Topic 1: Data Representation

IGCSE Computer Science 0478 | SenpaiCorner | Mr. Wei

Learning Objectives

- **1.1** Understand and use binary, denary, and hexadecimal; perform binary addition; explain overflow errors; apply logical binary shifts; represent signed integers using two's complement.
- **1.2** Describe how text (ASCII/Unicode), sound (sampling), and images (pixels/colour depth) are represented in binary; explain the effect of varying sample rate, resolution, and colour depth on quality and file size.
- **1.3** Know and apply storage units (bit to EiB); calculate image and sound file sizes; describe and compare lossy and lossless compression; explain run-length encoding (RLE).

1.1 Number Systems

Why Computers Use Binary

All data must be converted to binary before it can be processed by a computer. Computers use **transistors**, **logic gates**, and **switches** to process and store data. These electronic components can only exist in two states: **on (1)** or **off (0)**. Data is stored in **registers**, which hold binary values during processing.

Common Mistake

Do *not* say "computers use binary because it is faster." The correct reason is that the electronic components (transistors/logic gates) can only detect **two states**: high/low, on/off, 1/0. This is worth 1–2 marks in exam questions that ask you to *explain why*.

Number System Comparison

Property	Binary	Denary	Hexadecimal
Base	2	10	16
Digits used	0, 1	0–9	0–9, A–F
Column values	2^0, 2^1, 2^2, …	10^0, 10^1, 10^2, …	16^0, 16^1, 16^2, …
Digits for same value	Most digits	Medium	Fewest digits

Key unit definitions:

- **Bit** (binary digit) — smallest unit of data; either 0 or 1.
- **Nibble** — 4 bits.
- **Byte** — 8 bits.

Number Base Conversions

Denary to Binary

1. Write column headings right-to-left: 128, 64, 32, 16, 8, 4, 2, 1 (powers of 2).
2. From the leftmost column: write 1 if the column value $\leq$ the remaining amount; otherwise write 0. Subtract the column value when you place a 1.
3. Verify by summing all columns headed by 1.

Worked Example: 188 to Binary

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

Check: $128 + 32 + 16 + 8 + 4 = 188$ $\Rightarrow$ **Answer: 10111100**

Binary to Denary

Write column headings, fill in the binary digits, then add up all columns containing a 1.

Worked Example: `10111100` to Denary

$128 + 32 + 16 + 8 + 4 = \mathbf{188}$

Binary to Hexadecimal

1. Split the 8-bit number into two **nibbles** (groups of 4 bits) from the right.
2. Convert each nibble using column headings 8, 4, 2, 1.
3. Replace denary values 10–15 with the corresponding hex letter (A–F).

Worked Example: `10111100` to Hexadecimal

Split: `1011` | `1100`

$1011 = 8 + 2 + 1 = 11 = \mathbf{B}$ $1100 = 8 + 4 = 12 = \mathbf{C}$

Answer: BC

Hexadecimal to Binary

Convert each hex digit individually to a 4-bit nibble, then join the nibbles.

Worked Example: `BC` to Binary

B $= 11 \rightarrow$ `1011` C $= 12 \rightarrow$ `1100`

Answer: 10111100

Hexadecimal $\leftrightarrow$ Denary (Bridging Rule)

- **Hex to Denary:** convert hex $\rightarrow$ binary $\rightarrow$ denary.
- **Denary to Hex:** convert denary $\rightarrow$ binary $\rightarrow$ hex.

Exam Strategy

Never attempt a direct hex $\leftrightarrow$ denary conversion in an exam—always bridge through binary. More importantly, **always write your column headings** as working. Cambridge awards a method mark for visible working even if the final answer contains a small arithmetic slip.

Hexadecimal reference table (memorise the letter values A–F):

Den.	Hex	Den.	Hex	Den.	Hex	Den.	Hex
0	0	4	4	8	8	12	C
1	1	5	5	9	9	13	D
2	2	6	6	10	A	14	E
3	3	7	7	11	B	15	F

Binary Addition

Addition rules:

- $0 + 0 = 0$
- $0 + 1 = 1$
- $1 + 1 = 10$ (write 0, carry 1 into the column to the left)
- $1 + 1 + 1 = 11$ (write 1, carry 1 into the column to the left)

Common Error Trap

Cambridge marks binary addition **one mark per correct nibble** and **one mark for correct carries**. Always write the carry row explicitly — a correct digit produced by a carry you did not show will still lose the carry mark.

Worked Example: Binary Addition

Add `00110011` and `01100001`:

carry	1	1				1	1	
	0	0	1	1	0	0	1	1
$+$	0	1	1	0	0	0	0	1
$=$	1	0	0	1	0	1	0	0

Result: `10010100` $= 148$ in denary. No overflow — fits in 8 bits.

Overflow Error

An **overflow error** occurs when the result of a calculation is too large to be stored in the register. For an 8-bit register the maximum storable value is 255 (`11111111` in binary). If the result would require more than 8 bits, the most significant bit(s) are lost and the stored value becomes incorrect.

Common Mistake

A 2-mark overflow question requires **both** of these points:

- The register has a **fixed/predefined** number of bits.
- The result of the calculation **exceeds the maximum value** that can be stored (e.g., exceeds 255 for an 8-bit register).

Giving only one of these earns only 1 mark.

Minimum bits needed to store a value: convert to binary and count the digits.

Past-paper examples:

- Denary 2000 $\rightarrow$ `11111010000` $\rightarrow$ **11 bits**
- Denary 301 $\rightarrow$ `100101101` $\rightarrow$ **9 bits**
- Hexadecimal A4D $\rightarrow$ `101001001101` $\rightarrow$ **12 bits**

Logical Binary Shifts

A **logical binary shift** moves all bits in a register a fixed number of positions left or right. Zeros fill the vacated positions; bits shifted off the end are lost.

- **Left shift by n places:** multiplies the value by 2^n; **most significant bits** (leftmost) are lost.
- **Right shift by n places:** divides the value by 2^n; **least significant bits** (rightmost) are lost.

Common Mistake

Students frequently reverse MSB and LSB:

Most Significant Bits (MSB) = leftmost bits = lost in a *left* shift.
Least Significant Bits (LSB) = rightmost bits = lost in a *right* shift.

Common Error Trap

If any **1-bit** is shifted off the end, the result is **inaccurate** — the $\times 2^n$ or $\div 2^n$ relationship no longer holds. You must explicitly state: "the value becomes incorrect/inaccurate as the shifted bit(s) are lost." Do not state the arithmetic effect (e.g., 'multiplied by 8') without first confirming that no 1-bits were lost.

Two's Complement (Signed Binary)

Two's complement is used to represent **negative** integers in binary. The **leftmost (sign) bit** indicates the sign: **0 = positive, 1 = negative**.

To convert a negative denary number to 8-bit two's complement:

1. Find the 8-bit binary of the *positive* equivalent.
2. **Invert all bits** (flip every 0 to 1 and every 1 to 0).
3. **Add 1** to the inverted result.

Worked Example: -78 in Two's Complement

1. $+78 = 01001110$
2. Invert all bits: `10110001`
3. Add 1: `10110010`

Answer: $-78 =$ `10110010` (leftmost bit is **1** — confirms negative)

Mark Scheme: Two's Complement — 2 marks

Mark 1 (method): Working shown — conversion of $+78$ to binary, followed by inverting all bits, then adding 1.

Mark 2 (answer): Correct final register value `10110010`.

Both marks can be awarded even if one small arithmetic slip occurs, provided the correct three-step method is clearly visible.

Benefits and Uses of Hexadecimal

Why programmers use hexadecimal to represent binary values:

- Easier / quicker to read, write, and understand than long binary strings.
- Shorter representation — fewer digits for the same value (takes up less display space).
- Easier to spot and correct errors (debug).
- Less chance of making a copying error.

Uses of hexadecimal in computer science:

- MAC addresses and IPv6 IP addresses.
- Error codes and error messages.
- HTML / CSS colour codes (e.g., `#FF0000` = red).
- Assembly language / low-level language instructions.
- Memory addresses and memory dumps.
- ASCII and Unicode code points.

Exam Strategy: HTML Colour Codes

Memorise the six primary/secondary HTML hex colours:

`#FF0000` (red) `#00FF00` (green) `#0000FF` (blue)
`#00FFFF` (cyan) `#FF00FF` (magenta) `#FFFF00` (yellow)

Each pair of hex digits controls one RGB channel: `00` = none, `FF` = full. Different combinations of values create different shades and tones.

1.2 Text, Sound and Images

Representing Text: Character Sets

A **character set** is the complete collection of characters and symbols that a computer system can represent. Each character is assigned a **unique binary value**.

Feature	ASCII	Unicode
Bits per character	7 (sometimes extended to 8)	Variable: 8, 16, or 32 bits
Number of characters	128 (7-bit) / 256 (extended)	65,536 (16-bit) / ≈4 billion (32-bit)
Languages supported	English only	Most world languages + emojis
Encoding	Fixed-length	Variable-length
Relative file size	Smaller	Larger

Disadvantages of ASCII: limited number of characters; does not support languages other than English.

Disadvantages of Unicode: requires more bits per character than ASCII; larger file sizes; potentially slower processing.

Common Mistake

A common error: stating that Unicode "uses 16 bits per character." Unicode uses **variable-length** encoding — it can use 8, 16, or 32 bits depending on the character. The key point is that it uses *more* bits per character than ASCII.

Exam Strategy

For *compare* questions on ASCII vs Unicode, structure each point as an explicit contrasting pair: "ASCII has [X]; Unicode has [Y]." Listing features of only one standard without contrasting will not earn comparison marks.

Representing Sound

Sound is an **analogue** signal. To process it, a computer must convert it to **digital binary data** via **sampling**.

How analogue sound is digitised:

1. A microphone records the analogue sound wave.
2. The wave is **sampled** at regular time intervals — the amplitude (height) of the wave is measured at each sample point.
3. Each amplitude is assigned a unique binary value.
4. The **sample rate** (Hz) and **sample resolution** (bits per sample) are configured.
5. Each sample is converted to binary and stored sequentially.

Factor	Definition	Effect of increasing
Sample rate	Number of samples taken per second (Hz)	More accurate recording; larger file
Sample resolution	Number of bits used per sample	More amplitude detail; larger file

Common Mistake

Sample rate controls *time* accuracy — how closely the samples track the original wave over time. **Sample resolution** controls *amplitude* accuracy — how many distinct loudness levels can be recorded. These are two separate factors; do not conflate them in exam answers.

Advantages of higher sample rate / resolution:

- Better quality and accuracy of recording; recording is closer to the original.
- Less sound distortion.

Disadvantages of higher sample rate / resolution:

- Larger file size; fewer files can be stored on a device.
- Takes longer to download or transfer.
- Requires greater processing power.

Representing Images

A digital image is a grid of **pixels** (picture elements). Each pixel stores one colour, encoded as a binary value.

Factor	Definition	Effect of increasing
Resolution	Number of pixels (width $\times$ height)	Sharper, more detailed image; larger file
Colour depth	Number of bits used per pixel	More colours representable; larger file

How a digital image is stored:

- The image is composed of pixels arranged in a grid.
- Each pixel stores one colour as a unique binary code.
- Pixels are stored sequentially (left-to-right, top-to-bottom).
- The file includes **metadata** (colour depth and resolution) so the display knows how to reconstruct the image correctly.

Exam Strategy

For any "drawback of high resolution / high colour depth" question, the answer always follows the same chain:

larger file size $\Rightarrow$ more storage required $+$ slower to upload/download/transfer $+$ more bandwidth needed.

Learn this chain once and apply it to images, sound, and video — it is always the same.

1.3 Data Storage and Compression

Storage Units

Unit	Abbreviation	Equivalent
Bit	b	Smallest unit (0 or 1)
Nibble	—	4 bits
Byte	B	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

Common Error Trap

File size calculations use **1024** between units, not 1000. The Cambridge syllabus uses **IEC binary prefixes** (KiB, MiB, GiB, . . .). Using 1000 gives the SI prefixes (KB, MB, GB) — a direct exam trap. Past papers have explicitly tested this distinction (e.g., a company quotes 62 500 files using 1000; the correct answer using 1024 gives 64 000 files).

File Size Calculations

Image file size (in bits):

$$\text{File size} = \text{resolution (in pixels)} \times \text{colour depth (bits per pixel)}$$

where resolution = width (pixels) × height (pixels).

Worked Example: Image File Size

Image: 100×80 pixels, 128 colours.

Colour depth: $2^7 = 128$, so **7 bits per pixel**.

File size $= 100 \times 80 \times 7 = 56{,}000$ bits $= 7{,}000$ bytes $= 6.84$ KiB.

Sound file size (in bits):

$$\text{Mono} = \text{sample rate (Hz)} \times \text{sample resolution (bits)} \times \text{duration (s)}$$

$$\text{Stereo} = 2 \times \text{sample rate} \times \text{sample resolution} \times \text{duration}$$

Worked Example: Sound File Size

Mono clip: 48 000 Hz sample rate, 24-bit resolution, 30 seconds.

File size $= 48{,}000 \times 24 \times 30 = 34{,}560{,}000$ bits $= 4{,}320{,}000$ bytes ≈ 4.12 MiB.

Data Compression

Reasons to compress a file:

- Less storage space required.
- Less bandwidth required for transmission, streaming, uploading, or downloading.
- Shorter transmission time; faster upload/download speeds.
- Files small enough to attach to email (accounts often have attachment size limits).
- Reduces data usage and cloud storage costs.

Lossy Compression

Permanently removes data considered unnecessary or imperceptible to humans. The original file **cannot** be restored.

Examples: JPEG (images), MP3 (audio), MP4 (video).

How lossy compression reduces an image file:

- A compression algorithm is applied (e.g., JPEG).
- Redundant/unnecessary data is removed (e.g., details the human eye cannot perceive).
- Colour depth is reduced (fewer bits per pixel).
- Image resolution is reduced (fewer pixels).
- Data is permanently deleted — the original **cannot** be restored.

How lossy compression reduces a sound file:

- A compression algorithm is applied (e.g., MP3, using perceptual music shaping).
- Sounds outside human hearing range are removed.
- Sample rate and/or sample resolution are reduced.
- Data is permanently deleted — the original **cannot** be restored.

Lossless Compression

Reduces file size without removing any data. The original file **can** be fully restored.

Examples: PNG (images), ZIP (files), RLE (run-length encoding).

How lossless compression (RLE) works:

- A compression algorithm is applied (e.g., RLE).
- Repeated patterns (e.g., consecutive pixels of the same colour, repeated characters in text) are identified.
- Each run of repetitions is indexed: stored as a (count, value) pair rather than repeating the value individually.
- No data is removed — the original file can always be fully restored.

Worked Example: Run-Length Encoding

A row of pixels: `RRRRRRGGGGBBBBB`

RLE stores this as: `(6,R)(4,G)(5,B)` — three pairs instead of 15 values.
The original sequence can be reconstructed exactly from these three pairs.

Comparison Table

Property	Lossy	Lossless
File size reduction	Greater	Smaller
Data lost?	Yes (permanently)	No
Original restorable?	No	Yes
Quality impact	Reduced	None
Typical formats	JPEG, MP3, MP4	PNG, ZIP, RLE
Best for	Media (minor loss acceptable)	Text, code (no loss tolerable)

Common Mistake

A musician wanting to edit a recording, or a programmer compressing source code, must use **lossless** compression: lossy permanently removes data that cannot be recovered, corrupting a program file or degrading a master recording. A website serving background images can use **lossy**: the minor quality reduction is acceptable and the smaller file size reduces page load time.

Exam Strategy

"Why lossless instead of lossy?": no data permanently lost; original file can be restored; quality is unchanged.

"Why lossy instead of lossless?": smaller file size; faster to upload/download; less storage/bandwidth required; minor quality loss is acceptable for the intended purpose.

Both structures appear repeatedly in past papers — learn the bullet points for each direction.