APPENDIX C

Representation of Integers

Conversion of Decimal Numbers to Binary

We know from ordinary arithmetic that the decimal (base 10) number 356 stands for 6 ones, 5 tens and 3 hundreds. As we go left, the value of each digit is increased by 10; we say the number is written in base 10. Also, to *write* a number in base 10, we must use the digits 0 to 9 (1 less than the base).

Similarly, to write a number in binary (base 2), we must use the digits 0 and 1. And, as we go left, the value of each digit is increased by 2. For example, the value of the binary number 10011 is given by (reading from the right)

1 one, 1 two, 0 fours, 0 eights and 1 sixteen, that is

```
1 + 2 + 0 + 0 + 16 = 19
```

As another example, consider the binary number 110101. We can work out the decimal equivalent using:

Binary digit	1	1	0	1	0	1
Value	32	16	8	4	2	1

Adding the values where the binary digits are 1s gives us

```
32 + 16 + 4 + 1 = 53
```

Thus, the decimal equivalent of 110101 is 53.

How do we do the reverse? Given a decimal number (43, say), what is the binary equivalent?

One method is to write 43 as a sum of powers of 2 (1, 2, 4, 8, etc.), thus:

```
43 = 32 + 8 + 2 + 1
```

So 43 consists of: 1 one, 1 two, 0 fours, 1 eight, 0 sixteens and 1 thirty-two. Hence the binary equivalent of 43 is 101011.

Note that we must be careful to put a 0 in those positions where a power of 2 is absent. In this example, these are the positions corresponding to 4 and 16.

Another method is to perform repeated divisions by 2 and save the remainders. The remainders, in order, form the binary equivalent from *right to left*. For example,

```
43/2    =    21    remainder 1
21/2    =    10    remainder 1
10/2    =    5     remainder 0
5/2     =    2     remainder 1
2/2     =    1     remainder 0
1/2     =    0     remainder 1
```

The remainders, from top to bottom, form the binary equivalent from right to left, thus: 101011.

Representation of Integers

An integer is a whole number—positive, negative or zero—for example, 25, -16, 0, 32767, -1. We have just seen how to convert a positive integer from decimal to binary. Also, the bigger the number, the more bits we need to represent it. For instance, we needed 5 bits for 19 but 6 bits for 43.

A computer uses a fixed number of *bits* (short for *bi*nary digi*ts*) for storing integers. So we use terms such as 8-bit integers, 16-bit integers and 32-bit integers. Whatever the size, the principles remain the same; the only difference is that with more bits we can store a wider range of numbers. In order to keep things simple, we will consider 4-bit integers. The problem, therefore, is:

Using 4 bits, how can we store positive and negative integers?

The most common method for storing integers on a computer is called *two's complement*. Before we look at this method, let us say what is meant by *one's complement*.

The one's complement of a binary number is obtained by changing 1's to 0's and 0's to 1's (this is called *inverting the bits*). For example, the 4-bit one's complement of 0110 is 1001.

Note that if we were asked for the 4-bit one's complement of 11, we would need to write 11 as 0011 and *then* invert the bits, giving 1100.

Also note that it is *wrong* to simply say "the one's complement of 011"; we *must* specify how many bits are involved by saying, for instance, "the 4-bit one's complement of 011". As a matter of interest, observe that:

- the 3-bit one's complement of 011 is 100, and
- the 4-bit one's complement of 011 is 1100.

Two's Complement

The two's complement of a binary number is obtained by adding 1 to its one's complement. Above, we saw that the 4-bit one's complement of 0110 is 1001. Therefore, the 4-bit two's complement of 0110 is:

```
1001 + 1 = 1010
```

As another example, the 4-bit two's complement of 011 is 1100 + 1 = 1101.

Suppose we want to store integers on a computer using 4 bits. With 4 bits, we can have 16 different *bit patterns*—from 0000 to 1111. Therefore, we can represent 16 different integers using 4 bits. We just need to decide which integer to represent by which bit pattern.

Of the 16 bit patterns, 8 begin with 0 and 8 begin with 1. We will let those which begin with 0 represent the equivalent positive integer; for example, 0101 will represent +5. The full list is as follows:

```
0000  represents  0
0001  represents  1
0010  represents  2
0011  represents  3
0100  represents  4
0101  represents  5
0110  represents  6
0111  represents  7
```

Note that the largest positive integer we can represent using 4 bits is 7. In general, the largest positive integer we can represent using *n* bits is $2^{n-1} - 1$.

Next, we need to decide which negative integers to represent by the bit patterns beginning with 1. In *two's complement representation*, we represent a negative integer by the two's complement of the bit pattern representing the corresponding positive integer. For example, to represent -5, we take the bit pattern for +5, that is, 0101, and find the two's complement, that is, 1011. So -5 is represented by 1011. Using this procedure, we find that:

```
-1  is represented by  1111
-2  is represented by  1110
-3  is represented by  1101
-4  is represented by  1100
-5  is represented by  1011
-6  is represented by  1010
-7  is represented by  1001
```

There is still one bit pattern, 1000, which has not been used. Since the 4-bit two's complement of 1000 is, indeed, 1000, we can let 1000 represent either +8 or -8. But since, in all other cases, positive numbers begin with 0 and negative numbers begin with 1, we let 1000 represent -8.

Hence, using two's complement, the largest negative number we can represent is 1 more than the largest positive number. In general, the *range* of integers we can store with *n* bits, using two's complement, is

$$-2^{n-1} \text{ to } +2^{n-1} - 1$$

For example, using 16 bits, the range of integers we can store is

$$-2^{15} \text{ to } +2^{15} - 1\text{, that is, } -32768 \text{ to } +32767$$