Information representation

Learning Objectives:

- ☐ Understand the binary, decimal and hexadecimal number systems and Binary Coded Decimal (BCD)
- ☐ Understand the one's complement and two's complement representation used for positive and negative integers
- ☐ Perform binary addition and subtraction of integers
- ☐ Use the terms for the naming of large binary and large decimal numbers
- ☐ Understand how characters are represented using:
  - The ASCII system, including the extended character set
  - Unicode
- ☐ Bitmaps
  - Understand how data in a bitmap is encoded and the different bitmap file formats
  - Calculate a bitmap image file size
  - Understand the limitations of a bitmap image
- ☐ Vector graphics
  - Understand how a drawing is constructed by selecting shapes or objects from libraries
- ☐ Describe applications where bitmaps or vector graphics would be used
- ☐ Sound
  - Understand how sound data is encoded
  - Understand the effect of sampling rate and sampling resolution
- ☐ Understand the need for compression techniques for all of the above media and text files, and the terms run-length encoding (RLE), 'lossy' and 'lossless'

1.01 Number systems

Humans use the base 10 number system.

Computers use digital data in the form of electrical signals. Digital data is represented as **bits**.

Data values, such as numbers and characters, need more than a single bit. Most PCs store data as 8-bit patterns called **bytes**.

Any number system is founded on a **base**, for example, denary is base 10. The largest number used in any position will be one less than the number base. Each position has a **place value** and this depends on the number base.

Binary number system

Binary is the 'base 2' number system.

This is summarised in the following table:

System	Base	Possible digits	Place values
Binary	2	0, 1	etc. 2^3 2^2 2^1 Unit 1 1 0 0

Table 1.01 Binary – base 2.

To convert the binary number 1100 to a denary number, you write it as:

$(1 \times 8) + (1 \times 4) + (0 \times 2) + (0 \times 1) = 12$.

You can add a suffix to the binary number to make it clear that it is binary, i.e. 1100_2.

Hexadecimal number system

Hexadecimal is the 'base 16' number system.

System	Base	Possible digits	Place values
Hexadecimal	16	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ~~10~~, ~~11~~, ~~12~~, ~~13~~, ~~14~~, ~~15~~ A, B, C, D, E, F	etc. 16^3 16^2 16^1 Units 2 A C

Table 1.02 Hexadecimal – base 16.

The digits allowed in base 16 extend past 9, so you replace 10, 11, 12, 13, 14 and 15 with a letter. For hexadecimal you use the characters A to F as shown in Table **1.2**.

To convert the hexadecimal number $2AC_{16}$ to a denary number, you write it as:

$(2 \times 256) + (A \times 16) + (C \times 1) = (2 \times 256) + (10 \times 16) + (12 \times 1) = 512 + 160 + 12 = 684$.

Hexadecimal is a shorthand representation for a binary code. Applications where hexadecimal is used include:

- assembly language programming to represent instructions in the program code
- graphics packages to represent colour codes
- program code to represent characters.

Conversion between different bases

Worked example 1.1

Convert 69_{10} into binary.

Table **1.3** shows you how to divide the number repeatedly by 2 and record the remainders. You find the answer, 1000101_2 by collecting these remainders, starting *at the bottom*. Try to remember this, as it is not obvious.

	÷2	remainder
69	34	1
34	17	0
17	8	1
8	4	0
4	2	0
2	1	0
1	0	1
		$= 1000101_2$

Table 1.3 – Convert denary to binary.

Worked example 1.2

Convert 1000 1100$_2$ into denary.

You need to use the place values (2^0, 2^1, 2^2 etc).

	1	0	0	0	1	1	0	0
=	1×2^7	$+ 0 \times 2^6$	$+ 0 \times 2^5$	$+ 0 \times 2^4$	$+ 1 \times 2^3$	$+ 1 \times 2^2$	$+ 0 \times 2^1$	$+ 0 \times 2^0$
=	128	+ 0	+ 0	+ 0	+ 8	+ 4	+ 0	+ 0
=	140							

Progress check A

Convert these numbers to denary:

a 0100 0001

b 1010 1010

c 1111 1111

Progress check B

Write the 8-bit binary for the integers 3_{10}, 31_{10} and 96_{10}.

You might need to add 1, 2 or 3 zeros to the left side of the binary number so that each **nibble** is complete. Hence, you will write 10101 as 0001 0101.

Conversion between binary and hexadecimal

One approach would be to convert the binary number into denary first; but there is a more direct way:

Worked example 1.3

Convert `0111110101011111`$_2$ into hexadecimal.

Divide the binary number into nibbles:

`0111  1101  0101  1111`

Write the denary for each nibble:

`7  13  5  15`

Convert to hexadecimal: `7  D  5  F`

Written as `7D5F`$_{16}$ or `7D5F` hex

(Programmers who are used to working in hexadecimal or binary will often skip the denary step).

The method can be used in reverse to convert from hexadecimal to binary.

Worked example 1.4

Convert 1C9 Hex to a binary number that is to be stored as two bytes.

	1	C	9
Hexadecimal	1	12	9
Binary	00 01	1100	1001 = 1 1100 1001_2

'Stored as two bytes' means this will be stored as a 16-bit binary pattern.

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

The convention is to label the bit on the right-hand side as position 0.

Using 16 bits, bit position 0 is the **least significant bit**, and bit position 15 is the **most significant bit**.

Conversion between hexadecimal and denary

Worked example 1.5

Convert from hexadecimal to denary.

For hexadecimal > convert to binary > convert to decimal.

78 hex > 0111 1000_2 > 120_{10}

The opposite of the above example is to convert from denary to hexadecimal.

Worked example 1.6

To convert 93_{10} to hexadecimal:

It is easiest to convert the denary number to binary first – then to hex.

93_{10} > 0101 1101_2 > 5D hex

Worked example 1.7

Convert 93_{10} to hexadecimal – this time we shall not convert the denary number to binary first.

93 = 5 × 16 + 13

13 must be written as D, so the hexadecimal is:

5D hex

Progress check C

Convert these hexadecimal numbers to denary:

a 89 hex b 206 hex

Convert these hexadecimal numbers to 12-bit binary representations:

c 3F hex d 1EA hex

e CAB hex

Magnitude of numbers

The size of a file on the computer could be several thousand or several billion bytes. Hence, you need a notation to state the number concisely.

If you are counting in denary, then 1000 bytes is referred to as 1 kilobyte and 1,000,000 bytes is referred to as 1 megabyte.

However, the computer is more used to working with base 2.

In this case, 1 kibibyte is 1024 bytes (1024 is 2^{10}) and 1 mebibyte is 1,048,576 bytes (1024 × 1024 or 2^{20}).

Other multiples are in common use as the size of computer storage devices and memory continues to increase. The table below summarises the terms used.

Denary		Binary	
kilobyte	1000 (10^3) bytes	kibibyte	1,024 (2^{10}) bytes
megabyte	1,000,000 (10^6) bytes	mebibyte	1,048,578 (2^{20}) bytes
gigabyte	1,000,000,000 (10^9) bytes	gibibyte	1,073,741,824 (2^{30}) bytes
terabyte	1,000,000,000,000 (10^{12}) bytes	tebibyte	1,099,511,627,776 (2^{40}) bytes

You can remember these easily because they are increasing by a multiple of 1000 in the case of denary or 1024 in the case of binary, each time.

Denary		Binary	
kilobyte	1000 bytes	kibibyte	1024 bytes
megabyte	1000^2 bytes	mebibyte	1024^2 bytes
gigabyte	1000^3 bytes	gibibyte	1024^3 bytes
terabyte	1000^4 bytes	tebibyte	1024^4 bytes

Progress check D

File A has a file size of 2 kibibytes. File B has a file size of 2.1 kilobytes.

Which file has the larger file size?

Two's complement representation

Programs will need to use both positive and negative integers.

We are going to use a representation called two's complement.

Two's complement has a negative place value for the most significant bit.

For two's complement representation using a single byte (eight bits), the place values are as shown.

–128	64	32	16	8	4	2	1

Worked example 1.8

Convert the following denary numbers to 8-bit two's complement binary numbers.

a 56 = 32 + 16 + 8

–128	64	32	16	8	4	2	1
0	0	1	1	1	0	0	0

b –125 = –128 + 3 = –128 + 2 + 1

–128	64	32	16	8	4	2	1
1	0	0	0	0	0	1	1

c –17 = –128 + 111 = –128 + 64 + 32 + 8 + 4 + 2 + 1

–128	64	32	16	8	4	2	1
1	1	1	0	1	1	1	1

TIP

Note the method for the negative numbers. You need to start with 1x-128 and then work out what positive number to add to it as shown in b and c opposite.

1.02 Addition and subtraction of binary integers

The numbers will use two's complement.

All the examples below show each number stored with eight bits.

TIP

Using one's complement will show an alternative method to use for negative integers.

Addition

Worked example 1.9

Adding two positive integers (+31) + (+69)

+31		0	0	0	1	1	1	1	1	
+69		0	1	0	0	0	1	0	1	+
				1	1	1	1	1		This row shows the 'carry bit' from each addition
Answer		0	1	1	0	0	1	0	0	This is +100 denary

Worked example 1.10

Adding a positive and a negative integer (+56) + (−12)

+56		0	0	1	1	1	0	0	0	
−12		1	1	1	1	0	1	0	0	+
		1	1	1	1					This row shows the 'carry bit' from each bit addition
Answer		0	0	1	0	1	1	0	0	This is +44 denary

Worked example 1.11

Adding two positive integers (+114) + (+38)

+114		0	1	1	1	0	0	1	0	
+38		0	0	1	0	0	1	1	0	+
		1	1			1	1			This row shows the 'carry bit' from each bit addition
Answer		1	0	0	1	1	0	0	0	This is −104 denary

The pattern is not the answer that we expected, of +152.

The problem is that the correct answer is outside the range of numbers that it is possible to represent using 8-bit two's complement.

The range possible is: smallest number –128 and largest number +127.

Similarly, if you calculate (–106) + (–23), it would not show a correct answer of –129.

In both of these examples, **overflow** above has occurred in the most significant bit position.

One's complement and two's complement

Worked example 1.12

Express –27 denary in one's complement and two's complement.

$+27_{10}$ =	$0001\ 1011_2$
The one's complement is	1110 0100
Now add 1 to this	1110 0101 +
Gives the two's complement	1110 0101 = –27

So starting with the positive number (+27), the one's complement can be used to work out the two's complement for –27.

Subtraction

You can do a subtraction by either doing a binary subtraction or turning the calculation into an addition.

Worked example 1.13

Binary subtraction of 56 and 19.

Calculate +56 – 19

+56		0	0	1	1	1	0	0	0	
+19		0	0	0	1	0	0	1	1	(–)
						1	1	1		This row shows the 'carry bit' from each bit subtraction
Answer		0	0	1	0	0	1	0	1	This is + 37 denary

Worked example 1.14

Subtraction - by adding the two's complement

Calculate +59 – 19

This is the same as if you calculate (+56) + (–19).

+56		0	0	1	1	1	0	0	0	
–19		1	1	1	0	1	1	0	1	(+)
	1	1	1	1	1					This row shows the 'carry bit' from each bit addition
Answer		0	0	1	0	0	1	0	1	This is +37 denary

Progress check E

Show the binary calculations.

a (+13) + (+78)

b (+90) – (+92)

Binary Coded Decimal (BCD)

This is an alternative binary representation that can be used for a positive denary integer. It does not use place values.

Each denary digit in the sequence is represented as a group of four binary digits (a nibble).

Worked example 1.15

Represent the denary integer 571 in BCD.

5 7 1

0101 0111 0001

So, 571 denary is 0101 0111 0001 in BCD.

Applications of BCD

BCD is used in electronics systems where a string of digits is used to represent some value. BCD has the advantage that a given number is easily scalable by a factor of ten. To multiply the number by ten simply add a group of zero bits to the least significant end. This calculation is much simpler than multiplying a two's complement number by ten.

Progress check F

Write the denary number 184 in BCD.

1.03 Representing characters

All characters must be stored as numbers.

The character set will include upper case letters, lower case letters, the number digits and all the punctuation and other characters found on a standard QWERTY keyboard.

A coding system such as **ASCII** or **Unicode** will be used.

Each character will be encoded with a different number.

ASCII (American Standard Code for Information Interchange)

The ASCII coding system uses a 7-bit code to represent each of the characters. A selection of the codes is shown in the table below:

ASCII code table (part)

Character	Denary	Character	Denary	Character	Denary
<Space>	32	I	73	R	82
A	65	J	74	S	83
B	66	K	75	T	84
C	67	L	76	U	85
D	68	M	77	V	86
E	69	N	78	W	87
F	70	O	79	X	88
G	71	P	80	Y	89
H	72	Q	81	Z	90

The characters with codes 0 to 31 are called control characters. If you use them in a program, they will cause some effect – such as a 'bleep' (ASCII code 7).

For example, ASCII code 12 causes the paper in the printer to be ejected.

The maths tells us that 7-bits makes 128 different codes possible (with binary codes 0000000, 00000001, …, 1111111).

The computer stores all ASCII codes as a byte (8-bits).

Extended ASCII character set

Consider if all eight bits of the byte were to be used. The number of different characters that could then be represented increases to 256. This is called the extended character set.

Agreement was reached with a standard, called ANSI as to what all the character codes below 128 would represent.

But different standards emerged as to how the codes 128–255 would be used. This caused problems as different countries used a different standard.

This was the reason for the introduction of a new universally recognised character set called Unicode.

Unicode

Unicode provides a unique number for every character. This number will be recognised as the same character on different platforms, and in different programs and languages.

Different standards exist for Unicode - UTF-8, UTF-16, UTF-232 and others. UTF-8 is the most widely used on the WWW.

UTF-8 uses one byte for the first 128 characters, called code points (the upper and lower case letters, number digits, etc). The first 128 code points are encoded as a single byte. Up to 4 bytes are used for other characters. The first 128 Unicode code points have the same encoding as the 8-bit ASCII character codes.

Unicode codes have the format:

`U+0041`. This is the code for character A. The U+ indicates 'Unicode' and the digits are the hexadecimal code to be used. Note, this confirms that this code is the same as the ASCII code.

All data that is to be used by programs, for example, an email message or web page, must specify the encoding method used.

For a typical webpage, the HTML tag would be:

```
<meta http-equiv="Content-Type"
content="text/html; charset = utf-8">
```

1.04 Graphics

Bitmapped image

A bitmap graphic is a rectangular grid built up from a number of **pixels**.

Each pixel will be a particular colour and each pixel's colour will be stored as a binary number.

The contents of the bitmap file will be this sequence of binary colour codes, each representing a single pixel in the rectangular grid.

The table shows the various **encodings** used for bitmaps.

The number of bits used to encode a single pixel is called the **bit depth**.

The number of possible colours that can be used is called the **colour depth**.

Type of encoding	Bit depth	Colour depth	Explanation
Monochrome	1 bit	2	Only two colours (Black and white). One byte stores eight pixels.
16 colour	4 bits	16	One byte stores two pixels.
256 colour	8 bits (1 byte)	256	One byte stores one pixel.
24-bit colour or 'Tru-colour'	24 bits (3 bytes)	2^{24} i.e. 16,777,216	Millions of different colours are possible.

The **file header** data will include the width and height, measured in pixels, the type of encoding and other data, such as a date stamp.

Bitmap calculations

Worked example 1.16

A bitmapped image has a width of 100 pixels and a height of 50 pixels. The file header uses 60 bytes. The image is encoded as a '256 colour' image. Calculate the file size (in kilobytes).

Number of pixels = 100 x 50 = 5000

Each pixel is stored with one byte … so

Bytes used for the pixel data = 5000

Total file size = pixel data + the file header = 5000 + 60 = 5060 bytes = 5.06 kilobytes

The general formula is:

File size (in bytes) = (Width in pixels x Height in pixels x Bit depth) + Header bytes

Worked example 1.17

An image has a width of 2048 pixels and a height of 128 pixels. The file header uses 100 bytes. The image is saved as a '24-bit colour' image. Calculate the file size (in kibibytes).

Number of pixels = 2048 x 128

Each pixel is stored with 3 bytes …

Bytes used for the pixel data is = 2048 x 128 x 3 = 786432

Total file size in bytes = Pixel data + the file header = 786432 + 100 = 787532 bytes

File size = 787532 / 1024 = 769 kibibytes