
P2		0	0			1	0			0	0	Fail	1
P4				0	1	1	0					Pass	0
P8								0	1	0	0	Fail	1

Fig. 5.6 Error detection and correction

Note that, this method detects error in more than one position unlike the first method but overhead is more. In simple parity method, we add 1 additional bit for 7-bit data whreas it is 4 in this method. Also note, by further increasing this overhead, error in more than one position can also be corrected. However, more than one-bit error is unlikely for memory read. With overhead for one-bit correction, if there occurs error in more than one-bit positions, then the data needs to be read once again from the memory.

SELF-TEST

18. Can parity code detect even number of errors?
19. What is the full form of CRC?
20. What is the advantage of Hamming code?
21. What is error detection-correction overhead?

PROBLEM SOLVING WITH MULTIPLE METHODS

Problem Add two gray coded numbers 0100 and 0111 and express the result in gray code.

Solution Since gray coded numbers are not suitable for arithmetic operations, we have to convert the numbers to some other form, perform the addition and then convert the result to gray code. We first show how it can be done through lookup tables. It would require storage of large lookup tables, if the numbers are large in value. Next, we show the converter-based approach which only needs the implementation of conversion equations.

In Method-1, we take help of first two columns of Table 5.9 and convert these two numbers to decimal, add the decimal numbers and then again use the table to get corresponding gray coded number. This is shown in Fig. 5.7a.

In Method-2, we take help of last two columns of Table 5.9 and convert these two numbers to binary, perform binary addition and then again use the table to get corresponding gray coded number. This is shown in Fig. 5.7b.

In Method-3, we take help of gray to binary conversion relation shown in Fig. 5.2b and convert these two numbers to binary, perform binary addition and then use binary to gray conversion relation shown in Fig. 5.2a to get corresponding gray coded number. This is shown in Fig. 5.7c.

Using Table 5.9

Gray	Decimal
0100	7
0111	+5
	12

Decimal	Gray
12	1010

(a) Addition using Method-1

Using Table 5.9

Gray	Binary
0100	0111
0111	+0101
	1100

Binary	Gray
1100	1010

(b) Addition using Method-2

From Fig. 5.2b

Gray to Binary Conversion:	$B_3 = G_3$	$B_2 = B_3 \oplus G_2$	$B_1 = B_2 \oplus G_1$	$B_0 = B_1 \oplus G_0$	Binary
For $G_3G_2G_1G_0 = 0100$:	$B_3 = 0$	$B_2 = 0 \oplus 1 = 1$	$B_1 = 1 \oplus 0 = 1$	$B_0 = 1 \oplus 0 = 1$	0111
For $G_3G_2G_1G_0 = 0110$:	$B_3 = 0$	$B_2 = 0 \oplus 1 = 1$	$B_1 = 1 \oplus 1 = 0$	$B_0 = 0 \oplus 1 = 1$	+0101
					1100

From Fig. 5.2a

Binary to Gray Conversion:	$G_3 = B_3$	$G_2 = B_3 \oplus B_2$	$G_1 = B_2 \oplus B_1$	$G_0 = B_1 \oplus B_0$	Gray
For $B_3B_2B_1B_0 = 1100$:	$B_3 = 1$	$B_2 = 1 \oplus 0 = 0$	$B_1 = 1 \oplus 0 = 1$	$B_0 = 0 \oplus 0 = 0$	1010

(c) Addition using Method-3

Fig. 5.7

SUMMARY

To convert from binary to decimal numbers, add the weight of each bit position (1, 2, 4, 8, ...) when there is a 1 in that position. With fractions, the binary weights are $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}$, ..., and so on. To convert from decimal to binary, use double dabble for integers and the multiply-by-2 method for fractions.

The base of a number system equals the number of digits it uses. The decimal number system has a base of 10, while the binary number system has a base of 2. The octal number system has a base of 8. A useful model for counting is the octal odometer. When a display wheel turns from 7 back to 0, it sends a carry to the next-higher wheel.

Hexadecimal numbers have a base of 16. The model for counting is the hexadecimal odometer, whose wheels reset and carry beyond F. Hexadecimal numbers are easy to convert mentally into their binary equivalents. For this reason, people prefer using hexadecimal numbers because they are much shorter than the corresponding binary numbers.

The ASCII code is an alphanumeric code widely used for transferring data into and out of a computer. This 7-bit code is used to represent alphabet letters, numbers, and other symbols. The excess-3 code and the Gray code are two other codes that are used.

A logic pulser can temporarily change the state of a node under test. If the original state is low, the logic pulser drives the node briefly into the high state. If the state is high, the logic pulser drives the node briefly

into the low state. The output impedance of a logic pulser is so low that it can drive almost any normal node in a logic circuit. When a node is shorted to ground or to the supply voltage, the logic pulser is unable to change the voltage level; this is a confirmation of the shorted condition.

Parity code, Checksum code, and CRC code have been discussed for error detection code and Hamming code for error detection and correction. These techniques are used to ensure that data is correct and has not been corrupted, either by hardware failures or by noise occurring during transmission or a data read operation from memory.

GLOSSARY

- ***base*** The number of digits or basic symbols in a number system. The decimal system has a base of 10 because it uses 10 digits. Binary has a base of 2, octal a base of 8, and hexadecimal a base of 16.
- ***binary*** Refers to a number system with a base of 2, that is, containing two digits.
- ***bit*** An abbreviated form of binary digit. Instead of saying that 10110 has five binary digits, we can say that it has 5 bits.
- ***byte*** A binary number with 8 bits.
- ***checksum code*** A error detection code generating sum of a block of data.
- ***CRC code*** Cyclic Redundancy Code is a polynomial key based error detection code.
- ***digit*** A basic symbol used in a number system. The decimal system has 10 digits, 0 through 9.
- ***error detection and correction*** A method of detection of error in a group of bits and correction of the same.
- ***hamming code*** A parity bit based error detection and correction code.
- ***hexadecimal*** Refers to number system with a base of 16. The hexadecimal system has digits 0 through 9, followed by A through F.
- ***logic pulser*** A troubleshooting device that generates brief voltage pulses. The typical logic pulser has a push-button switch that produces a single pulse for each closure. More advanced logic pulsers can generate a pulse train with a specified number of pulses.
- ***nibble*** An binary number with 4 bits.
- ***octal*** Refers to a number system with a base of 8, that is, one that uses 8 digits. Normally, these are 0, 1, 2, 3, 4, 5, 6, and 7.
- ***parity code*** An error detection code using one additional parity bit.
- ***weight*** Refers to the decimal value of each digit position of a number. For decimal numbers, the weights are 1, 10, 100, 1000, ..., working from the decimal point to the left. For binary numbers the weights are 1, 2, 4, 8, ... to the left of the binary point. With octal numbers, the weights become 1, 8, 64, ... to the left of the octal point.

PROBLEMS

Section 5.1

5.1 What is the binary number that follows 01101111?

5.2 How many bits are there in 2 bytes?

5.3 How many nibbles are there in each of these:

a. 1001

b. 11110000

c. 110011110000
d. 1111000011001001

Section 5.2

5.4 Give the decimal equivalents for each of the following binary numbers:
a. 110101 b. 11001.011
5.5 Convert the following binary numbers to their decimal equivalents:
a. 1011 1100 b. 1111 1111
5.6 What is the decimal equivalent of 1000 1100 1011 0011?
5.7 A computer has 128K of memory. How many bytes does this represent?

Section 5.3

5.8 Convert the following decimal numbers to binary numbers: 24, 65, and 106.
5.9 What binary number does decimal 268 stand for?
5.10 Convert decimal 108.364 to a binary number.
5.11 Calculate the binary equivalent for 5280.

Section 5.4

5.12 Convert the following octal numbers to decimal equivalents:
a. 65 b. 216
c. 4073
5.13 What is the decimal equivalent of octal 325.736?
5.14 Convert these decimal numbers to octal numbers:
a. 4096 b. 65535
5.15 What is the octal equivalent of decimal 324.987?
5.16 Convert the following octal numbers to binary numbers: 34, 567, 4673.
5.17 Convert the following binary numbers to octal numbers:
a. 10101111
b. 1101.0110111
c. 1010011.101101

Section 5.5

5.18 What are the hexadecimal numbers that follow each of these:
a. ABCD b. 7F3F
c. BEEF
5.19 Convert the following hexadecimal numbers to binary numbers:
a. E5 b. B4D
c. 7AF4
5.20 Convert these binary numbers into hexadecimal numbers:
a. 1000 1100 b. 0011 0111
c. 1111 0101 0110
5.21 Convert hexadecimal 2F59 to its decimal equivalent.
5.22 What is the hexadecimal equivalent of decimal 62359?
5.23 Give the value of $Y_3Y_2Y_1Y_0$ in Fig. 5.8 for each of these:
a. All switches are open
b. Switch 4 is closed
c. Switch A is closed
d. Switch F is closed
5.24 A computer has the following hexadecimal contents stored at the addresses shown:

Address	Hexadecimal contents
2000	D5
2001	AA
2002	96
2003	DE
2004	AA
2005	EB

What are the binary contents at each address?

Section 5.6

5.25 Give the ASCII code for each of these:
a. 7 b. W
c. f d. y
5.26 Suppose that you type LIST with an ASCII keyboard. What is the binary output as you strike each letter.

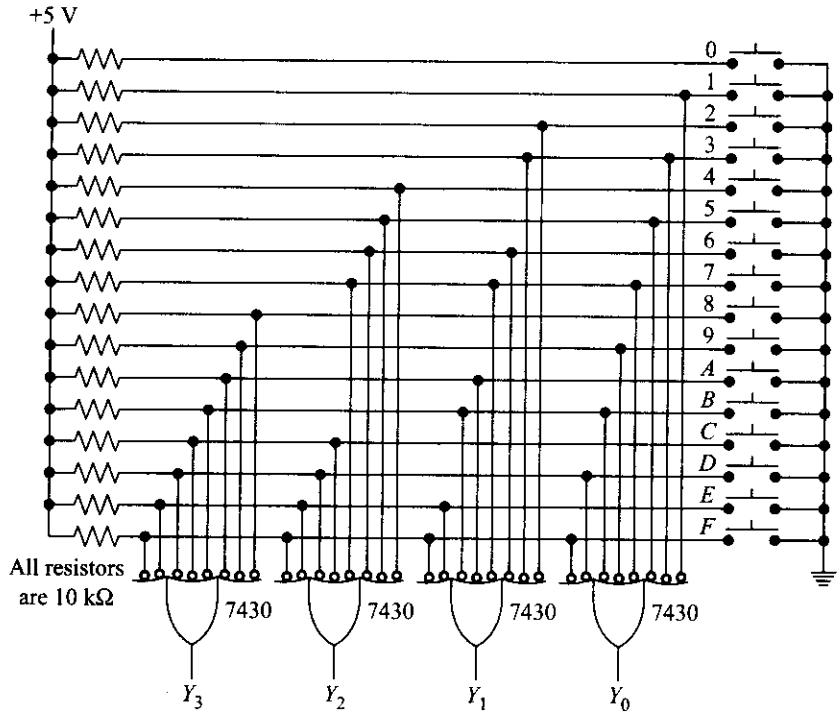


Fig. 5.8

5.27 In Example 5.15, a computer sends the word HELLO to another computer. The characters are coded in ASCII with an odd-parity bit. Here is how the word is stored in the memory of the receiving computer:

Address	Alphanumeric	Hexadecimal contents
2000	H	C8
2001	E	45
2002	L	4C
2003	L	4C
2004	O	4F

The transmitting computer then sends the word GOODBYE. Show how this word is stored in the receiving computer. Use a starting address of 2000 and include a parity bit.

Section 5.7

5.28 Express decimal 5280 in excess-3 code.

5.29 Here is an excess-3 number:

0110 1001 1100 0111

What is the decimal equivalent?

Section 5.8

5.30 What is the Gray code for decimal 8?

5.31 Convert Gray number 1110 to its BCD equivalent.

Section 5.9

5.32 Figure 5.9 shows the decimal-to-BCD encoder discussed in Sec. 4.6. Answer the following questions:

a. If all switches are open and the logic pulser is inactive, what voltage level does the logic probe indicate?

b. If switch 6 is closed and the logic pulser is inactive, what does the logic probe indicate?

c. If all switches are open and the logic pulser is activated, what does the logic probe do?

5.33 The push-button switch of the logic pulser shown in Fig. 5.9 is pressed. Suppose that the logic probe is initially dark and remains dark. Indicate which of the following are possible sources of trouble:

a. 74147 defective
b. Pin 9 shorted to ground
c. Pin 9 shorted to +5 V
d. Pin 10 shorted to ground

5.34 The instruction register shown in Fig. 5.10 on the next page is a logic circuit that stores a 16-bit number, $I_{15} \cdots I_0$. The first 4 bits, $I_{15} \cdots I_{12}$, are decoded by a 4 to 16-line decoder. Determine whether the logic probe indicates low, high, or blink for each of these conditions:

a. $I_{15} \cdots I_{12} = 0000$ and logic pulser inactive
b. $I_{15} \cdots I_{12} = 1000$ and logic pulser inactive
c. $I_{15} \cdots I_{12} = 1000$ and logic pulser active
d. $I_{15} \cdots I_{12} = 1111$ and logic pulser active

5.35 The logic pulser and logic probe shown in Fig. 5.10 are used to check the pins of the 7404 for stuck states. Suppose pin 8 is stuck in the high state. Indicate which of the following are possible sources of trouble:

a. No supply voltage anywhere in circuit
b. Pin 1 of IC2 shorted to ground
c. Pin 2 of IC4 shorted to the supply voltage
d. Pin 3 of IC5 shorted to ground
e. Pin 4 of IC8 shorted to the supply voltage

Section 5.10

5.36 Find Hamming code of data 11001.
5.37 Find Hamming code of data 1000111.
5.38 If an error occurs in the 3rd data bit, how will it be corrected for data of problem 5.37?
5.39 How many parity bits are needed to Hamming code (a) 16-bit data and (b) 24-bit data.

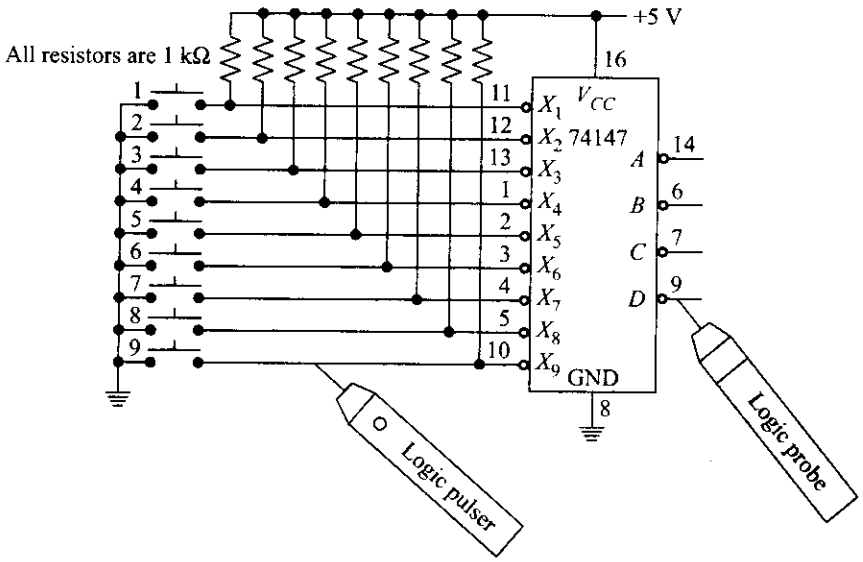


Fig. 5.9

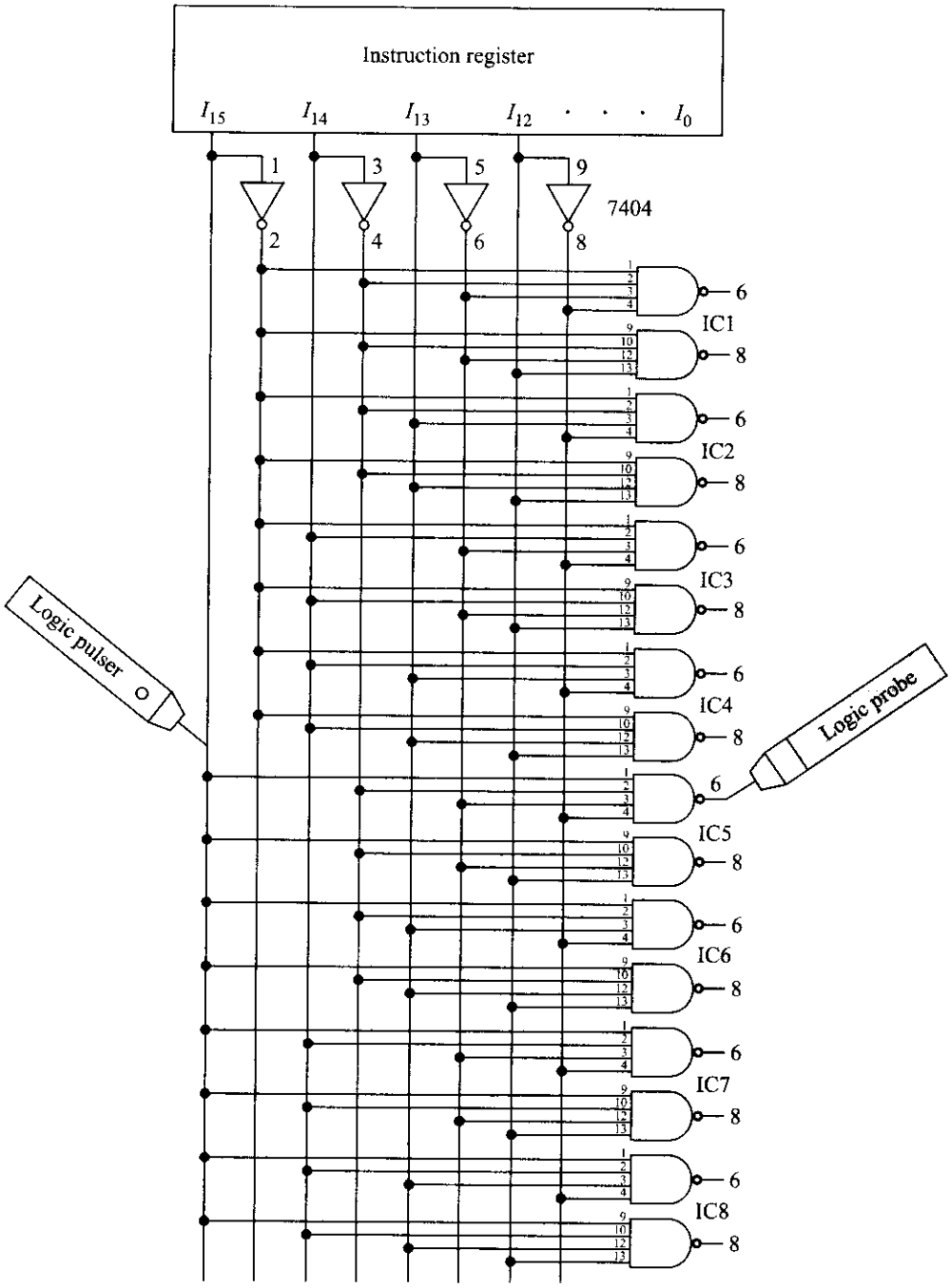

Instruction register
I_{15} I_{14} I_{13} I_{12} . . . I_0
1 3 5 9
2 4 6 8
7404
IC1
IC2
IC3
IC4
IC5
IC6
IC7
IC8
6
8
Logic pulser
Logic probe

Fig. 5.10

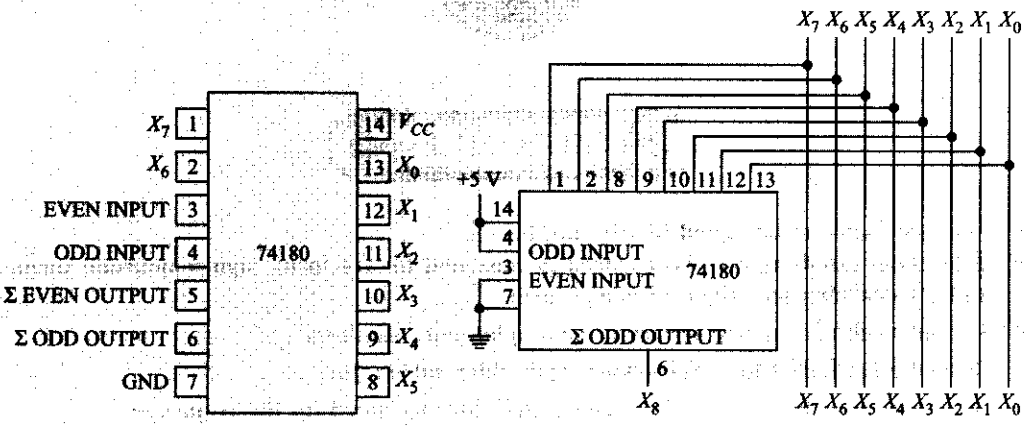
LABORATORY EXPERIMENT

AIM: The aim of this experiment is to generate and check parity code.

Theory: The parity code is obtained by exclusive-OR of data bits. The even parity makes the number of 1s even after the addition of the parity code while odd parity maintains it as odd. The parity in the bit streams, even or odd, is also checked by exclusive-OR of incoming data. Thus the same circuit can be used both for parity generation and checking after appropriate configuration.

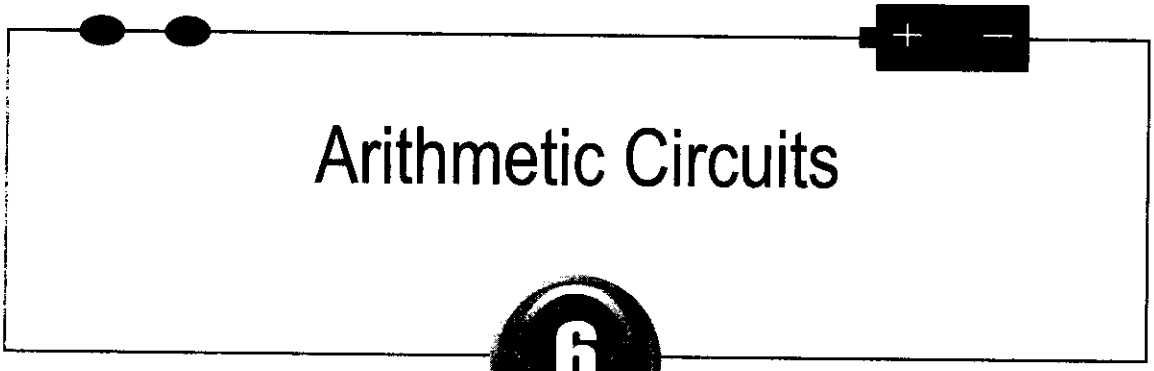
Apparatus: 5 VDC Power supply, Multimeter, and Bread Board

Work element: Verify the truth table of IC 74180, the 8-bit parity generator/checker. Connect it as shown to use it as parity generator. Submit 5 different numbers and check the parity of the coded data, i.e. data plus parity bit. Configure it in such a way that it becomes a parity checker and then check the parity of these 5 numbers. IC 7486 is a quad 2-input exclusive-OR gate with pin configuration similar to 7400 or 7408. Use this to generate parity and compare the result with 74180. Finally, find how 7- and 9-bit long data can be parity coded.



Answers to Self-tests

1. 1101
2. 9
3. Four
4. 18
5. 100011
6. $2^9 = 512$
7. *Double dabble* is a method for converting decimal numbers to binary numbers.
8. 4,095
9. 1111 1111
10. 0, 1, 2, 3, 4, 5, 6, and 7
11. 7 (octal) and 7 (decimal)
12. 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F
13. 0011 1100
14. 60
15. ASCII stands for American Standard Code for Information Interchange, a code used to represent alphanumeric information.
16. @
17. 010 0101
18. No
19. Cycle Redundancy Code
20. It can detect as well as correct one-bit error.
21. Additional bits to be included with data bits for this purpose.



Arithmetic Circuits

- **Add and subtract unsigned binary numbers**
- **Show how numbers are represented in unsigned binary form, sign-magnitude form, and 2's complement (signed binary) form**
- **Add and subtract signed binary (2's complement) numbers**
- **Describe the half-adder, full-adder, and adder-subtractor**
- **Design a fast adder circuit that user parallelism to speed up the responses**
- **Describe how an Arithmetic Logic Unit can be operated**
- **Explain the means by which multiplication and division are performed on typical 8-bit microprocessors**

Circuits that can perform binary addition and subtraction are constructed by combining logic gates. These circuits are used in the design of the arithmetic logic unit (ALU). The electronic circuits are capable of very fast switching action, and thus an ALU can operate at high clock rates. For instance, the addition of two numbers can be accomplished in a matter of nanoseconds! This chapter begins with binary addition and subtraction, then presents two different methods for representing negative numbers. You will see how an exclusive OR gate is used to construct a half-adder and a full-adder. You will see how to construct an 8-bit adder-subtracter using a popular IC. A technique to design a fast adder is discussed in detail followed by discussion on a multifunctional device called Arithmetic Logic Unit or ALU. Finally, an outline to perform binary multiplication and division is also presented.

6.1 BINARY ADDITION

Numbers represent physical quantities. Table 6.1 shows the decimal digits and the corresponding amount of pebbles. Digit 2 stands for two pebbles (••), 5 for five pebbles (•••••), and so on. Addition represents the combining of physical quantities. For instance:

$$2 + 3 = 5$$

symbolizes the combining of two pebbles with three pebbles to obtain a total of five pebbles. Symbolically, this is expressed

•• + ••• = •••••

Table 6.1 The Decimal Digits

Pebbles	*Symbol*
None	0
•	1
••	2
•••	3
••••	4
•••••	5
••••••	6
•••••••	7
••••••••	8
•••••••••	9

Four Cases to Remember

Computer circuits don't process decimal numbers; they process binary numbers. Before you can understand how a computer performs arithmetic, you have to learn how to add binary numbers. Binary addition is the key to binary subtraction, multiplication, and division. So, let's begin with the four most basic cases of binary addition:

$$0 + 0 = 0 \quad (6.1)$$

$$0 + 1 = 1 \quad (6.2)$$

$$1 + 0 = 1 \quad (6.3)$$

$$1 + 1 = 10 \quad (6.4)$$

Equation (6.1) is obvious; so are Eqs. (6.2) and (6.3) because they are identical to decimal addition. The fourth case, however, may bother you. If so, you don't understand what Eq. (6.4) represents in the physical world. Equation (6.4) represents the combining of one pebble and one pebble to obtain a total of two pebbles:

• + • = ••

Since binary 10 stands for ••, the binary equation

$$1 + 1 = 10$$

makes perfect sense. From now on, remember that numbers, whether binary, decimal, octal, or hexadecimal are codes for physical amounts. If you're in doubt about the meaning of a numerical equation, convert the numbers to pebbles to see if the two sides of the equation are equal.

Subscripts

The foregoing discussion brings up the idea of *subscripts*. Since we already have discussed four kinds of numbers (decimal, binary, octal, and hexadecimal), we have four different ways to code physical quantities. How do we know which code is being used? In other words, how can we tell when 10 is a decimal, binary, octal, or hexadecimal number?

Most of the time, it's clear from the discussion which kind of numbers are involved. For instance, if we have been discussing nothing but binary numbers for page after page, you can count on the next 10 being

binary 10, which represents •• in the physical world. On the other hand, if a discussion uses more than one type of number, it may be helpful to use subscripts for the base as follows:

$$2 \rightarrow \text{binary}$$
$$8 \rightarrow \text{octal}$$
$$10 \rightarrow \text{decimal}$$
$$16 \rightarrow \text{hexadecimal}$$

For instance, 11_2 represents binary 11, 23_8 stands for octal 23, 45_{10} for decimal 45, and $F4_{16}$ for hexadecimal F4. With the subscripts in mind, the following equations should make perfect sense:

$$1_2 + 1_2 = 10_2$$
$$7_8 + 1_8 = 10_8$$
$$9_{10} + 1_{10} = 10_{10}$$
$$F_{16} + 1_{16} = 10_{16}$$

Larger Binary Numbers

Column-by-column addition applies to binary as well as decimal numbers. For example, suppose you have this problem in binary addition:

$$\begin{array}{r} 11100 \\ +\,11010 \\ \hline ? \end{array}$$

Start by adding the least-significant column to get

$$\begin{array}{r} 11100 \\ +\,11010 \\ \hline 0 \end{array}$$

Next, add the bits in the second column as follows:

$$\begin{array}{r} 11100 \\ +\,11010 \\ \hline 10 \end{array}$$

The third column gives

$$\begin{array}{r} 11100 \\ +\,11010 \\ \hline 110 \end{array}$$

The fourth column results in

$$\begin{array}{rr} \text{Carry} \rightarrow & 1 \\ & 11100 \\ & +\,11010 \\ \hline & 0110 \end{array}$$

Notice the carry into the final column; this carry occurs because 1 + 1 = 10. As in decimal addition, you write down the 0 and carry the 1 to the next-higher column.

Finally, the last column gives

$$\begin{array}{rr} \text{Carry} \rightarrow & 1 \\ & 11100 \\ & +\,11010 \\ \hline & 110110 \end{array}$$

In the last column, 1 + 1 + 1 = 10 + 1 = 11.

8-Bit Arithmetic

That's all there is to binary addition. If you can remember the four basic rules, you can add column by column to find the sum of two binary numbers, regardless of how long they may be. In first-generation microcomputers (Apple II, TRS-80, etc.), addition is done on two 8-bit numbers such as

$$\begin{array}{r} A_7A_6A_5A_4 \quad A_3A_2A_1A_0 \\ +\,B_7B_6B_5B_4 \quad B_3B_2B_1B_0 \\ \hline ? \end{array}$$

The most-significant bit (MSB) of each number is on the left, and the least-significant bit (LSB) is on the right. For the first number, A_7 is the MSB and A_0 is the LSB. For the second number, B_7 is the MSB and B_0 is the LSB. Try to remember the abbreviations MSB and LSB because they are used frequently in computer discussions.

Example 6.1 Add these 8-bit numbers: 0101 0111 and 0011 0101. Then, show the same numbers in hexadecimal notation.

Solution This is the problem:

$$\begin{array}{r} 0101 \quad 0111 \\ +0011 \quad 0101 \\ \hline ? \end{array}$$

If you add the bits in each column as previously discussed, you will obtain

$$\begin{array}{r} 0101 \quad 0111 \\ +0011 \quad 0101 \\ \hline 1000 \quad 1100 \end{array}$$

Many people prefer hexadecimal notation because it's a faster code to work with. Expressed in hexadecimal numbers, the foregoing addition is

$$\begin{array}{r} 57 \\ +35 \\ \hline 8C \end{array}$$

Often, the letter H is used to signify hexadecimal numbers, so the foregoing addition may be written as

$$\begin{array}{r} 57H \\ +35H \\ \hline 8CH \end{array}$$

Example 6.2 Add these 16-bit numbers: 0000 1111 1010 1100 and 0011 1000 0111 1111. Show the corresponding hexadecimal and decimal numbers.

Solution Start at the right and add the bits, column by column:

Binary	Hexadecimal	Decimal
0000 1111 1010 1100	0FACH	4,012
+ 0011 1000 0111 1111	+ 387FH	+ 14,463
0100 1000 0010 1011	482BH	18,475

(*Note:* Remember Appendix 1; it takes most of the work out of conversions between number systems.)

Example 6.3 Repeat Example 6.2, showing how a first-generation microcomputer does the addition.

Solution First-generation microcomputers like the Apple II have an 8-bit *microprocessor* (a digital IC that performs binary arithmetic on 8-bit numbers). To add 16-bit numbers, a first-generation microcomputer adds the lower 8 bits in one operation and then the upper 8 bits in another operation.

Here is how it works for numbers of the preceding example. The original problem is

```
Upper bytes      Lower bytes
     ↓                ↓
 0000  1111       1010  1100
+0011  1000       0111  1111
-------------------------------
            ?
```

The microcomputer starts by adding the lower bytes:

```
 1010 1100
+0111 1111
----------
10010 1011
```

Notice the carry into the final column. The microcomputer will store the lower byte (0010 1011). Then, it will do another addition of the upper bytes, plus the carry, as follows:

```
          1 ← Carry
 0000  1111
+0011  1000
-----------
 0100  1000
```

The microcomputer then stores the upper byte. To output the total answer, the microcomputer pulls the upper and lower sums out of its memory to get

0100 1000 0010 1011

which is equivalent to 482BH or 18,475, the same as we found in the preceding example.

SELF-TEST

1. Write the four rules for binary addition.
2. What kind of number is 179FH?
3. What is the meaning of 111_2? Of 111_{10}?

6.2 BINARY SUBTRACTION

Let's begin with four basic cases of binary subtraction:

$$0 - 0 = 0 \tag{6.5}$$
$$1 - 0 = 1 \tag{6.6}$$
$$1 - 1 = 0 \tag{6.7}$$
$$10 - 1 = 1 \tag{6.8}$$

Equations (6.5) to (6.7) are easy to understand because they are identical to decimal subtraction. The fourth case will disturb you if you have lost sight of what it really means. Back in the physical world, Eq. (6.4) represents

●● – ● = ●

Two pebbles minus one pebble equals one pebble.

For larger binary numbers, subtract column by column, the same as you do with decimal numbers. This means that you sometimes have to borrow from the next-higher column. Here is an example:

$$\begin{array}{r} 1101 \\ -1010 \\ \hline ? \end{array}$$

Subtract the L'SBs to get

$$\begin{array}{r} 1101 \\ -1010 \\ \hline 1 \end{array}$$

To subtract the bits of the second column, borrow from the next-higher column to obtain

$$\begin{array}{rr} \text{Borrow} \rightarrow & 1 \\ & 1001 \\ & -1010 \\ \hline & 1 \end{array}$$

In the second column from the right, subtract as follows: 10 – 1 = 1, to get

$$\begin{array}{rr} \text{Borrow} \rightarrow & 1 \\ & 1001 \\ & -1010 \\ \hline & 11 \end{array}$$

Then subtract the remaining columns:

$$\begin{array}{rr} \text{Borrow} \rightarrow & 1 \\ & 1001 \\ & -1010 \\ \hline & 0011 \end{array}$$

After you get used to it, binary subtraction is no more difficult than decimal subtraction. In fact, it's easier because there are only four basic cases to remember.

Example 6.4 Show the binary subtraction of 125_{10} from 200_{10}.

Solution First, use Appendix 1 to convert the numbers as follows:

$$200 \rightarrow \text{C8H} \rightarrow 1100\ \ 1000$$
$$125 \rightarrow \text{7DH} \rightarrow 0111\ \ 1101$$

So, here is the problem:

$$\begin{array}{r} 1100\ \ 1000 \\ -0111\ \ 1101 \\ \hline ? \end{array}$$

Column-by-column subtraction gives:

$$\begin{array}{r} 1100\ \ 1000 \\ -0111\ \ 1101 \\ \hline 0100\ \ 1011 \end{array}$$

In hexadecimal notation, the foregoing appears as

$$\begin{array}{r} \text{C8H} \\ -\text{7DH} \\ \hline \text{4BH} \end{array}$$

SELF-TEST

4. Write the four rules for binary subtraction.

6.3 UNSIGNED BINARY NUMBERS

In some applications, all data is either positive or negative. When this happens, you can forget about + and – signs, and concentrate on the *magnitude* (absolute value) of numbers. For instance, the smallest 8-bit number is 0000 0000, and the largest is 1111 1111. Therefore, the total range of 8-bit numbers is

0000 0000 (00H)

to

1111 1111 (FFH)

This is equivalent to a decimal 0 to 255. As you can see, we are not including + or – signs with these decimal numbers.

With 16-bit numbers, the total range is

0000 0000 0000 0000 (0000H)

to

1111 1111 1111 1111 (FFFFH)

which represents the magnitude of all decimal numbers from 0 to 65,535.

Data of the foregoing type is called *unsigned binary* because all of the bits in a binary number are used to represent the magnitude of the corresponding decimal number. You can add and subtract unsigned binary

numbers, provided certain conditions are satisfied. The following examples will tell you more about unsigned binary numbers.

Limits

First-generation microcomputers can process only 8 bits at a time. For this reason, there are certain restrictions you should be aware of. With 8-bit unsigned arithmetic, all magnitudes must be between 0 and 255. Therefore, each number being added or subtracted must be between 0 and 255. Also, the answer must fall in the range of 0 to 255. If any magnitudes are greater than 255, you should use 16-bit arithmetic, which means operating on the lower 8 bits first, then on the upper 8 bits (see Example 6.3).

Overflow

In 8-bit arithmetic, addition of two unsigned numbers whose sum is greater than 255 causes an *overflow*, a carry into the ninth column. Most microprocessors have a logic circuit called a *carry flag*; this circuit detects a carry into the ninth column and warns you that the 8-bit answer is invalid (see Example 6.7).

Example 6.5 Show how to add 150_{10} and 85_{10} with unsigned 8-bit numbers.

Solution With Appendix 1, we obtain

$$150 \rightarrow 96H \rightarrow 1001\ 0110$$
$$85 \rightarrow 55H \rightarrow 0101\ 0101$$

Next, we can add these unsigned numbers to get

1001	0110	96H
+0101	0101	+55H
1110	1011	EBH

Again, Appendix 1 gives

$$1110\ \ 1011 \rightarrow EBH \rightarrow 235$$

Example 6.6 Show how to subtract 85_{10} from 150_{10} with unsigned 8-bit numbers.

Solution Use the same binary numbers as in the preceding example, but subtract to get

1001	0110	96H
−0101	0101	−55H
0100	0001	41H

Again, Appendix 1 gives

$$0100\ \ 0001 \rightarrow 41H \rightarrow 65$$

Example 6.7 In the two preceding examples, everything was well behaved because both decimal answers were between 0 and 255. Now, you will see how an overflow can occur to produce an invalid 8-bit answer.

Show the addition of 175_{10} and 118_{10} using unsigned 8-bit numbers.

Solution

$$\begin{array}{r} 175 \\ +118 \\ \hline 293 \end{array}$$

The answer is greater than 255. Here is what happens when we try to add 8-bit numbers:

Appendix 1 gives

$$175 \rightarrow \text{BFH} \rightarrow 1010\ \ 1111$$
$$118 \rightarrow \text{76H} \rightarrow 0111\ \ 0110$$

An 8-bit microprocessor adds like this:

$$\begin{array}{rrr} & 1010\ \ 1111 & \text{AFH} \\ & +\,0111\ \ 0110 & +\ \text{76H} \\ \hline \text{Overflow} \rightarrow & 1\ 0010\ \ 0101 & 125\text{H} \end{array}$$

With 8-bit arithmetic, only the lower 8 bits are used. Appendix 1 gives

$$0010\ \ 0101 \rightarrow 25\text{H} \rightarrow 37$$

As you see, the 8-bit answer is wrong. It is true that if you take the overflow into account, the answer is valid, but then you no longer are using 8-bit arithmetic. The point is that somebody (the programmer) has to worry about the possibility of an overflow and must take steps to correct the final answer when an overflow occurs. If you study assembly-language programming, you will learn more about overflows and what to do about them.

In summary, 8-bit arithmetic circuits can process decimal numbers between 0 and 255 only. If there is any chance of an overflow during an addition, the programmer has to write instructions that look at the carry flag and use 16-bit arithmetic to obtain the final answer. This means operating on the lower 8 bits, and then the upper 8 bits and the overflow (as done in Example 6.3).

SELF-TEST

5. What is the carry flag in a microprocessor?
6. What is the largest decimal number that can be represented with an 8-bit unsigned binary number?

6.4 SIGN-MAGNITUDE NUMBERS

What do we do when the data has positive and negative values? The answer is important because it determines how complicated the arithmetic circuits must be. The negative decimal numbers are –1, –2, –3, and so on. The magnitude of these numbers is 1, 2, 3, and so forth. One way to code these as binary numbers is to convert the magnitude to its binary equivalent and prefix the sign. With this approach, the sequence –1, –2, and –3 becomes –001, –010, and –011. Since everything has to be coded as strings of 0s and 1s, the + and – signs also have to be represented in binary form. For reasons given soon, 0 is used for the + sign and 1 for the – sign. Therefore, –001, –010, and –011 are coded as 1001, 1010, and 1011.

The foregoing numbers contain a sign bit followed by magnitude bits. Numbers in this form are called *sign-magnitude numbers*. For larger decimal numbers, you need more than 4 bits. But the idea is still the same: the MSB always represents the sign, and the remaining bits always stand for the magnitude. Here are

some examples of converting sign-magnitude numbers:

```
+7   → 0000 0111
−16  → 1001 0000
+25  → 0000 0000 0001 1001
−128 → 1000 0000 1000 0000
```

Range of Sign-Magnitude Numbers

As you know, the unsigned 8-bit numbers cover the decimal range of 0 to 255. When you use sign-magnitude numbers, you reduce the largest magnitude from 255 to 127 because you need to represent both positive and negative quantities. For instance, the negative numbers are

1000 0001 (−1)

to

1111 1111 (−127)

The positive numbers are

0000 0001 (+1)

to

0111 1111 (+127)

The largest magnitude is 127, approximately half of what is for unsigned binary numbers. As long as your input data is in the range of −127 to +127, you can use 8-bit arithmetic. The programmer still must check sums for an overflow because all 8-bit answers are between −127 and +127.

If the data has magnitudes greater than 127, then 16-bit arithmetic may work. With 16-bit numbers, the negative numbers are from

1000 0000 0000 0001 (−1)

to

1111 1111 1111 1111 (−32, 767)

and the positive numbers are from

0000 0000 0000 0001 (+1)

to

0111 1111 1111 1111 (+32, 767)

Again, you can see that the largest magnitude is approximately half that of unsigned binary numbers. Unless you actually need + and − signs to represent your data, you are better off using unsigned binary.

The main advantage of sign-magnitude numbers is their simplicity. Negative numbers are identical to positive numbers, except for the sign bit. Because of this, you can easily find the magnitude by deleting the sign bit and converting the remaining bits to their decimal equivalents. Unfortunately, sign-magnitude numbers have limited use because they require complicated arithmetic circuits. If you don't have to add or subtract the data, sign-magnitude numbers are acceptable. For instance, sign-magnitude numbers are often used in *analog-to-digital* (A/D) conversions (explained in a latter chapter).

SELF-TEST

7. What is the decimal number range that can be represented with an 8-bit sign-magnitude binary number?
8. In sign-magnitude form, what is the decimal value of 1000 1101? Of 0000 1101?

6.5 2'S COMPLEMENT REPRESENTATION

There is a rather unusual number system that leads to the simplest logic circuits for performing arithmetic. Known as *2's complement representation,* this system dominates microcomputer architecture and programming.

1's Complement

The 1's complement of a binary number is the number that results when we complement each bit. Figure 6.1 shows how to produce the 1's complement with logic circuits. Since each bit drives an inverter, the 4-bit output is the 1's complement of the 4-bit input. For instance, if the input is

$$X_3X_2X_1X_0 = 1000$$

the 1's complement is

$$\overline{X}_3\overline{X}_2\overline{X}_1\overline{X}_0 = 0111$$

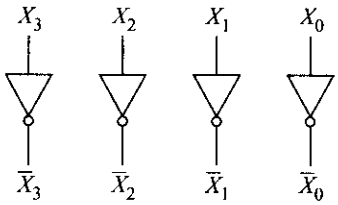


Fig. 6.1 **Inverters produce the 1's complement.**

The same principle applies to binary numbers of any length: complement each bit to obtain the 1's complement. More examples of 1's complements are

$$1010 \rightarrow 0101$$
$$1110\ \ 1100 \rightarrow 0001\ \ 0011$$
$$0011\ \ 1111\ \ 0000\ \ 0110 \rightarrow 1100\ \ 0000\ \ 1111\ \ 1001$$

2's Complement

The 2's complement is the binary number that results when we add 1 to the 1's complement. As a formula:

$$\text{2's complement} = \text{1's complement} + 1$$

For instance, to find the 2's complement of 1011, proceed like this:

$$1011 \rightarrow 0100 \qquad \text{(1's complement)}$$
$$0100 + 1 = 0101 \qquad \text{(2's complement)}$$

Instead of adding 1, you can visualize the next reading on a binary odometer. So, after obtaining the 1's complement 0100, ask yourself what comes next on a binary odometer. The answer is 0101.

Here are more examples of the 2's complements:

Number →	1's complement →	2's complement
1110 1100 →	0001 0011 →	0001 0100
1000 0001 →	0111 1110 →	0111 1111
0011 0110 →	1100 1001 →	1100 1010

Back to the Odometer

The binary odometer is a marvelous way to understand 2's complement representation. By examining the numbers of a binary odometer, we can see how the typical microcomputer represents positive and negative numbers. With a binary odometer, all bits eventually reset to 0s. Some readings before and after a complete reset look like this:

1000	(–8)
1001	(–7)
1010	(–6)
1011	(–5)
1100	(–4)
1101	(–3)
1110	(–2)
1111	(–1)
0000	(0)
0001	(+1)
0010	(+2)
0011	(+3)
0100	(+4)
0101	(+5)
0110	(+6)
0111	(+7)

Binary 1101 is the reading 3 miles before reset, 1110 occurs 2 miles before reset, and 1111 indicates 1 mile before reset. Then, 0001 is the reading 1 mile after reset, 0010 occurs 2 miles after reset, and 0011 indicates 3 miles after reset.

Positive and Negative Numbers

"Before" and "after" are synonymous with "negative" and "positive." Figure 6.2 illustrates this idea with the number line of basic algebra: 0 marks the origin, positive numbers are on the right, and negative numbers are on the left. The odometer readings are the binary equivalents of decimal numbers: 1000 is the binary equivalent of –8, 1001 stands for –7, 1010 stands for –6, and so on.

The odometer readings in Fig. 6.2 demonstrate how positive and negative numbers are stored in a typical microcomputer. Here are two important ideas to notice about these odometer readings. First, the MSB is the sign bit: 0 represents a + sign, and 1 stands for a – sign. Second, the negative numbers in Fig. 6.2 are the 2's complements of the positive numbers, as you can see in the following:

Magnitude	Positive	Negative
1	0001	1111
2	0010	1110
3	0011	1101
4	0100	1100
5	0101	1011
6	0110	1010
7	0111	1001
8	—	1000

Except for the last entry, the positive and negative numbers are 2's complements of each other.

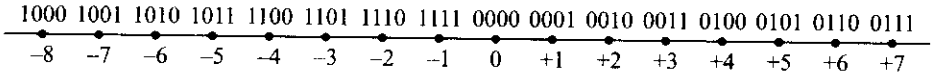


Fig. 6.2 Representing decimal numbers as 2's complements

In other words, you can take the 2's complement of a positive binary number to find the corresponding negative binary number. For instance:

$$3 \rightarrow 0011$$
$$-3 \leftarrow 1101$$

After taking the 2's complement of 0011, we get 1101, which represents –3. The principle also works in reverse:

$$-7 \rightarrow 1001$$
$$+7 \leftarrow 0111$$

After taking the 2's complement of 1001, we obtain 0111, which represents +7.

What does the foregoing mean? It means that taking the 2's complement is equivalent to *negation*, changing the sign of the number. Why is this important? Because it's easy to build a logic circuit that produces the 2's complement. Whenever this circuit takes the 2's complement, the output is the negative of the input. This key idea leads to an incredibly simple arithmetic circuit that can add and subtract.

In summary, here are the things to remember about 2's complement representation:

1. Positive numbers always have a sign bit of 0, and negative numbers always have a sign bit of 1.
2. Positive numbers are stored in sign-magnitude form.
3. Negative numbers are stored as 2's complements.
4. Taking the 2's complement is equivalent to a sign change.

Converting to and from 2's Complement Representation

We need a fast way to express numbers in 2's complement representation. Appendix 2 lists all 8-bit numbers in positive and negative form. You will come to love this Appendix if you have to work a lot with negative numbers. By reading either the positive or negative column, you can quickly convert from decimal to the 2's complement representation, or vice versa.

Here are some examples of using Appendix 2 to convert from decimal to 2's complement representation:

$$+23 \rightarrow 17H \rightarrow 0001\ 0111$$
$$-48 \rightarrow D0H \rightarrow 1101\ 0000$$
$$-93 \rightarrow A3H \rightarrow 1010\ 0011$$

Of course, you can use Appendix 2 in reverse. Here are examples of converting from 2's complement representation to decimal:

$$0111\ 0111 \rightarrow 77H \rightarrow +119$$
$$1110\ 1000 \rightarrow E8H \rightarrow -24$$
$$1001\ 0100 \rightarrow 94H \rightarrow -108$$

A final point. Look at the last two entries in Appendix 2. As you see, +127 is the largest positive number in 2's complement representation, and –128 is the largest negative number. Similarly, in the 4-bit odometer discussed earlier, +7 was the largest positive number, and –8 was the largest negative number. The largest negative number has a magnitude that is one greater than the largest positive number. This *slight asymmetry* of 2's complement representation has no particular meaning, but it is something to keep in mind when we discuss overflows.

Example 6.8 A first-generation microcomputer stores 1 byte at each address or memory location, Show how the following decimal numbers are stored with the use of 2's complement representation: +20, –35, +47, –67, –98, +112, and –125. The first byte starts at address 2000.

Solution With Appendix 2, we have

Address	Binary contents	Hexadecimal contents	Decimal contents
2000	0001 0100	14H	+20
2001	1101 1101	DDH	–35
2002	0010 1111	2DH	+47
2003	1011 1101	BDH	–67
2004	1001 1110	9EH	–98
2005	0111 0000	70H	+112
2006	1000 0011	83H	–125

The computer actually stores binary 0001 0100 at address 2000. Instead of saying 0001 0100, however, we may prefer to say that it stores 14H. To anyone who knows the hexadecimal code, 14H, means the same thing as 0001 0100, but 14H is much easier to say. To the person on the street who knows only the decimal code, we would say that +20 is stored at address 2000.

As you see, understanding computer operation requires knowledge of the different codes being used. Get this into your head, and you are on the way to understanding how computers work.

Example 6.9 Express –19,750 in 2's complement representation. Then show how this number is stored starting at address 2000. Use hexadecimal notation to compress the data.

Solution The number –19,750 is outside the range of Appendix 2, so we have to fall back on Appendix 1. Start by converting the magnitude to binary form. With Appendix 1, we have

$$19{,}750 \rightarrow 4D26H \rightarrow 0100\ 1101\ 0010\ 0110$$

Now, take the 2's complement to obtain the negative value:

$$1011\ 0010\ 1101\ 1001 + 1 = 1011\ 0010\ 1101\ 1010$$

This means that

$$-19{,}750 \rightarrow 1011\ 0010\ 1101\ 1010$$

In hexadecimal notation, this is expressed

$$1011\ 0010\ 1101\ 1010 \rightarrow \text{B2DAH}$$

The memory of a first-generation microcomputer is organized in bytes. Each address or memory location contains 1 byte. Therefore, a first-generation microcomputer has to break a 16-bit number like B2DA into 2 bytes: an upper byte of B2 and a lower byte of DA. The lower byte is stored at the lower address and the upper byte, at the next-higher address like this:

Address	Binary contents	Hexadecimal contents
2000	1101 1010	DA
2001	1011 0010	B2

The same approach, lower byte first and upper byte second, is used with first-generation microcomputers such as the Apple II and TRS-80.

SELF-TEST

9. What is the 1's complement representation of 1101 0110?
10. What is the 2's complement representation of 1101 0110?

6.6 2'S COMPLEMENT ARITHMETIC

Early computers used sign-magnitude numbers for positive and negative values. This led to complicated arithmetic circuits. Then an engineer discovered that 2's complement representation could simplify arithmetic *hardware*. (This refers to the electronic, magnetic, and mechanical devices of a computer.) Since then, 2's complement representation has become a universal code for processing positive and negative numbers.

Help from the Binary Odometer

Addition and subtraction can be visualized in terms of a binary odometer. When you add a positive number, this is equivalent to advancing the odometer reading. When you add a negative number, this has the effect of turning the odometer backward. Likewise, subtraction of a positive number reverses the odometer, but subtraction of a negative number advances it. As you read the following discussion of addition and subtraction, keep the binary odometer in mind because it will help you to understand what's going on.

Addition

Let us take a look at how binary numbers are added. There are four possible cases: both numbers positive, a positive number and a smaller negative number, a negative number and a smaller positive number, and both numbers negative. Let us go through all four cases for a complete coverage of what happens when a computer adds numbers.

Case 1 Both positive. Suppose that the numbers are +83 and +16. With Appendix 2, these numbers are converted as follows:

$$+83 \rightarrow 0101\ \ 0011$$
$$+16 \rightarrow 0001\ \ 0000$$

Then, here is how the addition appears:

+83	0101 0011
+16	+ 0001 0000
99	0110 0011

Nothing unusual happens here. Column-by-column addition produces a binary answer of 0110 0011. Mentally convert this to 63H. Now, look at Appendix 2 to get

$$63H \rightarrow 99$$

This agrees with the decimal sum.

Case 2 Positive and smaller negative. Suppose that the numbers are +125 and –68. With Appendix 2, we obtain

$$+125 \rightarrow 0111\ \ 1101$$
$$-68 \rightarrow 1011\ \ 1100$$

The computer will fetch these numbers from its memory and send them to an adding circuit. The numbers are then added column by column, including the sign bits to get

125	0111 1101
+ (−68)	+1011 1100
57	1 0011 1001 → 0011 1001

With 8-bit arithmetic, you *disregard the final carry* into the ninth column. The reason is related to the binary odometer, which ignores final carries. In other words, when the eighth wheel resets, it does not generate a carry because there is no ninth wheel to receive the carry. You can convert the binary answer to decimal as follows:

0011 1001 → 39H	(mental conversion)
39H → +57	(look in Appendix 2)

Case 3 Positive and larger negative. Let's use +37 and –115. Appendix 2 gives these 2's complement representations:

$$+37 \rightarrow 0010\ \ 0101$$
$$-115 \rightarrow 1000\ \ 1101$$

Then the addition looks like this:

+37	0010 0101
+ (−115)	+1000 1101
−78	1011 0010

Next, verify the binary answer as follows:

$$1011\ \ 0010 \rightarrow \text{B2H} \qquad \text{(mental conversion)}$$
$$\text{B2H} \rightarrow -78 \qquad \text{(look in Appendix 2)}$$

Incidentally, mentally converting to hexadecimal before reference to the appendix is an optional step. Most people find it easier to locate B2H in Appendix 2 than 1011 0010. It only saves a few seconds, but it adds up when you have to do a lot of binary-to-decimal conversions.

Case 4 Both negative. Assume that the numbers are –43 and –78. In 2's complement representation, the numbers are

$$-43 \rightarrow 1101\ \ 0101$$
$$-78 \rightarrow 1011\ \ 0010$$

The addition is

$$\begin{array}{rr} -43 & 1101\ \ 0101 \\ +\ (-78) & +\ 1011\ \ 0010 \\ \hline -121 & 1\ 1000\ \ 0111 \rightarrow 1000\ \ 0111 \end{array}$$

Again, we ignore the final carry because it's meaningless in 8-bit arithmetic. The remaining 8 bits convert as follows:

$$1000\ \ 0111 \rightarrow 83\text{H}$$
$$83\text{H} \rightarrow -121$$

This agrees with the answer we obtained by direct decimal addition.

Conclusion

We have exhausted the possibilities. In every case, 2's complement addition works. In other words, as long as positive and negative numbers are expressed in 2's complement representation, an adding circuit will automatically produce the correct answer. (This assumes the decimal sum is within the –128 to +127 range. If not, you get an overflow, which we will discuss later.)

Subtraction

The format for subtraction is

$$\begin{array}{r} \text{Minuend} \\ -\ \text{Subtrahend} \\ \hline \text{Difference} \end{array}$$

There are four cases: both numbers positive, a positive number and a smaller negative number, a negative number and a smaller positive number, and both numbers negative.

The question now is *how can we use an adding circuit* to do subtraction. By trickery, of course. From algebra, you already know that adding a negative number is equivalent to subtracting a positive number. If we take the 2's complement of the subtrahend, addition of the complemented subtrahend gives the correct answer. Remember that the 2's complement is equivalent to negation. One way to remove all doubt about this critical idea is to analyze the four cases that can arise during a subtraction.

Case 1 Both positive. Suppose that the numbers are +83 and +16. In 2's complement representation, these numbers appear as

$$+83 \rightarrow 0101\ \ 0011$$
$$+16 \rightarrow 0001\ \ 0000$$

To subtract +16 from +83, the computer will send the +16 to a 2's complementer circuit to produce

$$-16 \rightarrow 1111\ \ 0000$$

Then it will add +83 and −16 as follows:

$$\begin{array}{rr} 83 & 0101\ \ 0011 \\ +(-16) & +\ 1111\ \ 0000 \\ \hline 67 & 1\ 0100\ \ 0011 \rightarrow 0100\ \ 0011 \end{array}$$

The binary answer converts like this:

$$0100\ \ 0011 \rightarrow 43H$$
$$43H \rightarrow +67$$

Case 2 Positive and smaller negative. Suppose that the minuend is +68 and the subtrahend is −27. In 2's complement representation, these numbers appear as

$$+68 \rightarrow 0100\ \ 0100$$
$$-27 \rightarrow 1110\ \ 0101$$

The computer sends −27 to a 2's complementer circuit to produce

$$+27 \rightarrow 0001\ \ 1011$$

Then it adds +68 and +27 as follows:

$$\begin{array}{rr} +68 & 0100\ \ 0100 \\ +27 & +\ 0001\ \ 1011 \\ \hline 95 & 0101\ \ 1111 \end{array}$$

The binary answer converts to decimal as follows:

$$0101\ \ 1111 \rightarrow 5FH$$
$$5FH \rightarrow +95$$

Case 3 Positive and larger negative. Let's use a minuend of +14 and a subtrahend of −108. Appendix 2 gives these 2's complement representations:

$$+14 \rightarrow 0000\ \ 1110$$
$$-108 \rightarrow 1001\ \ 0100$$

The computer produces the 2's complement of −108:

$$+108 \rightarrow 0110\ \ 1100$$

Then it adds the numbers like this:

$$\begin{array}{rr} 14 & 0000\ \ 1110 \\ +108 & +\ 0110\ \ 1100 \\ \hline 122 & 0111\ \ 1010 \end{array}$$

The binary answer converts to decimal like this:

$$0111\ \ 1010 \rightarrow 7AH$$
$$7AH \rightarrow +122$$

Case 4 Both negative. Assume that the numbers are –43 and –78. In 2's complement representation, the numbers are

$$-43 \rightarrow 1101\ \ 0101$$
$$-78 \rightarrow 1011\ \ 0010$$

First, take the 2's complement of –78 to get

$$+78 \rightarrow 0100\ \ 1110$$

Then add to obtain

$$\begin{array}{rr} -43 & 1101\ \ 0101 \\ +78 & +\ 0100\ \ 1110 \\ \hline 35 & 1\ 0010\ \ 0011 \rightarrow 0010\ \ 0011 \end{array}$$

Then

$$0010\ \ 0011 \rightarrow 23H$$
$$23H \rightarrow +35$$

Overflow

We have covered all cases of addition and subtraction. As shown, 2's complement arithmetic works and is the standard method used in microcomputers. In 8-bit arithmetic, the only thing that can go wrong is a sum outside the range of –128 to +127. When this happens, there is an overflow into the sign bit, causing a sign change. With the typical microcomputer, the programmer has to write instructions that check for this change in the sign bit.

Let's take a look at overflow problems. Assume that both input numbers are in the range of –128 to +127. If a positive and a negative number are being added, an overflow is impossible because the answer is always less than the larger of the two numbers being added. Trouble can arise only when the arithmetic circuit adds two positive numbers or two negative numbers. Then, it is possible for the sum to be outside the range of –128 to +127. (Subtraction is included in the foregoing discussion because the arithmetic circuit adds the complemented subtrahend.)

Case 1 Two positive numbers. Suppose that the numbers being added are +100 and +50. The decimal sum is +150, so an overflow occurs into the MSB position. This overflow forces the sign bit of the answer to change. Here is how it looks:

$$\begin{array}{rr} 100 & 0110\ \ 0100 \\ +\ 50 & +\ 0011\ \ 0010 \\ \hline 150 & 1001\ \ 0110 \end{array}$$

The sign bit is negative, despite the fact that we added two positive numbers. Therefore, the overflow has produced an incorrect answer.

Case 2 Two negative numbers. Suppose that the numbers are –85 and –97. Then

$$\begin{array}{r} -85 \\ +\,(-97) \\ \hline 182 \end{array} \qquad \begin{array}{r} 1010\ \ 1011 \\ +\ 1001\ \ 1111 \\ \hline 1\ 0100\ \ 1010 \rightarrow 0100\ \ 1010 \end{array}$$

The 8-bit answer is 0100 1010. The sign bit is positive, but we know that the right answer should contain a negative sign bit because we added two negative numbers.

What to Do with an Overflow

Overflows are a software problem, not a hardware problem. (*Software* means a program or list of instructions telling the computer what to do.) The programmer must test for an overflow after each addition or subtraction. A change in the sign bit is easy to detect. All the programmer does is include instructions that compare the sign bits of the two numbers being added. When these are the same, the sign bit of the answer is compared to either of the preceding sign bits. If the sign bit is different, more instructions tell the computer to change the processing to 16-bit arithmetic. You will learn more about overflows, 16-bit arithmetic, and related topics if you study assembly-language programming.

Example 6.10 How would an 8-bit microcomputer process this:

$$\begin{array}{r} 18{,}357 \\ -\,12{,}618 \\ \hline ? \end{array}$$

Solution It would use *double-precision arithmetic*, synonymous with 16-bit arithmetic. This arithmetic is used with 16-bit numbers in this form:

$$X_{15}X_{14}X_{13}X_{12}\quad X_{11}X_{10}X_9X_8\quad X_7X_6X_5X_4\quad X_3X_2X_1X_0$$

Numbers like these have an upper byte $X_{15} \cdots X_8$ and a lower byte $X_7 \cdots X_0$. To perform 16-bit arithmetic, an 8-bit microcomputer has to operate on each byte separately. The idea is similar to Example 6.3, where the lower bytes were added and then the upper bytes.

Here is how it is done. With Appendix 1, we have

$$18{,}357 \rightarrow 47B5H \rightarrow 0100\ \ 0111\ \ 1011\ \ 0101$$
$$12{,}618 \rightarrow 314AH \rightarrow 0011\ \ 0001\ \ 0100\ \ 1010$$

The 2's complement of 12,618 is

$$-12{,}618 \rightarrow CEB6H \rightarrow 1100\ \ 1110\ \ 1011\ \ 0110$$

The addition is carried out in two steps of 8-bit arithmetic. First, the lower bytes are added:

$$\begin{array}{r} 1011\ \ 0101 \\ +\ 1011\ \ 0110 \\ \hline 1\ 0110\ \ 1011 \end{array} \rightarrow X_8X_7X_6X_5X_4\ \ X_3X_2X_1X_0$$

The computer will store $X_7 \cdots X_0$. The carry X_8 is used in the addition of the upper bytes.

Now, the computer adds the upper bytes plus the carry as follows:

$$\begin{array}{r} \qquad\qquad 1 \leftarrow X_8 \\ 0100\ \ 0111 \\ +\ 1100\ \ 1110 \\ \hline 1\ 0001\ \ 0110 \rightarrow 0001\ \ 0110 \end{array}$$

To obtain the final answer, the two 8-bit answers are combined:

0001 0110 0110 1011

Notice that the MSB is 0, which means that the answer is positive. With Appendix 1, we can convert this answer to decimal form:

0001 0110 0110 1011 → 166BH → +5739

SELF-TEST

11. What is the standard method for doing binary arithmetic in nearly all microprocessors?
12. How is 2's complement representation used to perform subtraction?

6.7 ARITHMETIC BUILDING BLOCKS

We are on the verge of seeing a logic circuit that performs 8-bit arithmetic on positive and negative numbers. But first we need to cover three basic circuits that will be used as building blocks. These building blocks are the half-adder, the full-adder, and the controller inverter. Once you understand how these work, it is only a short step to see how it all comes together, that is, how a computer is able to add and subtract binary numbers of any length.

Half-Adder

When we add two binary numbers, we start with the least-significant column. This means that we have to add two bits with the possibility of a carry. The circuit used for this is called a *half-adder*. Figure 6.3 shows how to build a half-adder. The output of the exclusive–OR gate is called the *SUM*, while the output of the AND gate is the *CARRY*. The AND gate produces a high output only when both inputs are high. The exclusive-OR gate produces a high output if either input, but not both, is high. Table 6.2 shows the truth table of a half-adder.

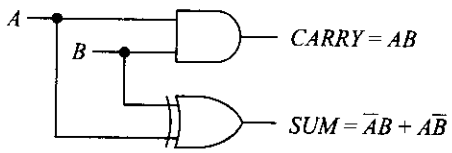


Fig. 6.3 Half-adder

When you examine each entry in Table 6.2, you are struck by the fact that a half-adder performs binary addition.

As you see, the half-adder mimics our brain processes in adding bits. The only difference is the half-adder is about a million times faster than we are.

Table 6.2 Half-adder Truth Table

A	*B*	*CARRY*	*SUM*
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Full-Adder

For the higher-order columns, we have to use a *full-adder*, a logic circuit that can add 3 bits at a time. The third bit is the carry from a lower column. This implies that we need a logic circuit with three inputs and two outputs, similar to the full-adder shown in Fig. 6.4a. (Other designs are possible. This one is the simplest.)

Table 6.3 shows the truth table of a full-adder. You can easily check this truth table for its validity. For instance, CARRY is high in Fig. 6.4a when two or more of the *ABC* inputs are high; this agrees with the CARRY column in Table 6.3. Also, when an odd number of high *ABC* inputs drives the exclusive-OR gate, it produces a high output; this verifies the SUM column of the truth table.

Table 6.3 Full-Adder Truth Table

A	*B*	*C*	*CARRY*	*SUM*
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

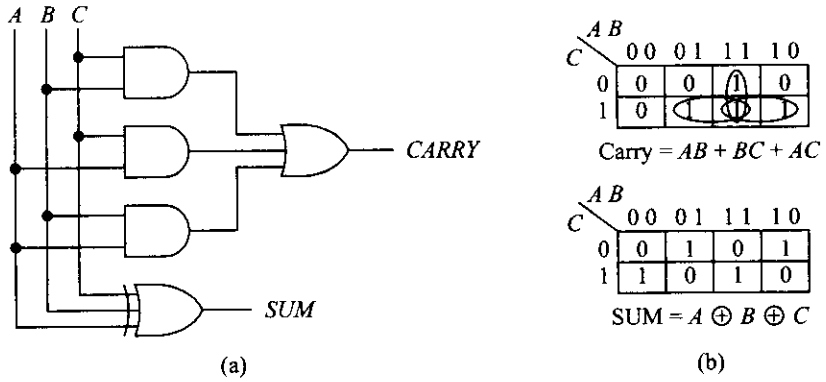


Fig. 6.4 (a) Full-adder, (b) Karnaugh map of Table 6.3

When you examine each entry in Table 6.3, you can see that a full-adder performs binary addition on 3 bits.

From this truth table we get Karnaugh map as shown in Fig. 6.4b that gives following logic equations,

$$\text{CARRY} = AB + BC + AC \quad \text{and} \quad \text{SUM} = A \oplus B \oplus C.$$

A general representation of full-adder which adds i-th bit A_i and B_i of two numbers A and B and takes carry from $(i-1)$th bit could be

$$C_i = A_iB_i + B_iC_{i-1} + A_iC_{i-1} \text{ or } C_i = A_iB_i + (A_i + B_i)C_{i-1} \quad \text{and} \quad S_i = A_i \oplus B_i \oplus C_{i-1}$$

where, C_i and S_i are carry and sum bits generated from the full adder. The second representation of C_i has an interesting meaning. The first term gives, if both A_i and B_i are 1 then $C_i = 1$. The second term gives if any of A_i or B_i is 1 and if there is carry from previous stage, i.e. $C_{i-1} = 1$ then also $C_i = 1$. That this is the case, we can verify from full adder truth table and this understanding is useful in design of fast adder in Section 6.9.

Controlled Inverter

Figure 6.5 shows a *controlled inverter*. When INVERT is low, it transmits the 8-bit input to the output; when INVERT is high, it transmits the 1's complement. For instance, if the input number is

$$A_7 \cdots A_0 = 0110 \quad 1110$$

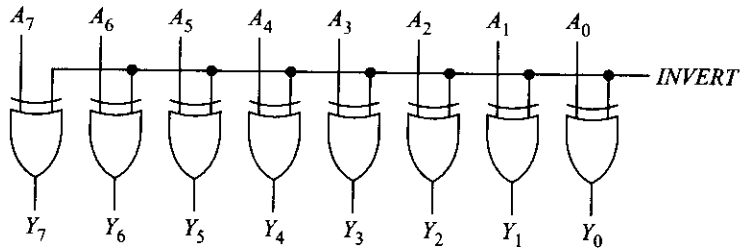


Fig. 6.5 Controlled inverter

a low INVERT produces

$$Y_7 \cdots Y_0 = 0110 \quad 1110$$

But a high INVERT results in

$$Y_7 \cdots Y_0 = 1001 \quad 0001$$

The controlled inverter is important because it is a step in the right direction. During a subtraction, we first need to take the 2's complement of the subtrahend. Then we can add the complemented subtrahend to obtain the answer. With a controlled inverter, we can produce the 1's complement. There is an easy way to get the 2's complement, discussed in the next section. So, we now have all the building blocks: half-adder, full-adder, and controlled inverter.

SELF-TEST

13. What are the inputs and outputs of a half-adder?
14. What are the inputs and outputs of a full-adder?
15. The SUM output of a full-adder is easily implemented using an exclusive-OR gate. (T or F)

6.8 THE ADDER-SUBTRACTER

We can connect full-adders as shown in Fig. 6.6 to add or subtract binary numbers. The circuit is laid out from right to left, similar to the way we add binary numbers. Therefore, the least-significant column is on the right, and the most-significant column is on the left. The boxes labeled FA are full-adders. (Some adding circuits use a half-adder instead of a full-adder in the least-significant column.)

The CARRY OUT from each full-adder is the CARRY IN to the next-higher full-adder. The numbers being processed are $A_7 \cdots A_0$ and $B_7 \cdots B_0$, and the answer is $S_7 \cdots S_0$. With 8-bit arithmetic, the final carry is ignored for reasons given earlier. With 16-bit arithmetic, the final carry is the carry into the addition of the upper bytes.

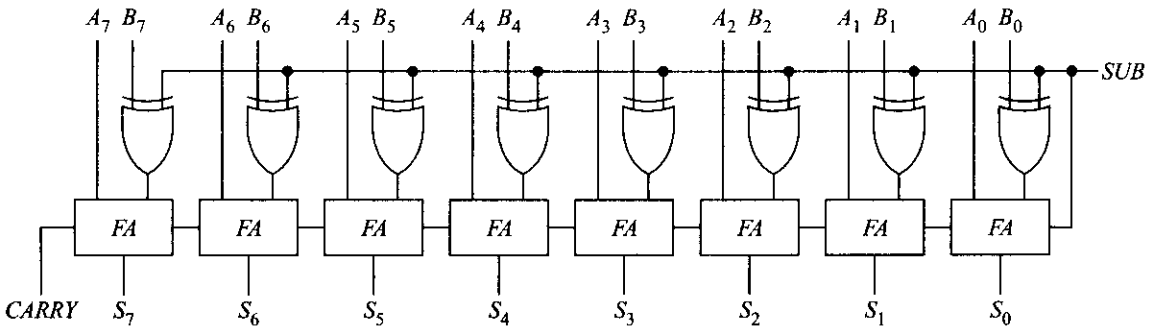


Fig. 6.6 Binary adder-subtracter

Addition

Here is how an addition appears:

$$\begin{array}{ll} A_7A_6A_5A_4 & A_3A_2A_1A_0 \\ +\ B_7B_6B_5B_4 & B_3B_2B_1B_0 \\ \hline S_7S_6S_5S_4 & S_3S_2S_1S_0 \end{array}$$

During an addition, the *SUB* signal is deliberately kept in the low state. Therefore, the binary number $B_7 \cdots B_0$ passes through the controlled inverter with no change. The full-adders then produce the correct output sum. They do this by adding the bits in each column, passing carries to the next higher column, and so on. For instance, starting at the LSB, the full-adder adds A_0, B_0, and SUB. This produces a SUM of S_0 and a CARRY OUT to the next-higher full-adder. The next-higher full-adder then adds A_1, B_1, and the CARRY IN to produce S_1 and a CARRY OUT. A similar addition occurs for each of the remaining full-adders, and the correct sum appears at the output lines.

For instance, suppose that the numbers being added are +125 and –67. Then, $A_7 \cdots A_0 = 0111\ 1101$ and $B_7 \cdots B_0 = 1011\ 1101$. This is the problem:

$$\begin{array}{r} 0111\ 1101 \\ +1011\ 1101 \\ \hline ? \end{array}$$

Since SUB = 0 during an addition, the CARRY IN to the least-significant column is 0:

$$\begin{array}{r} 0 \leftarrow \text{SUB} \\ 0111\ 1101 \\ +1011\ 1101 \\ \hline ? \end{array}$$

The first full-adder performs this addition:

$$0 + 1 + 1 = 0 \quad \text{with a carry of 1}$$

The CARRY OUT of the first full-adder is the CARRY IN to the second full-adder:

$$\begin{array}{r} 1 \leftarrow \text{Carry} \\ 0111\ 1101 \\ +1011\ 1101 \\ \hline 0 \end{array}$$

In the second column

$$1 + 0 + 0 = 1 \quad \text{with a carry of 0}$$

The carry goes to the third full-adder:

$$\begin{array}{r} 0 \leftarrow \text{Carry} \\ 0111\ \ 1101 \\ +1011\ \ 1101 \\ \hline 10 \end{array}$$

In a similar way, the remaining full-adders add their 3 input bits until we arrive at the last full-adder:

$$\begin{array}{r} 1 \leftarrow \text{Carry} \\ 0111\ \ 1101 \\ +1011\ \ 1101 \\ \hline 0011\ \ 1010 \end{array}$$

When the CARRY IN to the MSB appears, the full-adder produces

$$1 + 0 + 1 = 0 \quad \text{with a carry of 1}$$

The addition process ends with a final carry:

$$\begin{array}{r} 0111\ \ 1101 \\ +1011\ \ 1101 \\ \hline 1\ 0011\ \ 1010 \end{array}$$

During 8-bit arithmetic, this last carry is ignored as previously discussed; therefore, the answer is

$$S_7 \cdots S_0 = 0011\ \ 1010$$

This answer is equivalent to decimal +58, which is the algebraic sum of the numbers we started with: +125 and –67.

Subtraction

Here is how a subtraction appears:

$$\begin{array}{r} A_7A_6A_5A_4 \quad A_3A_2A_1A_0 \\ +\ B_7B_6B_5B_4 \quad B_3B_2B_1B_0 \\ \hline S_7S_6S_5S_4 \quad S_3S_2S_1S_0 \end{array}$$

During a subtraction, the SUB signal is deliberately put into the high state. Therefore, the controlled inverter produces the 1's complement of $B_7 \cdots B_0$. Furthermore, because SUB is the CARRY IN to the first full-adder, the circuit processes the data like this:

$$\begin{array}{r} 1 \leftarrow \text{SUB} \\ A_7A_6A_5A_4 \quad A_3A_2A_1A_0 \\ +\ \overline{B}_7\overline{B}_6\overline{B}_5\overline{B}_4 \quad \overline{B}_3\overline{B}_2\overline{B}_1\overline{B}_0 \\ \hline S_7S_6S_5S_4 \quad S_3S_2S_1S_0 \end{array}$$

When $A_7 \cdots A_0 = 0$, the circuit produces the 2's complement of $B_7 \cdots B_0$ because 1 is being added to the 1's complement $B_7 \cdots B_0$. When $A_7 \cdots A_0$ does not equal zero, the effect is equivalent to adding $A_7 \cdots A_0$ and the 2's complement of $B_7 \cdots B_0$.

Here is an example. Suppose that the numbers are +82 and +17. Then $A_7 \cdots A_0 = 0101\ 0010$ and $B_7 \cdots B_0 = 0001\ 0001$. The controlled inverter produces the 1's complement of B, which is 1110 1110. Since SUB = 1 during a subtraction, the circuit performs the following addition:

$$\begin{array}{rr} & 1 \leftarrow \text{SUB} \\ & 0101\ \ 0010 \\ +\ & 1110\ \ 1110 \\ \hline 1\ & 0100\ \ 0001 \end{array}$$

For 8-bit arithmetic, the final carry is ignored as previously discussed; therefore, the answer is

$$S_7 \cdots S_0 = 0100\ \ 0001$$

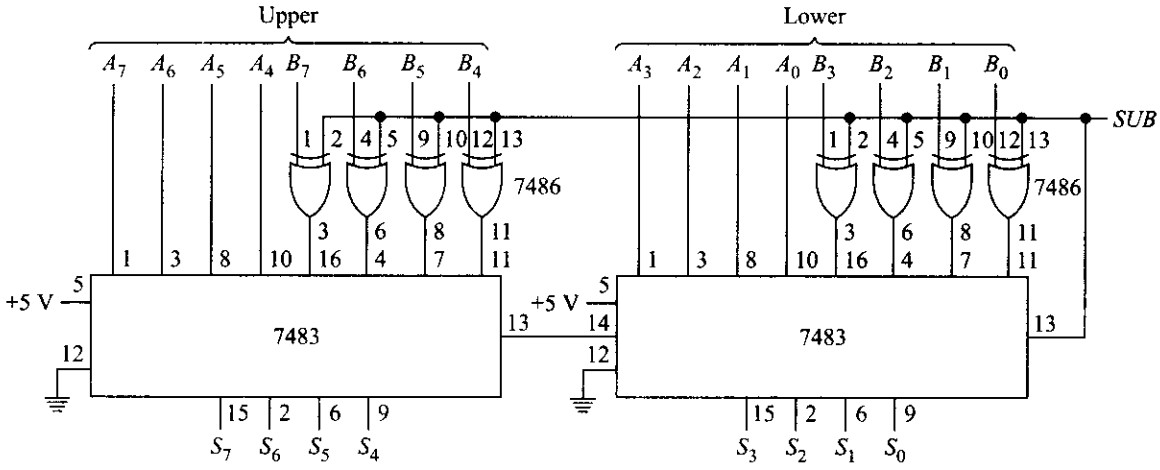


Fig. 6.7 **Two 7483s can add or subtract bytes**

This answer is equivalent to decimal +65, which is the algebraic difference between the numbers we started with: +82 and +17.

Example 6.11 Show how to build an 8-bit adder-subtracter with TTL circuits.

Solution The 7483 is a TTL circuit with four full-adders. This means that it can add nibbles. To add bytes, we need to use two 7483s as shown in Fig. 6.7. The CARRY OUT (pin 14) of the lower 7483 is used as the CARRY IN (pin 13) to the upper 7483. This allows the two 7483s to add 8-bit numbers. Two 7486s form the controlled inverter needed for subtraction.

The 74LS83, 74283, and 72LS283 are all TTL 4-bit adder ICs. They are pin-for-pin compatible, except that the '2.83 and 'LS283 have $+V_{CC}$ on pin 16 and GROUND on pin 8. The 74HC283 is the CMOS version of the same 4-bit adder.

The 74181, 74LS181, and 74LS381 are TTL ALUs, and the 74HC381 is the CMOS equivalent. Each is capable of adding two 4-bit binary numbers as well as performing numerous other logic operations.

6.9 FAST ADDER

Fast adder is also called *parallel adder* or *carry look ahead adder* because that is how it attains high speed in addition operation. Before we go into that circuit, let's see what limits the speed of an adder. Consider, the worst case scenario when two four bit numbers A: 1111 and B: 0001 are added. This generates a carry in the first stage that propagates to the last stage as shown next.

Carry:	111
A:	1111
B:	0001
	10000

Addition such as these (Fig. 6.6) is called *serial addition* or *ripple carry addition*. It also reveals from the adder equation (given in Section 6.8) result of every stage depends on the availability of carry from previous stage. The minimum delay required for carry generation in each stage is two gate delays, one coming from AND gates (1st level) and second from OR gate (2nd level). For 32-bit serial addition there will be 32 stages working in serial. In worst case, it will require $2 \times 32 = 64$ gate delays to generate the final carry. Though each gate delay is of nanosecond order, serial addition definitely limits the speed of high speed computing. Parallel adder increases the speed by generating the carry in advance (look ahead) and there is no need to wait for the result from previous stage. This is achieved by following method.

Let us use the second equation for carry generation from previous section, i.e.

$$C_i = A_iB_i + (A_i + B_i)C_{i-1}$$

This can be written as, $C_i = G_i + P_iC_{i-1}$

where, $G_i = A_iB_i$ and $P_i = A_i + B_i$

G_i stands for generation of carry and P_i stands for propagation of carry in a particular stage depending on input to that stage. As explained in previous section, if $A_iB_i = 1$, then ith stage will generate a carry, no matter previous stage generates it or not. And if $A_i + B_i = 1$ then this stage will propagate a carry if available from previous stage to next stage. Note that, all G_i and P_i are available after one gate delay once the numbers A and B are placed.

Starting from LSB, designated by suffix 0 if we proceed iteratively we get,

$C_0 = G_0 + P_0.C_{-1}$ [C_{-1} will normally be 0 if we are not using it as subtractor or cascading it.]

$C_1 = G_1 + P_1.C_0 = G_1 + P_1.(G_0 + P_0.C_{-1}) = G_1 + P_1.G_0 + P_1P_0.C_{-1}$ [Substituting C_0]

Similarly,

$C_2 = G_2 + P_2.C_1 = G_2 + P_2.(G_1 + P_1.G_0 + P_1P_0.C_{-1})$ [Substituting C_1]

$= G_2 + P_2.G_1 + P_2P_1.G_0 + P_2P_1P_0.C_{-1}$

$C_3 = G_3 + P_3.C_2 = G_3 + P_3(G_2 + P_2.G_1 + P_2P_1.G_0 + P_2P_1P_0.C_{-1})$ [Substituting C_2]

$= G_3 + P_3G_2 + P_3P_2.G_1 + P_3P_2P_1.G_0 + P_3P_2P_1P_0.C_{-1}$

etc.

The equations look pretty complicated. But do we gain in any way? Note that, these equations can be realized in hardware using multi-input AND and OR gates and in two levels. Now, for each carry whether C_0 or C_3 we require only two gate delays once the G_i and P_i are available. We have already seen they are available after 1 gate delay. Thus parallel adder (circuit diagram for 2-bit is shown in Fig. 6.8a) generates carry within

$1 + 2 = 3$ gate delays. Note that, after the carry is available at any stage there are two more gate delays from Ex-OR gate to generate the sum bit as we can write $S_i = G_i \oplus P_i \oplus C_{i-1}$.

Thus serial adder in worst case requires at least $(2n + 2)$ gate delays for n-bit addition and parallel adder requires only $3 + 2 = 5$ gate delays for that. One can imagine the gain for higher values of n. However, there is a caution. We cannot increase n indiscriminately for parallel adder as every logic gate has a capacity to accept at most a certain number of inputs, termed *fan-in*. This is a characteristic of the logic family to which the gate belongs. More about this is discussed in Chapter 14. The other disadvantage of parallel adder is increased hardware complexity for large n. In Fig. 6.8b we present functional diagram and pin connections of a popular fast adder, IC 74283.

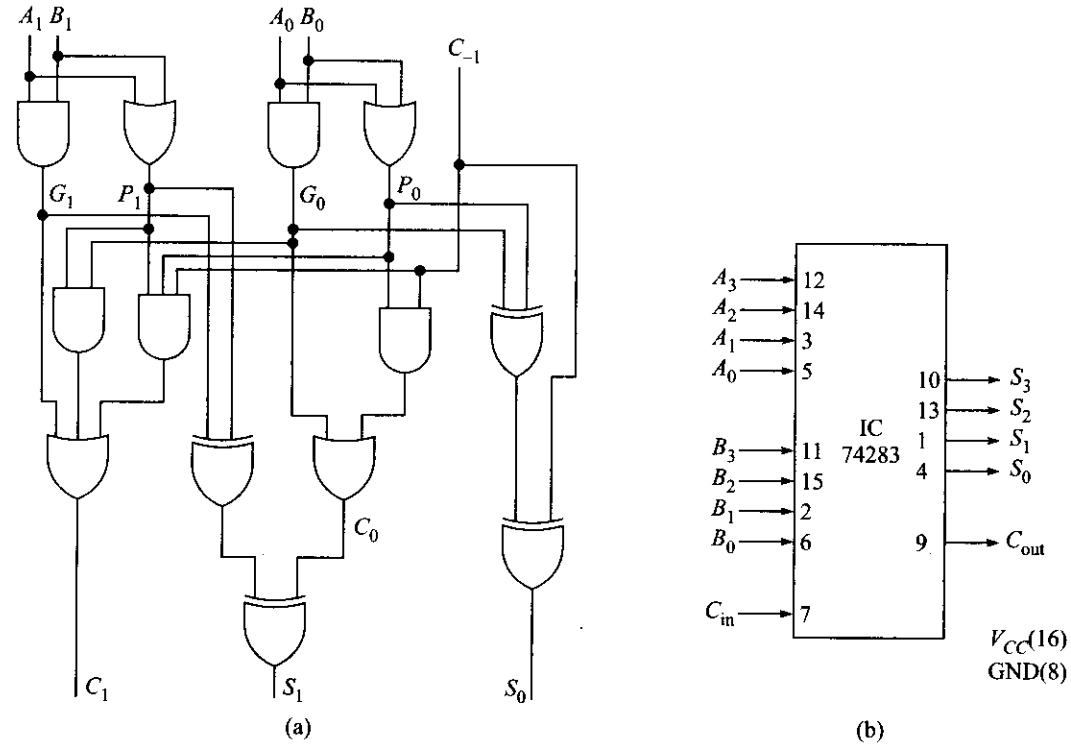


Fig. 6.8 **(a) Logic circuit for 2-bit fast adder, (b) Functional diagram of IC 74283**

Now, how do we add two 8-bit numbers using IC 74283? Obviously, we need two such devices and C_{out} of LSB adder will be fed as C_{in} of MSB unit. This way each individual 4-bit addition is done parallely but between two ICs carry propagates by rippling. To avoid carry ripple between two ICs and get truly parallel addition the following approach can be useful. Let each individual 4-bit adder unit generate two additional outputs Group Carry Generate (G_{3-0}) and Group Carry Propagate (P_{3-0}). They are defined as follows

$$G_{3-0} = G_3 + P_3G_2 + P_3P_2.G_1 + P_3P_2P_1.G_0$$
$$P_{3-0} = P_3P_2P_1P_0$$

so that

$$C_3 = G_{3\text{-}0} + P_{3\text{-}0}\, C_{-1} \qquad \text{[From equation of } C_3 \text{ in previous discussion]}$$

Now, let us see how this is useful in 8-bit parallel addition. For the 4-bit adder adding MSB taking C_3 as carry input, we can similarly write

$$C_7 = G_{7\text{-}4} + P_{7\text{-}4}\, C_3 \qquad [C_3 \text{ is equivalent to } C_{-1} \text{ input for this adder}]$$

where

$$G_{7\text{-}4} = G_7 + P_7G_6 + P_7P_6.G_5 + P_7\,P_6P_5.G_4$$
$$P_{7\text{-}4} = P_7P_6P_5P_4$$

Thus $C_7 = G_{7\text{-}4} + P_{7\text{-}4}(G_{3\text{-}0} + P_{3\text{-}0}\, C_{-1})$ [substituting C_3]

or $C_7 = G_{7\text{-}4} + P_{7\text{-}4}G_{3\text{-}0} + P_{7\text{-}4}P_{3\text{-}0}\, C_{-1}$

What do we get from above equation? Group carry generation and propagation terms are available from respective adder blocks ($G_{3\text{-}0}$, $P_{3\text{-}0}$ from LSB and $G_{7\text{-}4}$, $P_{7\text{-}4}$ from MSB) after 3 and 2 gate delays respectively. This comes from the logic equations that define them with G_i, P_i available after 1 gate delay.

Once these group-carry terms are available, we can generate C_7 from previous equation by designing a small Look Ahead Carry (LAC) Generator circuit. This requires a bank of AND gates (here one 2 input and one 3 input) followed by a multi-input OR gate (here, three input) totaling 2 gate delays. Thus final carry is available in 3 + 2 = 5 gate delays and this indeed is what we were looking for in parallel addition. In next section we discuss a versatile IC 74181 that while performing 4-bit addition generates this group carry generation and propagation terms. LAC generator circuits are also commercially available; IC 74182 can take up to four pairs of group carry terms from four adder units and generate final carry for 16 bit addition.

Before we go to next section can you answer after how many gate delays the sum bits ($S_{15}\ldots S_0$) of 16 bit fast adders will be available?

Example 6.12 Show how final carry is generated for a parallel adder when two numbers added are A: 1111 and B: 0001.

Solution First it calculates, G_i and P_i parallely.

$G_0 = 1.1 = 1$, $G_1 = 1.0 = 0$, $G_2 = 1.0 = 0$, $G_3 = 1.0 = 0$.

and $P_0 = 1 + 1 = 1$, $P_1 = 1 + 0 = 1$, $P_2 = 1 + 0 = 1$, $P_3 = 1 + 0 = 1$.

Note that $C_{-1} = 0$.

Then substituting these in equation of C_3 we get final carry as

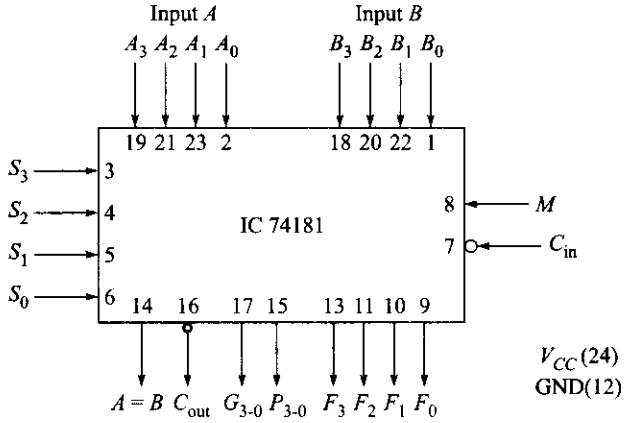
$$C_3 = G_3 + P_3G_2 + P_3P_2.G_1 + P_3\,P_2P_1.G_0 + P_3P_2P_1P_0.C_{-1}$$
$$= 0 + 1.0 + 1.1.0 + 1.1.1.1 + 1.1.1.1.0$$
$$= 0 + 0 + 0 + 1 + 0$$
$$= 1$$

SELF-TEST

16. What is the savings in time in a parallel adder?
17. What is the maximum number of inputs for an OR gate in a 4-bit parallel adder?

6.10 ARITHMETIC LOGIC UNIT

Arithmetic Logic Unit, popularly called ALU is multifunctional device that can perform both arithmetic and logic function. ALU is an integral part of central processing unit or CPU of a computer. It comes in various forms with wide range of functionality. Other than normal addition, subtraction it can also perform increment, decrement operations. As logic unit it performs usual AND, OR, NOT, EX-OR and many other complex logic functions. It also comes with PRESET and CLEAR options, invoking which all the function outputs are made 1 and 0 respectively. Normally, a mode selector input (M) decides whether ALU performs a logic operation or an arithmetic operation. In each mode different functions are chosen by appropriately activating a set of selection inputs.



(a)

S_3 S_2 S_1 S_0	$M = 1$ (Logic Function)	$M = 0$ (Arithmetic Function) $C_{in} = 1$ (For $C_{in} = 0$, add 1 to F)
0 0 0 0	$F = A'$	$F = A$
0 0 0 1	$F = (A+B)'$	$F = A + B$
0 0 1 0	$F = A'B$	$F = A + B'$
0 0 1 1	$F = 0$	$F =$ minus 1
0 1 0 0	$F = (AB)'$	$F = A$ plus (AB')
0 1 0 1	$F = B'$	$F = (A + B)$ plus (AB')
0 1 1 0	$F = A \oplus B$	$F = A$ minus B minus 1
0 1 1 1	$F = AB'$	$F = AB'$ minus 1
1 0 0 0	$F = A' + B$	$F = A$ plus (AB)
1 0 0 1	$F = (A \oplus B)'$	$F = A$ plus B
1 0 1 0	$F = B$	$F = (A + B')$ plus (AB)
1 0 1 1	$F = AB$	$F = AB$ minus 1
1 1 0 0	$F = 1$	$F = A$ plus A
1 1 0 1	$F = A + B'$	$F = (A + B)$ plus A
1 1 1 0	$F = A + B$	$F = (A + B')$ plus A
1 1 1 1	$F = A$	$F = A$ minus 1

(b)

Fig. 6.9 (a) Functional representation of ALU IC 74181, (b) Its truth table

In this section, we take up one very popular discrete ALU device from TTL family for discussion. IC 74181 is a 4-bit ALU that can generate 16 different kinds of outputs in each mode selected by four selection inputs S_3, S_2, S_1 and S_0. The functional diagram of this IC with pin numbers and corresponding truth table is shown in Fig. 6.9(a) and Fig. 6.9(b) respectively. Note that this truth table considers data inputs A and B are active high. A similar but different truth table is obtained if data is considered as active low.

Well, the truth table is pretty exhaustive though one might wonder what could be the utility of functions like $(A+B)$ plus AB'. But a careful observation shows one important function missing, that of a comparator. Is it truly so? No, it can be obtained in an indirect way. The C_{out} is activated (active low) by addition as well as subtraction because subtraction is carried out by 2's complement addition. Note that, if the result of an arithmetic operation is negative it will be available in 2's complement form. The $A = B$ output is activated when all the function outputs are 1, i.e. $F_3 \ldots F_0 = 1111$. Output $A = B$, together with C_{out} can give functions like $A > B$ and $A < B$. Note that $A = B$ is an open collector output; thus when more than 4-bits are to be compared this output of different ALU devices are wire-ANDed, simply by knotting outputs together to get the final result. To know more about open collector gates refer to Section 14.5 of Chapter 14.

The outputs C_{out}, $G_{3\text{-}0}$ and $P_{3\text{-}0}$ are useful when addition and subtraction of more than 4-bits are performed using more than one IC 74181 as discussed in previous sections.

Logic operations are done bit-wise by making $M = 1$ and choosing appropriate select inputs. Note that, carry is inhibited for $M = 1$. Let us see how AND operation between two 4-bit numbers 1101 and 0111 is to be performed. Enter input $A_3..A_0 = 1101$ and $B_3..B_0 = 0111$. Make $S_3..S_0 = 1011$ and of course $M = 1$ to choose logic function. The output is shown as $F_3..F_0 = 0011$.

For arithmetic operations $M = 0$ to be chosen and we have to appropriately place C_{in} (active low), if any. For example, if we want to add decimal numbers 6 with 4 we have to place 0110 for 6 at A and 0100 for 4 at B. Then with $S_3..S_0 = 1001$ (from truth table) and $C_{in} = 1$ (active low) the output generated is $F_3..F_0 = 1010$ which is decimal equivalent of 10.

Example 6.13

(a) Show how $A > B$ output can be generated in IC 74181 ALU. (b) Also show how $A \geq B$ condition can be checked.

Solution

(a) If $A > B$, then function A minus B will be positive and the result will not be in 2's complement form and more importantly will generate a carry. Refer to discussion and examples in Section 6.6 on 2's complement arithmetic. The final result for such subtraction is obtained by disregarding the carry. But here by checking output carry whether active we can conclude if $A > B$.
Thus to check $A > B$ put $M = 0$ (arithmetic operation), $S_3..S_0 = 0110$ (gives A minus B), $C_{in} = 0$ ($C_{in} = 1$ gives A minus B minus 1) and check carry is generated, i.e. $C_{out} = 0$ (active), which gives $A > B$.

(b) A similar reasoning shows by making $C_{in} = 1$ in above and checking if $C_{out} = 0$ we can verify $A \geq B$ condition.

Example 6.14 Show how bits of input A shifted to left by one unit appear at output F in IC 74181.

Solution We know shifting to left by one unit is equivalent to multiplication by two. Again multiplication by two can be achieved by adding the number with itself once. Thus make data input at A, $M = 0$ (arithmetic operation), $C_{in} = 1$ and $S_3..S_0 = 1100$ (gives A plus A) and we have A shifted by 1 unit to left at function output F.

SELF-TEST

18. What is an ALU?
19. How do you CLEAR all outputs of IC 74181?

6.11 BINARY MULTIPLICATION AND DIVISION

Typical 8-bit microprocessors like the 6502 and the 8085 use software multiplication and division. In other words, multiplication is done with addition instructions and division with subtraction instructions. Therefore, an adder-subtracter is all that is needed for addition, subtraction, multiplication, and division.

For example, multiplication is equivalent to repeated addition. Given a problem such as

$$8 \times 4 = ?$$

the first number is called the *multiplicand* and the second number, the *multiplier*. Multiplying 8 by 4 is the same as adding 8 four times:

$$8 + 8 + 8 + 8 = ?$$

One way to multiply 8 by 4 is to program a computer to add 8 until a total of four 8s have been added. This approach is known as programmed multiplication by repeated addition.

There are other software solutions to multiplication and division that you will learn about if you study assembly-language programming.

There are ICs available that will multiply two binary numbers. For instance, the 74284 and the 74285 will produce an 8-bit binary number that is the product of two 4-bit binary numbers. These ICs are very fast, and the total multiplication time is only about 40 nanoseconds (ns)!

SELF-TEST

20. Explain how one can do division of binary numbers.

6.12 ARITHMETIC CIRCUITS USING HDL

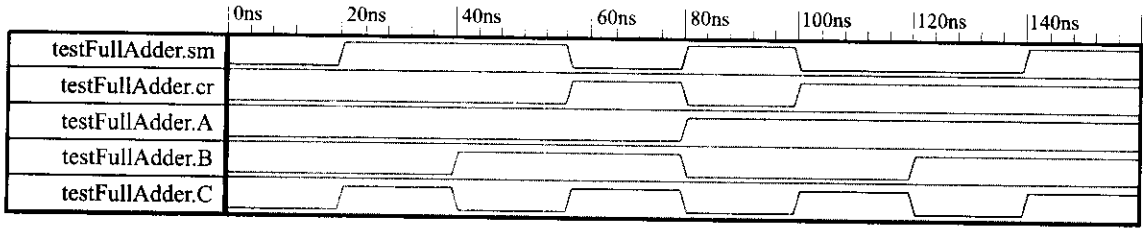
We first describe a full adder circuit and create a test bench to test it. Please refer to discussion of Section 6.7. If A and B are the binary digits to be added and C is the Carry input then output Sum and Carry (represented by sm and cr in following Verilog code) is expressed by equations

$$\text{Sum: sm} = AB + BC + CA \quad \text{and} \quad \text{Carry: cr} = A \oplus B \oplus C$$

We have used a test bench that generates all possible combinations of A, B and C by arithmetic addition and takes less space than test bench described in Chapter 2. The output sum (sm) and carry (cr) for this is shown in simulation waveform. One can see this verifies truth table of a full adder.

```
module testFullAdder;
reg A,B,C;
wire sm, cr;
fulladder fa1(A,B,C,sm,cr);// Circuit instantiated with fa1
initial // simulation begins
  begin
    {A,B,C} = 3'b000; //Initialization A=0,B=0,C=0
         repeat (7) //repeats following statement seven times
    #20 {A,B,C}={A,B,C} + 3'b001; //delay of 20 ns and then increment by 1
    #20 $finish; // simulation ends after generating 8 combinations of ABC
  end              //total time of simulation 7x20+20=160 ns
endmodule

module fulladder(A,B,C,sm,cr); // Description of fulladder Circuit
input A,B,C;
output sm,cr;
assign sm = (A&B)|(B&C)|(C&A);
assign cr = A^B^C;
endmodule
```



Example 6.15 Show Verilog design of 4-bit ripple carry adder.

Solution The code is given as follows. The one in the left hand side ensures ripple carry addition while the one in the right depends on the compiler. Based on considerations like speed, cost and other constraints Verilog compiler implements a 4-bit ripple carry adder in different manner.

```
module adder4bit (sum,cout,a,b,cin); //4-bit ripple carry adder
input [3:0] a,b; //Two 4 bit data to be added
input cin; //Input carry
output [3:0] sum; //4-bit sum output to be generated
output cout; //output carry
wire [2:0] cint; /*internal carry
generated in first three fulladder
fulladder*/
fa0(a[0],b[0],cin,sum[0],cint[0]);
// instantiates fulladder
```

```
/* 4-bit adder, compiler decides
if ripple carry */

module adder4bit
(sum,cout,a,b,cin);
input [3:0] a,b;
```

```
fulladder
fa1(a[1],b[1],cint[0],sum[1],cint[1]);
fulladder
fa2(a[2],b[2],cint[1],sum[2],cint[2]);
fulladder
fa3(a[3],b[3],cint[2],sum[3],cout);
endmodule

module fulladder(A,B,C,sm,cr);
input A,B,C;
output sm,cr;
assign sm = (A&B)|(B&C)|(C&A);
assign cr = A^B^C;
endmodule
```

```
input cin;
output [3:0] sum;
output cout;
assign {cout,sum} = a + b + cin;
endmodule
```

PROBLEM SOLVING WITH MULTIPLE METHODS

Problem Show how a half-adder can be realized.

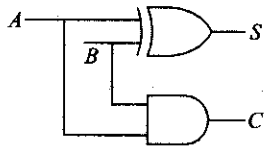
Solution The half-adder truth table and logic equations are reproduced from Section 6.7 in Fig. 6.10a.

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

(a)

$C = AB$

$S = A \oplus B$
$= AB' + A'B$

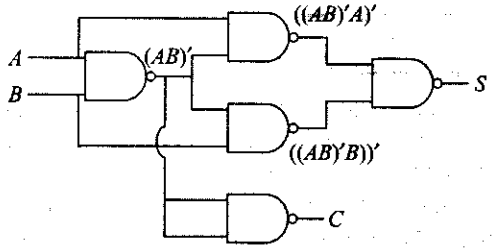


(b)

$C = A \cdot B = ((A \cdot B)')'$: double complement

$S = A \cdot B' + A'B$
$= (A \cdot B' + A'B)''$: double complement
$= ((A \cdot B')' \cdot (A'B)')'$: from Eq. 2.1
$= ((A'+B) \cdot (A+B')'$: from Eq. 2.2
$= (A'B'+AB)'$: since $X'X = 0$
$= (A'B')' \cdot (AB)'$: from Eq. 2.1
$= (A+B) \cdot (AB)'$: from Eq. 2.2
$= (AB)'A+(AB)'B$
$= ((AB)'A+(AB)'B)''$: double complement
$= (((AB)'A)' \cdot ((AB)'B))'$: from Eq. 2.1

(c)



(d)

Fig. 6.10 **(a) Truth table and logic equation for half-adder, (b) Realization using AND and exclusive-OR gate, (c) Derivation of AND and exclusive-OR relation, (d) Realization using only NAND gates**

In Method-1, the logic relations can directly be realized using AND and exclusive-OR gate as shown in Fig. 6.10b.

In Method-2, we show how it can be realized using only one type of basic gates, say NAND gate. The derivation is shown in Fig. 6.10c and the realization is shown in Fig. 6.10d.

It is left as an exercise to find how it can be realized using only NOR gates.

In Method-3, we show how it can be realized using two 4 to 1 multiplexers. We make use of the truth table to assign data inputs to the multiplexers while A and B are used as select inputs. The realization is shown in Fig. 6.11a.

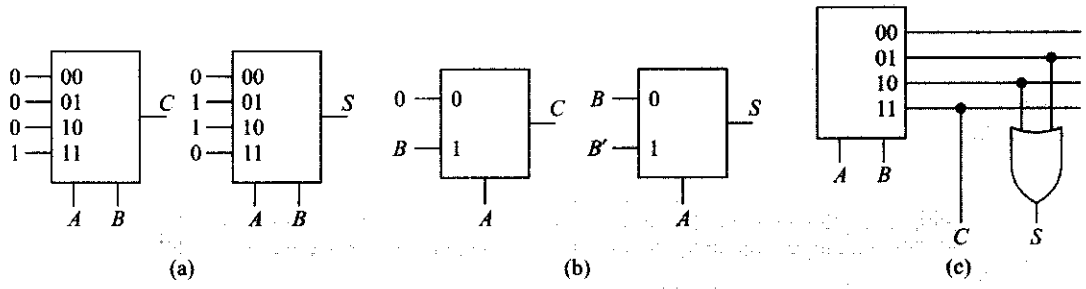


Fig. 6.11 **Realization using (a) 4 to 1 multiplexers, (b) 2 to 1 multiplexers, (c) Decoder, OR gate**

In Method-4, we show how it can be realized using two 2 to 1 multiplexers. Let the only selected input to the multiplexers be A.

We note from the equation, if $A = 0, C = 0$ and if $A = 1, C = B$

if $A = 0, C = B$ and if $A = 1, C = B'$

The realization from these is shown in Fig. 6.11b where B is used in the data input.

In Method-5, we show how it can be realized using a 2 to 4 decoder and OR gate. The decoder generates all the four minterms $A'B'$, $A'B$, AB' and AB. Carry output is generated directly from AB. Sum output is generated OR-ing $A'B$ and AB'. The realization is shown in Fig. 6.11c.

SUMMARY

Numbers represent physical quantities. As long as you know the number code being used, those strange-looking answers in other number systems make perfect sense. Subscripts can be used as a reminder of the base of the number system.

The unsigned 8-bit numbers are from 0000 0000 to 1111 1111, equivalent to decimal 0 to 255. The unsigned 16-bit numbers are from decimal 0 to 65,535. Overflows occur when a sum exceeds the range of the number system. With 8-bit arithmetic, an overflow occurs when the unsigned sum exceeds 255.

Sign-magnitude numbers use the MSB as a sign bit, with 0 for the + sign and 1 for the – sign. The rest of the bits are for the magnitude of the number. For this reason, 8-bit numbers cover the decimal range of –127 to +127, while 16-bit numbers cover –32,767 to +32,767.

The 2's complement representation is the most widespread code for positive and negative numbers. Positive numbers are coded as sign-magnitude numbers, and negative numbers are coded as 2's complements. The key feature of this number system is that taking the 2's complement of a number is equivalent to changing its sign. This characteristic allows us to subtract numbers by adding the 2's complement of the subtrahend. The advantage is simpler arithmetic hardware.

The half-adder has two inputs and two outputs; it adds 2 bits at a time. The full-adder has three inputs and two outputs; it adds 3 bits at a time. By connecting a controlled inverter and full-adders, we can build an adder-subtracter. This circuit can perform addition, subtraction, multiplication, and division.

A fast adder brings parallelism in addition process, more specifically by generating the carry using extra hardware through a look ahead logic. An Arithmetic Logic Unit is a versatile device, which can generate many useful arithmetic and logic functions with appropriate selection of inputs. Cascading of these devices is usually possible for working with larger sized numbers.

GLOSSARY

- ***arithmetic logic unit*** A device that can perform both arithmetic and logic function based on select inputs.
- ***full-adder*** A logic circuit with three inputs and two outputs. The circuit adds 3 bits at a time, giving a sum and a carry output.
- ***half-adder*** A logic circuit with two inputs and two outputs. It adds 2 bits at a time, producing a sum and a carry output.
- ***hardware*** The electronic, magnetic, and mechanical devices used in a computer or digital system.
- ***LSB*** Least-significant bit.
- ***look ahead carry*** Carry that need not ripple from one stage to other and is obtained through a look ahead logic after the binary numbers are placed in adder unit; useful in fast addition.
- ***magnitude*** The absolute or unsigned value of a number.
- ***microprocessor*** A digital IC that combines the arithmetic and control sections of a computer.
- ***MSB*** Most-significant bit.
- ***parallel addition*** A method of binary addition where carry generation at a particular stage does not depend on availability of carry from previous stage.
- ***overflow*** An unwanted carry that produces an answer outside the valid range of the numbers being represented.
- ***ripple carry*** Carry that ripples from one stage to other in serial addition.
- ***serial addition*** A method of binary addition where carry sequentially propagates from one stage to next stage.
- ***software*** A program or programs. The instructions that tell a computer how to process the data.
- ***2's complement*** The binary number that results when 1 is added to the 1's complement.

PROBLEMS

Section 6.1

6.1 Give the sum in each of the following:

a. $3_8 + 7_8 = ?$ b. $5_8 + 6_8 = ?$

c. $4_{16} + C_{16} = ?$ d. $8_{16} + F_{16} = ?$

6.2 Work out each of these binary sums:

a. 0000 1111 + 0011 0111

b. 0001 0100 + 0010 1001

c. 0001 1000 1111 0110 +
0000 1111 0000 1000

6.3 Show the binary addition of 750_{10} and 538_{10} using 16-bit numbers.

Section 6.2

6.4 Subtract the following: 0100 1111 −0000 0101.

6.5 Show this subtraction in binary form: $47_{10} - 23_{10}$.

Section 6.3

6.6 Indicate which of the following produces an overflow with 8-bit unsigned arithmetic:

a. $45_{10} + 78_{10}$ b. $34_8 + 56_8$

c. $CF_{16} + 67_{16}$

Section 6.4

6.7 Express each of the following in 8-bit sign-magnitude form:

a. +23 b. +123

c. −56 d. −107

6.8 Convert each of the following sign-magnitude numbers into decimal equivalents:

a. 00110110

b. 1010 1110

c. 1111 1000

d. 1000 1100 0111 0101

Section 6.5

6.9 Express the 1's complement of each of the following in hexadecimal notation:

a. 23H b. 45H

c. C9H d. FDH

6.10 What is the 2's complement of each of these:

a. 0000 1111

b. 0101 1010

c. 1011 1110

d. 1111 0000 1111 0000

6.11 Use Appendix 2 to convert each of the following to 2's complement representation:

a. +78 b. −23

c. −90 d. −121

6.12 Decode the following numbers into decimal values, using Appendix 2:

a. FCH b. 34H

c. 9AH d. B4H

Section 6.6

6.13 Show the 8-bit addition of these decimal numbers in 2's complement representation:

a. +45, +56 b. +89, −34

c. +67, −98

6.14 Show the 8-bit subtraction of these decimal numbers in 2's complement representation:

a. +54, +65 b. +68, −43

c. +16, −38 d. −28, −65

Section 6.7

6.15 Suppose that FD34H is the input to a 16-bit controlled inverter. What is the inverted output in hexadecimal notation? In binary?

Section 6.8

6.16 Expressed in hexadecimal notation, the two input numbers in Fig. 6.6 are 7FH and 4DH. What is the output when SUB is low?

6.17 The input numbers in Fig. 6.6 are 0001 0010 and 1011 1111. What is the output when SUB is high?

Section 6.9

6.18 Show how two IC 74283s can be connected to add two 8-bit numbers. Find the worst case delay.

6.19 Show how a parallel adder generates sum and carry bits while adding two numbers 1001 and 1011. What is the final result?

Section 6.10

6.20 How $A < B$ function is performed in IC 74181?

6.21 Show how 7 can be subtracted from 13 using IC 74181.

Section 6.11

6.22 Describe a program that multiplies 9×7 using repeated addition.

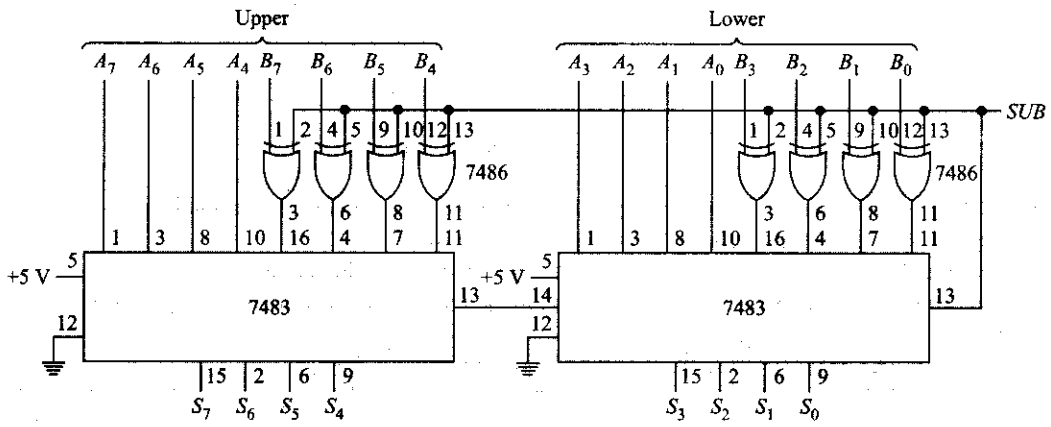
LABORATORY EXPERIMENT

AIM: The aim of this experiment is to perform addition and subtraction of two 8-bit data.

Theory: Two's complement arithmetic complements the number to be subtracted and adds one to it. Then this is added with the other number to perform subtraction. The addition is straight forward. IC 7483 is a 4-bit full adder with carry in input at pin 13. The exclusive-OR gate is useful to find complement of a binary number.

Apparatus: 5 VDC Power supply, Multimeter, and Bread Board

Work element: Verify the truth table of IC 7483. Connect two 7483 to perform 8-bit addition as shown. The exclusive-OR gate passes the same data to adder when *SUB* = 0 and a complement of the data when *SUB* = 1 where assertion of *SUB* stands for subtraction. Add and subtract five pair of numbers and compare with theoretical result.



Answers to Self-tests

1. See Eqs. (6.1) through (6.4).
2. This is the hexadecimal number 179F.
3. Binary 111; decimal 111
4. See Eqs. (6.5) through (6.8).
5. It is used to indicate an overflow.
6. 255
7. –127 to +127
8. –13; +13
9. 0010 1001
10. 0010 1010
11. 2's complement
12. Take the complement of the subtrahend and add it to the minuend.
13. Inputs: *A* and *B*; outputs: SUM and CARRY
14. Inputs: *A*, *B*, and CARRY IN; outputs: SUM and CARRY
15. T; see Fig. 6.4.
16. Parallel adder requires 5 gate delays and serial adder $(2n + 2)$ gate delays for n-bit addition.
17. Five.
18. ALU is short form of Arithmetic Logic Unit, a digital hardware that can perform both arithmetic and logic operations.
19. Substitute $M = 1$ and $S_3..S_0 = 0011$.
20. It realized only by repeatedly substracting one number from the other.

Clocks and Timing Circuits

OBJECTIVES

- **State the purpose of a clock in a digital system and demonstrate an understanding of basic terms and concepts related to clock waveforms**
- **Discuss the operation of the Schmitt trigger and its applications**
- **Recognize the astable and the monostable 555 timer circuits and compare the behavior of the two circuits**
- **Describe the retriggerable and nonretriggerable monostables**

The heart of every digital system is the system clock. The system clock provides the heartbeat without which the system would cease to function. In this chapter we consider the characteristics of a digital clock signal as well as some typical clock circuits. *Schmitt triggers* are used to produce nearly ideal digital signals from otherwise noisy or degraded signals. *Propagation delay* is the time required for a signal to pass from the input of a circuit to its output. You will see how to utilize logic gate propagation delay time to construct a pulse-forming circuit. A *monostable* is a basic digital timing circuit that is used in a wide variety of timing applications. We consider a number of different commercially available monostable circuits and examine some common applications.

7.1 CLOCK WAVEFORMS

Up to this point, we have been considering *static* digital logic levels, that is, voltage levels that do not change with time. However, all digital computer systems operate by "stepping through" a series of logical operations. The system signals are therefore changing with time: they are *dynamic*. The concept of a system *clock* was introduced in Chapter. 1. It is the clock signal that advances the system logic through its sequence of steps. The

square wave shown in Fig. 7.1a is a typical clock waveform used in a digital system. It should be noted that the clock need not be the perfectly symmetrical waveform shown. It could simply be a series of positive (or negative) pulses as shown in Fig. 7.1b. This waveform could of course be considered an asymmetrical square wave with a duty cycle other than 50 percent. The main requirement is that the clock be perfectly periodic, and stable.

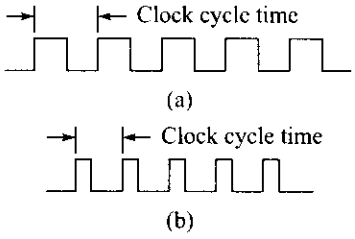


Fig. 7.1 **Ideal clock waveforms**

Notice that each signal in Fig. 7.1 defines a basic timing interval during which logic operations must be performed. This basic timing interval is defined as the *clock cycle time*, and it is equal to one period of the clock waveform. Thus all logic elements must complete their transitions in less than one clock cycle time.

Synchronous Operation

Nearly all of the circuits in a digital system (computer) change states in *synchronism* with the system clock. A change of state will either occur as the clock transitions from low to high or as it transitions from high to low. The low-to-high transition is frequently called the *positive transition* (*PT*), as shown in Fig. 7.2. The PT is given emphasis by drawing a small arrow on the *rising edge* of the clock waveform. A circuit that changes state at this time is said to be *positive-edge-triggered*. The high-to-low transition is called the *negative transition* (*NT*), as shown in Fig. 7.2. The NT is emphasized by drawing a small arrow on the *falling edge* of the clock waveform. A circuit that changes state at this time is said to be *negative-edge-triggered*. Virtually all circuits in a digital system are either positive-edge-triggered or negative-edge-triggered, and thus are synchronized with the system clock. There are a few exceptions. For instance, the operation of a push button (RESET) by a human operator might result in an instant change of state that is not in synchronism with the clock. This is called an *asynchronous operation*.

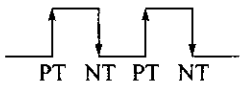


Fig. 7.2

Example 7.1 What is the clock cycle time for a system that uses a 500-kHz clock? An 8-MHz clock?

Solution The clock cycle is simply one period of the clock. For the 500-kHz clock,

$$\text{Cycle time} = \frac{1}{500 \times 10^3} = 2\ \mu\text{s}$$

For the 8-MHz clock,

$$\text{Cycle time} = \frac{1}{8 \times 10^6} = 125\ \text{ns}$$

Characteristics

The clock waveform drawn above the time line in Fig. 7.3a is a perfect, ideal clock. What exactly are the characteristics that make up an ideal clock? First, the clock levels must be absolutely stable. When the clock is high, the level must hold a steady value of +5 V, as shown between points *a* and *b* on the time line. When the clock is low, the level must be an unchanging 0 V, as it is between points *b* and *c*. In actual practice, the stability of the clock is much more important than the absolute value of the voltage level. For instance, it might be perfectly acceptable to have a high level of +4.8 V instead of +5.0 V, provided it is a *steady*, *unchanging*, +4.8 V.

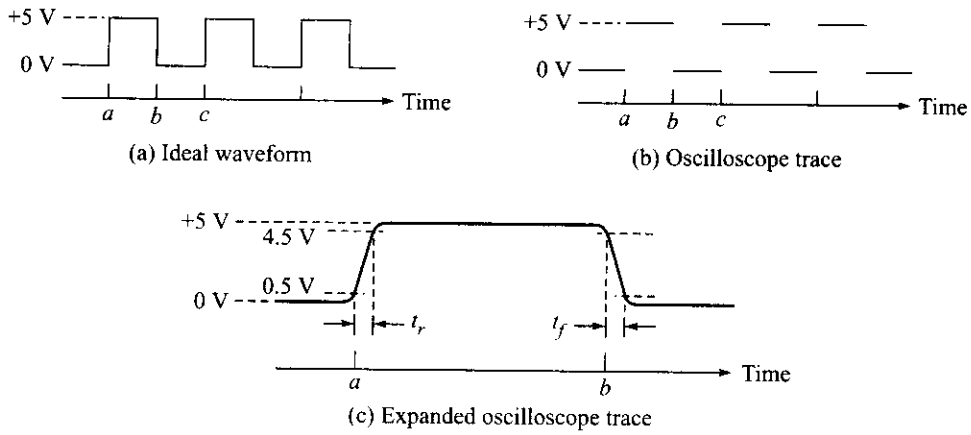


Fig. 7.3 Clock waveforms

The second characteristic deals with the time required for the clock levels to change from high to low or vice versa. The transition of the clock from low to high at point *a* in Fig. 7.3a is shown by a vertical line segment. This implies a time of zero; that is, the transition occurs instantaneously—it requires zero time. The same is true of the transition time from high to low at point *b* in Fig. 7.3a. Thus an ideal clock has zero transition time.

A nearly perfect clock waveform might appear on an oscilloscope trace as shown in Fig. 7.3b. At first glance this would seem to be two horizontal traces composed of line segments. On closer examination, however, it can be seen that the waveform is exactly like the ideal waveform in Fig. 7.3a if the vertical segments are removed. The vertical segments might not appear on the oscilloscope trace because the transition times are so small (nearly zero) and the oscilloscope is not capable of responding quickly enough. The vertical segments can usually be made visible by either increasing the oscilloscope "intensity," or by reducing the "sweep time."

Figure 7.3c shows a portion of the waveform in Fig. 7.3b expanded by reducing the "sweep time" such that the transition times are visible. Clearly it requires some time for the waveform to transition from low to high—this is defined as the rise time t_r. Remember, the time required for transition from high to low is defined as the fall time t_f. It is customary to measure the rise and fall times from points on the waveform referred to as the *10* and *90 percent* points. In this case, a 100 percent level change is 5.0 V, so 10 percent of this is 0.5 V and 90 percent is 4.5 V. Thus the rise time is that time required for the waveform to travel from 0.5 up to 4.5 V. Similarly, the fall time is that time required for the waveform to transition from 4.5 down to 0.5 V.

Finally, the third requirement that defines an ideal clock is its frequency stability. The frequency of the clock should be steady and unchanging over a specified period of time. Short-term stability can be specified by requiring that the clock frequency (or its period) not be allowed to vary by more than a given percentage over a short period of time—say, a few hours. Clock signals with short-term stability can be derived from straightforward electronic circuits as shown in the following sections.

Long-term stability deals with longer periods of time—perhaps days, months, or years. Clock signals that have long-term stability are generally derived from rather special circuits placed in a heated enclosure (usually called an "oven") in order to guarantee close control of temperature and hence frequency. Such circuits can provide clock frequencies having stabilities better than a few parts in 10^9 per day.

Propagation Delay Time

Propagation delay t_p is the time between a PT (or an NT) at the input of a digital circuit and the resulting change at the output. For all practical purposes, the time difference between fifty percent level of the input and corresponding output waveforms is used to calculate propagation delay. The box in Fig. 7.4 on the next page represents any TTL logic gate in the 74LSXX family. Notice that the waveform at the output is delayed in time from the input waveform, t_{pLH} is the delay time when the output is transitioning from low to high. t_{pHL} is the delay time when the output is transitioning from high to low. At temperatures below 75°C, t_{pHL} is only slightly larger than t_{pLH}. For the 74LSXX devices, we will simply assume they are equal, and for simplicity let's define propagation delay as

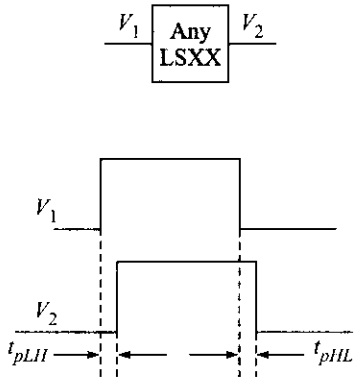


Fig. 7.4

$$\text{Propagation delay} \equiv t_p \approx t_{pLH} \approx t_{pHL}$$

The Texas Instruments data book gives a typical value of $t_p \approx 9$ ns for 74LSXX devices. For comparison, the high-speed CMOS has slightly longer delay times. For example, the 74HC04 inverter has $t_p = 24$ ns, which is typical.

Example 7.2 The total propogation delay through a 74HC04 inverter is known to be 24 ns. What is the maximum clock frequency that can be used with this device?

Solution An alternative way of posing the question is: How fast can the inverter operate? Remember, the circuit must complete any change of state within one clock cycle time. So,

$$\text{Clock cycle time} \geq t_p$$

The maximum clock frequency is then

$$\text{Frequency} = \frac{1}{t_p} = \frac{1}{24 \times 10^{-9}} = 41.7 \text{ MHz}$$

Pulse-Forming Circuits

It is sometimes necessary to use a series of narrow pulses in place of the rectangular clock waveform. Two such waveforms are shown in Fig. 7.5. The positive pulses occurring at the leading edge of the clock will define the PTs, while the negative pulses occurring at the falling edge will define the NTs. By taking advantage of the propagation delay time through a gate, it becomes a simple matter to change the rectangular clock into a series of pulses. There are numerous circuits that will change the clock into a pulse train, and here are two possibilities.

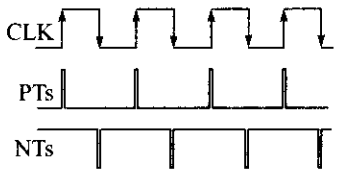


Fig. 7.5

In Fig. 7.6a, the clock (CLK) is applied to a NAND gate and an AND gate at the same time. The output of the NAND gate (A) is delayed by t_p. The output of the AND gate (PT) is high only when both its inputs are high. This is shown as the shaded region on the waveforms in Fig. 7.6b. The output (PT) is also delayed by t_p through the AND gate, and it appears as a positive pulse. Each output pulse (PT) is delayed by t_p from the leading edge of CLK, and each pulse has a width equal to t_p. Any digital circuit that incorporates the pulse-forming block in Fig 7.6a is said to be positive-edge-triggered, since it will change states in synchronism

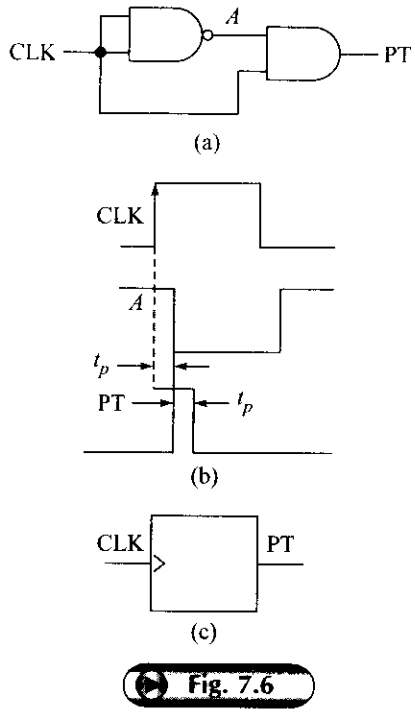


Fig. 7.6

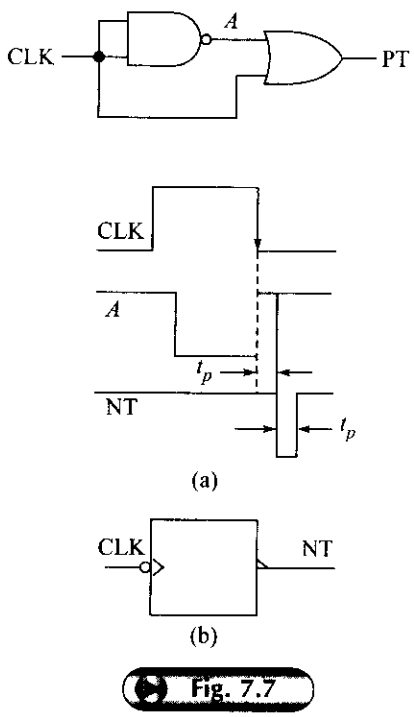


Fig. 7.7

with the PT of the clock. The box in Fig. 7.6c is a general symbol for a positive-edge-triggered circuit. The small triangle inside the box is called a *dynamic input indicator*, which simply means the circuit is sensitive to PTs.

In Fig. 7.7a, CLK is applied to a NAND gate and an OR gate simultaneously. The output of the NAND-gate (A) is delayed by t_p. The output of the OR gate (NT) is low only when both its inputs are low. This is the shaded region on the waveforms. The output (NT) is also delayed by t_p through the OR gate, and it appears as a negative pulse. Each output pulse has a width of t_p and each is delayed by t_p from the falling edge of CLK. Any digital circuit that incorporates this pulse-forming circuit is said to be negative-edge-triggered since it will change states in synchronism with the NT of the clock. The box in Fig. 7.7b is a general symbol for a negative-edge-triggered circuit. The small triangle is the dynamic input indicator, and the bubble shows that the input is active-low. The small triangle on the output indicates that NT is normally high, and is active when low. This triangle has the exact same meaning as the bubble. In fact, the IEEE standard uses these symbols interchangeably. You will see both symbols used in industry and on manufacturers' data sheets. Just remember, they both mean the same thing—active low!

It should be obvious that an inverter at the output of the AND gate in Fig. 7.6a will produce a series of negative pulses that synchronize with the leading edge of CLK. Similarly, an inverter at the output of the OR gate in Fig. 7.7a will produce a series of positive pulses in synchronization with the falling edge of CLK. These circuits, or variations of them, are used extensively with edge-triggered flip-flops—the subject of the next chapter. If you care to look ahead at the flip-flop symbols, you will see the dynamic indicator and the bubbled dynamic indicator used extensively.

SELF-TEST

1. Explain the meaning of positive-edge-triggered and negative-edge-triggered.
2. What is a dynamic input indicator?
3. What is the logic symbol for an input sensitive to NTs?

7.2 TTL CLOCK

A 7404 hexadecimal inverter can be used to construct an excellent TTL-compatible clock, as shown in Fig. 7.8. This clock circuit is well known and widely used. Two inverters are used to construct a two-stage amplifier with an overall phase shift of 360° between pins 1 and 6. Then a portion of the signal at pin 6 is fed back by means of a crystal to pin 1, and the circuit oscillates at a frequency determined by the crystal. Since the feedback element is a crystal, the frequency of oscillation is very stable. Here's how the oscillator works.

Inverter 1 has a 330-Ω feedback resistor (R_1) connected from output (pin 2) to input (pin 1). This forms a current-to-voltage amplifier with a gain of $A_1 = V_o / I_i = -R_1$. In this case, the gain is $A_1 = -330$ V/A, where the negative sign shows 180° of phase shift. For instance, an increase of 1 mA in I_i, will cause a negative-going voltage of 1 mA × 330 = 330 mV at V_o.

Inverter 2 is connected exactly as is inverter 1. Its gain is $A_2 = -R_2$. The two amplifiers are then ac-coupled with 0.01-μF capacitor to form an amplifier that has an overall gain of $A = A_1 \times A_2 = R_1R_2$. Notice that the overall gain has a positive sign, which shows 360° of phase shift. In this case, $A = 330 \times 330 = 1.09 \times 10^5$ V/A. For instance, an increase of 45 μA at I_i will result in a positive-going voltage of 5.0 V at pin 6 of inverter 2. Now, if a portion of the signal at pin 6 is fed back to pin 1, it will augment I_i (positive feedback) and the circuit will oscillate.

A series-mode crystal is used as the feedback element to return a portion of the signal at pin 6 to pin 1. The crystal acts as a series RLC circuit, and at resonance it ideally appears as a low-resistance element with no phase shift. The feedback signal must therefore be at resonance, and the two inverters in conjunction with the crystal form an oscillator operating at the crystal resonant frequency.

With the feedback element connected, the overall gain is sufficient to drive each inverter between saturation and cutoff, and the output signal is a periodic waveform as shown in Fig. 7.8. Typically, the output clock signal will transition between 0 and +5 V, will have rise and fall times of less than 10 ns, and will be essentially a square wave. The frequency of this clock signal determined by the crystal, and values between 1 and 20 MHz are common.

Inverter 3 is used as an output buffer amplifier and is capable of driving a load of 330 Ω in parallel with 100 pF while still providing rise and fall times of less than 10 ns.

Example 7.3 A TTL clock circuit as shown in Fig. 7.8 is said to provide a 5-MHz clock frequency with a stability better than 5 parts per million (ppm) over a 24-h time period. What are the frequency limits of the clock?

Solution A stability of 5 parts per million means that a 1-MHz clock will have a frequency of 1,000,000 plus or minus 5 Hz. So, this clock will have a frequency of 5,000,000 plus or minus 25 Hz. Over any 24-h period the clock frequency will be somewhere between 4,999,975 and 5,000,025 Hz:

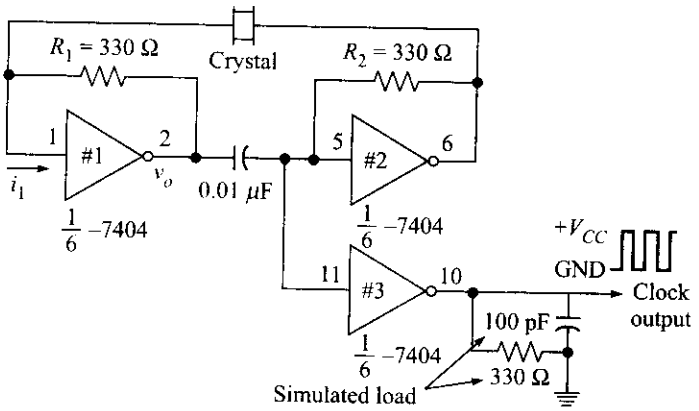


Fig. 7.8 TTL clock circuit

SELF-TEST

4. Why must the crystal in Fig. 7.8 be a series mode and not a parallel mode?
5. Are the 100-pF and 330-Ω loads necessary in Fig. 7.8?

7.3 SCHMITT TRIGGER

A *Schmitt trigger* is an electronic circuit that is used to detect whether a voltage has crossed over a given reference level. It has two stable states and is very useful as a signal-conditioning device. Given a sinusoidal waveform, a triangular wave, or any other periodic waveform, the Schmitt trigger will produce a rectangular output that has sharp leading and trailing edges. Such fast rise and fall times are desirable for all digital circuits.

Figure 7.9 shows the transfer function (V_o versus V_i) for any Schmitt trigger. The value of V_i that causes the output to jump from low to high is called the *positive-going threshold voltage* V_{T+}. The value of V_i causing the output to switch from high to low is called the *negative-going threshold voltage* V_{T-}.

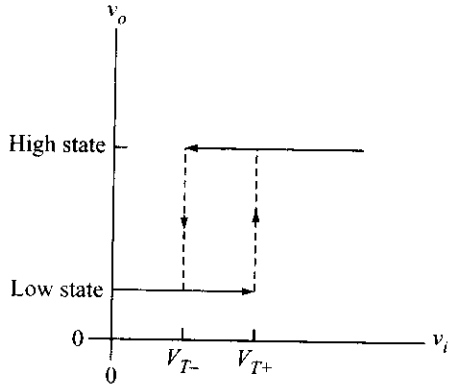


Fig. 7.9 Schmitt-trigger transfer characteristic

The output voltage is either high or low. When the output is low, it is necessary to raise the input to slightly more than V_{T+} to produce switching action. The output will then switch to the high state and remain there until the input is reduced to slightly below V_{T-}. The output will then switch back to the low state. The arrows and the dashed lines show the switching action.

The difference between the two threshold voltages is known as *hysteresis*. It is possible to eliminate hysteresis by circuit design, but a small amount of hysteresis is desirable because it ensures rapid switching action over a wide temperature range. Hysteresis can also be a very beneficial feature. For instance, it can be used to provide noise immunity in certain applications (digital modems for example).

The TTL 7414 is a hex Schmitt-trigger inverter. The *hex* means there are six Schmitt-trigger circuits in one DIP. In Fig. 7.10a, the standard logic symbol for one of the Schmitt-trigger inverters in a 7414 is shown along with a typical transfer characteristic. Because of the inversion, the characteristic curve is reversed from that shown in Fig. 7.9. Looking at the curve in Fig. 7.10b, when the input exceeds 1.7 V, the output will switch to the low state. When the input falls below 0.9 V, the output will switch back to the high state. The switching action is shown by the arrows and the dashed lines.

The TTL 74132 is a quad 2-input NAND gate that employs Schmitt-trigger with a similar hysteresis characteristics as described before for 7414. Figure 7.10c shows the standard logic symbol for one Schmitt-trigger NAND gate.

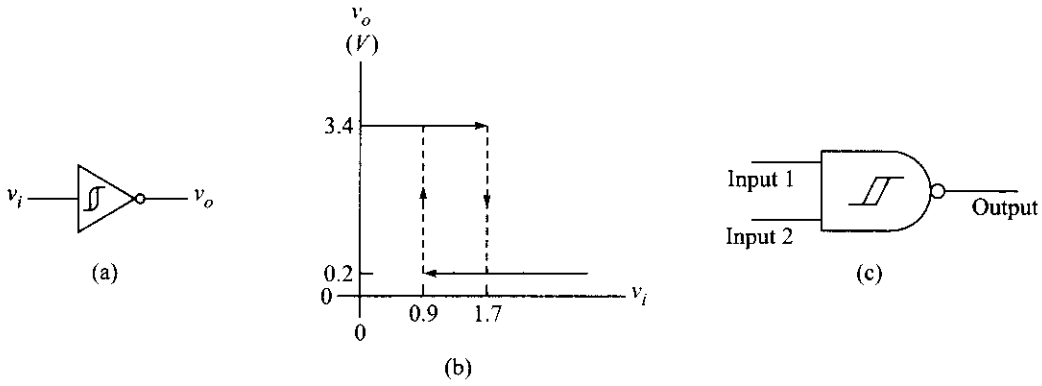


Fig. 7.10 **(a) Logic symbol of Schmitt-trigger inverter, (b) 7414 hysteresis characteristics, and (c) Logic symbol of Schmitt-trigger 2-input NAND gate**

Example 7.4 A sine wave with a peak of 2 V drives one of the inverters in a 7414. Sketch the output voltage.

Solution When the sinusoid exceeds 1.7 V, the output goes from high to low. The output stays in the low state until the input sinusoid drops below 0.9 V. Then the output jumps back to the high state. Figure 7.11 shows the input and output waveforms. This illustrates the signal-conditioning action of the Schmitt-trigger inverter. It has changed the sine wave into a rectangular pulse with fast rise and fall times. The same action would occur for any other periodic waveform.

Noisy Signals

The hysteresis characteristic of a Schmitt trigger is very useful in changing noisy signals, or signals with slow rise times, into more nearly ideal digital signals. A noisy signal is illustrated in Fig. 7.12a. Applying this signal to the input of a 7404 inverter will produce *multiple* pulses at its output, as shown in Fig. 7.12b. Each time the input signal crosses the threshold of the 7404, it will respond, and the multiple output transitions are the result. When used with an edge-triggered circuit, this will produce numerous unwanted PTs and NTs. The

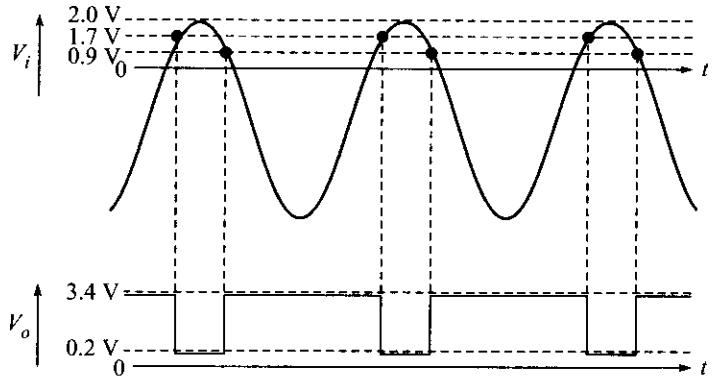
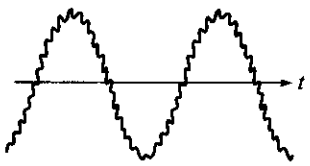
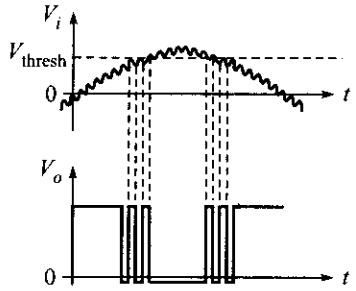


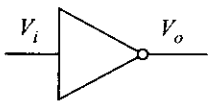
Fig. 7.11

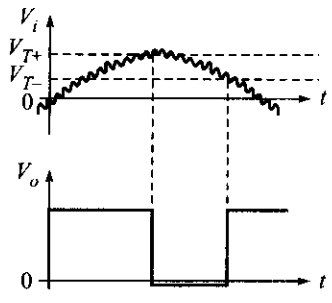


(a) A noisy signal



(b)





(c)

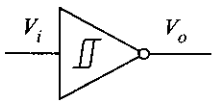


Fig. 7.12

Schmitt trigger will eliminate these multiple transitions, as shown in Fig. 7.12c. When the input rises above V_{T+}, the output will go low. However, the output will not again change state until the input falls below V_{T-}. Thus, multiple triggering is avoided! A Schmitt trigger is occasionally incorporated in an IC, for instance, the 74121, which is discussed in the next section.

SELF-TEST

6. What is the meaning of hysteresis when applied to a Schmitt trigger?
7. What is the difference between an inverting and a noninverting Schmitt trigger?
8. Schmitt triggers can be used as simple inverters. What is another good application for a Schmitt trigger?

7.4 555 TIMER—ASTABLE

The 555 timer is a TTL-compatible integrated circuit (IC) that can be used as an oscillator to provide a clock waveform. It is basically a switching circuit that has two distinct output levels. With the proper external components connected, neither of the output levels is stable. As a result, the circuit continuously switches back and forth between these two unstable states. In other words, the circuit oscillates and the output is a periodic, rectangular waveform. Since neither output state is stable, this circuit is said to be *astable* and is often referred to as a *free-running multivibrator* or *astable multivibrator*. The frequency of oscillation as well as the duty cycle are accurately controlled by two external resistors and a single timing capacitor. The internal circuit diagram of LM 555 timer is shown in Fig. 7.13(a). Note that the two comparators inside have two different reference voltages $V_{CC}/3$ and $2V_{CC}/3$ for comparisons, if $V_{CC}/3$ is the voltage between pin 1 and 8. Also note how they are connected to + and – input of the comparator. The Set Reset flip-flop sets or resets the output based on these comparator outputs in its usual operation. If required, it can be separately reset by asserting pin 4. More about this flip-flop will be discussed in next chapter. In this section, we show how 555 can be connected to get an astable multivibrator and in next section, we will discuss how it can be used in monostable mode.

The logic symbol for an LM555 timer connected as an oscillator is shown in Fig. 7.13. The timing capacitor C is charged toward $+V_{CC}$ through resistors R_A and R_B. The charging time t_1 is given as

$$t_1 = 0.693(R_A + R_B)C$$

This is the time during which the output is high as shown in Fig. 7.13.

The timing capacitor C is then discharged toward ground (GND) through the resistor R_B. The discharge time t_2 is given as

$$t_2 = 0.693R_BC$$

This is the time during which the output is low, as shown in Fig. 7.13.

The period T of the resulting clock waveform is the sum of t_1 and t_2. Thus

$$T = t_1 + t_2 = 0.693(R_A + 2R_B)C$$

The frequency of oscillation is then found as

$$f = \frac{1}{T} = \frac{1.44}{(R_A + 2R_B)C}$$

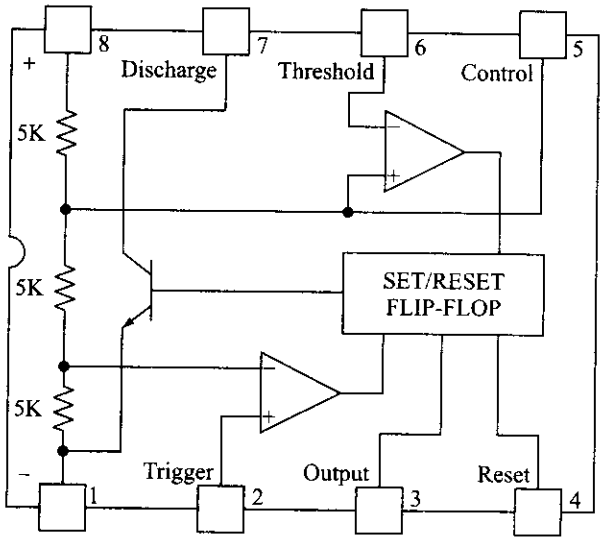
Example 7.5 Determine the frequency of oscillation for the 555 timer in Fig. 7.13, given $R_A = R_B = 1\ \text{k}\Omega$ and $C = 1000$ pF.

$$t_1 = 0.693\,(R_A + R_B)\,C$$

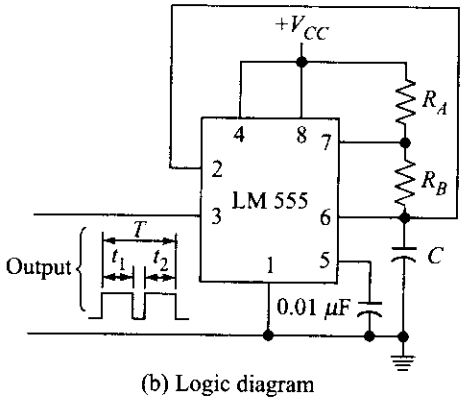
$$t_2 = 0.693\,R_B C$$

$$t = \frac{1.44}{(R_A + 2R_B)\,C}$$

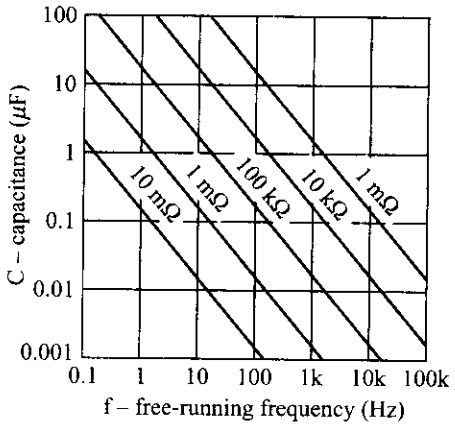
$$Duty\ cycle = \frac{t_2}{t_1 + t_2} = \frac{R_B}{R_A + 2R_B}$$



(a)



(b) Logic diagram



(c) Nomograph

Fig. 7.13 **(a) Internal diagram of LM 555, (b) LM 555 in astable mode, (c) Nomograph**

Solution Using the relationship given above, we obtain

$$f = \frac{1.44}{[1000 + 2(1000)] \times 10^{-9}} = 480\ \text{kHz}$$

The output of the 555 timer when connected this way is a periodic rectangular waveform but not a square wave. This is because t_1 and t_2 are unequal, and the waveform is said to be asymmetrical. A mea-

sure of the asymmetry of the waveform can be stated in terms of its duty cycle. Here we define the duty cycle to be the ratio of t_2 to the period.

Thus

$$\text{Duty cycle} = \frac{t_2}{t_1 + t_2}$$

As defined, the duty cycle is always a number between 0.0 and 1.0 but is often expressed as a percent. For instance, if the duty cycle is 0.45 (or 45 percent), the signal is at GND level 45 percent of the time and at high level 55 percent of the time.

Example 7.6

(a) Given $R_B = 750\ \Omega$, determine values for R_A and C in Fig. 7.13 to provide a 1.0-MHz clock that has a duty cycle of 25 percent.

(b) What change in the circuit shown in Fig. 7.13 gives duty cycle approximately 50%?

Solution

(a) A 1-MHz clock has a period of 1 μs. A duty cycle of 25 percent requires $t_1 = 0.75\ \mu$s and $t_2 = 0.25\ \mu$s. Solving the expression

$$\text{Duty cycle} = \frac{R_B}{R_A + 2R_B}$$

for R_A yields

$$R_A = \frac{R_B}{\text{Duty cycle}} = -2R_B = \frac{750}{0.25} - 2 \times 750 = 1500\ \Omega$$

Solving $t_2 = 0.693 R_B C$ for C yields

$$C = \frac{t_2}{0.693 R_B} = \frac{0.25 \times 10^{-6}}{0.693 \times 750} = 480\ \text{pF}$$

(b) Connect a diode across R_B pointing from pin 7 to 6 so that it conducts while charging capacitor C and make $R_A = R_B$. Then while charging, R_B is bypassed as diode is forward biased but discharging is through R_B as diode remains reverse biased and does not conduct. Thus we get same charging and discharging current. Neglecting small voltage drop across forward biased diode we approximately gate 50% duty cycle.

The nomogram given in Fig. 7.13b can be used to estimate the free-running frequency to be achieved with various combinations of external resistors and timing capacitors. For example, the intersection of the resistance line 10 k$\Omega = (R_A + 2R_B)$ and the capacitance line 1.0 μF gives a free-running frequency of just over 100 Hz. It should be noted that there are definite constraints on timing component values and the frequency of oscillation, and you should consult the 555 data sheets.

SELF-TEST

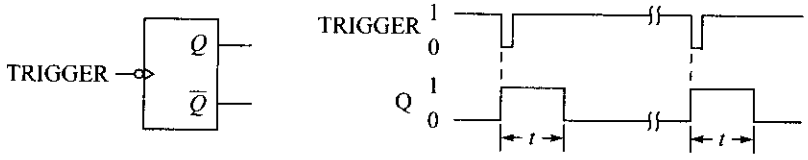
9. What is an astable circuit?
10. A 555 timer can be connected to form an oscillator. (T or F)
11. The oscillation frequency in astable 555 is (directly, inversely) proportional to the external timing capacitor.

7.5 555 TIMER—MONOSTABLE

With only minimal changes in wiring, the 555 timer discussed in Sec. 7.4 can be changed from a free-running oscillator (astable) into a switching circuit having one stable state and one quasistable state. The resulting *monostable* circuit is widely used in industry for many different timing applications. The normal mode of operation is to trigger the circuit into its quasistable state, where it will remain for a predetermined length of time. The circuit will then switch itself back (regenerate) into its stable state, where it will remain until it receives another input trigger pulse. Since it has only one stable state, the circuit is characterized by the term *monostable multivibrator*, or simply *monostable*.

The standard logic symbol for a monostable is shown in Fig. 7.14a. The input is labeled *TRIGGER*, and the output is Q. The complement of the Q output may also be available at $\overline{Q}$. The input trigger circuit may be sensitive to either a PT or an NT. In this case, it is negative-edge-triggered. Usually the output at Q is low when the circuit is in its stable state.

A typical set of waveforms showing the proper operation of a monostable circuit is shown in Fig. 7.14b. In this case, the circuit is sensitive to an NT at the trigger input, and the output is low when the circuit rests in its stable state. Once triggered, Q goes high and remains high for a predetermined time t and then switches back to its stable state until another NT appears at the trigger input.



(a) Logic symbol (b) Waveforms

Fig. 7.14 Monostable circuit

A 555 timer wired as a monostable switching circuit (sometimes called a *one-shot*) is shown in Fig. 7.15 on the next page. In its stable state, the timing capacitor C is completely discharged by means of an internal transistor connected to C at pin 7. In this mode, the output voltage at pin 3 is at ground potential.

A negative pulse at the trigger input (pin 2) will cause the circuit to switch to its quasistable state. The output at pin 3 will go high and the discharge transistor at pin 7 will turn off, thus allowing the timing capacitor to begin charging toward V_{CC}.

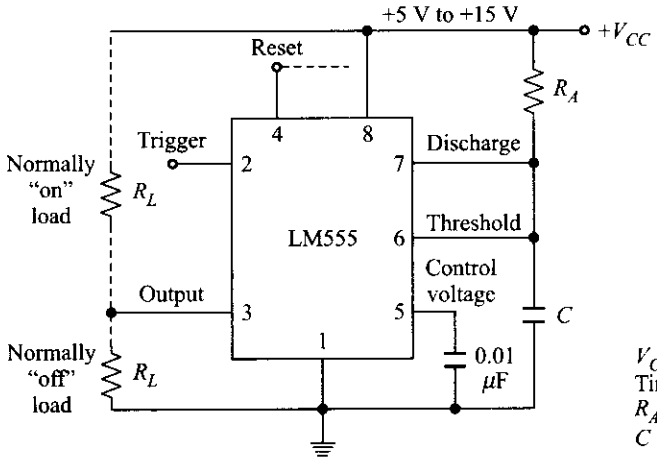
When the voltage across C reaches $^2/_3\ V_{CC}$, the circuit will regenerate back to its stable state. The discharge transistor will again turn on and discharge C to GND, the output will go back to GND, and the circuit will remain in this state until another pulse arrives at the trigger input. A typical set of waveforms is shown in Fig. 7.15b.

The output of the monostable can be considered a positive pulse with a width

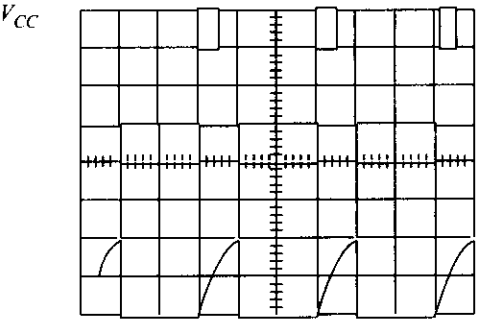
$$t = 1.1\ R_A C$$

Take care to note that the input voltage at the trigger input must be held at $+V_{CC}$, and that a negative pulse should then be applied when it is desired to trigger the circuit into its quasistable or timing mode.

Example 7.7 Find the output pulse width for the timer in Fig. 7.15 given $R_A = 10\ \text{k}\Omega$ and $C = 0.1\ \mu\text{F}$.



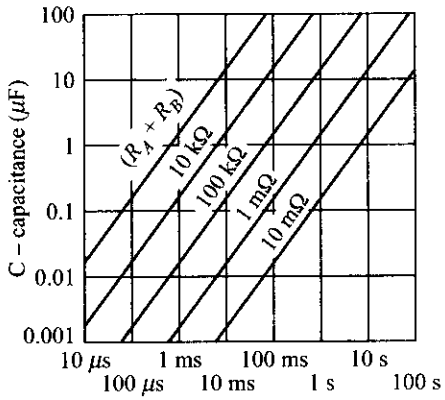
(a) Monostable



V_{CC} = 5 V
Time = 0.1 ms/DIV
R_A = 9.1 kΩ
C = 0.1 μF

Top trace: input 5 V/DIV
Middle trace: output 5 V/DIV
Bottom trace: capacitor voltage 2 V/DIV

(b) Monostable waveforms



(c) Time delay, $t = 1.1\ R_A C$

Fig. 7.15 **LM555 connected as a monostable circuit**

Solution The pulse width is found as

$$t = 1.1(R_A C) = 1.1(10^4 \times 10^{-7}) = 1.1 \text{ ms}$$

Example 7.8 Find the value of C necessary to change the pulse width in Example 7.7 to 10 ms.

Solution The timing equation can be solved for C as

$$C = \frac{t}{1.1R_A} = \frac{10^{-2}}{1.1 \times 10^4}$$

$$= 0.909\ \mu\text{F}$$

The nomograph shown in Fig. 7.15c can be used to obtain a quick, if not very accurate, idea of the sizes of R_A or C required for various pulse-width times. You can quickly check the validity of the results of Example 7.8 by following the $R_A = 10$ kΩ line up to the $C = 0.1$ μF line and noting that pulse-width time is about 1 ms.

Once the circuit is switched into its quasistable state (the output is high), the circuit is immune to any other signals at its trigger input. That is, the timing cannot be interrupted and the circuit is said to be *nonretriggerable*. However, the timing can be interrupted by the application of a negative signal at the reset input on pin 4. A voltage level going from $+V_{CC}$ to GND at the reset input will cause the timer to immediately switch back to its stable state with the output low. As a matter of practicality, if the reset function is not used, pin 4 should be tied to $+V_{CC}$ to prevent any possibility of false triggering.

SELF-TEST

12. **What is a monostable?**
13. **A 555 timer can be connected as a one-shot. (T or F)**
14. **Is the stable output state of a 555 timer connected in a monostable mode high or low?**

7.6 MONOSTABLES WITH INPUT LOGIC

The basic monostable circuit discussed in the previous section provides an output pulse of predetermined width in response to an input trigger. Logic gates have been added to the inputs of a number of commercially available monostable circuits to facilitate the use of these circuits as general-purpose delay elements. The 74121 *nonretriggerable* and the 74123 *retriggerable monostables* are two such widely used circuits.

The logic inputs on either of these circuits can be used to allow triggering of the device on either a high-to-low transition (NT) or on a low-to-high transition (PT). Whenever the value of the input logic equation changes from false to true, the circuit will trigger. Take care to note that a transition from false to true must occur, and simply holding the input logic equation in the true state will have no effect.

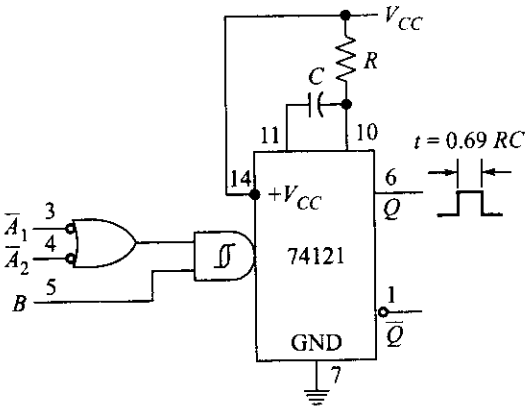
The logic diagram, truth table, and typical waveforms for a 74121 are given in Fig. 7.16. The inputs to the 74121 are $\overline{A}_1$, $\overline{A}_2$, and B. The trigger input to the monostable appears at the output of the AND gate. Here's how the gates work:

1. If B is held high, an NT at either $\overline{A}_1$ or $\overline{A}_2$ will trigger the circuit (see Fig. 7.16c). This corresponds to the bottom two entries in the truth table.
2. If either $\overline{A}_1$ or $\overline{A}_2$, or both are held low, a PT at B will trigger the circuit (see Fig. 7.16d).

This corresponds to the top two entries in the truth table. A logic equation for the trigger input can be written as

$$T = (A_1 + A_2)B\overline{Q}$$

Note that for T to be true (high), either A_1 or A_2 must be true—that is, either $\overline{A}_1$ or $\overline{A}_2$ at the gate input must be low. Also, since $\overline{Q}$ is low during the timing cycle (in the quasistable state), it is not possible for a transition to occur at T during this time. The logic equation for T must be low if $\overline{Q}$ is low. In other words, once the monostable has been triggered into its quasistable state, it must time out and switch back to its stable state before it can be triggered again. This circuit is thus nonretriggerable.



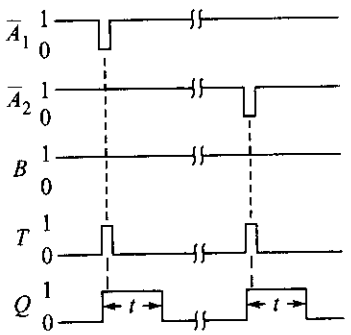
(a) Logic diagram

$\overline{A}_1$	$\overline{A}_2$	B	Result
L	X	↑	Trigger
X	L	↑	Trigger
↓	H	H	Trigger
H	↓	H	Trigger

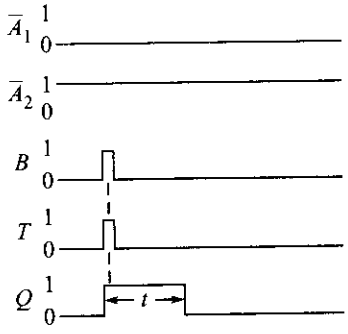
Note: Triggering can occur only when $\overline{Q}$ is H (not in timing cycle)

L = Low
H = High
X = Don't care
↑ = Low to high transition
↓ = High to low transition

(b) Truth table



(c) Negative triggering



(d) Positive triggering

Fig. 7.16 **74121 nonretriggerable monostable**

The output pulse width at Q is set according to the values of the timing resistor R and capacitor C as

$$t = 0.69RC$$

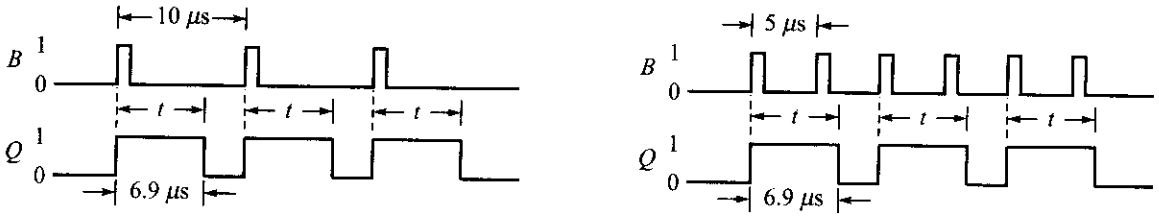
For instance, if $C = 1\ \mu F$ and $R = 10\ k\Omega$, the output pulse width will be $t = 0.69 \times 10^4 \times 10^{-6} = 6.9$ ms.

Example 7.9 The 74121 monostable in Fig. 7.16 is connected with $R = 1\ k\Omega$ and $C = 10{,}000$ pF. Pins 3 and 4 are tied to GND and a series of positive pulses are applied to pin 5. Describe the expected waveform at pin 6, assuming that the input pulses are spaced by (a) 10 μs and (b) 5 μs.

Solution The circuit is connected such that positive pulses applied to pin 5 will trigger it. The output pulse width at pin 6 will be $t = 0.69 \times 10^3 \times 10^{-3} = 6.9\ \mu s$.

(a) The monostable will trigger and time out for every input pulse appearing at B, as shown in Fig. 7.17a.
(b) Since the monostable is *not* retriggerable, it will trigger once and time out for every other input pulse as shown in Fig. 7.17b.

The logic diagram and truth table for a 74123 retriggerable monostable are given in Fig. 7.18. There are actually two circuits in each 16-pin DIP, and the pin numbers are given for one of them. The input logic is



(a) Triggers on every pulse at B

(b) Triggers on every other pulse at B

Fig. 7.17

simpler than for the 74121. The inputs are $\overline{A}$, B, and $\overline{R}$, and the truth table summarizes the operation of the circuit. The first entry in the truth table shows that the circuit will trigger if $\overline{R}$ and B are both high, and an NT occurs at $\overline{A}$.

The second truth table entry states the circuit will trigger if $\overline{A}$ is held low, $\overline{R}$ is held high, and a PT occurs at B.

In the third truth table entry, if $\overline{A}$ is low and B is high, a PT at $\overline{R}$ will trigger the circuit.

The last two truth table entries deal with direct reset of the circuit. Irrespective of the values of $\overline{A}$ or B, if the $\overline{R}$ input has an NT, or is held low, the circuit will immediately reset.

The logic equation for the trigger input to the monostable can be written $T = AB\overline{R}$. Notice that the state of the output Q does not appear in this equation (as it does for the 74121). This means that this circuit will trigger *every time* there is a PT at T. In other words, this is a *retriggerable* monostable!

The output pulse width at Q for the 74123 is set by the values of the timing resistor R and the capacitor C. It can be approximated by the equation

$$t = 0.33RC$$

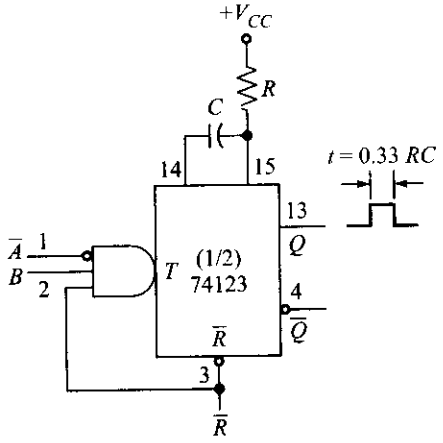
The waveforms in Fig. 7.19c show a series of negative pulses used to trigger the 74123. Notice carefully that the circuit triggers (Q goes high) at the first high-to-low transition on $\overline{A}$, but that the next two negative pulses on $\overline{A}$ retrigger the circuit and the timing cycle t does not begin until the very last trigger!

Example 7.10 The 74123 in Fig. 7.18 is connected with $\overline{A}$ at GND, $\overline{R}$ at $+V_{CC}$, R = 10 kΩ, and C = 10,000 pF. Describe the expected waveform at Q, assuming that a series of positive pulses are applied at B and the pulses are spaced at (a) 50 μs and (b) 10 μs.

Solution The output pulse width will be about

$$t = 0.33 \times 10^4 \times 10^{-8} = 33\ \mu s$$

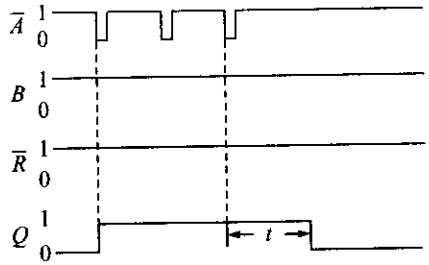
a. The circuit will trigger and time out with every pulse as shown in Fig. 7.19a.
b. The circuit will trigger with the first pulse and then retrigger with every following pulse. The timing cycle will be reset with every input pulse, and Q will simply remain high since the circuit will never be allowed to time out (see Fig 7.19b). If the pulses at B are stopped, Q will be allowed to time out and will go low 33 μs after the last pulse at B.



(a) Logic diagram

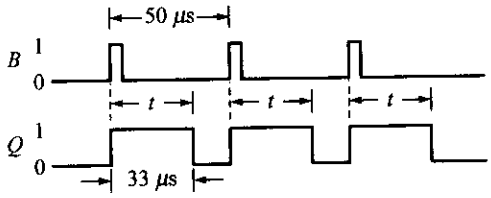
$\bar{A}$	B	$\bar{R}$	Q	
↑	H	H	Trigger	
L	↑	H	Trigger	
L	H	↑	Trigger	H = High
X	X	L	Reset	L = Low X = Don't care
X	X	↓	Reset	↑ = Low to high transition ↓ = High to low transition

(b) Truth table

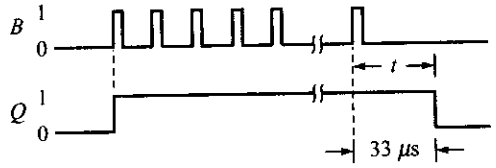


(c) Waveforms

Fig. 7.18 74123



(a)



(b)

Fig. 7.19

SELF-TEST

15. The 74121 is a (retriggerable, nonretriggerable) monostable.
16. The input logic used with a 74121 utilizes a Schmitt trigger. (T or F)
17. The output pulse width of a 74121 is *RC* multiplied by _____.

7.7 PULSE-FORMING CIRCUITS

The monostable circuits discussed in the previous sections have pulse-width times that are predictable to around 10 percent. As such, they do not represent precise timing circuits, but they do offer good short-term stability and are useful in numerous timing applications.

One such application involves the production of a pulse that occurs after a given event with a predictable time delay. For instance, suppose that you are required to generate a 1-ms pulse exactly 2 ms after the operation of a push-button switch. Look at the waveforms in Fig. 7.20b. If the operation of the switch occurs when the waveform labeled *SWITCH* goes high, the desired pulse is shown as OUTPUT. In this case, the delay time t_1 will be set to 2 ms, and the time of the pulse width t_2 will be 1 ms.

The two monostables in the 74123 shown in Fig. 7.20a are connected to provide a delayed pulse. The first circuit provides the delay time as $t_1 = 0.33R_1 \times C_1$, while the second circuit provides the output pulse width as $t_2 = 0.33R_2 \times C_2$. The PT at the INPUT triggers the first circuit into its quasistable state, and its output at $\overline{Q}_1$ goes low. After timing out t_1, $\overline{Q}_1$ goes high, and this transition triggers the second circuit into its quasistable state. The OUTPUT thus goes high until the second circuit times out t_2, and then it returns low.

Example 7.11 The input to the circuit in Fig. 7.20a is changed to a 100-kHz square wave. It is desired to produce a 1-μs pulse 2 μs after every positive transition of the input as shown in Fig. 7.21. Find the proper timing capacitor values, given that both timing resistors are set at 500 Ω.

Solution The capacitor value for the pulse width is found using $t = 0.33\ RC$. Thus:

$$C = \frac{10^{-6}}{0.33 \times 500} = 6000 \text{ pF}$$

The pulse delay capacitor is twice this value, or 0.012 μF.

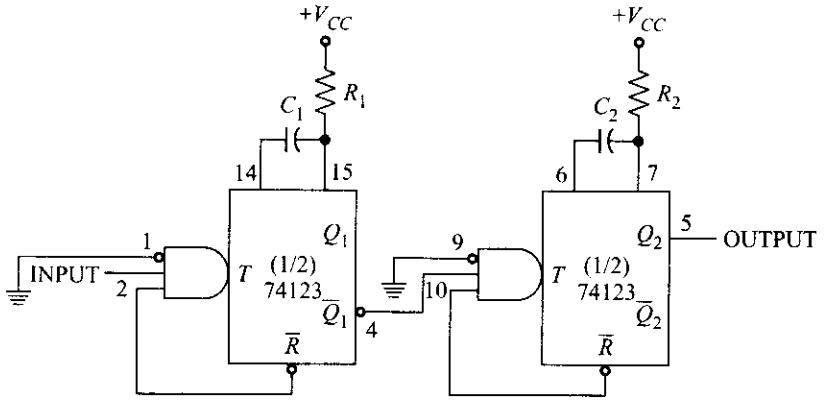
Glitches

Whenever two or more signals at the inputs of a gate are undergoing changes at the same time, an undesired signal may appear at the gate output—this undesired signal is called a *glitch*. For example, in Fig. 7.22a, the gate output at X should be low except during the time when $A = B = C = 1$ as shown. However, there is the possibility of a glitch appearing at the output at two different times. At time T_1, if C happens to go high before A and B go low, a narrow positive spike will appear at the gate output—a glitch! Similarly, a glitch could occur at time T_2 if B happens to go high before A goes low.

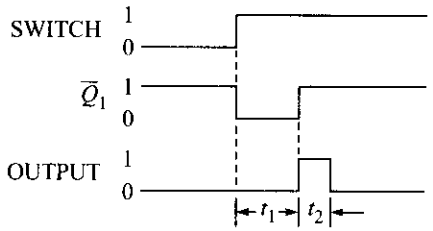
A glitch is an unwanted signal generated usually because of different propagation delay times through different signal paths, and they generally cause random errors to occur in a digital system. They are to be avoided at all costs, and a logic circuit designer must take them into account. One method of avoiding glitches in the instance shown in Fig. 7.22a is to use a strobe pulse.

It is a simple matter to use a pulse delay circuit such as the one shown in Fig. 7.20 to generate a strobe pulse. Consider using the waveform A in Fig. 7.22a as the input to the pulse delay circuit, and set the monostable times to generate a strobe pulse at the midpoint of the positive half cycle of A, as shown in Fig. 7.22b. If the inputs to the AND gate are now A, B, C, and the strobe pulse, the output will occur only when $A = B = C = 1$, and a strobe pulse occurs. The glitches are completely eliminated!

An interesting variation of the pulse delay circuit in Fig. 7.20 is shown in Fig. 7.23a. Here, we have simply connected the $\overline{Q}$ output of the second circuit back to the input of the first circuit. This is a form of positive



(a) A 74123 with DIP pin numbers



(b) Delayed pulse at *OUTPUT*

Fig. 7.20 **Delayed pulse generator**

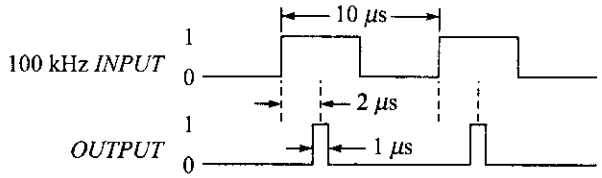
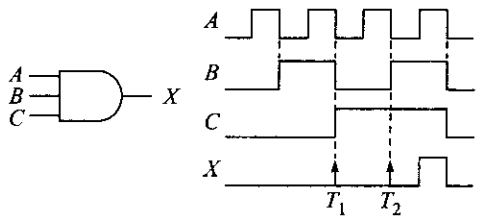
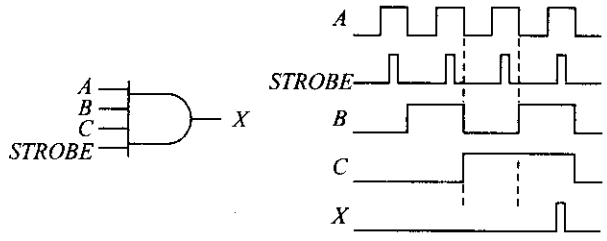


Fig. 7.21



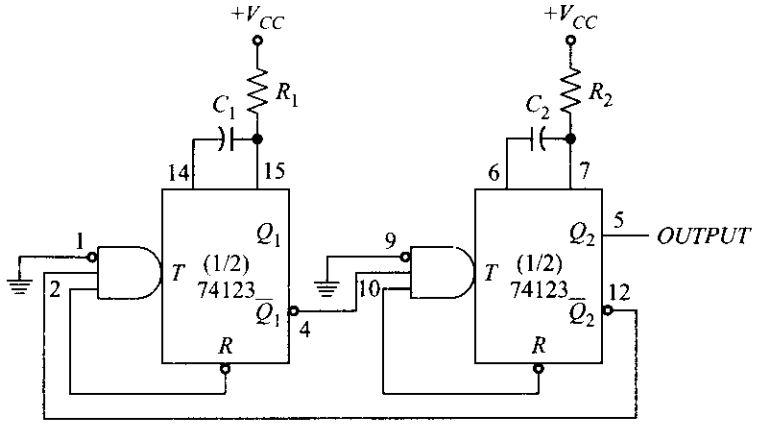
(a) Glitches at T_1 and T_2



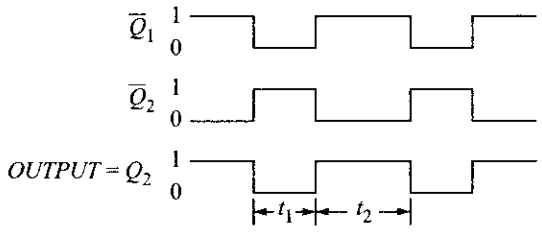
(b) Use of *STROBE* to remove glitches

Fig. 7.22

feedback. As a result, the circuit will oscillate—it becomes astable and generates a rectangular waveform as shown in Fig. 7.23. Here's how it works. The first circuit triggers into its quasistable state. When it times out t_1, the positive transition at $\overline{Q}_1$ will trigger the second circuit. When it times out t_2, the positive transition at $\overline{Q}_2$ will retrigger the first circuit and the cycle will repeat.



(a) Two monostable circuits connected to form an astable free-running oscillator



(b) Waveforms

Fig. 7.23

Independent adjustment of high and low levels of the output waveform is possible by setting the delay times of each individual monostable. Take care to note that since each circuit is edge-triggered, if a transition is missed by either circuit, oscillation will cease!

SELF-TEST

18. What is a glitch?
19. What is a strobe pulse?

PROBLEM SOLVING WITH MULTIPLE METHODS

Problem Design a 100 kHz pulse generator with 40 percent duty cycle.

Solution We can use 555 timer working in astable mode to generate this. Also, we can use monostable circuits 74121 or 74123 and positive feedback for this.

$$\text{Time period, } T = \frac{1}{10^5} \text{ sec.} = 10\ \mu\text{s}$$

If t_L are t_H are the times within T, during which pulse remain LOW and HIGH respectively

$$\text{Duty cycle} = \frac{t_L}{t_L + t_H} = \frac{t_L}{T} = 0.4$$

Thus,
$$t_L = 0.4 \times 10 = 4\ \mu\text{s}$$
$$t_H = T - t_L = 10 - 4 = 6\ \mu\text{s}$$

In Method-1, we show the calculation required for 555 based pulse generator that uses a circuit as shown in Fig.7.13a.

$$t_L = 0.693\ R_B C$$
$$t_H = 0.693\ (R_A + R_B)C$$

Taking ratio, $\frac{t_H}{t_L} = \frac{R_A + R_B}{R_B} = 1 + \frac{R_A}{R_B} = \frac{6}{4}$ Thus, $R_B = 2R_A$

If we choose, $R_A = 1000\ \Omega$ then $R_B = 2000\ \Omega$

Substituting this in say, t_L calculation: $4 \times 10^{-6} = 0.693 \times 2000 \times C$

or, $C = 2.9$ nF

In Method-2, we show the calculation required for 74123 based pulse generator that uses a circuit as shown in Fig.7.23a.

$$t_L = 0.33\ R_1 C_1 = 4\ \mu\text{s}$$
$$t_H = 0.33\ R_2 C_2 = 6\ \mu\text{s}$$

Select say $C_1 = C_2 = C = 1$ nF. Then, $R_1 = 12\ \text{k}\Omega$ and $R_2 = 18\ \text{k}\Omega$

In Method-3, we show the calculation required for 74121 based pulse generator that uses a circuit similar to Fig.7.23a where retriggerable 74123 is replaced by non-retriggerable 74121. From Section 7.6,

$$t_L = 0.69\ R_1 C_1 = 4\ \mu\text{s}$$
$$t_H = 0.69\ R_2 C_2 = 6\ \mu\text{s}$$

Select say $C_1 = C_2 = C = 1$ nF. Then $R_1 = 5.8\ \text{k}\Omega$ and $R_2 = 8.7\ \text{k}\Omega$

SUMMARY

A system clock signal is a periodic waveform (usually a square wave) that has stable high and low levels, very short rise and fall times, and good frequency stability. A circuit widely used to generate a good, stable, TTL-compatible clock waveform is the crystal-controlled circuit shown in Fig. 7.8.

A Schmitt trigger is a switching circuit having two input threshold voltage levels. It exhibits hysteresis, and is useful in cleaning up noisy signals.

The 555 timer is a digital timing circuit that can be connected as either a monostable or an astable circuit. It is widely used in a number of different applications. The 74121 and 74123 monostable circuits both have logic circuits at their inputs that increase the number of possible applications.

A pulse delay circuit, and a free-running astable with adjustable duty cycle are only a few of the many circuits that can be constructed with the use of basic monostable circuits.

GLOSSARY

- ***astable*** Having two output states, neither of which is stable.
- ***asynchronous*** Referring to random events, not coordinated closely with a system clock.
- ***clock*** A periodic waveform (usually a square wave) that is used as a synchronizing signal in a digital system.
- ***clock cycle time*** The time period of a clock signal.
- ***clock stability*** A measure of the frequency stability of a waveform; usually given in parts per million (ppm).
- ***contact bounce*** Opening and closing of a set of contacts as a result of the mechanical bounce that occurs when the device is switched.
- ***dynamic input indicator*** A small triangle used on an input signal line to indicate sensitivity to signal transitions—edge triggering.
- ***fall time*** The time required for a signal to transition from 90 percent of its maximum value down to 10 percent of its maximum.
- ***glitch*** Very narrow positive or negative pulse that appears as an unwanted signal.
- ***monostable*** A circuit that has two output states, only one of which is stable.
- ***NT*** Negative transition.
- ***negative-edge trigger*** An input sensitive to high-to-low signal transitions.
- ***one-shot*** Another term for a monostable circuit.
- ***PT*** Positive transition.
- ***positive-edge trigger*** An input sensitive to low-to-high signal transitions.
- ***propagation delay time*** The time required for a signal to propagate through a circuit, input to output.
- ***rise time*** The time required for a signal to transition from 10 percent of its maximum value up to 90 percent of its maximum.
- ***Schmitt trigger*** A bistable circuit used to produce a rectangular output waveform.
- ***TTL clock*** A circuit that generates a clock waveform that is compatible with standard TTL logic circuits.
- ***10 percent point*** A point on a rising or falling waveform that is equal to 0.1 times its highest value.
- ***90 percent point*** A point on a rising or falling waveform that is equal to 0.9 times its highest value.
- ***555 timer*** A digital timing circuit that can be connected as either an astable or a monostable circuit.

PROBLEMS

Section 7.1

7.1 Calculate the clock cycle time for a system that uses a clock that has a frequency of:

a. 10 MHz b. 6 MHz
c. 750 kHz

7.2 What is the clock frequency if the clock cycle time is 250 ns?

7.3 What is the maximum clock frequency that can be used with a logic gate having a propagation delay of 75 ns?

7.4 You are selecting logic gates that will be used in a system that has a clock frequency of 8 MHz. What is the maximum allowable propagation delay?

7.5 What would be the 10 and 90 percent points on the waveform in Fig. 7.3c if the amplitude goes from 0 to +4.5 V?

Section 7.2

7.6 Find the upper and lower frequency limits of a 5-MHz clock signal that has a stability of 10 ppm.

7.7 A TTL clock uses a series-mode crystal having a resonant frequency of 3.5 MHz. The circuit provides a 24-h stability of 8 ppm. Calculate the oscillator frequency limits.

7.8 The TTL clock shown in Fig. 7.8 uses a crystal that has frequency of 7.5 MHz. Draw the clock output waveform if $+V_{CC}$ is set at +5 V. What is the stability in ppm if the upper limit on the clock frequency is 7,499,900 Hz?

7.9 The NAND gate in Example 7.2 has a propagation delay of 50 ns, and A is a 15-MHz clock. Make a careful sketch of the waveform at the Y output. Assume that B is always high. (***Hint***: Be sure to consider the propagation delay time.)

Section 7.3

7.10 Draw the input and output waveforms for the Schmitt trigger in Fig. 7.10, assuming that the input voltage is $V = 3.0 \cos 1000t$.

7.11 Draw carefully the waveforms at points A, B, and C in Fig. 7.24.

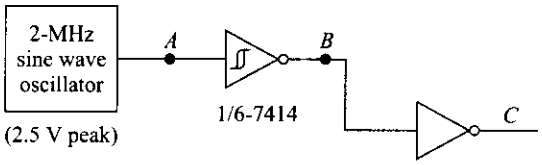


Fig. 7.24

7.12 Draw the transfer curve for a Schmitt trigger if $V_{T+} = +1.0$ V, $V_{T-} = -1.0$ V, high state = +5 Vdc, and low state = 0 Vdc.

7.13 Draw the output voltage for the Schmitt trigger in Prob. 7.12 if $V_i = 2 \sin \omega t$ V.

Section 7.4

7.14 Determine the frequency of oscillation for the 555 timer in Fig. 7.13, given $R_A = R_B = 47$ kΩ and $C = 1000$ pF. Calculate the values of t_1 and t_2, and carefully sketch the output waveform.

7.15 Determine the frequency of oscillation for the 555 timer in Fig. 7.13, given $R_A = 5000$ Ω, $R_B = 7500$ Ω, and $C = 1500$ pF. Calculate values for t_1 and t_2, and carefully sketch the output waveform.

7.16 Use the nomogram in Fig. 7.13b to find $(R_A + 2R_B)$, given $C = 0.1$ μF and that the desired frequency is 1 kHz. Check the results by using the formula given for the frequency.

7.17 Calculate the duty cycle for the circuit in Prob. 7.13. For Prob. 7.14.

7.18 Derive the expression

$$\text{Duty cycle} = R_B/(R_A + 2R_B)$$

7.19 It is desired to have a duty cycle of 25 percent for the circuit in Prob. 7.15. Find the correct values for the two resistors.

Section 7.5

7.20 Calculate the output pulse width for the timer in Fig. 7.15 for a 4,7-kΩ resistor and a 1.5-μF capacitor.

7.21 Calculate the output pulse width for the circuit in Problem 7.20, assuming that the resistor is halved.

7.22 Calculate the output pulse width for the circuit in Problem 7.20, assuming that the capacitor is doubled.

7.23 Find the capacitor value necessary to generate a 15-ms pulse width for the monostable in Fig. 7.15, given $R_A = 100$ kΩ.

7.24 A 500-Hz square wave is used as the trigger input for the circuit described in Example 7.7.

Make a careful sketch of the input and output waveforms (similar to those in Fig. 7.15b).

7.25 Repeat Prob. 7.24, assuming that the trigger input is changed to a 1-kHz square wave.

Section 7.6

7.26 In the 74121 in Fig. 7.16, $R = 47\ k\Omega$ and $C = 10{,}000$ pF. Calculate the output pulse width.

7.27 Redraw the 74121 logic diagram in Fig. 7.16a and show how to connect the circuit such that it will trigger on the positive transitions of a square wave. For $R = 51\ k\Omega$, determine a value of C such that the output pulse will have a width of 750 μs.

7.28 Repeat Prob. 7.27, but make the circuit trigger on negative transitions of the square wave.

7.29 Using the circuit described in Prob. 7.27, make a careful sketch of the input and output waveforms, assuming the input square wave has a frequency of:

a. 1 kHz b. 5 kHz

7.30 In the 74123 in Fig. 7.19, $R = 47\ k\Omega$ and $C = 10{,}000$ pF. Calculate the output pulse width.

7.31 Redraw the 74123 logic circuit shown in Fig. 7.18 and show how to connect the circuit such that it will trigger on positive transitions of a square wave. Given $R = 51\ k\Omega$, determine a value of C such that the output pulse will have a width of 750 μs.

7.32 Repeat Problem 7.31, but make the circuit trigger on negative transitions of the square wave.

7.33 Using the circuit described in Prob. 7.31, make a careful sketch of the input and output waveforms if the input square wave has a frequency of:

a. 1 kHz b. 5 kHz

Section 7.7

7.34 The input to the circuit in Fig. 7.20 is a 250-kHz square wave. Determine the proper timing capacitor values to generate a string of positive-going, 0.1-μs pulses, delayed by 2.0 μs from the rising edges of the input square wave. Assume $R_1 = R_2 = 1\ k\Omega$.

7.35 Draw the waveforms, input and output, for the circuit in Fig. 7.20, given that both timing resistors are 470 Ω, $C_1 = 0.1\ \mu F$, $C_2 = 0.01\ \mu F$, and the input waveform has a frequency of 20 kHz.

7.36 Show how to use the circuit in Fig. 7.20 to generate a 0.2-μs strobe pulse centered on the positive half cycle of a 200-kHz square wave (similar to Fig. 7.22b). Draw the complete circuit and calculate all timing resistor and capacitor values. Assume $R_1 = R_2 = 1\ k\Omega$.

7.37 Calculate values for the timing resistors and capacitors in Fig. 7.23 to generate a clock waveform that has:

a. A frequency of 100 kHz and a duty cycle of 25 percent

b. A frequency of 500 kHz and a duty cycle of 50 percent

LABORATORY EXPERIMENT

AIM: The aim of this experiment is to implement a 100 kHz pulse generator with 40 percent duty cycle.

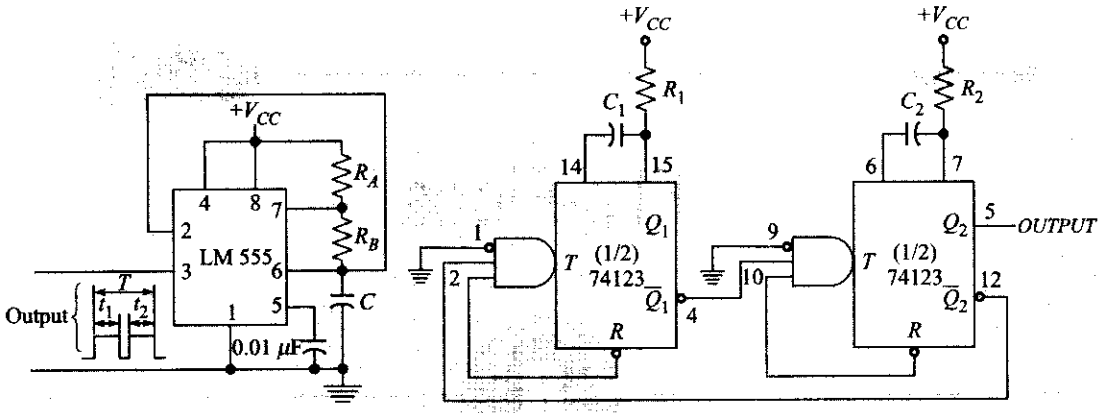
Theory: Refer to Fig. 7.13b. The 555 based pulse generator follows the following two relations.

$$t_L = 0.693\ R_B C$$
$$t_H = 0.693(R_A + R_B)C$$

Refer to Fig. 7.23a. 74123 essentially is a monostable and can be used in positive feedback. It follows the relation

$$t_L = 0.33\ R_1 C_1$$
$$t_H = 0.33\ R_2 C_2$$

Apparatus: 5 VDC Power supply, Multimeter, Bread Board, and Oscilloscope.



Work element: Study the working of IC 555 and 74123, and understand the different input outputs. From above relations, calculate the resistance and capacitance values. See the waveform in oscilloscope. Calculate duty cycle from the oscilloscope reading and compare with theoretical value. Conduct similar exercise for 74123 based circuit as shown. Repeat the experiment with other combinations of resistance and capacitance values.

Answers to Self-tests

1. An input is sensitive to PTs, and the circuit output changes synchronously with PTs. The circuit output changes in synchronism with NTs.
2. It means that a circuit input is sensitive to PTs. (See Fig. 7.6b.)
3. The logic symbol for an input sensitive to NTs is a bubble in front of a dynamic input indicator. (See Fig. 7.7b.)
4. A series mode offers low impedance at resonance, thus providing positive feedback for oscillation. A parallel mode offers high impedance at resonance, and thus provides insufficient feedback to produce oscillation.
5. Unnecessary. They simply simulate a load condition.
6. It means that the circuit has two input threshold voltage levels—an upper threshold and a lower threshold. By contrast, a simple inverter has only a single threshold voltage level.
7. Noninverting: the input and output are both high (or both low) at the same time (no phase shift). Inverting: 180° phase shift between input and output.
8. Schmitt triggers can be used to clean up a noisy signal or to change a signal having a slow rise time into one having a fast rise time.
9. A circuit has two output states, neither of which is stable.
10. True
11. Inversely
12. A circuit has two output states, one of which is stable.
13. True
14. The stable state is low.
15. Nonretriggerable
16. True
17. 0.69
18. Glitches are the unwanted pulses appearing at the output of a gate when two or more inputs change state simultaneously.
19. A strobe pulse is a pulse timed to eliminate glitches.

Flip-Flops

8

OBJECTIVES

- **Describe the operation of the basic RS flip-flop and explain the purpose of the additional input on the gated (clocked) RS flip-flop**
- **Show the truth table for the edge-triggered. RS flip-flop, edge-triggered D flip-flop, and edge-triggered JK flip-flop**
- **Discuss some of the timing problems related to flip-flops**
- **Draw a diagram of a JK master-slave flip-flop and describe its operation**
- **State the cause of contact bounce and describe a solution for this problem**
- **Describe characteristic equations of Flip-Flops and analysis techniques of sequential circuit**
- **Describe excitation table of Flip-Flops and explain conversion of Flip-Flops as synthesis example**

The outputs of the digital circuits considered previously are dependent entirely on their inputs. That is, if an input changes state, output may also change state. However, there are requirements for a digital device or circuit whose output will remain unchanged, once set, even if there is a change in input level(s). Such a device 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. The elimination of switch contact bounce is a clever application utilizing the unique operating characteristics of flip-flops. In a sequential logic circuit flip-flops serve as key memory elements. Analysis of such circuits are done through truth tables or characteristic equations of flip-flops. The analysis result is normally presented through state