Data Representation

IGCSE Computer Science ▪ Topic 1

IGCSE Computer Science

1 0 0 1 0 1 1 0

What we will cover

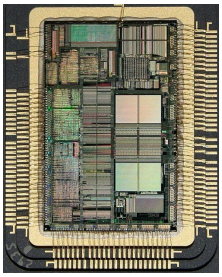
1. Binary & number systems
2. Binary addition & overflow
3. Shifts & two's complement
4. Text, sound & images
5. Storage & file size
6. Compression

1 Number systems

Why binary?

A computer has only two states: on and off, written 1 and 0. A system of two digits is **binary** 二进制 (base 2).

> Every kind of data – numbers, text, sound, images – must become binary before the computer can process it.



Three number systems

denary 十进制
base 10
digits 0–9

binary 二进制
base 2
digits 0 and 1

hexadecimal 十六进制
base 16
0–9 then A–F

> The **base** 基数 is how many different digits a system uses. Hex A–F stand for 10–15.

Hard disk storage

Place value

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

Binary place values double from right to left. For 8 bits:

$$128\ 64\ 32\ 16\ 8\ 4\ 2\ 1$$

- 1 **bit** 位 = a single 0 or 1
- 8 bits = 1 **byte** 字节
- 4 bits = 1 **nibble** 半字节

> Denary → binary: subtract the place values that fit. $150 = 128 + 16 + 4 + 2$.

Which end of a binary number matters

Most significant bit	The **most significant bit** 最高有效位 is the leftmost – worth the most. In two's complement it is also the sign.
Least significant bits	The **least significant bit(s)** 最低有效位 are the rightmost – worth the least, and the first to be lost when a number is truncated.
Why it matters	A logical shift left multiplies by two and pushes bits off the most significant end; a shift right divides and loses the least significant.
In hardware	The same convention orders the bits in **registers** 寄存器, in **transmission** 传输 down a wire, and in the outputs of **logic gates** 逻辑门.

> Say which end you mean. "The first bit" is ambiguous, and the mark scheme is not.

Worked example – denary to binary and hex

Convert denary 150 to binary, and 100 to hexadecimal.

$150 = 128 + 16 + 4 + 2$, so

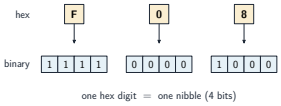
128 64 32 16 8 4 2 1

1 0 0 1 0 1 1 0

$= 10010110$

$100 = 64 + 32 + 4 = 0110\,0100$

nibbles $= 6$ and 4, so hex $= 64$



one hex digit = one nibble

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

Why hexadecimal?

Hex is shorter than binary and easier to read – one hex digit replaces four binary digits, so you make fewer mistakes. Used for:

- MAC and IPv6 addresses
- HTML colours (#FF0000 is red)
- **memory addresses** 内存地址, error codes, memory dumps

> Group binary into nibbles of 4 from the right; each nibble is exactly one hex digit. The value is unchanged – only the notation is shorter.

2 Addition and shifts

Binary addition

carries	1	1	1					
	0	1	1	1	0	1	1	0
+	0	0	1	1	0	0	0	0
	1	0	1	0	0	1	1	0

118

48

166

Add column by column from the right, carrying like denary:

A	B	bit	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

> $1 + 1 + 1 = 1$ carry 1. Example: $118 + 48 = 166$.

Overflow

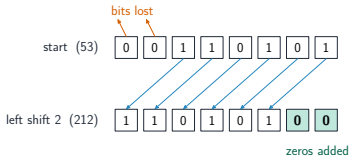
$200 + 72 = 272$, but 272 needs **9 bits**:

stored answer $= 00010000 = 16$, **not** 272 - this is **overflow**

An 8-bit register holds only 0–255. A bigger result needs a 9th bit that will not fit. $200 + 72 = 272 =$ 1 00010000 – 9 bits.

> The leading 1 is lost, so the register shows the wrong answer. That lost carry is **overflow** 溢出.

Logical binary shift



A **logical shift** 逻辑二进制移位 moves every bit left or right:

- bits off the end are lost
- zeros fill the empty end
- left $\times 2$ per place; right $\div 2$ per place

> Left-shift 00110101 (53) by $2 =$ 11010100 ($212 = 53 \times 4$). Push a 1 off the end and you lose value.

Two's complement

	sign bit (negative)							
place value	−128	64	32	16	8	4	2	1
bits	**1**	**1**	0	**1**	**1**	0	0	0

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

Two's complement 补码 stores negatives by making the **MSB** 最高有效位 negative:

$$-128\ 64\ 32\ 16\ 8\ 4\ 2\ 1$$

MSB $0 =$ positive, MSB $1 =$ negative.

> Make -40: $+40$ =00101000, flip =11010111, add 1 =11011000. Check: $-128 + 64 + 16 + 8 = -40$.

3 Text, sound and images

Representing text

Every character gets a number, stored in binary – a **character set** 字符集:

- **ASCII** ASCII 码: 7 bits = 128 characters (English)
- **Unicode** 统一码: more bits = far more, many languages plus emoji

> More characters needs more bits, so the same text in Unicode takes more storage than in ASCII.

Representing text, in detail

ASCII - 7 bits per character

$2^7 = 128$ characters
English letters, digits and common symbols

Unicode - more bits per character

far **more** characters
many languages, symbols and emoji - but more storage

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

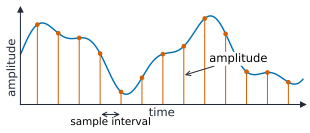
Sound and images

Sound: sample rate 采样率 (samples/s) and **sample resolution** 采样分辨率 (bits/sample).

Image: resolution 分辨率 (pixels) and **colour depth** 颜色深度 (bits/pixel).

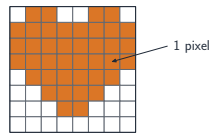
> Higher rate/resolution/depth ⇒ better quality but a larger file. Always the same trade-off.

Sound and images, in detail



sample rate = samples/s · Nyquist $\geq 2f_{max}$

sampling a sound wave



resolution $= 8 \times 8$ pixels

an image is a grid of pixels

Both are the same idea: measure the real thing at regular points and store the numbers. Sound is sampled in time, an image in space.

4 Storage and file size

Measuring storage

Unit	Equals
byte	8 bits
kibibyte (KiB)	1024 bytes
mebibyte (MiB)	1024 KiB
gibibyte (GiB)	1024 MiB
tebibyte (TiB)	1024 GiB

Each unit is $1024\times$ the one before, because $1024 = 2^{10}$ fits binary.

Divide by 1024 (not 1000) to step up a unit. This is why a "1 TB" drive shows fewer TiB in the operating system.

Worked example – file size

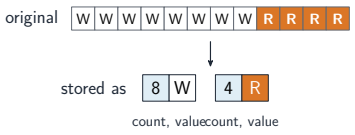
An image is 1024×1024 pixels, colour depth 2 bytes (16 bits). Find its size in MiB.

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

Image bits = width × height × depth; sound bits = rate × resolution × seconds. Then $\div 8$, $\div 1024$, $\div 1024$ – in the unit asked for.

5 Compression

Compression



Smaller files use less storage, less **bandwidth** 带宽 and send faster.

- **lossless** 无损: keep every bit – e.g. **RLE** 行程编码
- **lossy** 有损: permanently drop data for a much smaller file

Lossless for text and programs (every bit matters). Lossy for photos, music, video – a small quality loss is worth the size.

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.

6 Recap

Topic 1 – key takeaways

1. **Convert**
   subtract place values; group nibbles for hex
2. **Overflow**
   result needs a 9th bit an 8-bit register loses
3. **File size**
   bits → $\div 8$ → $\div 1024$; image/sound formulas
4. **Compress**
   lossless keeps every bit; lossy drops it

Check yourself

1. How many bits in a nibble?
2. Convert binary 1010 to hex.
3. What happens on overflow?
4. Lossless or lossy for a text file?

1 0 1 0

Answers **1.** 4 **2.** A **3.** the extra bit is lost (wrong answer) **4.** lossless