Chapter 1.1: Basic Concepts

Computer Fundamentals and Introduction to Programming

How a computer works | The binary system | Data representation | Types of computers | Information technology

Sercan KÜLCÜ · Giresun University

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/computer/slides/Chapter_01_01_Basic_Concepts.html

INTRODUCTION

Learning Outcomes

How a computer works

Explain the input, processing, output and storage stages.

Digital representation

Interpret the role of bits, bytes, the binary number system and logical operations.

Types of computers

Compare computers by purpose and capacity.

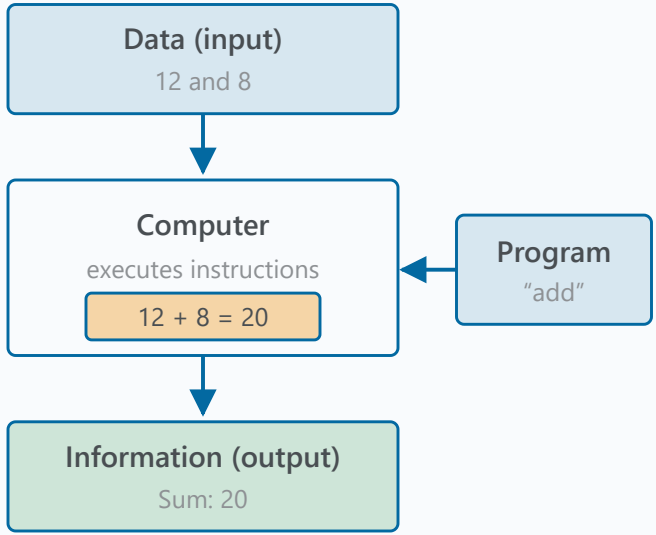
Information technology

Explain how data becomes information and what IT does in everyday life.

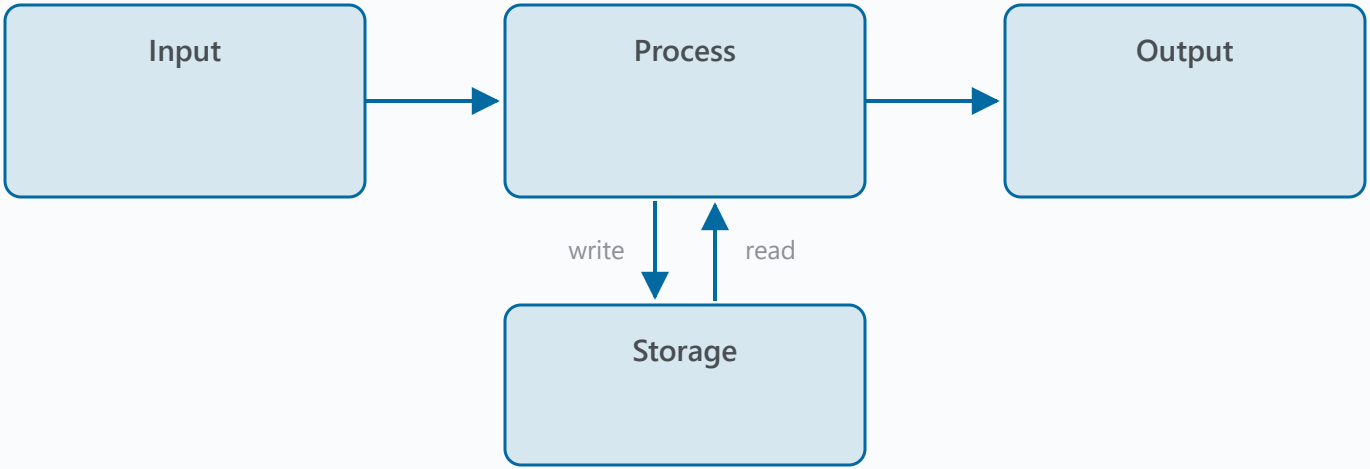
What Is a Computer?

A computer is an electronic system that processes data according to the instructions in a program.

- **Data**: raw values before processing. **Information**: a result you can interpret.
- It takes in data from users or other systems.
- It processes the data and produces a meaningful result.
- It presents the result or stores it for later use.

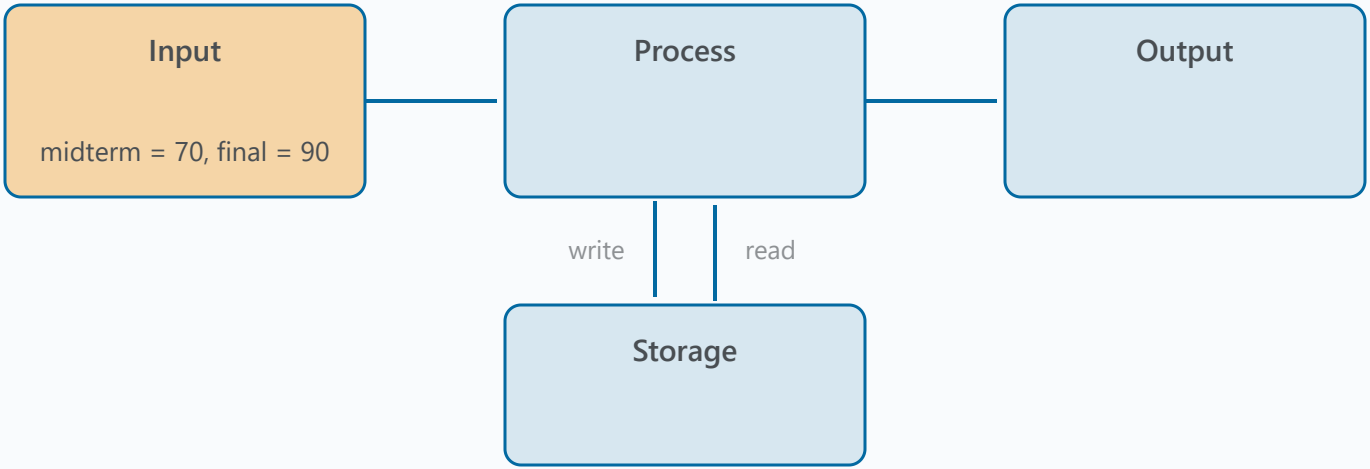


The Basic Cycle



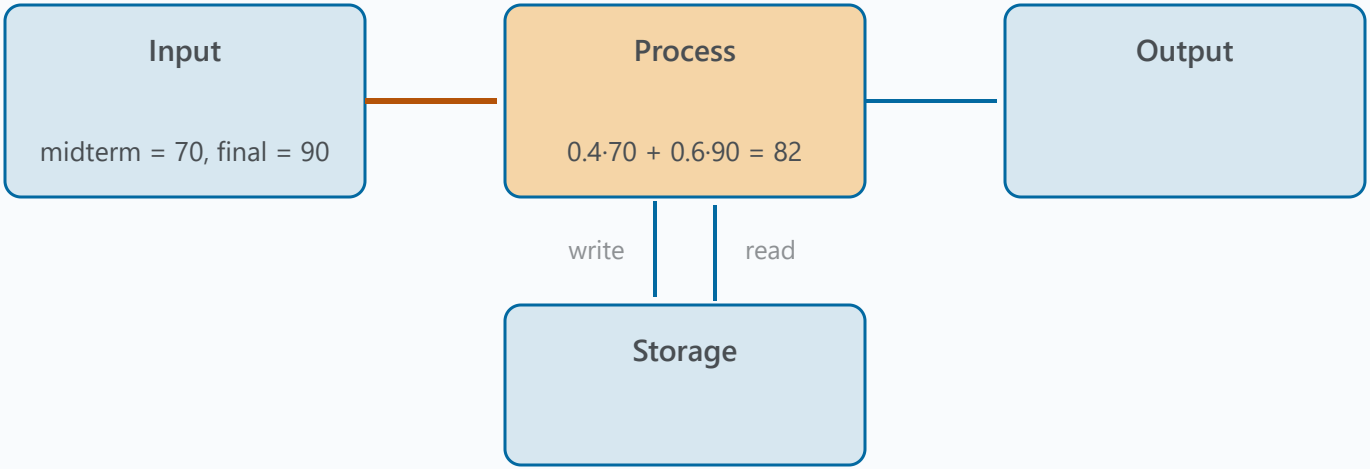
Every task goes through the same four stages. Example: computing a grade in a student information system.

The Basic Cycle



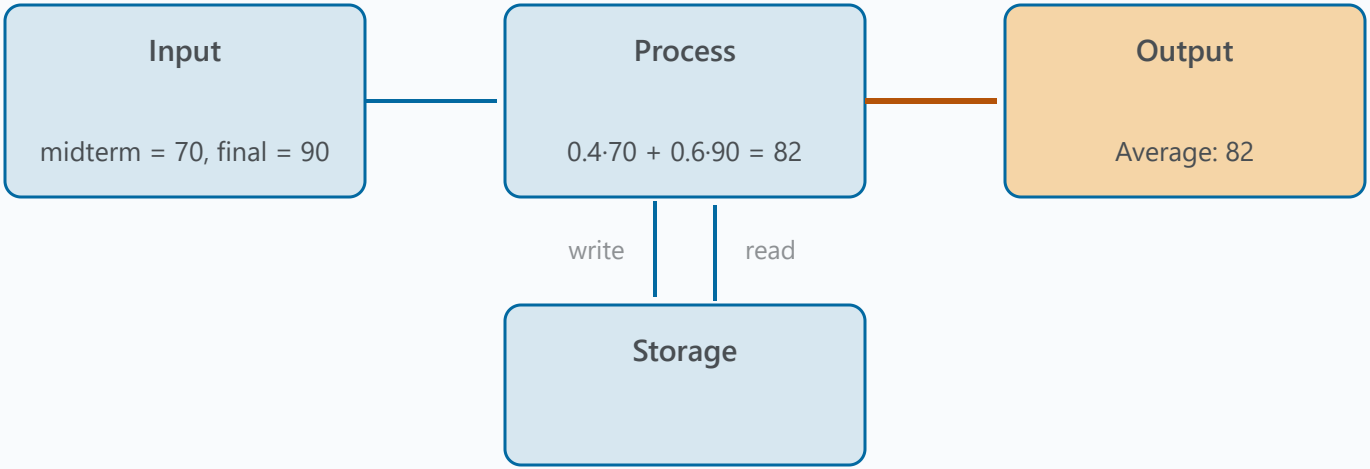
Input: The instructor types midterm = 70 and final = 90 into a form. The keyboard sends these values to the computer.

The Basic Cycle



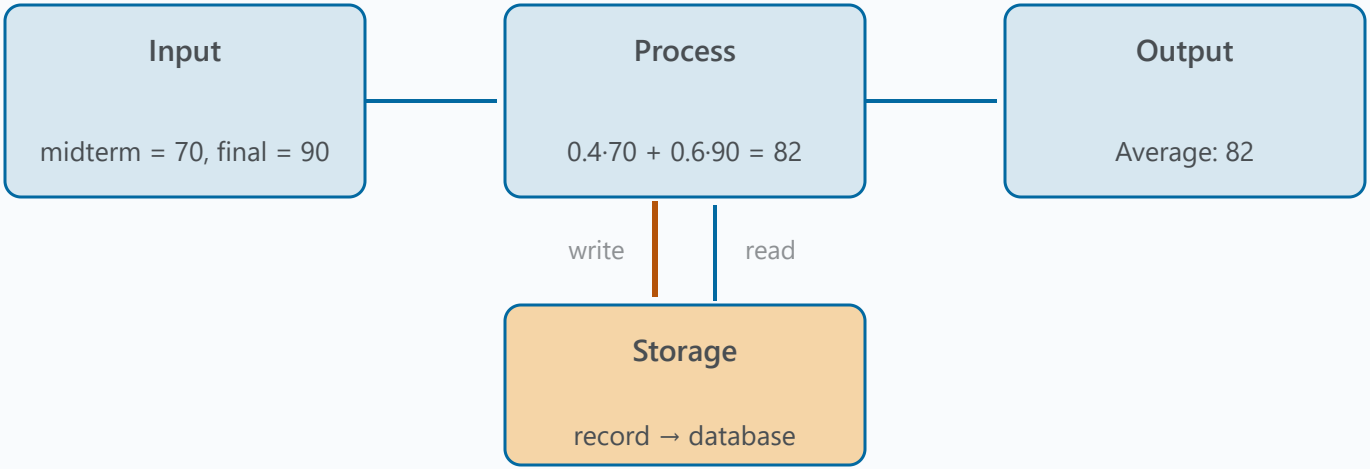
Process: The CPU applies the formula: 0.4 × 70 + 0.6 × 90 = **82**. This is an arithmetic operation.

The Basic Cycle



Output: The screen shows "Average: 82". The monitor is an output device.

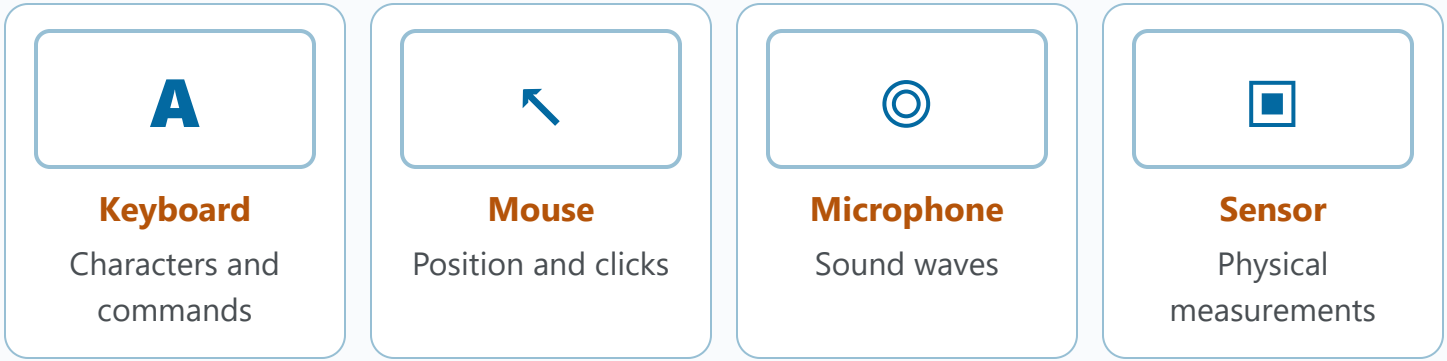
The Basic Cycle

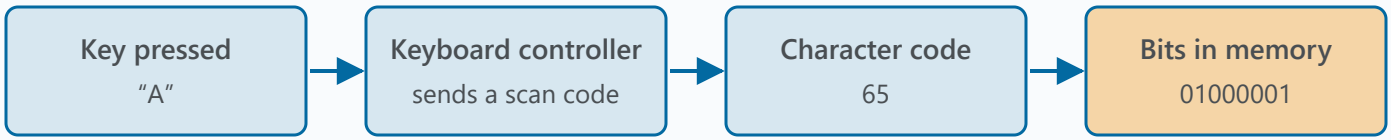


Storage: The grade record is written to a database, where it can be read and processed again later (two-way arrow).

Input

Input is how a computer receives data from a user, a sensor or another system.

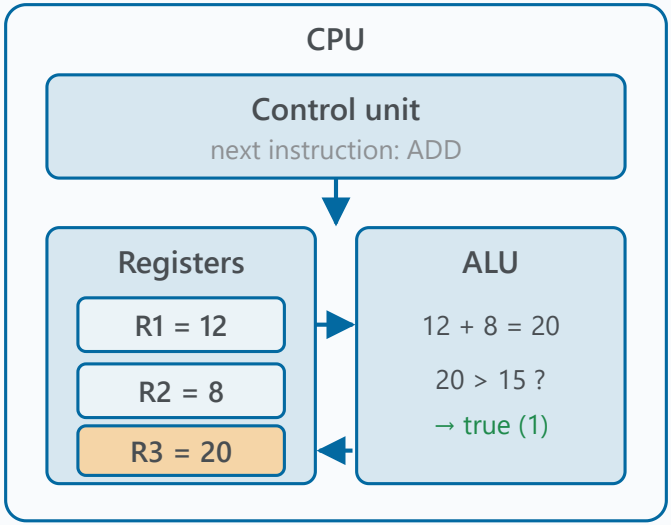




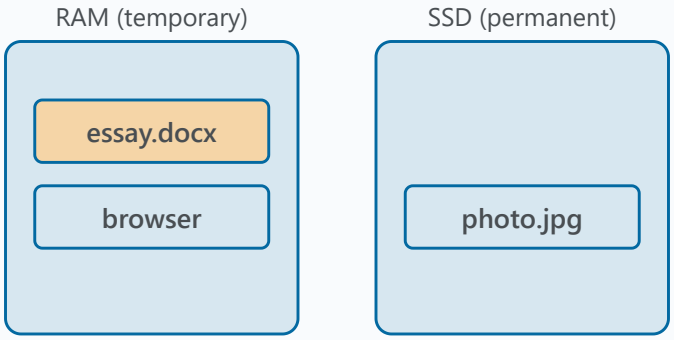
Every input device turns a physical event into numbers (bits).

Processing

- The central processing unit (CPU) executes program instructions in order; the **control unit** manages that sequence.
- The **ALU** (arithmetic logic unit) does the calculations: 12 + 8 = 20.
- Logical operations make comparisons and decisions possible.
- Both data and instructions are represented in binary.



Output and Storage



Open documents and running programs live in **RAM**: fast access.

Step 1 / 3

Output

Processed information is presented on a monitor, printer or speaker.

`result → screen`

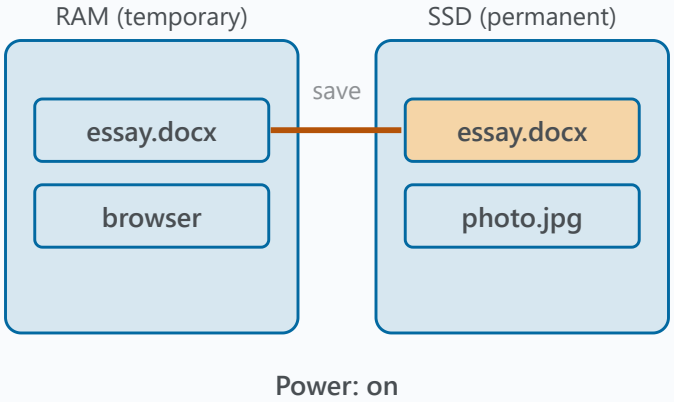
Temporary storage (RAM)

Fast, but loses its contents when the power goes off.

Permanent storage

SSD, hard disk, USB drive: keeps data without power.

Output and Storage



Save copies the document from RAM to the **SSD**.

Output

Processed information is presented on a monitor, printer or speaker.

`result → screen`

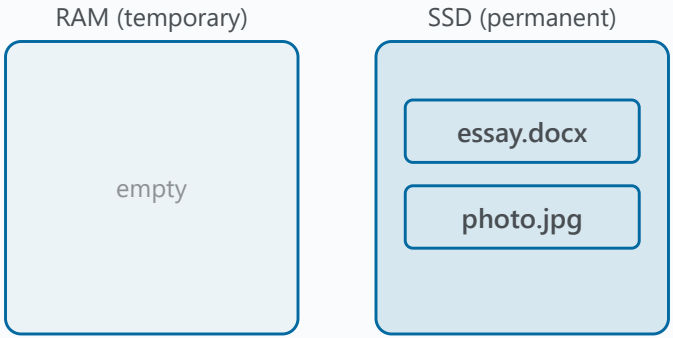
Temporary storage (RAM)

Fast, but loses its contents when the power goes off.

Permanent storage

SSD, hard disk, USB drive: keeps data without power.

Output and Storage



When the power goes off, RAM is **wiped**; files on the SSD are **kept**.

Step 3 / 3

Output

Processed information is presented on a monitor, printer or speaker.

`result → screen`

Temporary storage (RAM)

Fast, but loses its contents when the power goes off.

Permanent storage

SSD, hard disk, USB drive: keeps data without power.

Everything Becomes a Number

Text

Characters are stored as numeric codes.

`A → 65 → 01000001`

Mouse movement

A position on screen becomes a pair of coordinates.

`(x, y) → (420, 180)`

Images

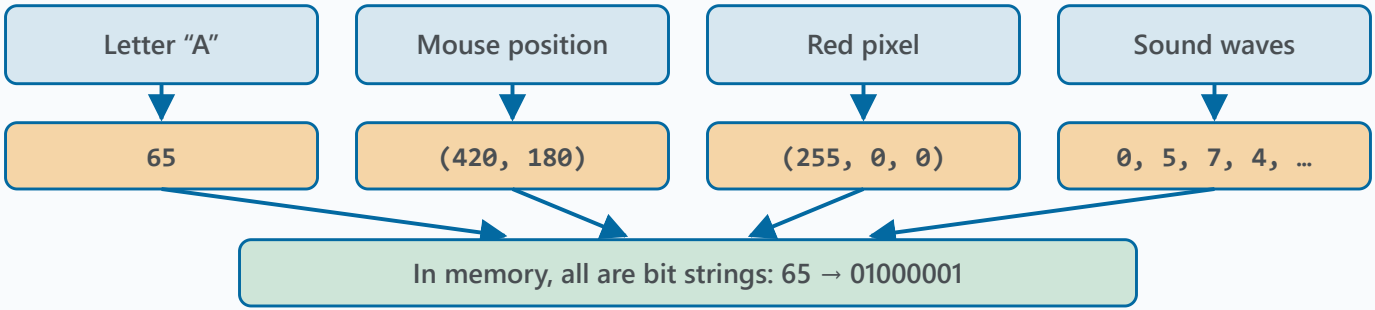
Each pixel is made of color components.

`RGB(255, 0, 0)`

Sound

A sound wave is sampled at regular intervals.

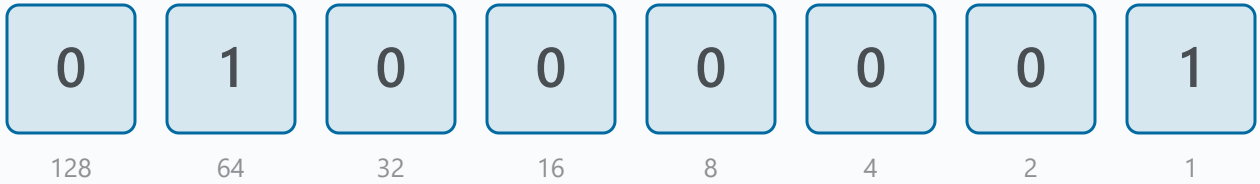
`amplitude → list of numbers`



Bits and Binary: From Binary to Decimal

A **bit** (binary digit) takes only the values 0 and 1. 8 bits = 1 byte.

Place values (powers of 2):

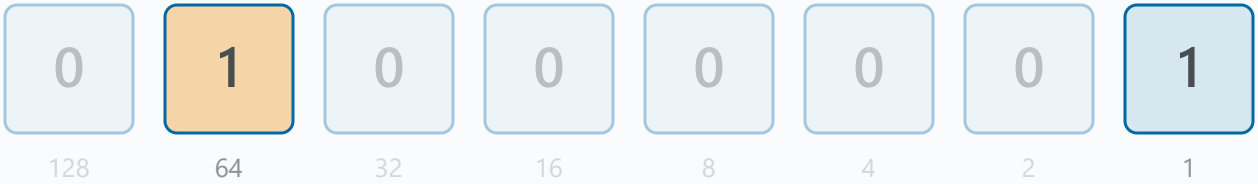


An 8-bit string: from right to left, each position's value **doubles**: 1, 2, 4, … 128.

Bits and Binary: From Binary to Decimal

A **bit** (binary digit) takes only the values 0 and 1. 8 bits = 1 byte.

Place values (powers of 2):



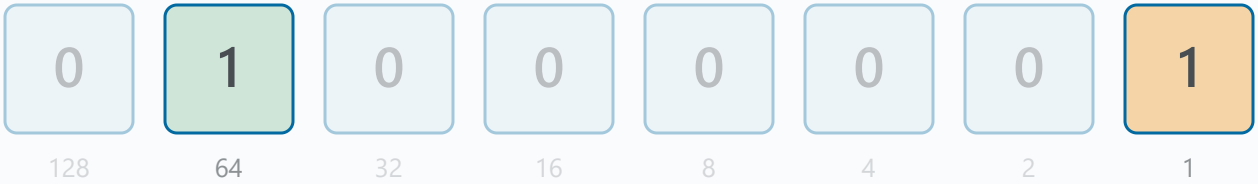
sum = 64 = 64

The bit in position 6 is **1** → its value **64** is added to the sum. Zero bits contribute nothing.

Bits and Binary: From Binary to Decimal

A **bit** (binary digit) takes only the values 0 and 1. 8 bits = 1 byte.

Place values (powers of 2):



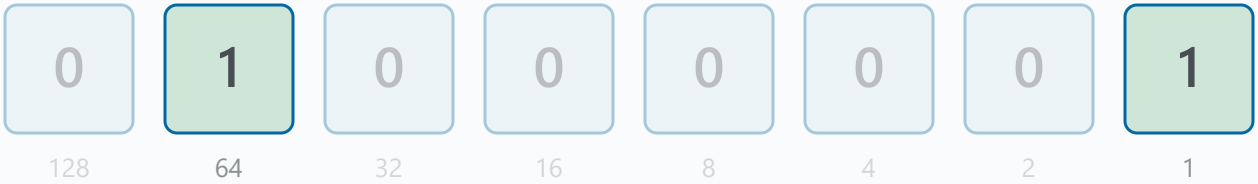
sum = 64 + 1 = 65

The bit in position 0 is **1** → its value **1** is added to the sum. Zero bits contribute nothing.

Bits and Binary: From Binary to Decimal

A **bit** (binary digit) takes only the values 0 and 1. 8 bits = 1 byte.

Place values (powers of 2):



sum = 64 + 1 = 65

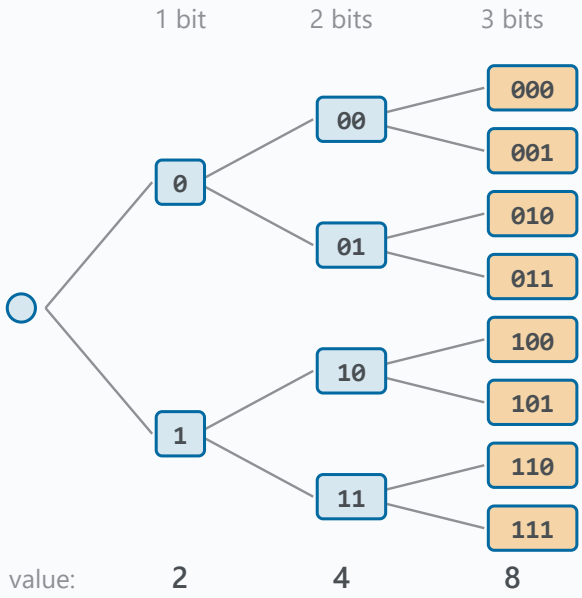
ASCII 65 → “A”

$01000001_2 = 65_{10}$. In the ASCII table, 65 is the letter **A**.

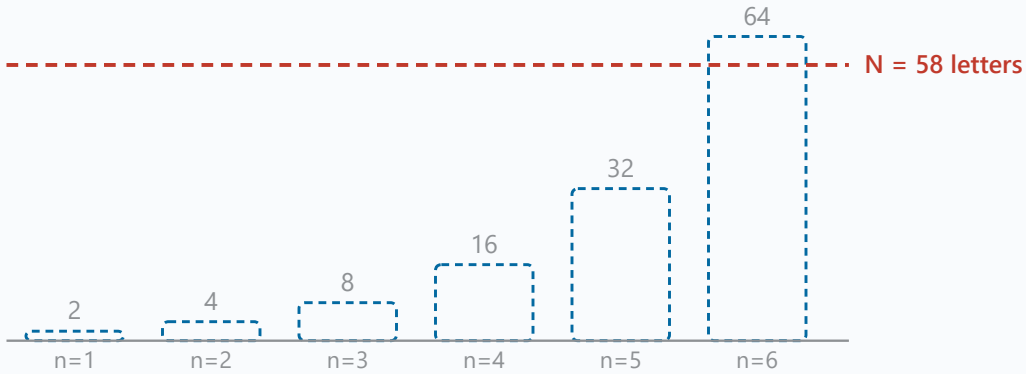
What Binary Digits Are Worth

Bits	Combinations	Unsigned range
1 bit	$2^1 = 2$	0–1
2 bits	$2^2 = 4$	0–3
8 bits	$2^8 = 256$	0–255
10 bits	$2^{10} = 1{,}024$	0–1,023
32 bits	2^{32}	0–4,294,967,295

Each extra bit doubles the number of distinct values: n bits → 2^n values.



How Many Bits? 58 Letters



The Turkish alphabet has 29 letters; coding upper and lower case separately needs **N = 58** distinct values. How many bits, at minimum?

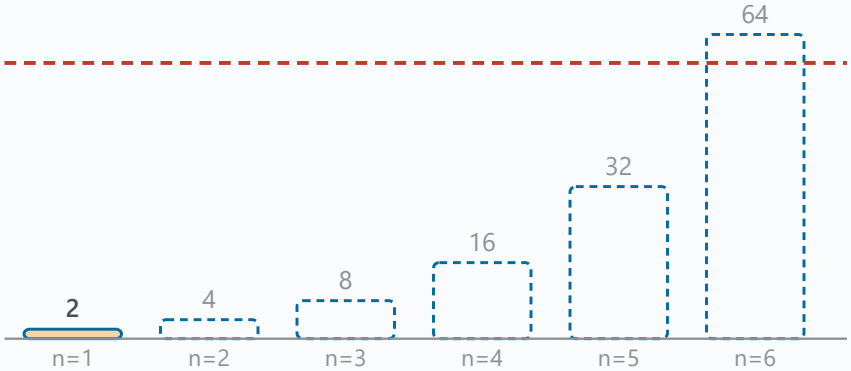
Step 1 / 8

```
▶ N = 58   (29 upper + 29 lower case)
2 n = 1; capacity = 2
3 while capacity < N: n = n + 1; capacity =
  capacity × 2
4 capacity ≥ N → n bits suffice
```

VARIABLES

`N = 58` `n = –` `capacity = –`

How Many Bits? 58 Letters



1 bit → 2 codes

2 < 58 → not enough

One bit gives only 2 codes (0 and 1). Since 2 < 58, the loop continues.

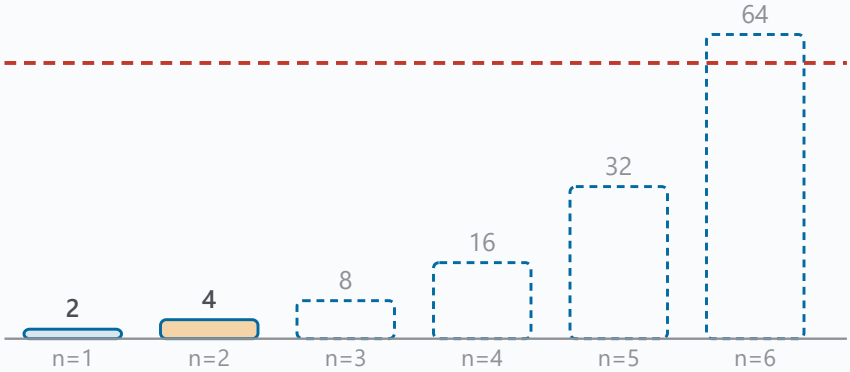
Step 2 / 8

```
1 N = 58   (29 upper + 29 lower case)
▶ n = 1; capacity = 2
3 while capacity < N: n = n + 1; capacity =
  capacity × 2
4 capacity ≥ N → n bits suffice
```

VARIABLES

`N = 58` `n = 1` `capacity = 2`

How Many Bits? 58 Letters



2 bits → 4 codes

4 < 58 → not enough

One more bit doubles the codes: **4**. Still fewer than 58.

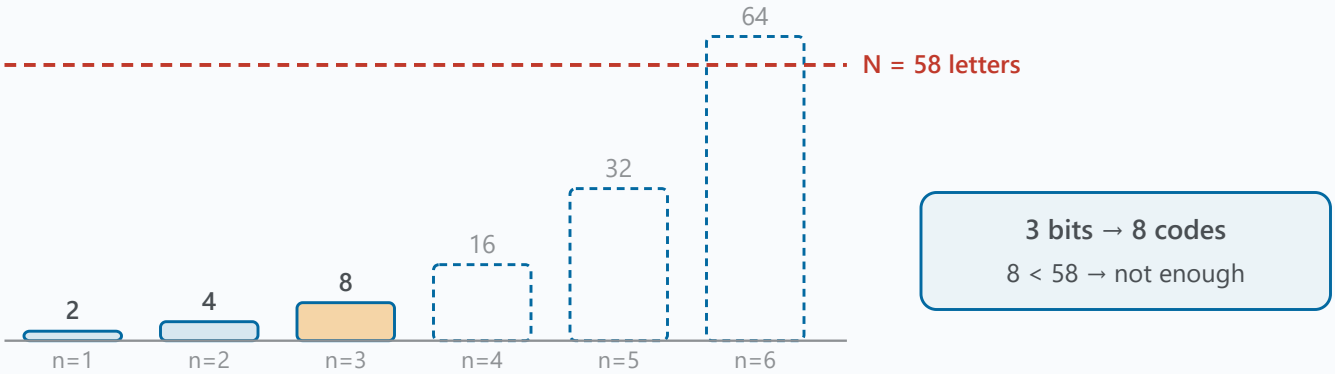
Step 3 / 8

```
1 N = 58   (29 upper + 29 lower case)
2 n = 1; capacity = 2
▶ while capacity < N: n = n + 1; capacity = capacity × 2
4 capacity ≥ N → n bits suffice
```

VARIABLES

`N = 58` `n = 2` `capacity = 4`

How Many Bits? 58 Letters



One more bit doubles the codes: **8**. Still fewer than 58.

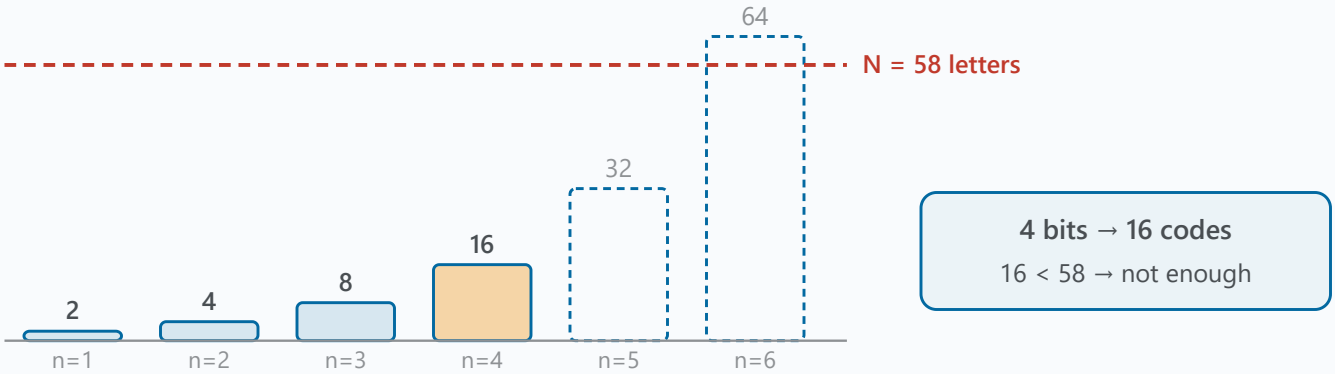
Step 4 / 8

```
1 N = 58   (29 upper + 29 lower case)
2 n = 1; capacity = 2
▶ while capacity < N: n = n + 1; capacity = capacity × 2
4 capacity ≥ N → n bits suffice
```

VARIABLES

`N = 58` `n = 3` `capacity = 8`

How Many Bits? 58 Letters



One more bit doubles the codes: **16**. Still fewer than 58.

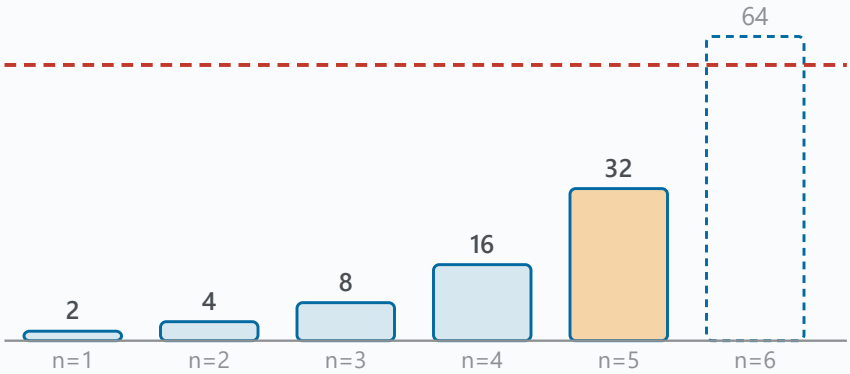
Step 5 / 8

```
1 N = 58   (29 upper + 29 lower case)
2 n = 1; capacity = 2
▶ while capacity < N: n = n + 1; capacity = capacity × 2
4 capacity ≥ N → n bits suffice
```

VARIABLES

`N = 58` `n = 4` `capacity = 16`

How Many Bits? 58 Letters



5 bits → 32 codes

32 < 58 → not enough

One more bit doubles the codes: **32**. Still fewer than 58.

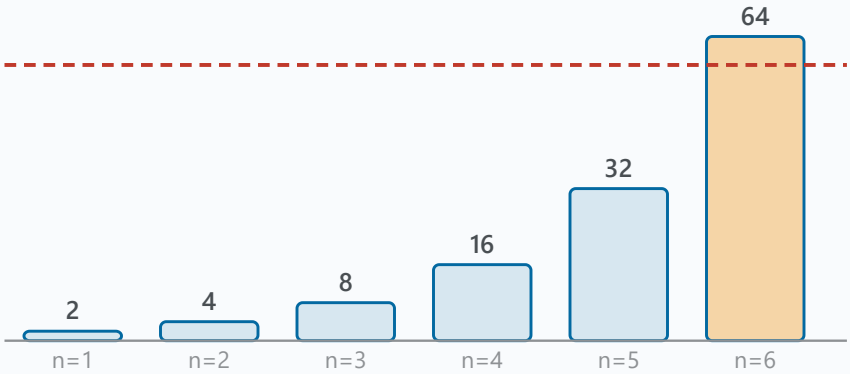
Step 6 / 8

```
1 N = 58   (29 upper + 29 lower case)
2 n = 1; capacity = 2
▶ while capacity < N: n = n + 1; capacity = capacity × 2
4 capacity ≥ N → n bits suffice
```

VARIABLES

`N = 58` `n = 5` `capacity = 32`

How Many Bits? 58 Letters



6 bits → 64 codes
64 ≥ 58 → enough

6 bits → **64** codes. 64 ≥ 58: the condition fails and the loop stops.

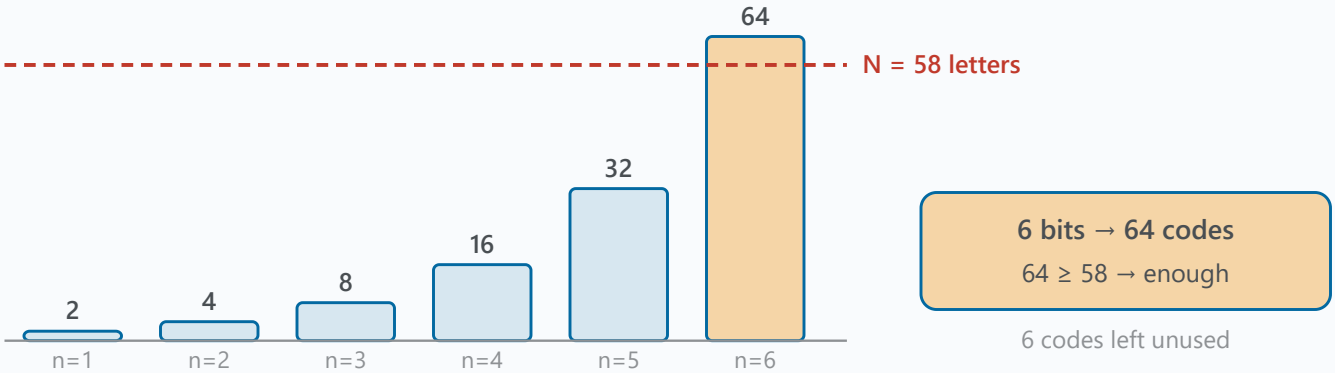
Step 7 / 8

```
1 N = 58   (29 upper + 29 lower case)
2 n = 1; capacity = 2
▶ while capacity < N: n = n + 1; capacity = capacity × 2
4 capacity ≥ N → n bits suffice
```

VARIABLES

`N = 58` `n = 6` `capacity = 64`

How Many Bits? 58 Letters



Answer: **6 bits**; 64 − 58 = 6 codes go unused. ASCII, designed for English, needs more than 64 codes for 52 letters, 10 digits, punctuation and control characters: 7 bits (128). Turkish letters (ç, ğ, ı, İ, ö, ş, ü) are not in ASCII; they need Unicode (UTF-8).

Step 8 / 8

```
1 N = 58   (29 upper + 29 lower case)
2 n = 1; capacity = 2
3 while capacity < N: n = n + 1; capacity =
  capacity × 2
▶ capacity ≥ N → n bits suffice
```

VARIABLES

`N = 58` `n = 6` `capacity = 64` `unused = 6`

Decimal to Binary: 45

Divide by 2, note the remainder

Read remainders bottom-up

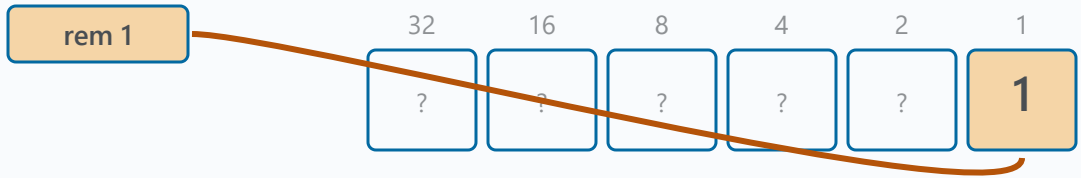


Let's convert **45** to binary. Method: divide by 2 until the quotient is 0, noting the remainder (0 or 1) at each step.

Decimal to Binary: 45

Divide by 2, note the remainder

45 ÷ 2 = 22



Read remainders bottom-up

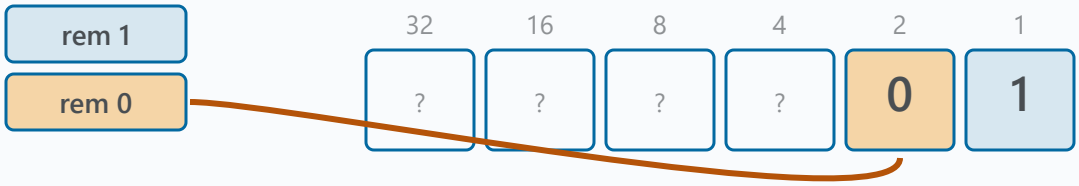
45 ÷ 2 = 22, **rem 1** (odd → remainder 1). The first remainder is the rightmost (ones) digit.

Decimal to Binary: 45

Divide by 2, note the remainder

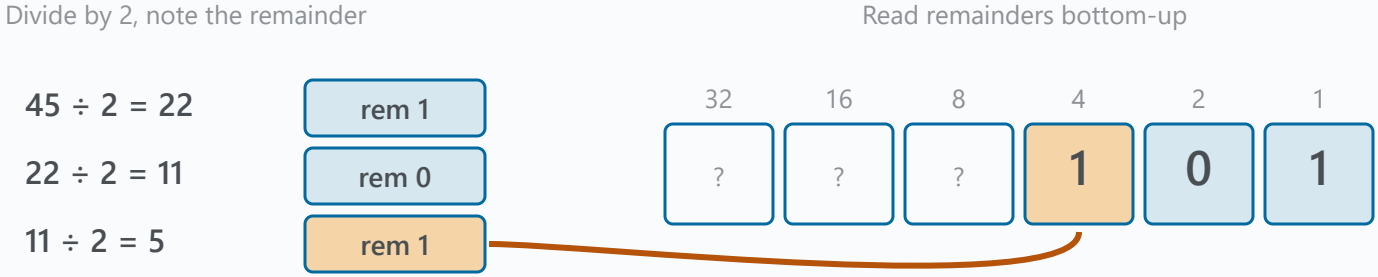
45 ÷ 2 = 22

22 ÷ 2 = 11



22 ÷ 2 = 11, **rem 0** (even → remainder 0). This remainder goes one place to the left.

Decimal to Binary: 45



11 ÷ 2 = 5, **rem 1** (odd → remainder 1). This remainder goes one place to the left.

Decimal to Binary: 45

Divide by 2, note the remainder

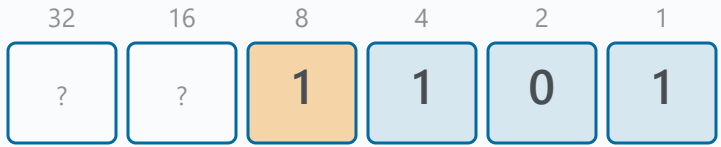
45 ÷ 2 = 22 — rem 1

22 ÷ 2 = 11 — rem 0

11 ÷ 2 = 5 — rem 1

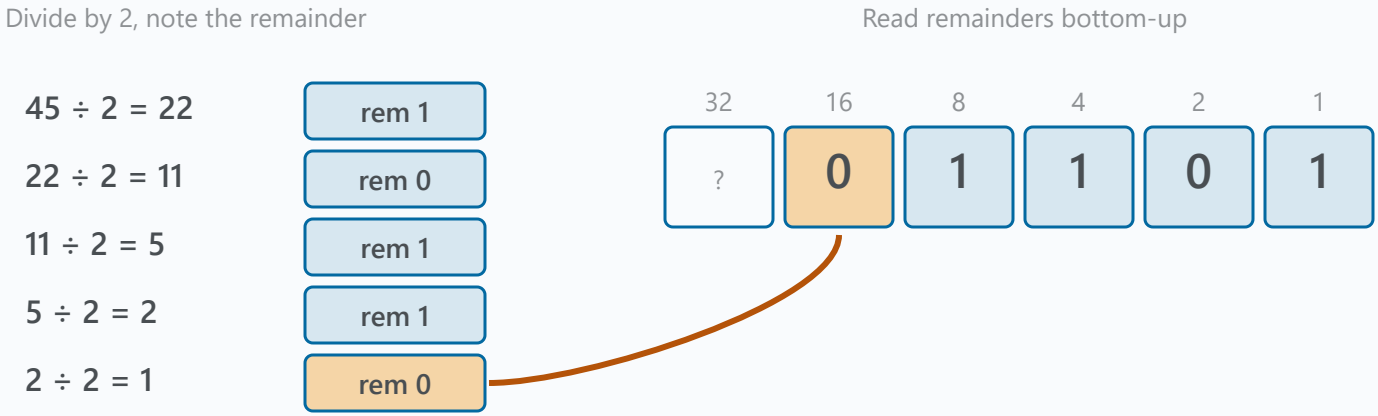
5 ÷ 2 = 2 — rem 1

Read remainders bottom-up



5 ÷ 2 = 2, **rem 1** (odd → remainder 1). This remainder goes one place to the left.

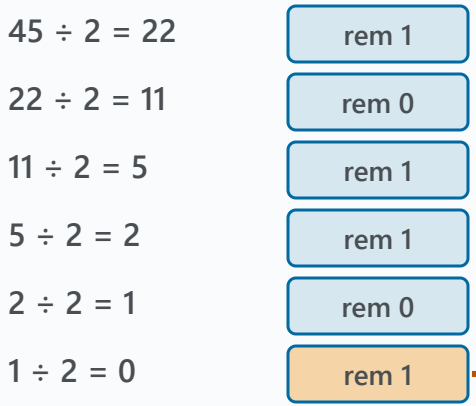
Decimal to Binary: 45



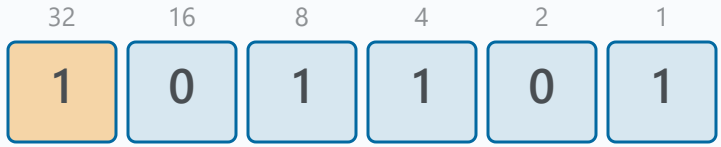
2 ÷ 2 = 1, **rem 0** (even → remainder 0). This remainder goes one place to the left.

Decimal to Binary: 45

Divide by 2, note the remainder



Read remainders bottom-up



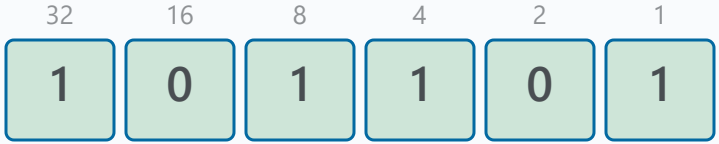
1 ÷ 2 = 0, **rem 1** (odd → remainder 1). This remainder goes one place to the left. The quotient is 0, so we stop.

Decimal to Binary: 45

Divide by 2, note the remainder

45 ÷ 2 = 22	rem 1
22 ÷ 2 = 11	rem 0
11 ÷ 2 = 5	rem 1
5 ÷ 2 = 2	rem 1
2 ÷ 2 = 1	rem 0
1 ÷ 2 = 0	rem 1

Read remainders bottom-up



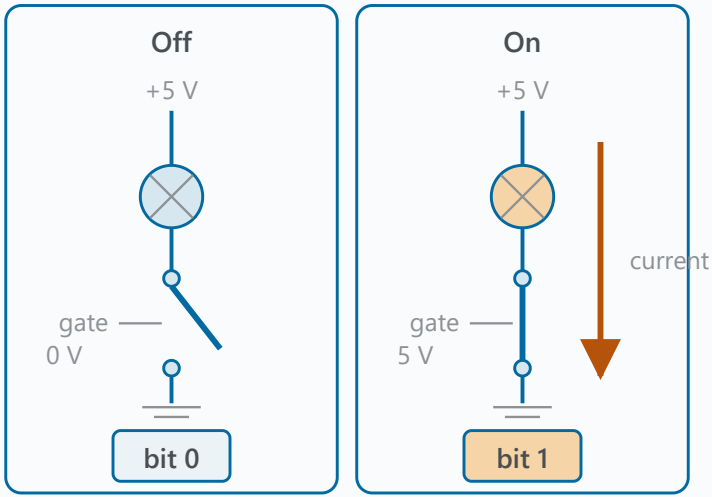
101101_2

check: 32 + 8 + 4 + 1 = 45

Reading the remainders **from the last division to the first** gives **$45_{10} = 101101_2$**. Check: add up the place values.

Transistors

- A processor is built from a huge number of transistors.
- A transistor works like a tiny switch that controls electric current.
- The gate voltage turns the transistor **on** (current flows) or **off** (no current); these two states are read as 1 and 0.
- Billions of switches working together make complex operations possible.



DATA REPRESENTATION

Text and Numbers

Numbers

Integers and floating-point numbers are stored with specific binary encoding rules.

`13 → 00001101`

Characters

ASCII and Unicode map characters to numeric codes.

`A → U+0041`

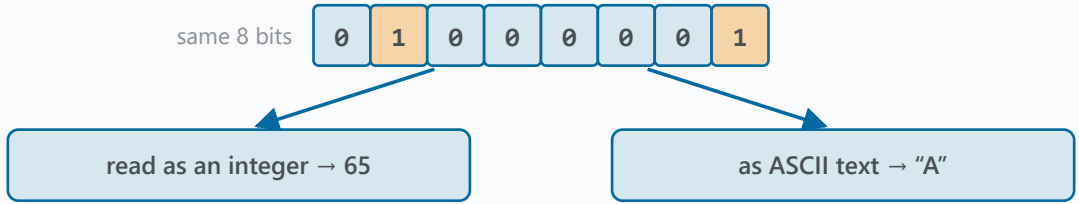
Encoding

The encoding standard decides what a given bit string means.

Unicode

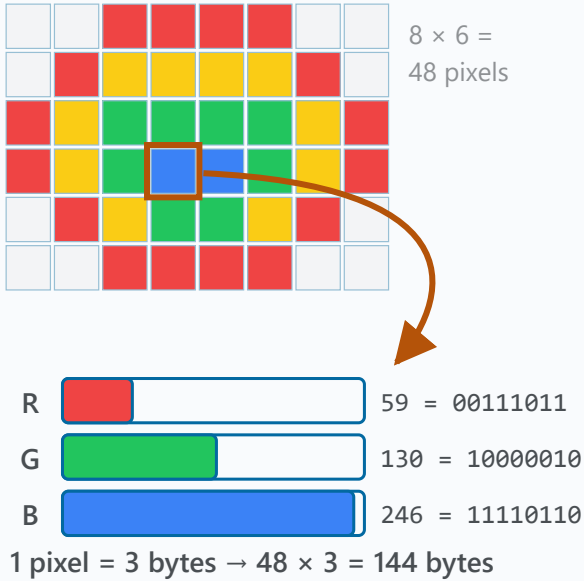
Provides a large character set covering many alphabets and symbols.

`Ğ · 中 · €`



Images

- A digital image is made of small colored cells called pixels.
- Resolution is the number of pixels horizontally and vertically.
- In the RGB model, a color is defined by red, green and blue components.
- Each component is a number from 0 to 255 (1 byte). `RGB(255, 255, 255)` is white, `RGB(0, 0, 0)` is black.



DATA REPRESENTATION

Color Depth and Image Size

Bits / pixel	Colors	Typical use
1	$2^1 = 2$	Line art, fax, scanned documents
8	$2^8 = 256$	Grayscale or palette images (GIF)
24	$2^{24} = 16{,}777{,}216$	Photos: 8 bits each for R, G, B
32	16.7 million + transparency	UI graphics, PNG (alpha channel)

Size formula

Uncompressed image = width × height × bit depth ÷ 8 bytes.

Example: 800 × 600

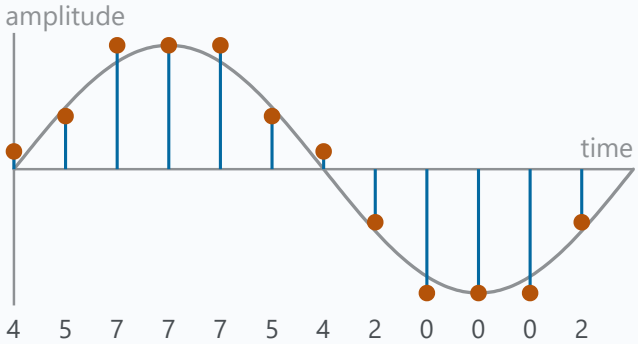
24 bits: 800 × 600 × 3 = 1,440,000 bytes ≈ 1.44 MB (1.37 MiB). The same image at 1 bit: 60,000 bytes ≈ 60 KB (58.6 KiB).

More color depth gives smoother gradients but larger files; with an 8-bit palette, a sky gradient looks banded.

Audio and Video

Audio

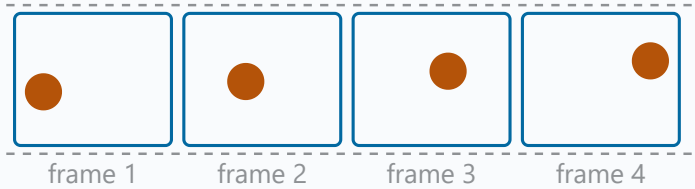
The analog wave is measured at equal intervals (**sampling**); each measurement is rounded to a number.



Samples: 4, 5, 7, 7, 7, 5, 4, 2, 0, 0, 0, 2

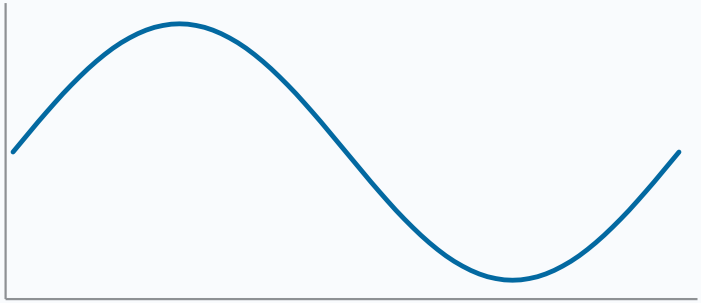
Video

Showing frames in quick succession creates the illusion of motion. Audio is usually stored in a separate track.



30 fps: 30 frames per second

Digitizing Sound: Sample, Quantize, Encode



Analog signal

microphone voltage
varies continuously in time
a value at every instant:
infinitely many points
a computer can't store that

A microphone turns sound into a continuously varying voltage. This **analog** signal has a value at every instant, and infinitely many values cannot be stored.

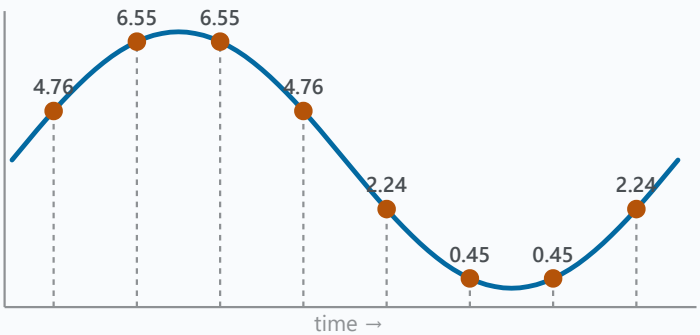
Step 1 / 6

```
▶ analog signal: continuous wave
2 sample: measure at equal intervals
3 quantize: round to nearest level (3 bits → 0…7)
4 encode: write each level in 3 bits
5 size = sample rate × bit depth × channels × time
6 lossy compression (MP3)
```

VARIABLES

`samples = –` `levels = –` `bits = –`

Digitizing Sound: Sample, Quantize, Encode



Sampling

8 measurements, equally spaced

changes in between are lost

sampling rate: measurements per second (Hz)

CD: 44,100 Hz

Sampling: the signal is measured at equal intervals. The measurements are decimals (4.76, 6.55, …); whatever happens between two samples is not recorded.

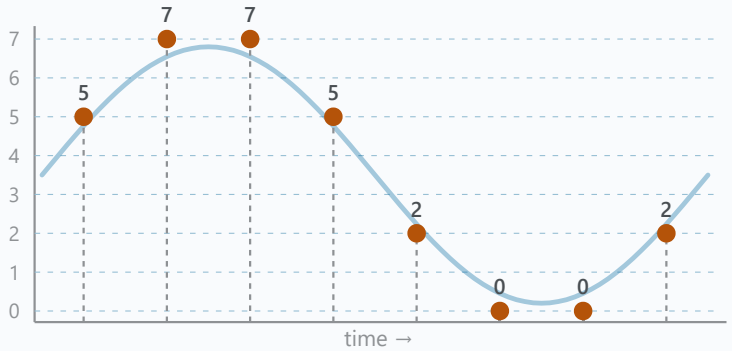
Step 2 / 6

```
1 analog signal: continuous wave
▶ sample: measure at equal intervals
3 quantize: round to nearest level (3 bits → 0…7)
4 encode: write each level in 3 bits
5 size = sample rate × bit depth × channels × time
6 lossy compression (MP3)
```

VARIABLES

`samples = 8` `levels = –` `bits = –`

Digitizing Sound: Sample, Quantize, Encode



Quantization

3 bits → 2^3 = 8 levels (0…7)

each sample is rounded

to nearest level: 4.76 → 5

max error: 0.45 levels

16 bits → 65,536 levels

Quantization: 3 bits allow only 8 levels, so each sample is rounded to the nearest one. The difference (at most 0.45) is the quantization error.

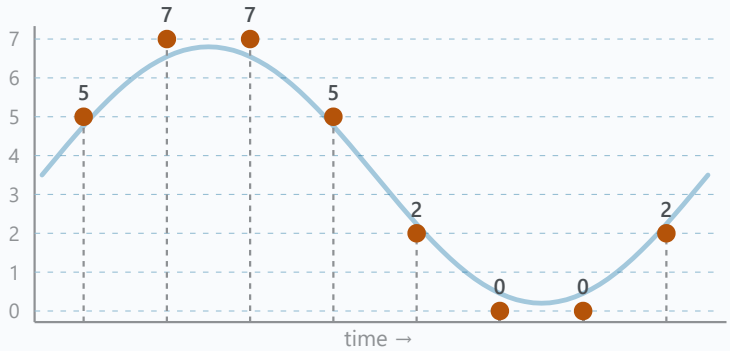
Step 3 / 6

```
1 analog signal: continuous wave
2 sample: measure at equal intervals
▶ quantize: round to nearest level (3 bits → 0…7)
4 encode: write each level in 3 bits
5 size = sample rate × bit depth × channels × time
6 lossy compression (MP3)
```

VARIABLES

`samples = 8` `levels = 5,7,7,5,2,0,0,2` `bits = –`

Digitizing Sound: Sample, Quantize, Encode



3-bit codes

101	111	111	101
010	000	000	010

8 samples × 3 bits = 24 bits

Encoding: each level is written as a 3-bit binary number; 5 → 101, 7 → 111. Eight samples take 24 bits.

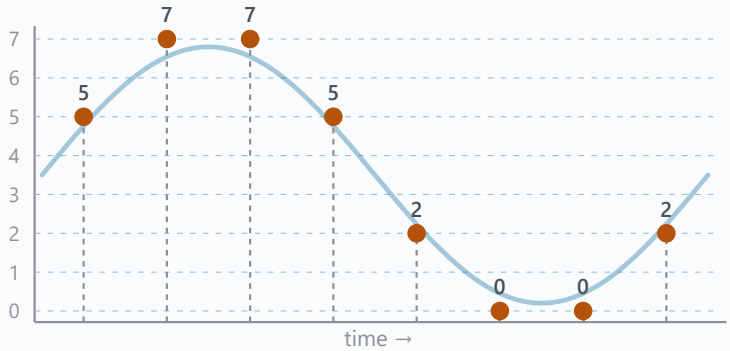
Step 4 / 6

```
1 analog signal: continuous wave
2 sample: measure at equal intervals
3 quantize: round to nearest level (3 bits → 0…7)
▶ encode: write each level in 3 bits
5 size = sample rate × bit depth × channels × time
6 lossy compression (MP3)
```

VARIABLES

`samples = 8` `levels = 5,7,7,5,2,0,0,2` `bits = 24`

Digitizing Sound: Sample, Quantize, Encode



CD quality, 1 minute

44,100/s × 16 bits × 2 channels

= 1,411,200 bit/s

× 60 s = 84,672,000 bits

÷ 8 = 10,584,000 bytes

≈ 10.6 MB (10.1 MiB)

Real audio takes thousands of samples per second. CD quality: 44,100 Hz, 16 bits, 2 channels → one minute ≈ **10.6 MB** (10.1 MiB).

Step 5 / 6

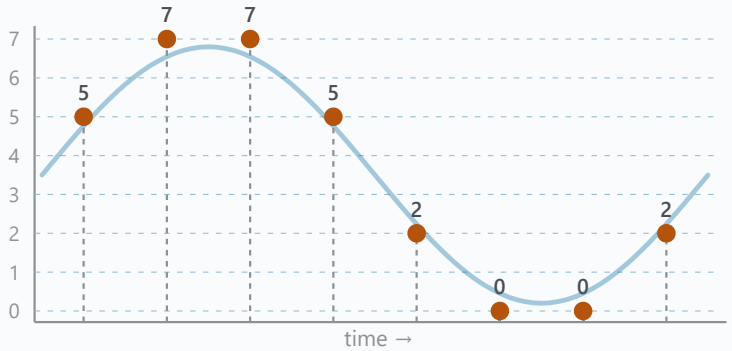
```
1 analog signal: continuous wave
2 sample: measure at equal intervals
3 quantize: round to nearest level (3 bits → 0…7)
4 encode: write each level in 3 bits
▶ size = sample rate × bit depth × channels × time
6 lossy compression (MP3)
```

VARIABLES

`sample rate = 44,100 Hz` `bit depth = 16`

`channels = 2` `1 min = 10,584,000 bytes`

Digitizing Sound: Sample, Quantize, Encode



1 minute of music

Uncompressed (WAV)

≈ 10.6 MB (10.1 MiB)

MP3, 128 kbit/s

960,000 bytes ≈ 0.96 MB

Lossy formats such as MP3 drop details the ear barely notices: at 128 kbit/s one minute is 960,000 bytes, about 11 times smaller.

Step 6 / 6

```
1 analog signal: continuous wave
2 sample: measure at equal intervals
3 quantize: round to nearest level (3 bits → 0…7)
4 encode: write each level in 3 bits
5 size = sample rate × bit depth × channels × time
▶ lossy compression (MP3)
```

VARIABLES

`WAV 1 min = 10,584,000 bytes`

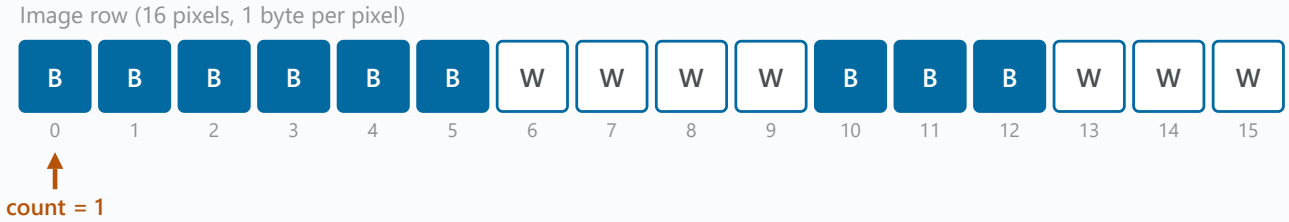
`MP3 1 min = 960,000 bytes`

Lossless and Lossy Compression

	Lossless	Lossy
After decompression	The original data comes back bit for bit	You get a close approximation of the original
Method	Shortens repetitions (RLE, ZIP)	Drops detail the eye and ear barely notice
Example formats	ZIP, PNG, FLAC	JPEG, MP3, MP4 (H.264)
Typical reduction	Usually 2–3×	10× or more
Suitable data	Text, programs, spreadsheets, medical images	Photos, music, video

A lossy file degrades a little more with every edit-and-save; discarded detail never comes back. Changing even one bit of a program or text can ruin it, so those are only ever compressed losslessly.

Run-Length Encoding (RLE)



The row has two colors: **B** (blue) and **W** (white). At the first pixel, count is 1 and the current color is B.

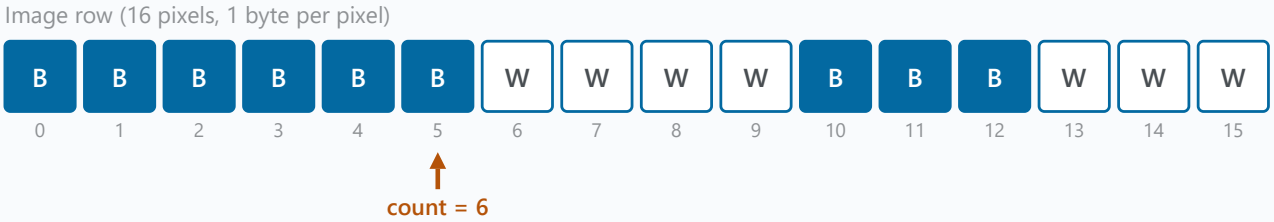
Step 1 / 7

```
▶ count = 1; color = first pixel
2 if next pixel has same color: count = count + 1
3 on color change: write (count, color); count = 1
4 at end of row: write last pair
5 size: 16 bytes → 4 pairs × 2 bytes
6 worst case: B W B W …
```

VARIABLES

`count = 1` `color = B` `output = –`

Run-Length Encoding (RLE)



Pixels 1–5 are also B: count goes up by one at each. At pixel 5, count = 6.

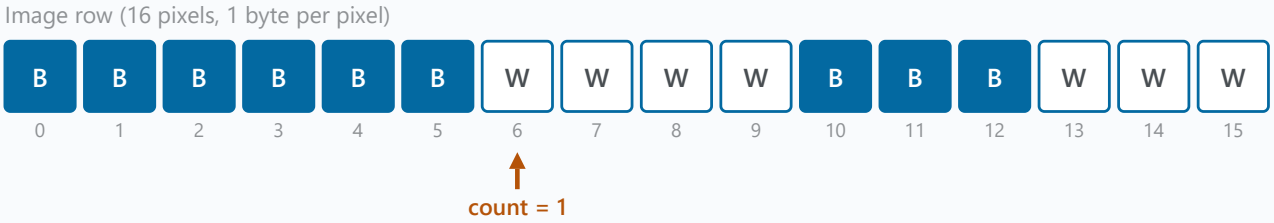
Step 2 / 7

```
1 count = 1; color = first pixel
▶ if next pixel has same color: count = count + 1
3 on color change: write (count, color); count = 1
4 at end of row: write last pair
5 size: 16 bytes → 4 pairs × 2 bytes
6 worst case: B W B W …
```

VARIABLES

`count = 6` `color = B` `output = –`

Run-Length Encoding (RLE)



Pixel 6 is W: the color changed. The finished run is written as **(6, B)**; count resets to 1 and the new color is W.

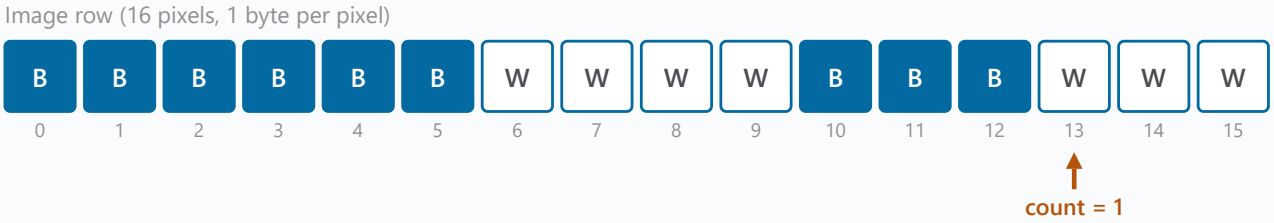
Step 3 / 7

```
1 count = 1; color = first pixel
2 if next pixel has same color: count = count + 1
▶ on color change: write (count, color); count = 1
4 at end of row: write last pair
5 size: 16 bytes → 4 pairs × 2 bytes
6 worst case: B W B W …
```

VARIABLES

`count = 1` `color = W` `output = (6,B)`

Run-Length Encoding (RLE)



The same rule fires twice more: four W → (4, W), three B → (3, B). A new W run starts at pixel 13.

Step 4 / 7

```
1 count = 1; color = first pixel
2 if next pixel has same color: count = count + 1
▶ on color change: write (count, color); count = 1
4 at end of row: write last pair
5 size: 16 bytes → 4 pairs × 2 bytes
6 worst case: B W B W …
```

VARIABLES

`count = 1` `color = W`

`output = (6,B) (4,W) (3,B)`

Run-Length Encoding (RLE)



End of row: the last open run **(3, W)** is written too. The output has 4 pairs.

Step 5 / 7

```
1 count = 1; color = first pixel
2 if next pixel has same color: count = count + 1
3 on color change: write (count, color); count = 1
▶ at end of row: write last pair
5 size: 16 bytes → 4 pairs × 2 bytes
6 worst case: B W B W …
```

VARIABLES

`count = 3` `color = W`

`output = (6,B) (4,W) (3,B) (3,W)`

Run-Length Encoding (RLE)

Image row (16 pixels, 1 byte per pixel)



Output: `(6, B)` `(4, W)` `(3, B)` `(3, W)`

Decoding: (6, B) → BBBBBB, (4, W) → WWWW … the row comes back exactly: lossless.

Each pair is 2 bytes: 4 × 2 = **8 bytes**, half of the original 16. Decoding restores the row exactly; nothing is lost.

Step 6 / 7

```
1 count = 1; color = first pixel
2 if next pixel has same color: count = count + 1
3 on color change: write (count, color); count = 1
4 at end of row: write last pair
▶ size: 16 bytes → 4 pairs × 2 bytes
6 worst case: B W B W …
```

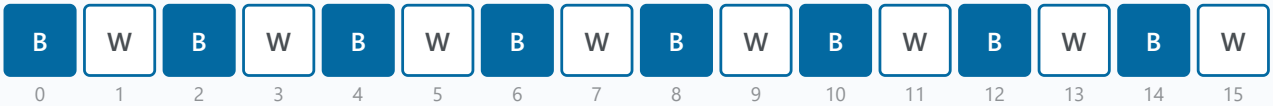
VARIABLES

`original = 16 bytes`

`compressed = 8 bytes` `ratio = 50%`

Run-Length Encoding (RLE)

Image row (16 pixels, 1 byte per pixel)



Output:

16 pairs × 2 B = 32 bytes: bigger than the original!

Worst case: if the color changes at every pixel, every run has length 1; 16 pairs = 32 bytes. RLE only pays off with long runs (line art, icons, scans).

Step 7 / 7

```
1 count = 1; color = first pixel
2 if next pixel has same color: count = count + 1
3 on color change: write (count, color); count = 1
4 at end of row: write last pair
5 size: 16 bytes → 4 pairs × 2 bytes
▶ worst case: B W B W …
```

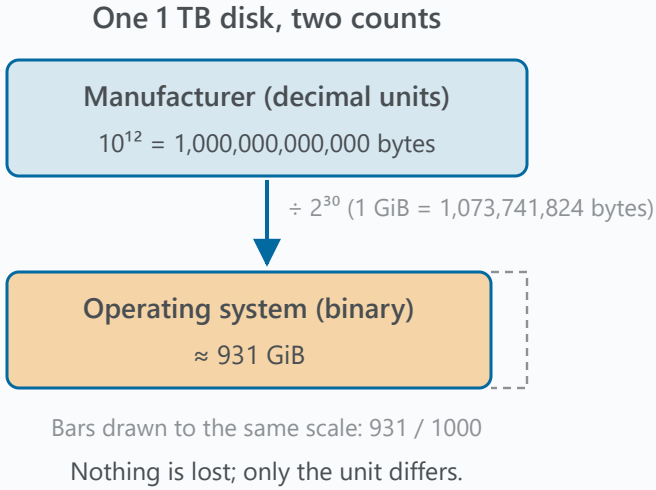
VARIABLES

`original = 16 bytes`

`compressed = 32 bytes` `ratio = 200%`

Units of Memory

Decimal (SI)	Value	Binary (IEC)	Value
bit	0 or 1 (binary digit)		
byte (B)	8 bits		
KB	1,000 B	KiB	1,024 B
MB	1,000 KB	MiB	1,024 KiB
GB	1,000 MB	GiB	1,024 MiB
TB	1,000 GB	TiB	1,024 GiB
PB	1,000 TB	PiB	1,024 TiB



Disk makers use SI units (1 TB = 10^{12} bytes). The operating system often computes the binary value but still labels it “GB”: a 1 TB disk ≈ 931 GiB, shown as “931 GB”.

The Scale of Units



bit: the smallest unit, 0 or 1. Example: a single switch.

Step 1 / 6

Bar heights are not to scale; each step is roughly a thousand times larger.

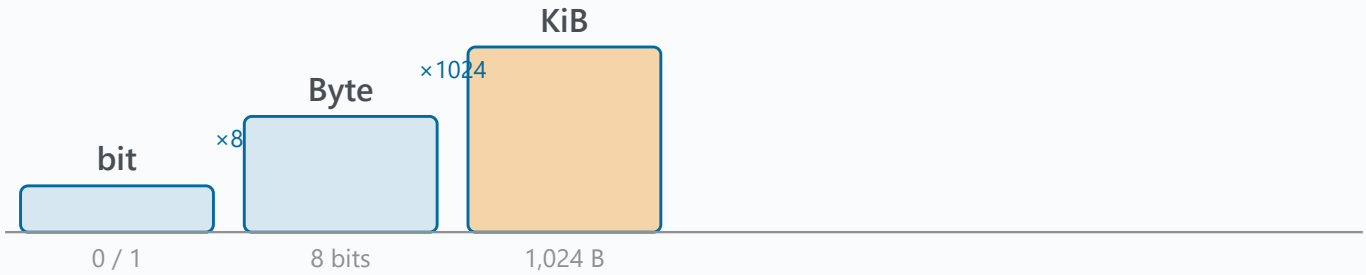
The Scale of Units



1 byte = 8 bits. A byte can take 2^8 = 256 values; example: one character, “A”.

Bar heights are not to scale; each step is roughly a thousand times larger.

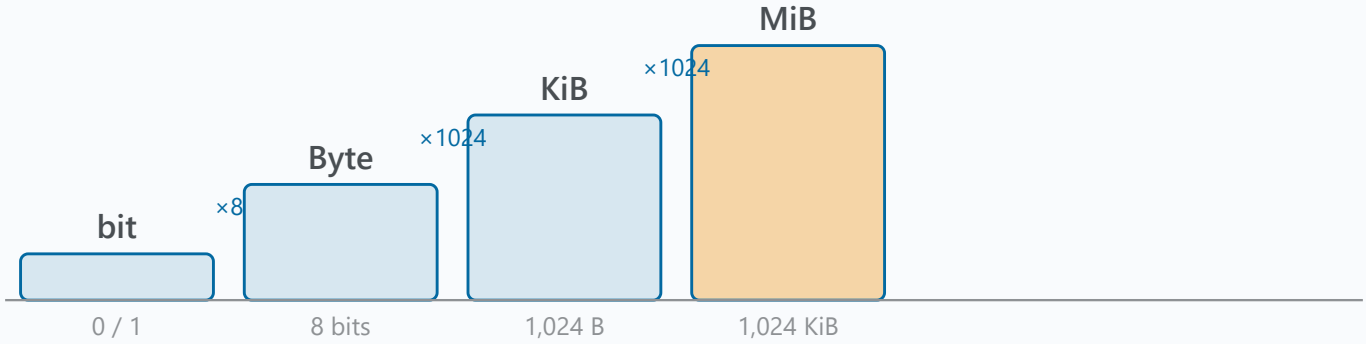
The Scale of Units



1 KiB = 1,024 B (2^{10}×; in SI, 1 KB = 1,000 B). Example: half a page of plain text ≈ 2 KB.

Bar heights are not to scale; each step is roughly a thousand times larger.

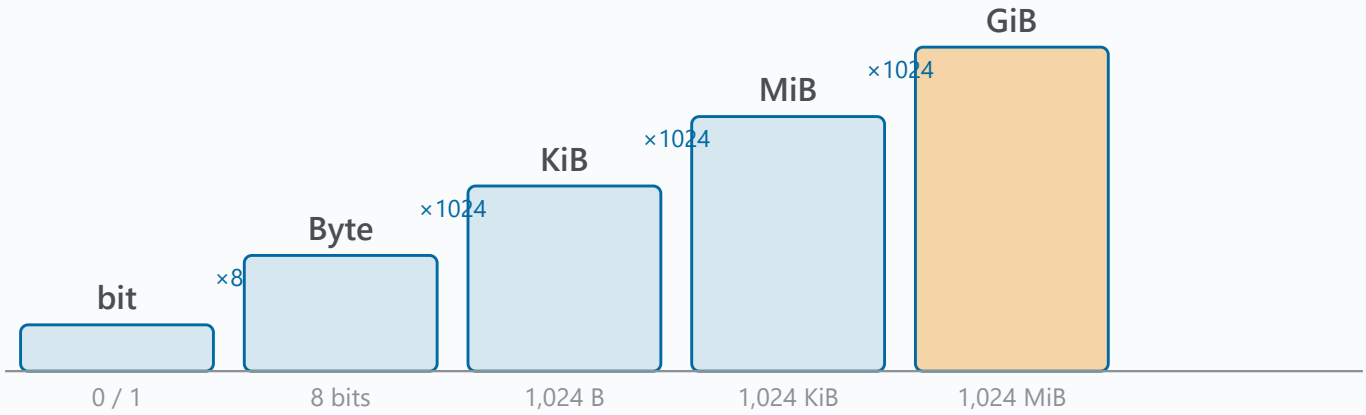
The Scale of Units



1 MiB = 1,024 KiB (2^{10}×). Example: a phone photo ≈ 3 MB.

Bar heights are not to scale; each step is roughly a thousand times larger.

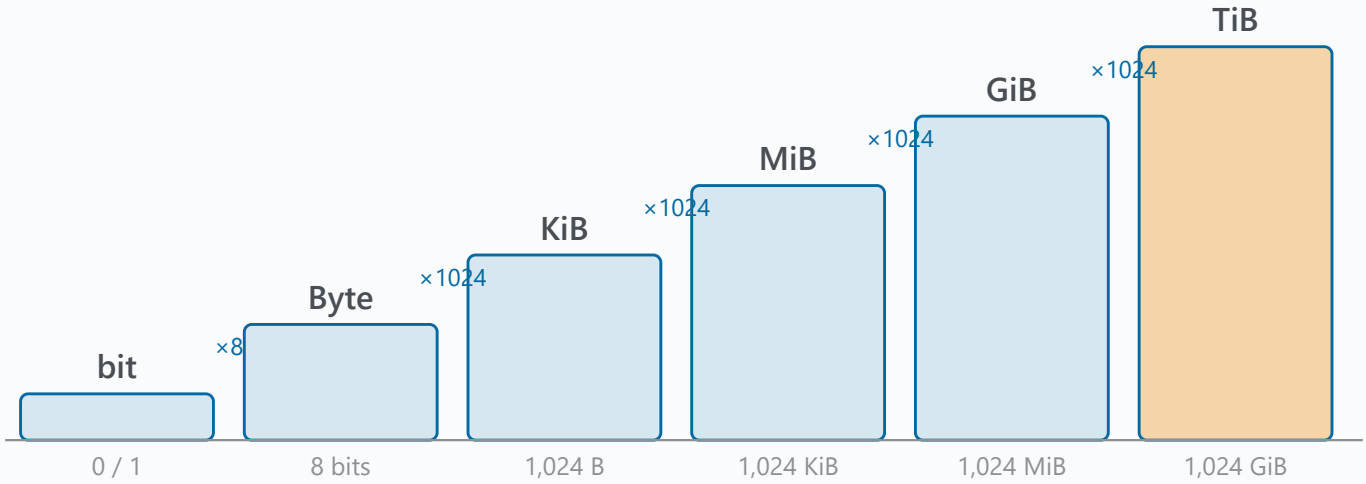
The Scale of Units



1 GiB = 1,024 MiB (2^{10}×). Example: an HD movie ≈ 4 GB.

Bar heights are not to scale; each step is roughly a thousand times larger.

The Scale of Units



1 TiB = 1,024 GiB ($2^{10}\times$). Example: a laptop drive ≈ 1 TB.

Bar heights are not to scale; each step is roughly a thousand times larger.

Bits or Bytes? Speed and Size

Lowercase b, uppercase B

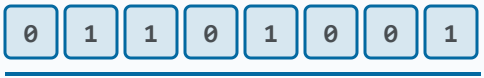
b is a bit, **B** is a byte: 100 Mb and 100 MB differ by a factor of 8.

Speed is measured in bits

Network speed is bits per second: 100 Mbps = 100,000,000 bit/s (M = 10^6).

Size is measured in bytes

File and disk sizes are in bytes: MB, GB.



8 bits (b) = 1 byte (B)

Example: a 3 GB file over a 100 Mbps link	Result
Speed in bytes: 100,000,000 ÷ 8	12,500,000 bytes/s = 12.5 MB/s
File in bits: 3 × 10^9 bytes × 8	24 × 10^9 bits
Time: 24 × 10^9 ÷ 10^8	240 s = 4 minutes (best case)

In practice it takes longer because of protocol overhead and other devices sharing the network.

PROCESSING DATA

Arithmetic and Logical Operations

Arithmetic

Calculates with numeric values.

`+ − × ÷ %`

Comparison

Tests how two values relate.

`> < ≥ ≤ == !=`

Logical

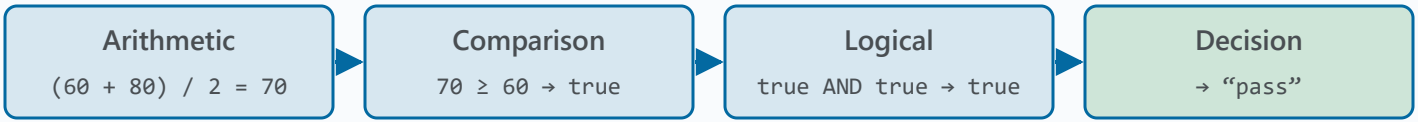
Combines or negates conditions.

`AND OR NOT`

Decision

Picks the path the program follows based on the result.

`score ≥ 60 → pass`



Example: midterm 60, final 80, attended = true

Catching Errors with a Parity Bit

Sender



number of 1s: 4 → even

Even parity rule

the number of 1s in 8 bits must always be even

odd → error

even → no error (?)

The data to send is **1011001** (“Y” in ASCII, 89). The seven data bits contain four 1s.

Step 1 / 5

```
▶ data = 1011001; count the 1s
2 even parity: parity = 0 → send 10110010
3 bit 3 flipped in transit
4 receiver counts 1s: odd means error
5 two bits flipped: even → error missed
```

VARIABLES

`data = 1011001` `ones = 4` `parity = –`

Catching Errors with a Parity Bit

Sender

1	0	1	1	0	0	1	0
bit 1	bit 2	bit 3	bit 4	bit 5	bit 6	bit 7	parity

number of 1s: 4 → even

Even parity rule

the number of 1s in 8 bits
must always be even

odd → error
even → no error (?)

Even parity: the count of 1s is already even, so the parity bit is **0**. The 8 bits sent: 10110010.

Step 2 / 5

```
1 data = 1011001; count the 1s
▶ even parity: parity = 0 → send 10110010
3 bit 3 flipped in transit
4 receiver counts 1s: odd means error
5 two bits flipped: even → error missed
```

VARIABLES

`data = 1011001` `ones = 4`
`parity = 0` `sent = 10110010`

Catching Errors with a Parity Bit

Receiver



Even parity rule

the number of 1s in 8 bits must always be even

odd → error

even → no error (?)

Noise during transmission flips bit 3 from 1 to 0. The receiver gets **10010010**.

Step 3 / 5

```
1 data = 1011001; count the 1s
2 even parity: parity = 0 → send 10110010
▶ bit 3 flipped in transit
4 receiver counts 1s: odd means error
5 two bits flipped: even → error missed
```

VARIABLES

`sent = 10110010` `received = 10010010` `flipped bits = 1`

Catching Errors with a Parity Bit

Receiver

1	0	0	1	0	0	1	0
bit 1	bit 2	bit 3	bit 4	bit 5	bit 6	bit 7	parity

number of 1s: 3 → odd

✗ Odd: error detected → resend

Even parity rule

the number of 1s in 8 bits must always be even

odd → error

even → no error (?)

The receiver counts the 1s: 3, which is **odd**. The rule is broken, so the error is detected; which bit flipped is unknown, so the data is requested again.

Step 4 / 5

```
1 data = 1011001; count the 1s
2 even parity: parity = 0 → send 10110010
3 bit 3 flipped in transit
▶ receiver counts 1s: odd means error
5 two bits flipped: even → error missed
```

VARIABLES

`received = 10010010` `ones = 3` `result = error`

Catching Errors with a Parity Bit

Receiver



number of 1s: 2 → even

✓ Even: no error seen (but the data is corrupt!)

Even parity rule

the number of 1s in 8 bits must always be even

odd → error
even → no error (?)

If two bits flip, the count of 1s stays even and the receiver accepts bad data. Parity only catches an **odd number** of bit errors.

Step 5 / 5

```
1 data = 1011001; count the 1s
2 even parity: parity = 0 → send 10110010
3 bit 3 flipped in transit
4 receiver counts 1s: odd means error
▶ two bits flipped: even → error missed
```

VARIABLES

`received = 10000010` `ones = 2`

`result = not detected`

TYPES OF COMPUTERS

By Capacity and Purpose

Supercomputer
Heavy scientific computing

Mainframe
Huge volumes of reliable transactions

Server
Services over a network

Personal computer
Individual use

Embedded system
A specific function inside a device

Thin client
Access to a central system

Mobile device
Portable use

Quantum computer
Special classes of problems

The categories have no hard boundaries. The same device can fit more than one category depending on how it is used.

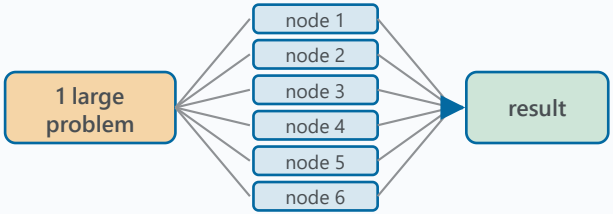
Supercomputers and Mainframes

Supercomputer

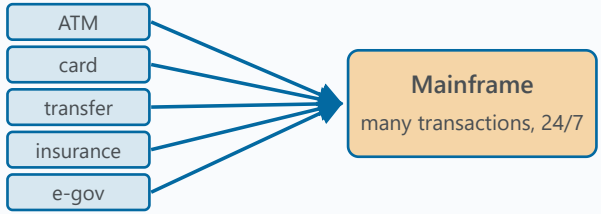
Used for scientific problems that need enormous computing power.
Examples: climate modeling, physics simulations, training AI models.

Mainframe

Built to run many concurrent transactions reliably and without interruption.
Examples: banking, insurance, large government systems.



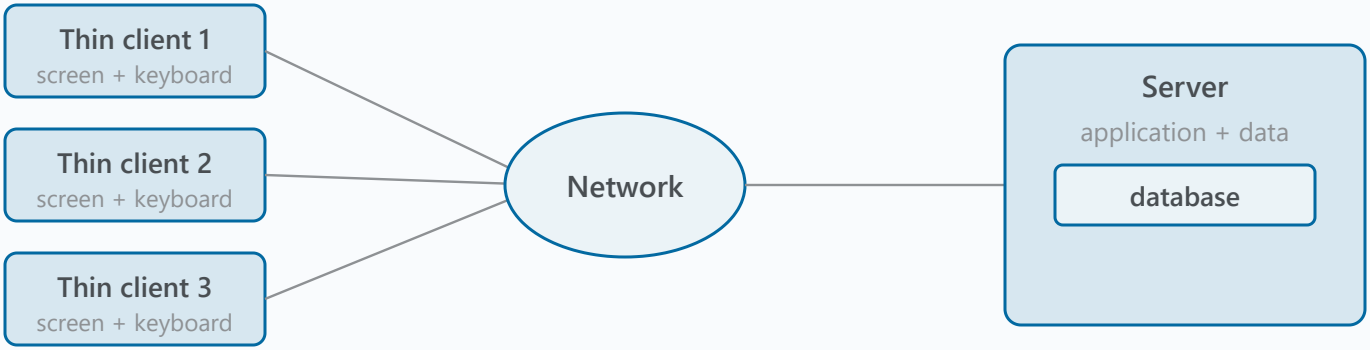
parallel computing → speed



many small transactions → availability

Priority	Supercomputer	Mainframe
Main goal	Computing speed	Transaction volume and availability
Typical workload	Large scientific computations	Many business transactions

Servers and Thin Clients



Many clients connect to the same **server** over the network.

Step 1 / 4

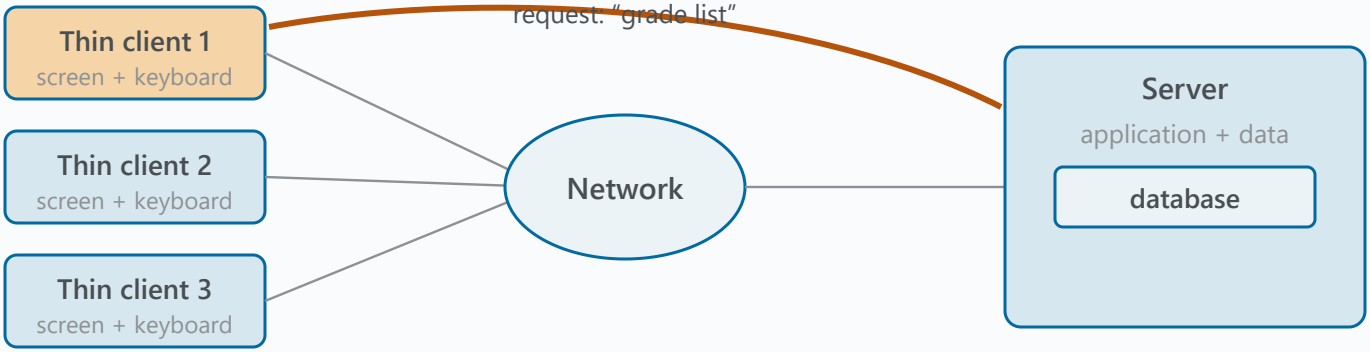
Server

Provides web, email, file, database or application services over a network.

Thin client

Leaves most of the work to the server, so it needs less local hardware and administration.

Servers and Thin Clients



Request: Client 1 asks for the grade list; the request travels over the network to the server.

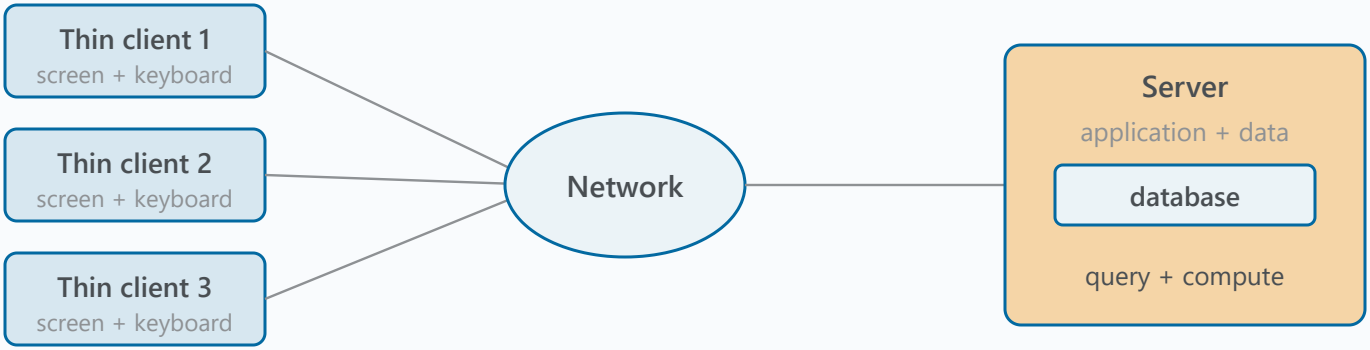
Server

Provides web, email, file, database or application services over a network.

Thin client

Leaves most of the work to the server, so it needs less local hardware and administration.

Servers and Thin Clients

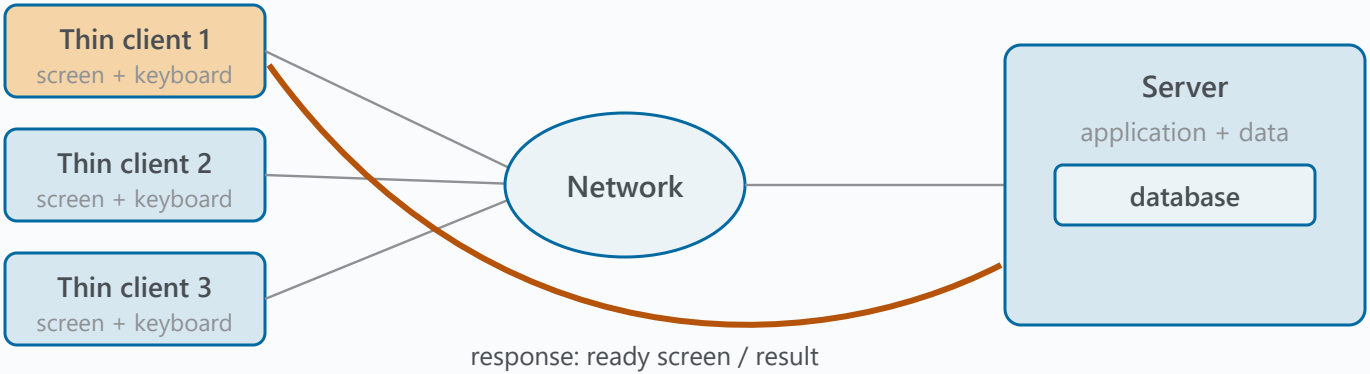


Processing: The real work happens on the server: it queries the database and computes the result.

Server
Provides web, email, file, database or application services over a network.

Thin client
Leaves most of the work to the server, so it needs less local hardware and administration.

Servers and Thin Clients



Response: The result goes back to the client. The thin client only displays it and takes input.

Server

Provides web, email, file, database or application services over a network.

Thin client

Leaves most of the work to the server, so it needs less local hardware and administration.

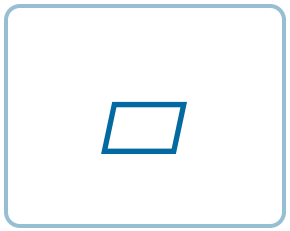
TYPES OF COMPUTERS

Personal and Mobile Devices



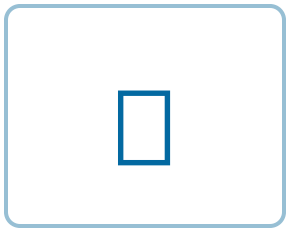
Desktop

Stationary, expandable hardware



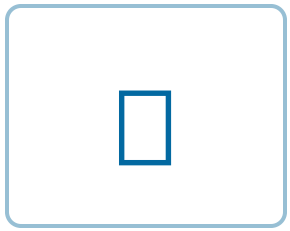
Laptop

Portable, battery-powered



Tablet

Touchscreen, light use



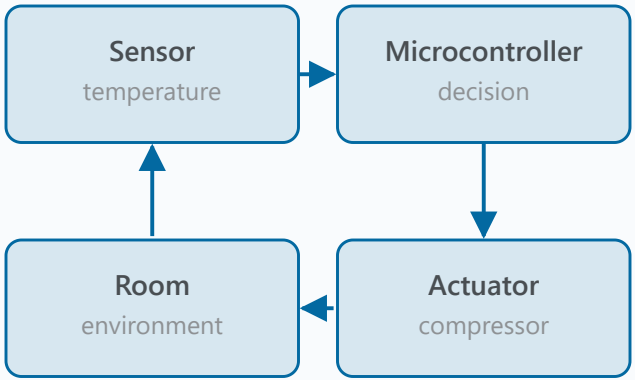
Phone

Mobile connectivity, sensors, apps

When choosing a device, portability, power use, screen size and purpose matter as much as performance.

Embedded Computers

An embedded computer performs a specific task inside a larger device.



target temperature: 24 °C

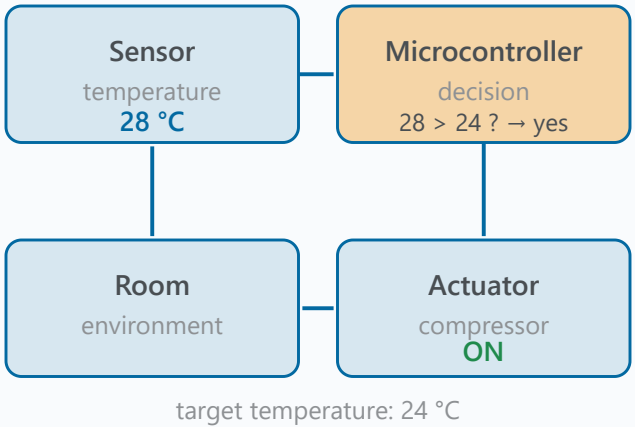
The embedded computer in an air conditioner runs a continuous **measure** → **decide** → **act** loop.

Step 1 / 4

- **Examples:** engine control units, washing machines, air conditioners, medical monitors.
- Usually low-power, running on limited resources.
- Reliability and real-time response matter.
- Users may not even realize there is a computer inside.

Embedded Computers

An embedded computer performs a specific task inside a larger device.

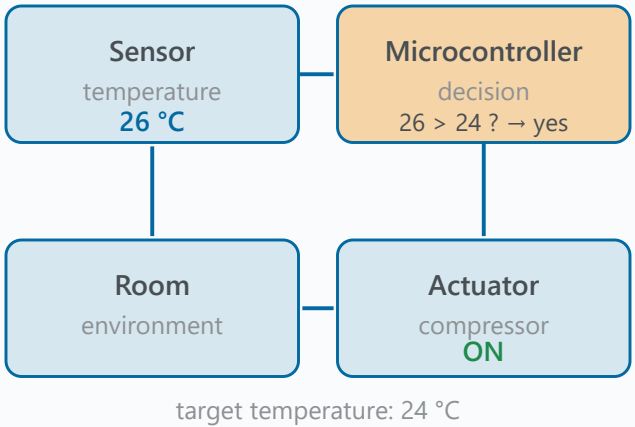


The sensor reads **28 °C**. 28 > 24 → compressor **on**; the room cools.

- **Examples:** engine control units, washing machines, air conditioners, medical monitors.
- Usually low-power, running on limited resources.
- Reliability and real-time response matter.
- Users may not even realize there is a computer inside.

Embedded Computers

An embedded computer performs a specific task inside a larger device.

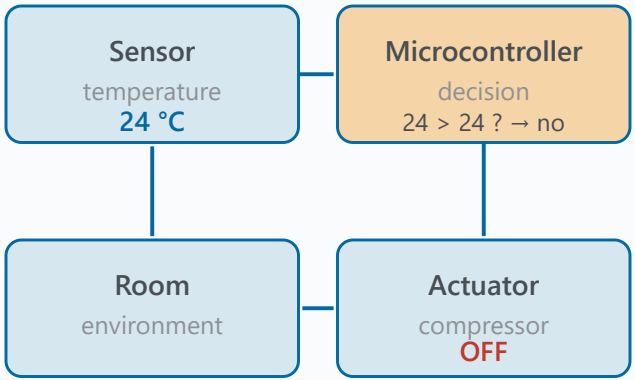


The sensor reads **26 °C**. 26 > 24 → compressor **on**; the room cools.

- **Examples:** engine control units, washing machines, air conditioners, medical monitors.
- Usually low-power, running on limited resources.
- Reliability and real-time response matter.
- Users may not even realize there is a computer inside.

Embedded Computers

An embedded computer performs a specific task inside a larger device.



target temperature: 24 °C

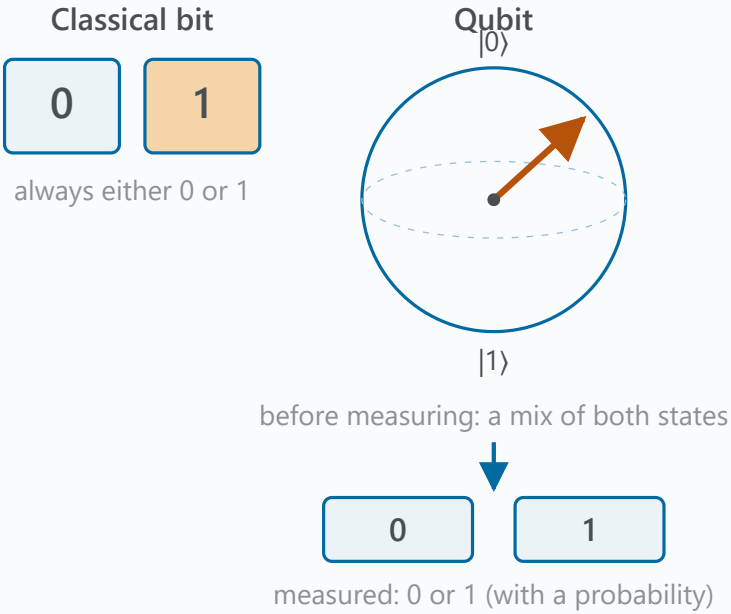
The sensor reads **24 °C**. 24 > 24 is false → compressor **turns off**. The loop keeps measuring.

- **Examples:** engine control units, washing machines, air conditioners, medical monitors.
- Usually low-power, running on limited resources.
- Reliability and real-time response matter.
- Users may not even realize there is a computer inside.

Quantum Computers

Quantum computers process information with quantum bits, called **qubits**.

- They offer different ways of computing for certain problems.
- Studied for cryptography, optimization, chemistry and materials research.
- They have not replaced classical computers for everyday general-purpose use.



Information Technology (IT)

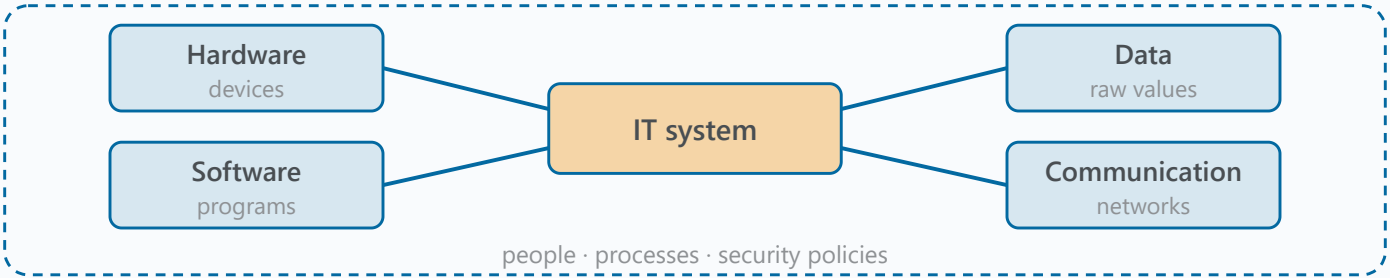
Information technology is the set of systems used to collect, process, store and transmit data.

Hardware
Physical devices

Software
Programs and operating systems

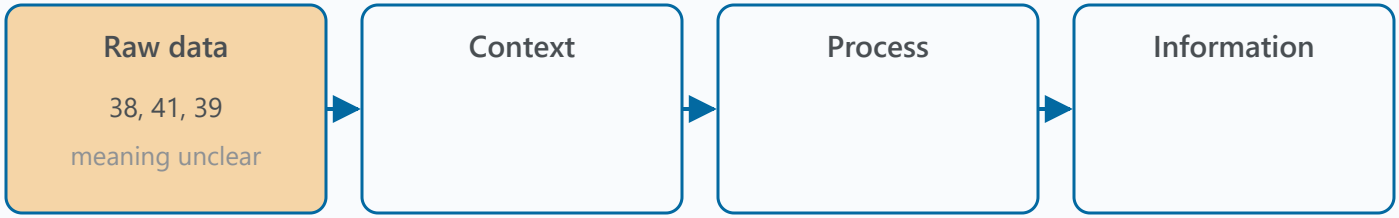
Data
Raw values being processed

Communication
Networks and data transfer



An IT system is more than computers. People, processes and security policies also shape how it works.

Data vs. Information



38, 41, 39: just numbers. Without knowing what was measured, they mean nothing.

Step 1 / 4

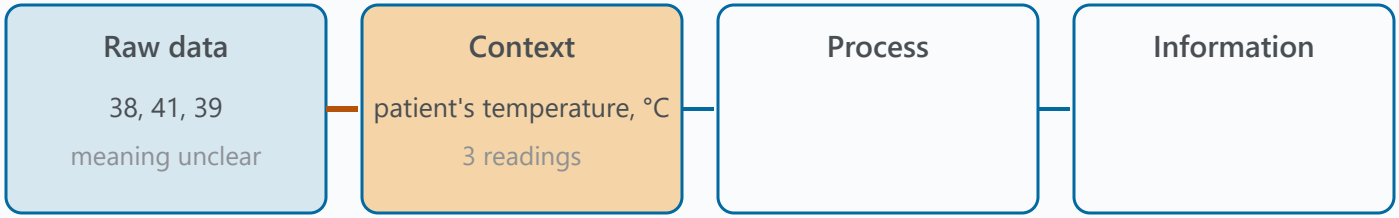
Data

Raw measurements, numbers, text or records. Without context, their meaning may be unclear.

Information

The meaningful result of organizing and interpreting data.

Data vs. Information



Add **context**: these are three temperature readings from one patient (°C).

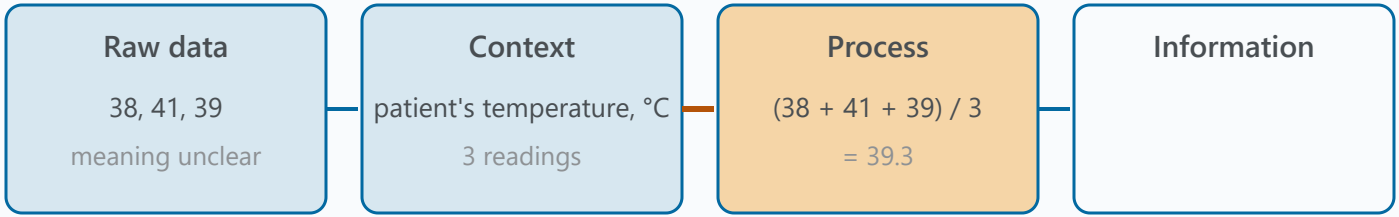
Data

Raw measurements, numbers, text or records. Without context, their meaning may be unclear.

Information

The meaningful result of organizing and interpreting data.

Data vs. Information



Process: average = (38 + 41 + 39) / 3 = **39.3**.

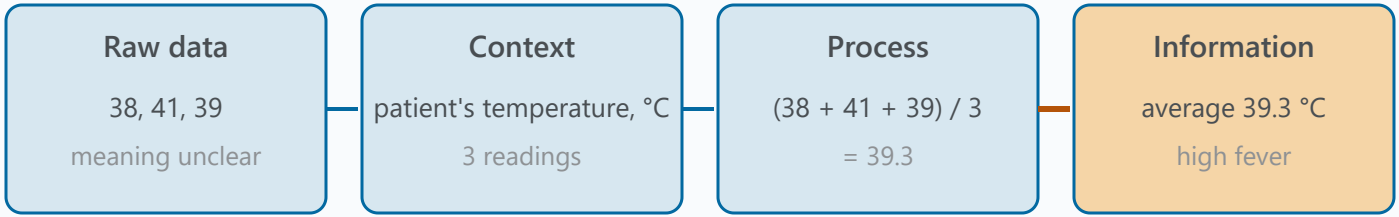
Data

Raw measurements, numbers, text or records. Without context, their meaning may be unclear.

Information

The meaningful result of organizing and interpreting data.

Data vs. Information



Information: "Average temperature 39.3 °C, high fever." Now a decision can be made.

Data

Raw measurements, numbers, text or records. Without context, their meaning may be unclear.

Information

The meaningful result of organizing and interpreting data.

IT in Everyday Life

Education

Online courses, digital resources and learning platforms.

Health

Electronic records, imaging systems and patient monitoring.

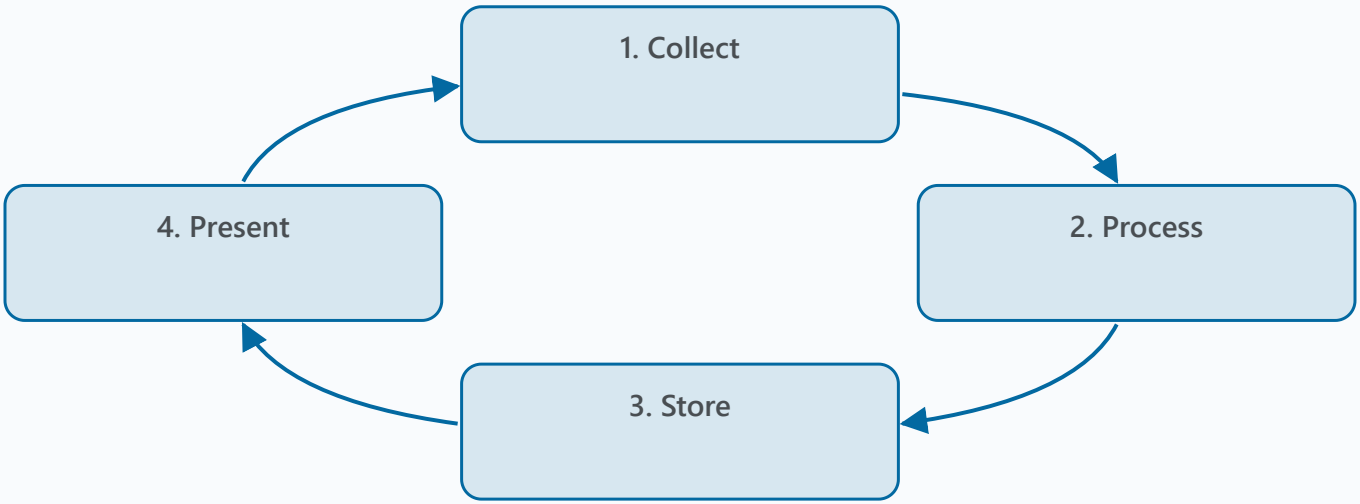
Finance

Banking, payment systems and transaction security.

Business and entertainment

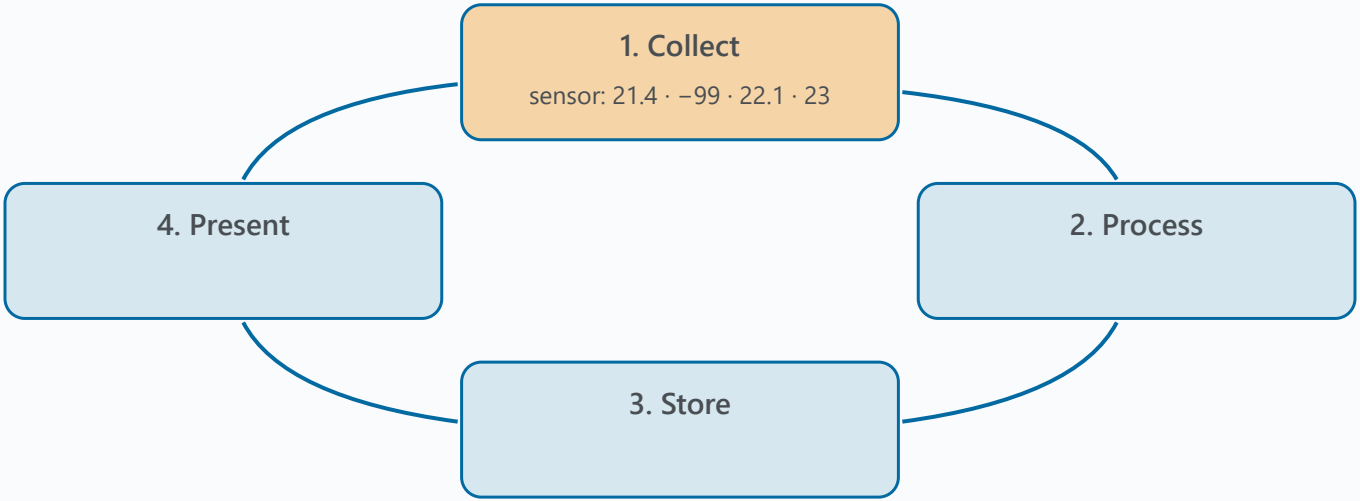
Corporate communication, automation, games and media services.

The Data Life Cycle



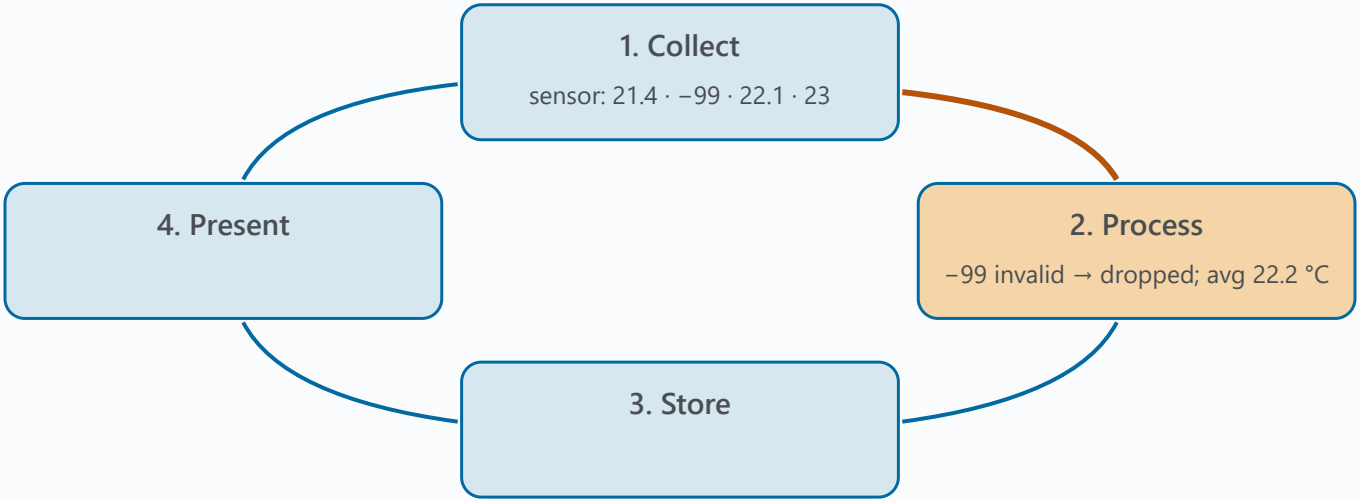
Example: data from a weather station goes through four stages, and then the cycle starts again.

The Data Life Cycle



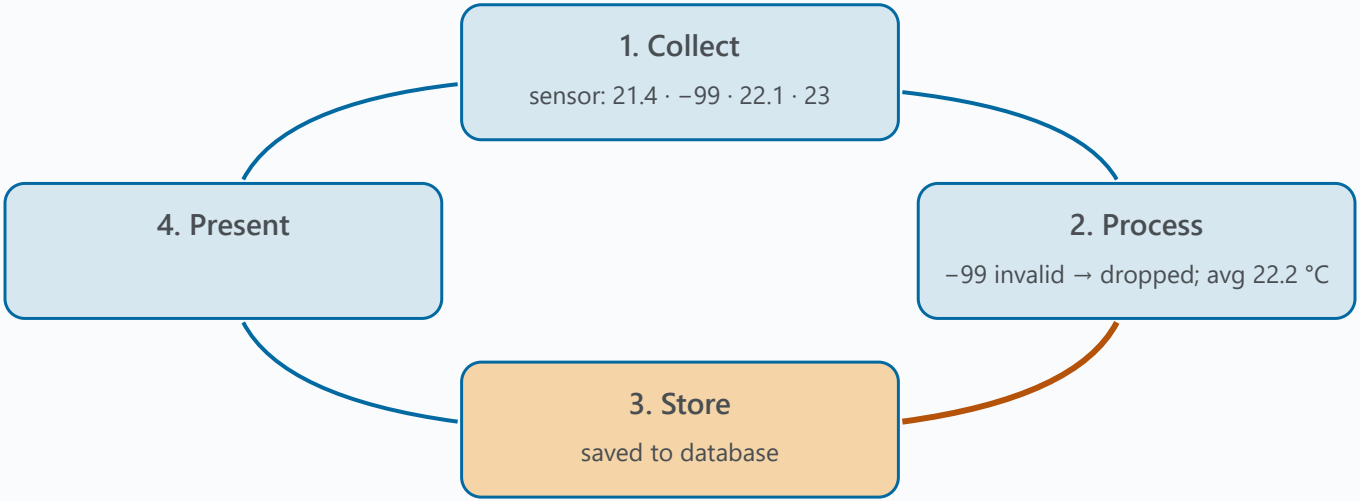
Collect: Data comes from people, machines or sensors. The sensor sent 4 readings.

The Data Life Cycle



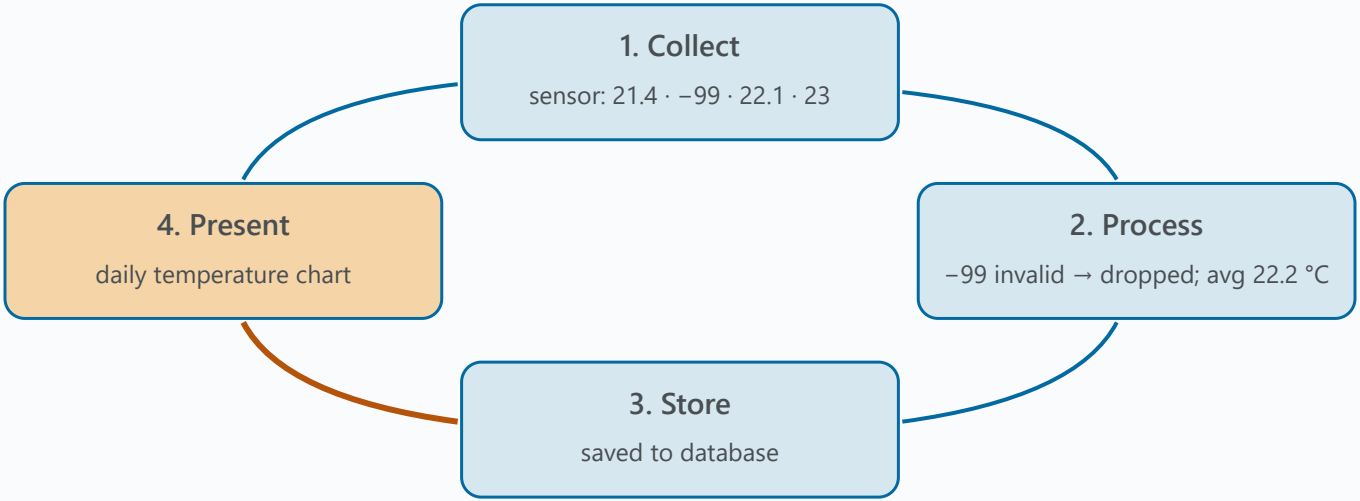
Process: Errors are removed and calculations done. −99 (a faulty reading) is dropped; the average is 22.2 °C.

The Data Life Cycle



Store: Data is kept in files, databases or the cloud.

The Data Life Cycle



Present: The result is shared as a report, table, chart or app screen; new decisions trigger new data collection.

Benefits and Responsibilities

Speed and automation

Data is processed quickly and repetitive tasks are automated.

Access and communication

Information is easier to reach and share from anywhere.

Privacy and security

Personal data must be protected from unauthorized access and loss.

Accuracy and ethics

Bad data and biased systems can produce wrong results.

Convenience does not remove the duty to use data accurately, securely and responsibly.

The Future of IT

Artificial intelligence

Systems that learn patterns from data to support decisions and generate content.

Internet of Things

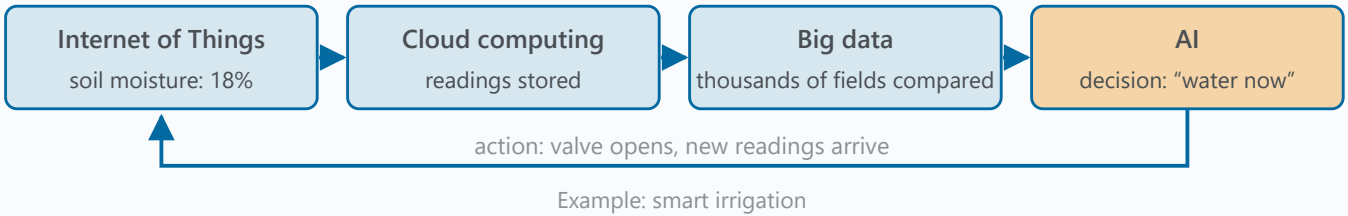
Devices with sensors sharing data over a network.

Cloud computing

Computing and storage resources delivered over a network.

Big data analytics

Extracting meaningful results from large, varied datasets.



Example: smart irrigation

Alongside new technology, energy use, accessibility, security and ethics matter more and more.

SUMMARY

Chapter Summary

- Computers process data in a cycle of input, processing, output and storage.
- All kinds of data are represented as bit strings.
- Each extra bit doubles the number of representable values; binary–decimal conversion uses place values.
- AND, OR and NOT combine conditions; addition and comparison happen in the ALU.
- Sound becomes numbers through sampling, quantization and encoding; compression is lossless (ZIP, PNG) or lossy (JPEG, MP3).
- A parity bit catches single-bit transmission errors; network speed is measured in bit/s, file size in bytes.
- Computer types differ in purpose, capacity and operating environment.
- Information technology consists of hardware, software, data, communication, people and processes.

A computer processes data. People add context and turn it into information.

SUMMARY

Review Questions

1. What are the input, processing, output and storage steps in an ATM transaction?
2. Why can 8 bits represent 256 different values?
3. What is the key difference between RAM and an SSD?
4. How does a server differ from a mainframe?
5. What context does a temperature reading need to become information?
6. Why is security in IT not just a technical issue?

Exercises

1. Convert 37 to binary (by repeated division by 2).
2. What is 101011_2 in decimal?
3. What is the minimum number of bits needed to encode only the 29 uppercase letters of the Turkish alphabet? How many codes are left unused?
4. Write the word “Hi” as ASCII codes and as 8-bit bytes (H = 72, i = 105).
5. How many bytes does an uncompressed 1920×1080 image take at 3 bytes per pixel? Roughly how many MB is that?
6. How many frames are in a 10-second video at 30 fps?

▶ **Show answers**

References

- Brookshear and Brylow, *Computer Science: An Overview*, Pearson.
- Stallings, *Computer Organization and Architecture*, Pearson.
- Unicode Consortium, *The Unicode Standard*.
- NIST, *Glossary of Key Information Security Terms*.

The End

Chapter 1.1: Basic Concepts

Next: Chapter 1.2: Computer Hardware