

CHAPTER TWO

Data Representation

(选自 M. MORRIS MANO 《COMPUTER SYSTEM ARCHITECTURE》THIRD EDITION)

IN THIS CHAPTER

- 2-1 Data Types
- 2-2 Complements
- 2-3 Fixed-Point Representation
- 2-4 Floating-Point Representation
- 2-5 Other Binary Codes
- 2-6 Error Detection Codes

2-1 Data Types

Binary information in digital computers is stored in memory or processor registers. Registers contain either data or control information. Control information is a bit or a group of bits used to specify the sequence of command signals needed for manipulation of the data in other registers. Data are numbers and other binary-coded information that are operated on to achieve required computational results. In this chapter we present the most common types of data found in digital computers and show how the various data types are represented in binary-coded form in computer registers.

The data types found in the registers of digital computers may be classified as being one of the following categories: (1) numbers used in arithmetic computations, (2) letters of the alphabet used in data processing, and (3) other discrete symbols used for specific purposes. All types of data, except binary numbers, are represented in computer registers in binary-coded form. This is because registers are made up of flip-flops and flip-flops are two-state devices that can store only 1's and 0's. The binary number system is the most natural system to use in a digital computer. But sometimes it is convenient to employ different number systems, especially the decimal number system, since it is used by people to perform arithmetic computations.

Number Systems

A number system of *base*, or *radix*, r is a system that uses distinct symbols for r digits. Numbers are represented by a string of digit symbols. To determine the quantity that the number represents, it is necessary to multiply each digit by an integer power of r and then form the sum of all weighted digits. For example, the decimal number system in everyday use employs the radix 10 system. The 10 symbols are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. The string of digits 724.5 is interpreted to represent the quantity.

$$7 \times 10^2 + 2 \times 10^1 + 4 \times 10^0 + 5 \times 10^{-1}$$

that is, 7 hundreds, plus 2 tens, plus 4 units, plus 5 tenths. Every decimal number can be similarly interpreted to find the quantity it represents.

The *binary* number system uses the radix 2. The two digit symbols used are 0 and 1. The string of digits 101101 is interpreted to represent the quantity

$$1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 45$$

To distinguish between different radix numbers, the digits will be enclosed in parentheses and the radix of the number inserted as a subscript. For example, to show the equality between decimal and binary forty-five we will write $(101101)_2 = (45)_{10}$

Besides the decimal and binary number systems, the *octal* (radix 8) and *hexadecimal* (radix 16) are important in digital computer work. The eight symbols of the octal system are 0,1,2,3,4,5,6, and 7. The 16 symbols of the hexadecimal system are 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E, and F. The last six symbols are, unfortunately, identical to the letters of the alphabet and can cause confusion at times. However, this is the convention that has been adopted. When used to represent hexadecimal digits, the symbols A,B,C,D,E,F correspond to the decimal numbers 10,11,12,13,14,15, respectively.

A number in radix r can be converted to the familiar decimal system by forming the sum of the weighted digits. For example, octal 736.4 is converted to decimal as follows:

$$\begin{aligned} (736.4)_8 &= 7 \times 8^2 + 3 \times 8^1 + 6 \times 8^0 + 4 \times 8^{-1} \\ &= 7 \times 64 + 3 \times 8 + 6 \times 1 + 4/8 = (478.5)_{10} \end{aligned}$$

The equivalent decimal number of hexadecimal F3 is obtained from the following calculation:

$$(F3)_{16} = F \times 16 + 3 = 15 \times 16 + 3 = (243)_{10}$$

Conversion from decimal to its equivalent representation in the radix r system is carried out by separating the number into its *integer* and *fraction* parts and converting each part separately. The conversion of a decimal integer into a base r representation is done by successive divisions by r and accumulation of the remainders. The conversion of a decimal fraction to radix r representation is accomplished by successive multiplications by r and accumulation of the integer digits so obtained. Figure 2-1 demonstrates these procedures.

Figure 2-1 Conversion of decimal 41.6875 into binary.

Integer = 41	Fraction = 0.6875
$\begin{array}{r l} 41 & 1 \\ 20 & 0 \\ 10 & 0 \\ 5 & 0 \\ 2 & 1 \\ 1 & 0 \\ 0 & 1 \end{array}$	$\begin{array}{r} 0.6875 \\ \times 2 \\ \hline 1.3750 \\ \times 2 \\ \hline 0.7500 \\ \times 2 \\ \hline 1.5000 \\ \times 2 \\ \hline 1.0000 \end{array}$
$(41)_{10} = (101001)_2$	$(0.6875)_{10} = (0.1011)_2$
$(41.6875)_{10} = (101001.1011)_2$	

The conversion of decimal 41.6875 into binary is done by first separating the number into its integer part 41 and fraction part .6875. The integer part is converted by dividing 41 by $r=2$ to give an integer quotient of 20 and a remainder of 1. The quotient is again divided by 2 to give a new quotient and remainder. This process is repeated until the integer quotient becomes 0. The coefficients of the binary number are obtained from the remainders with the first remainder giving the low-order bit of the converted binary number.

The fraction part is converted by multiplying it by $r=2$ to give an integer and a fraction. The

new fraction (*without* the integer) is multiplied again by 2 to give a new integer and a new fraction. This process is repeated until the fraction part becomes zero or until the number of digits obtained gives the required accuracy. The coefficients of the binary fraction are obtained from the integer digits with the first integer computed being the digit to be placed next to the binary point. Finally, the two parts are combined to give the total required conversion.

Octal and Hexadecimal Numbers

The conversion from and to binary, octal, and hexadecimal representation plays an important part in digital computers. Since $2^3=8$ and $2^4=16$, each octal digit corresponds to three binary digits and each hexadecimal digit corresponds to four binary digits. The conversion from binary to octal is easily accomplished by partitioning the binary number into groups of three bits each. The corresponding octal digit is then assigned to each group of bits and the string of digits so obtained gives the octal equivalent of the binary number. Consider, for example, a 16-bit register. Physically, one may think of the register as composed of 16 binary storage cells, with each cell capable of holding either a 1 or a 0. Suppose that the bit configuration stored in the register is as shown in fig. 2-2. Since a binary number consists of a string of 1's and 0's, the 16-bit register can be used to store any binary number from 0 to $2^{16}-1$. For the particular example shown, the binary number stored in the register is the equivalent of decimal 44899. Starting from the low-order bit, we partition the register into groups of three bits each (the sixteenth bit remains in a group by itself). Each group of three bits is assigned its octal equivalent and placed on top of the register. The string of octal digits so obtained represents the octal equivalent of the binary number.

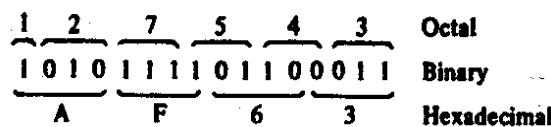


Figure 2-2 Binary, octal, and hexadecimal conversion.

Conversion from binary to hexadecimal is similar except that the bits are divided into groups of four. The corresponding hexadecimal digit for each group of four bits is written as shown below the register of Fig. 2-2. The string of hexadecimal digits so obtained represents the hexadecimal equivalent of the binary number. The corresponding octal digit for each group of three bits is easily remembered after studying the first eight entries listed in table 2-1. The correspondence between a hexadecimal digit and its equivalent 4-bit code can be found in the first 16 entries of Table 2-2.

Table 2-1 lists a few octal numbers and their representation in registers in binary-coded form. The binary code is obtained by the procedure explained above. Each octal digit is assigned a 3-bit code as specified by the entries of the first eight digits in the table. Similarly, Table 2-2 lists a few hexadecimal numbers and their representation in registers in binary-coded form. Here the binary code is obtained by assigning to each hexadecimal digit the 4-bit code listed in the first 16 entries of the table.

TABLE 2-1 Binary-Coded Octal Numbers

Octal number	Binary-coded octal	Decimal equivalent	
0	000	0	Code for one octal digit
1	001	1	
2	010	2	
3	011	3	
4	100	4	
5	101	5	
6	110	6	
7	111	7	
10	001 000	8	
11	001 001	9	
12	001 010	10	
24	010 100	20	
62	110 010	50	
143	001 100 011	99	
370	011 111 000	248	

TABLE 2-2 Binary-Coded Hexadecimal Numbers

Hexadecimal number	Binary-coded hexadecimal	Decimal equivalent	
0	0000	0	Code for one hexadecimal digit
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	
A	1010	10	
B	1011	11	
C	1100	12	
D	1101	13	
E	1110	14	
F	1111	15	
14	0001 0100	20	
32	0011 0010	50	
63	0110 0011	99	
F8	1111 1000	248	

Comparing the binary-coded octal and hexadecimal numbers with their binary number equivalent we find that the bit combination in all three representations is exactly the same. For example, decimal 99, when converted to binary, becomes 1100011. The binary-coded octal equivalent of decimal 99 is 001 100 011 and the binary-coded hexadecimal of decimal 99 is 0110 0011. If we neglect the leading zeros in these three binary representations, we find that their bit combination is identical. This should be so because of the straightforward conversion that exists between binary numbers and octal or hexadecimal. The point of all this is that a string of 1's and 0's stored in a register could represent a binary number, but this same string of bits may be interpreted as holding an octal number in binary-coded form (if we divide the bits in groups of three) or as holding a hexadecimal number in binary-coded form (if we divide the bits in groups of four).

The registers in a digital computer contain many bits. Specifying the content of registers by their binary values will require a long string of binary digits. It is more convenient to specify content of registers by their octal or hexadecimal equivalent. The number of digits is reduced by one-third in the octal designation and by one-fourth in the hexadecimal designation. For example, the binary number 1111 1111 1111 has 12 digits. It can be expressed in octals as 7777(four digits) or in hexadecimal as FFF (three digits). Computer manuals invariably choose either the octal or the hexadecimal designation for specifying contents of registers.

Decimal representation

The binary number system is the most natural system for a computer, but people are accustomed to the decimal system. One way to solve this conflict is to convert all input decimal numbers into binary numbers, let the computer perform all arithmetic operations in binary and then convert the binary results back to decimal for the human user to understand. However, it is also possible for the computer to perform arithmetic operations directly with decimal numbers provided they are placed in registers in a coded form. Decimal numbers enter the computer usually as binary-coded alphanumeric characters. These codes, introduced later, may contain from six to eight bits for each decimal digit. When decimal numbers are used for internal arithmetic computations, they are

converted to a binary code with four bits per digit.

A binary code is a group of n bits that assume up to 2^n distinct combinations of 1's and 0's with each combination representing one element of the set that is being coded. For example, a set of four elements can be coded by a 2-bit code with each element assigned one of the following bit combinations; 00,01,10, or 11. A set of eight elements requires a 3-bit code, a set of 16 elements requires a 4-bit code, and so on. A binary code will have some unassigned bit combinations if the number of elements in the set is not a multiple power of 2. The 1- decimal digits form such a set. A binary code that distinguishes among 10 elements must contain at least four bits, but six combinations will remain unassigned. Numerous different codes can be obtained by arranging four bits in 10 distinct combinations. The bit assignment most commonly used for the decimal digits is the straight binary assignment listed in the first 10 entries of Table 2-3. This particular code is called *binary-coded decimal* and is commonly referred to by its abbreviation BCD. Other decimal codes are sometimes used and a few of them are given in Sec. 2-5.

It is very important to understand the difference between the *conversion* of decimal numbers into binary and the *binary coding* of decimal numbers. For example, when converted to a binary number, the decimal number 99 is represented by the string of bits 1100011, but when represented in BCD, it becomes 1001 1001. The *only* difference between a decimal number represented by the familiar digit symbols 0,1,2,...,9 and the BCD symbols 0001,0010,...,1001 is in the symbols used to represent the digits—the number itself is exactly the same. A few decimal numbers and their representation in BCD are listed in Table 2-3.

TABLE 2-3 Binary-Coded Decimal (BCD) Numbers

Decimal number	Binary-coded decimal (BCD) number	
0	0000	Code for one decimal digit
1	0001	
2	0010	
3	0011	
4	0100	
5	0101	
6	0110	
7	0111	
8	1000	
9	1001	
10	0001 0000	
20	0010 0000	
50	0101 0000	
99	1001 1001	
248	0010 0100 1000	

TABLE 2-4 American Standard Code for Information Interchange (ASCII)

Character	Binary code	Character	Binary code
A	100 0001	0	011 0000
B	100 0010	1	011 0001
C	100 0011	2	011 0010
D	100 0100	3	011 0011
E	100 0101	4	011 0100
F	100 0110	5	011 0101
G	100 0111	6	011 0110
H	100 1000	7	011 0111
I	100 1001	8	011 1000
J	100 1010	9	011 1001
K	100 1011		
L	100 1100		
M	100 1101	space	010 0000
N	100 1110	.	010 1110
O	100 1111	(010 1000
P	101 0000	+	010 1011
Q	101 0001	\$	010 0100
R	101 0010	*	010 1010
S	101 0011)	010 1001
T	101 0100	—	010 1101
U	101 0101	/	010 1111
V	101 0110	,	010 1100
W	101 0111	=	011 1101
X	101 1000		
Y	101 1001		
Z	101 1010		

Alphanumeric representation

Many applications of digital computers require the handling of data that consist not only of numbers, but also of the letters of the alphabet and certain special characters. An *alphanumeric character* set is a set of elements that includes the 10 decimal digits, the 26 letters of the alphabet and a number of special characters, such as \$, +, and =. Such a set contains between 32 and 64 elements (if only uppercase letters are included) or between 64 and 128 (if both uppercase and lowercase letters are included). In the first case, the binary code will require six bits and in the second case, seven bits. The standard alphanumeric binary code is the ASCII (American Standard code for Information Interchange), which uses seven bits to code 128 characters. The binary code for the uppercase letters, the decimal digits, and a few special characters is listed in Table 2-4. Note that the decimal digits in ASCII can be converted to BCD by removing the three high-order bits, 011. A complete list of ASCII characters is provided in Table 11-1.

Binary codes play an important part in digital computer operations. The codes must be in binary because registers can only hold binary information. One must realize that binary codes merely change the symbols, not the meaning of the discrete elements they represent. The operations specified for digital computers must take into consideration the meaning of the bits stored in registers so that operations are performed on operands of the same type. In inspecting the bits of a computer register at random, one is likely to find that it represents some type of coded information rather than a binary number.

Binary codes can be formulated for any set of discrete elements such as the musical notes and chess pieces and their positions on the chessboard. Binary codes are also used to formulate instructions that specify control information for the computer. This chapter is concerned with *data* representation. Instruction codes are discussed in Chap. 5.

2-2 Complements

Complements are used in digital computers for simplifying the subtraction operation and for logical manipulation. There are two types of complements for each base r system: the r 's complement and the $(r-1)$'s complement. When the value of the base r is substituted in the name, the two types are referred to as the 2's and 1's complement for binary numbers and the 10's and 9's complement for decimal numbers.

$(r-1)$'s Complement

Given a number N in base r having n digits, the $(r-1)$'s complement of N is defined as $(r^n-1)-N$. For decimal numbers $r=10$ and $r-1=9$, so the 9's complement of N is $(10^n-1)-N$. Now, 10^n represents a number that consists of a single 1 followed by n 0's. 10^n-1 is a number represented by n 9's. For example, with $n=4$ we have $10^4=10000$ and $10^4-1=9999$. It follows that the 9's complement of a decimal number is obtained by subtracting each digit from 9. For example, the 9's complement of 546700 is $999999-546700=453299$ and the 9's complement of 12389 is $99999-12389=87610$.

For binary numbers, $r=2$ and $r-1=1$, so the 1's complement of N is $(2^n-1)-N$. Again, 2^n is represented by a binary number that consists of a 1 followed by n 0's. 2^n-1 is a binary number represented by n 1's. For example, with $n=4$, we have $2^4=(10000)_2$ and $2^4-1=(1111)_2$. Thus the

1's complement of a binary number is obtained by subtracting each digit from 1. However, the subtraction of a binary digit from 1 causes the bit to change from 0 to 1 or from 1 to 0. Therefore, the 1's complement of a binary number is formed by changing 1's into 0's and 0's into 1's. For example, the 1's complement of 1011001 is 0100110 and the 1's complement of 0001111 is 1110000.

The $(r-1)$'s complement of octal or hexadecimal numbers are obtained by subtracting each digit from 7 or F (decimal 15) respectively.

$(r$'s) Complement

The r 's complement of an n -digit number N in base r is defined as $r^n - N$ for $N \neq 0$ and 0 for $N = 0$. Comparing with the $(r-1)$'s complement, we note that the r 's complement is obtained by adding 1 to the $(r-1)$'s complement since $r^n - N = [(r^n - 1) - N] + 1$. Thus the 10's complement of the decimal 2389 is $7610 + 1 = 7611$ and is obtained by adding 1 to the 9's complement value. The 2's complement of binary 101100 is $010011 + 1 = 010100$ and is obtained by adding 1 to the 1's complement value.

Since 10^n is a number represented by a 1 followed by n 0's, then $10^n - N$, which is the 10's complement of N , can be formed also by leaving all least significant 0's unchanged, subtracting the first nonzero least significant digit from 10, and then subtracting all higher significant digits from 9. The 10's complement of 246700 is 753300 and is obtained by leaving the two zeros unchanged, subtracting 7 from 10, and subtracting the other three digits from 9. Similarly, the 2's complement can be formed by leaving all least significant 0's and the first 1 unchanged, and then replacing 1's by 0's and 0's by 1's in all other higher significant bits. The 2's complement of 1101100 is 0010100 and is obtained by leaving the two low-order 0's and the first 1 unchanged, and then replacing 1's by 0's and 0's by 1's in the other four most significant bits.

In the definitions above it was assumed that the numbers do not have a radix point. If the original number N contains a radix point, it should be removed temporarily to form the r 's or $(r-1)$'s complement. The radix point is then restored to the complemented number in the same relative position. It is also worth mentioning that the complement of the complement restores the number to its original value. The r 's complement of N is $r^n - N$. The complement of the complement is $r^n - (r^n - N) = N$ giving back the original number.

Subtraction of Unsigned Numbers

The direct method of subtraction taught in elementary schools uses the borrow concept. In this method we borrow a 1 from a higher significant position when the minuend digit is smaller than the corresponding subtrahend digit. This seems to be easiest when people perform subtraction with paper and pencil. When subtraction is implemented with digital hardware, this method is found to be less efficient than the method that uses complements.

The subtraction of two n -digit unsigned numbers $M - N$ ($N \neq 0$) in base r can be done as follows:

1. Add the minuend M to the r 's complement of the subtrahend N . This performs $M + (r^n - N) = M - N + r^n$.
2. If $M \geq N$, the sum will produce an end carry r^n which is discarded, and what is left is the

result $M-N$.

3. If $M < N$, the sum does not produce an end carry and is equal to $r^n - (N-M)$, which is the r 's complement of $(N-M)$. To obtain the answer in a familiar form, take the r 's complement of the sum and place a negative sign in front.

Consider, for example, the subtraction $72532-13250=59282$. The 10's complement of 13250 is 86750. Therefore:

$$\begin{array}{rcl} M & = & 72532 \\ 10\text{'s complement of } N & = & +86750 \\ \hline \text{Sum} & = & 159282 \\ \text{Discard end carry } 10^5 & = & -100000 \\ \hline \text{Answer} & = & 59282 \end{array}$$

Now consider an example with $M < N$. The subtraction $13250-72532$ produces negative 59282. Using the procedure with complements. We have

$$\begin{array}{rcl} M & = & 13250 \\ 10\text{'s complement of } N & = & +27468 \\ \hline \text{Sum} & = & 40718 \end{array}$$

There is no end carry

Answer is negative $59282=10\text{'s complement of } 40718$

Since we are dealing with unsigned numbers, there is really no way to get an unsigned result for the second example. When working with paper and pencil, we recognize that the answer must be changed to a signed negative number. When subtracting with complements, the negative answer is recognized by the absence of the end carry and the complemented result.

Subtraction with complements is done with binary numbers in a similar manner using the same procedure outlined above. Using the two binary numbers $X=1010100$ and $Y=1000011$, we perform the subtraction $X-Y$ and $Y-X$ using 2's complements:

$$\begin{array}{rcl} X & = & 1010100 \\ 2\text{'s complement of } Y & = & +0111101 \\ \hline \text{Sum} & = & 10010001 \\ \text{Discard end carry } 2^7 & = & -10000000 \\ \hline \text{Answer } X-Y & = & 0010001 \end{array}$$

$$\begin{array}{rcl} Y & = & 1000011 \\ 2\text{'s complement of } X & = & +0101100 \\ \hline \text{Sum} & = & 1101111 \end{array}$$

There is no end carry

Answer is negative $0010001 = 2\text{'s complement of } 1101111$

2-3 Fixed-Point Representation

Positive integers, including zero, can be represented as unsigned numbers. However, to represent negative integers, we need a notation for negative values. In ordinary arithmetic, a negative

number is indicated by a minus sign and a positive number by a plus sign. Because of hardware limitations, computers must represent everything with 1's and 0's, including the sign of a number. As a consequence, it is customary to represent the sign with a bit placed in the leftmost position of the number. The convention is to make the sign bit equal to 0 for positive and to 1 for negative.

In addition to the sign, a number may have a binary (or decimal) point. The position of the binary point is needed to represent fractions, integers, or mixed integer-fraction numbers. The representation of the binary point in a register is complicated by the fact that it is characterized by a position in the register. There are two ways of specifying the position of the binary point in a register: by giving it a fixed position or by employing a floating-point representation. The fixed-point method assumes that the binary point is always fixed in one position. The two positions most widely used are (1) a binary point in the extreme left of the register to make the stored number a fraction, and (2) a binary point in the extreme right of the register to make the stored number an integer. In either case, the binary point is not actually present, but its presence is assumed from the fact that the number stored in the register is treated as a fraction or as an integer. The floating-point representation uses a second register to store a number that designates the position of the decimal point in the first register. Floating-point representation is discussed further in the next section.

Integer Representation

When an integer binary number is positive, the sign is represented by 0 and the magnitude by a positive binary number. When the number is negative, the sign is represented by 1 but the rest of the number may be represented in one of three possible ways:

1. Signed-magnitude representation
2. Signed-1's complement representation
3. Signed 2's complement representation

The signed-magnitude representation of a negative number consists of the magnitude and a negative sign. In the other two representations, the negative number is represented in either the 1's or 2's complement of its positive value. As an example, consider the signed number 14 stored in an 8-bit register. +14 is represented by a sign bit of 0 in the leftmost position followed by the binary equivalent of 14.00001110. Note that each of the eight bits of the register must have a value and therefore 0's must be inserted in the most significant positions following the sign bit. Although there is only one way to represent +14, there are three different ways to represent -14 with eight bits.

In signed-Magnitude representation	1 0001110
In signed-1's complement representation	1 1110001
In signed-2's complement representation	1 1110010

The signed-magnitude representation of -14 is obtained from +14 by complementing only the sign bit. The signed-1's complement representation of -14 is obtained by complementing all the bits of +14, including the sign bit. The signed-2's complement representation is obtained by taking the 2's complement of the positive number, including its sign bit.

The signed-magnitude system is used in ordinary arithmetic but is awkward when employed in computer arithmetic. Therefore, the signed-complement is normally used. The 1's complement

imposes difficulties because it has two representations of 0 (+0 and -0). It is seldom used for arithmetic operations except in some older computers. The 1's complement is useful as a logical operation since the change of 1 to 0 or 0 to 1 is equivalent to a logical complement operation. The following discussion of signed binary arithmetic deals exclusively with the signed-2's complement representation of negative numbers.

Arithmetic Addition

The addition of two numbers in the signed-magnitude system follows the rules of ordinary arithmetic. If the signs are the same, we add the two magnitudes and give the sum the common sign. If the signs are different, we subtract the smaller magnitude from the larger and give the result the sign of the larger magnitude. For example, $(+25)+(-37)=-(37-25)=-12$ and is done by subtracting the smaller magnitude 25 from the larger magnitude 37 and using the sign of 37 for the sign of the result. This is a process that requires the comparison of the signs and the magnitudes and then performing either addition or subtraction. (The procedure for adding binary numbers in signed-magnitude representation is described in Sec. 10-2.) By contrast, the rule for adding numbers in the signed-2's complement system does not require a comparison or subtraction, only addition and complementation. The procedure is very simple and can be stated as follows: Add the two numbers, including their sign bits, and discard any carry out of the sign (leftmost) bit position. Numerical examples for addition are shown below. Note that negative numbers must initially be in 2's complement and that if the sum obtained after the addition is negative, it is in 2's complement form.

$$\begin{array}{rcl}
 +6 & 00000110 & -6 & 11111010 \\
 +13 & 00001101 & +13 & \underline{00001101} \\
 +19 & 00010011 & +7 & \underline{00000111}
 \end{array}$$

$$\begin{array}{rcl}
 +6 & 00000110 & -6 & 11111010 \\
 -13 & \underline{11110011} & -13 & \underline{11110011} \\
 -7 & 00010011 & -19 & \underline{00000111}
 \end{array}$$

In each of the four cases, the operation performed is always addition, including the sign bits. Any carry out of the sign bit position is discarded, and negative results are automatically in 2's complement form.

The complement form of representing negative numbers is unfamiliar to people used to the signed-magnitude system. To determine the value of a negative number when in signed-2's complement, it is necessary to convert it to a positive number to place it in signed-2's complement, it is necessary to convert it to a positive number to place it in a more familiar form. For example, the signed binary number 11111001 is negative because the left most bit is 1. Its 2's complement is 00000111, which is the binary equivalent of +7. We therefore recognize the original negative number to be equal to -7.

Arithmetic Subtraction

Subtraction of two signed binary numbers when negative numbers are in 2's complement form is very simple and can be stated as follows: Take the 2's complement of the subtrahend (including the sign bit) and add it to the minuend (including the sign bit). A carry out of the sign bit position is discarded.

This procedure stems from the fact that a subtraction operation can be changed to an addition operation if the sign of the subtrahend is changed. This is demonstrated by the following relationship:

$$(\pm A) - (+B) = (\pm A) + (-B)$$

$$(\pm A) - (-B) = (\pm A) + (+B)$$

But changing a positive number to a negative number is easily done by taking its 2's complement. The reverse is also true because the complement of a negative number in complement form produces the equivalent positive number. Consider the subtraction of $(-6) - (-13) = +7$. In binary with eight bits this is written as 11111010-11110011. The subtraction is changed to addition by taking the 2's complement of the subtrahend (-13) to give (+13). In binary this is 11111010 = 00001101 = 10000111. Removing the end carry, we obtain the correct answer 00000111 (+7).

It is worth noting that binary numbers in the signed-2's complement system are added and subtracted by the same basic addition and subtraction rules as unsigned numbers. Therefore, computers need only one common hardware circuit to handle both types of arithmetic. The user or programmer must interpret the results of such addition or subtraction differently depending on whether it is assumed that the numbers are signed or unsigned.

Overflow

When two numbers of n digits each are added and the sum occupies $n+1$ digits, we say that an overflow occurred. When the addition is performed with paper and pencil, an overflow is not a problem since there is no limit to the width of the page to write down the sum. An overflow is a problem in digital computers because the width of registers is finite. A result that contains $n+1$ bits cannot be accommodated in a register with a standard length of n bits. For this reason, many computers detect the occurrence of an overflow, and when it occurs, a corresponding flip-flop is set which can then be checked by the user.

The detection of an overflow after the addition of two binary numbers depends on whether the numbers are considered to be signed or unsigned. When two unsigned numbers are added, an overflow is detected from the end carry out of the most significant position. In the case of signed numbers, the leftmost bit always represents the sign, and negative numbers are in 2's complement form. When two signed numbers are added, the sign bit is treated as part of the number and the end carry does not indicate an overflow.

An overflow cannot occur after an addition if one number is positive and the other is negative, since adding a positive number to a negative number produces a result that is smaller than the larger of the two original numbers. An overflow may occur if the two numbers added are both positive or both negative. To see how this can happen, consider the following example. Two signed binary numbers, +70 and +80, are stored in two 8-bit registers. The range of numbers that each register can accommodate is from binary +127 to binary -128. Since the sum of the two

numbers is +150, it exceeds the capacity of the 8-bit register. This is true if the numbers are both positive or both negative. The two additions in binary are shown below together with the last two carries.

Carries: 0 1	+70 0 1000110	carries: 1 0	-70 1 0111010
	+80 0 1010000		-80 1 0110000
	+150 1 0010110		-150 0 1101010

Note that the 8-bit result that should have been positive has a negative sign bit and the 8-bit result that should have been negative has a positive sign bit. If, however, the carry out of the sign bit position is taken as the sign bit of the result, the 9-bit answer so obtained will be correct. Since the answer cannot be accommodated within 8 bits, we say that an overflow occurred.

An overflow condition can be detected by observing the carry into the sign bit position and the carry out of the sign bit position. If these two carries are not equal, an overflow condition is produced. This is indicated in the examples where the two carries are explicitly shown. If the two carries are applied to an exclusive-OR gate, an overflow will be detected when the output of the gate is equal to 1.

Decimal Fixed-Point Representation

The representation of decimal numbers in registers is a function of the binary code used to represent a decimal digit. A 4-bit decimal code requires four flip-flops for each decimal digit. The representation of 4385 in BCD requires 16 flip-flops, four flip-flops for each digit. The number will be represented in a register with 16 flip-flops as follows:

0100 0011 1000 0101

By representing numbers in decimal we are wasting a considerable amount of storage space since the number of bits needed to store a decimal number in a binary code is greater than the number of bits needed for its equivalent binary representation. Also, the circuits required to perform decimal arithmetic are more complex. However, there are some advantages in the use of decimal representation because computer input and output data are generated by people who use the decimal system. Some applications, such as business data processing, require small amounts of arithmetic computations compared to the amount required for input and output of decimal data. For this reason, some computers and all electronic calculators perform arithmetic operations directly with the decimal data (in a binary code) and thus eliminate the need for conversion to binary and back to decimal. Some computer systems have hardware for arithmetic calculations with both binary and decimal data.

The representation of signed decimal numbers in BCD is similar to the representation of signed numbers in binary. We can either use the familiar signed-magnitude system or the signed-complement system. The sign of a decimal number is usually represented with four bits to conform with the 4-bit code of the decimal digits. It is customary to designate a plus with four 0's and a minus with the BCD equivalent of 9, which is 1001.

The signed-magnitude system is difficult to use with computers. The signed-complement

system can be either the 9's or the 10's complement, but the 10's complement is the one most often used. To obtain the 10's complement of a BCD number, we first take the 9's complement and then add one to the least significant digit. The 9's complement is calculated from the subtraction of each digit from 9.

The procedures developed for the signed-2's complement system apply also to the signed-10's complement system for decimal numbers. Addition is done by adding all digits, including the sign digit, and discarding the end carry. Obviously, this assumes that all negative numbers are in 10's complement form. Consider the addition $(+375) + (-240) = +135$ done in the signed-10's complement system.

$$\begin{array}{r} 0\ 375 \quad (0000\ 0011\ 0111\ 0101)_{\text{BCD}} \\ 9\ 760 \quad (1001\ 0111\ 0110\ 0000)_{\text{BCD}} \\ \hline 0\ 135 \quad (0000\ 0001\ 0011\ 0101)_{\text{BCD}} \end{array}$$

The 9 in the leftmost position of the second number indicates that the number is negative. 9760 is the 10's complement of 0240. The two numbers are added and the end carry is discarded to obtain +135. Of course, the decimal numbers inside the computer must be in BCD, including the sign digits. The addition is done with BCD adders (see Fig. 10-18).

The subtraction of decimal numbers either unsigned or in the signed-10's complement system is the same as in the binary case. Take the 10's complement of the subtrahend and add it to the minuend. Many computers have special hardware to perform arithmetic calculations directly with decimal numbers in BCD. The user of the computer can specify by programmed instructions that the arithmetic operations be performed with decimal numbers directly without having to convert them to binary.

2-4 Floating-Point Representation

The floating-point representation of a number has two parts. The first part represents a signed, fixed-point number called the mantissa. The second part designates the position of the decimal (or binary) point and is called the exponent. The fixed-point mantissa may be a fraction or an integer. For example, the decimal number +6132.789 is represented in floating-point with a fraction and an exponent as follows:

Fraction	Exponent
+0.6132789	+4

The value of the exponent indicates that the actual position of the decimal point is four positions to the right of the indicated decimal point in the fraction. This representation is equivalent to the scientific notation $+0.6132789 \times 10^{-4}$.

Floating-point is always interpreted to represent a number in the following form:

$$m \times r^e$$

Only the mantissa m and the exponent e are physically represented in the register (including their signs). The radix r and the radix-point position of the mantissa are always assumed. The circuits that manipulate the floating-point numbers in registers conform with these two assumptions in order to provide the correct computational results.

A floating-point binary number is represented in a similar manner except that it uses base 2

for the exponent. For example, the binary number +1001.11 is represented with an 8-bit fraction and b-bit exponent as follows:

<i>Fraction</i>	<i>Exponent</i>
01001110	000100

The fraction has a 0 in the leftmost position to denote positive. The binary point of the fraction follows the sign bit but is not shown in the register. The exponent has the equivalent binary number +4. The floating-point number is equivalent to

$$m \times 2^e = +(.1001110)_2 \times 2^{+4}$$

A floating-point number is said to be *normalized* if the most significant digit of the mantissa is nonzero. For example, the decimal number 350 is normalized but 00035 is not. Regardless of where the position of the radix point is assumed to be in the mantissa, the number is normalized only if its leftmost digit is nonzero. For example, the 8-bit binary number 00011010 is not normalized because of the three leading 0's. The number can be normalized by shifting it three positions to the left and discarding the leading 0's to obtain 11010000. The three shifts multiply the number by $2^3=8$. To keep the same value for the floating-point number, the exponent must be subtracted by 3. Normalized numbers provide the maximum possible precision for the floating-point number. A zero cannot be normalized because it does not have a nonzero digit. It is usually represented in floating-point by all 0's in the mantissa and exponent.

Arithmetic operations with floating-point numbers are more complicated than arithmetic operations with fixed-point numbers and their execution takes longer and requires more complex hardware. However, floating-point representation is a must for scientific computations because of the scaling problems involved with fixed-point computations. Many computers and all electronic calculators have the built-in capability of performing floating-point computations have a set of subroutines to help the user program scientific problems with floating-point numbers. Arithmetic operations with floating-point numbers are discussed in Sec. 10-5.

2-5 Other Binary Codes

In previous sections we introduced the most common type of binary-coded data found in digital computers. Other binary codes for decimal numbers and alphanumeric character are sometimes used. Digital computers also employ other binary codes for special applications. A few additional binary codes encountered in digital computers are presented in this section.

Gray Code

Digital systems can process data in discrete form only. Many physical systems supply continuous output data. The data must be converted into digital form before they can be used by a digital computer. Continuous, or analog, information is converted into digital form by means of an analog-to-digital converter. The reflected binary or *Gray code*, shown in Table 2-5, is sometimes used for the converted digital data. The advantage of the Gray code over straight binary numbers is that the Gray code changes by only one bit as it sequences from one number to the next. In other words, the change from any number to the next in sequence is recognized by a change of only one bit from 0 to 1 or from 1 to 0. A typical application of the Gray code occurs when the analog data are represented by the continuous change of a shaft position. The shaft is partitioned

into segments with each segment assigned a number. If adjacent segments are made to correspond to adjacent Gray code numbers, ambiguity is reduced when the shaft position is in the line that separates any two segments.

Gray code counters are sometimes used to provide the timing sequences that control the operations in a digital system. A Gray code counter is a counter whose flip-flops go through a sequence of states as specified in Table 2-5. Gray code counters remove the ambiguity during the change from one state of the counter to the next because only one bit can change during the state transition.

TABLE 2-5 4-Bit Gray Code

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

Other Decimal Codes

Binary codes for decimal digits require a minimum of four bits. Numerous different codes can be formulated by arranging four or more bits in 10 distinct possible combinations. A few possibilities are shown in Table 2-6.

TABLE 2-6 Four Different Binary Codes for the Decimal Digit

Decimal digit	BCD 8421	2421	Excess-3	Excess-3 gray
0	0000	0000	0011	0010
1	0001	0001	0100	0110
2	0010	0010	0101	0111
3	0011	0011	0110	0101
4	0100	0100	0111	0100
5	0101	1011	1000	1100
6	0110	1100	1001	1101
7	0111	1101	1010	1111
8	1000	1110	1011	1110
9	1001	1111	1100	1010
Unused bit combinations	1010	0101	0000	0000
	1011	0110	0001	0001
	1100	0111	0010	0011
	1101	1000	1101	1000
	1110	1001	1110	1001
	1111	1010	1111	1011

The BCD (binary-coded decimal) has been introduced before. It uses a straight assignment of

the binary equivalent of the digit. The six unused bit combinations listed have no meaning when BCD is used, just as the letter H has no meaning when decimal digit symbols are written down. For example, saying that 1001 1110 is a decimal number in BCD is like saying that 9H is a decimal number in the conventional symbol designation. Both cases contain an invalid symbol and therefore designate a meaningless number.

One disadvantage of using BCD is the difficulty encountered when the 9's complement of the number is to be computed. On the other hand, the 9's complement is easily obtained with the 2421 and the excess-3 codes listed in Table 2-6. These two codes have a self-complementing property which means that the 9's complement of a decimal number, when represented in one of these codes, is easily obtained by changing 1's to 0's and 0's to 1's. This property is useful when arithmetic operations are done in signed-complement representation.

The 2421 is an example of a weighted code. In a *weighted code*, the bits are multiplied by the weights indicated and the sum of the weighted bits gives the decimal digit. For example, the bit combination 1101, when weighted by the respective digits 2421, gives the decimal equivalent of $2 \times 1 + 4 \times 1 + 2 \times 0 + 1 \times 1 = 7$. The BCD code can be assigned the weights 8421 and for this reason it is sometimes called the 8421 code.

The excess-3 code is a decimal code that has been used in older computers. This is an unweighted code. Its binary code assignment is obtained from the corresponding BCD equivalent binary number after the addition of binary 3 (0011).

From Table 2-5 we note that the Gray code is not suited for a decimal code if we were to choose the first 10 entries in the table. This is because the transition from 9 back to 0 involves a change of three bits (from 1101 to 0000). To overcome this difficulty, we choose the 10 numbers starting from the third entry 0010 up to the twelfth entry 1010. Now the transition from 1010 to 0010 involves a change of only one bit. Since the code has been shifted up three numbers, it is called the excess-3 Gray. This code is listed with the other decimal codes in Table 2-6.

Other Alphanumeric Codes

The ASCII code (Table 2-4) is the standard code commonly used for the transmission of binary information. Each character is represented by a 7-bit code and usually an eighth bit is inserted for parity (see Sec.2-6). The code consists of 128 characters. Ninety-five characters represent *graphic symbols* that include upper- and lowercase letters, numerals zero to nine, punctuation marks, and special symbols. Twenty-three characters represent *format effectors*, which are functional characters for controlling the layout of printing or display devices such as carriage return, line feed, horizontal tabulation, and back space. The other 10 characters are used to direct the data communication flow and report its status.

Another alphanumeric (sometimes called *alphameric*) code used in IBM equipment is the EBCDIC (Extended BCD Interchange Code). It uses eight bits for each character (and a ninth bit for parity). EBCDIC has the same character symbols as ASCII but the bit assignment to characters is different.

When alphanumeric characters are used internally in a computer for data processing (not for transmission purposes) it is more convenient to use a 6-bit code to represent 64 characters. A 6-bit code can specify the 26 uppercase letters of the alphabet, numerals zero to nine, and up to 28

special characters. This set of characters is usually sufficient for data-processing purposes. Using fewer bits to code characters has the advantage of reducing the memory space needed to store large quantities of alphanumeric data.

2-6 Error Detection Codes

Binary information transmitted through some form of communication medium is subject to external noise that could change bits from 1 to 0, and vice versa. An error detection code is a binary code that detects digital errors during transmission. The detected errors cannot be corrected but their presence is indicated. The usual procedure is to observe the frequency of errors. If errors occur infrequently at random, the particular erroneous information is transmitted again. If the error occurs too often, the system is checked for malfunction.

The most common error detection code used is the parity bit. A *parity bit* is an extra bit included with a binary message to make the total number of 1's either odd or even. A message of three bits and two possible parity bits is shown in Table 2-7. The $P(\text{odd})$ bit is chosen in such a way as to make the sum of 1's (in all four bits) odd. The $P(\text{even})$ bit is chosen to make the sum of all 1's even. In either case, the sum is taken over the message and the P bit. In any particular application, one or the other type of parity will be adopted. The even-parity scheme has the disadvantage of having a bit combination of all 0's, while in the odd parity there is always one bit (of the four bits that constitute the message and P) that is 1. Note that the $P(\text{odd})$ is the complement of the $P(\text{even})$.

During transfer of information from one location to another, the parity bit is handled as follows. At the sending end, the message (in this case three bits) is applied to a *parity generator*, where the required parity bit is generated. The message, including the parity bit, is transmitted to its destination. At the receiving end, all the incoming bits (in this case, four) are applied to a *parity checker* that checks the proper parity adopted (odd or even). An error is detected if the checked parity does not conform to the adopted parity. The parity method detects the presence of one, three, or any odd number of errors. An even number of errors is not detected.

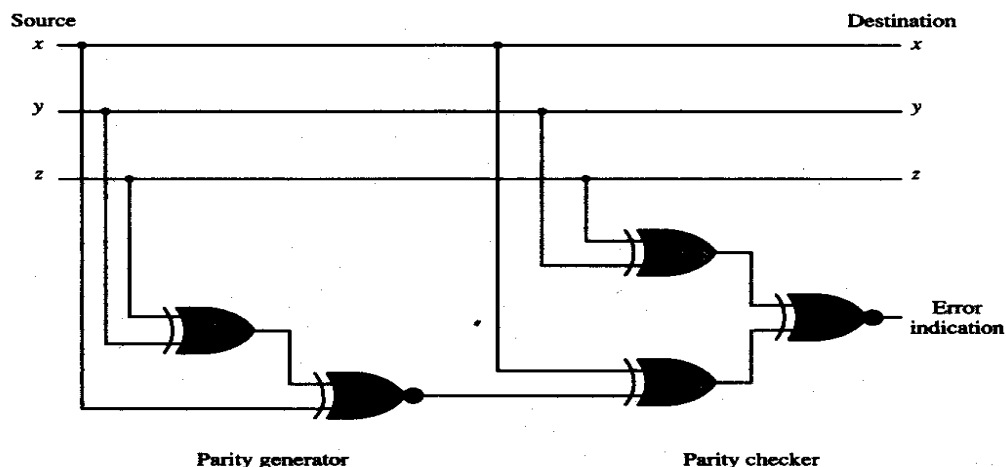
TABLE 2-7 Parity Bit Generation

Message <i>xyz</i>	$P(\text{odd})$	$P(\text{even})$
000	1	0
001	0	1
010	0	1
011	1	0
100	0	1
101	1	0
110	1	0
111	0	1

Parity generator and checker networks are logic circuits constructed with exclusive-OR functions. This is because, as mentioned in Sec. 1-2, the exclusive-OR function of three or more variables is by definition an odd function. An odd function is a logic function whose value is binary 1 if, and only if, an odd number of variables are equal to 1. According to this definition, the $P(\text{even})$ function is the exclusive-OR of x , y , and z because it is equal to 1 when either one or all

three of the variables are equal to 1 (Table 2-7). The $P(\text{odd})$ function is the complement of the $P(\text{even})$ function.

Figure 2-3 Error detection with odd parity bit.



As an example, consider a 3-bit message to be transmitted with an odd parity bit. At the sending end, the odd-parity bit is generated by a parity generator circuit. As shown in Fig.2-3, this circuit consists of one exclusive-OR and one exclusive-NOR gate. Since $P(\text{even})$ is the exclusive-OR of x , y , z , and $P(\text{odd})$ is the complement of $P(\text{even})$, it is necessary to employ an exclusive-NOR gate for the needed complementation. The message and the odd-parity bit are transmitted to their destination where they are applied to a parity checker. An error has occurred during transmission if the parity of the four bits received is even, since the binary information transmitted was originally odd. The output of the parity checker would be 1 when an error occurs, that is, when the number of 1's in the four inputs is even. Since the exclusive-OR function of the four inputs is an odd function, we again need to complement the output by using an exclusive-NOR gate.

It is worth noting that the parity generator can use the same circuit as the parity checker if the fourth input is permanently held at a logic-0 value. The advantage of this is that the same circuit can be used for both parity generation and parity checking.

It is evident from the example above that even-parity generators and checkers can be implemented with exclusive-OR functions. Odd-parity networks need an exclusive-NOR at the output to complement the function.

PROBLEMS

- 2-1. Convert the following binary numbers to decimal: 101110; 1110101; and 110110100.
- 2-2. Convert the following numbers with the indicated bases to decimal: $(12121)_3$; $(4310)_5$; $(50)_7$; and 9198_{12} .
- 2-3. Convert the following decimal numbers to binary: 1231; 673; and 1998.
- 2-4. Convert the following decimal numbers to the bases indicated.
 - a. 7562 to octal

- b. 1938 to hexadecimal
 - c. 175 to binary
- 2-5. Convert the hexadecimal number F3A7C2 to binary and octal.
- 2-6. What is the radix of the numbers if the solution to the quadratic equation $x^2 - 10x + 31 = 0$ is $x=5$ and $x=8$?
- 2-7. Show the value of all bits of a 12-bit register that hold the number equivalent to decimal 215 in (a) binary; (b) binary-coded octal; (c) binary-coded hexadecimal; (d) binary-coded decimal (BCD).
- 2-8. Show the bit configuration of a 24-bit register when its content represents the decimal equivalent of 295: (a) in binary; (b) in BCD; (c) in ASCII using eight bits with even parity.
- 2-9. Write your name in ASCII using an 8-bit code with the leftmost bit always 0. Include a space between names and a period after a middle initial.
- 2-10. Decode the following ASCII code:
1001010 1001111 1001000 1001110 0100000 1000100 1001111 100101
- 2-11. Obtain the 9's complement of the following eight-digit decimal numbers: 12349876; 00980100; 90009951; and 00000000.
- 2-12. Obtain the 10's complement of the following six-digit decimal numbers: 123900; 090657; 100000; and 000000.
- 2-13. Obtain the 1's and 2's complements of the following eight-digit binary numbers: 10101110; 10000001; 1000000; 0000001; and 0000000.
- 2-14. Perform the subtraction with the following unsigned decimal numbers by taking the 10's complement of the subtrahend.
- a. $5250 - 1321$
 - b. $1753 - 8640$
 - c. $20 - 100$
 - d. $1200 - 250$
- 2-15. Perform the subtraction with the following unsigned binary numbers by taking the 2's complement of the subtrahend.
- a. $11010 - 10000$
 - b. $11010 - 1101$
 - c. $100 - 110000$
 - d. $1010100 - 1010100$
- 2-16. Perform the arithmetic operations $(+42) + (-13)$ and $(-42) - (-13)$ in binary using signed-2's complement representation for negative numbers.
- 2-17. Perform the arithmetic operations $(+70) + (+80)$ and $(-70) + (-80)$ with binary numbers in signed-2's complement representation. Use eight bits to accommodate each number together with its sign. Show that overflow occurs in both cases, that the last two carries are unequal, and that there is a sign reversal.
- 2-18. Perform the following arithmetic operations with the decimal numbers using signed-10's complement representation for negative numbers.
- a. $(-638) + (+785)$
 - b. $(-638) - (+185)$
- 2-19. A 36-bit floating-point binary number has eight bits plus sign for the exponent and 26 bits plus sign for the mantissa. The mantissa is a normalized fraction. Numbers in the mantissa and exponent are in signed-magnitude representation. What are the largest and smallest positive quantities that can be represented, excluding zero?

- 2-20. Represent the number $(+46.5)_{10}$ as a floating-point binary number with 24 bits. The normalized fraction mantissa has 16 bits and the exponent has 8 bits.
- 2-21. The Gray code is sometimes called a reflected code because the bit values are reflected on both sides of any 2^n value. For example, as shown in Table 2-5, the values of the three low-order bits are reflected over a line drawn between 7 and 8. Using this property of the Gray code, obtain:
- The Gray code numbers for 16 through 31 as a continuation of Table 2-5.
 - The excess-3 Gray code for decimals 10 to 19 as a continuation of the list in Table 2-6.
- 2-22. Represent decimal number 8620 in (a) BCD; (b) excess-3 code; (c) 2421 code; (d) as a binary number.
- 2-23. List the 10 BCD digits with an even parity in the leftmost position. (total of five bits per digit). Repeat with an odd-parity bit.
- 2-24. Represent decimal 3984 in the 2421 code of Table 2-6. Complement all bits of the coded number and show that the result is the 9's complement of 3984 in the 2421 code.
- 2-25. Show that the exclusive-OR function $x = A \oplus B \oplus C \oplus D$ is an odd function. One way to show this is to obtain the truth table for $y = A \oplus B$ and for $z = C \oplus D$ and then formulate the truth table for $x = y \oplus z$. Show that $x = 1$ only when the total number of 1's in A, B, C, and D is odd.
- 2-26. Derive the circuits for a 3-bit parity generator and 4-bit parity checker using an even-parity bit. (The circuits of Fig. 3-3 use odd parity.)

CHARTER FOUR**Register Transfer and Microoperations**

(选自 M. MORRIS MANO 《COMPUTER SYSTEM ARCHITECTURE》THIRD EDITION)

IN THIS CHAPTER

- 4-1 Register Transfer Language
- 4-2 Register Transfer
- 4-3 Bus and Memory Transfers
- 4-4 Arithmetic Microoperations
- 4-5 Logic Microoperations
- 4-6 Shift Microoperations
- 4-7 Arithmetic Logic Shift Unit

4-1 Register Transfer Language

A digital system is an interconnection of digital hardware modules that accomplish a specific information-processing task. Digital systems vary in size and complexity from a few integrated circuits to a complex of interconnected and interacting digital computers. Digital system design invariably uses a modular approach. The modules are constructed from such digital components as registers, decoders, arithmetic elements, and control logic. The various modules are interconnected with common data and control paths to form a digital computer system.

Digital modules are best defined by the registers they contain and the operations that are performed on the data stored in them. The operations executed on data stored in registers are called microoperations. A microoperation is an elementary operation performed on the information stored in one or more registers. The result of the operation may replace the previous binary information of a register or may be transferred to another register. Examples of microoperations are shift, count, clear, and load. Some of the digital components introduced in Chap. 2 are registers that implement microoperations. For example, a counter with parallel load is capable of performing the microoperations increment and load. A bidirectional shift register is capable of performing the shift right and shift left microoperations.

The internal hardware organization of a digital computer is best defined by specifying:

1. The set of registers it contains and their function.
2. The sequence of microoperations performed on the binary information stored in the registers.
3. The control that initiates the sequence of microoperations.

It is possible to specify the sequence of microoperations in a computer by explaining every operation in words, but this procedure usually involves a lengthy descriptive explanation. It is more convenient to adopt a suitable symbology to describe the sequence of transfers between registers and the various arithmetic and logic microoperations associated with the transfers. The

use of symbols instead of a narrative explanation provides an organized and concise manner for listing the microoperation sequences in registers and the control functions that initiate them.

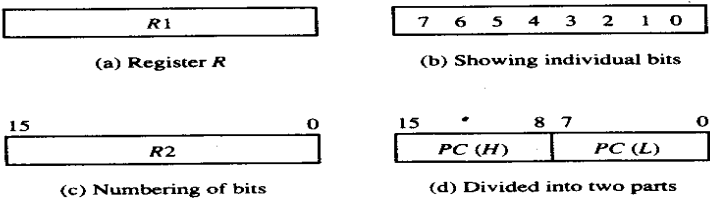
The symbolic notation used to describe the microoperation transfers among registers is called a register transfer language. The term “register transfer” implies the availability of hardware logic circuits that can perform a stated microoperation and transfer the result of the operation to the same or another register. The word “language” is borrowed from programmers, who apply this term to programming languages. A programming language is a procedure for writing symbols to specify a given computational process. Similarly, a natural language such as English is a system for writing symbols and combining them into words and sentences for the purpose of communication between people. A register transfer language is a system for expressing in symbolic form the microoperation sequences among the registers of a digital module. It is a convenient tools for describing the internal organization of digital computers in concise and precise manner. It can also be used to facilitate the design process of digital systems.

The register transfer language adopted here is believed to be as simple as possible, so it should not take very long to memorize. We will proceed to define symbols for various types of microoperations, and at the same time, describe associated hardware that can implement the stated microoperations. The symbolic designation introduced in this chapter will be utilized in subsequent chapters to specify the register transfers, the microoperations, and the control functions that describe the internal hardware organization of digital computers. Other symbology in use can easily be learned once this language has become familiar, for most of the differences between register transfer languages consist of variations in detail rather than in overall purpose.

4-2 Register Transfer

Computer registers are designated by capital letters (sometimes followed by numerals) to denote the function of the register. For example, the register that holds an address for the memory unit is usually called a memory address register and is designated by the name *MAR*. Other designations for registers are *PC*(for program counter), *IR*(for instruction register, and *RI*(for processor register). The individual flip-flops in an n -bit register are numbered in sequence from – through $n-1$, starting from 0 in the rightmost position and increasing the numbers toward the left. Figure 4-1 shows the representation of registers in block diagram form. The most common way to represent a register is by a rectangular box with the name of the register inside, as in Fig. 4-1(a). The individual bits can be distinguished as in (b). The numbering of bits in a 16-bit register can be marked on top of the box as shown in (c). A 16-bit register is partitioned into two parts in (d). Bits 0 through 7 are assigned the symbol *L*(for low byte) and bits 8 through 15 are assigned the symbol *H*(for high byte). The name of the 16-bit register is *PC*. The symbol *PC*(0-7) or *PC*(*L*) refers to the low-order byte and *PC*(8-15) or *PC*(*H*) to the high-order byte.

Figure 4-1 Block diagram of register.



Information transfer from one register to another is designate in symbolic form by means of a replacement operator. The statement

$$R2 \leftarrow R1$$

Denotes a transfer of the content of register $R1$ into register $R2$. It designates a replacement of the content of $R2$ by the content of $R1$. By definition, the content of the source register $R1$ does not change after the transfer.

A statement that specifies a register transfer implies that circuits are available from the outputs of the source register to the inputs of the destination register and that the destination register has a parallel load capability. Normally, we want the transfer to occur only under a predetermined control condition. This can be shown by means of an if-then statement.

$$\text{If } (P=1) \text{ then } (R2 \leftarrow R1)$$

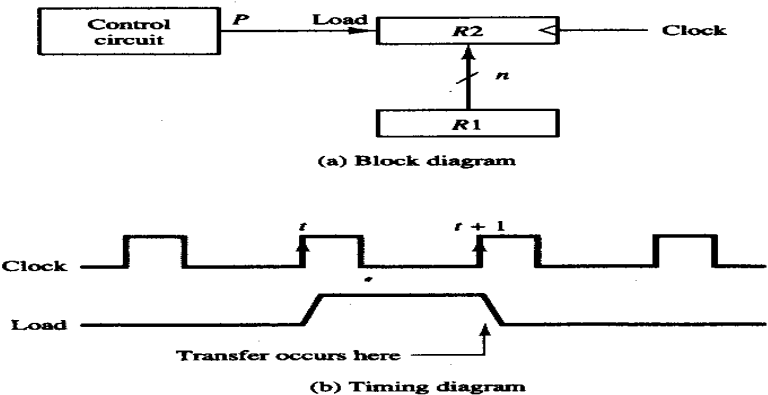
where P is a control signal generated in the control section. It is sometimes convenient to separate the control variables from the register transfer operation by specifying a control function. A control function is a Boolean variable that is equal to 1 or 0. The control function is included in the statement as follows:

$$P: R2 \leftarrow R1$$

The control condition is terminated with a colon. It symbolizes the requirement that the transfer operation be executed by the hardware only if $P=1$.

Every statement written in a register transfer notation implies a hardware construction for implementing the transfer. Figure 4-2 shows the block diagram that depicts the transfer from $R1$ to $R2$. The n outputs of register $R1$ are connected to the n inputs of register $R2$. The letter n will be used to indicate any number of bits for the register. It will be replaced by an actual number when the length of the register is known. Register $R2$ has a load input that is activated by the control variable P . It is assumed that the control variable is synchronized with the same clock as the one applied to the register. As shown in the timing diagram, P is activate in the control section by the rising edge of a clock pulse at time t . The next positive transition of the clock at time $t+1$ finds the load input active and the data inputs of $R2$ are then loaded into the register in parallel. P may go back to 0 at time $t+1$; otherwise, the transfer will occur with every clock pulse transition while P remains active.

Figure 4-2 Transfer from $R1$ to $R2$ when $P = 1$.



Note that the clock is not included as a variable in the register transfer statements. It is assumed that all transfers occur during a clock edge transition. Even though the control condition such as P becomes active just after time t , the actual transfer does not occur until the register is triggered by the next positive transition of the clock at time $t+1$.

The basic symbols of the register transfer notation are listed in Table 4-1. Registers are denoted by capital letters, and numerals may follow the letters. Parentheses are used to denote a part of a register by specifying the range of bits or by giving a symbol name to a portion of a register. The arrow denotes a transfer of information and the direction of transfer. A comma is used to separate two or more operations that are executed at the same time. The statement

$$T: R2 \leftarrow R1, R1 \leftarrow R2$$

denotes an operation that exchanges the contents of two registers during one common clock pulse provided that $T=1$. This simultaneous operation is possible with registers that have edge-triggered flip-flops.

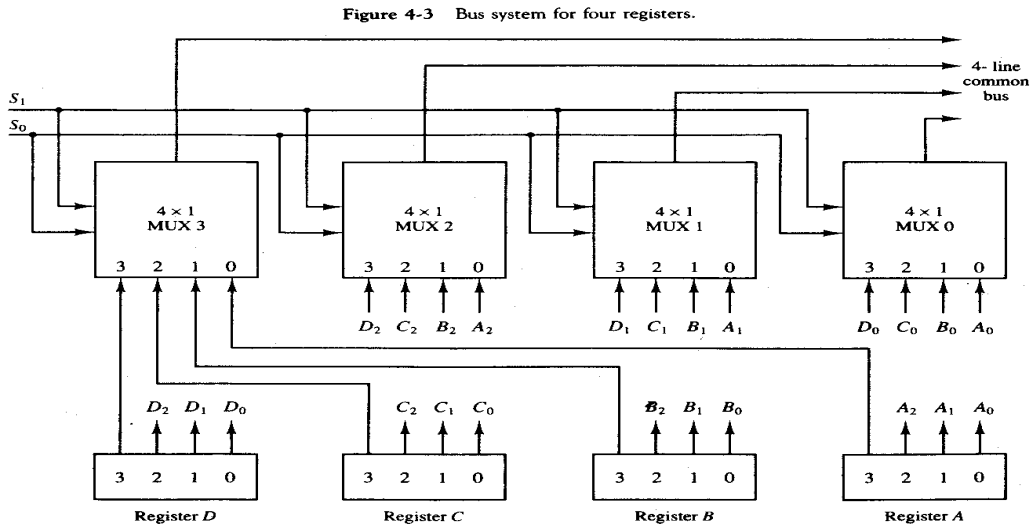
TABLE 4-1 Basic Symbols for Register Transfers

Symbol	Description	Examples
Letters (and numerals)	Denotes a register	<i>MAR, R2</i>
Parentheses ()	Denotes a part of a register	<i>R2(0-7), R2(L)</i>
Arrow \leftarrow	Denotes transfer of information	<i>R2 \leftarrow R1</i>
Comma ,	Separates two microoperations	<i>R2 \leftarrow R1, R1 \leftarrow R2</i>

4-3 Bus and Memory Transfers

A typical digital computer has many registers, and paths must be provided to transfer information from one register to another. The number of wires will be excessive if separate lines are used between each register and all other registers in the system. A more efficient scheme for transferring information between registers in a multiple-register configuration is a common bus system. A bus structure consists of a set of common lines, one for each bit of a register, through which binary information is transferred one at a time. Control signals determine which register is selected by the bus during each particular register transfer.

One way of constructing a common bus system is with multiplexers. The multiplexers select the source register whose binary information is then placed on the bus. The construction of a bus system for four registers is shown in Fig. 4-3. Each register has four bits, numbered 0 through 3. The bus consists of four 4×1 multiplexers each having four data inputs, 0 through 3, and two selection inputs, S_1 and S_0 . In order not to complicate the diagram with 16 lines crossing each other, we use labels to show the connections from the outputs of the registers to the inputs of the multiplexers. For example, output 1 of register A is connected to input 0 of MUX 1 because this input is labeled A_1 . The diagram shows that the bits in the same significant position in each register are connected to the data inputs of one multiplexer to form one line of the bus. Thus MUX 0 multiplexes the four 0 bits of the registers, MUX 1 multiplexes the four 1 bits of the registers, and similarly for the other two bits.



The two selection lines S_1 and S_0 are connected to the selection inputs of all four multiplexers. The selection lines choose the four bits of one register and transfer them into the four-line common bus. When $S_1S_0 = 00$, the 0 data inputs of all four multiplexers are selected and applied to the outputs that form the bus. This causes the bus lines to receive the content of register A since the outputs of this register are connected to the 0 data inputs of the multiplexers. Similarly, register B is selected if $S_1S_0 = 01$, and so on. Table 4-2 shows the register that is selected by the bus for each of the four possible binary value of the selection lines.

TABLE 4-2 Function Table for Bus of Fig. 4-3

S_1	S_0	Register selected
0	0	A
0	1	B
1	0	C
1	1	D

In general, a bus system will multiplex k registers of n bits each to produce an n -line common bus. The number of multiplexers needed to construct the bus is equal to n , the number of bits in each register. The size of each multiplexer must be $k \times 1$ since it multiplexes k data lines. For example, a common bus for eight registers of 16 bits each requires 16 multiplexers, one for each line in the bus. Each multiplexer must have eight data input lines and three selection lines to multiplex one significant bit in the eight registers.

The transfer of information from a bus into one of many destination registers can be accomplished by connecting the bus lines to the inputs of all destination registers and activating the load control of the particular destination register selected. The symbolic statement for a bus transfer may mention the bus or its presence may be implied in the statement. When the bus is included in the statement, the register transfer is symbolized as follows:

$$BUS \leftarrow C, \quad RI \leftarrow BUS$$

The content of register C is placed on the bus, and the content of the bus is loaded into register RI by activating its load control input. If the bus is known to exist in the system, it may be convenient just to show the direct transfer.

$$RI \leftarrow C$$

From this statement the designer knows which control signals must be activated to produce the transfer through the bus.

Three-State Bus Buffers

A bus system can be constructed with three-state gates instead of multiplexers. A three-state gate is a digital circuit that exhibits three states. Two of the states are signals equivalent to logic 1 and 0 as in a conventional gate. The third state is a high-impedance state. The high-impedance state behaves like an open circuit, which means that the output is disconnected and does not have a logic significance. Three-state gates may perform any conventional logic, such as AND or NAND. However, the one most commonly used in the design of a bus system is the buffer gate.

The graphic symbol of a three-state buffer gate is shown in fig. 4-4. It is distinguished from a normal buffer by having both a normal input and a control input. The control input determines the output state. When the control input is equal to 1, the output is enabled and the gate behaves like any conventional buffer, with the output equal to the normal input. When the control input is 0, the output is disabled and the gate goes to a high-impedance state of a three-state gate provides a special feature not available in other gates. Because of this feature, a large number of three-state gate outputs can be connected with wires to form a common bus line without endangering loading effects.

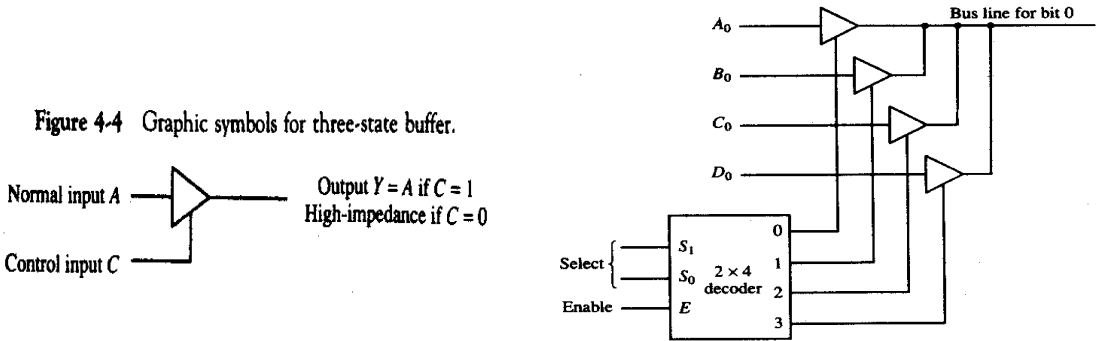


Figure 4-5 Bus line with three state-buffers.

The construction of a bus system with three-state buffers is demonstrated in Fig. 4-5. The outputs of four buffers are connected together to form a single bus line. (It must be realized that this type of connection cannot be done with gates that do not have three-state outputs.) The control inputs to the buffers determine which of the four normal inputs will communicate with the bus line. No more than one buffer may be in the active state at any given time. The connected buffers must be controlled so that only one three-state buffer has access to the bus line while all other buffers are maintained in a high-impedance state.

One way to ensure that no more than one control input is active at any given time is to use a decoder, as shown in the diagram. When the enable input of the decoder is 0, all of its four outputs

are 0, and the bus line is in a high-impedance state because all four buffers are disabled. When the enable input is active, one of the three-state buffers will be active, depending on the binary value in the select inputs of the decoder. Careful investigation will reveal that Fig. 4-5 is another way of constructing a 4×1 multiplexer since the circuit can replace the multiplexer in Fig. 4-3.

To construct a common bus for four registers of n bits each using three-state buffers, we need n circuits with four buffers in each as shown in Fig. 4-5. Each group of four buffers receives one significant bit from the four registers. Each common output produces one of the lines for the common bus for a total of n lines. Only one decoder is necessary to select between the four registers.

Memory Transfer

The operation of a memory unit was described in Sec. 2-7. The transfer of information from a memory word to the outside environment is called a read operation. The transfer of new information to be stored into the memory is called a write operation. A memory word will be symbolized by the letter M . The particular "memory word among the many available is selected by the memory address during the transfer. It is necessary to specify the address of M when writing memory transfer operations. This will be done by enclosing the address in square brackets following the letter M .

Consider a memory unit that receives the address from a register; called the address register, symbolized by AR . The data are transferred to another register, called the data register, symbolized by DR . The read operation can be stated as follows:

Read: $DR \leftarrow M[AR]$

This causes a transfer of information into DR from the memory word M selected by the address in AR .

The write operation transfers the content of a data register to a memory word M selected by the address. Assume that the input data are in register RI and the address is in AR . The write operation can be stated symbolically as follows:

Write: $M[AR] \leftarrow RI$

This causes a transfer of information from RI into the memory word M selected by the address in AR .

4-4 Arithmetic Microoperations

A microoperation is an elementary operation performed with the data stored in registers. The microoperations most often encountered in digital computers are classified into four categories:

1. Register transfer microoperations transfer binary information from one register to another.
2. Arithmetic microoperations perform arithmetic operations on numeric data stored in registers.
3. Logic microoperations perform bit manipulation operations on non-numeric data stored in registers.
4. Shift microoperations perform shift operations on data stored in registers.

The register transfer microoperation was introduced in Sec. 4-2. This type of microoperation does not change the information content when the binary information moves from the source

register to the destination register. The other three types of microoperations change the information content during the transfer. In this section we introduce a set of arithmetic microoperations. In the next two sections we present the logic and shift microoperations.

The basic arithmetic microoperations are addition, subtraction, increment, decrement, and shift. Arithmetic shifts are explained later in conjunction with the shift microoperations. The arithmetic microoperation defined by the statement

$$R3 \leftarrow R1 + R2$$

Specifies an add microoperation. It states that the contents of register *R1* are added to the contents of register *R2* and the sum transferred to register *R3*. To implement this statement with hardware we need three registers and the digital component that performs the addition operation. The other basic arithmetic microoperations are listed in Table 4-3. Subtraction is most often implemented through complementation and addition. Instead of using the minus operator, we can specify the subtraction by the following statement:

$$R3 \leftarrow R1 + /R2 + 1$$

/R2 is the symbol for the 1's complement of *R2*. Adding 1 to the 1's complement produces the 2's complement. Adding the contents of *R1* to the 2's complement of *R2* is equivalent to *R1*−*R2*.

TABLE 4-3 Arithmetic Microoperations

Symbolic designation	Description
$R3 \leftarrow R1 + R2$	Contents of <i>R1</i> plus <i>R2</i> transferred to <i>R3</i>
$R3 \leftarrow R1 - R2$	Contents of <i>R1</i> minus <i>R2</i> transferred to <i>R3</i>
$R2 \leftarrow \overline{R2}$	Complement the contents of <i>R2</i> (1's complement)
$R2 \leftarrow \overline{R2} + 1$	2's complement the contents of <i>R2</i> (negate)
$R3 \leftarrow R1 + \overline{R2} + 1$	<i>R1</i> plus the 2's complement of <i>R2</i> (subtraction)
$R1 \leftarrow R1 + 1$	Increment the contents of <i>R1</i> by one
$R1 \leftarrow R1 - 1$	Decrement the contents of <i>R1</i> by one

The increment and decrement microoperations are symbolized by plus-one and minus-one operations, respectively. These microoperations are implemented with a combinational circuit or with a binary up-down counter.

The arithmetic operations of multiply and divide are not listed in Table 4-3. These two operations are valid arithmetic operations but are not included in the basic set of microoperations. The only place where these operations can be considered as microoperations is in a digital system, where they are implemented by means of a combinational circuit. In such a case, the signals that perform these operations propagate through gates, and the result of the operation can be transferred into a destination register by a clock pulse as soon as the output signal propagates through the combinational circuit. In most computers, the multiplication operation is implemented with a sequence of add and shift microoperations. Division is implemented with a sequence of subtract and shift microoperations. To specify the hardware in such a case requires a list of statements that use the basic microoperations of add, subtract, and shift (see Chapter 10).

Binary Adder

To implement the add microoperation with hardware, we need the registers that hold the data and the digital component that performs the arithmetic addition. The digital circuit that forms the arithmetic sum of two bits and a previous carry is called a full-adder (see Fig. 1-17). The digital

circuit that generates the arithmetic sum of two binary numbers of any length is called a binary adder. The binary adder is constructed with full-adder circuits connected in cascade, with the output carry from one full-adder connected to the input carry of the next full-adder. Figure 4-6 shows the interconnections of four full-adders (FA) to provide a 4-bit binary adder. The augend bits of A and the addend bits of B are designated by subscript numbers from right to left, with subscript 0 denoting the low-order bit. The carries are connected in a chain through the full-adders. The input carry to the binary adder is C_0 and the output carry is C_4 . The S outputs of the full-adders generate the required sum bits.

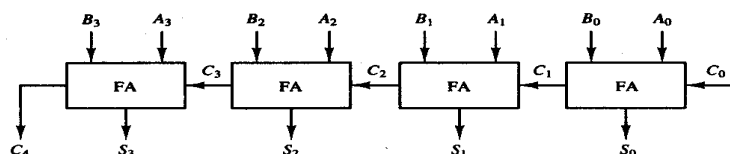


Figure 4-6 4-bit binary adder.

An n -bit binary adder requires n full-adders. The output carry from each full-adder is connected to the input carry of the next-high-order full-adder. The n data bits for the A inputs come from one register (such as $R1$), and the n data bits for the B inputs come from another register (such as $R2$). The sum can be transferred to a third register or to one of the source registers ($R1$ or $R2$), replacing its previous content.

Binary Adder-Subtractor

The subtraction of binary numbers can be done most conveniently by means of complements as discussed in Sec. 3-2. Remember that the subtraction $A-B$ can be done by taking the 2's complement of B and adding it to A . The 2's complement can be obtained by taking the 1's complement and adding one to the least significant pair of bits. The 1's complement can be implemented with inverters and a one can be added to the sum through the input carry.

The addition and subtraction operations can be combined into one common circuit by including an exclusive-Or gate with each full-adder. A 4-bit adder-subtractor circuit is shown in Fig. 4-7. The mode input M controls the operation. When $M=0$ the circuit is an adder and when $M=1$ the circuit becomes a subtractor. Each exclusive-Or gate receives input M and one of the inputs of B . When $M=0$, we have $B \oplus 0 = B$ and $C_0=1$. The B inputs are all complemented and a 1 is added through the input carry. The circuit performs the operation A plus the 2's complement of B . For unsigned numbers, this gives $A-B$ if $A \geq B$ or the 2's complement of $(B-A)$ if $A < B$. For signed numbers, the result is $A-B$ provided that there is no overflow.

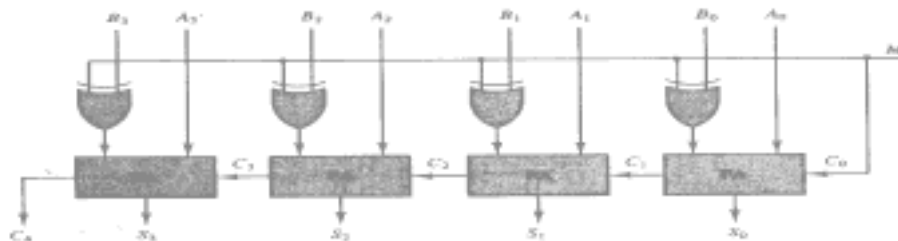


Figure 4-7 4-bit adder-subtractor.

Binary Incrementer

The increment microoperation adds one to a number in a register. For example, if a 4-bit register has a binary value 0110, it will go to 0111 after it is incremented. This microoperation is easily implemented with a binary counter (see Fig. 2-10). Every time the count enable is active, the clock pulse transition increments the content of the register by one. There may be occasions when the increment microoperation must be done with a combinational circuit independent of a particular register. This can be accomplished by means of half-adders (see Fig. 1-16) connected in cascade.

The diagram of a 4-bit combinational circuit incrementer is shown in Fig. 4-8. One of the inputs to the least significant half-adder (HA) is connected to logic-1 and the other input is connected to the least significant bit of the number to be incremented. The output carry from one half-adder is connected to one of the inputs of the next-higher-order half-adder. The circuit receives the four bits from A_0 through A_3 , adds one to it, and generates the incremented output in 0 through S_3 . The output carry C_4 will be 1 only after incrementing binary 1111. This also causes outputs S_0 through S_3 to go to 0.

The circuit of Fig. 4-8 can be extended to an n -bit binary incrementer by extending the diagram to include n half-adders. The least significant bit must have one input connected to logic-1. The other inputs receive the number to be incremented or the carry from the previous stage.

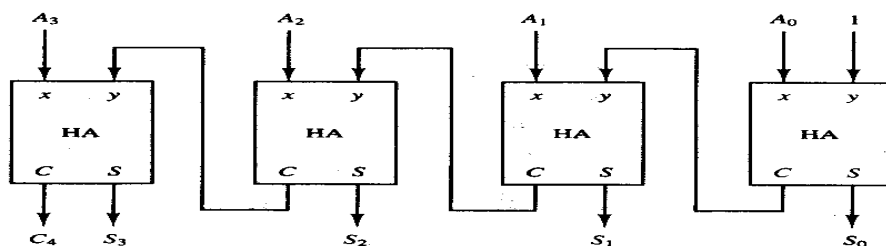


Figure 4-8 4-bit binary incrementer.

Arithmetic Circuit

The arithmetic microoperations listed in Table 4-3 can be implemented in one composite arithmetic circuit. The basic component of an arithmetic circuit is the parallel adder. By controlling the data inputs to the adder, it is possible to obtain different types of arithmetic operations.

The diagram of a 4-bit arithmetic circuit is shown in Fig. 4-9. It has four full-adder circuits that constitute the 4-bit adder and four multiplexers for choosing different operations. There are two 4-bit inputs A and B and a 4-bit output D . The four inputs from A go directly to the X inputs of the binary adder. Each of the four inputs from B are connected to the data inputs of the multiplexers. The multiplexers data inputs also receive the complement of B . The other two data inputs are connected to logic-0 and logic-1. Logic-0 is a fixed voltage value (0 volts for TTL integrated circuits) and the logic-1 signal can be generated through an inverter whose input is 0. The four multiplexers are controlled by two selection inputs, S_1 and S_0 . The input carry C_{in} goes to

the carry input of the FA in the least significant position. The other carries are connected from one stage to the next.

The output of the binary adder is calculated from the following arithmetic sum:

$$D = A + Y + C_{in}$$

where A is the 4-bit binary number at the X inputs and Y is the 4-bit binary number at the Y inputs of the binary adder. C_{in} is the input carry, which can be equal to 0 or 1. Note that the symbol + in the equation above denotes an arithmetic plus. By controlling the value of Y with the two selection inputs S_1 and S_0 and making C_{in} equal to 0 or 1, it is possible to generate the eight arithmetic microoperations listed in Table 4-4.

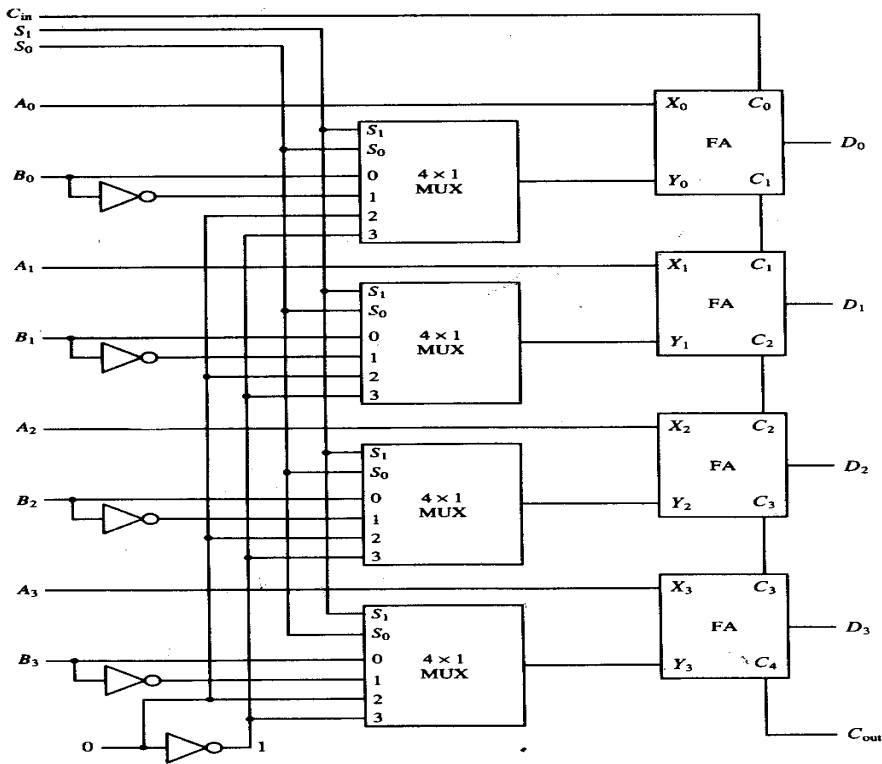


Figure 4-9 4-bit arithmetic circuit.

TABLE 4-4 Arithmetic Circuit Function Table

Select			Input Y	Output $D = A + Y + C_{in}$	Microoperation
S_1	S_0	C_{in}			
0	0	0	B	$D = A + B$	Add
0	0	1	B	$D = A + B + 1$	Add with carry
0	1	0	\bar{B}	$D = A + \bar{B}$	Subtract with borrow
0	1	1	\bar{B}	$D = A + \bar{B} + 1$	Subtract
1	0	0	0	$D = A$	Transfer A
1	0	1	0	$D = A + 1$	Increment A
1	1	0	1	$D = A - 1$	Decrement A
1	1	1	1	$D = A$	Transfer A

When $S_1S_0=00$, the value of B is applied to the Y inputs of the adder. If $C_{in} = 0$, the output $D = A + B + 1$. Both cases perform the add microoperation with or without adding the input carry.

When $S_1S_0=01$, the complement of B is applied to the Y inputs of the adder. If $C_{in} = 1$, then $D = A + /B + 1$. This produces A plus the 2's complement of B , which is equivalent to a subtraction of $A-B$. When $C_{in} = 0$, then $D = A + /B$. This is equivalent to a subtract with borrow, that is, $A - B - 1$.

When $S_1S_0 = 10$, the inputs from B are neglected, and instead, all 0's are inserted into the Y inputs. The output becomes $D = A + 0 + C_{in}$. This gives $D = A$ when $C_{in} = 0$ and $D = A + 1$ when $C_{in} = 1$. In the first case we have a direct transfer from input A to output D . In the second case, the value of A is incremented by 1.

When $S_1S_0 = 11$, all 1's are inserted into the Y inputs of the adder to produce the decrement operation $D = A - 1$ when $C_{in} = 0$. This is because a number with all 1's is equal to the 2's complement of 1 (the 2's complement of binary 0001 is 1111). Adding a number A to the 2's complement of 1 produces $F = A + 2$'s complement of 1 = $A - 1$. When $C_{in} = 1$, then $D = A - 1 + 1 = A$, which causes a direct transfer from input A to output D . Note that the microoperation $D = A$ is generated twice, so there are only seven distinct microoperations in the arithmetic circuit.

4-5 Logic Microoperations

Logic microoperations specify binary operations for strings of bits stored in registers. These operations consider each bit of the register separately and treat them as binary variables. For example, the exclusive-OR microoperation with the contents of two registers $R1$ and $R2$ is symbolized by the statement

$$P: R1 \leftarrow R1 \oplus R2$$

It specifies a logic microoperation to be executed on the individual bits of the registers provided that the control variable $P = 1$. As a numerical example, assume that each register has four bits. Let the content of $R1$ be 1010 and the content of $R2$ be 1100. The exclusive-OR microoperation stated above symbolizes the following logic computation:

$$\begin{array}{rcl} 1010 & \text{Content of } R1 & \\ 1100 & \text{Content of } R2 & \\ \hline 0110 & \text{Content of } R1 \text{ after } P=1 & \end{array}$$

The content of $R1$, after the execution of the microoperation, is equal to the bit-by-bit exclusive-OR operation on pairs of bits in $R2$ and previous values of $R1$. The logic microoperations are seldom used in scientific computations, but they are very useful for bit manipulation of binary data and for making logical decisions.

Special symbols will be adopted for the logic microoperations OR, AND, and complement, to distinguish them from the corresponding symbols used to express Boolean functions. The symbol \vee will be used to denote an OR microoperation and the symbol \wedge to denote an AND microoperation. The complement microoperation is the same as the 1's complement and uses a bar on top of the symbol that denotes the register name. By using different symbols, it will be possible to differentiate between a logic microoperation and a control (or Boolean) function. Another reason for adopting two sets of symbols is to be able to distinguish the symbol $+$, when

used to symbolize an arithmetic plus, from a logic OR operation. Although the + symbol has two meanings, it will be possible to distinguish between them by noting where the symbol occurs. When the symbol + occurs in a microoperation, it will denote an arithmetic plus. When it occurs in a control (or Boolean) function, it will denote an OR operation. We will never use it to symbolize an OR microoperation. For example, in the statement

$$P + Q: R1 \leftarrow R2 + R3, R4 \leftarrow R5 \vee R6$$

The + between P and Q is an OR operation between two binary variables of a control function. The + between R2 and R3 specifies an add microoperation. The OR microoperation is designated by the symbol \vee between registers R5 and R6.

List of Logic Microoperations

There are 16 different logic operations that can be performed with two binary variables. They can be determined from all possible truth tables obtained with two binary variables as shown in Table 4-5. In this table, each of the 16 columns F_0 through F_{15} represents a truth table of one possible Boolean function for the two variables x and y. Note that the functions are determined from the 16 binary combinations that can be assigned to F.

TABLE 4-5 Truth Tables for 16 Functions of Two Variables

x	y	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

The 16 Boolean functions of two variables x and y are expressed in algebraic form in the first column of Table 4-6. The 16 logic microoperations are derived from these functions by replacing variable x by the binary content of register A and variable y by the binary content of register B. It is important to realize that the Boolean functions listed in the first column of Table 4-6 represent a relationship between two binary variables x and y. The logic microoperations listed in the second column represent a relationship between the binary content of two registers A and B. Each bit of the register is treated as a binary variable and the microoperation is performed on the string of bits stored in the registers.

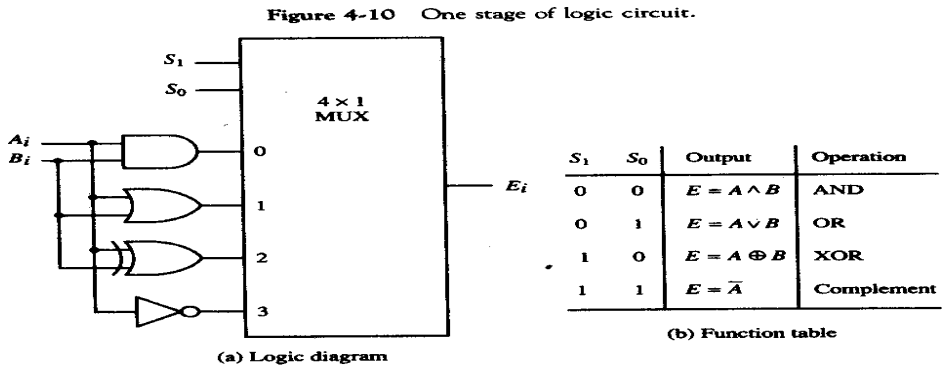
TABLE 4-6 Sixteen Logic Microoperations

Boolean function	Microoperation	Name
$F_0 = 0$	$F \leftarrow 0$	Clear
$F_1 = xy$	$F \leftarrow A \wedge B$	AND
$F_2 = xy'$	$F \leftarrow A \wedge \overline{B}$	
$F_3 = x$	$F \leftarrow A$	Transfer A
$F_4 = x'y$	$F \leftarrow \overline{A} \wedge B$	
$F_5 = y$	$F \leftarrow B$	Transfer B
$F_6 = x \oplus y$	$F \leftarrow A \oplus B$	Exclusive-OR
$F_7 = x + y$	$F \leftarrow A \vee B$	OR
$F_8 = (x + y)'$	$F \leftarrow \overline{A \vee B}$	NOR
$F_9 = (x \oplus y)'$	$F \leftarrow \overline{A \oplus B}$	Exclusive-NOR
$F_{10} = y'$	$F \leftarrow \overline{B}$	Complement B
$F_{11} = x + y'$	$F \leftarrow A \vee \overline{B}$	
$F_{12} = x'$	$F \leftarrow \overline{A}$	Complement A
$F_{13} = x' + y$	$F \leftarrow \overline{A} \vee B$	
$F_{14} = (xy)'$	$F \leftarrow \overline{A \wedge B}$	NAND
$F_{15} = 1$	$F \leftarrow \text{all 1's}$	Set to all 1's

Hardware Implementation

The hardware implementation of logic microoperations requires that logic gates be inserted for each bit or pair of bits in the registers to perform the required logic function. Although there are 16 logic microoperations, most computers use only four – AND, OR, XOR (exclusive-OR), and complement – from which all others can be derived.

Figure 4-10 shows one stage of a circuit that generates the four basic logic microoperations. It consists of four gates and a multiplexer. Each of the four logic operations is generated through a gate that performs the required logic. The outputs of the gates are applied to the data inputs of the multiplexer. The two selection inputs S_1 and S_0 choose one of the data inputs of the multiplexer and direct its value to the output. The diagram shows one typical stage with subscript i . For a logic circuit with n bits, the diagram must be repeated n times for $i = 0, 1, 2, \dots, n-1$. The selection variables are applied to all stages. The function table in Fig. 4-10(b) lists the logic microoperations obtained for each combination of the selection variables.



Some Applications

Logic microoperations are very useful for manipulating individual bits or a portion of a word stored in a register. They can be used to change bit values, delete a group of bits, or insert new bit values into a register. The following examples show how the bits of one register (designated by A) are manipulated by logic microoperations as a function of the bits of another register (designated by B). In a typical application, register A is a processor register and the bits of register B constitute a logic operand extracted from memory and placed in register B.

The selective-set operation sets to 1 the bits in register A where there are corresponding 1's in register B. It does not affect bit positions that have 0's in B. The following numerical example clarifies this operation:

$$\begin{array}{rcl}
 1010 & \text{A before} & \\
 \underline{1100} & \text{B (logic operand)} & \\
 1110 & \text{A after} &
 \end{array}$$

The two leftmost bits of B are 1's, so the corresponding bits of A are set to 1. One of these two bits was already set and the other has been changed from 0 to 1. The two bits of A with corresponding 0's in B remain unchanged. The example above serves as a truth table since it has all four possible combinations of two binary variables. From the truth table we note that the bits

of A after the operation are obtained from the logic-OR microoperation can be used to selectively set bits of a register.

The selective-complement operation complements bits in A where there are corresponding 1's in B. It does not affect bit positions that have 0's in B. For example:

1010	A before
<u>1100</u>	B (logic operand)
0110	A after

Again the two leftmost bits of B are 1's, so the corresponding bits of A are complemented. This example again can serve as a truth table from which one can deduce that the selective-complement operation is just an exclusive-OR microoperation. Therefore, the exclusive-OR microoperation can be used to selectively complement bits of a register.

The selective-clear operation clears to 0 the bits in A only where there are corresponding 1's in B. For example:

1010	A before
<u>1100</u>	B (logic operand)
0010	A after

Again the two leftmost bits of B are 1's, so the corresponding bits of A are cleared to 0. One can deduce that the Boolean operation performed on the individual bits is AB' . The corresponding logic microoperation is

$$A \leftarrow A \wedge / B$$

The mask operation is similar to the selective-clear operation except that the bits of A are cleared only where there are corresponding 0's in B. The mask operation is an AND micro operation as seen from the following numerical example:

1010	A before
<u>1100</u>	B (logic operand)
1000	A after

The two rightmost bits of A are cleared because the corresponding bits of B are 0's. The two leftmost bits are left unchanged because the corresponding bits of B are 1's. The mask operation is more convenient to use than the selective-clear operation because most computers provide an AND instruction, and few provide an instruction that executes the microoperation for selective-clear.

The insert operation inserts a new value into a group of bits. This is done by first masking the bits and then ORing them with the required value. For example, suppose that an A register contains eight bits, 0110 1010. To replace the four leftmost bits by the value 1001 we first mask the four unwanted bits:

0110 1010	A before
<u>0000 1111</u>	B (mask)
0000 1010	A after

and then insert the new value:

$$\begin{array}{rcl} 0110 & 1010 & \text{A before} \\ \underline{1001} & 0000 & \text{B (insert)} \\ 1001 & 1010 & \text{A after} \end{array}$$

The mask operation is an AND microoperation and the insert operation is an OR microoperation.

The clear operation compares the words in A and B and produces an all 0's result if the two numbers are equal. This operation is achieved by an exclusive-OR microoperation as shown by the following example:

$$\begin{array}{rcl} 1010 & \text{A} \\ \underline{1010} & \text{B} \\ 0000 & \text{A} \leftarrow \text{A} \oplus \text{B} \end{array}$$

When A and B are equal, the two corresponding bits are either both 0 or both 1. In either case the exclusive-OR operation produces a 0. The all-0's result is then checked to determine if the two numbers were equal.

4-6 Shift Microoperations

Shift microoperations are used for serial transfer of data. They are also used in conjunction with arithmetic, logic, and other data-processing operations. The contents of a register can be shifted to the left or the right. At the same time that the bits are shifted, the first flip-flop receives its binary information from the serial input. During a shift-left operation the serial input transfers a bit into the rightmost position. During a shift-right operation the serial input transfers a bit into the leftmost position. The information transferred through the serial input determines the type of shift. There are three types of shifts: logical circular, and arithmetic.

A logical shift is one that transfers 0 through the serial input. We will adopt the symbols shl and shr for logical shift-left and shift-right microoperations. For example:

$$\begin{array}{l} R1 \leftarrow \text{shl } R1 \\ R2 \leftarrow \text{shr } R2 \end{array}$$

are two microoperations that specify a 1-bit shift to the left of the content of register *R1* and a 1-bit shift to the right of the content of register *R2*. The register symbol must be the same on both sides of the arrow. The bit transferred to the end position through the serial input is assumed to be 0 during a logical shift.

The circular shift (also known as a rotate operation) circulates the bits of the register around the two ends without loss of information. This is accomplished by connecting the serial output of the shiftregister to its serial input. We will use the symbols cil and cir for the circular shift left and right, respectively. The symbolic notation for the shift microoperations is shown in Table 4-7.

TABLE 4-7 Shift Microoperations

Symbolic designation	Description
$R \leftarrow \text{shl } R$	Shift-left register <i>R</i>
$R \leftarrow \text{shr } R$	Shift-right register <i>R</i>
$R \leftarrow \text{cil } R$	Circular shift-left register <i>R</i>
$R \leftarrow \text{cir } R$	Circular shift-right register <i>R</i>
$R \leftarrow \text{ashl } R$	Arithmetic shift-left <i>R</i>
$R \leftarrow \text{ashr } R$	Arithmetic shift-right <i>R</i>

An arithmetic shift is a microoperation that shifts a signed binary number to the left or right. An arithmetic shift-left multiplies a signed binary number by 2. An arithmetic shift-right divides the number by 2. Arithmetic shifts must leave the sign bit unchanged because the sign of the number remains the same when it is multiplied or divided by 2. The leftmost bit in a register holds the sign bit, and the remaining bits hold the number. The sign bit is 0 for positive and 1 for negative. Negative numbers are in 2's complement form. Figure 4-11 shows a typical register of n bits. Bit R_{n-1} in the leftmost position holds the sign bit. R_{n-2} is the most significant bit of the number and R_0 is the least significant bit. The arithmetic shift-right leaves the sign bit unchanged and shifts the number (including the sign bit) to the right. Thus R_{n-1} remains the same, R_{n-2} receives the bit from R_{n-1} , and so on for the other bits in the register. The bit in R_0 is lost.

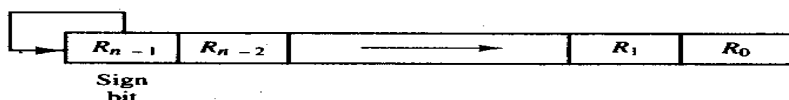


Figure 4-11 Arithmetic shift right.

The arithmetic shift-left inserts a 0 into R_0 , and shifts all other bits to the left. The initial bit of R_{n-1} is lost and replaced by the bit from R_{n-2} . A sign reversal occurs if the bit in R_{n-1} changes in value after the shift. This happens if the multiplication by 2 causes an overflow. An overflow occurs after an arithmetic shift left if initially, before the shift, R_{n-1} is not equal to R_{n-2} . An overflow flip-flop V_s can be used to detect an arithmetic shift-left overflow.

$$V_s = R_{n-1} \oplus R_{n-2}$$

If $V_s = 0$, there is no overflow, but if $V_s = 1$, there is an overflow and a sign reversal after the shift. V_s must be transferred into the overflow flip-flop with the same clock pulse that shifts the register.

Hardware Implementation

A possible choice for a shift unit would be a bidirectional shift register with parallel load (see Fig. 2-9). Information can be transferred to the register in parallel and then shifted to the right or left. In this type of configuration, a clock pulse is needed for loading the data into the register, and another pulse is needed to initiate the shift. In a processor unit with many registers it is more efficient to implement the shift operation with a combinational circuit. In this way the content of a register that has to be shifted is first placed onto a common bus whose output is connected to the combinational shifter, and the shifted number is then loaded back into the register. This requires only one clock pulse for loading the shifted value into the register.

A combinational circuit shifter can be constructed with multiplexers as shown in Fig. 4-12. The 4-bit shifter has four data inputs, A_0 through A_3 , and four data outputs, H_0 through H_3 . There are two serial inputs, one for shift left (I_L) and the other for shift right (I_R). When the selection input $S = 0$, the input data are shifted right (down in the diagram). When $S = 1$, the input data are shifted left (up in the diagram). The function table in Fig. 4-12 shows which input goes to each output after the shift. A shifter with n data inputs and outputs requires n multiplexers. The two serial inputs can be controlled by another multiplexer to provide the three possible types of shifts.

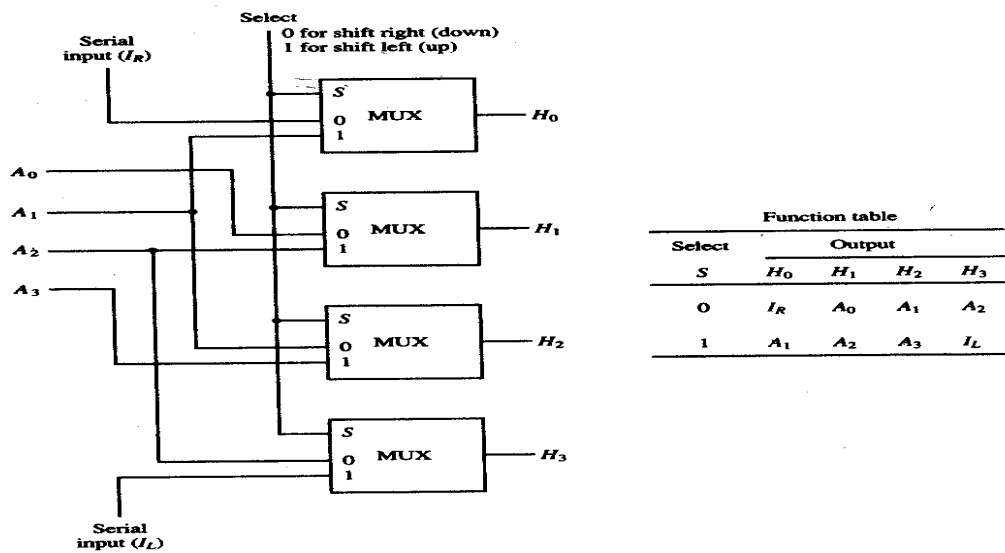


Figure 4-12 4-bit combinational circuit shifter.

4-7 Arithmetic Logic Shift Unit

Instead of having individual registers performing the microoperations directly, computer systems employ a number of storage registers connected to a common operational unit called an arithmetic logic unit, abbreviated ALU. To perform a microoperation, the contents of specified registers are placed in the inputs of the common ALU. The ALU performs an operation and the result of the operation is then transferred to a destination register. The ALU is a combinational circuit so that the entire register transfer operation from the source registers through the ALU and into the destination register can be performed during one clock pulse period. The shift microoperations are often performed in a separate unit, but sometimes the shift unit is made part of the overall ALU.

The arithmetic, logic, and shift circuits introduced in previous sections can be combined into one ALU with common selection variables. One stage of an arithmetic logic shift unit is shown in Fig. 4-13. The subscript i designates a typical stage. Inputs A_i and B_i are applied to both the arithmetic and logic units. A particular microoperation is selected with inputs S_1 and S_0 . A 4×1 multiplexer at the output chooses between an arithmetic output in E_i and a logic output in H_i . The data in the multiplexer are selected with inputs S_3 and S_2 . The other two data inputs to the multiplexer receive inputs A_{i-1} for the shift-right operation and A_{i+1} for the shift-left operation. Note that the diagram shows just one typical stage. The circuit of Fig 4-13 must be repeated n times for an n -bit ALU. The output carry C_{i+1} of a given arithmetic stage must be connected to the input carry C_i of the next stage in sequence. The input carry to the first stage is the input carry C_{in} , which provides a selection variable for the arithmetic operations.

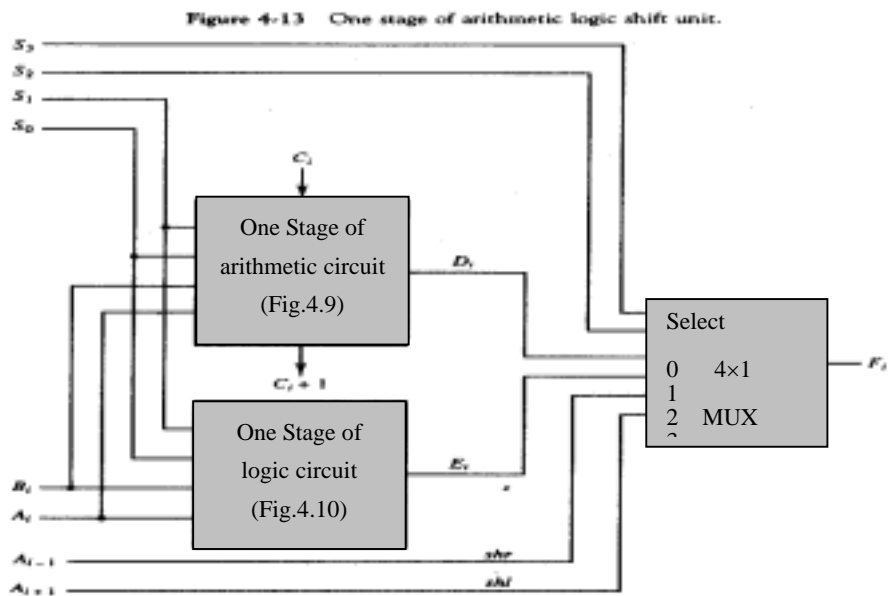


Table 4-8 lists the 14 operations of the ALU. The first eight are arithmetic operations (see Table 4-4) and are selected with $S_3S_2=00$. The next four are logic operations (see Fig. 4-10) and are selected with $S_3S_2 = 01$. The input carry has no effect during the logic operations and is marked with don't-care \times 's. The last two operations are shift operations and are selected with $S_3S_2 = 10$ and 11 . The other three selection inputs have no effect on the shift.

TABLE 4-8 Function Table for Arithmetic Logic Shift Unit

Operation select					Operation	Function
S_3	S_2	S_1	S_0	C_{in}		
0	0	0	0	0	$F = A$	Transfer A
0	0	0	0	1	$F = A + 1$	Increment A
0	0	0	1	0	$F = A + B$	Addition
0	0	0	1	1	$F = A + B + 1$	Add with carry
0	0	1	0	0	$F = A + \overline{B}$	Subtract with borrow
0	0	1	0	1	$F = A + \overline{B} + 1$	Subtraction
0	0	1	1	0	$F = A - 1$	Decrement A
0	0	1	1	1	$F = A$	Transfer A
0	1	0	0	\times	$F = A \wedge B$	AND
0	1	0	1	\times	$F = A \vee B$	OR
0	1	1	0	\times	$F = A \oplus B$	XOR
0	1	1	1	\times	$F = \overline{A}$	Complement A
1	0	\times	\times	\times	$F = shr A$	Shift right A into F
1	1	\times	\times	\times	$F = shl A$	Shift left A into F

PROBLEMS

4-1. Show the block diagram of the hardware (similar to Fig. 4-2a) that implements the following register transfer statement:

$$yT_2: R2 \leftarrow R1, R1 \leftarrow R2$$

- 4-2. The outputs of four registers, R_0 , R_1 , R_2 , and R_3 , are connected through 4-to-1-line multiplexers to the inputs of a fifth register, R_5 . Each register is eight bits long. The required transfers are dictated by four timing variables T_0 through T_3 as follows:

$T_0: R_5 \leftarrow R_0$

$T_1: R_5 \leftarrow R_1$

$T_2: R_5 \leftarrow R_2$

$T_3: R_5 \leftarrow R_3$

The timing variables are mutually exclusive, which mean that only one variable is equal to 1 at any given time, while the other three are equal to 0. Draw a block diagram showing the hardware implementation of the register transfers. Include the connections necessary from the four timing variables to the selection inputs of the multiplexers and to the load input of register R_5 .

- 4-3. Represent the following conditional control statement by two register transfer statements with control functions.

If ($P = 1$) then ($R_1 \leftarrow R_2$) else if ($Q = 1$) then ($R_1 \leftarrow R_3$)

- 4-4. What has to be done to the bus system of Fig. 4-3 to be able to transfer information from any register to any other register? Specifically, show the connections that must be included to provide a path from the outputs of register C to the inputs of register A.
- 4-5. Draw a diagram of a bus system similar to the one shown in Fig 4-3, but use three-state buffers and a decoder instead of the multiplexers.
- 4-6. A digital computer has a common bus system for 16 registers of 32 bits each. The bus is constructed with multiplexers.
- How many selection inputs are there in each multiplexer?
 - What size of multiplexers are needed?
 - How many multiplexers are there in the bus?
- 4-7. The following transfer statements specify a memory. Explain the memory operation in each case.
- $R_2 \leftarrow M[AR]$
 - $M[AR] \leftarrow R_3$
 - $R_5 \leftarrow [R_5]$
- 4-8. Draw the block diagram for the hardware that implements the following statements:

$x+yz: AR \leftarrow AR + BR$

where AR and BR are two n -bit registers and x , y , and z are control variables. Include the logic gates for the control function. (Remember that the symbol $+$ designates an OR operation in a control or Boolean function but that it represents an arithmetic plus in a microoperation.)

- 4-9. Show the hardware that implements the following statement. Include the logic gates for the control function and a block diagram for the binary counter with a count enable input.

$XyT_0 + T_1 + y'T_2: AR \leftarrow AR + 1$