CHAPTER 3

Computer representation and arithmetic

3.1 Representing numbers in a computer

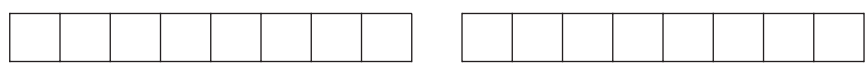
In Chapter 2, we looked at how numbers can be represented in the binary number system. In this chapter, we shall use the techniques we developed in Chapter 2 to investigate the ways in which numbers are represented and manipulated in binary form in a computer.

Numbers are usually represented in a digital computer as sequences of bits of a fixed length. Integers and real numbers are handled in different ways, so we need to deal with the two cases separately. The details vary from one type of computer to another, but our aim here is to gain an overview of the way in which computers handle numbers, rather than to study any particular type of machine in detail.

3.2 Representing integers

An integer (positive, negative or zero) is stored in a computer in a sequence of bytes, where each byte consists of 8 bits. While various sizes, corresponding to different numbers of bytes, may be available to the programmer, the size of the integers manipulated directly by the processor is determined by the design of the processor, with 4 bytes being typical. In this section, however, we will generally work with 2-byte integers for convenience; the principles are the same.

If integers are stored using 2 bytes, the registers in which an integer is stored can be visualised in the following way, where each box represents the storage location for one bit:



Since there are two possible values (0 or 1) for each bit, the number of different integers that can be stored in 16 bits is:

$$\begin{aligned} &2 \times 2 \times 2 \times \ldots \times 2 \text{ (16 times)} \\ &= 2^{16} \\ &= 65536 \end{aligned}$$

For example, it is possible to store any integer n in the range $-32768 \leq n \leq 32767$ by assigning a unique 16-bit pattern to each integer in that range. Other ranges of values containing 65536 integers could be used instead, such as $0 \leq n \leq 65535$, but a range that includes approximately the same number of positive and negative integers is generally the most useful.

The most commonly used way of representing an integer n using 16 bits is as follows:

1. The first bit is the *sign bit*; it is 0 if n is zero or positive, and 1 if n is negative.
2. If $n \geq 0$, the remaining 15 bits are the binary representation of n as described in Chapter 2 (with leading zeros if necessary, to bring the number of bits up to 15). If $n < 0$, the remaining bits are the binary representation of the non-negative integer $n + 32768$.

Adding 32768 to n when n is negative might seem an unnecessary complication, and you might wonder why the remaining bits could not instead be set equal to the 15-bit binary representation of $|n|$.[1] The reason is that by adding 32768 we obtain a representation of n that allows a simpler process to be used to carry out arithmetic with integers, as we will see in the next section.

The addition of 32768 is actually easier to perform *after* the negative integer n has been converted to binary, as the following example illustrates. Suppose we want to find the 16-bit computer representation of –6772. Converting –6772 to binary (using 15 bits, including leading zeros), we obtain $-6772_{10} = -001101001110100_2$. Now, since the binary representation of 32768 is 1000000000000000_2, we need to perform the following calculation:

$$\begin{array}{rr} & 1000000000000000_2 \\ - & 001101001110100_2 \\ \hline & 110010110001100_2 \end{array}$$

There is a short-cut method of doing this subtraction:

1. All the zeros at the right-hand end of the number being subtracted remain as zeros in the answer.

---

1 $|n|$ denotes the *absolute value* of n, defined by:

$$|n| = \begin{cases} n & \text{if } n \geq 0 \\ -n & \text{if } n < 0 \end{cases}$$

For example, $|3| = 3$ and $|-3| = 3$.

2. The rightmost 1 of the number remains as a 1 in the answer.
3. All the other bits change (0 changes to 1, 1 changes to 0).

The result of applying these steps is called the *2's complement* of the original number with respect to the base 2. The '2's' is included in the name to distinguish this type of complement from the 1's complement (still with respect to base 2). The 1's complement of a binary number is one less than its 2's complement, and is obtained simply by changing each bit of the number (0 changes to 1, 1 changes to 0). We will not use 1's complements in what follows.

More generally, let n be an integer in the range $1 \leq n \leq 2^k - 1$, written in its k-bit binary representation. The k-bit[2] 2's complement of n is $2^k - n$ written in its k-bit binary representation. It is obtained by applying the above steps to n.

The short-cut method cannot be used to find the computer representation of –32768, because the binary representation of 32768 has 16 bits, not 15. In this case the result of the subtraction is a string of 15 zeros.

Example 3.2.1 Find the 16-bit computer representations of the following integers:

(a) 9843 (b) –15728 (c) –4961

Solution

(a) The sign bit is 0, and $9843_{10} = 010011001110011_2$ (using 15 bits). The 16-bit representation is:

00100110 01110011

(b) The sign bit is 1.

Convert 15728 to binary: $15728_{10} = 011110101110000_2$
Find the 2's complement: 100001010010000_2
The 16-bit representation is:

11000010 10010000

(c) The sign bit is 1.

Convert 4961 to binary: $4961_{10} = 001001101100001_2$
Find the 2's complement: 110110010011111_2
The 16-bit representation is:

11101100 10011111

If the computer uses 4 bytes (32 bits) to store integers instead of 2, any integer n in the range $-2147483648 \leq n \leq 2147483647$ can be represented, because $2147483648 = 2^{31}$. The method for finding the representation is essentially unchanged, except that 2147483648 takes the place of 32768, and the binary conversion must be taken to 31 bits instead of 15.

2 The number of bits is usually not stated explicitly, but is implied by the number of bits given in n.

3.3 Arithmetic with integers

In order for us to investigate how a computer adds and subtracts integers, it is convenient to work with examples in which integers are represented using a small number of bits – far smaller than any computer would use in practice. The examples in this section all involve an imaginary computer in which integers are stored as four bits. Only the integers from –8 to 7 can be represented on this computer. Table 3.1 gives a complete list of these integers, together with their representations according to the rules given in the previous section.

Table 3.1

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

(Note that to calculate the representation of –8 we cannot use the short-cut rule for finding the 2's complement.)

By examining Table 3.1, we can make the following observations:

- For the non-negative integers (0 to 7), the last 3 bits of the computer representation form the 3-bit binary representation of the integer. Since the sign bit is 0 for these integers, the computer representation is just the 4-bit binary representation of the integer.

- For a negative integer n (–8 to –1), the last 3 bits of the computer representation form the 3-bit binary representation of $n + 8$. Since the sign bit is 1, with a place value of 8 if the computer representation is treated as a binary number, the computer representation is the 4-bit representation of $n + 16$. (For example, the computer representation of –3 is 1101, which is the binary representation of 13, and $13 = (-3) + 16$.)

Suppose now that we want to add two integers on this computer. Notice first that this will not always be possible; we cannot add 6 and 7, for example, since the result is too large to be represented. Provided we restrict ourselves to numbers that *can* be added, however, addition can be done simply by adding the computer representations in the usual way that binary numbers are added, except that *if a 1 arises in the fifth column from the right, then it is ignored.*

Example 3.3.1 Verify that the following additions are done correctly on the 4-bit computer:

(a) 2 + 3 (b) (–4) + 7

(c) (–3) + (–4) (d) (–6) + 5

Solution We write the computation in two columns, with the left column containing the numbers to be added and their sum, and the right column containing the corresponding computer representations.

(a)

2	0010
3	0011
5	0101

(b)

–4	1100
7	0111
3	1 0011

Ignoring the 1 in the leftmost column of the result gives 0011, which is the computer representation of 3, the correct answer to (–4) + 7.

(c)

–3	1101
–4	1100
–7	1 1001

Ignoring the 1 gives 1001, the computer representation of –7.

(d)

–6	1010
5	0101
–1	1111

It will be easier to see why this process works after we have studied modular arithmetic in Chapter 12, but the general idea can be explained as follows. The value of the representation of an integer on this computer is always the integer itself or 16 more than the integer. Therefore, when two representations are added, the value of the result either equals the answer to the addition, or exceeds it by 16 or 32 (= 2 × 16). Ignoring a 1 in the fifth column from the right is equivalent to subtracting 16. Therefore, when the process is complete, if the result is not the correct answer to the addition, it must differ from the correct answer by a multiple of 16. As long as the sum of the two numbers falls in the allowed range from –8 to 7, the process must give the correct result, because the difference between any two integers in the allowed range is less than 16.

If we try to use this method to add two integers that give a result outside the allowed range, we will obtain the wrong answer. For example, you can check that using the method to add 6 and 7 gives an answer of –3. This *overflow* problem can be easily detected, however. If adding two positive integers appears to give a negative result, or if adding two negative integers appears to give zero or a positive result, then overflow must have occurred. Some applications software that uses integer arithmetic will generate an appropriate error message whenever this happens. However, in many programming environments, arithmetic with integers is performed exactly as we have described, and no error is generated when overflow occurs. For this reason, you need to be aware of the possibility of overflow when you are writing programs, and know how to deal with it.

Subtraction of one integer from another using the computer representation is quite straightforward, once we have made two observations:

- A number can be subtracted by adding its negative, that is, $a - b = a + (-b)$.
- The representation of the negative of an integer is the 2's complement of the representation of the integer.

Subtraction of a number can therefore be carried out by *adding the 2's complement of the number*. This means that any subtraction problem is easily converted into an addition problem. The advantage of doing subtraction in this way is that a computer does not need 'subtracters' as well as 'adders' in its circuitry (although it does need circuitry to calculate 2's complements).

Overflow can occur when subtracting, just as it does when adding, so you need to be alert to this possibility when you are programming.

Example 3.3.2 Evaluate 5 – 3 on the 4-bit computer.

Solution The representations of 5 and 3 are 0101 and 0011 respectively. The 2's complement of 0011 is 1101. Now carry out the addition:

$$\begin{array}{r} 0101 \\ 1101 \\ \hline 1\ 0010 \end{array}$$

Ignore the leftmost 1 to obtain 0010, which is the representation of 2.

Subtraction using complements can be performed in any base. The subtraction of numbers written in decimal notation using this method is explored in Exercises 3 and 4 at the end of the chapter.

3.4 Representing real numbers

The representation of real numbers is more complicated than that of integers, not just in a computer but in printed form as well. In Chapter 2, we dealt with real numbers expressed in the decimal system with digits before and after the decimal point. Not only is this not a convenient representation for a computer, it is often not suitable as a written representation either. One reason for this is that in many problems of practical interest, it is necessary to deal with real numbers that are extremely small or extremely large, and that occur as a result of performing a physical measurement to a certain degree of accuracy. This is a subtle point, but an important one.

An example will help to make this clear. The mass of the Earth is sometimes quoted as 0.5976×10^{25} kg. Notice that this number is not written as 5976 followed by 21 zeros. Not only would it be inconvenient to write the number in that way, it would also not be correct to do so. The figure of 0.5976×10^{25} kg would have been calculated from physical measurements, and no measurement is ever free of error. In this case, the accuracy of the figure is implied by the way it is written. We cannot say that the mass of the Earth is exactly 0.5976×10^{25} kg, only that it lies somewhere between 0.59755×10^{25} kg and 0.59765×10^{25} kg. When more accurate measurements become available, it might turn out that a closer value is, say, 0.59758×10^{25} kg.

The notation used in this example is called *exponential* notation (or 'powers-of-ten' notation). The use of exponential notation is especially appropriate in scientific work, where the numbers arise from physical measurements.

There is some terminology associated with exponential notation. In the number 0.5976×10^{25}, 0.5976 is the *significand*, 10 is the *base* (or radix), and 25 is the *exponent*. If the significand m lies in the range $0.1 \leq m < 1$, as it does in this example, the representation is said to be *normalised*. (If we

had written 0.005976×10^{27}, for example, the number would still be in exponential notation, but it would not be normalised.)

The precise way in which real numbers are represented in a computer varies between types of machines, although in recent years there has been a trend towards adopting the representation specified in the IEEE Standard 754 (1985) for Binary Floating Point Arithmetic. (IEEE stands for the Institute of Electrical and Electronics Engineers.) Our purpose here is not to describe a particular representation, but rather to give a general description of how real numbers are stored and manipulated.

The exponential notation described above is similar to the way in which real numbers are represented in a computer, but there are some differences. The main difference, not surprisingly in view of the way in which computers manipulate data, is that a computer uses powers of 2 rather than 10. It is also usual for the exponent to be stored in a modified form, as we will see shortly.

Before the computer representation of a real number can be found, the number must first be converted to binary form and then be expressed in *normalised binary exponential form*. For the representations we will use in our examples, we say that a real number is expressed in normalised binary exponential form if it is expressed in the form:

$$\pm m \times 2^e$$

where the significand m is written in its binary representation and lies in the range $0.1_2 \leq m < 1_2$, and the exponent e is an integer written in its decimal representation. (The base 2 is also in decimal notation, of course.) Note that other conventions for normalised binary exponential form could be used; in particular, some computers use a representation based on $1_2 \leq m < 10_2$. The reason for normalising is to ensure that the representation of any number is unique.

Note that zero cannot be expressed in normalised binary exponential form, because $m = 0$ would fall outside the allowed range of values of m.

Example 3.4.1 Express the following numbers in normalised binary exponential form:

(a) 11001.101_2 (b) 0.000110111_2

Solution

(a) Move the point 5 places to the left to obtain the significand. The exponent is 5.

$$11001.101_2 = 0.11001101_2 \times 2^5$$

(b) Move the point 3 places to the right to obtain the significand. The exponent is –3.

$$0.000110111_2 = 0.110111_2 \times 2^{-3}$$

A real number is typically stored in a computer as 4 bytes (32 bits) or 8 bytes (64 bits)[3]. The first bit is the sign bit, and the remaining bits are divided between the exponent and the significand. The number of bits used for the exponent determines the *range* of numbers that can be represented, while the number of bits used for the significand determines the *precision* with which the numbers are represented. If the total number of bits is fixed, there is a trade-off between range and precision. A common format in modern computers is 8 bits for the exponent and 23 bits for the significand.

It is usually not the binary representation of the exponent itself that is stored, but a number called the *characteristic*. The characteristic is a non-negative integer obtained by adding to the exponent a number called the *exponent bias*. The exponent bias is typically $2^{n-1} - 1$, where n is the number of bits available to store the characteristic. The reason for storing the exponent in this way is that it allows the computer to use simpler algorithms to perform arithmetic with real numbers, in much the same way that integer arithmetic is simplified by representing negative integers in 2's complement form.

In some representations, including the IEEE standard, the first bit of the significand (which must always be a 1) is not stored, thus increasing the precision by making one extra bit available. We will include the first bit of the significand in all our examples.

Example 3.4.2 Find the 32-bit computer representations of

(a) $0.11001101_2 \times 2^5$ (b) $0.110111_2 \times 2^{-3}$

where 8 bits are used for the characteristic, and the exponent bias is $2^7 - 1$.

Solution

(a) The sign bit is 0. The characteristic is the 8-bit binary representation of $5 + 2^7 - 1$, which can be obtained by adding 10000000_2 (the binary representation of 2^7) to the binary representation of 4, giving 10000100. The significand is extended to 23 bits by appending trailing zeros, giving 11001101000000000000000. (Note that the zero before the point is not stored.) The computer representation is:

01000010 01100110 10000000 00000000

(b) The sign bit is 0. The characteristic is $-3 + 2^7 - 1$, represented as an 8-bit binary number. The simplest way to calculate the characteristic here is to find the 7-bit 2's complement of the binary representation of 4 (= 3 + 1), and adjoin a leading zero:

Binary representation of 4: 0000100_2

2's complement: 1111100_2

Characteristic: 01111100

3 If the two different formats are available on the same machine, they are typically referred to as *single precision* and *double precision* respectively.

The computer representation is:

$$00111110\ 01101110\ 00000000\ 00000000$$

Putting together what we have established, the process for finding the computer representation of a real number can be described as follows:

1. Convert the number to binary form, working to the precision required by the number of bits used for the significand.
2. Express the binary number in normalised binary exponential form.
3. Calculate the characteristic.
4. Write down the computer representation.

Example 3.4.3 Find the 32-bit computer representation of –1873.42, where 8 bits are used for the characteristic, and the exponent bias is $2^7 - 1$.

Solution Using the methods of Chapter 2, convert –1873.42 to a binary number with 23 bits:

$$-1873.42_{10} = -11101010001.011010111000_2$$

Express the result in normalised binary exponential form:

$$-11101010001.011010111000_2 = -0.11101010001011010111000 \times 2^{11}$$

Sign bit: 1

Characteristic: 10001010

Computer representation:

$$11000101\ 01110101\ 00010110\ 10111000$$

Example 3.4.3 illustrates the important fact that *the computer representation of a real number may not be exact*, because the conversion to binary is truncated according to the precision available. Further inaccuracy can occur as a result of round-off errors when arithmetic is performed with real numbers. This is in contrast with the situation for integers, which are stored and manipulated exactly in a computer.

One practical consequence of this fact is that it is often risky to test for equality of real numbers (in an **If-then** statement, for example) in a computer program. If x and y are real numbers, it is usually safer to test for approximate equality by testing whether $|x - y|$ is less than some small positive number. For example, we might write:

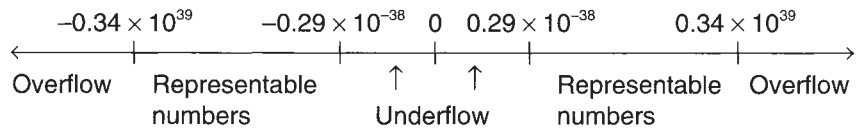
If $|x - y| < 10^{-6}$ **then** ...

to test whether x and y are approximately equal.

What is the range of real numbers that can be represented? Suppose a computer stores real numbers as 32 bits, with 8 bits for the characteristic, and an exponent bias of $2^7 - 1$. The characteristic can take values from 0

to $2^8 - 1$, so the exponent must lie within the range from $0 - (2^7 - 1)$ to $2^8 - 1 - (2^7 - 1)$, that is, from -127 to 128. The significand ranges from 0.1_2 to (just under) 1_2. Therefore the range of positive real numbers that can be represented is from $0.1_2 \times 2^{-127}$ to $1_2 \times 2^{128}$, or about 0.29×10^{-38} to 0.34×10^{39} in decimal notation. Negative real numbers whose absolute values fall in this range can also be represented. If the result of a computation falls outside the allowed range, overflow or underflow[4] occurs. The numbers that can be represented are indicated on the number line shown in Figure 3.1; note that the line is not drawn to scale.

Figure 3.1



Example 3.4.4 Find the approximate range of positive real numbers that can be represented if 10 bits are available for the characteristic, and the exponent bias is $2^9 - 1$.

Solution The characteristic can take values from 0 to $2^{10} - 1$, so the exponent must lie within the range from $0 - (2^9 - 1)$ to $(2^{10} - 1) - (2^9 - 1)$, that is, from -511 to 512. Therefore the range of positive real numbers that can be represented is from $0.1_2 \times 2^{-511}$ to $1_2 \times 2^{512}$. Some calculators give an error message if an attempt is made to evaluate 2^{-511} directly, but the calculation can be done using base 10 logarithms:

$$\begin{aligned} \log_{10} 2^{-511} &= -511 \log_{10} 2 \\ &= -153.8263278 \end{aligned}$$

Therefore:

$$\begin{aligned} 2^{-511} &= 10^{-153.8263278} \\ &= 10^{-153} \times 10^{-0.8263278} \\ &= 0.149 \times 10^{-153} \end{aligned}$$

We can calculate 2^{512} in a similar fashion. The final result is that numbers from 0.75×10^{-154} to 0.13×10^{155} can be represented on this machine.

The number zero is not included among the real numbers whose representations we have been discussing, because it cannot be expressed

4 *Underflow* occurs when the absolute value of the result of a computation involving real numbers is less than the smallest positive number that can be represented. Depending on the context in which the computation occurs, it may or may not be appropriate to approximate the result of an underflow by zero.

in normalised form. The representation of zero has to be treated as a special case, using a string of bits that cannot be interpreted as any other number.

As we noted earlier, the representation of real numbers varies to some extent between machines. In the IEEE standard, the range of real numbers that can be represented is a little smaller than we have described, because the exponents at the extremes of the range have been set aside for special purposes.[5]

3.5 Arithmetic with real numbers

We will not look at how a computer does arithmetic with real numbers using their computer representations, because it is tedious to perform these calculations by hand. However, we can gain a general idea of what the process involves by looking at how arithmetic is done with decimal numbers in normalised exponential form.

The rules for arithmetic in normalised exponential form are as follows. To add or subtract:

1. Write the numbers in (non-normalised) exponential form with the same exponent, using significands less than 1.
2. Add or subtract the significands to obtain the significand of the answer. The common exponent is the exponent of the answer.
3. Normalise the answer if necessary.

To multiply or divide:

1. Multiply or divide the significands to obtain the significand of the answer.
2. Add or subtract the exponents to obtain the exponent of the answer.
3. Normalise the answer if necessary.

In order to imitate the way in which a computer carries out the computations, we will assume that the number of decimal digits available for the significand in the normalised form is fixed, and that the answer is rounded off to that number of digits. (In some machines, greater accuracy is achieved by appending additional "guard" digits to the significand while an arithmetic operation is being carried out. In particular, this is always the case in machines that comply with the IEEE standard.)

Example 3.5.1 Perform the following computations with the aid of a calculator, assuming a precision of four decimal places in the significand:

(a) $0.4932 \times 10^3 + 0.2881 \times 10^4 - 0.3096 \times 10^4$

5 IEEE arithmetic has representations for positive and negative infinity, and undefined quantities (called "not-a-number"). It also allows numbers smaller than the smallest positive normalised number to be represented in non-normalised form, thus allowing "gradual underflow".

(b) $(0.2174 \times 10^{-5}) \times (0.1482 \times 10^{7}) \div (0.9497 \times 10^{4})$

Solution

(a)
$$\begin{aligned} & 0.4932\times10^{3} + 0.2881\times10^{4} - 0.3096\times10^{4} \\ &= 0.0493\times10^{4} + 0.2881\times10^{4} - 0.3096\times10^{4} \\ &= 0.0278\times10^{4} \\ &= 0.2780\times10^{3} \end{aligned}$$

(b)
$$\begin{aligned} & (0.2174\times10^{-5})\times(0.1482\times10^{7})\div(0.9497\times10^{4}) \\ &= 0.03393\times10^{-2} \\ &= 0.3393\times10^{-3} \end{aligned}$$

Example 3.5.1 illustrates how round-off errors can arise when real number calculations are performed.

3.6 Binary coded decimal representation

Modern digital computers work in the binary number system. However, some early computers performed calculations in the decimal system, using a binary code for each decimal digit. This system, known as *binary coded decimal*, or BCD, is also commonly used in electronic calculators.

The BCD code of each decimal digit from 0 to 9 is its 4-bit binary representation, as shown in Table 3.2.

Table 3.2

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

The BCD representation of a non-negative integer is obtained by replacing each decimal digit by its BCD code.

For example, the BCD representation of 8365 is:

1000 0011 0110 0101

BCD arithmetic is essentially decimal arithmetic done according to the familiar rules, but with BCD codes in place of the digits. We shall illustrate this by investigating the addition of two numbers in the BCD system.

When we add two numbers in the decimal system, we add pairs of digits a column at a time, and sometimes there is a 'carry' to the next column to the left. Essentially the same process is used in the BCD system.

How are BCD digit codes added? By examining Table 3.2, we see that any two BCD codes can be added using the rules of binary addition to give the BCD code of the result, provided that the result is less than 10 (for example, adding the codes for 3 and 4 gives the code for 7: $0011_2 + 0100_2 = 0111_2$). If the result is 10 or more (1010_2 or more in binary), there is a (decimal) digit to 'put down' and a 1 to 'carry'. We could find the digit to 'put down' by subtracting ten from the result, but it is simpler to use the following rule: *add six (0110_2) and disregard the leftmost 1*. Applying the rule is equivalent to subtracting ten, because disregarding the leftmost 1 is equivalent to subtracting sixteen (10000_2).

For example, adding 0101_2 and 0111_2 (5 and 7) as binary numbers gives 1100_2. Now add 0110_2 to obtain 10010_2. Interpret this binary number as 0001 0010; that is, put down 0010 (the BCD representation of 2) and carry 0001. On replacing each BCD code with its corresponding decimal digit, we obtain 12, the correct answer to the addition.

Example 3.6.1 Calculate 274 + 163 in BCD.

Solution The decimal and BCD calculations are shown side by side to show the relationship between them.

274	0010	0111	0100
163	0001	0110	0011
	0011	1101	0111
		0110	
	1	0011	
437	0100	0011	0111

BCD calculations take longer to perform than the corresponding calculations in the binary system. The advantage of the BCD representation is that it avoids the need to convert the input from decimal to binary and the output from binary to decimal.

EXERCISES

1 Find the 2's complements of the following 8-bit binary numbers:

(a) 11010100_2 (b) 01101001_2

2 Write an algorithm in pseudocode for finding the 2's complement of a positive binary integer. (Assume that the integer is input as a sequence of bits: $b_1b_2b_3...b_n$.)

3 The 10's complement of a positive decimal integer n is $10^k - n$, where k is the number of digits in the decimal representation of n. It can be calculated in the following way:

1. All the zeros at the right-hand end of the number remain as zeros in the answer.
2. The rightmost non-zero digit d of the number is replaced by $10 - d$ in the answer.
3. Each other digit d is replaced by $9 - d$.

Find the 10's complements of the following decimal numbers using the rules given above, and check your answers by evaluating $10^k - n$ on a calculator:

(a) 3296 (b) 10350

4 Subtraction $a - b$ (with $a > b$) can be performed in the decimal system by adding the 10's complement of b to a and ignoring the leftmost 1 of the answer.

(a) Evaluate 39842 – 17674 using this method.

(b) Explain why the method works.

5 Find the 16-bit computer representations of the following integers:

(a) 29803 (b) –8155

6 The maximum unsigned integer on a CRAY-1 computer is approximately 2.8×10^{14}. How many bits are used to store unsigned integers on a CRAY-1? (Unsigned integers are non-negative integers stored without a sign bit.)

7 Verify that 3 + (–5) is evaluated correctly on the 4-bit computer.

8 Evaluate 7 – 6 on the 4-bit computer.

9 Express 1101110100.1001_2 in normalised binary exponential form. Hence find its 32-bit computer representation, assuming 8 bits are used for the characteristic, and the exponent bias is $2^7 - 1$.

10 Find the 32-bit computer representations of the following numbers, assuming 8 bits are used for the characteristic, and the exponent bias is $2^7 - 1$:

(a) 5894.376 (b) –0.0387

11 Repeat Exercise 9 for a 32-bit computer in which 12 bits are used for the characteristic, and the exponent bias is $2^{11} - 1$.

12 Find, in decimal form, the approximate range of positive real numbers that can be represented in 64 bits, where 11 bits are used for the characteristic, and the exponent bias is $2^{10} - 1$.

13 The algorithm below represents an attempt to print out a table of cubes of numbers from 0.1 to 10 in steps of 0.1. Explain the problem that might arise if the algorithm is implemented on a computer:

1. $x \leftarrow 0.0$
2. **Repeat**
   2.1. $x \leftarrow x + 0.1$
   2.2. $x\_cubed \leftarrow x^3$
   2.3. Output x, x_cubed
   until $x = 10.0$

Rewrite the algorithm so that the problem is avoided.

14 Perform the following decimal computations, assuming a precision of 4 decimal places in the significand:

(a) $0.8463 \times 10^6 + 0.7012 \times 10^8$

(b) $(0.3315 \times 10^{-5}) \times (0.2089 \times 10^9)$

(c) $(0.5160 \times 10^3) \div (0.1329 \times 10^4) - (0.3816 \times 10^0)$

15 The following statement is sometimes made about real number arithmetic: 'Precision is lost when two almost equal numbers are subtracted.' Explain this statement with reference to calculations with real numbers in exponential form.

16 Perform the following calculations in BCD arithmetic:

(a) 3711 + 5342 (b) 2859 + 3264