1 Data representation

In this chapter you will learn about:

- ★ number systems
  - how and why computers use binary to represent data
  - the denary, binary and hexadecimal number systems
  - converting numbers between denary, binary and hexadecimal
  - how and why hexadecimal is used for data representation
  - how to add two positive 8-bit numbers
  - overflow when performing binary addition
  - logical binary shifts on positive 8-bit integers
  - two's complement notation to represent positive and negative binary numbers
- ★ text, sound and images
  - how and why a computer represents text
  - the use of character sets including ASCII and Unicode
  - how and why a computer represents sound
  - sound sample rate and sample resolution
  - how and why a computer represents an image
  - the effects of the resolution and colour depth on images
- ★ data storage and compression
  - how data storage is measured
  - calculating the file size of an image and sound file
  - the purpose of and need for data compression
  - lossy and lossless compression.

This chapter considers the three key number systems used in computer science, namely binary, denary and hexadecimal. It also discusses how these number systems are used to measure the size of computer memories and storage devices, together with how sound and images can be represented digitally.

1.1 Number systems

1.1.1 Binary represents data

As you progress through this book you will begin to realise how complex computer systems really are. By the time you reach Chapter 10 you should have a better understanding of the fundamentals behind computers themselves and the software that controls them.

You will learn that any form of data needs to be converted into a binary format so that it can be processed by the computer.

However, no matter how complex the system, the basic building block in all computers is the binary number system. This system is chosen because it only consists of 1s and 0s. Since computers contain millions and millions of tiny 'switches', which must be in the ON or OFF position, they can be represented by the binary system. A switch in the ON position is represented by 1; a switch in the OFF position is represented by 0.

Switches used in a computer make use of logic gates (see Chapter 10) and are used to store and process data.

1.1.2 Binary, denary and hexadecimal systems

The binary system

We are all familiar with the denary number system which counts in multiples of 10. This gives us the well-known headings of units, 10s, 100s, 1000s, and so on:

(10^4)	(10^3)	(10^2)	(10^1)	(10^0)
10 000	**1000**	**100**	**10**	**1**
2	5	1	7	7

Denary uses ten separate digits, 0-9, to represent all values. Denary is known as a base 10 number system.

The **binary number system** is a base 2 number system. It is based on the number 2. Thus, only the two 'values' 0 and 1 can be used in this system to represent all values. Using the same method as denary, this gives the headings 2^0, 2^1, 2^2, 2^3, and so on. The typical headings for a binary number with eight digits would be:

(2^7)	(2^6)	(2^5)	(2^4)	(2^3)	(2^2)	(2^1)	(2^0)
128	**64**	**32**	**16**	**8**	**4**	**2**	**1**
1	1	1	0	1	1	1	0

A typical binary number would be: 11101110.

Converting from binary to denary

The conversion from binary to denary is a relatively straightforward process. Each time a 1-value appears in a binary number column, the column value (heading) is added to a total. This is best shown by three examples which use 8-bit, 12-bit and 16-bit binary numbers:

Example 1

Convert the binary number, 11101110, into a denary number.

128	64	32	16	8	4	2	1
1	1	1	0	1	1	1	0

The equivalent denary number is 128 + 64 + 32 + 8 + 4 + 2 = 238

Example 2

Convert the following binary number, 011110001011, into a denary number.

2048	1024	512	256	128	64	32	16	8	4	2	1
0	1	1	1	1	0	0	0	1	0	1	1

The equivalent denary number is 1024 + 512 + 256 + 128 + 8 + 2 + 1 = 1931

Example 3

Convert the following binary number, 0011000111100110, into a denary number.

32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
0	0	1	1	0	0	0	1	1	1	1	0	0	1	1	0

As with the two examples above, to convert this number to denary, each time a 1 appears in a column the column value is added to the total:

8192 + 4096 + 256 + 128 + 64 + 32 + 4 + 2 = 12 774

The same method can be used for a binary number of any size.

Activity 1.1

Convert the following binary numbers into denary:

a 0 0 1 1 0 0 1 1
b 0 1 1 1 1 1 1 1
c 1 0 0 1 1 0 0 1
d 0 1 1 1 0 1 0 0
e 1 1 1 1 1 1 1 1
f 0 0 0 0 1 1 1 1
g 1 0 0 0 1 1 1 1
h 1 0 1 1 0 0 1 1
i 0 1 1 1 0 0 0 0
j 1 1 1 0 1 1 1 0
k 0001 1110 0111
l 0101 0101 0100
m 1111 0000 1111
n 0111 1100 1000
o 0111 1111 1111
p 0111 1100 1111 0000
q 0011 1111 0000 1101
r 1100 0011 0011 1111
s 1000 1000 1000 1000
t 0111 1111 1111 1111

Converting from denary to binary

The conversion from denary numbers to binary numbers can be done in two different ways. The first method involves successive subtraction of powers of 2 (that is, 128, 64, 32, 16, and so on); whilst the second method involves successive division by 2 until the value "0" is reached. This is best shown by two examples:

Example 1

Consider the conversion of the denary number, 142, into binary:

Method 1

The denary number 142 is made up of 128 + 8 + 4 + 2 (that is, 142 – 128 = 14; 14 – 8 = 6; 6 – 4 = 2; 2 – 2 = 0; in each stage, subtract the largest possible power of 2 and keep doing this until the value 0 is reached. This will give us the following 8-bit binary number:

128	64	32	16	8	4	2	1
1	0	0	0	1	1	1	0

Method 2

This method involves successive division by 2. Start with the denary number, 142, and divide it by 2. Write the result of the division including the remainder (even if it is 0) under the 142 (that is, 142 ÷ 2 = 71 remainder 0); then divide again by 2 (that is, 71 ÷ 2 = 35 remainder 1) and keep dividing until the result is zero. Finally write down all the remainders in reverse order:

2	142				read the remainders from bottom to top to get the binary number: **1 0 0 0 1 1 1 0**
2	71	remainder:	0		
2	35	remainder:	1		
2	17	remainder:	1		
2	8	remainder:	1		
2	4	remainder:	0		
2	2	remainder:	0		
2	1	remainder:	0		
	0	remainder:	1		

▲ **Figure 1.1**

We end up with an 8-bit binary number which is the same as that found by Method 1.

Example 2

Consider the conversion of the denary number, 59, into binary:

Method 1

The denary number 59 is made up of 32 + 16 + 8 + 2 + 1 (that is, 59 – 32 = 27; 27 – 16 = 11; 11 – 8 = 3; 3 – 2 = 1; 1 – 1 = 0; in each stage, subtract the largest possible power of 2 and keep doing this until the value 0 is reached. This will give us the following 8-bit binary number:

128	64	32	16	8	4	2	1
0	0	1	1	1	0	1	1

Method 2

This method involves successive division by 2. Start with the denary number, 59, and divide it by 2. Write the result of the division including the remainder (even if it is 0) under the 59 (that is, 59 ÷ 2 = 29 remainder 1); then divide again by 2 (that is, 29 ÷ 2 = 14 remainder 1) and keep dividing until the result is zero. Finally write down all the remainders in reverse order:

2	59				write the remainders from bottom to top to get the binary number: **1 1 1 0 1 1**
2	29	remainder:	1		
2	14	remainder:	1		
2	7	remainder:	0		
2	3	remainder:	1		
2	1	remainder:	1		
	0	remainder:	1		

▲ **Figure 1.1b**

If we want to show this as an 8-bit binary number (as shown in Method 1), we now simply add two 0's from the left-hand side to give the result: 0 0 1 1 1 0 1 1. The two results from both methods clearly agree.

Both the above examples use 8-bit binary numbers. This third example shows how the method can still be used for any size of binary number; in this case a 16-bit binary number.

Example 3

Consider the conversion of the denary number, 35 000, into a 16-bit binary number:

Method 1

The denary number 35 000 is made up of 32 768 + 2048 + 128 + 32 + 16 + 8 (that is, 35 000 – 32 768 = 2232; 2232 – 2048 = 184; 184 – 128 = 56; 56 – 32 = 24; 24 – 16 = 8; 8 – 8 = 0; in each stage, subtract the largest possible power of 2 and keep doing this until the value 0 is reached. This will give us the following 16-bit binary number:

32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
1	0	0	0	1	0	0	0	1	0	1	1	1	0	0	0

Method 2

This method involves successive division by 2. Start with the denary number, 35000, and divide it by 2. Write the result of the division including the remainder (even if it is 0) under the 35 000 (that is, 35 000 ÷ 2 = 17 500 remainder 0); then divide again by 2 (that is, 17 500 ÷ 2 = 8750 remainder 0) and keep dividing until the result is zero. Finally write down all the remainders in reverse order:

2	35 000		
2	17 500	remainder:	0
2	8750	remainder:	0
2	4375	remainder:	0
2	2187	remainder:	1
2	1093	remainder:	1
2	546	remainder:	1
2	273	remainder:	0
2	136	remainder:	1
2	68	remainder	0
2	34	remainder	0
2	17	remainder	0
2	8	remainder	1
2	4	remainder	0
2	2	remainder	0
2	1	remainder	0
	0	remainder	1

read the remainder from bottom to top to get the binary number:

1 0 0 0 1 0 0 0 1 0
1 1 1 0 0 0

▲ **Figure 1.1c**

Activity 1.2

Convert the following denary numbers into binary (using both methods):

a 41	**d** 100	**g** 144	**j** 255	**m** 4095
b 67	**e** 111	**h** 189	**k** 33000	**n** 16400
c 86	**f** 127	**i** 200	**l** 888	**o** 62307

The hexadecimal system

The **hexadecimal number system** is very closely related to the binary system. Hexadecimal (sometimes referred to as simply 'hex') is a base 16 system and therefore needs to use 16 different 'digits' to represent each value.

Because it is a system based on 16 different digits, the numbers 0 to 9 and the letters A to F are used to represent each hexadecimal (hex) digit. A in hex = 10 in denary, B = 11, C = 12, D = 13, E = 14 and F = 15.

Using the same method as for denary and binary, this gives the headings 16^0, 16^1, 16^2, 16^3, and so on. The typical headings for a hexadecimal number with five digits would be:

(16^4)	(16^3)	(16^2)	(16^1)	(16^0)
65 536	**4096**	**256**	**16**	**1**
2	1	F	3	A

A typical example of hex is 2 1 F 3 A.

Since $16 = 2^4$ this means that FOUR binary digits are equivalent to each hexadecimal digit. The following table summarises the link between binary, hexadecimal and denary:

▼ **Table 1.1**

Binary value	Hexadecimal value	Denary value
0 0 0 0	0	0
0 0 0 1	1	1
0 0 1 0	2	2
0 0 1 1	3	3
0 1 0 0	4	4
0 1 0 1	5	5
0 1 1 0	6	6
0 1 1 1	7	7
1 0 0 0	8	8
1 0 0 1	9	9
1 0 1 0	A	10
1 0 1 1	B	11
1 1 0 0	C	12
1 1 0 1	D	13
1 1 1 0	E	14
1 1 1 1	F	15

Converting from binary to hexadecimal and from hexadecimal to binary

Converting from binary to hexadecimal is a fairly easy process. Starting from the right and moving left, split the binary number into groups of 4 bits. If the last group has less than 4 bits, then simply fill in with 0s from the left. Take each group of 4 bits and convert it into the equivalent hexadecimal digit using Table 1.1. Look at the following two examples to see how this works.

Example 1

1 0 1 1 1 1 1 0 0 0 0 1

First split this up into groups of 4 bits:

1 0 1 1 1 1 1 0 0 0 0 1

Then, using Table 1.1, find the equivalent hexadecimal digits:

B E 1

Example 2

1 0 0 0 0 1 1 1 1 1 1 1 0 1

First split this up into groups of 4 bits:

1 0 0 0 0 1 1 1 1 1 1 0 1

The left group only contains 2 bits, so add in two 0s:

0 0 1 0 0 0 0 1 1 1 1 1 1 0 1

Now use Table 1.1 to find the equivalent hexadecimal digits:

2 1 F D

Activity 1.3

Convert the following binary numbers into hexadecimal:

a 1 1 0 0 0 0 1 1
b 1 1 1 1 0 1 1 1
c 1 0 0 1 1 1 1 1 1 1
d 1 0 0 1 1 1 0 1 1 1 0
e 0 0 0 1 1 1 1 0 0 0 0 1
f 1 0 0 0 1 0 0 1 1 1 1 0
g 0 0 1 0 0 1 1 1 1 1 1 1 0
h 0 1 1 1 0 1 0 0 1 1 1 0 0
i 1 1 1 1 1 1 1 1 0 1 1 1 1 1 0 1
j 0 0 1 1 0 0 1 1 1 1 0 1 0 1 1 1 0

Converting from hexadecimal to binary is also very straightforward. Using the data in Table 1.1, simply take each hexadecimal digit and write down the 4-bit code which corresponds to the digit.

Example 3

4	5	A

Using Table 1.1, find the 4 bit code for each digit:

0 1 0 0	0 1 0 1	1 0 1 0

Put the groups together to form the binary number:

0 1 0 0 0 1 0 1 1 0 1 0

Example 4

B	F	0	8

Again just use Table 1.1:

1 0 1 1	1 1 1 1	0 0 0 0	1 0 0 0

Then put all the digits together:

1 0 1 1 1 1 1 1 0 0 0 0 1 0 0 0

Activity 1.4

Convert the following hexadecimal numbers into binary:

a 6 C
b 5 9
c A A
d A 0 0
e 4 0 E
f B A 6
g 9 C C
h 4 0 A A
i D A 4 7
j 1 A B 0

Converting from hexadecimal to denary and from denary to hexadecimal

To ***convert hexadecimal numbers into denary*** involves the value headings of each hexadecimal digit; that is, 4096, 256, 16 and 1.

Take each of the hexadecimal digits and multiply it by the heading values. Add all the resultant totals together to give the denary number. Remember that the hex digits A → F need to be first converted to the values 10 → 15 before carrying out the multiplication. This is best shown by two examples:

Example 1

Convert the hexadecimal number, 4 5 A, into denary.

First of all we have to multiply each hex digit by its heading value:

256	16	1	
4	5	A	
(4 × 256 = 1024)	(5 × 16 = 80)	(10 × 1 = 10)	(NOTE: A = 10)

Then we have to add the three totals together (1024 + 80 + 10) to give the denary number:

1 1 1 4

Example 2

Convert the hexadecimal number, C 8 F, into denary.

First of all we have to multiply each hex digit by its heading value:

256	16	1	
C	8	F	
(12 × 256 = 3072)	(8 × 16 = 128)	(15 × 1 = 15)	(NOTE: C = 12, F = 15)

Then we have to add the three totals together (3072 + 128 + 15) to give the denary number:

3 2 1 5

Activity 1.5

Convert the following hexadecimal numbers into denary:

a 6 B
b 9 C
c 4 A
d F F
e 1 F F
f A 0 1
g B B 4
h C A 8
i 1 2 A E
j A D 8 9

To ***convert from denary to hexadecimal*** involves successive division by 16 until the value "0" is reached. This is best shown by two examples:

Example 1

Convert the denary number, 2004, into hexadecimal.

This method involves successive division by 16 until the value 0 is reached. We start by dividing the number 2004 by 16. The result of the division including the remainder (even if it is 0) is written under 2004 and then further divisions by 16 are carried out (that is, 2004 ÷ 16 = 125 remainder 4; 125 ÷ 16 = 7 remainder 13; 7 ÷ 16 = 0 remainder 7). The hexadecimal number is obtained from the remainders written in reverse order:

16	2004				write the remainders from bottom to top to get the hexadecimal number:
16	125	remainder:	4		
16	7	remainder:	13		7 D 4 (D=13)
	0	remainder:	7		

▲ **Figure 1.2a**

Example 2

Convert the denary number, 8463, into hexadecimal.

We start by dividing the number 8463 by 16. The result of the division including the remainder (even if it is 0) is written under 8463 and then further divisions by 16 are carried out (that is, 8463 ÷ 16 = 528 remainder 15; 528 ÷ 16 = 33 remainder 0; 33 ÷ 16 = 2 remainder 1; 2 ÷ 16 = 0 remainder 2). The hexadecimal number is obtained from the remainders written in reverse order:

16	8463				read the remainder from bottom to top to get the hexadecimal number:
16	528	remainder:	15		
16	33	remainder:	0		2 1 0 F (F=15)
16	2	remainder:	1		
	0	remainder:	2		

▲ **Figure 1.2b**

Activity 1.6

Convert the following denary numbers into hexadecimal:

a	98	f	1000
b	227	g	2634
c	490	h	3743
d	511	i	4007
e	826	j	5000

1.1.3 Use of the hexadecimal system

As we have seen, a computer can only work with binary data. Whilst computer scientists can work with binary, they find hexadecimal to be more convenient to use. This is because one hex digit represents four binary digits. A complex binary number, such as 1101001010101111 can be written in hex as D2AF. The hex number is far easier for humans to remember, copy and work with. This section reviews four uses of the hexadecimal system:

- error codes
- MAC addresses
- IPv6 addresses
- HTML colour codes

The information in this section gives the reader sufficient grounding in each topic at this level. Further material can be found by searching the internet, but be careful that you don't go off at a tangent.

Error codes

Error codes are often shown as hexadecimal values. These numbers refer to the memory location of the error and are usually automatically generated by the computer. The programmer needs to know how to interpret the hexadecimal error codes. Examples of error codes from a Windows system are shown below:

```
e HexErrorCode ErrorDescription
- ------------ ----------------
0 0x0          Success
1 0x1          Incorrect function.
2 0x2          The system cannot find the file specified.
3 0x3          The system cannot find the path specified.
4 0x4          The system cannot open the file.
5 0x5          Access is denied.
6 0x6          The handle is invalid.
7 0x7          The storage control blocks were destroyed.
8 0x8          Not enough storage is available to process this command.
9 0x9          The storage control block address is invalid.
0 0xa          Unit test error string
1 0xb          An attempt was made to load a program with an incorrect format.
2 0xc          The access code is invalid.
3 0xd          The data is invalid.
4 0xe          Not enough storage is available to complete this operation.
5 0xf          The system cannot find the drive specified.
6 0x10         The directory cannot be removed.
7 0x11         The system cannot move the file to a different disk drive.
8 0x12         There are no more files.
9 0x13         The media is write protected.
0 0x14         The system cannot find the device specified.
1 0x15         The device is not ready.
2 0x16         The device does not recognize the command.
3 0x17         Data error (cyclic redundancy check).
4 0x18         The program issued a command but the command length is incorrect.
5 0x19         The drive cannot locate a specific area or track on the disk.
6 0x1a         The specified disk or diskette cannot be accessed.
7 0x1b         The drive cannot find the sector requested.
8 0x1c         The printer is out of paper.
9 0x1d         The system cannot write to the specified device.
0 0x1e         The system cannot read from the specified device.
1 0x1f         A device attached to the system is not functioning.
2 0x20         The process cannot access the file because it is being used by another process.
3 0x21         The process cannot access the file because another process has locked a portion of the file.
4 0x22         The wrong diskette is in the drive....
6 0x24         Too many files opened for sharing.
8 0x26         Reached the end of the file.
9 0x27         The disk is full.
0 0x32         The request is not supported.
```

▲ **Figure 1.3** Example of error codes

Find out more

Another method used to trace errors during program development is to use memory dumps, where the memory contents are printed out either on screen or using a printer. Find examples of memory dumps and find out why these are a very useful tool for program developers.

Media Access Control (MAC) addresses

Media Access Control (MAC) address refers to a number which uniquely identifies a device on a network. The MAC address refers to the network interface card (NIC) which is part of the device. The MAC address is rarely changed so that a particular device can always be identified no matter where it is.

A MAC address is usually made up of 48 bits which are shown as 6 groups of two hexadecimal digits (although 64-bit addresses also exist):

NN – NN – NN – DD – DD – DD

or

NN:NN:NN:DD:DD:DD

where the first half (NN – NN – NN) is the identity number of the manufacturer of the device and the second half (DD – DD – DD) is the serial number of the device. For example:

00 – 1C – B3 – 4F – 25 – FE is the MAC address of a device produced by the Apple Corporation (code: 001CB3) with a serial number of: 4F25FE. Very often lowercase hexadecimal letters are used in the MAC address: 00-1c-b3-4f-25-fe. Other manufacturer identification numbers include:

00 – 14 – 22 which identifies devices made by Dell

00 – 40 – 96 which identifies devices made by Cisco

00 – a0 – c9 which identifies devices made by Intel, and so on.

Link

Refer to Chapter 3 for more detail on MAC addresses.

Find out more

Try to find the MAC addresses of some of your own devices (e.g. mobile phone and tablet) and those found in the school.

Internet Protocol (IP) addresses

Each device connected to a network is given an address known as the **Internet Protocol (IP) address**. An IPv4 address is a 32-bit number written in denary or hexadecimal form: e.g. 109.108.158.1 (or 77.76.9e.01 in hex). IPv4 has recently been improved upon by the adoption of IPv6. An IPv6 address is a 128-bit number broken down into 16-bit chunks, represented by a hexadecimal number. For example:

a8fb:7a88:fff0:0fff:3d21:2085:66fb:f0fa

Note IPv6 uses a colon (:) rather than a decimal point (.) as used in IPv4.

Link

Refer to Chapter 3 for more detail on IP addresses.

Find out more

Try to find the IPv4 and IPv6 addresses of some of your own devices (e.g. mobile phone and tablet) and those found in the school.

HyperText Mark-up Language (HTML) colour codes

HyperText Mark-up Language (HTML) is used when writing and developing web pages. HTML isn't a programming language but is simply a mark-up language. A mark-up language is used in the processing, definition and presentation of text (for example, specifying the colour of the text).

HTML uses `<tags>` which are used to bracket a piece of text for example, <h1> and </h1> surround a top-level heading. Whatever is between the two tags has been defined as heading level 1. Here is a short example of HTML code:

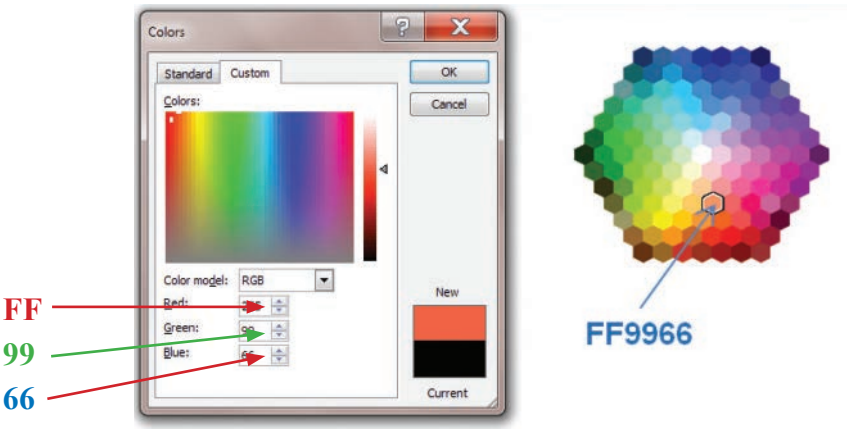
```
<h1 style="color:#FF0000;">This is a red heading</h1>

<h2 style="color:#00FF00;">This is a green heading</h2>

<h3 style="color:#0000FF;">This is a blue heading</h3>
```

▲ **Figure 1.4**

HTML is often used to represent colours of text on the computer screen. All colours can be made up of different combinations of the three primary colours (red, green and blue). The different intensity of each colour (red, green and blue) is determined by its hexadecimal value. This means different hexadecimal values represent different colours. For example:

- # FF 00 00 represents primary colour red
- # 00 FF 00 represents primary colour green
- # 00 00 FF represents primary colour blue
- # FF 00 FF represents fuchsia
- # FF 80 00 represents orange
- # B1 89 04 represents a tan colour,

and so on producing almost any colour the user wants. The following diagrams show the various colours that can be selected by altering the hex 'intensity' of red, green and blue primary colours. The colour ***'FF9966'*** has been chosen as an example:



▲ **Figure 1.5** Examples of HTML hex colour codes

The # symbol always precedes hexadecimal values in HTML code. The colour codes are always six hexadecimal digits representing the red, green and blue components. There are a possible 256 values for red, 256 values for green and 256 values for blue giving a total of 256 × 256 × 256 (i.e. 16 777 216) possible colours.

Activity 1.7

1 Using software on your computer (for example, text colour option in *Word*), find out what colours would be represented by the following RGB denary value combinations:

a	Red	53	**b**	Red	201	**c**	Red	12
	Green	55		Green	122		Green	111
	Blue	139		Blue	204		Blue	81

2 Convert each of the above denary numbers into hexadecimal.

1.1.4 Addition of binary numbers

This section will look at the addition of two 8-bit positive binary numbers.

Note the following key facts when carrying out ***addition*** of two binary digits:

binary addition	carry	sum
0+0	0	0
0+1	0	1
1+0	0	1
1+1	1	0

This can then be extended to consider the addition of three binary digits:

binary digit	carry	sum
0+0+0	0	0
0+0+1	0	1
0+1+0	0	1
0+1+1	1	0
1+0+0	0	1
1+0+1	1	0
1+1+0	1	0
1+1+1	1	1

For comparison: if we add 7 and 9 in denary the result is: carry = 1 and sum = 6; if we add 7, 9 and 8 the result is: carry = 2 and sum = 4, and so on.

Advice

Here's a quick recap on the role of carry and sum. If we want to add the numbers 97 and 64 in decimal, we:

- add the numbers in the right hand column first
- if the sum is greater than 9 then we carry a value to the next column
- we continue moving left, adding any carry values to each column until we are finished.

For instance:

```
  9 7
+ 6 4
  1 1  CARRY VALUES
1 6 1  SUM VALUES
```

Adding in binary follows the same rules except that we carry whenever the sum is greater than 1.

Example 1

Add 00100111 + 01001010

We will set this out showing carry and sum values:

```
  0 0 1 0 0 1 1 1
+ 0 1 0 0 1 0 1 0
  -----------------
    1 1 1          <- carry values
  -----------------
  0 1 1 1 0 0 0 1  <- sum values
```

column 1: 1 + 0 = 1 no carry
column 2: 1 + 1 = 0 carry 1
column 3: 1 + 0 + 1 = 0 carry 1
column 4: 0 + 1 + 1 = 0 carry 1
column 5: 0 + 0 + 1 = 1 no carry
column 6: 1 + 0 = 1 no carry
column 7: 0 + 1 = 1 no carry
column 8: 0 + 0 = 0 no carry

Answer: **01110001**

Example 2

a Convert 126 and 62 into binary.
b Add the two binary values in part a and check the result matches the addition of the two denary numbers

a 126 = 0 1 1 1 1 1 1 0 and 62 = 0 0 1 1 1 1 1 0

b

```
  0 1 1 1 1 1 1 0
+ 0 0 1 1 1 1 1 0
  -----------------
  1 1 1 1 1 1      <- carry values
  -----------------
  1 0 1 1 1 1 0 0  <- sum values
```

column 1: 0 + 0 = 0 no carry
column 2: 1 + 1 = 0 carry 1
column 3: 1 + 1 + 1 = 1 carry 1
column 4: 1 + 1 + 1 = 1 carry 1
column 5: 1 + 1 + 1 = 1 carry 1
column 6: 1 + 1 + 1 = 1 carry 1
column 7: 1 + 0 + 1 = 0 carry 1
column 8: 0 + 0 + 1 = 1 no carry

Answer: **10111100**

1 0 1 1 1 1 0 0 has the equivalent denary value of 128 + 32 + 16 + 8 + 4 = 188 which is the same as 126 + 62.

Activity 1.8

Carry out the following binary additions:

a 0 0 0 1 1 1 0 1 + 0 1 1 0 0 1 1 0
b 0 0 1 0 0 1 1 1 + 0 0 1 1 1 1 1 1
c 0 0 1 0 1 1 1 0 + 0 1 0 0 1 1 0 1
d 0 1 1 1 0 1 1 1 + 0 0 1 1 1 1 1 1
e 0 0 1 1 1 1 0 0 + 0 0 1 1 0 0 1 1
f 0 0 1 1 1 1 0 0 + 0 1 1 1 1 0 1 1
g 0 0 1 1 1 1 1 1 + 0 0 1 1 1 1 1 1
h 0 0 1 1 0 0 0 1 + 0 0 1 1 1 1 1 1
i 0 1 1 1 1 1 1 1 + 0 1 1 1 1 1 1 1
j 1 0 1 0 0 0 1 0 + 0 0 1 1 1 0 1 1

Activity 1.9

Convert the following denary numbers into binary and then carry out the binary addition of the two numbers and check your answer against the equivalent denary sum:

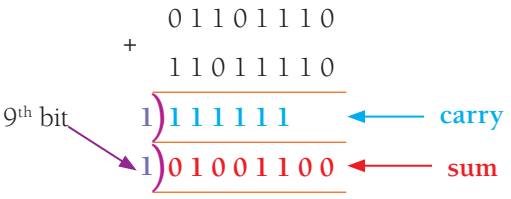
a 98 + 15
b 29 + 88
c 49 + 100
d 51 + 171
e 82 + 69
f 100 + 140
g 19 + 139
h 203 + 30
i 66 + 166
j 211 + 35

Overflow

Now consider the following example:

Example 3

Add 0 1 1 0 1 1 1 0 and 1 1 0 1 1 1 1 0 (using 8 bits)



This addition has generated a 9th bit. The 8 bits of the answer are 0 1 0 0 1 1 0 0 – this gives the denary value (64 + 8 + 4) of 76 which is incorrect because the denary value of the addition is 110 + 222 = 332.

The maximum denary value of an 8-bit binary number is 255 (which is $2^8 - 1$). The generation of a 9th bit is a clear indication that the sum has exceeded this value. This is known as an **overflow error** and in this case is an indication that a number is too big to be stored in the computer using 8 bits.

The greater the number of bits which can be used to represent a number then the larger the number that can be stored. For example, a 16-bit register would allow a maximum denary value of 65 535 (i.e. $2^{16} - 1$) to be stored, a 32-bit register would allow a maximum denary value of 4 294 967 295 (i.e. $2^{32} - 1$), and so on.

Activity 1.10

1 Convert the following pairs of denary numbers to 8-bit binary numbers and then add the binary numbers. Comment on your answers in each case:

   a 89 + 175 **b** 168 + 99 **c** 88 + 215

2 Carry out the following 16-bit binary additions and comment on your answers:

   a 0111 1111 1111 0001 + 0101 1111 0011 1001

   b 1110 1110 0000 1011 + 1111 1101 1101 1001

1.1.5 Logical binary shifts

Computers can carry out a **logical shift** on a sequence of binary numbers. The logical shift means moving the binary number to the ***left*** or to the ***right***. Each shift ***left*** is equivalent to ***multiplying*** the binary number by 2 and each shift ***right*** is equivalent to ***dividing*** the binary number by 2.

As bits are shifted, any empty positions are replaced with a zero – see examples below. There is clearly a limit to the number of shifts which can be carried out if the binary number is stored in an 8-bit register. Eventually after a number of shifts the register would only contain zeros. For example, if we shift 01110000 (denary value 112) five places left (the equivalent to multiplying by 2^5, i.e. 32), in an 8-bit register we would end up with 00000000. This makes it seem as though 112 × 32 = 0! This would result in the generation of an error message.

Example 1

The denary number 21 is 00010101 in binary. If we put this into an 8-bit register:

128	64	32	16	8	4	2	1
0	0	0	1	0	1	0	1

The left-most bit is often referred to as the MOST SIGNIFICANT BIT

If we now shift the bits in this register one place to the left, we obtain:

128	64	32	16	8	4	2	1
0	0	1	0	1	0	1	0

Note how the empty right-most bit position is now filled with a 0

The left-most bit is now lost following a left shift

The value of the binary bits is now 21×2^1 i.e. 42. We can see this is correct if we calculate the denary value of the new binary number 101010 (i.e. 32 + 8 + 2).

Suppose we now shift the original number two places left:

128	64	32	16	8	4	2	1
0	1	0	1	0	1	0	0

The binary number 1010100 is 84 in denary – this is 21×2^2.

And now suppose we shift the original number three places left:

1	0	1	0	1	0	0	0

The binary number 10101000 is 168 in denary – this is 21×2^3.

So, let us consider what happens if we shift the original binary number 00010101 four places left:

128	64	32	16	8	4	2	1
0	1	0	1	0	0	0	0

Losing 1 bit following a shift operation will cause an error

The left-most 1-bit has been lost. In our 8-bit register the result of 21×2^4 is 80 which is clearly incorrect. This error is because we have exceeded the maximum number of left shifts possible using this register.

Example 2

The denary number 200 is 11001000 in binary. Putting this into an 8-bit register gives:

128	64	32	16	8	4	2	1
1	1	0	0	1	0	0	0

The right-most bit is often referred to as the LEAST SIGNIFICANT BIT

If we now shift the bits in this register one place to the right:

128	64	32	16	8	4	2	1
0	1	1	0	0	1	0	0

Note how the left-most bit position is now filled with a 0

The value of the binary bits is now 200 ÷ 2^1 i.e. 100. We can see this is correct by converting the new binary number 01100100 to denary (64 + 32 + 4).

Suppose we now shift the original number two places to the right:

128	64	32	16	8	4	2	1
0	0	1	1	0	0	1	0

The binary number 00110010 is 50 in denary – this is 200 ÷ 2^2.

And suppose we now shift the original number three places to the right:

Notice the 1-bit from the right-most bit position is now lost causing an error

128	64	32	16	8	4	2	1
0	0	0	1	1	0	0	1

The binary number 00011001 is 25 in denary – this is 200 ÷ 2^3.

Now let us consider what happens if we shift four places right:

128	64	32	16	8	4	2	1
0	0	0	0	1	1	0	0

The right-most 1-bit has been lost. In our 8-bit register the result of 200 ÷ 2^4 is 12, which is clearly incorrect. This error is because we have therefore exceeded the maximum number of right shifts possible using this 8-bit register.

Example 3

a Write 24 as an 8-bit register.
b Show the result of a logical shift 2 places to the left.
c Show the result of a logical shift 3 places to the right.

a

128	64	32	16	8	4	2	1
0	0	0	1	1	0	0	0

b

128	64	32	16	8	4	2	1
0	1	1	0	0	0	0	0

⇦ 24 × 2^2 = 96

c

128	64	32	16	8	4	2	1
0	0	0	0	0	0	1	1

⇦ 24 ÷ 2^3 = 3

Example 4

a Convert 19 and 17 into binary.
b Carry out the binary addition of the two numbers.
c Shift your result from part **b** two places left and comment on the result.
d Shift your result from part **b** three places right and comment on the result.

a

128	64	32	16	8	4	2	1	
0	0	0	1	0	0	1	1	19
0	0	0	1	0	0	0	1	17

b

```
  00010011
+ 00010001
  ---------
    1  11     <- carry
  ---------
  00100100     <- sum
```

c

128	64	32	16	8	4	2	1
1	0	0	1	0	0	0	0

d

128	64	32	16	8	4	2	1
0	0	0	0	0	1	0	0

In **c** the result is $36 \times 2^2 = 144$ (which is correct).

In **d** the result of the right shift gives a value of 4, which is incorrect since $36 \div 2^3$ is not 4; therefore, the number of possible right shifts has been exceeded. You can also see that a 1 has been lost from the original binary number, which is another sign that there have been too many right shifts.

Activity 1.11

1 a Write down the denary value of the following binary number.

0	1	1	0	1	0	0	0

b Shift the binary number three places to the right and comment on your result.

c Write down the denary value of the following binary number.

0	0	0	0	1	1	1	1

d Shift the binary number four places to the left and comment on your result.

2 a Convert 29 and 51 to 8-bit binary numbers.

b Add the two binary numbers in part **a**.

c Shift the result in part **b** three places to the right.

d Convert 75 to an 8-bit binary number.

e Add the two binary numbers from parts **c** and **d**.

f Shift your result from part **e** one place to the left.

1.1.6 Two's complement (binary numbers)

Up until now, we have assumed all binary numbers are positive integers. To allow the possibility of representing negative integers we make use of **two's complement**. In this section we will again assume 8-bit registers are being used. Only one minor change to the binary headings needs to be introduced here:

−128	64	32	16	8	4	2	1

In two's complement the left-most bit is changed to a negative value. For instance, for an 8-bit number, the value 128 is now changed to –128, but all the other headings remain the same. This means the new range of possible numbers is: –128 (10000000) to +127 (01111111).

It is important to realise when applying two's complement to a binary number that the left-most bit always determines the sign of the binary number. A 1-value in the left-most bit indicates a negative number and a 0-value in the left-most bit indicates a positive number (for example, 00110011 represents 51 and 11001111 represents –49).

Writing positive binary numbers in two's complement format

Example 1

The following two examples show how we can write the following positive binary numbers in the two's complement format 19 and 4:

–128	64	32	16	8	4	2	1
0	0	0	1	0	0	1	1
0	0	0	0	0	1	0	0

As you will notice, for positive binary numbers, it is no different to what was done in Section 1.1.2.

Converting positive denary numbers to binary numbers in the two's complement format

If we wish to convert a positive denary number to the two's complement format, we do exactly the same as in Section 1.1.2:

Example 2

Convert **a** 38 **b** 125 to 8-bit binary numbers using the two's complement format.

a Since this number is positive, we must have a zero in the –128 column. It is then a simple case of putting 1-values into their correct positions to make up the value of 38:

–128	64	32	16	8	4	2	1
0	0	1	0	0	1	1	0

b Again, since this is a positive number, we must have a zero in the –128 column. As in part **a**, we then place 1-values in the appropriate columns to make up the value of 125:

–128	64	32	16	8	4	2	1
0	1	1	1	1	1	0	1

Converting positive binary numbers in the two's complement format to positive denary numbers

Example 3

Convert 01101110 in two's complement binary into denary:

−128	64	32	16	8	4	2	1
0	1	1	0	1	1	1	0

As in Section 1.1.2, each time a 1 appears in a column, the column value is added to the total. For example, the binary number (01101110) above has the following denary value: 64 + 32 + 8 + 4 +2 = 110.

Example 4

Convert 00111111 in two's complement binary into denary:

−128	64	32	16	8	4	2	1
0	0	1	1	1	1	1	1

As above, each time a 1 appears in a column, the column value is added to the total. For example, the binary number (00111111) above has the following denary value: 32 + 16 + 8 + 4 +2 + 1 = 63.

Activity 1.12

1 Convert the following positive denary numbers into 8-bit binary numbers in the two's complement format:

a	39	**c**	88	**e**	111	**g**	77	**i**	49
b	66	**d**	102	**f**	125	**h**	20	**j**	56

2 Convert the following binary numbers (written in two's complement format) into positive denary numbers:

	−128	64	32	16	8	4	2	1
a	0	1	0	1	0	1	0	1
b	0	0	1	1	0	0	1	1
c	0	1	0	0	1	1	0	0
d	0	1	1	1	1	1	1	0
e	0	0	0	0	1	1	1	1
f	0	1	1	1	1	1	0	1
g	0	1	0	0	0	0	0	1
h	0	0	0	1	1	1	1	0
i	0	1	1	1	0	0	0	1
j	0	1	1	1	1	0	0	0

Writing negative binary numbers in two's complement format and converting to denary

Example 1

The following three examples show how we can write negative binary numbers in the two's complement format:

–128	64	32	16	8	4	2	1
1	0	0	1	0	0	1	1

By following our normal rules, each time a 1 appears in a column, the column value is added to the total. So, we can see that in denary this is: –128 + 16 + 2 + 1 = –109.

–128	64	32	16	8	4	2	1
1	1	1	0	0	1	0	0

Similarly, in denary this number is –128 + 64 + 32 + 4 = –28.

–128	64	32	16	8	4	2	1
1	1	1	1	0	1	0	1

This number is equivalent to –128 + 64 + 32 + 16 + 4 + 1 = –11.

Note that a two's complement number with a 1-value in the -128 column must represent a negative binary number.

Converting negative denary numbers into binary numbers in two's complement format

Consider the number +67 in 8-bit (two's complement) binary format:

–128	64	32	16	8	4	2	1
0	1	0	0	0	0	1	1

Method 1

Now let's consider the number -67. One method of finding the binary equivalent to -67 is to simply put 1s in their correct places:

–128	64	32	16	8	4	2	1
1	0	1	1	1	1	0	1

–128 + 32 + 16 + 8 + 4 + 1 = –67

Method 2

However, looking at the two binary numbers above, there is another possible way to find the binary representation of a negative denary number:

first write the number as a positive binary value – in this case 67:	0 1 0 0 0 0 1 1
we then invert each binary value, which means swap the 1s and 0s around:	1 0 1 1 1 1 0 0
then add 1 to that number:	1
this gives us the binary for –67:	1 0 1 1 1 1 0 1

Example 2

Convert –79 into an 8-bit binary number using two's complement format.

Method 1

As it is a negative number, we need a 1-value in the –128 column.

–79 is the same as –128 + 49

We can make up 49 from 32 + 16 + 1; giving:

–128	64	32	16	8	4	2	1
1	0	1	1	0	0	0	1

Method 2

write 79 in binary:	0 1 0 0 1 1 1 1
invert the binary digits:	1 0 1 1 0 0 0 0
add 1 to the inverted number	1
thus giving –79:	1 0 1 1 0 0 0 1

–128	64	32	16	8	4	2	1
1	0	1	1	0	0	0	1

It is a good idea to practise both methods.

When applying two's complement, it isn't always necessary for a binary number to have 8 bits:

Example 3

The following 4-bit binary number represents denary number 6:

–8	4	2	1
0	1	1	0

Applying two's complement (1 0 0 1 + 1) would give:

–8	4	2	1
1	0	1	0

in other words: –6

Example 4

The following 12-bit binary number represents denary number 1676:

–2048	1024	512	256	128	64	32	16	8	4	2	1
0	1	1	0	1	0	0	0	1	1	0	0

Applying two's complement (1 0 0 1 0 1 1 1 0 0 1 1 + 1) would give:

–2048	1024	512	256	128	64	32	16	8	4	2	1
1	0	0	1	0	1	1	1	0	1	0	0

In other words: –1676

Activity 1.13

Convert the following negative denary numbers into binary numbers using the two's complement format:

a –18 **b** –31 **c** –47 **d** –63 **e** –88 **f** –92 **g** –100 **h** –1 **i** –16 **j** –127

Activity 1.14

Convert the following negative binary numbers (written in two's complement format) into negative denary numbers:

a	1	1	0	0	1	1	0	1
b	1	0	1	1	1	1	1	0
c	1	1	1	0	1	1	1	1
d	1	0	0	0	0	1	1	1
e	1	0	1	0	0	0	0	0
f	1	1	1	1	1	0	0	1
g	1	0	1	0	1	1	1	1
h	1	1	1	1	1	1	1	1
i	1	0	0	0	0	0	0	1
j	1	1	1	1	0	1	1	0

1.2 Text, sound and images

1.2.1 Character sets – ASCII code and Unicode

The **ASCII code** system (American Standard Code for Information Interchange) was set up in 1963 for use in communication systems and computer systems. A newer version of the code was published in 1986. The standard ASCII code **character set** consists of 7-bit codes (0 to 127 in denary or 00 to 7F in