Architetture Degli Elaboratori

Lezione 1.a
Struttura di un calcolatore

Alessandro Fabris

big thanks to Luca Manzoni for most course materials

Informazioni sul corso

Aspetti pratici

- Slide, codice e **materiali** caricati su **Moodle** prima della lezione
  - Moodle: ARCHITETTURE DEGLI ELABORATORI 2025
  - Pw: archi2025
- **Comunicazioni**, lezioni e registrazioni su **Teams**
- Come **aiutare**
  - In presenza: partecipare, chiedere chiarimenti, rispondere, buttarsi
  - Da remoto: dire se audio ok, schermo condiviso, registrazione attiva
- Lezioni: mar 2-4.30
  - Teoria + pratica (6): 7 Ott – 11 Nov
  - Esercitazioni (2): 18 Nov – 25 Nov

Informazioni sul corso

Esami

- L'esame della parte di architetture (3CFU) consiste di uno **scritto**
  - Bozza appelli prima sessione: 13 Gen, 30 Gen, 20 Feb (h15)
- Struttura dello scritto:
  - Domande a risposta multipla…
  - …ma con giustificazione (necessaria perché la risposta sia considerata valida)
  - Le domande possono richiedere lo svolgimento di esercizi, non sono solo di teoria
- Prova di esame nelle ultime lezioni
- Il corso è costruito in modo che ogni parte si appoggi sulle parti precedenti: fate esercizi e studiate durante tutto il corso!

Cosa

Architettura degli elaboratori:

Architettura del set di istruzioni (ISA): quali sono le operazioni basilari che il software può far fare all'hardware

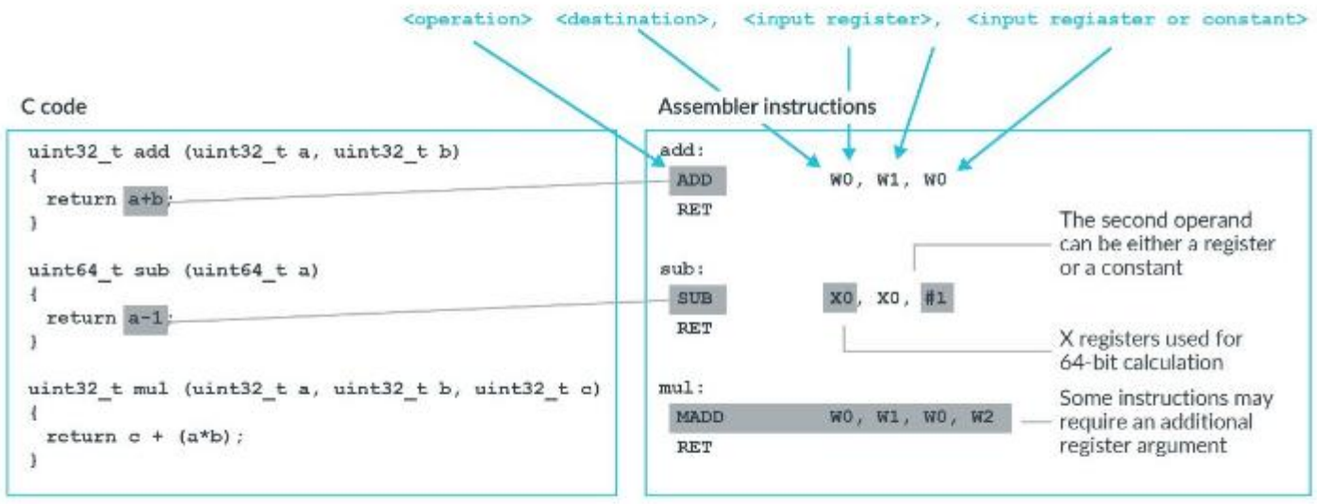
Microarchitettura (μarch): descrive componenti hardware impiegate e loro interconnesioni che insieme **implementano** un set di istruzioni

Cosa

7 Data processing

7.1 Arithmetic and logic operations

The basic format of logical and integer arithmetic instructions is:



The parts of the instruction are as follows:

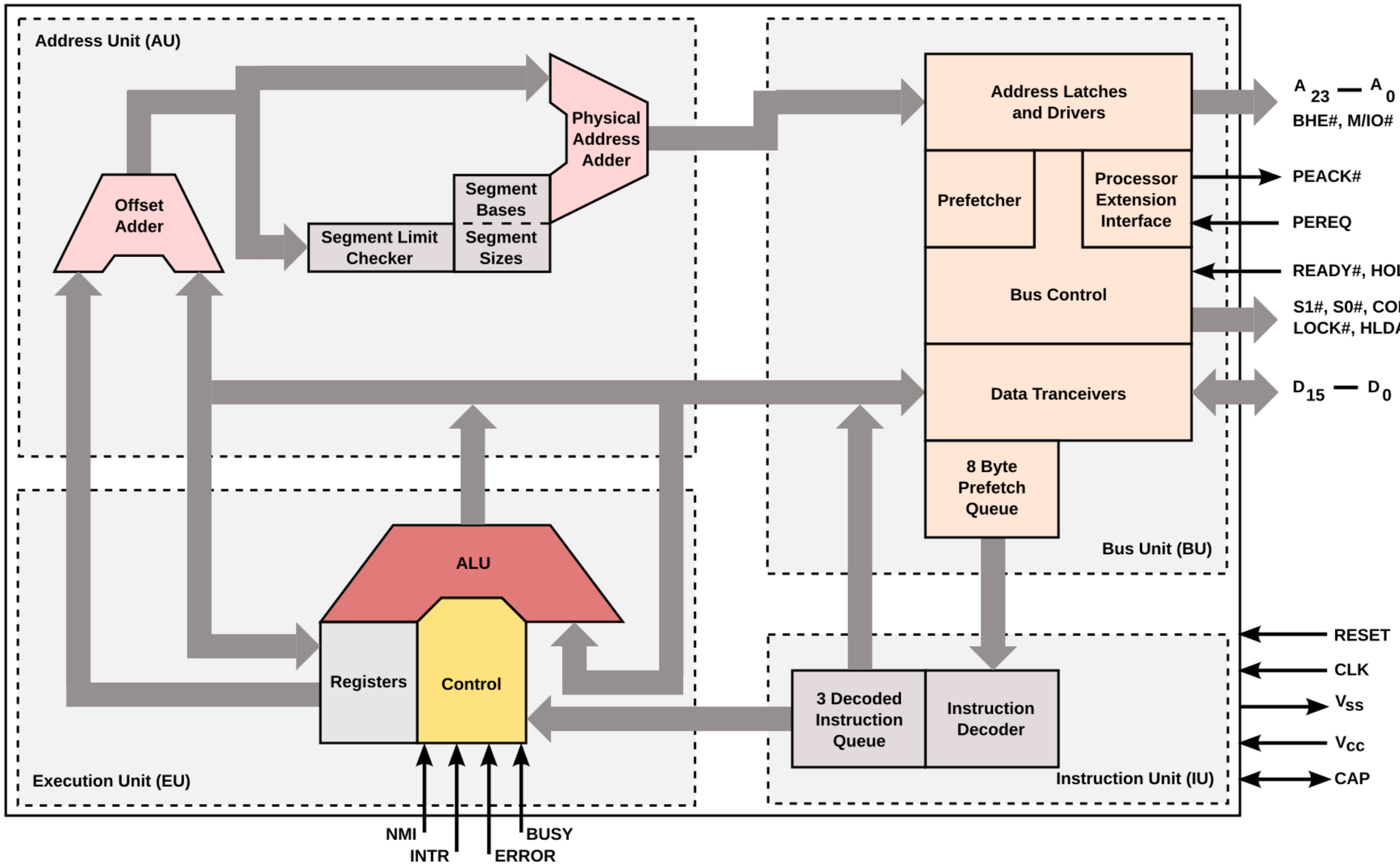
- Operation. This defines what the instruction does. For example, `ADD` does addition and `AND` performs a logical AND.

  An `S` can be added to the operation to set flags. For example, `ADD` becomes `ADDS`. This `S` tells the processor to update the `ALU` flags based on the result of instruction. We discuss `ALU` flags in the section on generating condition code.

D: ISA o μarch?

Cosa

Intel 80286 architecture



D: ISA o μarch?

Ma perché?

💡 1. Capire come funziona un computer ‘sotto il cofano’

🛠️ 2. Scrivere software più efficiente

🧠 3. Migliorare il pensiero logico e sistemico

🔐 4. Capire la sicurezza e l'affidabilità

Guardare dentro la scatola

Come fa un computer a eseguire un programma?

Il vostro codice

```
#include <stdio.h>

int main()
{
   printf("Hello World\n");
   return 0;
}
```

Some magic here

Vogliamo scoprire quello che accade
tra quando scriviamo codice
e quando il codice viene eseguito

Image from: https://commons.wikimedia.org/wiki/File:Iie-system.jpg

Il processore non conosce il C

E altri linguaggi di alto livello

- Come vi ricorderete, il linguaggio C viene compilato.
- Il computer non può eseguire direttamente il **codice C**, deve essere prima **trasformato** in qualcosa che può essere **compreso dal processore**
- Le **istruzioni** eseguibili direttamente da un processore sono molto limitate e dipendono dall'**architettura** del processore (i.e., il linguaggio "parlato" dal processore)
- Vediamo tutti i passi che portano da un pezzo di codice C a istruzioni eseguibili dal processore

Da C ad Assembly

Un primo passaggio intermedio

```
#include <stdio.h>

int main()
{
    printf("Hello World\n");
    return 0;
}
```

```
.LC0:
        .string "Hello World!"
main:
        stp  x29, x30, [sp, -16]!
        mov  x29, sp
        adrp  x0, .LC0
        add  x0, x0, :lo12:.LC0
        bl  puts
        mov  w0, 0
        ldp  x29, x30, [sp], 16
        ret
```

Questa trasformazione è svolta dal compilatore in modo automatico.

Il codice assembly ottenuto dipende dal compilatore e dall'**architettura** del processore

Assembly per ARM64, compilato con gcc 10.2

Da C ad Assembly

Un primo passaggio intermedio

```
#include <stdio.h>

int main()
{
   printf(“Hello World\n”);
   return 0;
}
```

Notate come le istruzioni siano diverse a seconda dell’architettura

```
.LC0:
        .string "Hello World!"
main:
        push  rbp
        mov  rbp, rsp
        mov  edi, OFFSET FLAT:.LC0
        call  puts
        mov  eax, 0
        pop  rbp
        ret
```

Assembly per X86-64, compilato con gcc 10.2

```
.LC0:
        .string "Hello World!"
main:
        addi  sp,sp,-16
        sw  ra,12(sp)
        sw  s0,8(sp)
        addi  s0,sp,16
        lui  a5,%hi(.LC0)
        addi  a0,a5,%lo(.LC0)
        call  puts
        li  a5,0
        mv  a0,a5
        lw  ra,12(sp)
        lw  s0,8(sp)
        addi  sp,sp,16
        jr  ra
```

Assembly per RISC-V, compilato con gcc 10.2

```
.LC0:
        .string "Hello World!"
main:
        stp  x29, x30, [sp, -16]!
        mov  x29, sp
        adrp  x0, .LC0
        add  x0, x0, :lo12:.LC0
        bl  puts
        mov  w0, 0
        ldp  x29, x30, [sp], 16
        ret
```

Assembly per ARM64, compilato con gcc 10.2

Assembly

Vicino al linguaggio macchina

D: questo è linguaggio macchina?

add r1, r1, #3

D: e questo?

00011010 10101101 01101010 10101101

Assembly

Vicino al linguaggio macchina

- Il codice **assembly** rappresenta una **forma "leggibile"** e poco astratta delle **operazioni** effettivamente comprese dal **processore**
- L'**assembler** trasforma il codice assembly in **codice macchina**
- Il lavoro dell'assembler è più semplice di quello del compilatore, dato che molte delle istruzioni in assembly sono semplicemente una versione "leggibile" di una corrispondente istruzione in codice macchina
- Architetture di processori diverse hanno istruzioni diverse, quindi il codice **assembly non è portabile**
- Possiamo esplorare questo processo di conversione su https://godbolt.org

Da assembly a codice macchina

Quasi alla fine…

```
.LC0:
        .string "Hello World!"
main:
        push  rbp
        mov  rbp, rsp
        mov  edi, OFFSET FLAT:.LC0
        call  puts
        mov  eax, 0
        pop  rbp
        ret
```

```
cffa edfe 0700 0001 0300 0000 0200 0000
1000 0000 5805 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 0040 0000 0000 0000
0000 0000 0000 0000 0040 0000 0000 0000
0500 0000 0500 0000 0500 0000 0000 0000
5f5f 7465 7874 0000 0000 0000 0000 0000
5f5f 5445 5854 0000 0000 0000 0000 0000
```

- **L’assembler converte** il codice **assembly** in **codice macchina**
- Il linker aggiungerà le librerie necessarie
- Il formato esatto dipende anche dal sistema operativo utilizzato

Eseguire il codice

Dal codice macchina all'esecuzione

- Il **codice compilato** è una forma **statica** del nostro programma
- Il programma deve ora essere eseguito
- Abbiamo convertito il codice C in codice macchina…
- …ma come fa il processore a eseguire le istruzioni?
- Dove sono memorizzate le istruzioni?
- Come facciamo a interagire con lo schermo?

Architettura del calcolatore

Struttura di base del calcolatore

E l’architettura di Von Neumann

Nella sua forma “di base” un computer può essere astratto in 4 componenti

CPU

Una **CPU** o unità centrale di elaborazioni che esegue le istruzioni

Memoria primaria

Una **memoria primaria** che contiene sia le istruzioni che i dati su cui esse lavorano

Un **Bus** per permettere la comunicazione tra le diverse component → collo di botiglia?

Periferiche

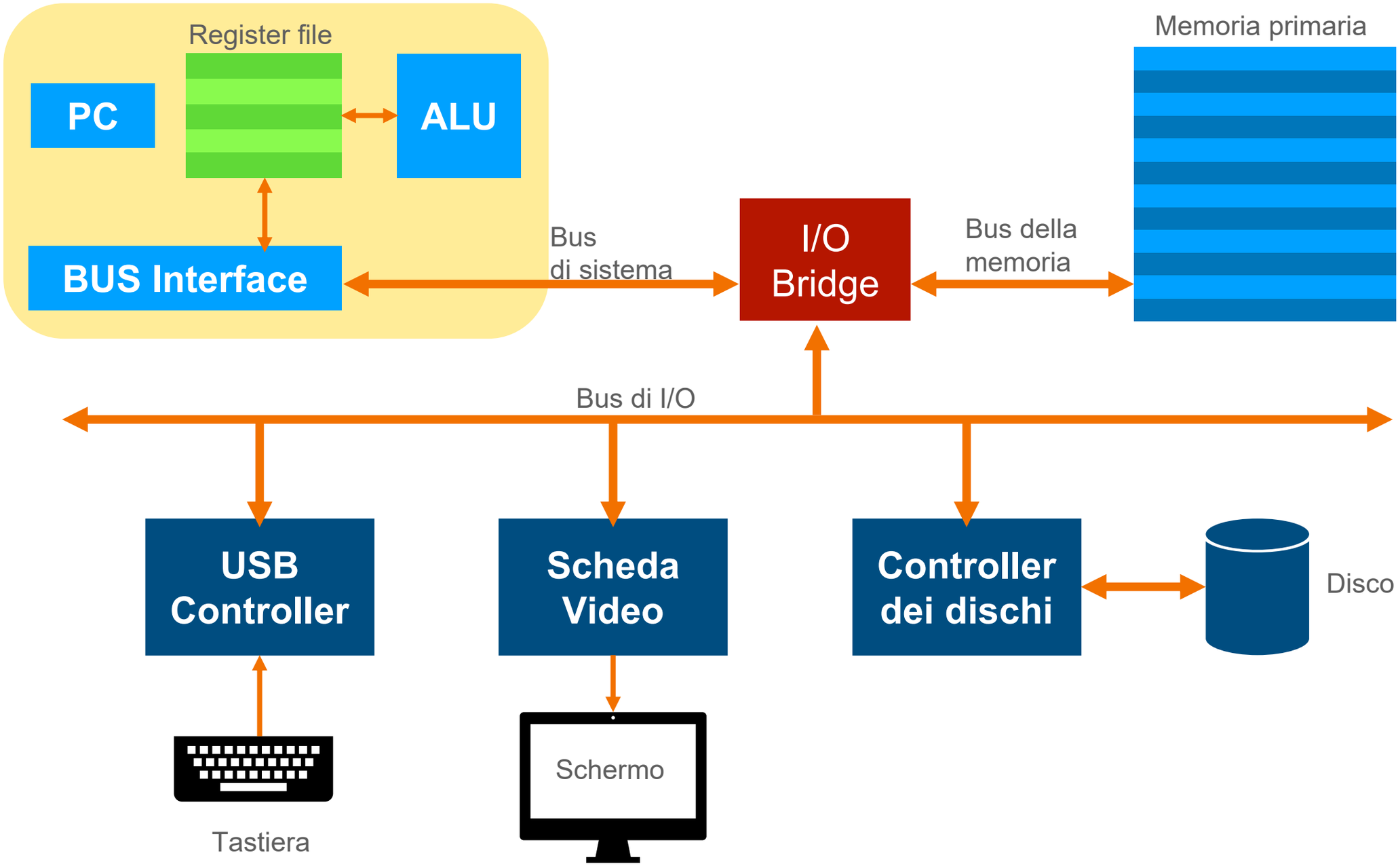
Un insieme di **periferiche**, che vanno dalla memoria di massa (dischi fissi, SSD), ai dispositivi di input (tastiere, mouse), ai dispostivi di output (monitor, stampanti, etc).

Caratteristica essenziale dell’architettura di Von Neumann: i dati e le istruzioni risiedono nella stessa memoria.
In altre architetture questo potrebbe essere diverso (e.g., nell’architettura Harvard ci sono due memorie, una per i dati e una per le istruzioni)

Non tutto però è così semplice…

Struttura di base del calcolatore

E l'architettura di Von Neumann



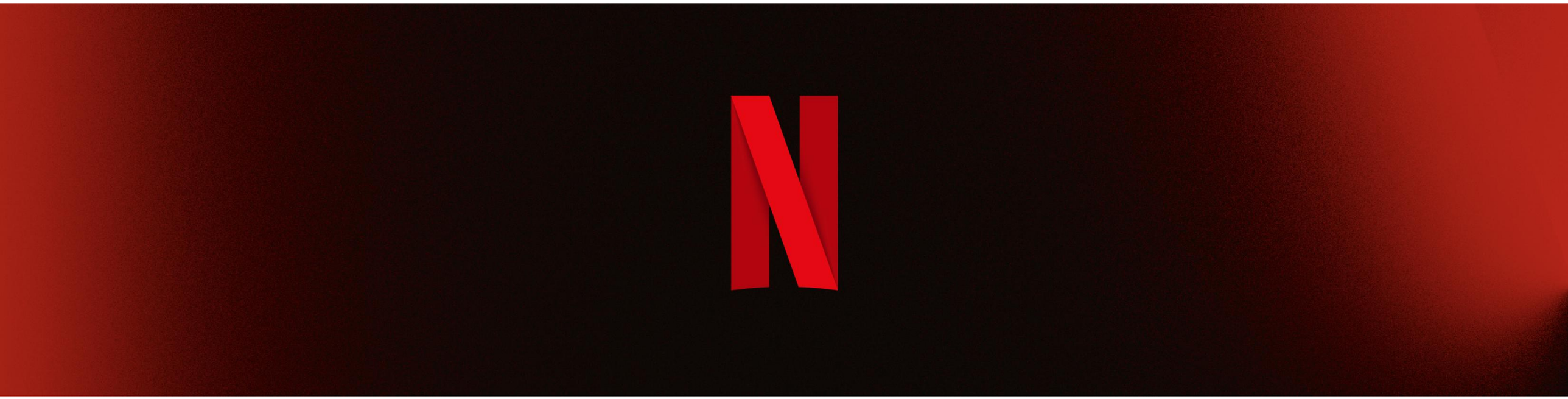
Dispositivi di I/O

- Permettono di comunicare con l'utente o con altri dispositivi
- Ci sono periferiche di **input**: mouse, tastiera, …
- Periferiche di **output**: schermo, stampante, …
- **Memorie di massa**
  - Utilizzate per memorizzare l'informazione a **lungo termine** (anche a dispositivo spento), dato che la memoria principale è solitamente volatile e meno capiente
  - Generalmente lettura e scrittura sono **più lente** della memoria principale
  - Possono essere distinte in **accesso sequenziale** (tutti i dati devono essere letti in sequenza) e **accesso casuale** (possiamo scegliere a quale dato accedere direttamente)

I/O

D: accesso tipico sequenziale o casuale?

I/O



D: accesso tipico sequenziale o casuale?

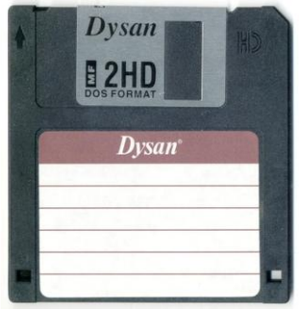
Dispositivi di storage

Alcuni 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.

Floppy Disk
Ormai obsoleti, capienza fino a oltre 1MB



Dischi ottici
Capienza da alcune centinaia di MB a decine di GB

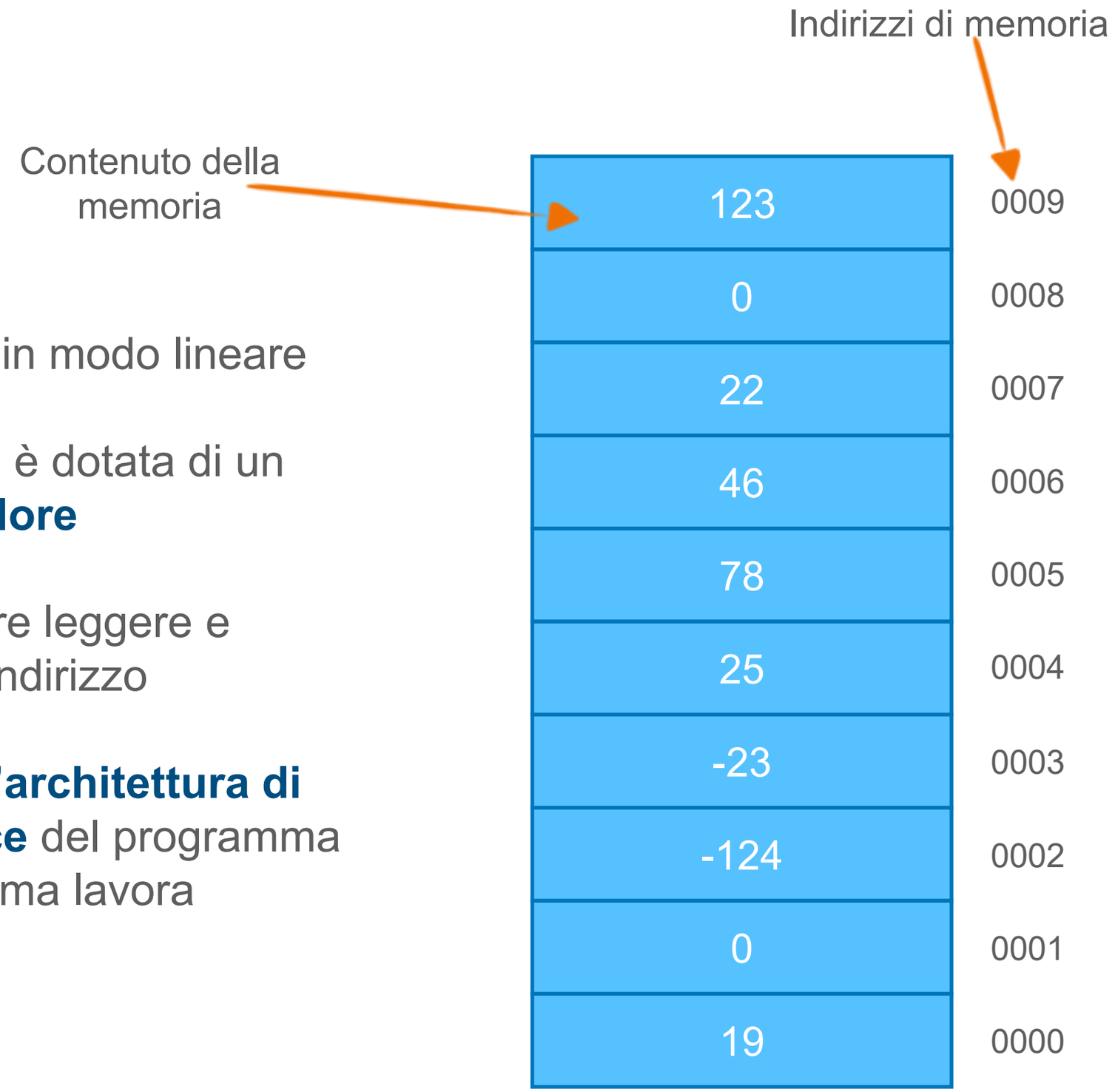
La memoria principale

La memoria è organizzata in modo lineare

Ogni locazione di memoria è dotata di un **indirizzo** e contiene un **valore**

È possibile per il processore leggere e scrivere valori ad un dato indirizzo

La memoria contiene, **nell'architettura di Von Neumann** sia il **codice** del programma che i **dati** su cui il programma lavora



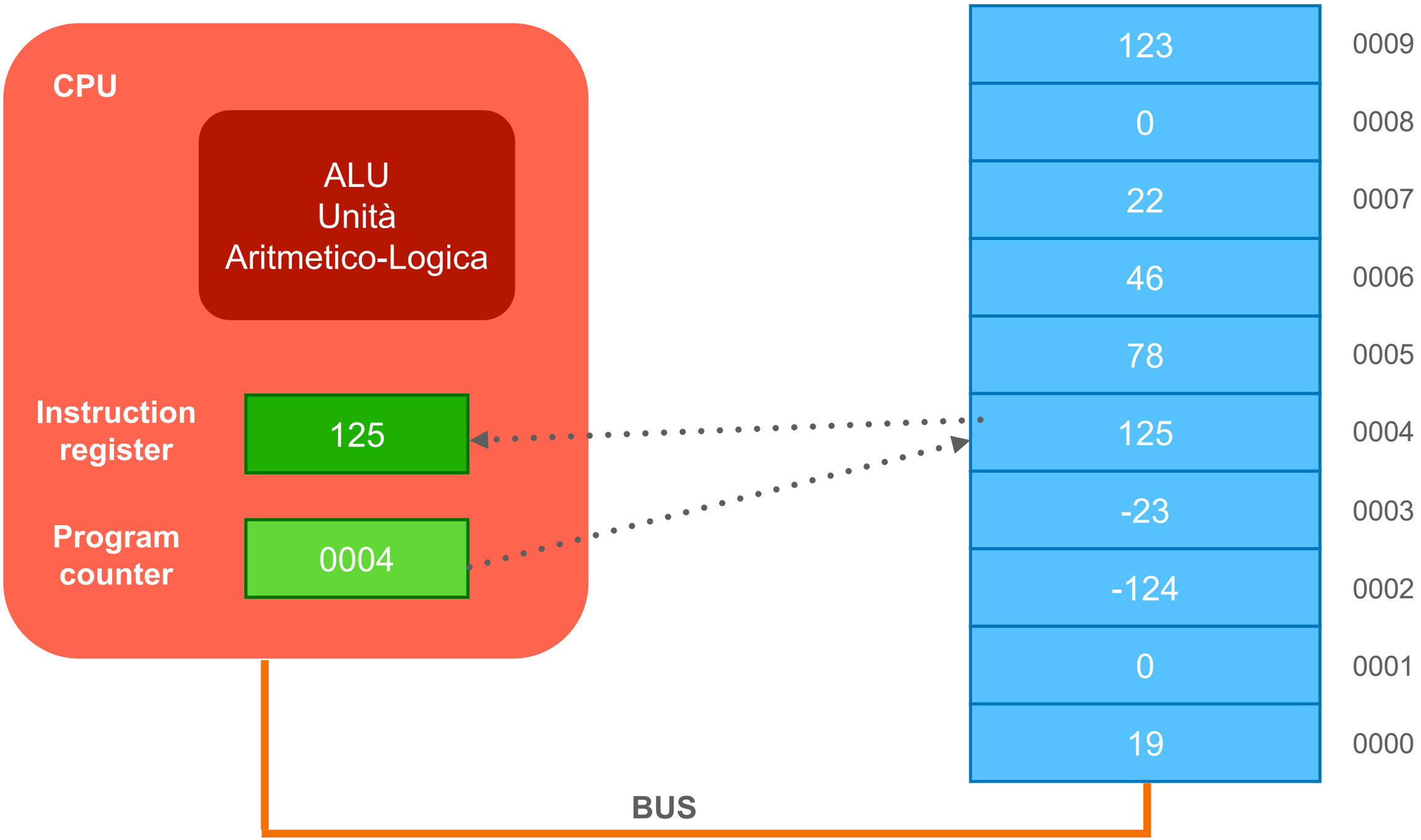
LA CPU

Ciclo di fetch-decode-execute

- È la CPU che esegue le istruzioni
- Effettua ripetutamente tre operazioni
  - **Fetch**: recupero della prossima istruzione da eseguire
  - **Decode**: decodifica dell'istruzione recuperata
  - **Execute**: esecuzione dell'istruzione decodificata
- Vediamo un esempio semplificato di come funzionano queste operazioni

Fetch

Ottenere una istruzione



Fetch

Ottenere una istruzione

- Il **program counter** contiene l'indirizzo della prossima istruzione da eseguire
- L'istruzione da eseguire viene ottenuta dalla memoria…
- …e salvata nell'**instruction register**
- È ora necessario decodificare l'istruzione nella fase di **decode** per "attivare" le parti del processore che possono effettivamente eseguirla (e recuperare i dati su cui l'istruzione deve agire)

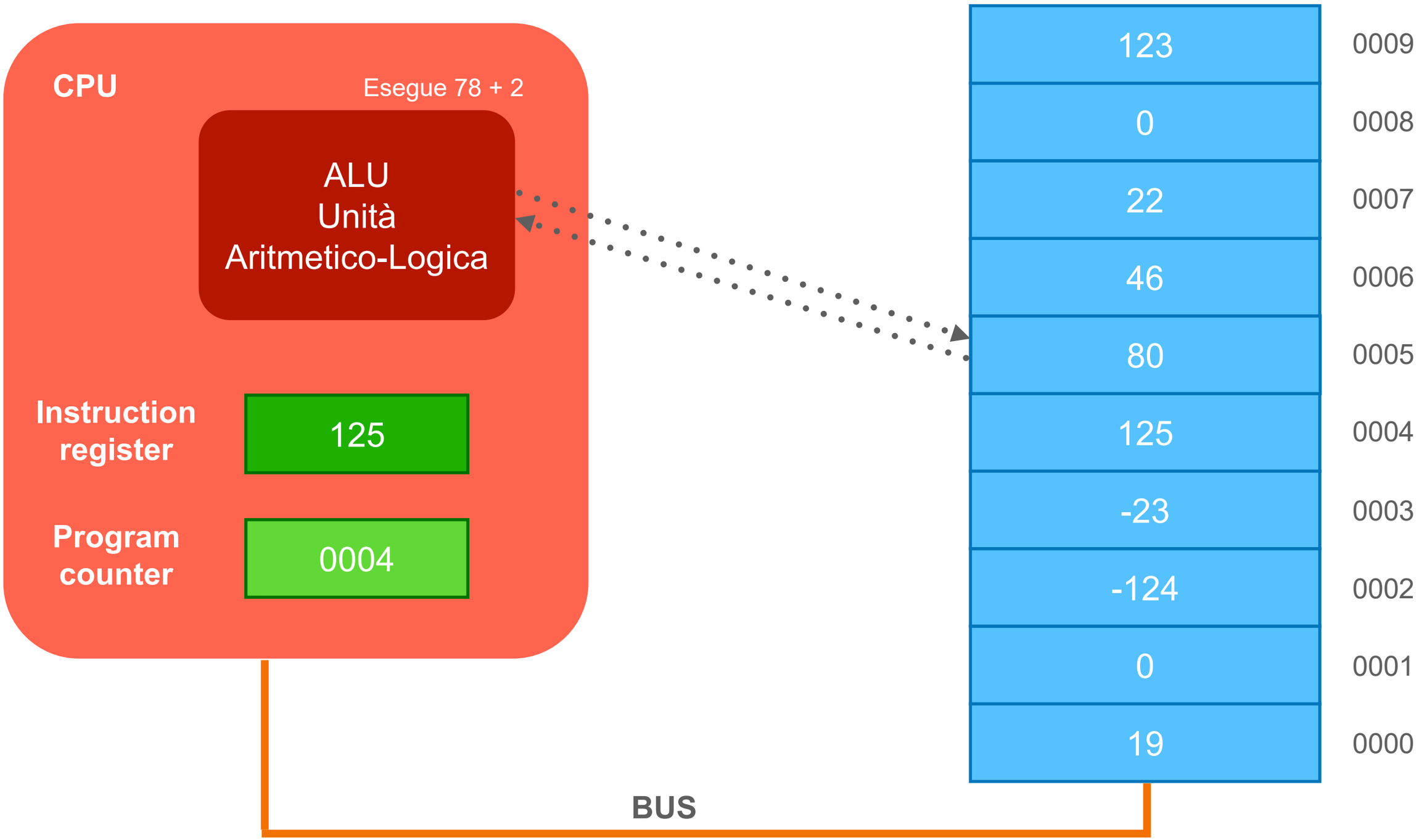
Decode

Cosa chiede l'istruzione?

- Il valore "125" salvato nell'istruction register deve essere interpretato come istruzione
- Potrebbe significare, per fare un esempio, che è necessario sommare 2 al valore contenuto nella locazione di memoria 5 e salvare il contenuto nella stessa locazione di memoria
- Ovviamente il significato dipende dall'architettura del processore!
- Siamo ora pronti a eseguire l'operazione richiesta

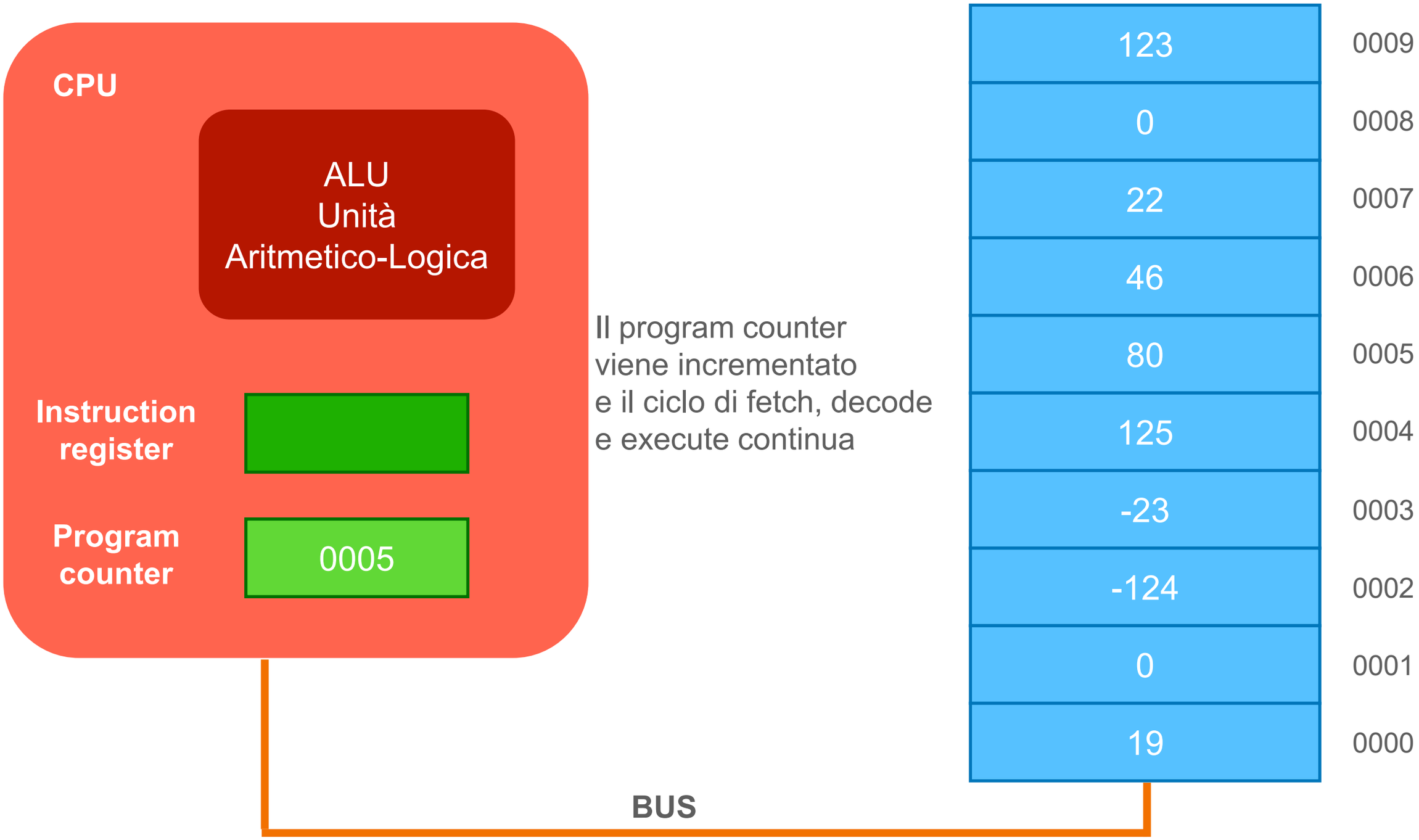
Execute

Eseguire l’operazione



Pronti a ripetere il ciclo?

Passare all'istruzione successiva



Struttura del corso

Struttura del corso

Architettura dei calcolatori

- Porte logiche
- Rappresentazione di numeri
- Logica combinatoria logica sequenziale
- L’unità aritmetico-logica
- Struttura di controllo e codice macchina

Struttura del corso

Architettura dei calcolatori

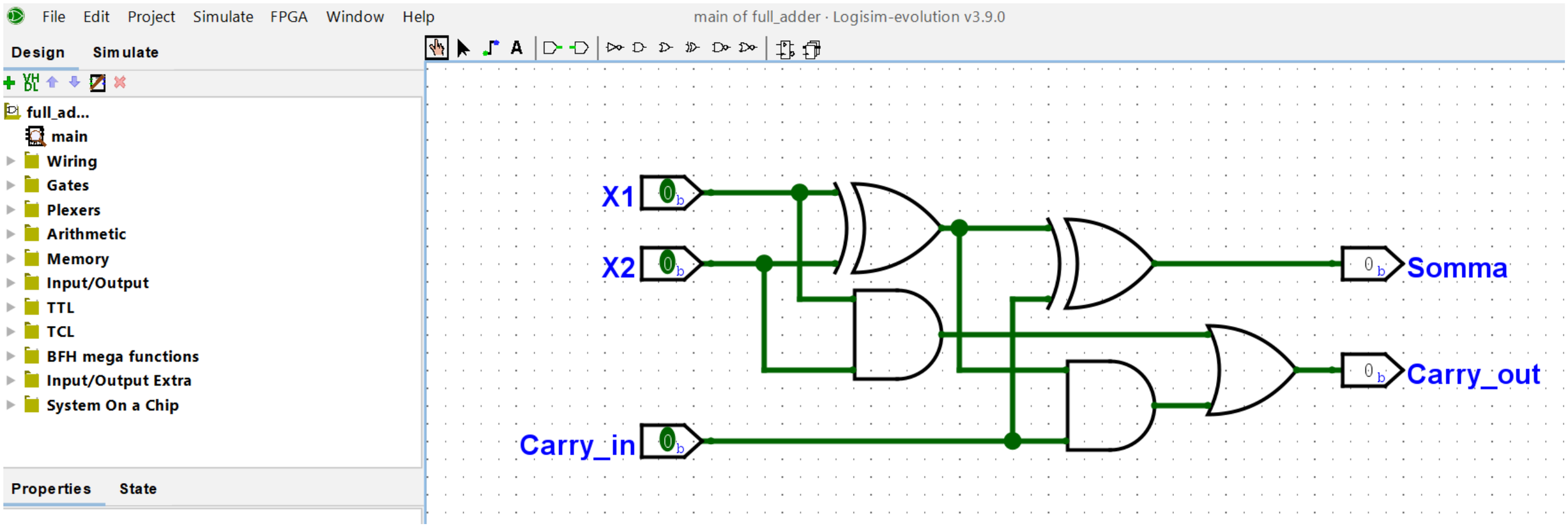
- Accesso ai dispositivi di I/O
- Assembly per ARM
- Cache
- Memoria virtuale

Tool

Architettura dei calcolatori

Logisim evolution

- https://github.com/logisim-evolution/logisim-evolution/releases



Tool

Architettura dei calcolatori

Simulatore processore ARM

- https://peterhigginson.co.uk/ARMlite/