Chapter 1.3: Computers

Computer Fundamentals and Introduction to Programming

Data representation | Text | Stored programs | CPU and memory | Abstraction

Sercan KÜLCÜ · Giresun University

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/computer/slides/Chapter_01_03_Computers.html

INTRODUCTION

Learning Outcomes

Number representation

Explain the consequences of a fixed number of bits (negative numbers, overflow, floating point).

Text representation

Distinguish ASCII, Unicode and strings.

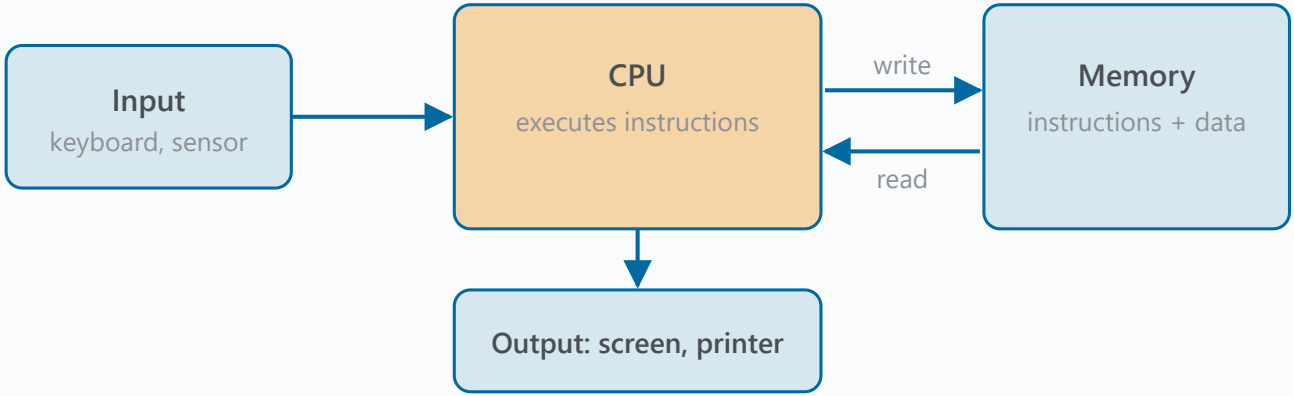
Program execution

Explain the stored-program concept and the fetch–decode–execute cycle.

Architecture

Interpret how the CPU, memory, registers and buses work together.

The Basic Parts of a Computer



CPU
Executes instructions.

Memory
Stores data and instructions.

Input / output
Exchanges data with the outside world.

From Information to Bits

Numbers, letters, images and program instructions are all stored in memory as bit strings. The data type and encoding decide what a bit string means.




```
01000001 (65) + 00100000 (32) = 01100001 (97) = “a”
```

Number Bases

System	Base	Digits	Example
Binary	2	0–1	1101_2
Octal	8	0–7	15_8
Decimal	10	0–9	13_{10}
Hexadecimal	16	0–9, A–F	D_{16}

45_{10} = **1 0 1 1 0 1** $_2$ (32 + 8 + 4 + 1)

45_{10} = **101101_2**. Octal and hexadecimal are shorthand for long binary numbers.

Number Bases

System	Base	Digits	Example
Binary	2	0–1	1101_2
Octal	8	0–7	15_8
Decimal	10	0–9	13_{10}
Hexadecimal	16	0–9, A–F	D_{16}

45_{10} = **1 0 1 1 0 1** $_2$ (32 + 8 + 4 + 1)

groups of 3: **101** **101** → 55_8 (5·8 + 5 = 45)

= 5 = 5

Since 8 = 2^3, group the bits in **threes** from the right: 101 101 → 55_8.

Number Bases

System	Base	Digits	Example
Binary	2	0–1	1101_2
Octal	8	0–7	15_8
Decimal	10	0–9	13_{10}
Hexadecimal	16	0–9, A–F	D_{16}

45_{10} = **1 0 1 1 0 1** $_2$ (32 + 8 + 4 + 1)

groups of 3: **101** **101** → 55_8 (5·8 + 5 = 45)

= 5 = 5

groups of 4: **0010** **1101** → $2D_{16}$ (2·16 + 13 = 45)

= 2 = 2 = 13 = D

Since $16 = 2^4$, group them in **fours**: 0010 1101 → 2 and 13 (D) → **$2D_{16}$**.

Negative Integers

Modern computers usually store negative integers in **two's complement**.

A single zero

There is no separate positive and negative zero.

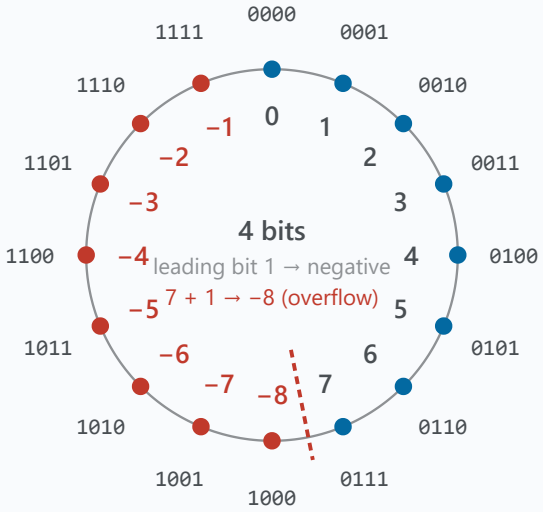
Arithmetic

The same adder circuit works for positive and negative numbers.

Range of n bits

From -2^{n-1} to $2^{n-1}-1$.

`8 bits: −128 … 127`

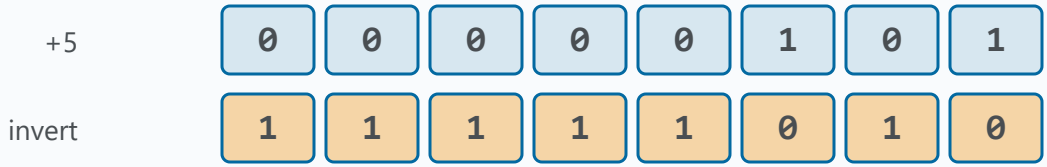


Two's Complement: How Do We Write −5?



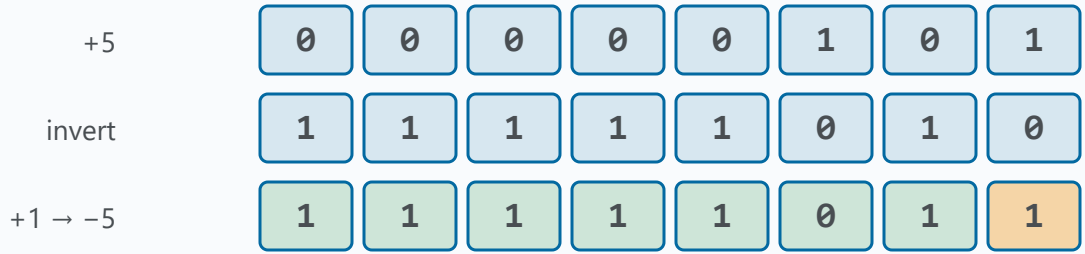
In 8 bits, **+5 = 00000101**. Two steps give −5.

Two's Complement: How Do We Write −5?



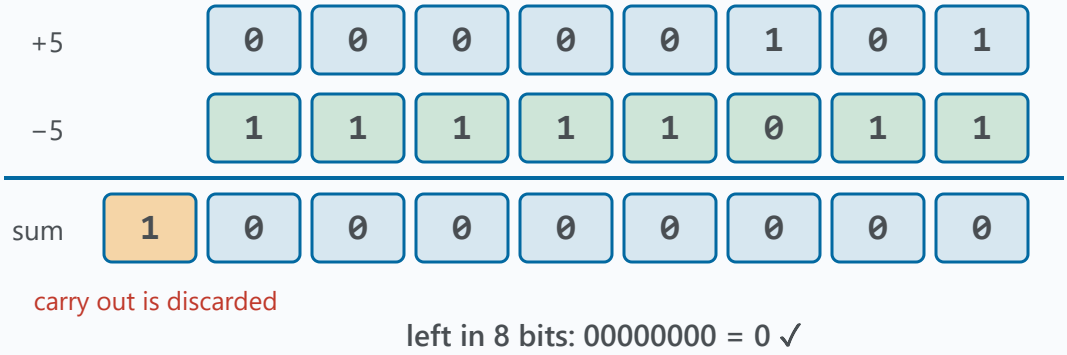
1. **Invert** every bit (0 ↔ 1): 11111010. This is the ones' complement.

Two's Complement: How Do We Write −5?



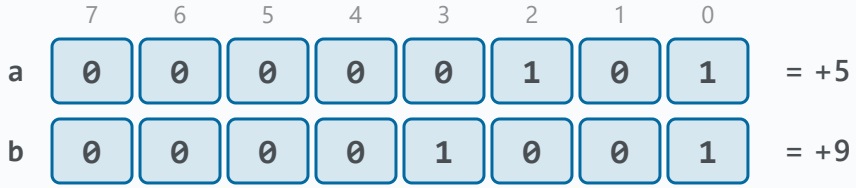
2. **Add 1**: 11111010 + 1 = **11111011** = −5. The leading 1 marks the number as negative.

Two's Complement: How Do We Write −5?



Check: 00000101 + 11111011 = 100000000. The 9th bit doesn't fit in 8 bits and is dropped → **0**. The adder needs no special case for negatives.

Subtraction Is Really Addition: 5 − 9



The processor needs no separate subtractor: **a − b = a + (−b)**. First we find the two's complement of b.

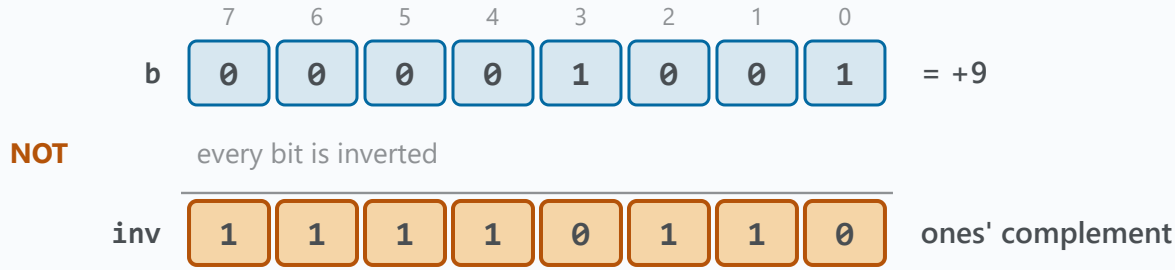
Step 1 / 5

```
▶ a ← 00000101      ; +5
▶ b ← 00001001      ; +9
3 inv ← NOT b       ; flip bits
4 neg ← inv + 1     ; −9
5 s ← a + neg       ; 5 + (−9)
6 print(s)          ; read signed
```

VARIABLES

a = 00000101 | b = 00001001 | inv = – | neg = – | s = –

Subtraction Is Really Addition: 5 − 9



Invert all the bits (0 ↔ 1): **11110110**.

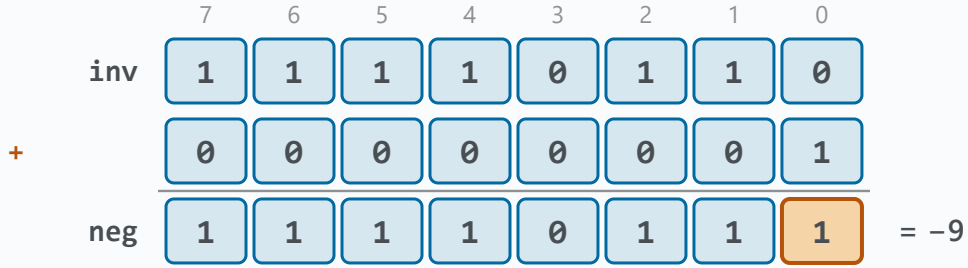
Step 2 / 5

```
1 a ← 00000101      ; +5
2 b ← 00001001      ; +9
▶ inv ← NOT b       ; flip bits
4 neg ← inv + 1     ; −9
5 s ← a + neg       ; 5 + (−9)
6 print(s)          ; read signed
```

VARIABLES

a = 00000101 | b = 00001001 | inv = 11110110 | neg = – | s = –

Subtraction Is Really Addition: 5 − 9



Add 1: **11110111** = −9. (Read as unsigned, 247 = 256 − 9.)

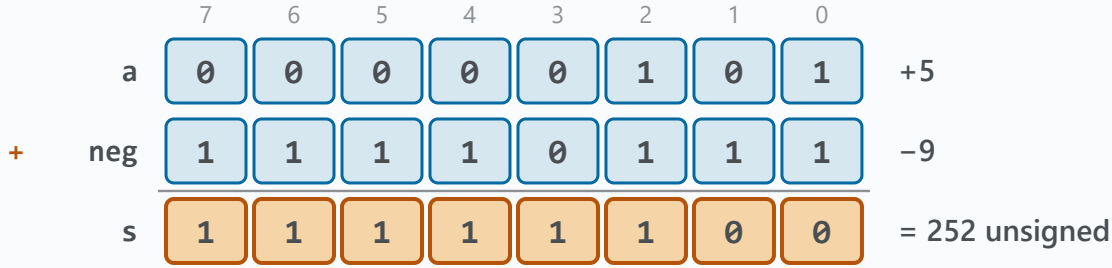
Step 3 / 5

```
1 a ← 00000101      ; +5
2 b ← 00001001      ; +9
3 inv ← NOT b       ; flip bits
▶ neg ← inv + 1     ; -9
5 s ← a + neg       ; 5 + (-9)
6 print(s)          ; read signed
```

VARIABLES

a = 00000101 | b = 00001001 | inv = 11110110 | neg = 11110111 | s = –

Subtraction Is Really Addition: 5 − 9



Plain binary addition: 5 + 247 = 252 = **11111100**. It doesn't reach 256, so there is no carry into a 9th bit.

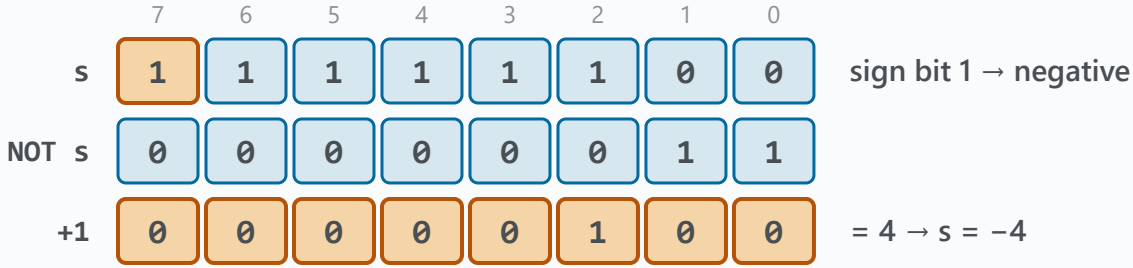
Step 4 / 5

```
1 a ← 00000101      ; +5
2 b ← 00001001      ; +9
3 inv ← NOT b       ; flip bits
4 neg ← inv + 1     ; −9
▶ s ← a + neg       ; 5 + (−9)
6 print(s)          ; read signed
```

VARIABLES

a = 00000101 b = 00001001 inv = 11110110 neg = 11110111 s = 11111100

Subtraction Is Really Addition: 5 − 9



Signed reading: the leading bit is 1, so the number is negative. For its magnitude, invert and add 1 again: 00000100 = 4 → **s = −4** ✓.

Step 5 / 5

```
1 a ← 00000101      ; +5
2 b ← 00001001      ; +9
3 inv ← NOT b       ; flip bits
4 neg ← inv + 1     ; −9
5 s ← a + neg       ; 5 + (−9)
▶ print(s)          ; read signed
```

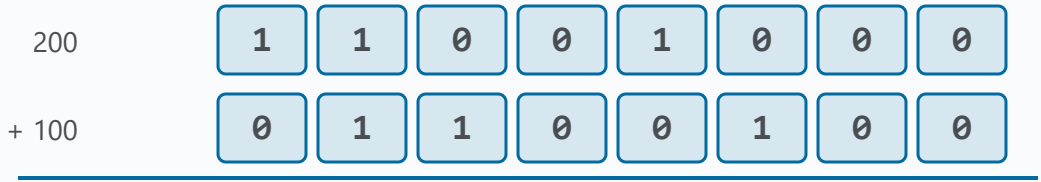
VARIABLES

`a = 00000101` `b = 00001001`

`inv = 11110110` `neg = 11110111`

`s = −4`

Overflow: 200 + 100 in 8 Bits

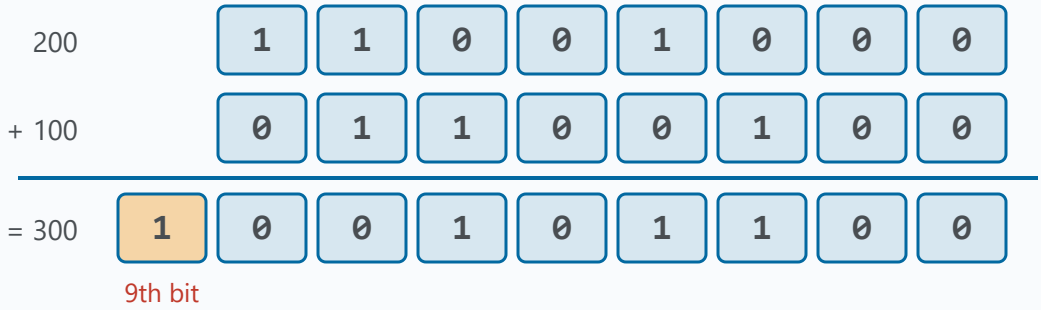


An 8-bit unsigned number is at most **255**. Let's add 200 + 100.

Step 1 / 3

In signed 8 bits, 127 + 1 also overflows to −128. That is why choosing types matters in programs.

Overflow: 200 + 100 in 8 Bits

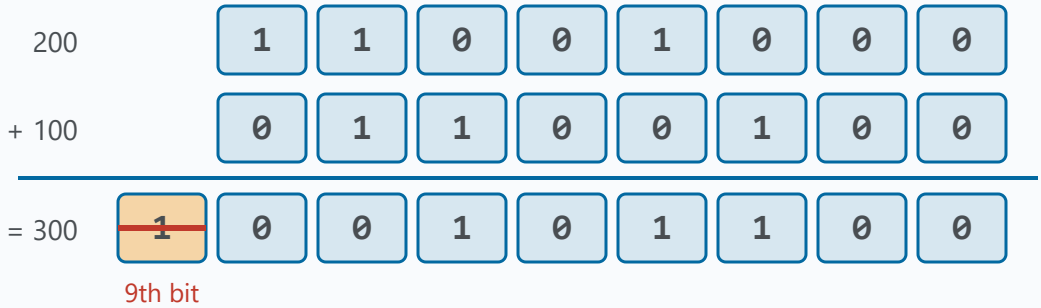


The true sum **300** = 100101100_2 needs **9 bits**.

Step 2 / 3

In signed 8 bits, 127 + 1 also overflows to −128. That is why choosing types matters in programs.

Overflow: 200 + 100 in 8 Bits



The register holds only 8 bits; the 9th bit (256) is lost → the result is **44**. This is **overflow**: 300 − 256 = 44.

Step 3 / 3

In signed 8 bits, 127 + 1 also overflows to −128. That is why choosing types matters in programs.

Signed Overflow vs. Carry

Operation (8 bits)	Binary sum	Carry	Signed overflow	Result
5 + (−9)	`00000101 + 11110111 = 11111100`	0	no	−4 ✓
(−1) + (−1)	`11111111 + 11111111 = 1 11111110`	1	no	−2 ✓
100 + 50	`01100100 + 00110010 = 10010110`	0	**yes**	−106 ✗
(−100) + (−50)	`10011100 + 11001110 = 1 01101010`	1	**yes**	+106 ✗

Carry (9th bit)

Means the unsigned result doesn't fit in 8 bits. In signed addition the carry is discarded and the result can still be correct: (−1) + (−1) = −2.

Signed overflow

Happens when two numbers of the same sign add up to a result of the opposite sign. Adding numbers of different signs never overflows.

In Java, `(byte) (100 + 50)` is −106: integer overflow raises no error, it silently gives a wrong result.

Floating-Point Numbers

Floating point

Stores very large and very small values, much like scientific notation.

Limited precision

Not every decimal value can be represented exactly with finitely many bits.

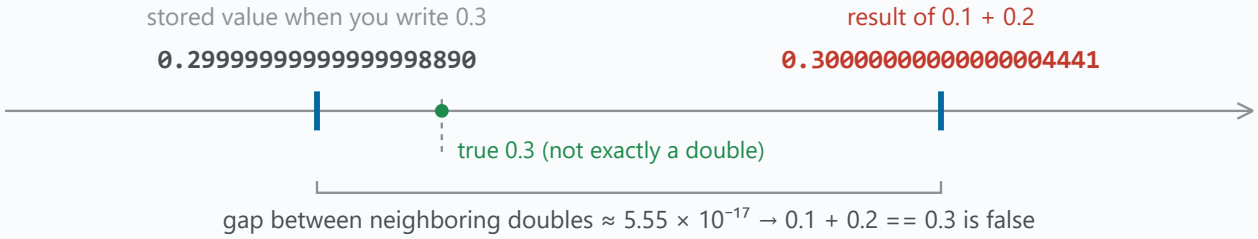
Rounding

Calculations can pick up small errors.

Comparison

Compare floating-point results within a tolerance.

```
0.1 + 0.2  →  0.30000000000000004   // computed with double
```



Why Can't 0.1 Be Stored Exactly?

Fraction × 2 repeatedly; the integer part is the next bit

$$0.1_{10} = \quad 0,$$

How do we convert a decimal fraction to binary? Multiply by 2, take the integer part (0 or 1) as a bit, and continue with the remaining fraction.

Why Can't 0.1 Be Stored Exactly?

Fraction × 2 repeatedly; the integer part is the next bit

$0.1 \times 2 = 0.2$ bit 0

$0.1_{10} =$ `0.0`

$0.1 \times 2 = 0.2 \rightarrow$ integer part **0**.

Why Can't 0.1 Be Stored Exactly?

Fraction × 2 repeatedly; the integer part is the next bit

0.1 × 2 = 0.2 → bit 0

0.2 × 2 = 0.4 → bit 0

0.1_{10} = `0.00`

0.2 × 2 = 0.4 → integer part **0**.

Why Can't 0.1 Be Stored Exactly?

Fraction × 2 repeatedly; the integer part is the next bit

0.1 × 2 = 0.2	bit 0
0.2 × 2 = 0.4	bit 0
0.4 × 2 = 0.8	bit 0

0.1_{10} = **0.000**

0.4 × 2 = 0.8 → integer part **0**.

Why Can't 0.1 Be Stored Exactly?

Fraction × 2 repeatedly; the integer part is the next bit

0.1 × 2 = 0.2	bit 0
0.2 × 2 = 0.4	bit 0
0.4 × 2 = 0.8	bit 0
0.8 × 2 = 1.6	bit 1

0.1_{10} = **`0.0001`**

0.8 × 2 = 1.6 → integer part **1**; subtract 1 and continue with the fraction.

Why Can't 0.1 Be Stored Exactly?

Fraction × 2 repeatedly; the integer part is the next bit

0.1 × 2 = 0.2	bit 0
0.2 × 2 = 0.4	bit 0
0.4 × 2 = 0.8	bit 0
0.8 × 2 = 1.6	bit 1
0.6 × 2 = 1.2	bit 1

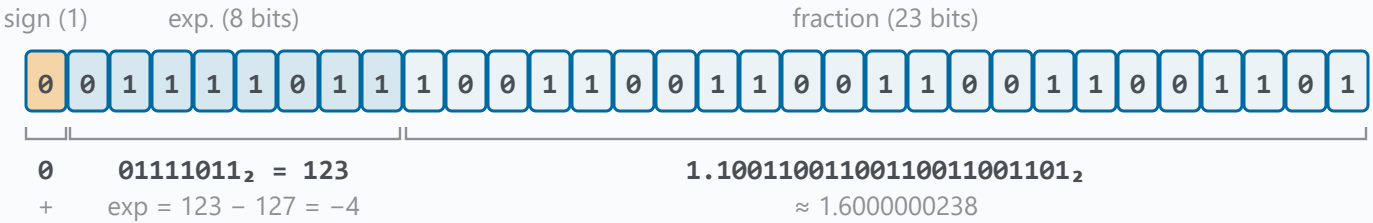
0.1_{10} = **0.00011…**

0.2 is back → “0011” repeats forever

Cut off at finitely many bits → small rounding error

0.6 × 2 = 1.2 → integer part **1**; subtract 1 and continue. The fraction is 0.2 again: the expansion **has entered a cycle** and never ends.

Floating-Point Numbers (IEEE 754)



$$\text{value} = (-1)^{\text{sign}} \times 1.\text{frac} \times 2^{(\text{exp} - 127)}$$

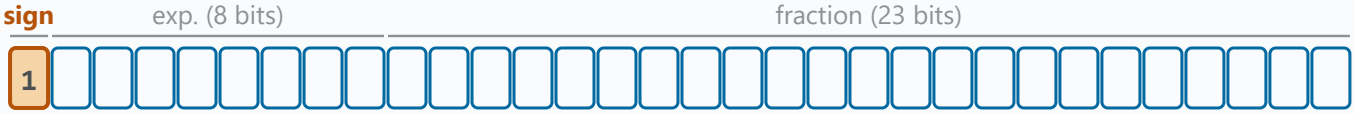
`0.1 (float) = 0x3DCCCCCD`

$\approx 1.6000000238 \times 2^{-4} \approx 0.10000000149$

- A **float** is 32 bits (1 + 8 + 23); a **double** is 64 bits (1 + 11 + 52). For normal numbers the leading 1 is not stored.
- $0.1 = 1.1001\ 1001\ 1001\ldots_2 \times 2^{-4}$: the fraction repeats forever and is rounded when it doesn't fit in 23 bits.
- The stored value is exactly 0.100000001490116119384765625: very close to 0.1, but not equal.

Encoding −6.25 as a `float`

x = −6.25 → sign bit 1



A float is 32 bits: 1 sign + 8 exponent + 23 fraction. The number is negative, so the **sign bit is 1**; the remaining steps work on 6.25.

Step 1 / 6

```
▶ x ← -6.25
▶ sign ← 1            ; negative
3 bin ← 110.01        ; 6 = 110, 0.25 = 0.01
4 norm ← 1.1001 × 2²  ; point 2 left
5 exp ← 2 + 127 = 129 ; 10000001
6 frac ← 1001 0000…   ; leading 1 dropped
7 print(hex(bits))
```

VARIABLES

`sign = 1` `bin = –` `exp = –` `frac = –`

Encoding −6.25 as a `float`

$$6 = 110_2 \cdot 0.25 \times 2 = 0.5 \to 0 \cdot 0.5 \times 2 = 1.0 \to 1$$



The integer part 6 = 110_2; the fraction 0.25 becomes **0.01_2** by “multiply by 2, take the integer part”, and reaches 0. 6.25 = **110.01_2**: unlike 0.1, finite and exact.

Step 2 / 6

```
1 x ← -6.25
2 sign ← 1              ; negative
▶ bin ← 110.01          ; 6 = 110, 0.25 = 0.01
4 norm ← 1.1001 × 2²    ; point 2 left
5 exp ← 2 + 127 = 129   ; 10000001
6 frac ← 1001 0000…     ; leading 1 dropped
7 print(hex(bits))
```

VARIABLES

`sign = 1` `bin = 110.01` `exp = –` `frac = –`

Encoding −6.25 as a `float`

$$110.01_2 = 1.1001_2 \times 2^2$$



As in scientific notation, the point moves until a single 1 is left of it: 2 places left → **1.1001×2^2**.

Step 3 / 6

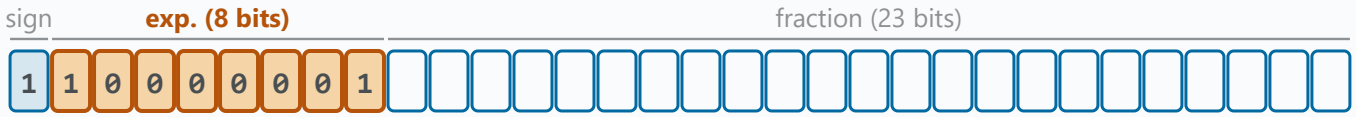
```
1 x ← −6.25
2 sign ← 1            ; negative
3 bin ← 110.01        ; 6 = 110, 0.25 = 0.01
▶ norm ← 1.1001 × 2²  ; point 2 left
5 exp ← 2 + 127 = 129 ; 10000001
6 frac ← 1001 0000…   ; leading 1 dropped
7 print(hex(bits))
```

VARIABLES

`sign = 1` `bin = 1.1001 × 2²` `exp = –` `frac = –`

Encoding −6.25 as a `float`

exp field = 2 + 127 = 129 = 10000001_2



The exponent field stores the true exponent **plus 127** (the bias): 2 + 127 = 129. This lets negative exponents be stored unsigned.

Step 4 / 6

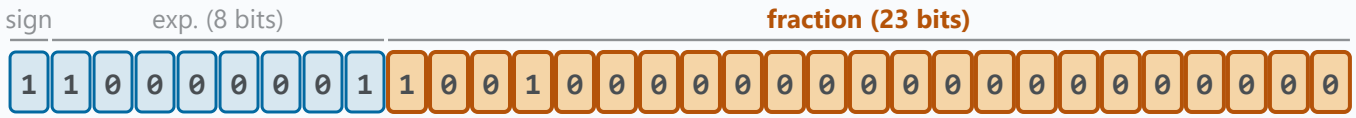
```
1 x ← -6.25
2 sign ← 1            ; negative
3 bin ← 110.01        ; 6 = 110, 0.25 = 0.01
4 norm ← 1.1001 × 2²  ; point 2 left
▶ exp ← 2 + 127 = 129 ; 10000001
6 frac ← 1001 0000…   ; leading 1 dropped
7 print(hex(bits))
```

VARIABLES

`sign = 1` `bin = 1.1001 × 2²` `exp = 10000001` `frac = –`

Encoding −6.25 as a `float`

fraction = bits after the point of 1.1001 → 1001 and 19 zeros



For normal numbers the leading 1 is always there, so it is not stored; only **1001** after the point is written and the remaining 19 bits are 0.

Step 5 / 6

```
1 x ← -6.25
2 sign ← 1            ; negative
3 bin ← 110.01        ; 6 = 110, 0.25 = 0.01
4 norm ← 1.1001 × 2²  ; point 2 left
5 exp ← 2 + 127 = 129 ; 10000001
▶ frac ← 1001 0000…   ; leading 1 dropped
7 print(hex(bits))
```

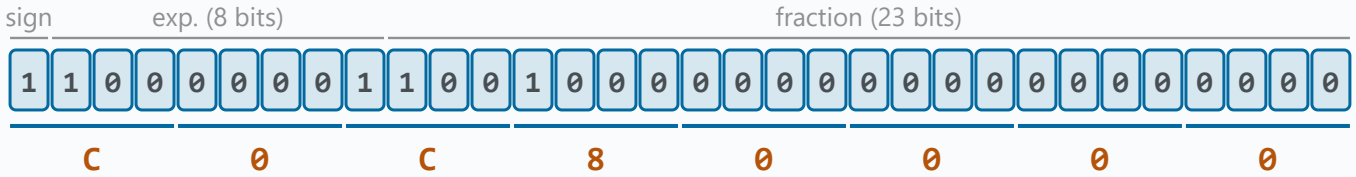
VARIABLES

`sign = 1` `bin = 1.1001 × 2²`

`exp = 10000001` `frac = 1001000…0`

Encoding −6.25 as a `float`

$$-6.25 = \mathrm{C0C80000}_{16}$$



The 32 bits in groups of four: 1100 0000 1100 1000 0000 … → **C0C80000**. In Java, `Float.floatToIntBits(-6.25f)` returns this value.

Step 6 / 6

```
1 x ← -6.25
2 sign ← 1            ; negative
3 bin ← 110.01        ; 6 = 110, 0.25 = 0.01
4 norm ← 1.1001 × 2²  ; point 2 left
5 exp ← 2 + 127 = 129 ; 10000001
6 frac ← 1001 0000…   ; leading 1 dropped
▶ print(hex(bits))
```

VARIABLES

`sign = 1` `bin = 1.1001 × 2²`

`exp = 10000001` `frac = 1001000…0`

ASCII

ASCII maps basic Latin letters, digits, punctuation and control characters to numeric codes.

ASCII is a **7-bit** code: it defines 128 characters (0–127); stored in a byte, the top bit is 0.

A
65
`01000001`

a
97
`01100001`

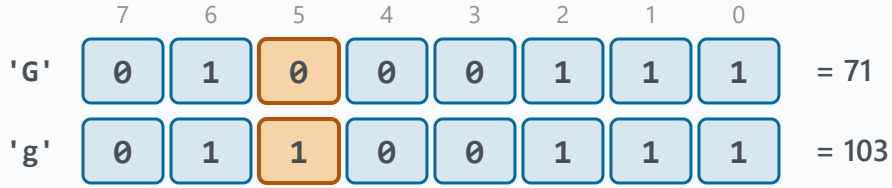
0
48
`00110000`

Line feed (LF)
Control character:
10
`00001010`



Upper and lower case differ in a single bit:
97 − 65 = 32

Bitwise Operations: Changing a Letter's Case



"G" = 71, "g" = 103: the difference is 32 = 2^5. Upper and lower case differ only in **bit 5** (counting from 0 on the right).

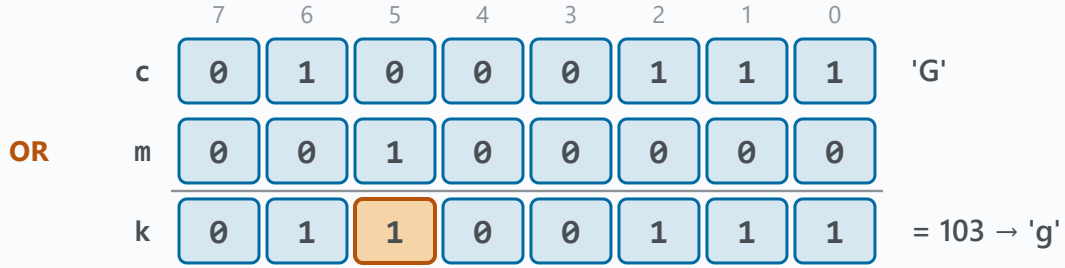
Step 1 / 5

```
▶ c ← 'G'            ; 71 = 01000111
▶ m ← 00100000       ; mask: bit 5 (32)
3 k ← c OR m         ; set the bit
4 b ← k AND (NOT m)  ; clear the bit
5 t ← b XOR m        ; flip the bit
6 s ← c << 1         ; shift left
```

VARIABLES

c = 01000111 | m = 00100000 | k = – | b = – | t = – | s = –

Bitwise Operations: Changing a Letter's Case



OR: a bit that is 1 in the mask becomes 1; the other bits are unchanged. Uppercase became lowercase.

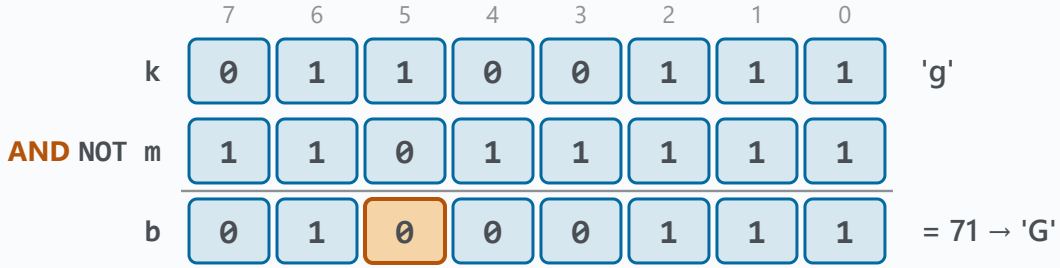
Step 2 / 5

```
1 c ← 'G'              ; 71 = 01000111
2 m ← 00100000         ; mask: bit 5 (32)
▶ k ← c OR m           ; set the bit
4 b ← k AND (NOT m)    ; clear the bit
5 t ← b XOR m          ; flip the bit
6 s ← c << 1           ; shift left
```

VARIABLES

c = 01000111 | m = 00100000 | k = 01100111 | b = – | t = – | s = –

Bitwise Operations: Changing a Letter's Case



AND with the inverted mask: the masked bit is cleared to 0, the rest are kept. Lowercase became uppercase again.

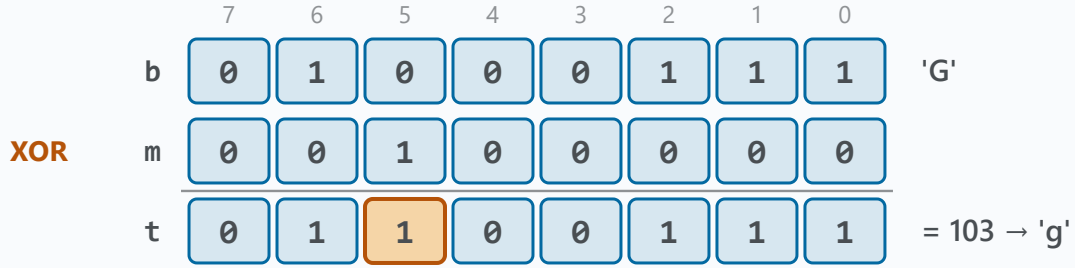
Step 3 / 5

```
1 c ← 'G'              ; 71 = 01000111
2 m ← 00100000         ; mask: bit 5 (32)
3 k ← c OR m           ; set the bit
▶ b ← k AND (NOT m)    ; clear the bit
5 t ← b XOR m          ; flip the bit
6 s ← c << 1           ; shift left
```

VARIABLES

c = 01000111 m = 00100000 k = 01100111 b = 01000111 t = – s = –

Bitwise Operations: Changing a Letter's Case



XOR: the masked bit is flipped. XOR with the same mask twice gives back the original value.

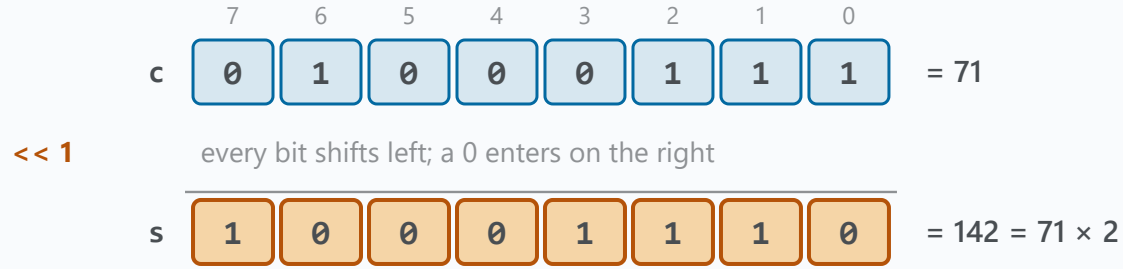
Step 4 / 5

```
1 c ← 'G'              ; 71 = 01000111
2 m ← 00100000         ; mask: bit 5 (32)
3 k ← c OR m           ; set the bit
4 b ← k AND (NOT m)    ; clear the bit
▶ t ← b XOR m          ; flip the bit
6 s ← c << 1           ; shift left
```

VARIABLES

c = 01000111	m = 00100000
k = 01100111	b = 01000111
t = 01100111	s = –

Bitwise Operations: Changing a Letter's Case



Left shift: every bit moves up one place and a 0 enters on the right → the value is multiplied by 2. Right shift (`>> 1`) halves positive numbers (rounding down): 71 → 35. Negative odd numbers differ: −3 >> 1 = −2, but −3 / 2 = −1.

Step 5 / 5

```
1 c ← 'G'              ; 71 = 01000111
2 m ← 00100000         ; mask: bit 5 (32)
3 k ← c OR m           ; set the bit
4 b ← k AND (NOT m)    ; clear the bit
5 t ← b XOR m          ; flip the bit
▶ s ← c << 1           ; shift left
```

VARIABLES

c = 01000111	m = 00100000
k = 01100111	b = 01000111
t = 01100111	s = 10001110

TEXT REPRESENTATION

Unicode

Unicode assigns a unique code point to every character in every writing system.

Code point

The character's universal number.

`A → U+0041`

UTF-8

Encodes Unicode characters as variable-length byte sequences.

Wide coverage

Turkish letters, other alphabets, math symbols and emoji.

File encoding

The same code point can be stored in different Unicode transformation formats.

character	code point	UTF-8 bytes (hex)	length
A	U+0041	41	1 byte
ğ	U+011F	C4 9F	2 bytes
€	U+20AC	E2 82 AC	3 bytes
😀	U+1F600	F0 9F 98 80	4 bytes

UTF-8 Encoding: “ğ”

“ğ” → code point U+011F = 287

Unicode assigns the letter “ğ” the code point **U+011F** (decimal 287). How does UTF-8 write this number as bytes?

UTF-8 Encoding: “ǧ”

“ǧ” → code point U+011F = 287

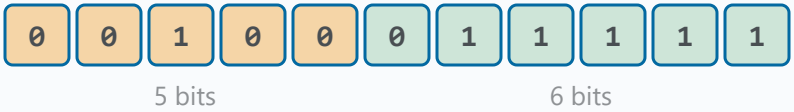
11 bits: 

287 in binary is 00100011111. It doesn't fit in 7 bits (not ASCII) but fits in 11 → **2 bytes** are needed.

UTF-8 Encoding: “ǧ”

“ǧ” → code point U+011F = 287

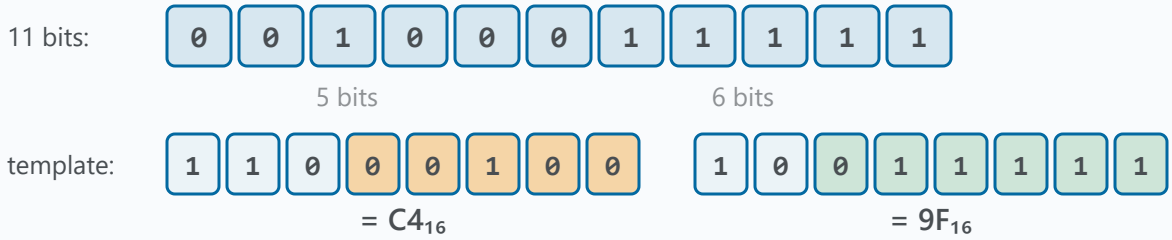
11 bits:



The 2-byte UTF-8 template is **110xxxxx 10xxxxxx**: the 11 bits are split as 5 + 6.

UTF-8 Encoding: “ǧ”

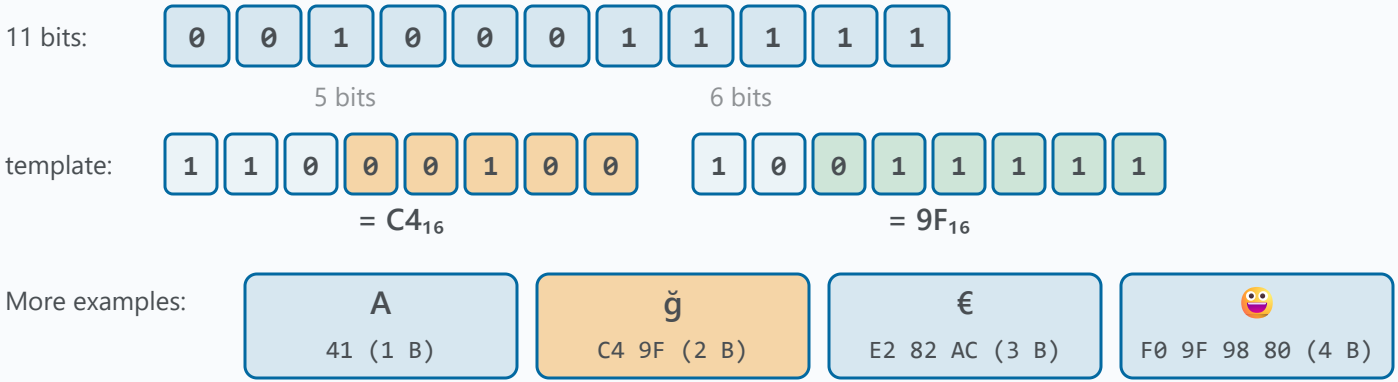
“ǧ” → code point U+011F = 287



The bits drop into the template: **11000100 10011111** = **C4 9F**. The leading 110 and 10 tell each byte's role.

UTF-8 Encoding: “ğ”

“ğ” → code point U+011F = 287

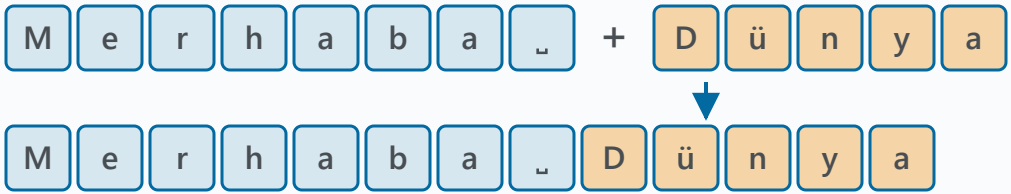


ASCII characters take 1 byte; Turkish letters 2, € 3, and emoji 4 bytes. That is why UTF-8 is **variable-length**.

TEXT REPRESENTATION

Strings

A string is a data type made of a sequence of characters.



length: 13 characters · 14 bytes in UTF-8 (“ü” is 2 bytes)

Length
The number of characters or code units.

Concatenation
Joining two strings end to end.

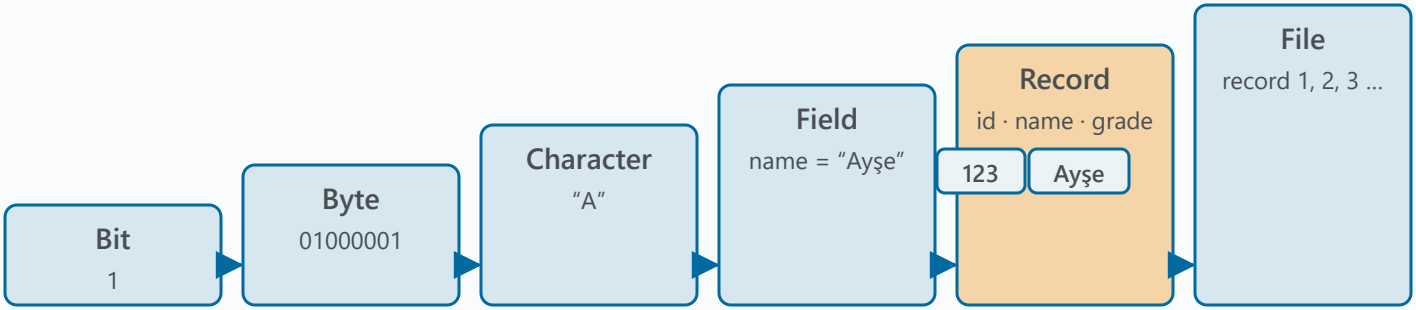
Search
Finding a character or substring.

Comparison
Checking contents and ordering.

The number of characters on screen and the number of bytes (code units) in memory are not always the same.

Structured Information

Bits gain meaning when organized into larger structures.



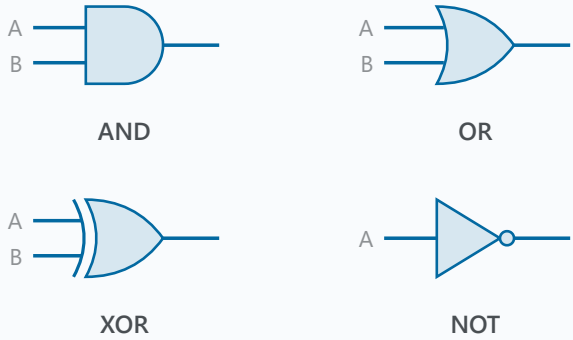
- A student record might consist of id, name and grade fields.
- A file format defines how the fields are ordered and interpreted.

Common Mistakes in Data Representation

Common mistake	Correct
Reading without the base: “1101 is one thousand one hundred one”	1101_2 = 13. Give the base as a subscript or with a `0b` / `0x` prefix.
Taking 10_{16} for ten	10_{16} = 16; in hexadecimal, ten is written A.
In two's complement, only inverting the bits	11111010 is the ones' complement (not −5); adding 1 gives 11111011 = −5.
Comparing floating-point results with ==	0.1 + 0.2 ≠ 0.3; check whether |a − b| is below a small tolerance.
Confusing a digit character with a number	The character '7' has code 55; its numeric value is '7' − '0' = 7.
Taking character count for byte count	In UTF-8, “çağ” is 3 characters but 5 bytes (ç 2 + a 1 + ğ 2).
Confusing carry with signed overflow	Carry is a problem for the unsigned reading, overflow for the signed one; either can happen without the other.

Logic Gates

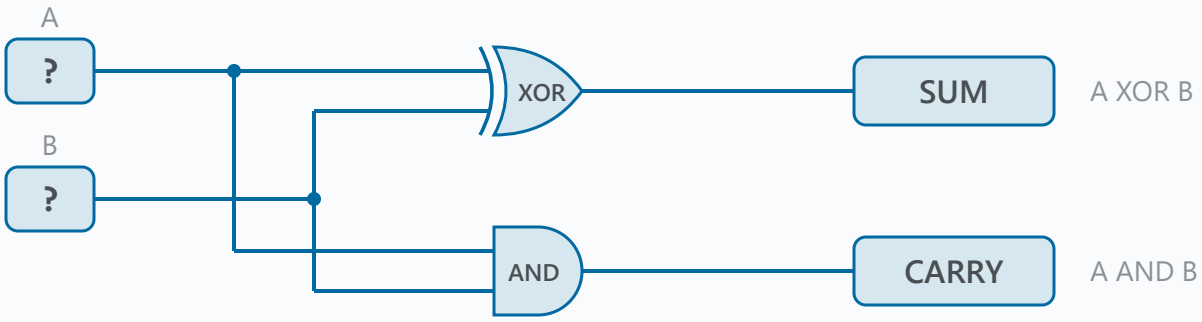
A	B	AND	OR	XOR	NOT A
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0



- **AND** gives 1 if both inputs are 1, **OR** if at least one is 1; **NOT** inverts its input.
- **XOR** (exclusive or): 1 if the inputs differ. In an adder it produces the sum digit.

Gates are built from a few transistors; every operation in a processor is a combination of these basic gates.

The Half Adder: From Gates to Addition



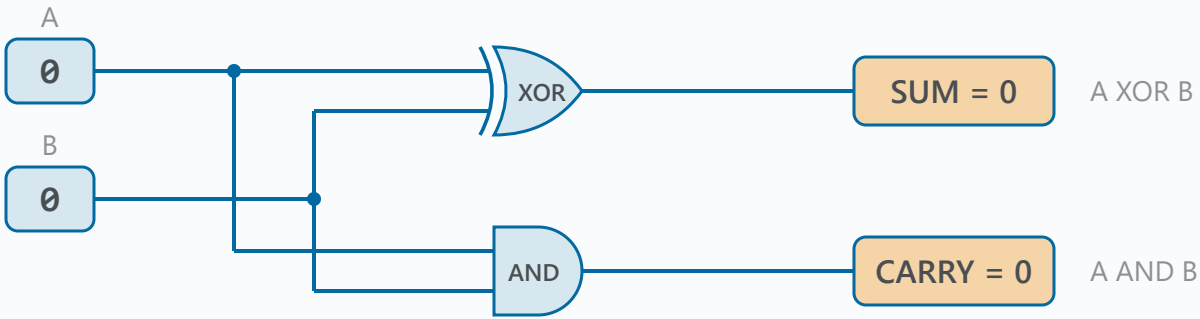
The result in two bits: CARRY SUM

A half adder adds two one-bit numbers (A, B). **SUM = A XOR B** gives the result digit and **CARRY = A AND B** the carry. Two gates are enough.

Step 1 / 4

A half adder does not take an incoming carry. Multi-bit addition needs a three-input **full adder** (two half adders and an OR gate).

The Half Adder: From Gates to Addition



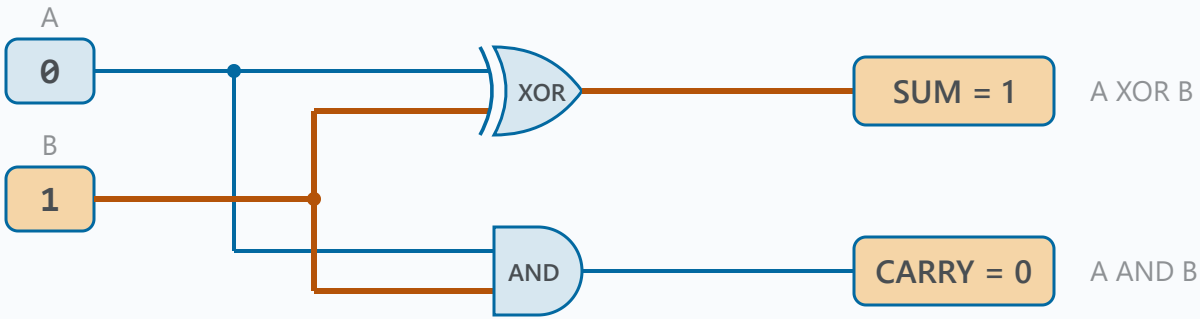
$0 + 0 = 00_2 = 0$

written in the order (CARRY, SUM)

0 + 0: XOR(0, 0) = 0, AND(0, 0) = 0 → **00_2 = 0**.

A half adder does not take an incoming carry. Multi-bit addition needs a three-input **full adder** (two half adders and an OR gate).

The Half Adder: From Gates to Addition



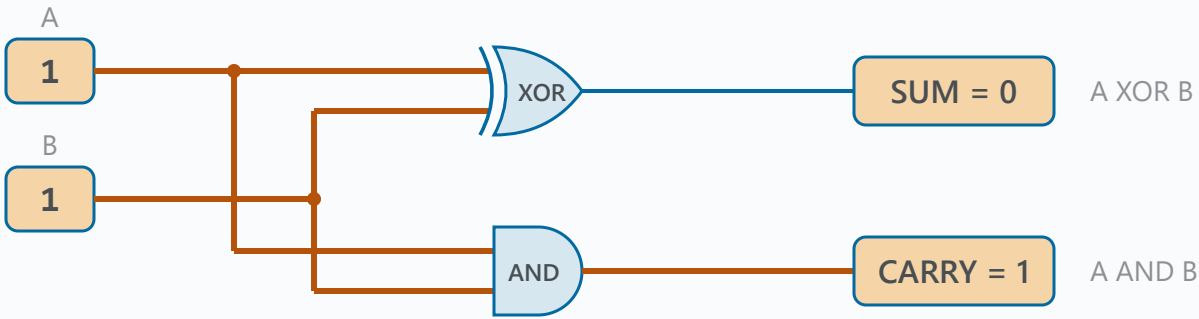
$0 + 1 = 01_2 = 1$

written in the order (CARRY, SUM)

0 + 1: XOR(0, 1) = 1, AND(0, 1) = 0 → **$01_2 = 1$**. No carry.

A half adder does not take an incoming carry. Multi-bit addition needs a three-input **full adder** (two half adders and an OR gate).

The Half Adder: From Gates to Addition



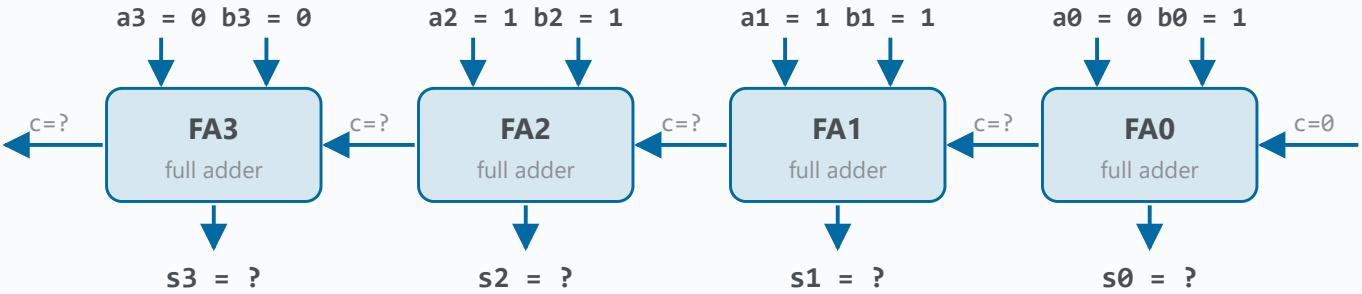
$1 + 1 = 10_2 = 2$

written in the order (CARRY, SUM)

1 + 1: XOR(1, 1) = 0 (digit 0), AND(1, 1) = 1 (carry 1) → **10_2 = 2**. As with 9 + 1 = 10 in decimal, the carry moves to the next place.

A half adder does not take an incoming carry. Multi-bit addition needs a three-input **full adder** (two half adders and an OR gate).

A Chain of Full Adders: 0110 + 0111



Four full adders (FA) are chained: each one's carry out is the carry in of the one to its left. We compute 6 + 7.

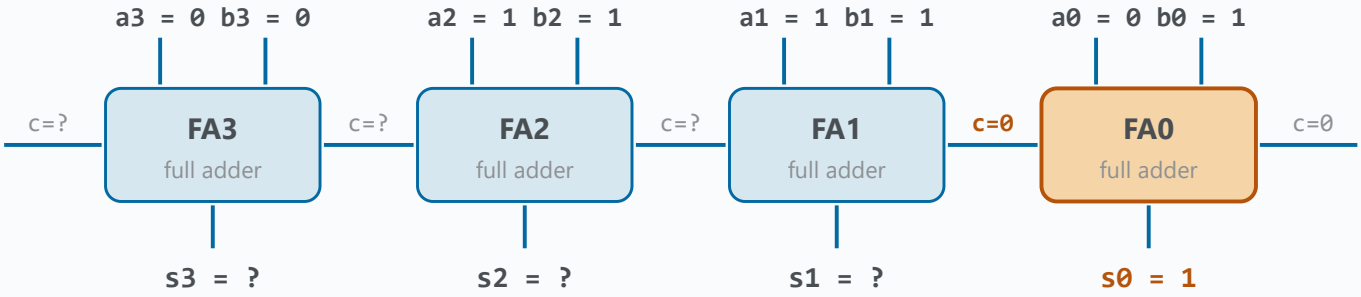
Step 1 / 6

```
a ← 0110   ; 6
b ← 0111   ; 7
c ← 0      ; initial carry
for i = 0, 1, 2, 3: ; right to left
  t ← a[i] XOR b[i]
  s[i] ← t XOR c
  c ← (a[i] AND b[i]) OR (t AND c)
print(c, s)
```

VARIABLES

`i = –` `t = –` `c = 0` `s = ????`

A Chain of Full Adders: 0110 + 0111



i = 0: a0 = 0, b0 = 1, c = 0 → t = 1, **s0 = 1**; the inputs are not both 1, so the carry is **0**.

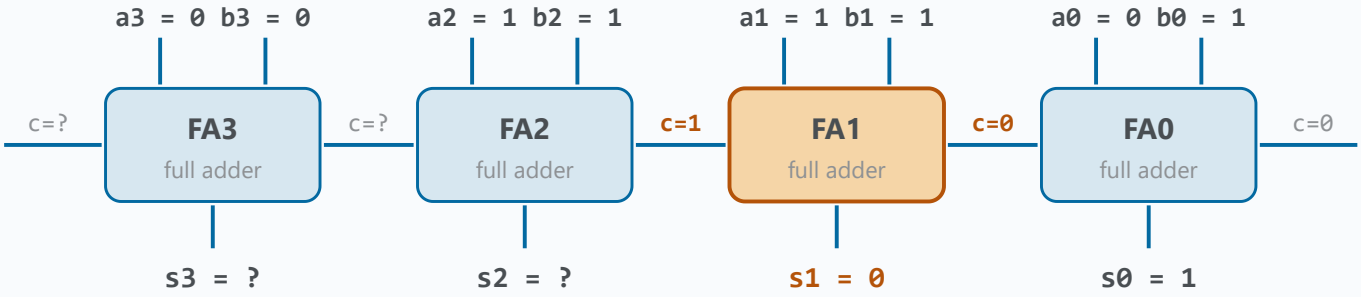
Step 2 / 6

```
1 a ← 0110  ; 6
2 b ← 0111  ; 7
3 c ← 0     ; initial carry
▶ for i = 0, 1, 2, 3: ; right to left
▶   t ← a[i] XOR b[i]
▶   s[i] ← t XOR c
▶   c ← (a[i] AND b[i]) OR (t AND c)
8 print(c, s)
```

VARIABLES

`i = 0` `t = 1` `c = 0` `s = ???1`

A Chain of Full Adders: 0110 + 0111



i = 1: 1 + 1 + 0 → t = 0, **s1 = 0**; a1 AND b1 = 1, so the carry is **1** (like 5 + 5 in decimal).

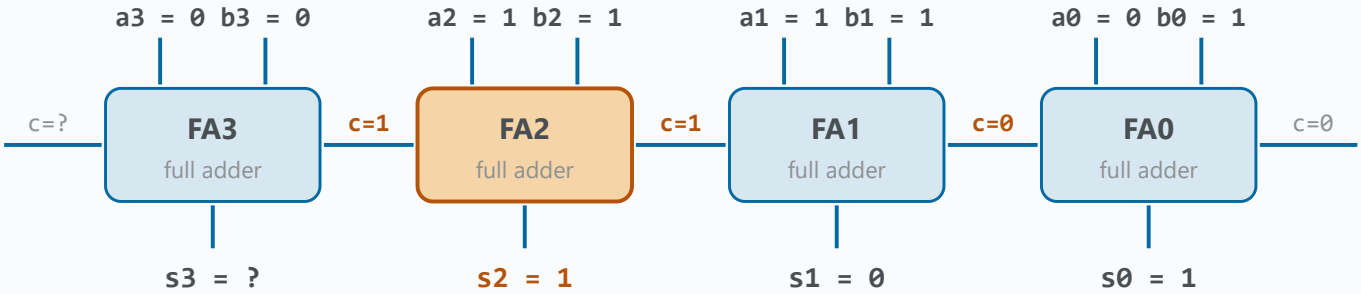
Step 3 / 6

```
1 a ← 0110   ; 6
2 b ← 0111   ; 7
3 c ← 0      ; initial carry
▶ for i = 0, 1, 2, 3: ; right to left
▶   t ← a[i] XOR b[i]
▶   s[i] ← t XOR c
▶   c ← (a[i] AND b[i]) OR (t AND c)
8 print(c, s)
```

VARIABLES

`i = 1` `t = 0` `c = 1` `s = ??01`

A Chain of Full Adders: 0110 + 0111



i = 2: 1 + 1 + 1 = 3 = 11_2 → **s2 = 1**, carry **1** again.

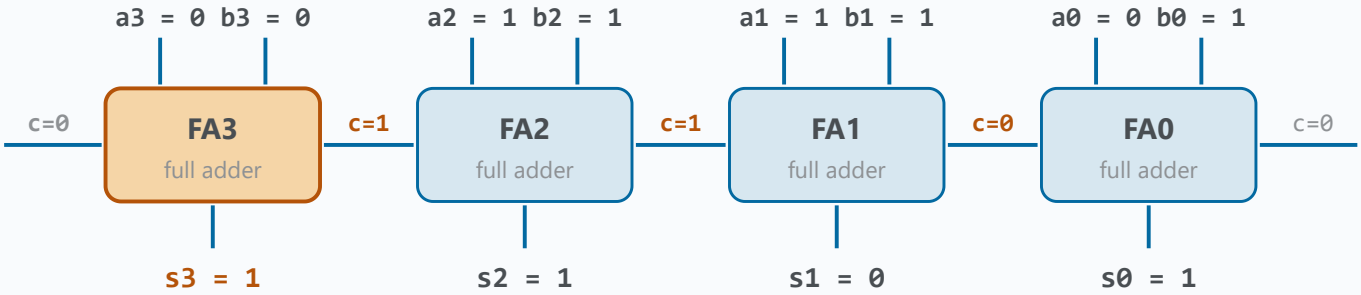
Step 4 / 6

```
1 a ← 0110  ; 6
2 b ← 0111  ; 7
3 c ← 0     ; initial carry
▶ for i = 0, 1, 2, 3: ; right to left
▶   t ← a[i] XOR b[i]
▶   s[i] ← t XOR c
▶   c ← (a[i] AND b[i]) OR (t AND c)
8 print(c, s)
```

VARIABLES

`i = 2` `t = 0` `c = 1` `s = ?101`

A Chain of Full Adders: 0110 + 0111



i = 3: 0 + 0 + 1 → **s3 = 1**, carry 0. The final carry is 0, so the result fits in 4 bits.

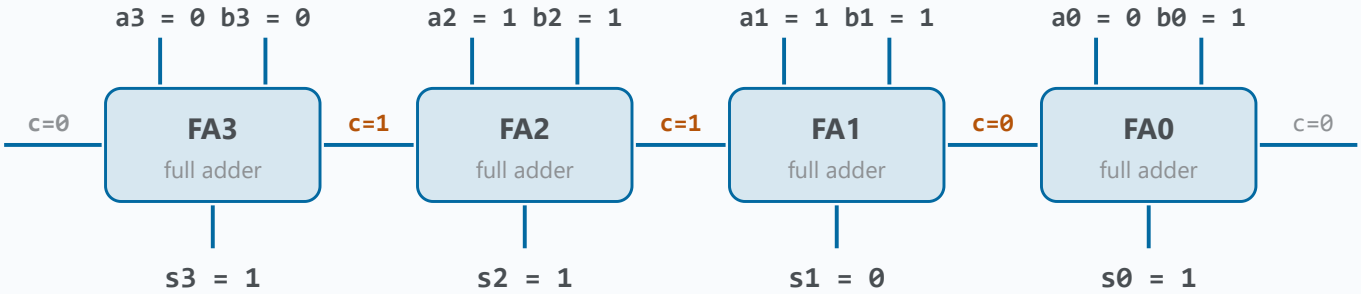
Step 5 / 6

```
1 a ← 0110  ; 6
2 b ← 0111  ; 7
3 c ← 0     ; initial carry
▶ for i = 0, 1, 2, 3: ; right to left
▶   t ← a[i] XOR b[i]
▶   s[i] ← t XOR c
▶   c ← (a[i] AND b[i]) OR (t AND c)
8 print(c, s)
```

VARIABLES

`i = 3` `t = 0` `c = 0` `s = 1101`

A Chain of Full Adders: 0110 + 0111



The result is **1101_2 = 13** = 6 + 7 ✓. The carry ripples from right to left (**ripple-carry adder**): each FA waits for the previous one, so the delay grows with the number of bits.

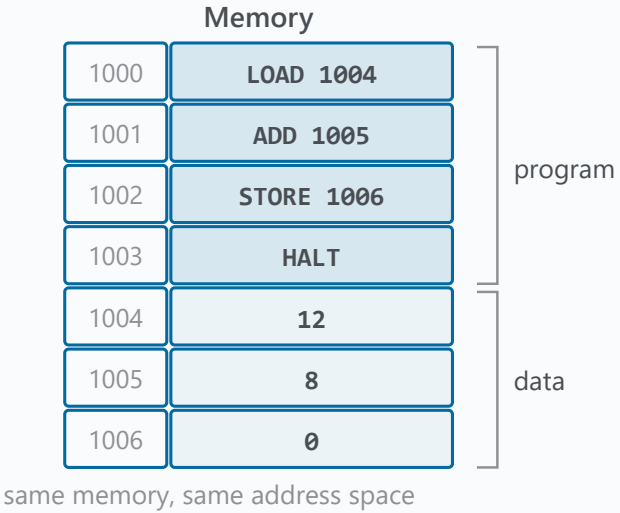
Step 6 / 6

```
1 a ← 0110  ; 6
2 b ← 0111  ; 7
3 c ← 0     ; initial carry
4 for i = 0, 1, 2, 3: ; right to left
5   t ← a[i] XOR b[i]
6   s[i] ← t XOR c
7   c ← (a[i] AND b[i]) OR (t AND c)
▶ print(c, s)
```

VARIABLES

`i = –` `t = –` `c = 0` `s = 1101`

The Stored-Program Concept



Program instructions and the data they work on are stored as numeric codes in **the same memory**. The processor reads and executes the instructions in order.

Flexibility

The same hardware does different jobs with different programs.

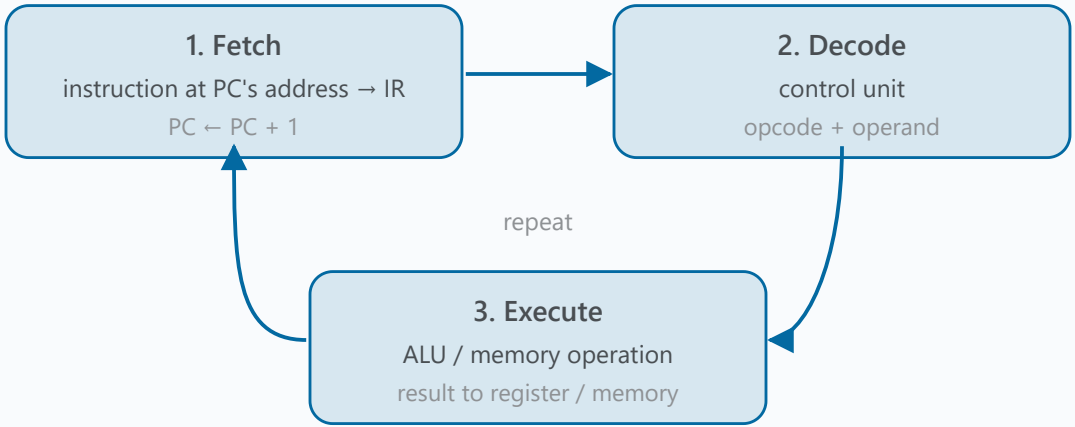
Updates

Behavior can change through software, without changing the hardware.

General purpose

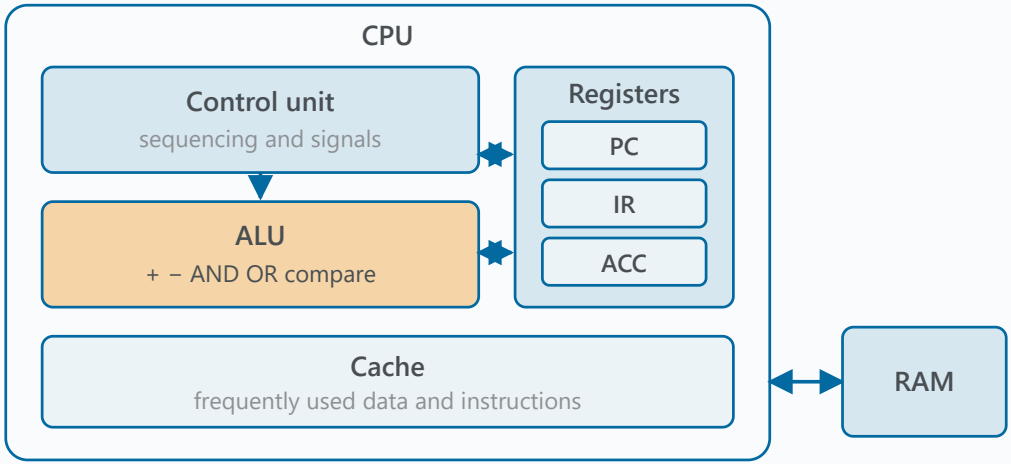
One system can handle computing, communication and media.

The Fetch–Decode–Execute Cycle



- The **program counter (PC)** holds the address of the next instruction; the **instruction register (IR)** holds the fetched instruction.
- The control unit interprets the operation code (opcode).
- The ALU or another unit carries out the operation.
- Instructions and data usually come from the processor's **cache** rather than RAM; registers and cache are much faster than RAM.

The Main Parts of a CPU



Control unit

Manages the order of instructions and the movement of data.

ALU

Performs arithmetic and logical operations.

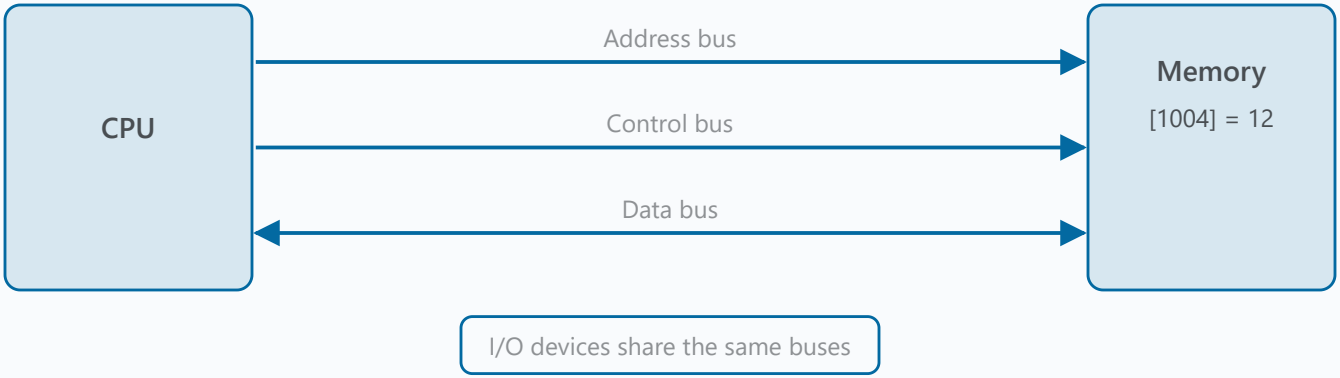
Registers

Small, fast storage used during an operation.

Cache

Keeps frequently used data and instructions close to the processor.

Buses: Reading from Memory

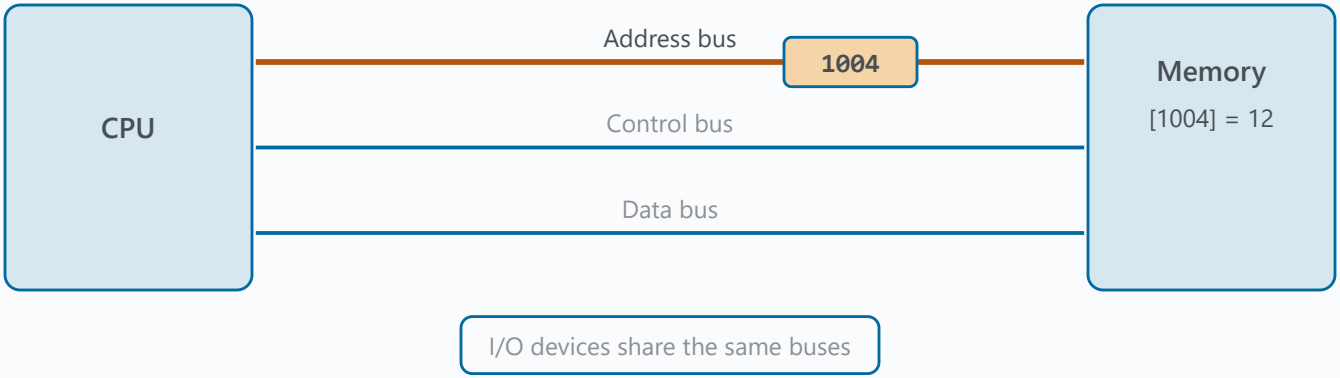


Three buses link the CPU to memory and I/O. Example: the CPU reads address 1004.

Step 1 / 4

Bus	Role
Data bus	Carries data between CPU, memory and devices.
Address bus	Selects the memory or device location.
Control bus	Carries read, write and timing signals.

Buses: Reading from Memory

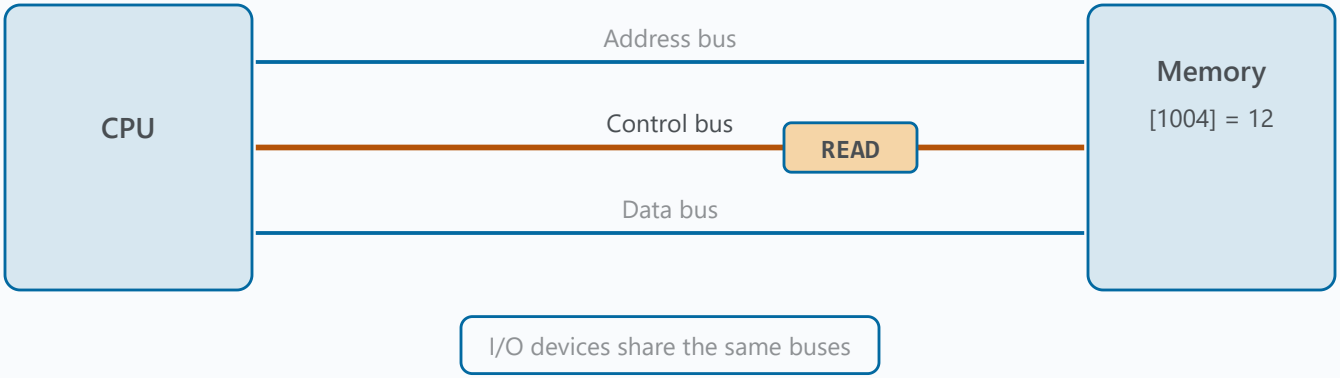


Address bus: the CPU sends the address → **1004**. One-way.

Step 2 / 4

Bus	Role
Data bus	Carries data between CPU, memory and devices.
Address bus	Selects the memory or device location.
Control bus	Carries read, write and timing signals.

Buses: Reading from Memory

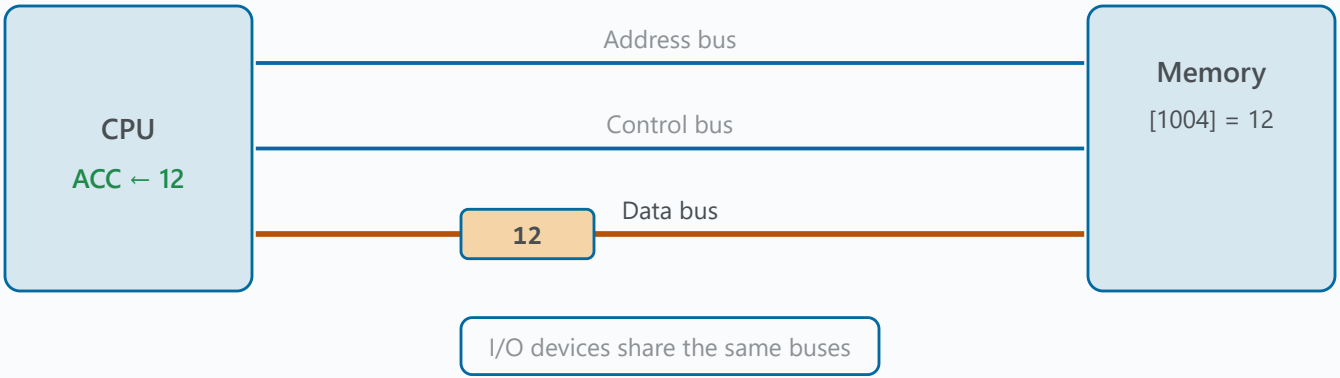


Control bus: the CPU sends **READ** (or WRITE), plus timing signals.

Step 3 / 4

Bus	Role
Data bus	Carries data between CPU, memory and devices.
Address bus	Selects the memory or device location.
Control bus	Carries read, write and timing signals.

Buses: Reading from Memory



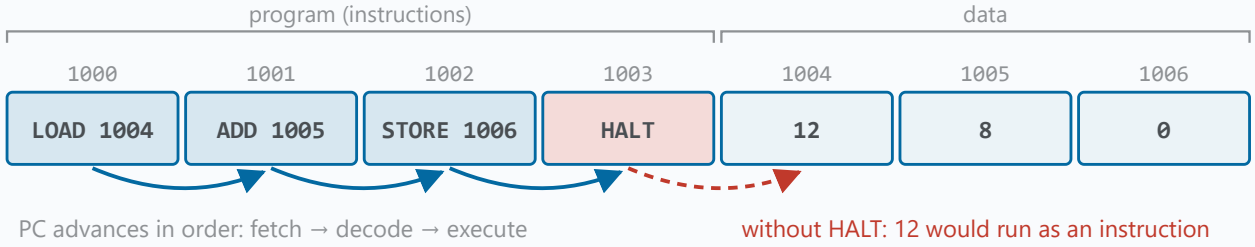
Data bus: memory returns the contents (**12**). It is bidirectional.

Bus	Role
Data bus	Carries data between CPU, memory and devices.
Address bus	Selects the memory or device location.
Control bus	Carries read, write and timing signals.

Memory Addresses

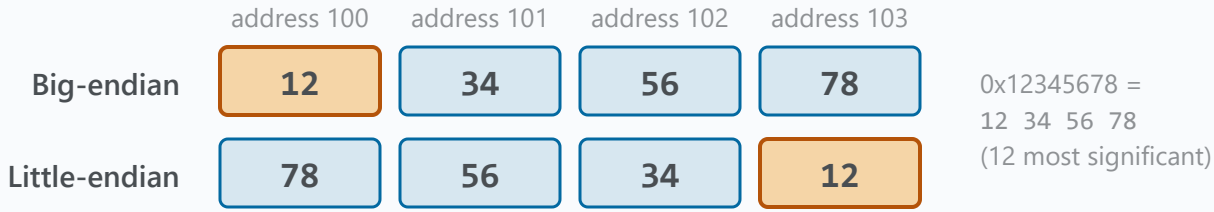
Address	Contents
1000	LOAD 1004
1001	ADD 1005
1002	STORE 1006
1003	HALT
1004	12
1005	8
1006	0

- Every memory cell has an address.
- Instructions reach data through addresses.
- The same bit string can be read as an instruction or as data.
- Without **HALT**, the CPU would treat the 12 at 1004 as an instruction and try to run it.



Multi-Byte Values and Byte Order

Memory is **byte-addressable**: each address holds 1 byte. A 32-bit `int` spans 4 consecutive addresses; the order in which its bytes are written is the **byte order** (endianness).



Big-endian

The most significant byte at the lowest address. Network protocols use this order (network byte order).

Little-endian

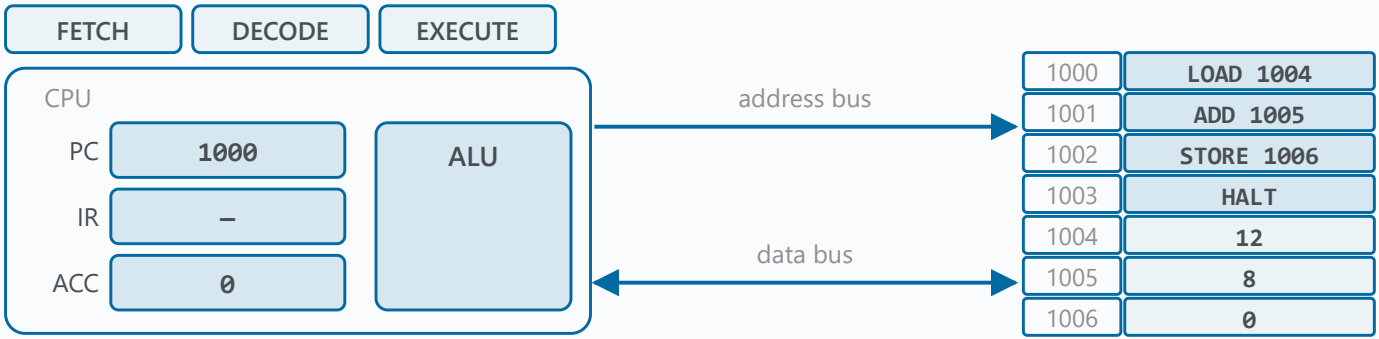
The least significant byte at the lowest address. x86-64 and most ARM systems work this way.

The problem

Raw binary data read on a machine with the opposite order turns 0x12345678 into 0x78563412.

Single bytes (e.g. ASCII characters) have no order problem; UTF-8 is read byte by byte, so byte order doesn't affect it either.

Running a Program Step by Step



Start: **PC = 1000** (address of the first instruction), ACC = 0. The CPU starts the cycle.

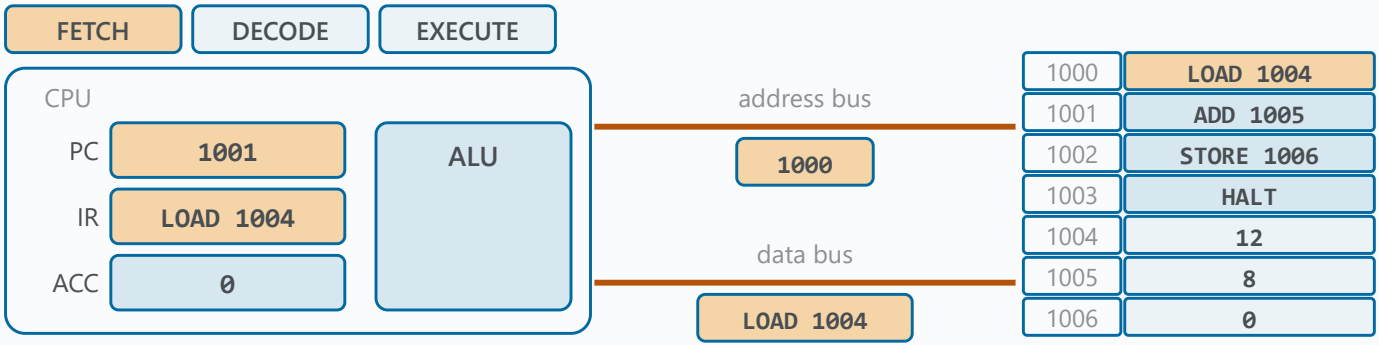
Step 1 / 12

```
▶ 1000: LOAD  1004 ; ACC ← mem[1004]
  1001: ADD   1005 ; ACC ← ACC + mem[1005]
  1002: STORE 1006 ; mem[1006] ← ACC
  1003: HALT       ; stop
  1004: 12         ; data
  1005: 8          ; data
  1006: 0          ; result
```

VARIABLES

`PC = 1000` `IR = –` `ACC = 0` `mem[1006] = 0`

Running a Program Step by Step



Fetch: PC's 1000 goes out on the address bus; the instruction `LOAD 1004` arrives in **IR** over the data bus. PC increments → **1001**.

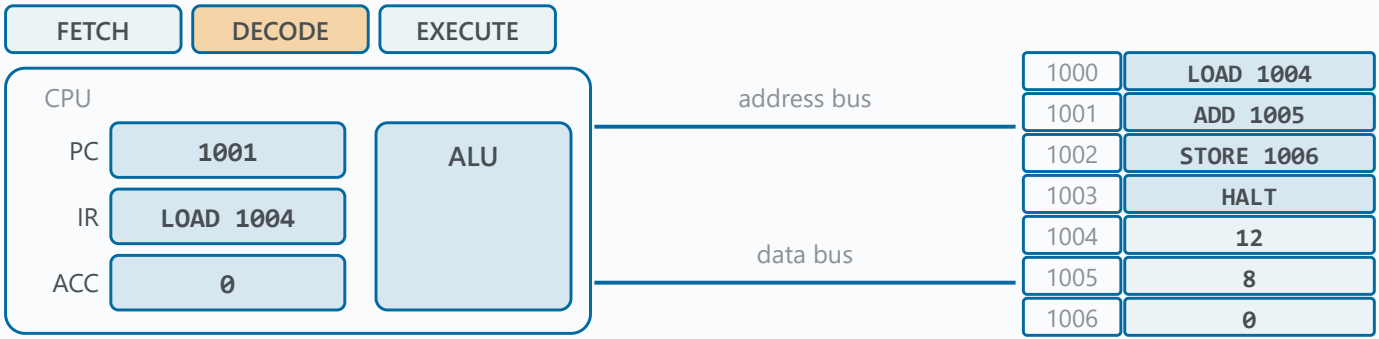
Step 2 / 12

```
▶ 1000: LOAD  1004 ; ACC ← mem[1004]
  1001: ADD   1005 ; ACC ← ACC + mem[1005]
  1002: STORE 1006 ; mem[1006] ← ACC
  1003: HALT       ; stop
  1004: 12         ; data
  1005: 8          ; data
  1006: 0          ; result
```

VARIABLES

`PC = 1001` `IR = LOAD 1004` `ACC = 0` `mem[1006] = 0`

Running a Program Step by Step



Decode: the control unit splits IR: opcode **LOAD**, address **1004** → mem[1004] will be read into ACC.

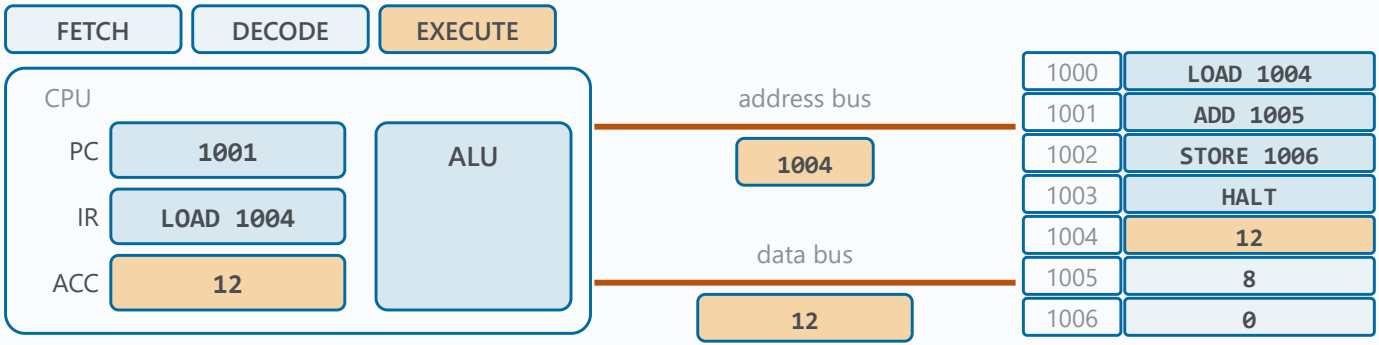
Step 3 / 12

```
▶ 1000: LOAD  1004 ; ACC ← mem[1004]
  1001: ADD   1005 ; ACC ← ACC + mem[1005]
  1002: STORE 1006 ; mem[1006] ← ACC
  1003: HALT       ; stop
  1004: 12         ; data
  1005: 8          ; data
  1006: 0          ; result
```

VARIABLES

PC = **1001** · IR = **LOAD 1004** · ACC = **0** · mem[1006] = **0**

Running a Program Step by Step



Execute: address 1004 is read and **12** is loaded into ACC.

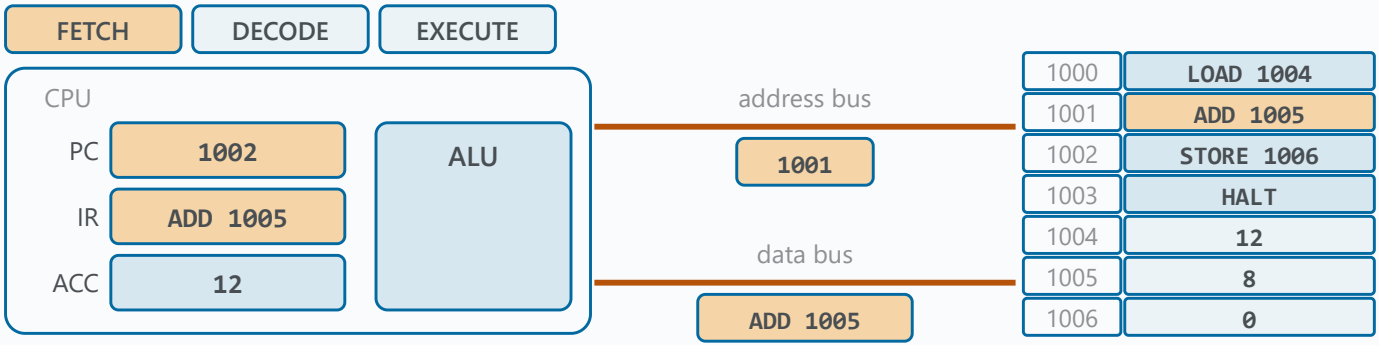
Step 4 / 12

```
▶ 1000: LOAD  1004 ; ACC ← mem[1004]
  1001: ADD   1005 ; ACC ← ACC + mem[1005]
  1002: STORE 1006 ; mem[1006] ← ACC
  1003: HALT       ; stop
▶ 1004: 12         ; data
  1005: 8          ; data
  1006: 0          ; result
```

VARIABLES

PC = **1001** IR = **LOAD 1004** ACC = **12** mem[1006] = **0**

Running a Program Step by Step



Fetch: PC's 1001 goes out on the address bus; the instruction `ADD 1005` arrives in **IR** over the data bus. PC increments → **1002**.

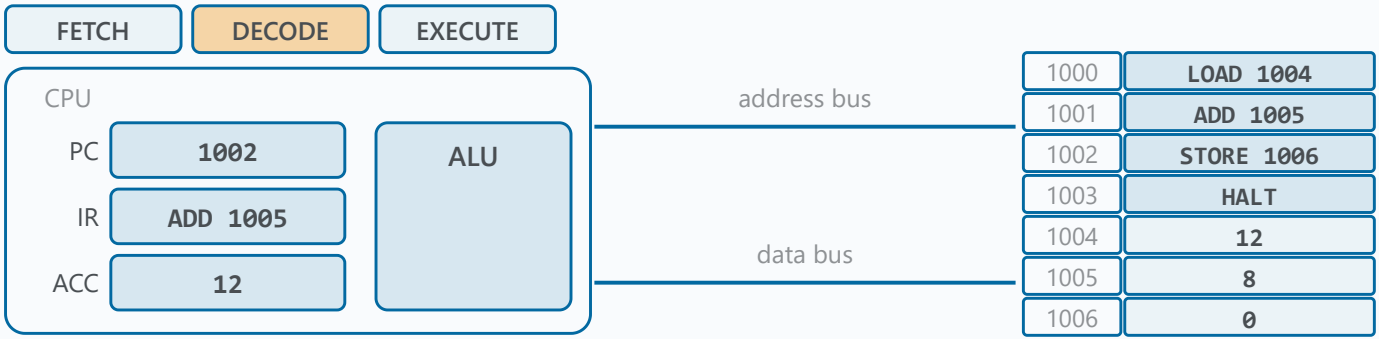
Step 5 / 12

```
1 1000: LOAD  1004 ; ACC ← mem[1004]
▶ 1001: ADD   1005 ; ACC ← ACC + mem[1005]
3 1002: STORE 1006 ; mem[1006] ← ACC
4 1003: HALT       ; stop
5 1004: 12         ; data
6 1005: 8          ; data
7 1006: 0          ; result
```

VARIABLES

PC = **1002** IR = **ADD 1005** ACC = **12** mem[1006] = **0**

Running a Program Step by Step



Decode: the control unit splits IR: opcode **ADD**, address **1005** → mem[1005] will be added to ACC.

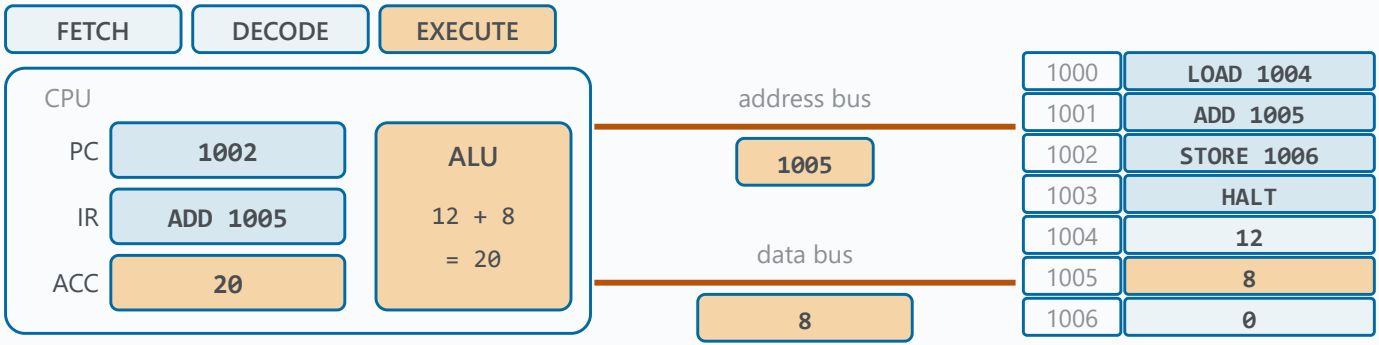
Step 6 / 12

```
1 1000: LOAD  1004 ; ACC ← mem[1004]
▶ 1001: ADD   1005 ; ACC ← ACC + mem[1005]
3 1002: STORE 1006 ; mem[1006] ← ACC
4 1003: HALT       ; stop
5 1004: 12         ; data
6 1005: 8          ; data
7 1006: 0          ; result
```

VARIABLES

PC = **1002** · IR = **ADD 1005** · ACC = **12** · mem[1006] = **0**

Running a Program Step by Step



Execute: the 8 at address 1005 is read, the **ALU** computes 12 + 8 = **20**, and the result goes into ACC.

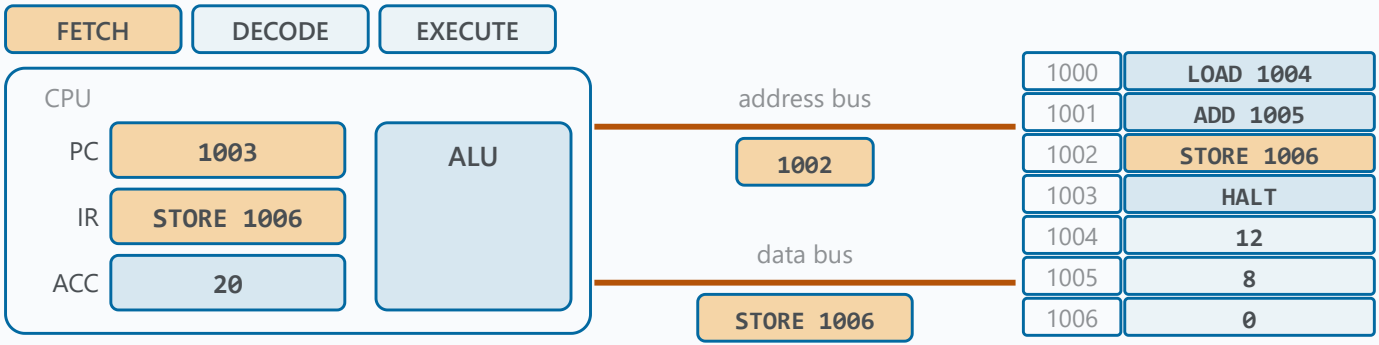
Step 7 / 12

```
1 1000: LOAD  1004 ; ACC ← mem[1004]
▶ 1001: ADD   1005 ; ACC ← ACC + mem[1005]
3 1002: STORE 1006 ; mem[1006] ← ACC
4 1003: HALT       ; stop
5 1004: 12         ; data
▶ 1005: 8          ; data
7 1006: 0          ; result
```

VARIABLES

PC = **1002** · IR = **ADD 1005** · ACC = **20** · mem[1006] = **0**

Running a Program Step by Step



Fetch: PC's 1002 goes out on the address bus; the instruction `STORE 1006` arrives in **IR** over the data bus. PC increments → **1003**.

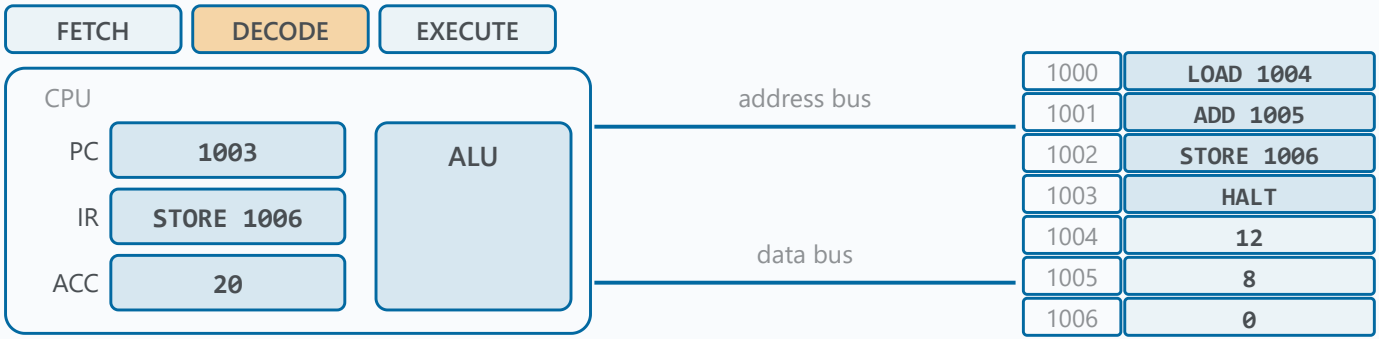
Step 8 / 12

```
1 1000: LOAD  1004 ; ACC ← mem[1004]
2 1001: ADD   1005 ; ACC ← ACC + mem[1005]
▶ 1002: STORE 1006 ; mem[1006] ← ACC
4 1003: HALT       ; stop
5 1004: 12         ; data
6 1005: 8          ; data
7 1006: 0          ; result
```

VARIABLES

`PC = 1003` `IR = STORE 1006` `ACC = 20` `mem[1006] = 0`

Running a Program Step by Step



Decode: the control unit splits IR: opcode **STORE**, address **1006** → ACC will be written to mem[1006].

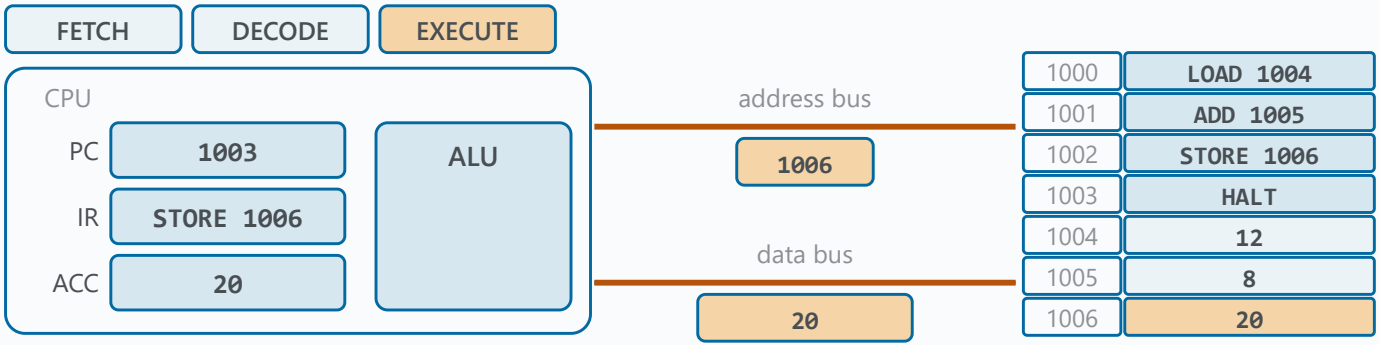
Step 9 / 12

```
1 1000: LOAD  1004 ; ACC ← mem[1004]
2 1001: ADD   1005 ; ACC ← ACC + mem[1005]
▶ 1002: STORE 1006 ; mem[1006] ← ACC
4 1003: HALT       ; stop
5 1004: 12         ; data
6 1005: 8          ; data
7 1006: 0          ; result
```

VARIABLES

`PC = 1003` `IR = STORE 1006` `ACC = 20` `mem[1006] = 0`

Running a Program Step by Step



Execute: the **20** in ACC is **written** to address 1006 over the data bus.

Step 10 / 12

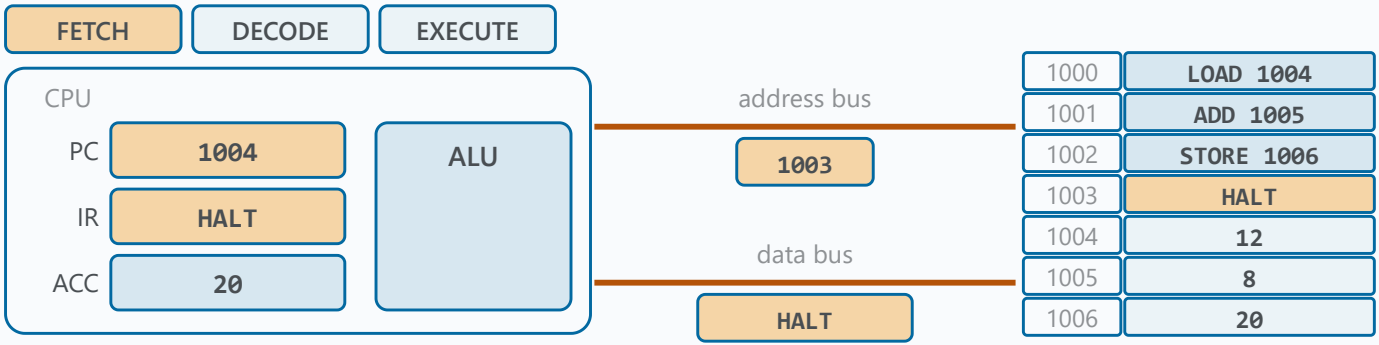
```
1 1000: LOAD  1004 ; ACC ← mem[1004]
2 1001: ADD   1005 ; ACC ← ACC + mem[1005]
▶ 1002: STORE 1006 ; mem[1006] ← ACC
4 1003: HALT       ; stop
5 1004: 12         ; data
6 1005: 8          ; data
▶ 1006: 0          ; result
```

VARIABLES

PC = **1003** · IR = **STORE 1006** · ACC = **20**

mem[1006] = **20**

Running a Program Step by Step



Fetch: PC's 1003 goes out on the address bus; the instruction `HALT` arrives in **IR** over the data bus. PC increments → **1004**.

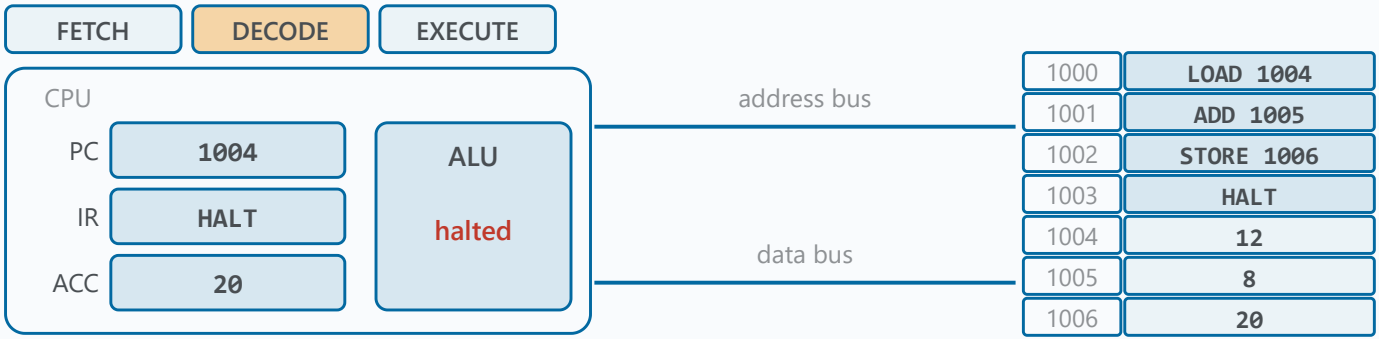
Step 11 / 12

```
1 1000: LOAD  1004 ; ACC ← mem[1004]
2 1001: ADD   1005 ; ACC ← ACC + mem[1005]
3 1002: STORE 1006 ; mem[1006] ← ACC
▶ 1003: HALT       ; stop
5 1004: 12         ; data
6 1005: 8          ; data
7 1006: 0          ; result
```

VARIABLES

`PC = 1004` `IR = HALT` `ACC = 20` `mem[1006] = 20`

Running a Program Step by Step



Decode: opcode HALT → the CPU stops. Result: mem[1006] = **20**. 4 instructions, 4 fetch–decode–execute cycles.

Step 12 / 12

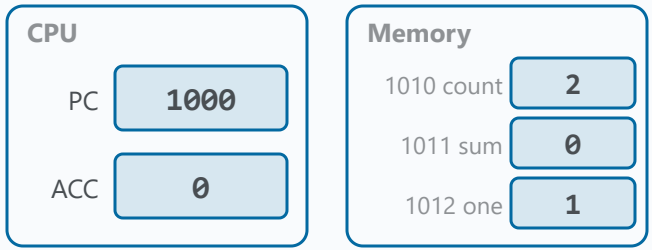
```
1 1000: LOAD  1004 ; ACC ← mem[1004]
2 1001: ADD   1005 ; ACC ← ACC + mem[1005]
3 1002: STORE 1006 ; mem[1006] ← ACC
▶ 1003: HALT       ; stop
5 1004: 12         ; data
6 1005: 8          ; data
7 1006: 0          ; result
```

VARIABLES

PC = **1004** IR = **HALT** ACC = **20** mem[1006] = **20**

Branching: How Does PC Change in a Loop?

```
1000: LOAD  1010  ; ACC ← count
1001: JUMPZ 1008  ; jump if ACC = 0
1002: ADD   1011  ; ACC ← ACC + sum
1003: STORE 1011  ; sum ← ACC
1004: LOAD  1010
1005: SUB   1012  ; ACC ← ACC − 1
1006: STORE 1010  ; count ← ACC
1007: JUMP  1000  ; PC ← 1000
1008: HALT
1010: 2   1011: 0   1012: 1
```



count runs down from 2; when the loop ends, sum = 2 + 1.

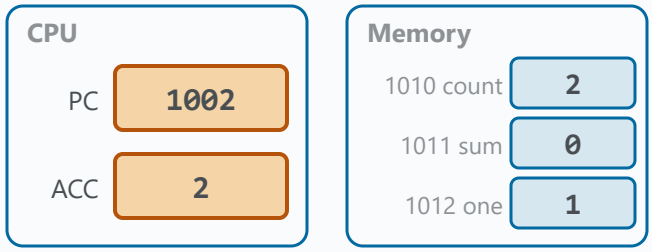
Step 1 / 8

VARIABLES

`PC = 1000` `ACC = 0` `count = 2` `sum = 0` `instr = 0`

Branching: How Does PC Change in a Loop?

```
▶ 1000: LOAD  1010  ; ACC ← count
▶ 1001: JUMPZ 1008  ; jump if ACC = 0
3 1002: ADD   1011  ; ACC ← ACC + sum
4 1003: STORE 1011  ; sum ← ACC
5 1004: LOAD  1010
6 1005: SUB   1012  ; ACC ← ACC − 1
7 1006: STORE 1010  ; count ← ACC
8 1007: JUMP  1000  ; PC ← 1000
9 1008: HALT
10 1010: 2   1011: 0   1012: 1
```



LOAD → ACC = 2. **JUMPZ**: ACC ≠ 0, no jump; PC stays at the next address, 1002.

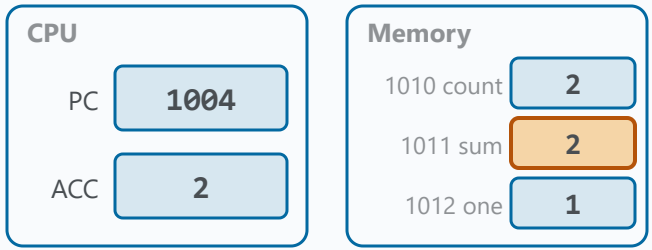
Step 2 / 8

VARIABLES

`PC = 1002` `ACC = 2` `count = 2` `sum = 0` `instr = 2`

Branching: How Does PC Change in a Loop?

```
1  1000: LOAD  1010  ; ACC ← count
2  1001: JUMPZ 1008  ; jump if ACC = 0
▶  1002: ADD   1011  ; ACC ← ACC + sum
▶  1003: STORE 1011  ; sum ← ACC
5  1004: LOAD  1010
6  1005: SUB   1012  ; ACC ← ACC − 1
7  1006: STORE 1010  ; count ← ACC
8  1007: JUMP  1000  ; PC ← 1000
9  1008: HALT
10 1010: 2   1011: 0   1012: 1
```



ADD: 2 + 0 = 2; STORE → sum = **2**.

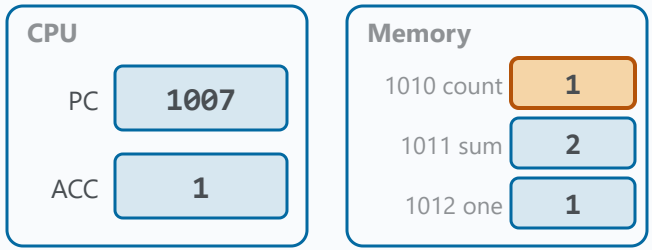
Step 3 / 8

VARIABLES

PC = **1004** | ACC = **2** | count = **2** | sum = **2** | instr = **4**

Branching: How Does PC Change in a Loop?

```
1  1000: LOAD  1010  ; ACC ← count
2  1001: JUMPZ 1008  ; jump if ACC = 0
3  1002: ADD   1011  ; ACC ← ACC + sum
4  1003: STORE 1011  ; sum ← ACC
▶  1004: LOAD  1010
▶  1005: SUB   1012  ; ACC ← ACC - 1
▶  1006: STORE 1010  ; count ← ACC
8  1007: JUMP  1000  ; PC ← 1000
9  1008: HALT
10 1010: 2   1011: 0   1012: 1
```



LOAD, SUB, STORE: count 2 − 1 = **1**.

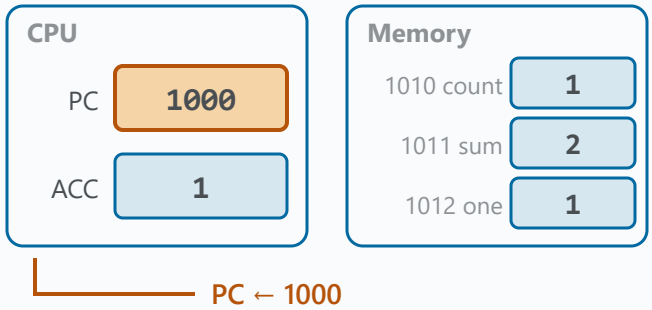
Step 4 / 8

VARIABLES

`PC = 1007` `ACC = 1` `count = 1` `sum = 2` `instr = 7`

Branching: How Does PC Change in a Loop?

```
1 1000: LOAD  1010  ; ACC ← count
2 1001: JUMPZ 1008  ; jump if ACC = 0
3 1002: ADD   1011  ; ACC ← ACC + sum
4 1003: STORE 1011  ; sum ← ACC
5 1004: LOAD  1010
6 1005: SUB   1012  ; ACC ← ACC − 1
7 1006: STORE 1010  ; count ← ACC
▶ 1007: JUMP  1000  ; PC ← 1000
9 1008: HALT
10 1010: 2   1011: 0   1012: 1
```



JUMP 1000: PC, which the fetch set to 1008, is overwritten with 1000. The loop goes back to the top.

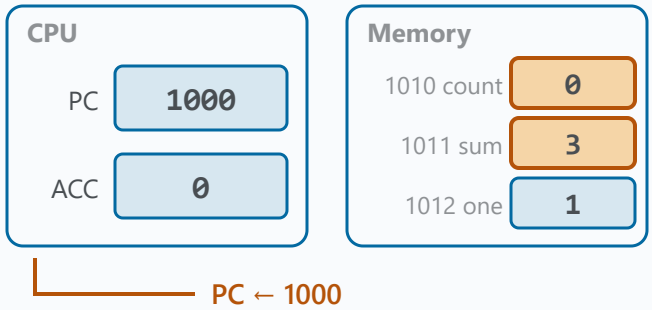
Step 5 / 8

VARIABLES

PC = 1000 | ACC = 1 | count = 1 | sum = 2 | instr = 8

Branching: How Does PC Change in a Loop?

```
1000: LOAD  1010  ; ACC ← count
1001: JUMPZ 1008  ; jump if ACC = 0
1002: ADD   1011  ; ACC ← ACC + sum
1003: STORE 1011  ; sum ← ACC
1004: LOAD  1010
1005: SUB   1012  ; ACC ← ACC − 1
1006: STORE 1010  ; count ← ACC
1007: JUMP  1000  ; PC ← 1000
1008: HALT
1010: 2   1011: 0   1012: 1
```



The second pass runs the same 8 instructions: sum = 1 + 2 = **3**, count = **0**, JUMP again.

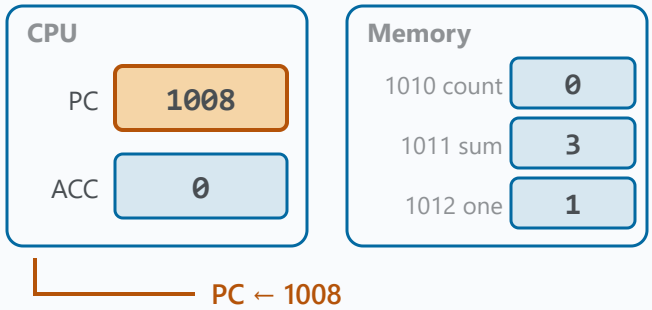
Step 6 / 8

VARIABLES

PC = **1000** ACC = **0** count = **0** sum = **3** instr = **16**

Branching: How Does PC Change in a Loop?

```
▶ 1000: LOAD  1010  ; ACC ← count
▶ 1001: JUMPZ 1008  ; jump if ACC = 0
  1002: ADD   1011  ; ACC ← ACC + sum
  1003: STORE 1011  ; sum ← ACC
  1004: LOAD  1010
  1005: SUB   1012  ; ACC ← ACC − 1
  1006: STORE 1010  ; count ← ACC
  1007: JUMP  1000  ; PC ← 1000
  1008: HALT
  1010: 2   1011: 0   1012: 1
```



LOAD → ACC = 0. This time **JUMPZ** jumps: PC ← 1008, skipping the loop body.

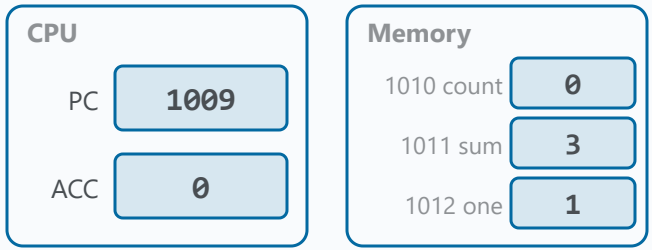
Step 7 / 8

VARIABLES

`PC = 1008` `ACC = 0` `count = 0` `sum = 3` `instr = 18`

Branching: How Does PC Change in a Loop?

```
1 1000: LOAD  1010  ; ACC ← count
2 1001: JUMPZ 1008  ; jump if ACC = 0
3 1002: ADD   1011  ; ACC ← ACC + sum
4 1003: STORE 1011  ; sum ← ACC
5 1004: LOAD  1010
6 1005: SUB   1012  ; ACC ← ACC − 1
7 1006: STORE 1010  ; count ← ACC
8 1007: JUMP  1000  ; PC ← 1000
▶ 1008: HALT
10 1010: 2   1011: 0   1012: 1
```



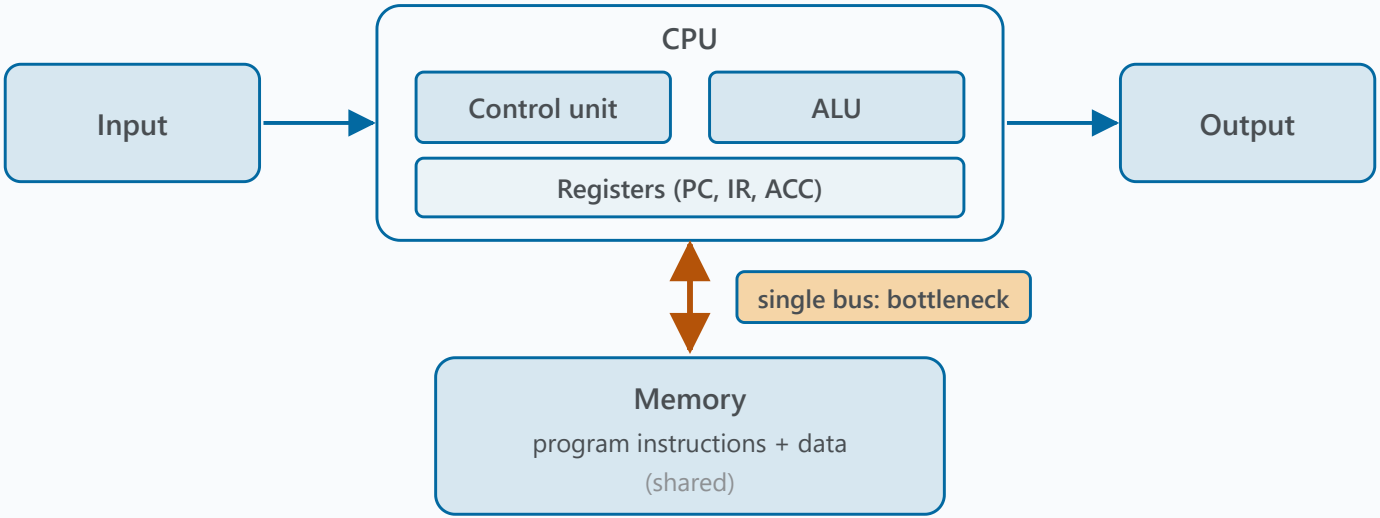
HALT: the CPU stops. **19 instructions** were fetched (8 + 8 + 3). A Java `while` loop compiles to conditional and unconditional jumps like these.

Step 8 / 8

VARIABLES

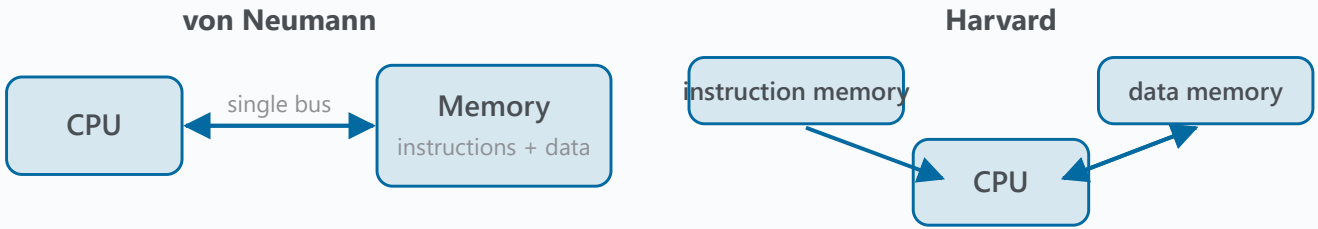
`PC = 1009` `ACC = 0` `count = 0` `sum = 3` `instr = 19`

The von Neumann Architecture



- Program instructions and data are kept in a shared memory.
- The CPU processes instructions in order; branch instructions can change that order.
- Instructions and data arrive over the same path, so CPU–memory transfer limits performance (the **von Neumann bottleneck**).

von Neumann vs. Harvard



Property	von Neumann	Harvard
Memory	Instructions and data share one memory	Instructions and data in separate memories
Bus	One bus: instructions and data move in turn	Two buses: instructions and data are read at once
Flexibility	A program can be loaded and modified like data	Program memory is usually fixed (flash)
Typical use	General-purpose computers	Microcontrollers, signal processors

Modern processors combine both: main memory is shared, but the L1 cache next to the core is split into instruction and data caches (modified Harvard).

COMPUTER ARCHITECTURE

Instruction Encoding

A machine instruction usually specifies the operation and the values or addresses it uses.

ADD 1005 →



Example (hypothetical) codes: HALT = 0000 · LOAD = 0001 · ADD = 0010 · STORE = 0011

Opcode

Identifies the kind of operation.

`ADD`

Operand

The register, value or address the operation uses.

`R1, R2`

Instruction format

The architecture fixes how many bit fields there are and where.

Instruction set

All the instructions the processor understands directly.

Instruction Set Design: RISC and CISC

Property	RISC	CISC
Instruction set	A few simple instructions	Many complex instructions
Instruction length	Fixed (e.g. 32 bits)	Variable (1–15 bytes on x86-64)
Memory access	Only LOAD / STORE instructions	Arithmetic instructions can take memory operands directly
Example	ARM, RISC-V	x86-64

RISC: `a = a + b` in memory

```
LOAD  R1, a
LOAD  R2, b
ADD   R1, R1, R2
STORE R1, a
```

CISC (x86-64): the same job

```
MOV EAX, [b]
ADD [a], EAX
```

2 instructions instead of 4, but the second is split internally into read, add and write steps.

Architectural Limits

Finite memory
Only so much data can be stored at once.

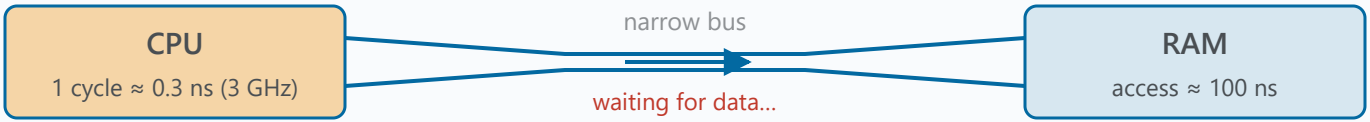
Finite speed
Every instruction costs time and energy.

Representation errors
A fixed number of bits can cause overflow and rounding.

Data bottleneck
The CPU can sit idle while waiting for data from memory.

Limits of parallelism
Not every problem can be parallelized equally well.

Security
Programs and data share system resources, so they must be isolated.



100 ns ÷ 0.3 ns ≈ 300 cycles: caches exist to cut this wait

Performance: Instruction Count, CPI and Clock Rate

CPU time = instruction count × CPI ÷ clock rate

CPI: average clock cycles per instruction

Program: 2×10^9 instructions	Clock rate	CPI	Time
Processor A	3 GHz	1.5	$2 \times 10^9 \times 1.5 \div (3 \times 10^9)$ = **1.0 s**
Processor B	4 GHz	2.5	$2 \times 10^9 \times 2.5 \div (4 \times 10^9)$ = **1.25 s**
A, with a better compiler (1.6×10^9 instructions)	3 GHz	1.5	$1.6 \times 10^9 \times 1.5 \div (3 \times 10^9)$ = **0.8 s**

Instruction count

Depends on the algorithm, compiler and instruction set.

CPI

Depends on the internal design and memory stalls (cache misses).

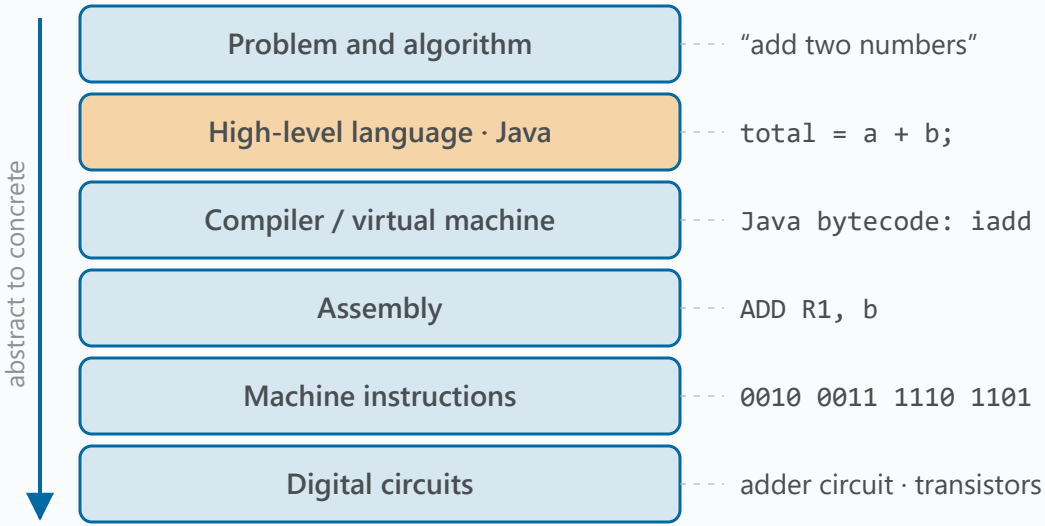
Clock rate

Limited by process technology, power and heat.

B has the higher clock rate, yet A is 1.25× faster: no single number determines performance.

PROGRAMMING

Levels of Abstraction in Programming



Each level hides the details of the one below. High-level languages are readable and portable; assembly and its binary form, machine code, are specific to a processor architecture. The sample lines are illustrative.

Tracing `total = a + b`



One line of Java becomes **three** processor instructions. Memory holds a = 12, b = 8.

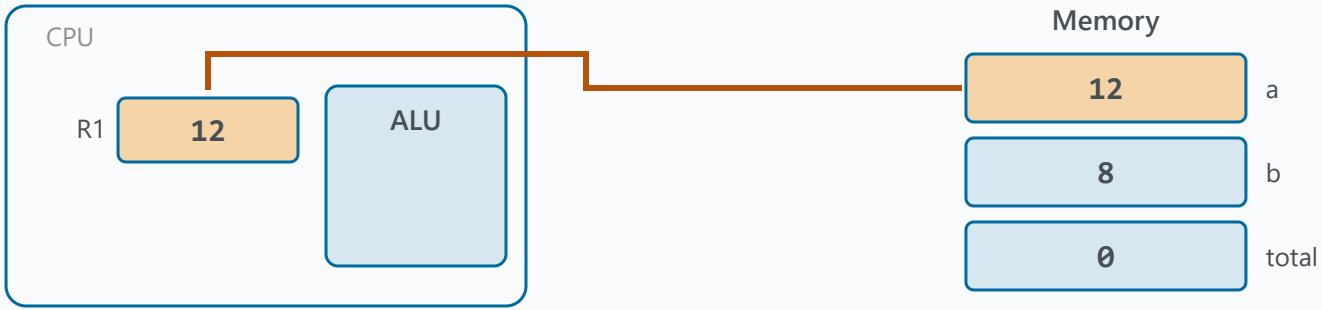
Step 1 / 4

```
▶ // Java: int total = a + b;
2 LOAD  R1, a      ; R1 ← a
3 ADD   R1, b      ; R1 ← R1 + b
4 STORE R1, total  ; total ← R1
```

VARIABLES

`R1 = 0` `a = 12` `b = 8` `total = 0`

Tracing `total = a + b`



LOAD: the value of a moves from memory into a register → R1 = **12**. The ALU only works on registers.

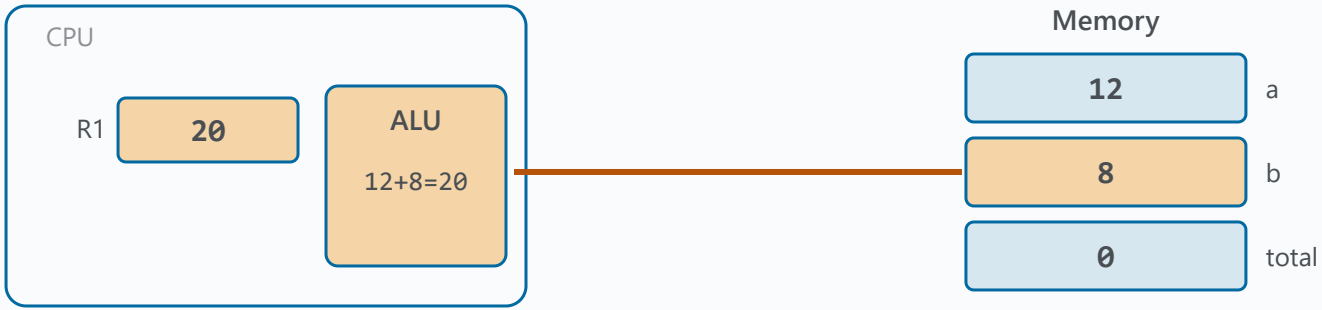
Step 2 / 4

```
1 // Java: int total = a + b;
▶ LOAD  R1, a       ; R1 ← a
3 ADD   R1, b       ; R1 ← R1 + b
4 STORE R1, total   ; total ← R1
```

VARIABLES

R1 = 12 | a = 12 | b = 8 | total = 0

Tracing `total = a + b`



ADD: b is read, the ALU computes 12 + 8 = **20**, and the result goes into R1.

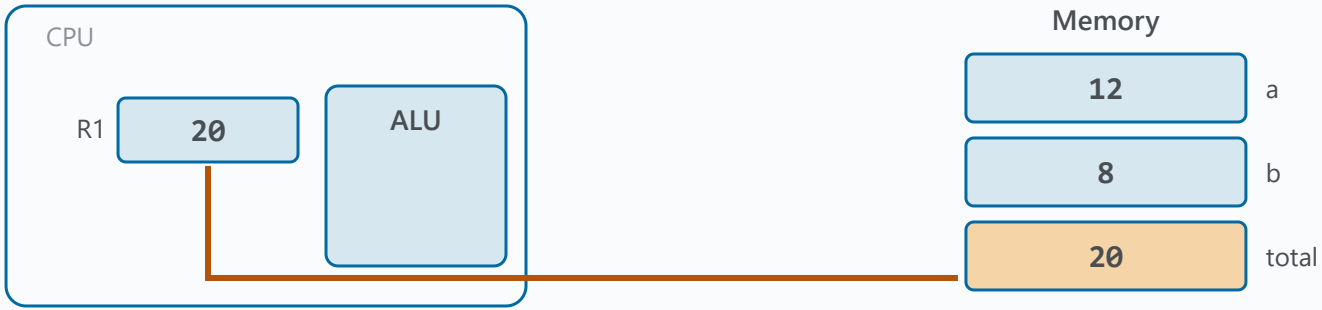
Step 3 / 4

```
1 // Java: int total = a + b;
2 LOAD  R1, a       ; R1 ← a
▶ ADD   R1, b       ; R1 ← R1 + b
4 STORE R1, total   ; total ← R1
```

VARIABLES

`R1 = 20` `a = 12` `b = 8` `total = 0`

Tracing `total = a + b`



STORE: the 20 in R1 is written to memory, at the location of total. The Java line is done: total = **20**.

Step 4 / 4

```
1 // Java: int total = a + b;
2 LOAD  R1, a       ; R1 ← a
3 ADD   R1, b       ; R1 ← R1 + b
▶ STORE R1, total   ; total ← R1
```

VARIABLES

`R1 = 20` `a = 12` `b = 8` `total = 20`

SUMMARY

Chapter Summary

- A computer represents all data as bit strings.
- Binary is exact for integers; floating point can only approximate some decimals.
- ASCII and Unicode map characters to numeric codes.
- In the stored-program model, instructions and data are kept in memory.
- Logic gates are the building blocks of circuits such as adders; binary addition uses a chain of full adders.
- The CPU runs instructions in the fetch–decode–execute cycle.
- High-level languages abstract away hardware details.

Review Questions

1. What is 1101_2 in decimal?
2. What is the range of an 8-bit signed integer?
3. What is the key difference between ASCII and Unicode?
4. What is the advantage of storing program instructions in memory?
5. What do the data, address and control buses do?
6. How does an assembly instruction relate to a machine instruction?

Exercises

1. In 8-bit two's complement, what number is 11111110?
2. Convert $3F_{16}$ to binary and decimal.
3. Encode “ç” (U+00E7) in UTF-8; how many bytes does it take?
4. What is 250 + 10 in an 8-bit unsigned register? Why?
5. In “Running a Program Step by Step”, if 1004 = 7 and 1005 = 5, what would be in 1006 when the program ends? How many instructions would be fetched?
6. Write the assembly for `int diff = a - b;` using LOAD / SUB / STORE.

▶ **Show answers**

References

- Patterson and Hennessy, Computer Organization and Design.
- Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective.
- Unicode Consortium, The Unicode Standard.
- IEEE 754, Floating-Point Arithmetic.

The End

Chapter 1.3: Computers

Previous: Chapter 1.2: Computer Hardware · Next: Chapter 1.4: The Internet