Data representation

IGCSE Computer Science

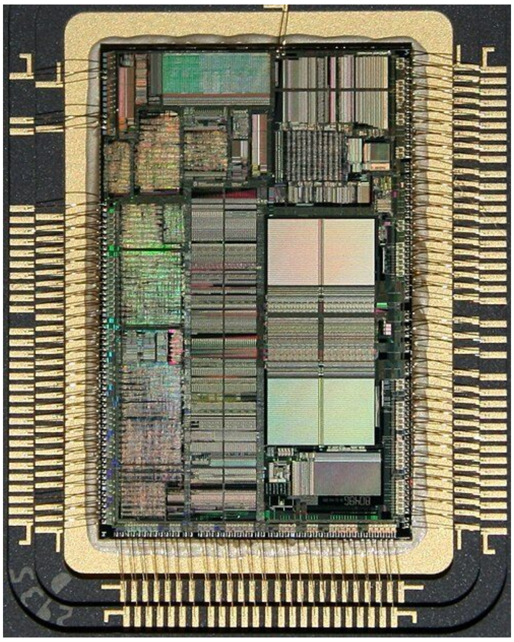
Why computers use binary

A computer can only work with two states: on and off. You write these as 1 and 0. A system that uses only two digits 数字 is called **binary** 二进制 (base 2).

Computers represent all data —numbers, text, sound and images —as binary strings of 0s and 1s

Image: geralt, CC0 (commons.wikimedia.org)

Every kind of data 数据—numbers, text, sound and images —must be changed into binary before a computer can use it. The computer processes this binary using **logic gates** 逻辑门, and stores it in **registers** 寄存器 (small, fast stores inside the processor 处理器).



A microprocessor holds millions of tiny transistors, each a switch that is on (1) or off (0) —the physical basis of binary

Image: Thomy pc at German Wikipedia, Public domain (commons.wikimedia.org)

Number systems

A **number system** 数制 is a way of writing numbers using a fixed set of digits. You need three of them.

System	Base	Digits used
Denary	10	0–9
Binary	2	0 and 1
Hexadecimal	16	0–9 then A–F

- **denary** 十进制 is the normal counting system (also called decimal).
- **binary** uses only 0 and 1.
- **hexadecimal** 十六进制 (hex) uses sixteen digits: 0–9, then A, B, C, D, E, F stand for 10, 11, 12, 13, 14, 15.

The **base** 基数 tells you how many different digits a system uses.

Place value

Each column in a number has a **place value** 位值. In binary the place values double from right to left. For an 8-bit number they are:

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

place value	128	64	32	16	8	4	2	1
bits	**1**	0	0	**1**	0	**1**	**1**	0

$150 = 128 + 16 + 4 + 2 \Rightarrow$ `10010110`

An 8-bit place-value chart: the 1s sit under the values that add up to 150

One **bit** 位 is a single 0 or 1. Eight bits make one **byte** 字节. Four bits (half a byte) is a **nibble** 半字节.

Converting between number systems

Denary → binary. Write the place values. Put a 1 under each value you need so they add up to your number; put 0 under the rest.

Example: change denary 150 to binary. $150 = 128 + 16 + 4 + 2$.

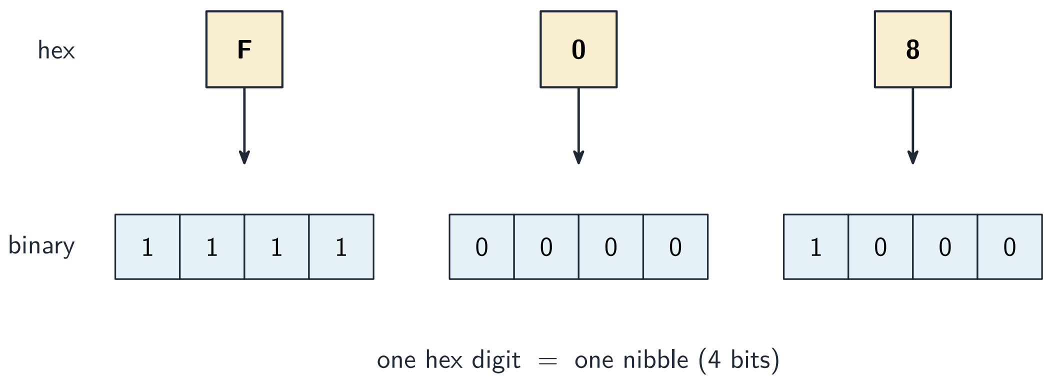
```
128 64 32 16 8 4 2 1
  1  0  0  1 0 1 1 0
```

So $150 =$ `10010110`.

Binary → denary. Add the place values where there is a 1. `10010110` $= 128+16+4+2 = 150$.

Hexadecimal → binary. Change each hex digit into its own 4-bit group (a nibble).

Example: hex F08. $F = 1111$, $0 = 0000$, $8 = 1000$, so F08 = `1111 0000 1000`.



Each hex digit maps to its own 4-bit nibble —F08 = 1111 0000 1000

Binary → hexadecimal. Group the bits into nibbles of 4, starting from the right. Change each nibble to one hex digit.

Denary → hexadecimal. The easy way is to change to binary first, then binary to hex.

This table helps with the hex letters:

Denary	Binary	Hex
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Cambridge questions use binary numbers up to 16 bits long.

Worked example. Convert denary 100 to 8-bit binary, then to hexadecimal.

$100 = 64 + 32 + 4$, so the binary is `01100100`. Splitting into nibbles, `0110 0100` $= 6$ and 4, so the hexadecimal is `64`.

Why hexadecimal is used

Hex is shorter than binary and easier for people to read and write. One hex digit replaces 4 binary digits, so you make fewer mistakes. The value does not change —hex is just a shorter way to show the same binary.

Computer scientists use hex for:

- MAC addresses and IPv6 addresses
- colour codes in HTML (for example `#FF0000` is red)
- **memory addresses** 内存地址 and error codes
- showing the contents of memory (a ”memory dump”)

Binary addition

You can add two 8-bit binary numbers, column by column from the right, just like denary. The rules for one column are:

A	B	Result bit	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

When a carry also comes in, $1 + 1 + 1 = 1$ with a carry of 1.

Example: add `01110110` (118) and `00110000` (48).

```
  0 1 1 1 0 1 1 0    (118)
+ 0 0 1 1 0 0 0 0    (48)
-------------------
  1 0 1 0 0 1 1 0    (166)
```

carries	1	1	1						
	0	1	1	1	0	1	1	0	118
+	0	0	1	1	0	0	0	0	48
	1	0	1	0	0	1	1	0	166

Adding column by column; the carries ripple to the left. 118 + 48 = 166

Overflow

An 8-bit register can hold denary values from 0 to 255 only. If an addition gives a result above 255, the answer needs a 9th bit. The register cannot hold this extra bit, so it is lost. This is called **overflow** 溢出 (an overflow error). It happens when a value goes outside the limit the register can store.

Example: `11001000` (200) + `01001000` (72) = 272. In binary that is `1 00010000`, which needs 9 bits. The leading 1 will not fit in 8 bits, so the stored answer is wrong.

Adding 200 and 72 needs 9 bits, but an 8-bit register drops the ninth, so the answer is wrong

Logical binary shift

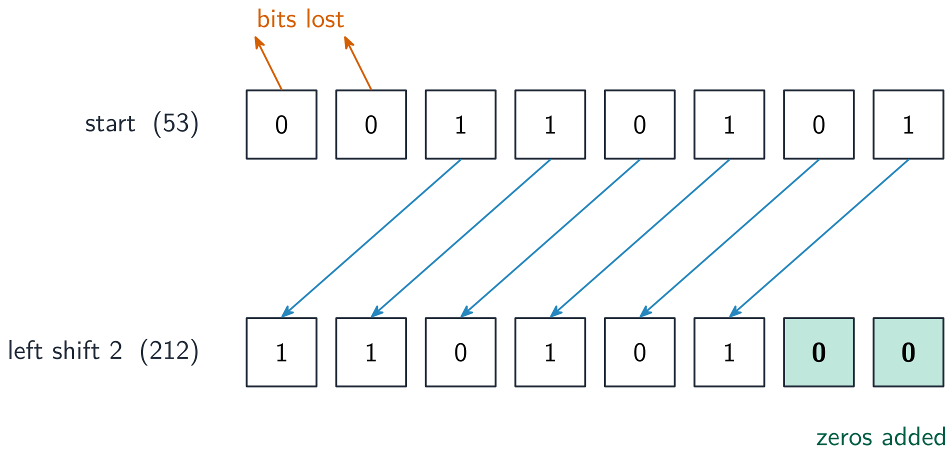
A **logical binary shift** 逻辑二进制移位 moves all the bits left or right by a number of places.

- Bits that move off the end of the register are **lost**.
- **Zeros** are added at the empty end.

A **left shift** multiplies the number by 2 for each place moved. A **right shift** divides it by 2 for each place; the right-most bits (the **least significant bit(s)** 最低有效位) are lost.

Example: left shift `00110101` (53) by 2 places.

```
start:         0 0 1 1 0 1 0 1
left shift 2:  1 1 0 1 0 1 0 0
```



A left shift of 2: every bit moves 2 places left, the top bits are lost and zeros fill the right

The result is `11010100` (212), which is 53×4. The two left-most bits were lost and two zeros came in on the right. If a 1 is pushed off the end, that information is gone for good.

Two's complement

So far the numbers were positive. **Two's complement** 补码 lets an 8-bit register hold negative numbers too.

In two's complement, the left-most bit (the **most significant bit** 最高有效位, or MSB) has a *negative* place value:

```
-128  64  32  16  8  4  2  1
```

- If the MSB is 0, the number is positive.
- If the MSB is 1, the number is negative.

To make a positive number negative: write the positive binary, flip every bit (0 1), then add 1.

Example: make -40.

- $+40 =$ `00101000`
- flip the bits $=$ `11010111`
- add $1 =$ `11011000`

So $-40 =$ `11011000`. Check by adding the place values: $-128 + 64 + 16 + 8 = -40$.

sign bit (negative)

place value	-128	64	32	16	8	4	2	1
bits	**1**	**1**	0	**1**	**1**	0	0	0

$-128 + 64 + 16 + 8 = -40$

The most significant bit is worth −128, so 11011000 = −128 + 64 + 16 + 8 = −40

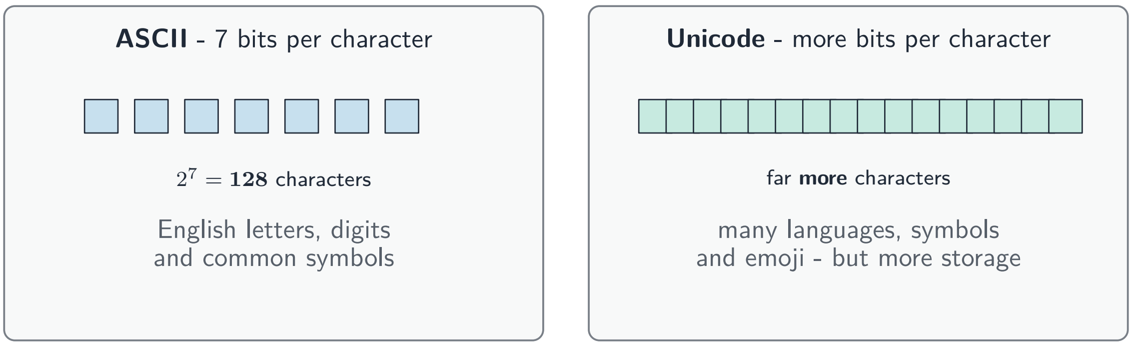
To read a negative two's complement number, just add the place values (the MSB counts as -128). The range of an 8-bit two's complement number is -128 to $+127$.

Representing text

Computers store text by giving every character a number, then storing that number in binary. The set of characters a computer can use, together with their numbers, is a **character set** 字符集.

- **ASCII** uses 7 bits per character, so it has 128 different characters. This is enough for English letters, digits and common symbols.
- **Unicode** uses more bits per character. It can represent far more characters —many languages, plus symbols and **emoji** 表情符号.

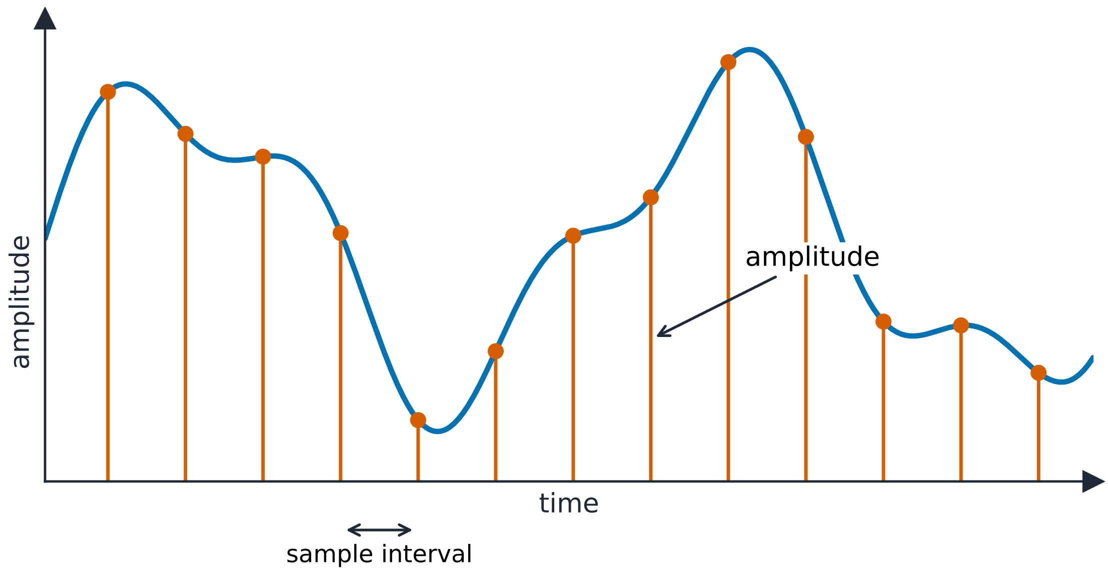
Because Unicode has more characters, it needs more bits per character than ASCII, so the same text takes more storage 存储.



ASCII uses 7 bits for 128 characters; Unicode uses more bits for far more characters but more storage

Representing sound

A **sound wave** 声波 is smooth and always changing. To store it, the computer measures the height of the wave at regular moments. This is called **sampling** 采样, and each measurement is a sample.



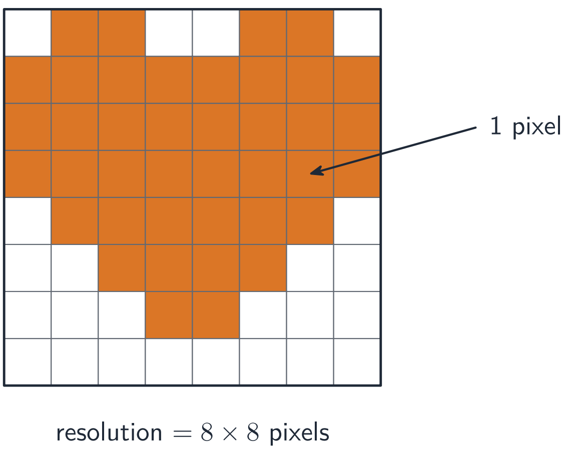
Sampling records the wave's height (amplitude) at regular moments

- **sample rate** 采样率 is the number of samples taken each second (measured in Hz).
- **sample resolution** 采样分辨率 is the number of bits used for each sample. The height of the wave at a sample point is its **amplitude** 振幅.

A higher sample rate and a higher sample resolution give a more accurate recording, but a larger file.

Representing images

A computer image is made of a grid of small dots called **pixels** 像素.



A bitmap is a grid of pixels; the resolution is how many pixels it has

- **resolution** 分辨率 is the number of pixels in the image (for example 1920 × 1080).
- **colour depth** 颜色深度 is the number of bits used to store the colour of each pixel.

A higher resolution and a higher colour depth give a better-quality image, but a larger file.

Measuring data storage

Data storage is measured in the units below. Each unit is 1024 times the one before it (because $1024 = 2^{10}$, which fits binary).

Unit	Equals
bit	a single 0 or 1
nibble	4 bits
byte	8 bits
kibibyte (KiB)	1024 bytes
mebibyte (MiB)	1024 KiB
gibibyte (GiB)	1024 MiB
tebibyte (TiB)	1024 GiB
pebibyte (PiB)	1024 TiB
exbibyte (EiB)	1024 PiB

Calculating file size

Image file size (in bits) = resolution × colour depth = width × height × colour depth.

Example: an image is 1024 × 1024 pixels with a colour depth of 2 bytes (= 16 bits).

- bits = $1024 \times 1024 \times 16 = 16\,777\,216$ bits
- bytes = $\div 8 = 2\,097\,152$ bytes
- KiB = $\div 1024 = 2048$ KiB
- MiB = $\div 1024 = 2$ MiB

Sound file size (in bits) = sample rate × sample resolution × length in seconds.

Always divide by 1024 (not 1000) to change to KiB, MiB and so on. Give your answer in the unit the question asks for.

Worked example. A sound is recorded for 30 seconds at a sample rate of 8,000 Hz with a sample resolution of 16 bits. Find the file size in kibibytes (KiB).

- bits = $8\,000 \times 16 \times 30 = 3\,840\,000$ bits
- bytes = $3\,840\,000 \div 8 = 480\,000$ bytes
- KiB = $480\,000 \div 1024 \approx 469$ KiB

Compression

Compression 压缩 makes a file smaller. A smaller file:

- uses less storage space,
- needs less **bandwidth** 带宽 (the amount of data a connection can carry),

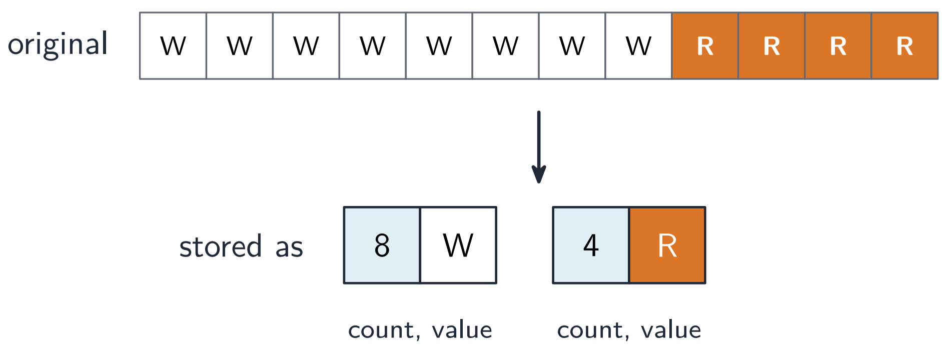
- takes a shorter time to send (a shorter **transmission** 传输 time).

There are two types.

Lossless compression

Lossless 无损 compression makes the file smaller with **no permanent loss** of data. The original file can be rebuilt exactly.

One method is **run-length encoding** 行程编码 (RLE). It replaces a run of repeated values with one copy of the value and a count of how many times it repeats. For example `WWWWWWWW` (8 whites) is stored as "8 W". This works well when data has many repeats.



Run-length encoding stores each run once as a count and a value

Lossy compression

Lossy 有损 compression makes the file much smaller by **permanently removing** some data. The removed data cannot be got back. For example:

- reducing the resolution or colour depth of an image,
- reducing the sample rate or sample resolution of a sound.

Use lossless when you must keep every detail (text and program files). Use lossy for photos, music and video, where a small loss of quality is worth a much smaller file.

Exam tips

- Convert denary → binary by subtracting the place values (128, 64, 32 ⋯); binary → denary by adding the place values that hold a 1.
- To convert to hex, group the binary into **nibbles** of 4 bits from the right; each nibble is exactly one hex digit.
- **Overflow** happens when a result needs more bits than the register has (an 8-bit register only holds 0–255), so the extra bit is lost.
- File size in bits: for an image, width × height × colour depth; for sound, sample rate × resolution × seconds. Divide by 8 for bytes, then by 1024 for each larger unit.

- **Lossless** compression keeps every bit (text; run-length encoding); **lossy** permanently removes data (photos, music) for a much smaller file.