Data Representation

IN THIS CHAPTER

3-1 Data Types

The term data refers to factual information used for analysis or reasoning. Data itself has no meaning, but becomes information when it is assigned a meaning or interpreted. Information is a collection of facts or data that is communicated. 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 compu tations, (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 l's and 0's. The binary number system is the most natural system to use in a dig ital computer. But sometimes it is convenient to employ different number sys tems, especially the decimal number system, since it is used by people to perform arithmetic computations.

Number

Systems

radix

decimal

binary

octal
hexadecimal conversion

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 num ber 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 sym bols of the octal system are 0, 1, 2, 3, 4, 5, 6, and 7. The 16 symbols of the hexa decimal 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:

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

The equivalent decimal number of hexadecimal F3 is obtained from the fol lowing 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 3-1 demonstrates these procedures.

The conversion of decimal 41.6875 into binary is done by first separat ing 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 com bined 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 cor responds 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 num ber. Consider, for example, a 16-bit register. Physically, one may think of the

Figure 3-1 Conversion of decimal 41.6875 into binary.

Integer 41 Fraction 0.6875

412010521 10010
   0   1

$$\frac{2}{0.7500}$$

$$\frac{2}{1.5000}$$

$(41)_{10}$ $(101001)_2$

0.6875

$$\frac{2}{1.3750}$$

$$\frac{2}{1.0000}$$

$(0.6875)_{10}$ $(0.1011)_2$

$(41.6875)_{10}$ $(101001.1011)_2$

2
7 5 4 3 Octal

1

1 0 1 0 1 1 1 1 0 1 1 0 0 Binary

0 1 1 A F 6 3   Hexadecimal

Figure 3-2 Binary, octal, and hexadecimal conversion.

register as composed of 16 binary storage cells, with each cell capable c hold ing either a 1 or a 0. Suppose that the bit configuration stored in the register is as shown in Fig. 3-2. Since a binary number consists of a string of l'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.

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. 3-2. 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 3-1. The correspondence between a hexadecimal digit and its equivalent 4-bit code can be found in the first 16 entries of Table 3-2.

TABLE 3-1 Binary-Coded Octal Numbers

Octal number	Binary-coded octal	Decimal equivalent	
0	000	0	
1	001	1	
2	010	2	Code
3	011	3	for one
4	100	4	octal
5	101	5	digit
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 3-1 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 3-2 lists a few hexa decimal 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.

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 con verted 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 represen tations, we find that their bit combination is identical. This should be so because of the straightforward conversion that exists between binary num bers and octal or hexadecimal. The point of all this is that a string of l'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 num ber in binary-coded form (if we divide the bits in groups of four).

TABLE 3-2 Binary-Coded Hexadecimal Numbers

Hexadecimal number	Binary-coded hexadecimal	Decimal equivalent	
0	0000	0	
1	0001	1	
2	0010	2	
3	0011	3	
4	0100	4	
5	0101	5	
6	0110	6	Code
7	0111	7	for one
8	1000	8	hexadecimal
9	1001	9	digit
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	

The registers in a digital computer contain many bits. Specifying the con tent of registers by their binary values will require a long string of binary dig its. 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 peo ple 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 per form 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 num bers 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.

binary code BCD

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 repre sented 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

A binary code is a group of n bits that assume up to 2^n distinct combina tions of l'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 10 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 3-3. 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. 3-5.

It is very important to understand the difference between the *conversion* of decimal numbers into binary and the *binary coding* of decimal

8 1000
9 1001

10 0001 0000
20 0010 0000
50 0101 0000
99 1001 1001
248 0010 0100 1000

number itself is exactly the same. A few decimal numbers and their repre sentation in BCD are listed in Table 3-3.

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 cer tain 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 stan dard 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 3-4. 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 infor mation. One must realize that binary codes merely change the symbols, not the meaning of the discrete elements they represent. The operations speci fied for digital computers must take into consideration the meaning of the

character ASCII

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

Decimal number	Binary-coded decimal (BCD) number
0	0000
1	0001
2	0010
3	0011	Code
4	0100	for one
5	0101	decimal
6	0110	digit
7	0111

TABLE 3-4 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		

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* represen tation. Instruction codes are discussed in Chap. 5.

3-2 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 $(r \quad l)$'s complement.

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 exam ple, with $n = 4$, we have $2^4 = (10000)_2$ and $2^4 - 1 = (1111)_2$. Thus the 1's com plement 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 comple ment 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 be leaving all least sig nificant 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

---

9's complement1's complement

10's complement 2'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

the two zeros unchanged, sub tracting 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 l's by 0's and 0's by l's in all other higher, signif icant 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 l's by 0's and 0's by l'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 comple ment 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 bor row 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 pen cil. When subtraction is implemented with digital hardware, this method is found to be less efficient than the method that uses complements.

subtraction end carry

sign in front.

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

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

M 72532
10's complement of N 86750

Sum   159282
Discard end carry 10^5 <u>100000</u>

M   13250
10's complement of N   <u>27468</u>

Answer   59282

Now consider an example with M   N. The
subtraction 13250   72532 produces negative
59282. Using the procedure with complements,
we have

Sum   40718

binary point

There is no end carry

Answer is negative 59282   10'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 recog nized 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:

Answer: X Y 0010001

Y _000011
2's complement of X 0101100

X 1010100
2's complement of Y _0111101

Sum 10010001
Discard end carry 2^7 10000000

Sum 1101111

There is no end carry

Answer is negative 0010001 2's complement of 1101111

3-3 Fixed-Point Representation

Positive integers, including zero, can be represented as unsigned numbers. However, to represent negative integers, we need a notation for negative val ues. 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 l'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 left most 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 pres ent, but its presence is assumed from the fact that the number stored in the reg ister 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 dec imal point in the first register. Floating-point representation is discussed fur ther in the next section.

Integer Representation

signed numbers

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 nega tive number is represented in either the l's or 2's complement of its positive value. As an example, consider the signed number 14 stored in an 8-bit reg ister. 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 com plementing 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 com plement 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 l's complement imposes difficulties because

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 com parison of the signs and the magnitudes and then performing either addition or subtraction. (The procedure for adding binary numbers in signed-magni tude 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 ini tially be in 2's complement and that if the sum obtained after the addition is negative, it is in 2's complement form.

```
  6 00000110   6 11111010
 13 00001101  13 00001101
 19 00010011   7 00000111

  6 00000110   6 11111010
 13 11110011  13 11110011
  7 11111001  19 11101101
```

In each of the four cases, the operation performed is always addition, includ ing the sign bits. Any carry out of the sign bit position is discarded, and nega tive 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 neg ative number when 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 leftmost bit is 1. Its 2's com plement is 00000111, which is the binary equivalent of 7. We therefore rec ognize the original negative number to be equal to 7.

2's complement addition

it has two representations of 0 (0 and 0). It is seldom used for arithmetic operations except in some older computers. The l's complement is useful as a logical operation since the change of 1 to 0 or 0 to 1 is equivalent to a log ical complement operation. The following discussion of signed binary arith metic 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

Arithmetic

Subtraction

2's complement subtraction

number is easily done by tak ing its 2's complement. The reverse is also true because the complement of a negative number in complement form produces the equivalent positive num ber. 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 100000111. Removing the end carry, we obtain the correct answer 00000111 (7).

It is worth noting that binary numbers in the signed-2's complement sys tem are added and subtracted by the same basic addition and subtraction rules as unsigned numbers. Therefore, computers need only one common hard ware 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 dig ital 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.

overflow

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 minu end (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:

(A) (B) (A) (B)

(A) (B) (A) (B)

But changing a positive number to a negative

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 num bers, the leftmost bit always represents the sign, and negative numbers are in 2's complement form. When two signed numbers are added, the

both positive or both negative. The two additions in binary are shown below together with the last two carries.

```
carries: 0 1    carries: 1 0
  70 0 1000110   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 exam ples 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:

overflow detection

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 capac ity of the 8-bit register. This is true if the numbers are

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

0100 0011 1000 0101

arithmetic are more complex. However, there are some advantages in the use of decimal representation because computer input and output data are gen erated 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 elimi nate the need for conversion to binary and

back to decimal. Some computer systems have hardware for arithmetic calculations with both binary and dec imal 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 dec imal 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 comple ment 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 subtrac tion 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 comple ment form. Consider the addition (375) (240) 135 done in the signed-10's complement system.

$$0\ 375\ (0000\ 0011\ 0111\ 0101)_{BCD}$$
$$9\ 760\ (1001\ 0111\ 0110\ 0000)_{BCD}$$
$$0\ 135\ (0000\ 0001\ 0011\ 0101)_{BCD}$$

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 num bers inside the computer must be in BCD, including the sign digits. The addi tion 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 comple ment of the subtrahend and add it to the minuend. Many computers have spe cial 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.

mantissa
exponent

3-4 Floating-Point Representation

The floating-point representation of a number has two parts. The first part rep resents a signed, fixed-point number called the mantissa. The second part desig nates 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 expo nent as follows:

Fraction Exponent

0.6132789 04

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 follow ing form:

$$m \times r^e$$

Only the mantissa *m* and the exponent *e* are physically represented in the reg ister (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

fraction

normalization

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 6-bit exponent as follows:

Fraction Exponent

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
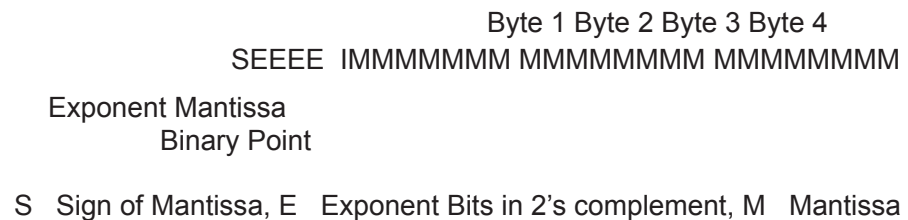
$$m \quad 2^e \quad (.1001110)_2 \quad 2^4$$

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 float ing-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 rep resented in floating-point by all 0's in the mantissa and exponent.

Two main standard forms of floating-point numbers are from the follow ing organizations that decide standards: ANSI (American National Standards Institute) and IEEE (Institute of Electrical and Electronic Engineers). The ANSI 32-bit floating-point numbers in byte format with examples are given below:

Byte Format:

Byte 1 Byte 2 Byte 3 Byte 4

SEEEE IMMMMMMM MMMMMMMM MMMMMMMM

Exponent Mantissa
Binary Point

S   Sign of Mantissa, E   Exponent Bits in 2's complement, M   Mantissa

Bits *Examples:*

13   1101   0.1101 2^4
00000100 11010000 00000000 00000000

17   10001   0.10001 2^5
10000101 10001000 00000000 00000000

0.125   0.001   .1 2^2
11111110 10000000 00000000 00000000

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 prob lems involved with fixed-point computations. Many computers and all elec tronic calculators have the built-in capability of performing floating-point arithmetic operations. Computers that do not have hardware for floating-point computations have a set of subroutines to help the user program scientific prob lems with floating-point numbers. Arithmetic operations with floating-point numbers are discussed in Sec. 10-5.

3-5 Other Binary Codes

In previous sections we introduced the most common types of

binary-coded data found in digital computers. Other binary codes for decimal numbers and