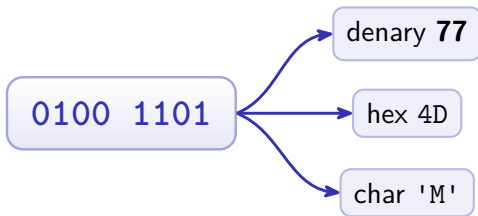


# Information Representation

## AS Computer Science ■ Topic 1

### A-Level Computer Science



## What we will cover

- 1 Number systems
- 2 Binary arithmetic
- 3 BCD & hexadecimal
- 4 Character codes
- 5 Images
- 6 Sound
- 7 Compression

$$200 = 128 + 64 + 8 = 11001000_2 = \text{C8}_{16}$$

# Three number systems

Three ways to write the **same number** – here, thirteen:

**denary**

▪ base 10  
digits 0–9

13

**binary**

▪ base 2  
0 and 1

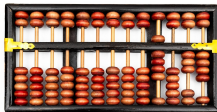
1101

**hexadecimal**

▪ base 16  
0–9, A–F

D

a **byte** = 8 **bits** ; one hex digit = 4 bits



*An abacus counts by place value too*

## Converting between bases

- **denary** → **binary**: subtract the largest **place value**
- **binary** → **hex**: group into **nibbles**
- **hex** → **denary**: digit × place value

| place value | 128      | 64 | 32 | 16 | 8        | 4 | 2 | 1 |
|-------------|----------|----|----|----|----------|---|---|---|
| bits        | 1        | 1  | 0  | 0  | 1        | 0 | 0 | 0 |
|             | 1100 = C |    |    |    | 1000 = 8 |   |   |   |

$$200 = 128 + 64 + 8 = 11001000_2 = C8_{16}$$

### Convert denary 200

$$200 = 128 + 64 + 8 \Rightarrow 1100 \ 1000; 1100=C, 1000=8 \Rightarrow \text{hex } C8.$$

## Binary vs decimal prefixes

Two families look alike – powers of 10 against powers of 2. A 1 TB drive holds  $10^{12}$  bytes; an OS reporting in TiB shows a *smaller* number for the same drive.

| Decimal       | Binary                 |
|---------------|------------------------|
| kilo = $10^3$ | kibi = $2^{10} = 1024$ |
| mega = $10^6$ | mebi = $2^{20}$        |
| giga = $10^9$ | gibi = $2^{30}$        |

kilo



$10^3 = 1000$

kibi



$2^{10} = 1024$

same name, different size

-128  
10000000

-1  
11111111

0  
00000000

127  
01111111

8-bit two's complement: -128...127 · MSB = sign bit

# Signed and unsigned integers

A bit pattern carries **no sign of its own** – the same byte is two numbers, read as **unsigned** or as **signed**.

| 8 bits             | as unsigned<br>every bit a place value | as signed<br>top bit is the sign |
|--------------------|----------------------------------------|----------------------------------|
| 00000000           | 0                                      | 0                                |
| 01111111           | 127                                    | +127                             |
| 10000000           | 128                                    | -128                             |
| 11111111           | 255                                    | -1                               |
| range for $n$ bits | $0 \dots 2^n - 1$                      | $-2^{n-1} \dots 2^{n-1} - 1$     |

## Addition & overflow

Add column by column from the right, carrying as in denary.

**Overflow** : the result needs more bits than the **register** can hold.

$$\begin{array}{r} 1111 \\ + 0001 \\ \hline \end{array}$$

overflow bit  $\rightarrow 10000$

## Subtraction by two's complement

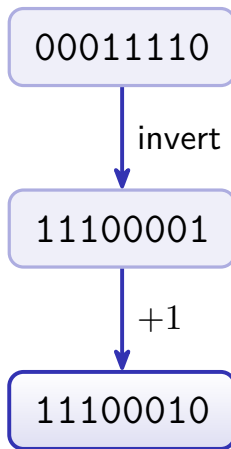
For  $A - B$ : form the **two's complement** of  $B$  (invert, add 1), add, drop the final carry.

**100 - 30 (8-bit)**

$00011110 \rightarrow 11100010$ ;

$01100100 + 11100010$

$\rightarrow 01000110 = 70. \checkmark$

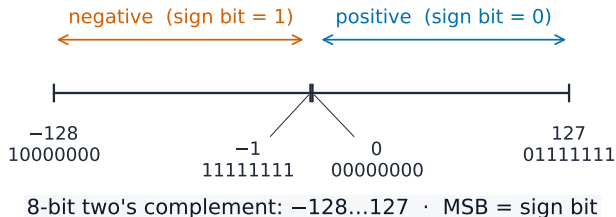


# Two's complement signed integers

- **sign bit** = **most significant bit** (1 = negative)
- read it: invert, +1, then negate
- range:  $-2^{n-1}$  to  $2^{n-1} - 1$

10110100

invert+1 = 01001100 = 76  $\Rightarrow$   
-76



## One's complement – the older scheme

**One's complement** : negate by **only inverting** (no +1).

- $+30 = 00011110$ ,  $-30 = 11100001$
- drawback: **two zeros**

Two's complement has one zero and shares a circuit for + and -.

00000000 = +0

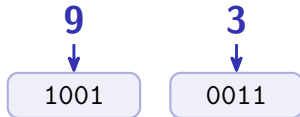
11111111 = -0

two zeros – wasteful



# Binary Coded Decimal

In **BCD** , each denary digit gets *its own 4 bits*.



In pure binary the same number is 0101 1101 – a *different pattern* of the same 8 bits.

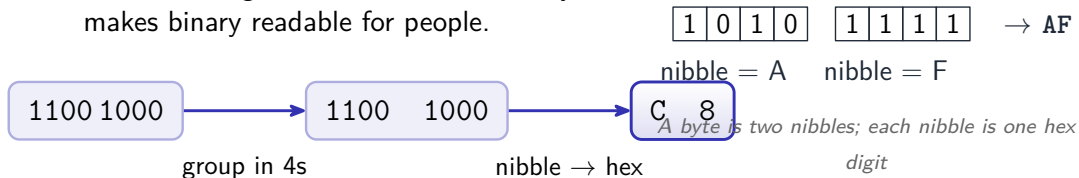
- only 0000–1001 are valid; 1010–1111 unused
- each digit drives a display directly, and money avoids binary rounding



*Drives a 7-segment display*

## Hexadecimal in practice

One hex digit = 4 bits, so one byte is exactly **two hex digits** – used for **memory addresses** (0x7FFE), colour codes and MAC addresses. Hex does **not** change the stored data – it only makes binary readable for people.



- memory address** 0x7FFE    ■    colour #FF8800
- MAC AC:DE:48:00:11:22

(space)

32

00100000

# ASCII

Text = numbers – each character has a **code point** in a **character set** .

**ASCII**: 7 bits, 128 code points    ▪    **Extended ASCII**: 8 bits, 256

| character | code (denary) | binary   |
|-----------|---------------|----------|
| A         | 65            | 01000001 |
| a         | 97            | 01100001 |
| 0         | 48            | 00110000 |
| (space)   | 32            | 00100000 |

# Unicode

A universal character set – almost every script.

- **encodings** : **UTF-8** (1–4 bytes), UTF-16, UTF-32
- **portable**, multilingual in one file
- larger files for English-only text

A → U+0041

→ U+4E2D

é → U+00E9

one code point each



**A**

$16 \times 16$  pixels (high)



**B**

$8 \times 8$  pixels



**C**

$4 \times 4$  pixels (low)

fewer, larger pixels  $\Rightarrow$  lower resolution, less detail

# Bitmap images

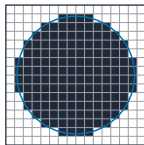
A **bitmap** stores every **pixel** in a grid, behind a **file header** that records the resolution and colour depth.

- **resolution** :  $w \times h$
- **colour depth** : bits/pixel

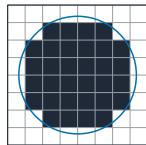
size in bits =  $w \times h \times \text{depth}$

**3000 × 2000 at 24 bpp**

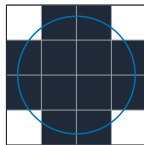
$1.44 \times 10^8 \text{ bits} \div 8 \div 1024^2 \approx$   
**17 MiB**



**A**  
16 × 16 pixels (high)



**B**  
8 × 8 pixels



**C**  
4 × 4 pixels (low)

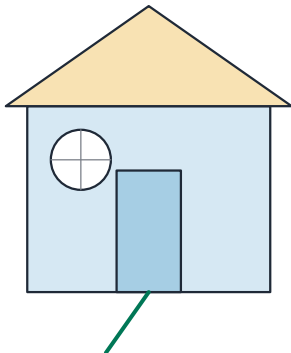
fewer, larger pixels  $\Rightarrow$  lower resolution, less detail

## Changing settings

- **lower image resolution** → smaller file, less detail; blocky when enlarged (**screen resolution** is a different thing – the monitor's own pixel grid).
- **lower colour depth** → smaller file, but smooth shades show banding.
- **higher** of either → larger file, better quality.

# Vector graphics

A **vector graphic** stores the **instructions** to draw the image – a **drawing list** of **drawing objects**, each a geometric **primitive** with its own properties. Redrawn crisply at **any size**.



*the file is a list of shapes:*

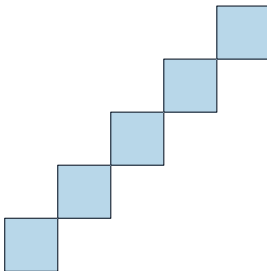
- 
- |                                                                                   |                         |                                     |
|-----------------------------------------------------------------------------------|-------------------------|-------------------------------------|
|  | <b>triangle</b> — roof  | 3 points                            |
|  | <b>circle</b> — window  | $(0.75, 1.85)$ , $r\ 0.42$          |
|  | <b>rectangle</b> — body | $(0, 0)$ , $3.4 \times 2.6$         |
|  | <b>rectangle</b> — door | $(1.25, 0)$ , $0.9 \times 1.7$      |
|  | <b>line</b> — path      | $(1.7, 0) \rightarrow (1.1, -0.85)$ |

# Bitmap vs vector

| Use               | Better        |
|-------------------|---------------|
| Photograph        | <b>bitmap</b> |
| Painting, texture | bitmap        |
| Logo, icon        | vector        |
| Engineering dwg   | vector        |

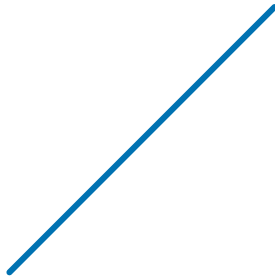
Bitmap wins on **smooth tonal detail** per area; vector **scales without blur**.

bitmap, enlarged

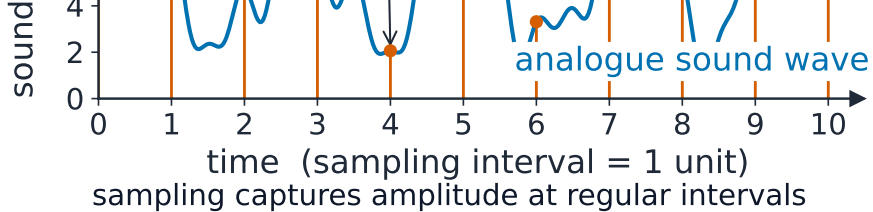


edges look jagged

vector, enlarged



edges stay smooth



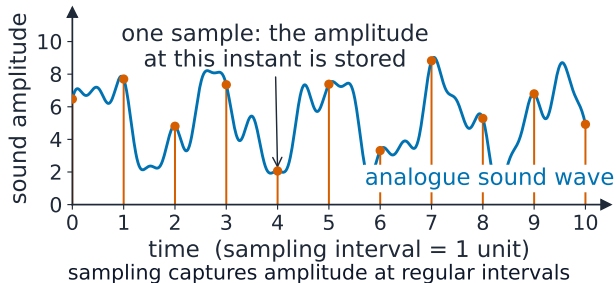
# Sampling sound

**Analogue data** → **digital data**  
by **sampling** :

- **sampling rate** (CD 44.1 kHz)
- **sampling resolution** – bits per **amplitude** , set by **quantisation**

**10 s stereo CD**

$44100 \times 16 \times 10 \times 2 \approx 1.68 \text{ MiB}$



**Run-length encoding**

store “the next  $n$  values are  $x$ ”

**Dictionary (ZIP, PNG)**

repeats  $\rightarrow$  one short reference

**Huffman coding**

common symbols get the shortest codes

**Images (JPEG)**

drops detail the eye barely sees

**Sound (MP3, AAC)**

drops pitches we hear less well

**Video**

each frame stores only what changed

# Lossless vs lossy

**lossless** – recovers data exactly (ZIP, PNG).

**lossy** – drops detail for smaller files (JPEG, MP3).

**Compression** saves storage and **bandwidth**. Video does both at once: **spatial** within each frame, **temporal** between them – most frames store only what *changed*.

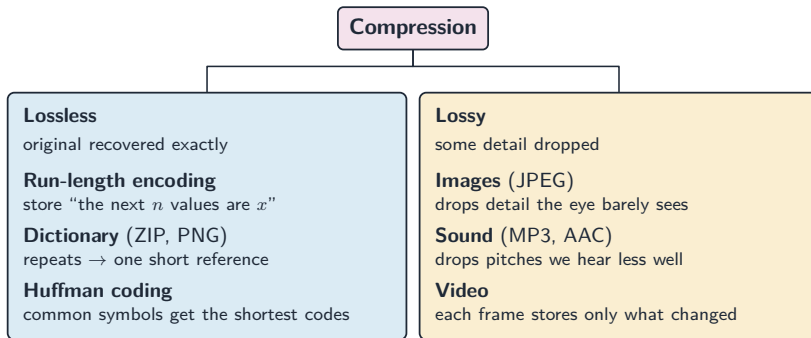
original  100%

lossless   $\approx 70\%$

lossy   $\approx 30\%$

same picture, three file sizes

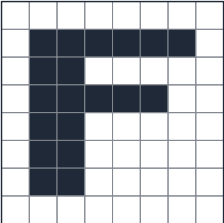
# Compression methods



**lossless** recovers the original **exactly** – use it where every bit must survive (documents, code, medical images). **lossy** **drops** detail for much smaller files (JPEG, MP3, streaming video).

# Run-length encoding

Store each **run** of identical values once, as a **(count, value)** pair – **lossless**, so the original comes back exactly. Best on **flat areas**; useless on **noisy** data, where a run of one costs more than the value it replaces.

| 8×8 bitmap                                                                        | binary   | RLE code |
|-----------------------------------------------------------------------------------|----------|----------|
|  | 00000000 | 8W       |
|                                                                                   | 01111110 | 1W 6B 1W |
|                                                                                   | 01100000 | 1W 2B 5W |
|                                                                                   | 01111100 | 1W 5B 2W |
|                                                                                   | 01100000 | 1W 2B 5W |
|                                                                                   | 01100000 | 1W 2B 5W |
|                                                                                   | 01100000 | 1W 2B 5W |
|                                                                                   | 00000000 | 8W       |

**Key:** 1 = black (B), 0 = white (W). RLE stores *count + colour*: e.g. 01111110 → 1W 6B 1W

# Dictionary coding

**Dictionary methods** store each repeated **sequence** once in a **dictionary**; every occurrence then becomes its short **index**. **Lossless**, and best where whole sequences repeat – text, source code, flat colour in a PNG.

**source**    ABC ABC ABC XYZ    12 characters

**dictionary**    1 → ABC    2 → XYZ    stored once each

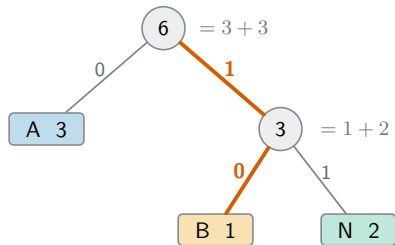
**encoded**    1 1 1 2    4 tokens + dictionary

---

The decoder rebuilds the dictionary as it reads, so the original returns exactly — **lossless** (ZIP, PNG).

# Huffman coding

Give the **commonest** symbol the **shortest** code – a **variable-length code**, still **lossless**. No code starts another, so the decoder needs no separator, and the average code length approaches the data's **entropy**.



Walk **down from the root**, writing a bit at each branch: left = 0, right = 1.

**A** = 0

**B** = 10      right, then left

**N** = 11

each circle joins the two smallest counts

BANANA becomes 100110110 — **10 bits**, against 12 for fixed 2-bit codes: the commonest letter is nearest the root.

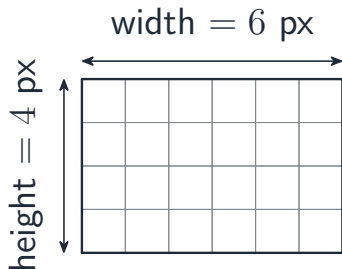
## Exam tips

- Show working for base conversions: denary  $\rightarrow$  binary by place values, binary  $\rightarrow$  hexadecimal in **nibbles** (groups of 4 bits).
- For **two's complement** the MSB is negative; to negate, **invert and add 1**; watch for **overflow** when the sign bit flips wrongly.
- Distinguish **bitmap** (pixels; file size = width  $\times$  height  $\times$  colour depth) from **vector** (drawing commands; scales without loss).
- Sound file size depends on **sample rate**  $\times$  **bit depth**  $\times$  **time** – more of each means better quality but a bigger file.
- Compare **lossless vs lossy** compression and give a use for each.



## Topic 1 – key takeaways

- 1 denary / binary / hex; one hex digit = 4 bits
- 2 two's complement for signed integers
- 3 text = code points; images **bitmap**/**vector**; sound **sampled**
- 4 **compression**: lossless or lossy



$$\text{pixels} = 6 \times 4 = 24$$

$$\text{size} = 24 \times 8 = \mathbf{192} \text{ bits}$$

(**8** = colour depth, bits per pixel)

# Check yourself

- 1 Convert denary 173 to binary and hex.
- 2 8-bit two's-complement  $11111011 = ?$
- 3  $1920 \times 1080$  at 24 bpp – size in MiB?
- 4 Why is JPEG lossy but PNG lossless?



16 pixels  
 $\Rightarrow$  **6W 4B 6W**  
 (3 runs, not 16)

Answers: 10101101/AD ■ -5 ■  $\approx 5.9$  MiB ■ JPEG drops unseen detail; PNG keeps every pixel.