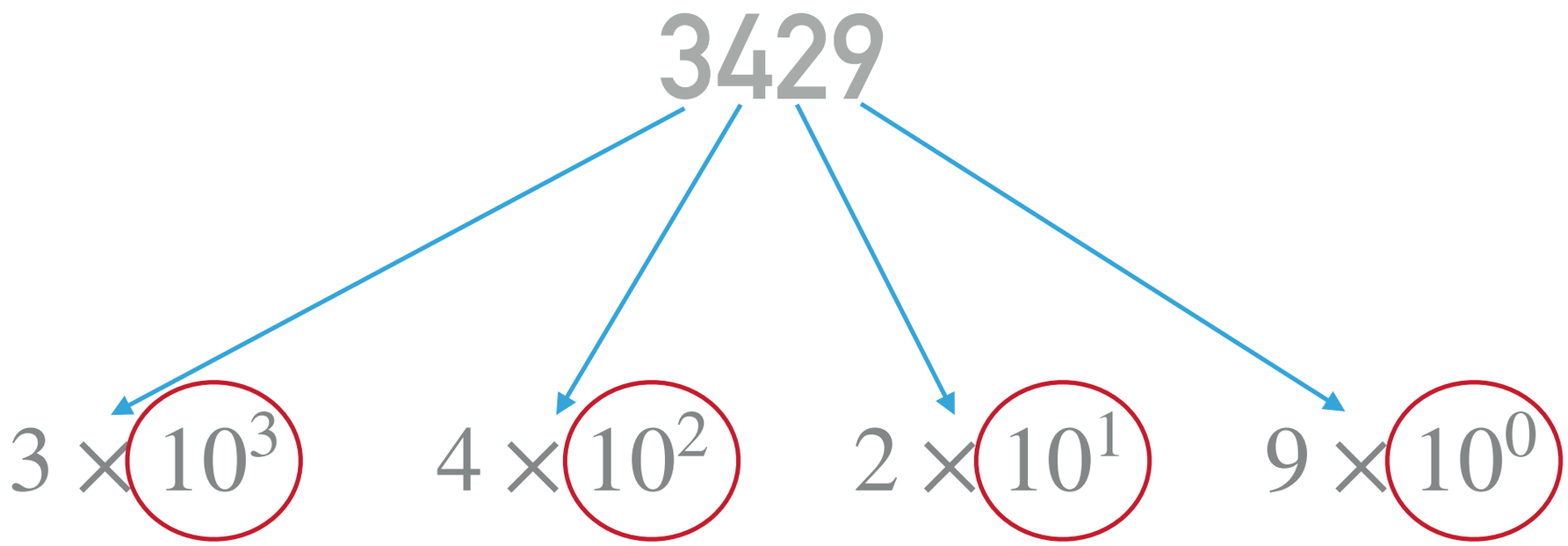
RAPPRESENTARE NUMERI
ARCHITETTURA DEL COMPUTER
COSA È UN SISTEMA OPERATIVO

INFORMATICA

RAPPRESENTAZIONE DEI NUMERI

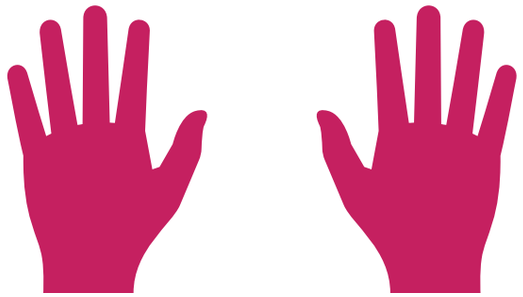
COME RAPPRESENTARE UN NUMERO



BASE 10

PERCHÉ BASE 10?

Suggerimento:



L’utilizzo di 10 cifre (da 0 a 9) non ha nessun particolare vantaggio (a parte contare con le dita)

Proviamo quindi a contare con meno dita (con altre basi)

CONTARE IN BASE 8

0, 1, 2, 3, 4, 5, 6, 7,...

E dopo?

Abbiamo incontrato un multiplo di 8 (e finito le cifre), quindi:

Non confondere 10 in base 8 con 10 in base 10!

CONTARE IN BASI DIVERSE

Base 8

0, 1, 2, 3, 4, 5, 6, 7,10,11,12,13,14,15,16,17,20,...

Base 4

0, 1, 2, 3,10,11,12,13,20,21,22,23,30,31,32,33,100,101,102,103...

Ma quanto vale, per esempio, 175 in base 8?

$$
\begin{aligned}
175_8 &= 1 \times 8^2 + 7 \times 8^1 + 5 \times 8^0 \\
&= 64 + 56 + 5 \\
&= 125_{10}
\end{aligned}
$$

BASI MAGGIORI DI 10

Possiamo usare basi maggiori di 10 aggiungendo “nuove cifre”.
Per convenzione si utilizzano le lettere dell’alfabeto

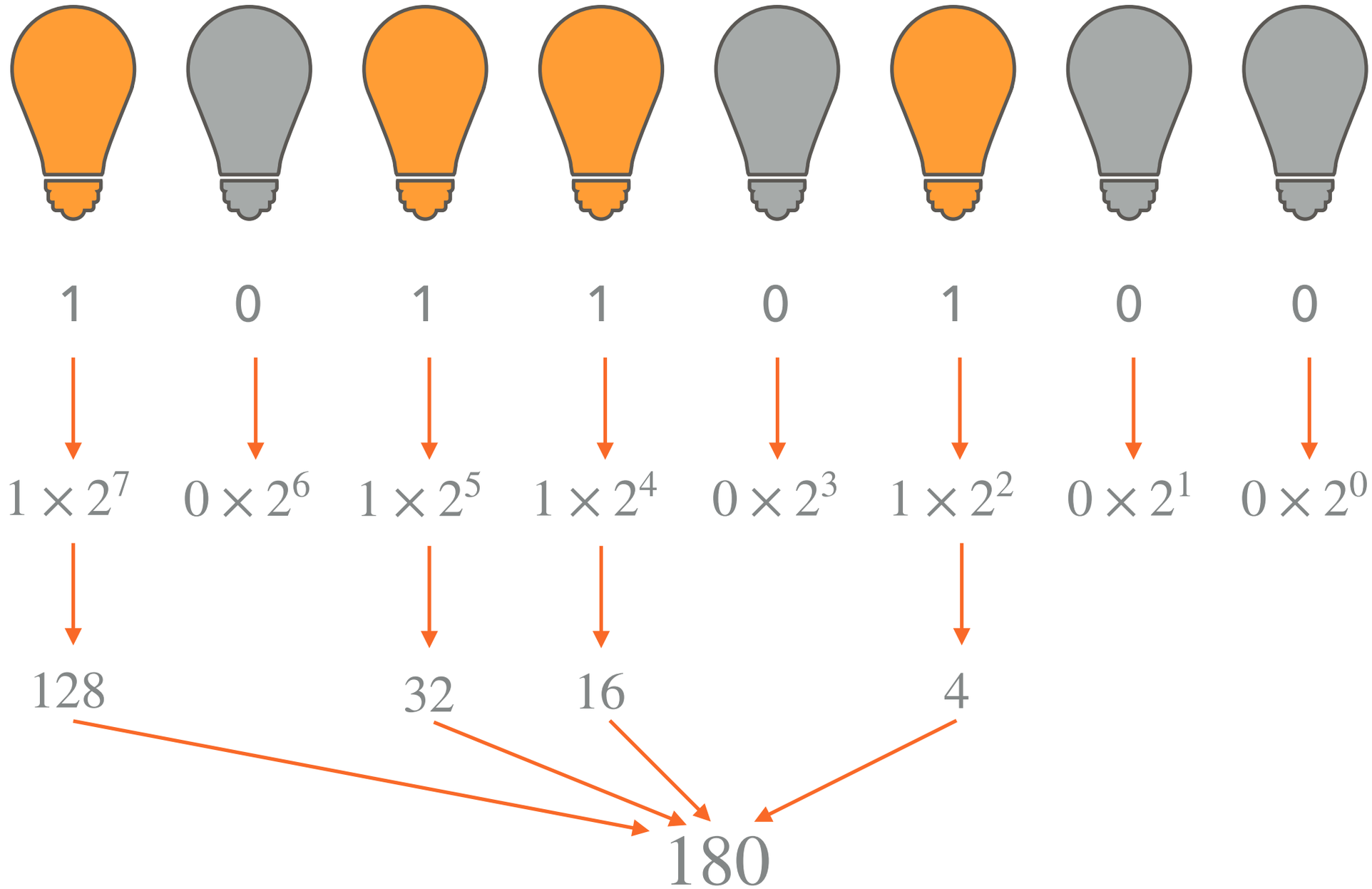
Base 16

Cifre: 0, 1, 2, 3, 4, 5, 6, 7,8,9,A,B,C,D,E,F

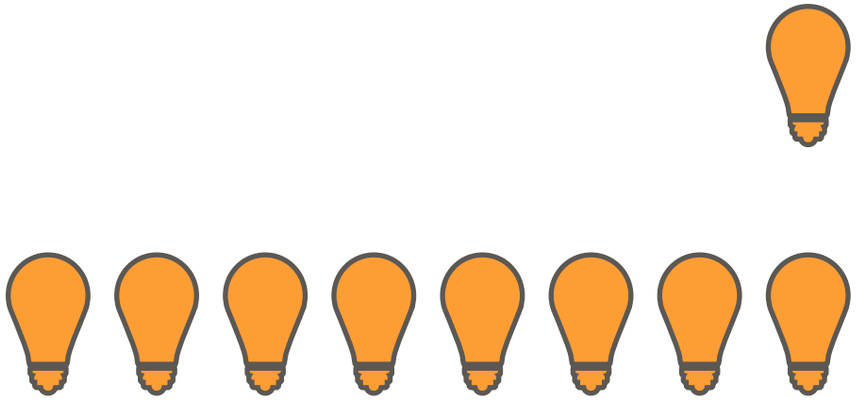
Quindi 7A5 in base 16 rappresenta il numero

$$
\begin{aligned}
7A5_{16} &= 7 \times 16^2 + 10 \times 16^1 + 5 \times 16^0 \\
&= 1792 + 160 + 5 \\
&= 1957_{10}
\end{aligned}
$$

LA BASE 2


1 0 1 1 0 1 0 0
1×2^7 0×2^6 1×2^5 1×2^4 0×2^3 1×2^2 0×2^1 0×2^0
128 32 16 4
180

UNITÀ DI MISURA STANDARD



Bit	0 o 1, 2 valori rappresentabili
Byte	8 bit, 256 valori rappresentabili
Kilobyte (KB)	1024 byte
Megabyte (MB)	1024 KB
Gigabyte (GB)	1024 MB
Terabyte (TB)	1024 GB

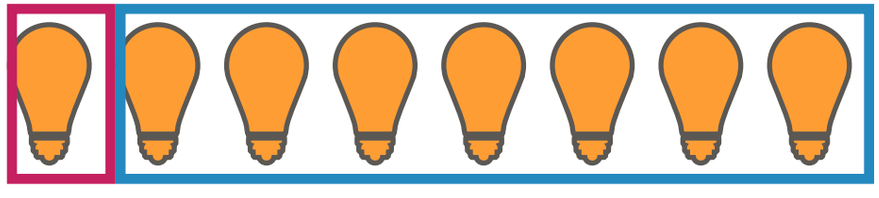
Nota Bene

A seconda dell'ambito a volte si considerano KB, MB, etc. come 1000 byte, un milione di byte, etc.

NUMERI NEGATIVI

- Fino ad ora abbiamo visto come rappresentare interi non negativi.
- Come facciamo ad avere numeri negativi?
- Non possiamo aggiungere un simbolo extra (abbiamo solo “lampadine”)
- Usiamo un interpretazione diversa dei valori binari

BIT DI SEGNO



Numero (7 bit)

Bit di segno (e.g., 0 = positivo, 1 = negativo)

`10001110` equivale a `−0001110` ovvero `−14`

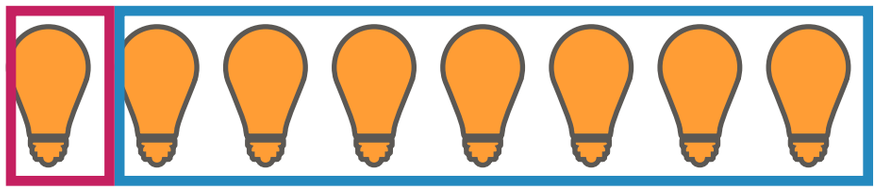
`00011001` equivale a `+0011001` ovvero `+25`

Problemi:

`10000000` e `00000000` rappresentano `−0` e `+0`

Circuiti più complessi

COMPLEMENTO A DUE



Numero (7 bit)

Bit interpretato “con segno meno”
(i.e., se a 1vale -128)

`10001110` equivale a `−10000000+00001110`

ovvero `−128 +8 +4 +2 = −128+14 = −112`

In generale:

Con n bit il primo bit rappresenta il valore $-2^{(n-1)}$
per gli altri non cambia nulla

QUIZ SUL COMPLEMENTO A DUE

```
11111111
01111111
```

In complemento a due che valori rappresentano quei due numeri di 8bit?

1. 255 e 127
2. -128 e +128
3. -1 e +127
4. 🤷

$$2^0 = 1$$
$$2^1 = 2$$
$$2^2 = 4$$
$$2^3 = 8$$
$$2^4 = 16$$
$$2^5 = 32$$
$$2^6 = 64$$
$$2^7 = 128$$

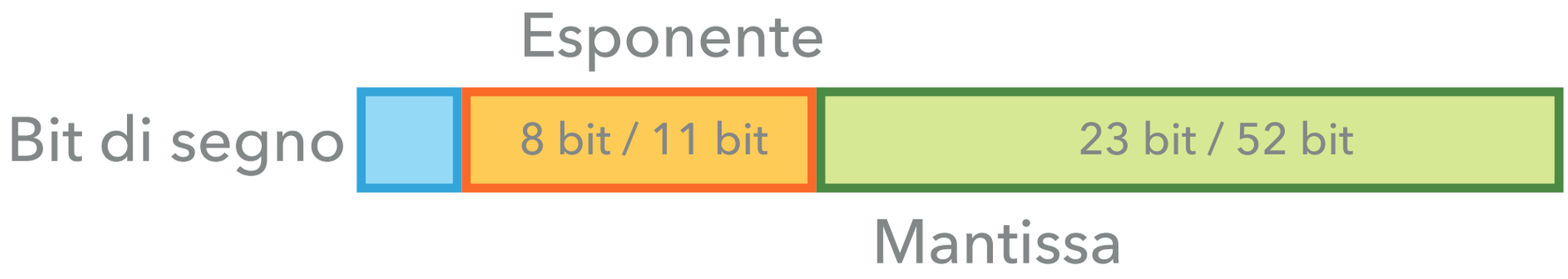
STORIA DEI NUMERI FLOATING POINT

- La rappresentazione dei numeri con la virgola non è immediata...
- ...soprattutto perché ci sono usi diversi:
  a volte è richiesta una precisione fissata,
  a volte invece serve un numero di cifre significative fissate
- Prima della definizione di uno standard ogni computer aveva un suo modo di rappresentare questi numeri (non necessariamente compatibile con altri)

STANDARD IEEE754

- Definisce come sono rappresentati i numeri float e come funzionano le operazioni su di essi.
- Single precision: 32 bit, double precision: 64 bit
- I numeri sono rappresentati con segno, esponente e mantissa
- Rappresentazione per NaN (not a number)
- Rappresentazione per +Inf e -Inf

IEEE 754



$$\pm \text{mantissa} \times 2^{\text{esponente}}$$

- La precisione è quindi limitata: abbiamo solo un numero di bit finito...
- ...quindi i calcoli che facciamo sono approssimati

FLOAT: GOTCHA

- Non è detto che le operazioni siano associative
- Non testate l'uguaglianza ma guardate se siete "abbastanza vicini" al valore corretto
- Operazioni matematicamente equivalenti possono dare risultati diversi a causa delle diverse approssimazioni
- L'ambito di ricerca di che si occupa di far uscire i risultati giusti con calcoli approssimati è "calcolo scientifico"

QUIZ SULLA RAPPRESENTAZIONE DEI NUMERI

Se avete il seguenti 32 bit:

```
11001011 11000001 00001100 00000101
```

Che tipo di numero rappresentano?
e.g., intero, intero in complemento a due, float, etc.

1. Intero positivo
2. Intero in complemento a due
3. Float
4. Dipende da come lo interpretiamo!

RAPPRESENTARE TESTO

Come possiamo rappresentare del testo?
Usando comunque dei bit, ma con *interpretazione* diversa

Avrete notato che è l'interpretazione (codifica o encoding) che diamo a sequenze di bit che fornisce il significato

Numero	Carattere
40	(
41	)
48	0
65	A

Sono presenti più modi di codificare i caratteri: tradizionalmente ASCII (7 bit per carattere), oggi Unicode

Notare la differenza tra il numero (48 in base 10) e il carattere 0 (che rappresenta graficamente la cifra 0)

ARCHITETTURA DEL CALCOLATORE

FORME DI UN PROGRAMMA

- File sorgente
  Viene scritto dalle persone
  Linguaggi: C, Fortran, Lisp, Python, Basic, ...

- File eseguibile
  Sequenza di istruzioni interpretabile dal calcolatore
  Linguaggio macchina: x86, x64, arm, arm64,...

- Forma dinamica (processo)
  Il programma in esecuzione

FILE SORGENTE

- Scritto in un linguaggio di programmazione
- Generalmente un file di testo
- Il linguaggio ha una grammatica ben precisa con una sua sintassi e semantica
- Noi studiamo Python, ma i principi sono applicabili a molti altri linguaggi
- Non è in un linguaggio direttamente comprensibile dal calcolatore

CI SONO TANTI LINGUAGGI DIVERSI

Common Lisp

```
(loop for i from 0 to 9 do
      (format t "~A * ~A = ~A~%" i i (* i i)))
```

Python 3

```
for i in range(0, 10):
    print(f"{i} * {i} = {i * i}\n")
```

Java

```
public static void main (String[] args) {
    for (int i = 0; i < 10; i++) {
        System.out.print("" + i + " * " + i + " = ");
        System.out.println("" + (i * i));
    }
}
```

BUONE NORME PER LA SCRITTURA DEL CODICE

- Il codice deve essere leggibile, non solo funzionante (conta anche come è impaginato)
- Il codice deve essere documentato, spiegando non solo cosa fa ma anche perché lo fa

MA PERCHÉ?

- Altre persone devono essere in grado di lavorare sul codice sorgente…
- …o anche l'autore originale dopo qualche mese potrebbe non ricordarsi più tutti i dettagli

COME FA IL COMPUTER A "CAPIRE" IL CODICE SORGENTE?

- Il computer non esegue il codice sorgente
- Il codice sorgente viene trasformato in istruzioni valide tramite un **compilatore**
- Il compilatore legge file contenente il codice sorgente e genera un file con il **codice eseguibile**
- Una alternativa, usata da Python, è l'utilizzo di un programma chiamato **interprete**.

IL COMPILATORE

```
#include<stdio.h>
#include<stdlib.h>

int main()
{
    for (int i = 0; i < 10; i++) {
       printf("%d * %d = %d\n", i, i, i*i);
    }
    exit(EXIT_SUCCESS);
}
```

COMPILATORE
(ANCHE LUI UN PROGRAMMA)

Codice macchina

```
cffa edfe 0700 0001 0300 0080 0200 0000
0f00 0000 c004 0000 8500 2000 0000 0000
1900 0000 4800 0000 5f5f 5041 4745 5a45
524f 0000 0000 0000 0000 0000 0000 0000
0000 0000 0100 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 1900 0000 d801 0000
5f5f 5445 5854 0000 0000 0000 0000 0000
0000 0000 0100 0000 0010 0000 0000 0000
0000 0000 0000 0000 0010 0000 0000 0000
0700 0000 0500 0000 0500 0000 0000 0000
5f5f 7465 7874 0000 0000 0000 0000 0000
5f5f 5445 5854 0000 0000 0000 0000 0000
470f 0000 0100 0000 3100 0000 0000 0000
470f 0000 0000 0000 0000 0000 0000 0000
0004 0080 0000 0000 0000 0000 0000 0000
5f5f 7374 7562 7300 0000 0000 0000 0000
5f5f 5445 5854 0000 0000 0000 0000 0000
780f 0000 0100 0000 0c00 0000 0000 0000
780f 0000 0100 0000 0000 0000 0000 0000
0804 0080 0000 0000 0600 0000 0000 0000
5f5f 7374 7562 5f68 656c 7065 7200 0000
…
```

ASSEMBLY

Esiste una rappresentazione simbolica delle istruzioni in codice macchina

Ad esempio se incontriamo `4889E5`
lo scriviamo come `mov rbp, rsp`

```
#include<stdio.h>
#include<stdlib.h>

int main()
{
    for (int i = 0; i < 10; i++) {
       printf("%d * %d = %d\n", i, i, i*i);
    }
    exit(EXIT_SUCCESS);
}
```

```
.LC0:
    .string "%d * %d = %d\n"
main:
    push rbp
    mov rbp, rsp
    sub rsp, 16
    mov DWORD PTR[rbp-4], 0
    jmp .L2
.L3:
    mov eax, DWORD PTR[rbp-4]
    imul eax, DWORD PTR[rbp-4]
    mov ecx, eax
    mov edx, DWORD PTR[rbp-4]
    mov eax, DWORD PTR[rbp-4]
    mov esi, eax
    mov edi, OFFSET FLAT:.LC0
    mov eax, 0
    call printf
    add DWORD PTR[rbp-4], 1
.L2:
    cmp DWORDP TR[rbp-4], 9
    jle .L3
    mov edi, 0
    call exit
```

ASSEMBLY: NON È UNO SOLO

Ogni architettura ha un suo linguaggio macchina
(e assembly corrispondente)

```
.LC0:
    .string "%d * %d = %d\n"
main:
    push rbp
    mov rbp, rsp
    sub rsp, 16
    mov DWORD PTR[rbp-4], 0
    jmp .L2
.L3:
    mov eax, DWORD PTR[rbp-4]
    imul eax, DWORD PTR[rbp-4]
    mov ecx, eax
    mov edx, DWORD PTR[rbp-4]
    mov eax, DWORD PTR[rbp-4]
    mov esi, eax
    mov edi, OFFSET FLAT:.LC0
    mov eax, 0
    call printf
    add DWORD PTR[rbp-4], 1
.L2:
    cmp DWORDP TR[rbp-4], 9
    jle .L3
    mov edi, 0
    call exit
```

x86-64

ARM64

```
.LC0:
    .string"%d * %d = %d\n"
main:
    stp x29, x30, [sp, -32]!
    mov x29, sp
    str wzr, [sp, 28]
    b .L2
.L3:
    ldr w1, [sp, 28]
    ldr w0, [sp, 28]
    mul w0, w1, w0
    mov w3, w0
    ldr w2, [sp, 28]
    ldr w1, [sp, 28]
    adrp x0, .LC0
    add x0, x0, :lo12:.LC0
    bl printf
    ldr w0, [sp, 28]
    add w0, w0, 1
    str w0, [sp, 28]
.L2:
    ldr w0, [sp, 28]
    cmp w0, 9
    ble .L3
    mov w0, 0
    bl exit
```

PERCHÉ NON USARE DIRETTAMENTE IL LINGUAGGIO MACCHINA?

- Le istruzioni sono molto più a basso livello
- È più difficile scrivere programmi corretti
- È più difficile scrivere programmi leggibili
- Ogni architettura di processore ha un linguaggio macchina diverso
- Lo stesso programma in un linguaggio di alto livello può essere ricompilato su architetture diverse

INTERPRETI

```
for i in range(0,10):
    print(i*i)
```


INTERPRETE (ANCHE LUI UN PROGRAMMA)

ESEGUE DIRETTAMENTE LE ISTRUZIONI CONTENUTE

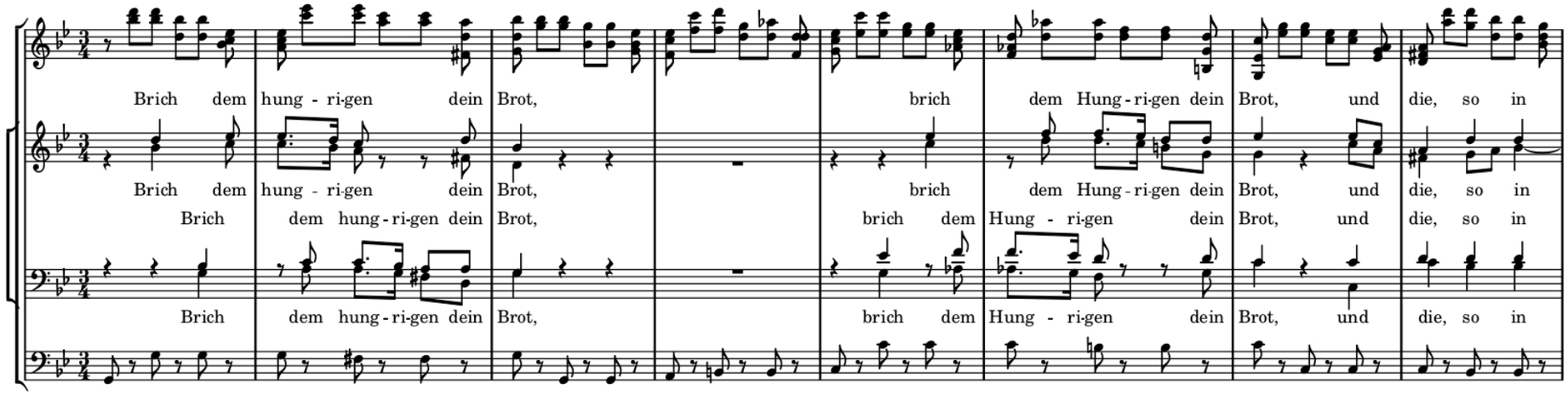
Non viene creato direttamente il codice macchina

```
cffa edfe 0700 0001 0300 0080 0200 0000
0f00 0000 c004 0000 8500 2000 0000 0000
1900 0000 4800 0000 5f5f 5041 4745 5a45
524f 0000 0000 0000 0000 0000 0000 0000
0000 0000 0100 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
…
```

INTERPRETI VS COMPILATORI

- Un modo di vedere la differenza tra interpreti e compilatori è il seguente:
- Il vostro codice sorgente è come un libro scritto in una lingua che il computer non parla
- Il compilatore prende il libro e prepara una sua traduzione
- L'interprete legge ad alta voce il libro traducendolo sul momento

COSA CI MANCA?



Istruzioni

Esecuzione

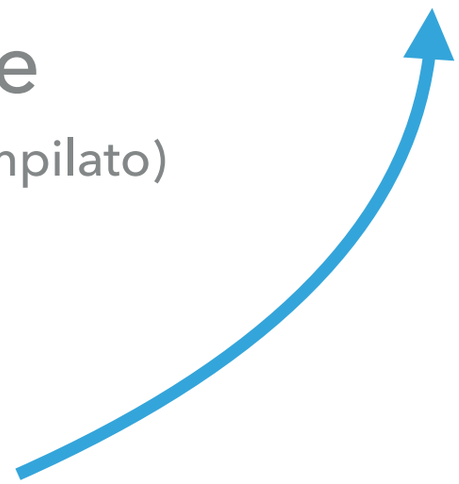
Forma dinamica:
Serve qualcuno/qualcosa
che esegua le istruzioni

FORMA DINAMICA (PROCESSO)

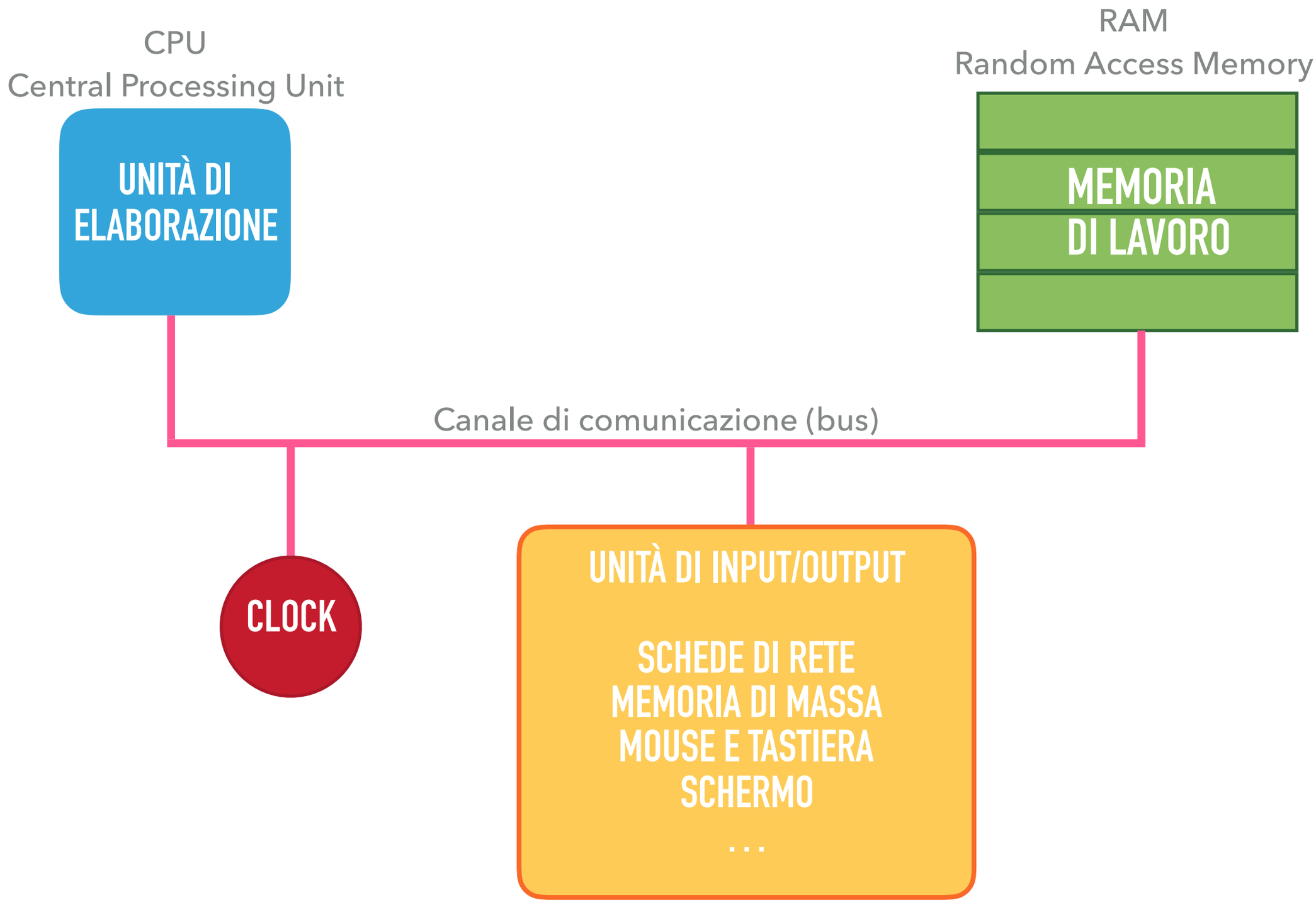
- Ci serve ricordare quale istruzione dobbiamo andare ad eseguire
- Dobbiamo tenere traccia delle risorse che ci servono: memoria, file aperti, etc.
- E, soprattutto, ci serve qualcuno/qualcosa in grado di eseguire le istruzioni

PROCESSO DI PROGRAMMAZIONE

- Enunciato del problema e requisiti della soluzione
- Ideazione di un algoritmo risolutivo
- Stesura del codice sorgente
- Compilazione
  (se il linguaggio è compilato)
- Esecuzione
- Debugging

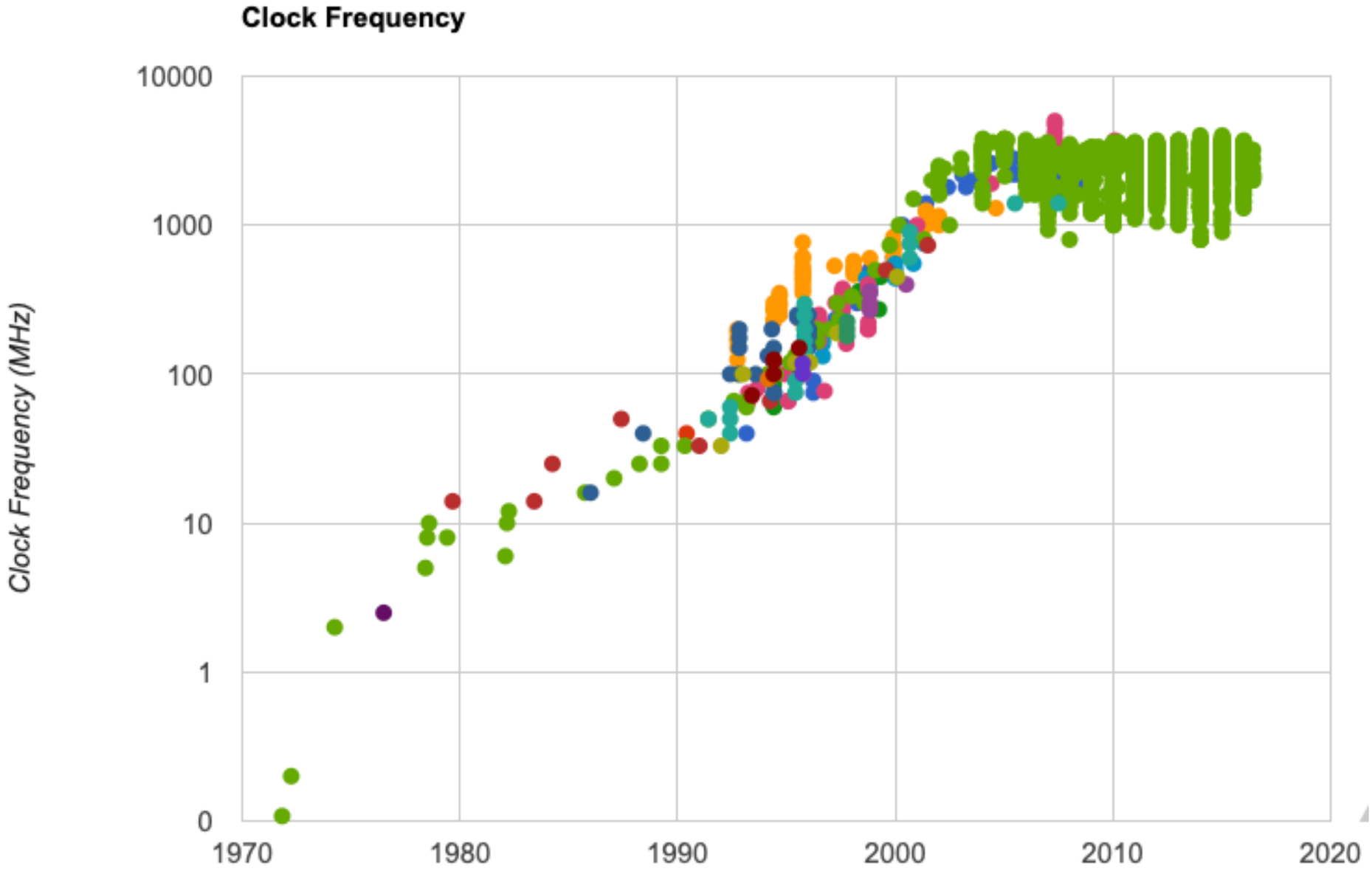


ARCHITETTURA DI VON NEUMANN



IL CLOCK

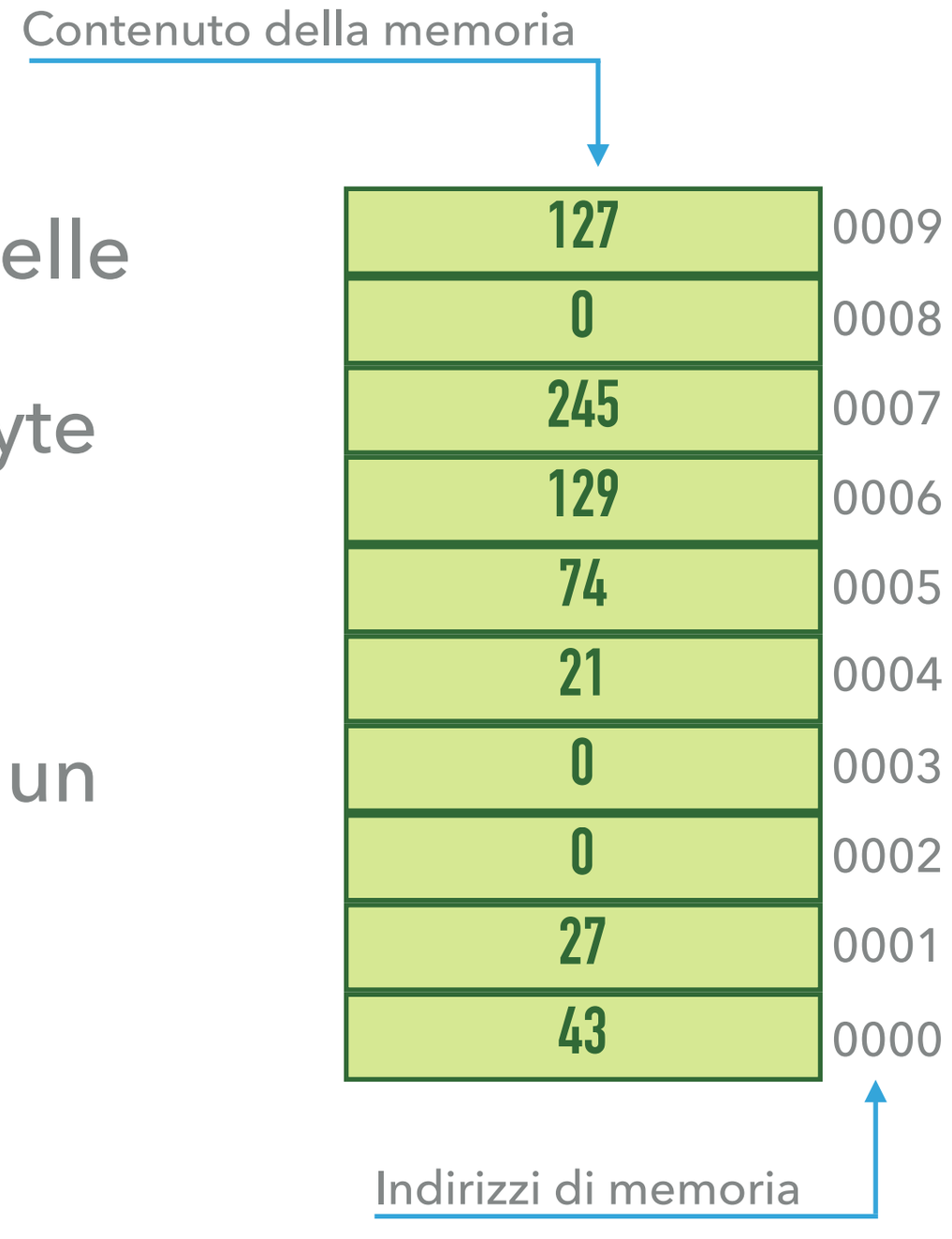
- Serve a permettere la sincronizzazione dei vari componenti



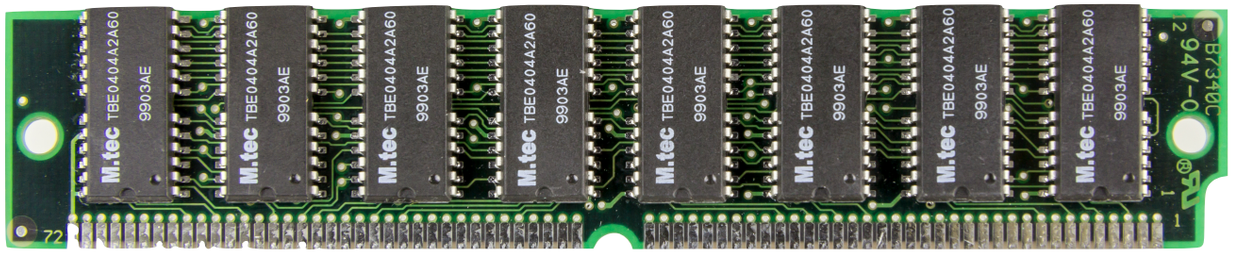
http://cpudb.stanford.edu/visualize/clock_frequency

LA MEMORIA DI LAVORO

- La memoria è organizzata in celle
- Ogni cella ha dimensione 1 byte
- Ogni cella ha un suo indirizzo
- È possibile leggere o scrivere un valore ad un dato indirizzo

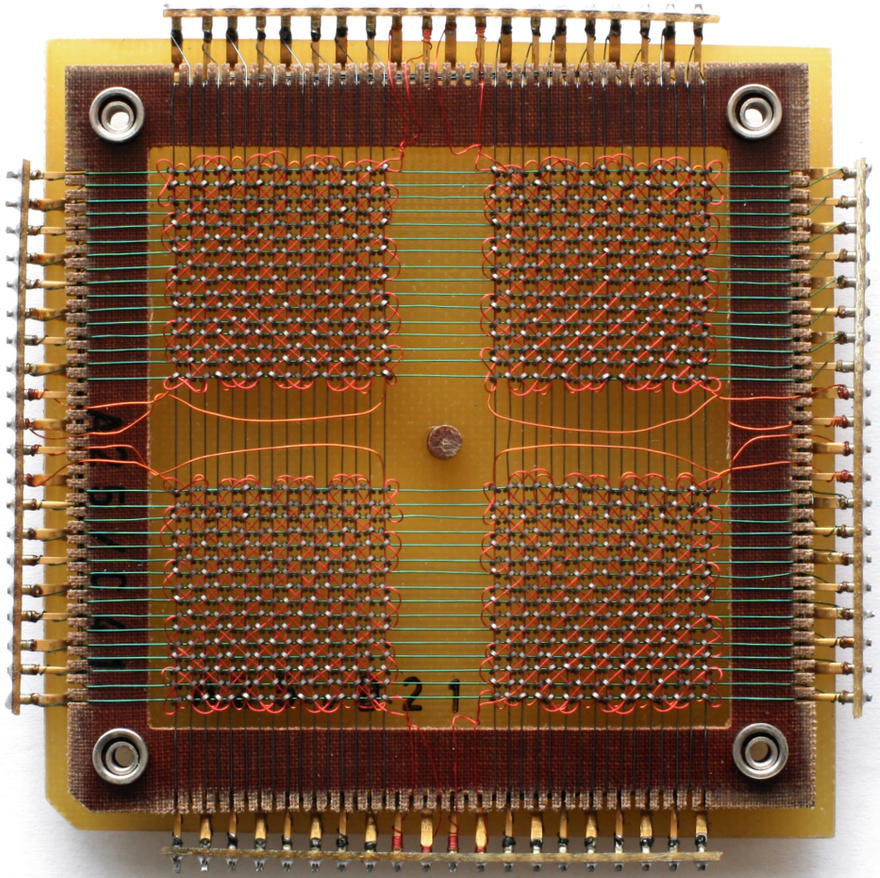


MEMORIA DI LAVORO: ESEMPI



Memoria basta sui semiconduttori

Usata ancora oggi, sono comuni diversi GB nei personal computer e nei dispositivi mobili



Memoria a core ferromagnetici

Nel 1967 il più potente supercomputer aveva meno di 4MB di memoria

PERIFERICHE DI I/O

- Permettono di comunicare con l'utente o con altri dispositivi
- Periferiche di input: tastiera, mouse,...
- Periferiche di output: schermo, altoparlanti, stampanti,...
- Memorie di massa

MEMORIA DI MASSA

- Per la memorizzazione a lungo termine di dati
  (e non serve che il computer rimanga acceso)
- Generalmente la lettura e la scrittura sono molto più lente
- Alcune ad accesso casuale:
  è possibile accedere ad ogni “celletta”
- Altre ad accesso sequenziale:
  per accedere alle “celletta” n bisogna attendere
  le n-1 “cellette” precedenti

MEMORIA DI MASSA: ESEMPI

Hard disk

Informazioni memorizzate nelle cariche magnetiche di piatti che girano a migliaia di giri al minuto

Capienza fino ad alcuni TB

Dischi a stato solido

Nessuna parte mobile, generalmente molto più veloci dei dischi magnetici, che stanno rimpiazzando

Generalmente di capienza inferiore ai dischi magnetici.
Comuni fino a diverse centinaia di GB

MEMORIA DI MASSA: ESEMPI

Dischi ottici (CD/DVD/Blue Ray)

Da diverse centinaia di MB a diverse decine di GB

Memorie USB

Stessa tecnologia degli SSD
(usata anche nelle microSD, etc)
da pochi GB a centinaia di GB

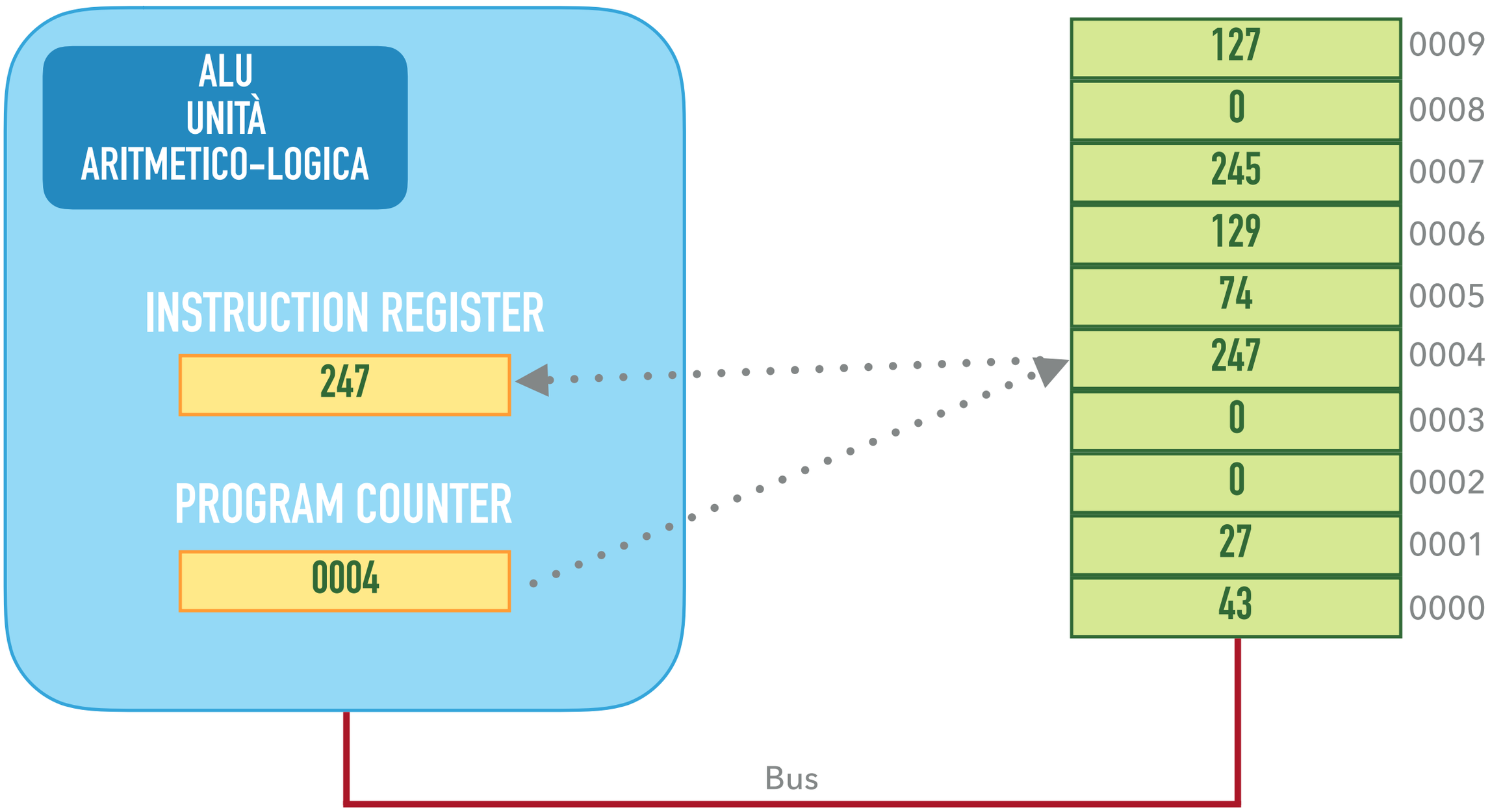
Floppy disk (8, 5.25 e 3.5 pollici)

Da centinaia di KB a oltre 1MB

LA CPU

- È la CPU che esegue le istruzioni
- Deve effettuare ripetutamente 3 operazioni
  - **Fetch**. Ottiene l'istruzione da eseguire dalla memoria
  - **Decode**. Interpreta che azioni l'istruzione dice di fare
  - **Execute**. Esegue effettivamente l'istruzione

FASE DI FETCH



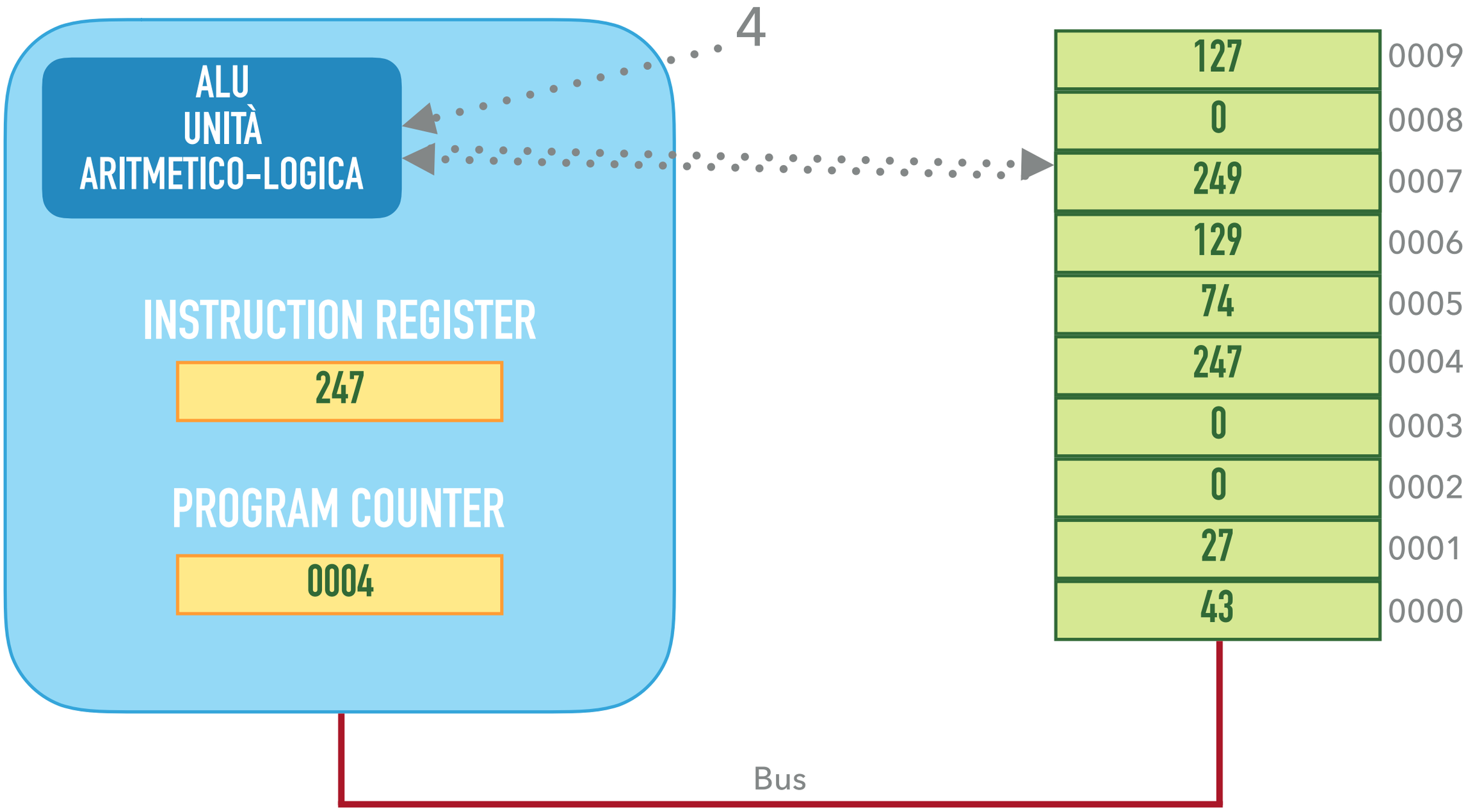
FASE DI FETCH

- Il **Program Counter** (PC) indica l'indirizzo in memoria in cui si trova la prossima istruzione da eseguire
- Il valore contenuto nella posizione in memoria indicata dal PC viene memorizzato nell'**Instruction Register**

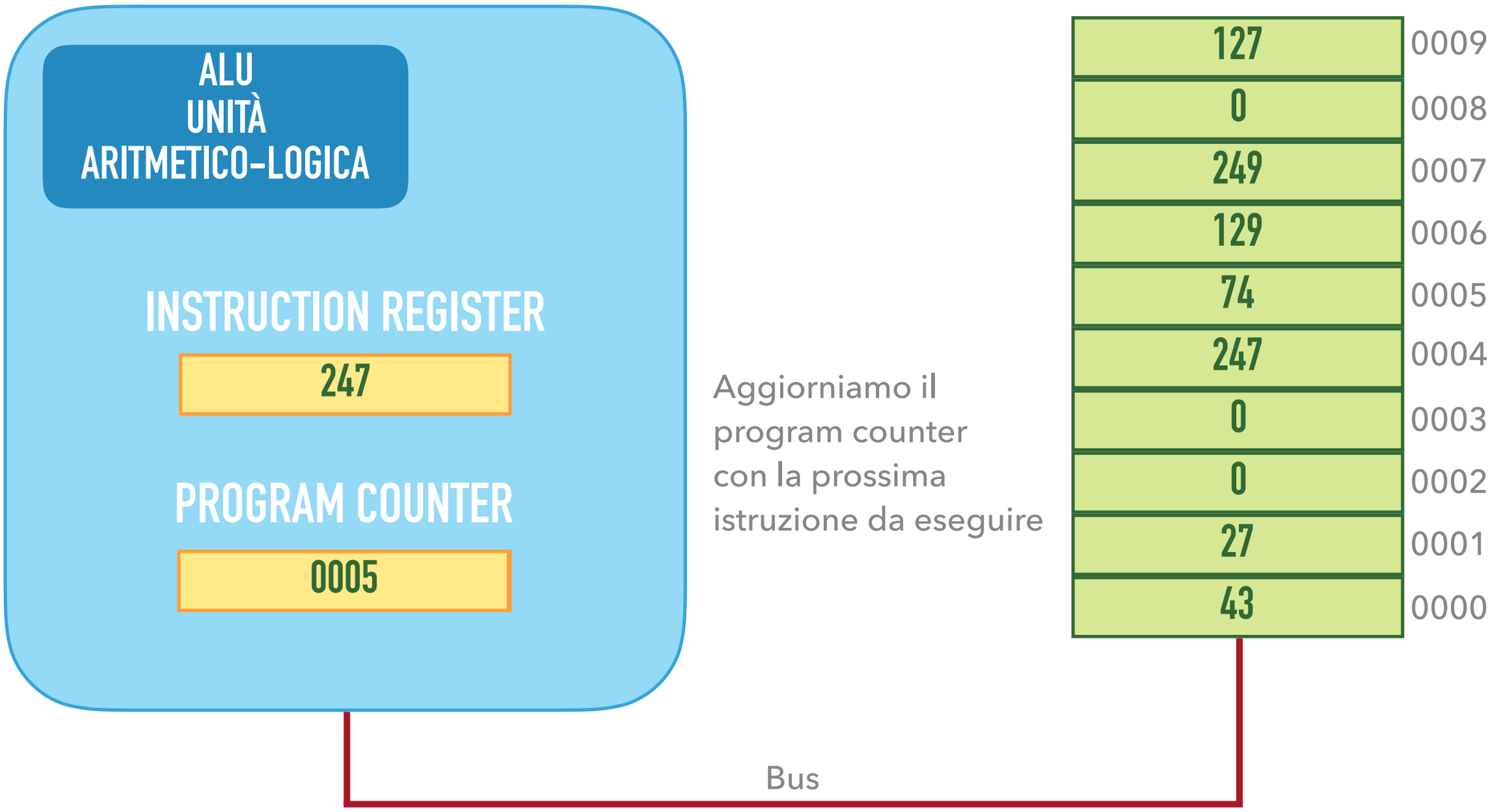
FASE DI DECODE

- Serve “interpretare” l’istruzione recuperata e attivare le parti del processore che servono
- 247 potrebbe significare
  “somma 4 al valore nella locazione di memoria 7”
- Ora l’istruzione deve essere spezzata in più parti:
  - Recupero del valore nella cella 7 dalla memoria
  - Somma di 4 al valore ottenuto
  - Scrittura del risultato nella cella di memoria 7

FASE DI EXECUTE



PRONTI PER RIPETERE IL CICLO



PROCESSORI: QUANTE ISTRUZIONI AL SECONDO?

	MIPS		Anno
CDC 6600	10	SUPERCOMPUTER	1965
CRAY 1	160	SUPERCOMPUTER	1975
MOS 6502 (@5.8MHz)	2,5		1981
Intel Pentium	188		1994
Intel Pentium III	2054		1999
Intel Core i7	238310		2014

CICLO DI ESECUZIONE DEL PROGRAMMA

- Caricare il programma in memoria
- Impostare il program counter alla prima istruzione del programma
- Far partire l'esecuzione del programma
- Ottenere i risultati

IL SISTEMA OPERATIVO

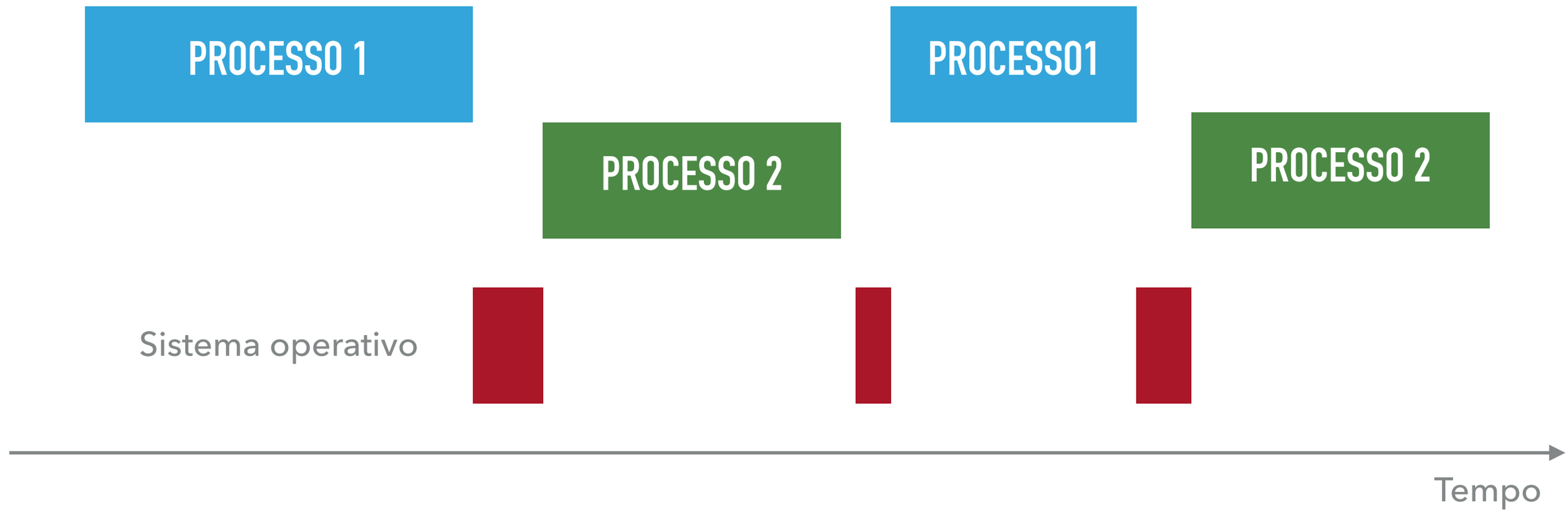
Il sistema operativo è un programma sempre attivo che regola:

- Gestione dei processi
- Gestione della memoria di lavoro
- Gestione delle operazioni di I/O
- Gestione del filesystem

ESEMPI DI SISTEMI OPERATIVI

- Microsoft Windows
- Sistemi UNIX e Unix-like
  - Linux
  - BSD: OpenBSD, FreeBSD, NetBSD
  - macOS (OS X e successivi)
- Altri di ricerca o per usi specifici:
  IBM Z/OS, OpenVMS, Singularity

GESTIONE DEI PROCESSI



Il sistema operativo alloca “slice” di tempo ai processi dando l’illusione che tutti eseguano contemporaneamente

GESTIONE DELLA MEMORIA DI LAVORO E DELL'I/O

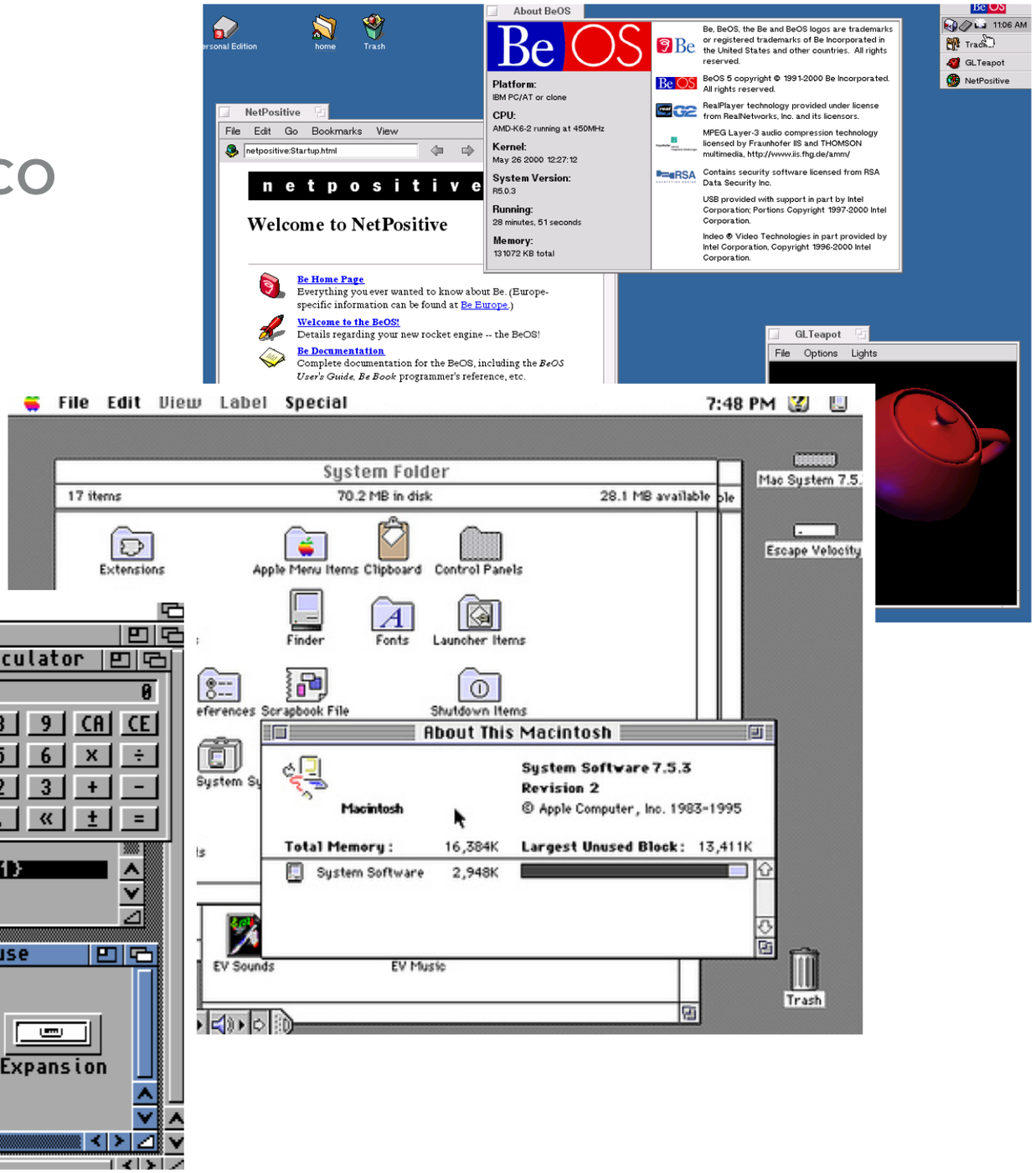
- Quando al programma serve della memoria, la deve richiedere al sistema operativo
- Il sistema operativo tiene la memoria di processi diversi separata
- Lo stesso per l'I/O: il sistema operativo interagisce direttamente con l'hardware e media tutti gli accessi

GESTIONE DELL'ACCESSO ALLA MEMORIA DI MASSA

- Generalmente non vogliamo che i programmi vedano direttamente la memoria di massa
- Il sistema operativo crea una astrazione (il **file**) che viene visto come usa sequenza di byte a cui è dato un nome
- Il sistema operativo permette di organizzare i file in una struttura gerarchica tramite le **directory**, che a loro volta possono contenere file o altre directory

TIPI DI INTERAZIONE

- Tramite ambiente grafico
- Da linea di comando



LA LINEA DI COMANDO

- Interazione tramite comandi testuali
- Si dà un comando ed il sistema risponde
  - Il sistema mostra un **prompt** per indicare che è pronto a ricevere comandi
- Nei sistemi Windows è detta **prompt dei comandi**
- Nei sistemi UNIX e Unix-like è detta **shell**

ESEMPIO DI INTERAZIONE


```
~% cd /usr/local

/usr/local% pwd

/usr/local

/usr/local% ls

include         texlive         bin             lib             sbin
var             etc             opt             share

/usr/local% cd ..

/usr%
```