Programmazione e Architetture (Modulo B)

Lezione 1
Struttura di un calcolatore

Luca Manzoni

Informazioni sul corso

Esami

- L’esame consiste di due parti:
  - Esame scritto
  - Esame orale
- Il superamento dello scritto è prerequisito per accedere all’orale
- Il superamento dello scritto permette di svolgere l’orale nello stesso appello
- Il corso è costruito in modo che ogni parte si appoggi sulle parti precedenti: fate esercizi e studiate durante tutto il corso!

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:lie-system.jpg

Il computer 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
- 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 computer

Da C ad Assembly

Un primo passaggio intermedio

```c
#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

Notate come le istruzioni siano diverse a seconda dell'architettura

```c
#include <stdio.h>

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

```
.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:
    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:
    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

- 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 di 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 componenti

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)

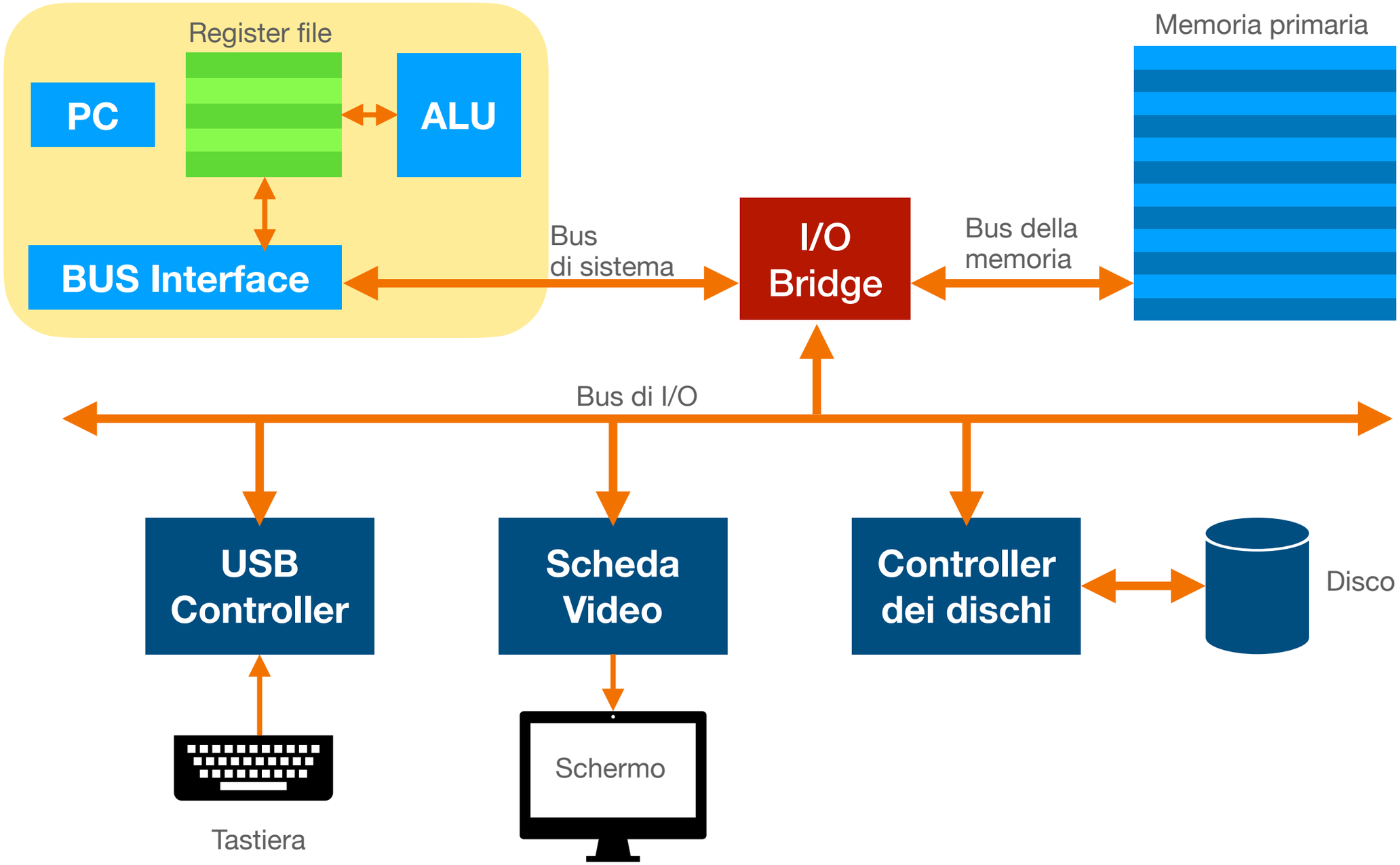
Periferiche

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).

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
  - 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)

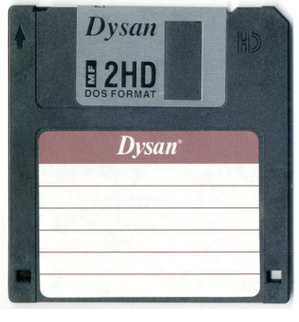
Dispositivi di storage

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

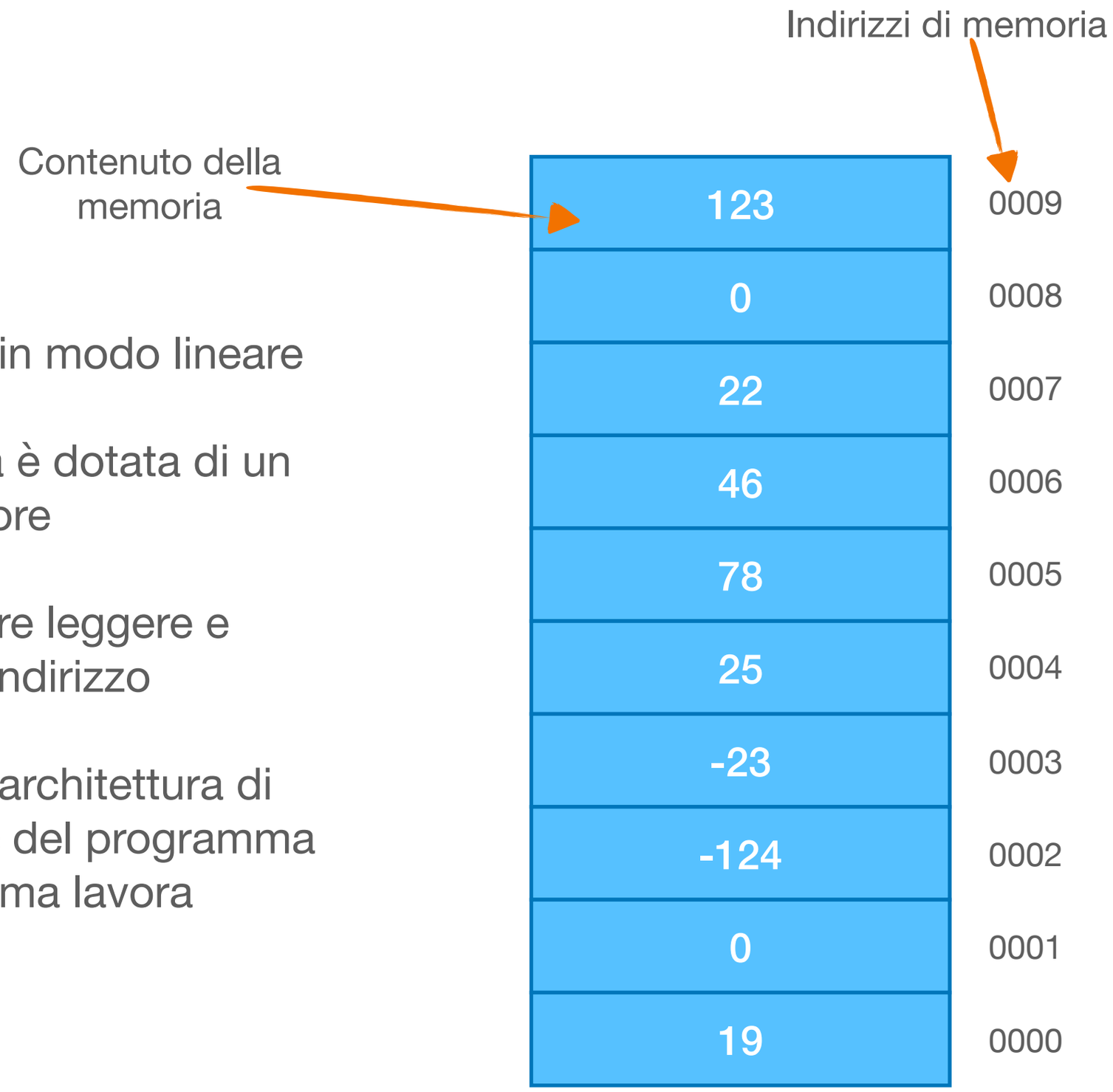
La memoria principale

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



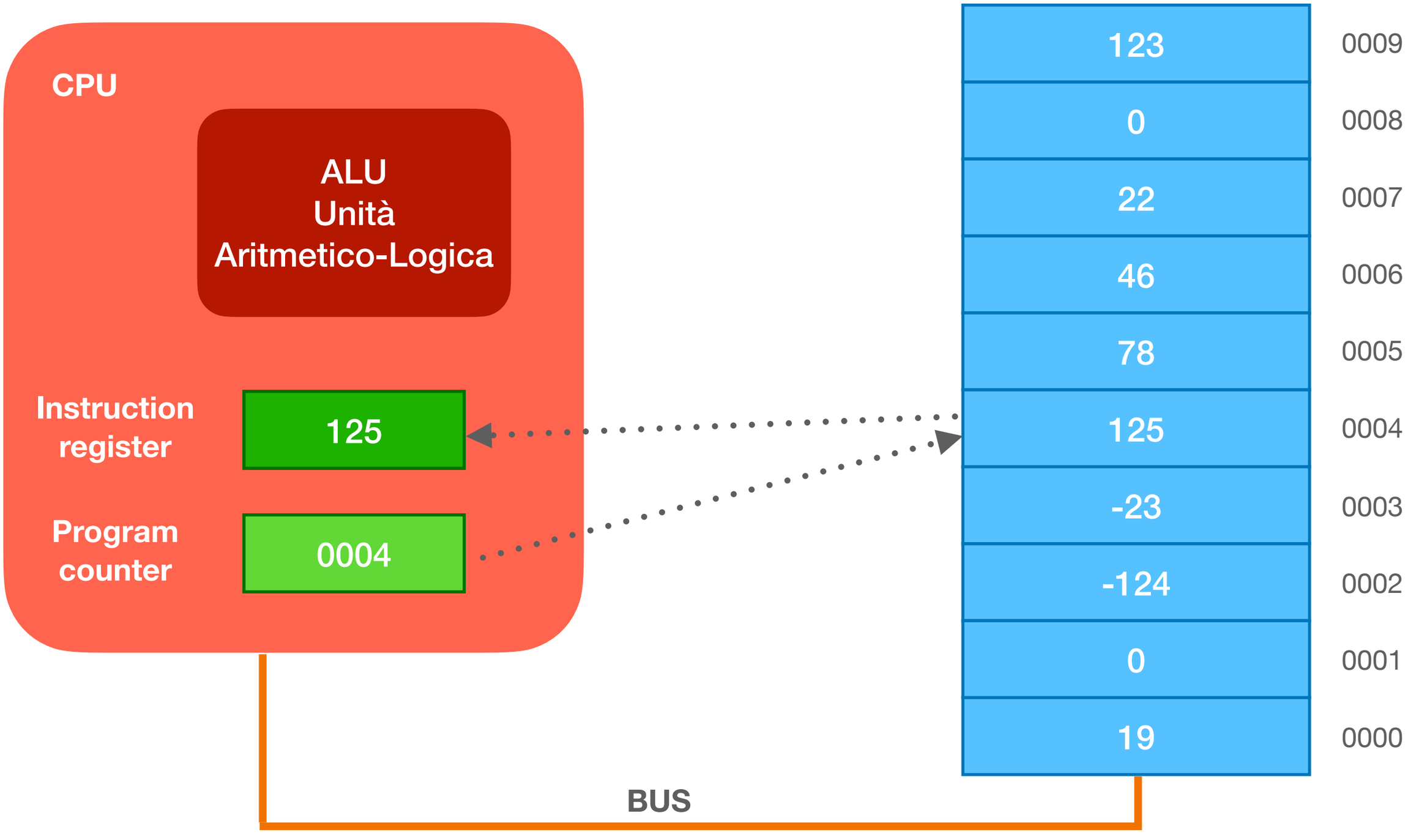
LA CPU

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)

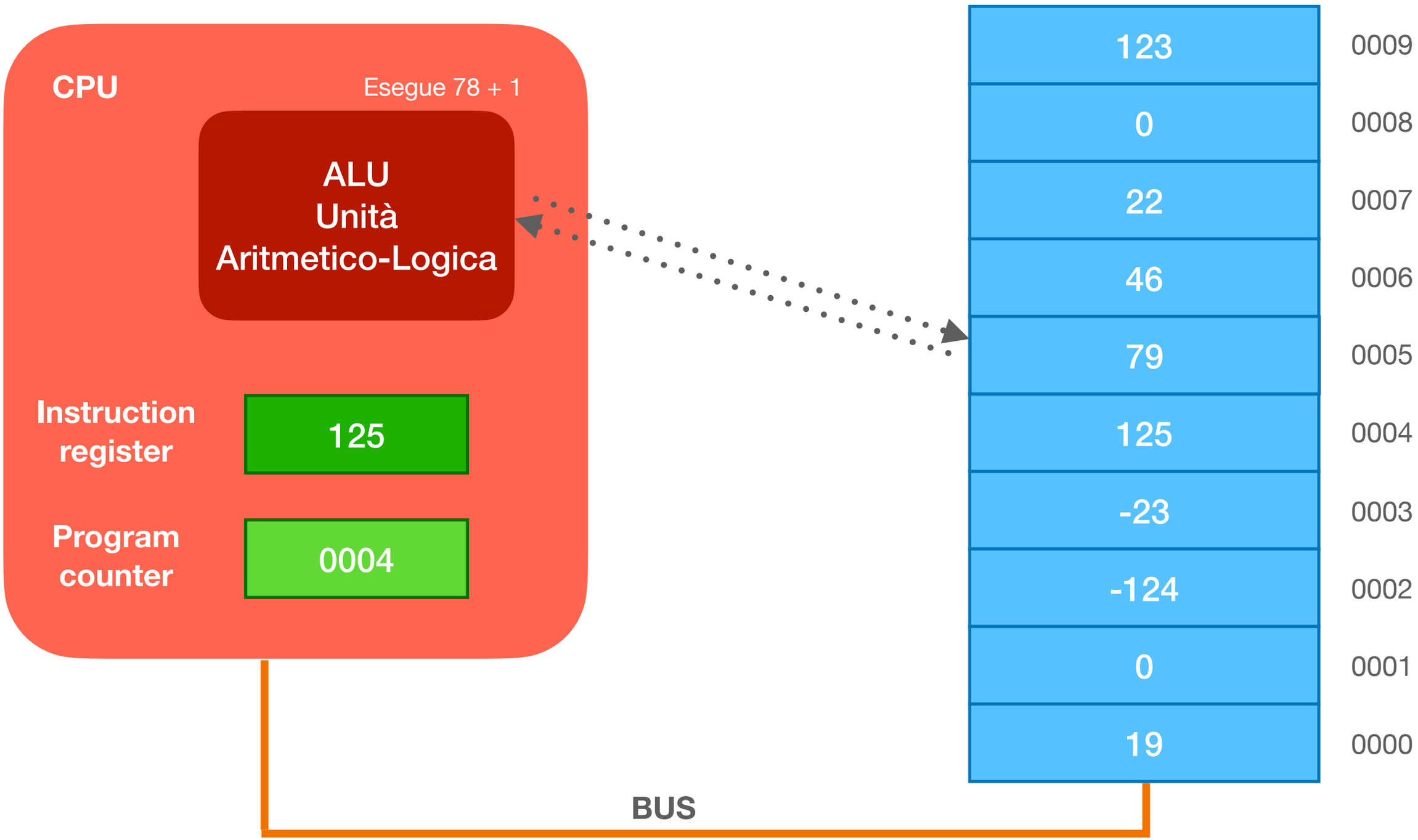
Decode

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

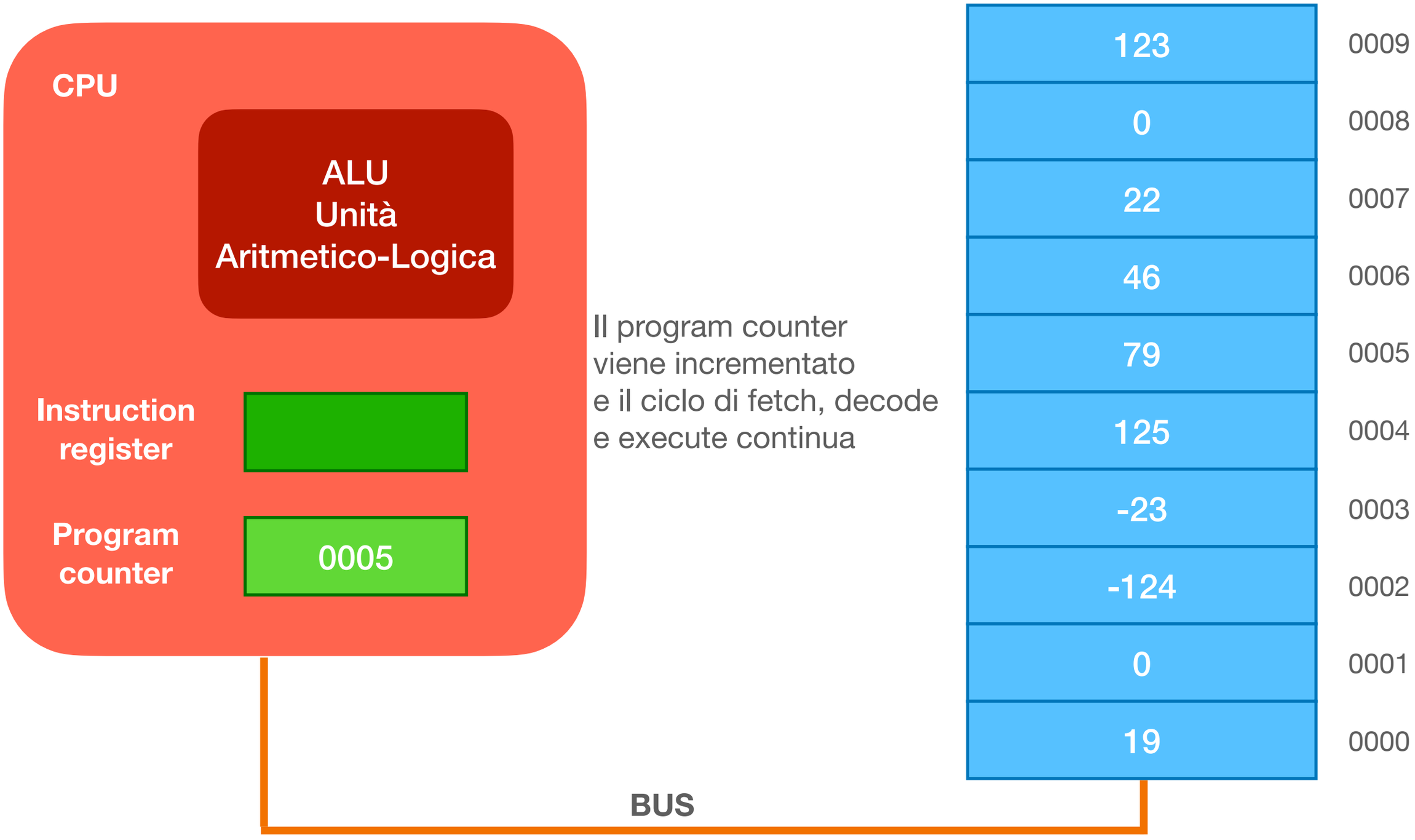
Execute

Eseguire l’operazione



Pronti a ripetere il ciclo?

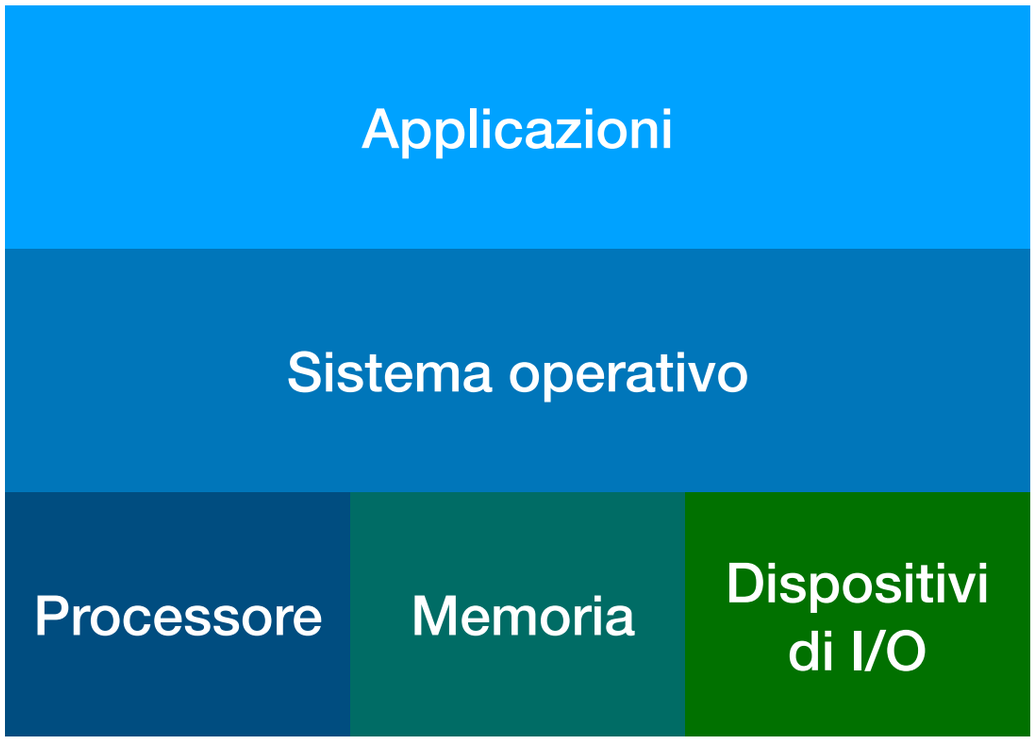
Passare all’istruzione successiva



Sistemi operativi

Molti livelli di astrazione

Dalle applicazioni all'hardware



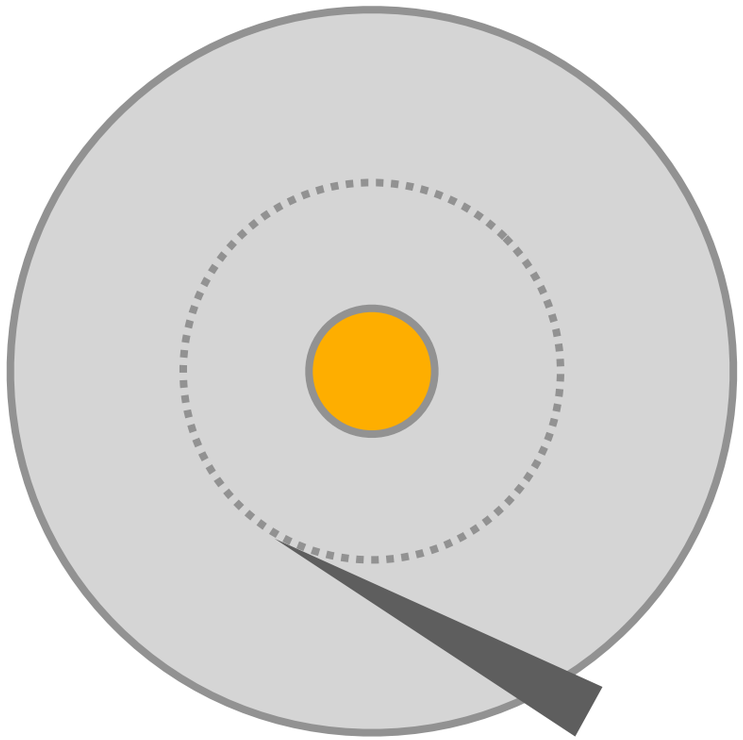
Codice scritto fino a questo momento

Fornisce una interfaccia
tra le applicazioni e l'hardware

L'hardware che, con una certa approssimazione,
possiamo dividere in tre categorie

I File

Astrazione dello storage



Piatto di un disco fisso

Le informazioni sono salvate su tracce concentriche

Per sapere come accedere dobbiamo sapere che traccia e in che punto della traccia leggere*

Come sono salvati realmente i dati

Il sistema operativo si occupa di fare questa conversione

Quello che vede la nostra applicazione

ciao_mondo.txt

“Ciao mondo”

Ma i file potrebbero anche essere su

- Disco a stato solido
- DVD
- In rete su un altro computer
- …

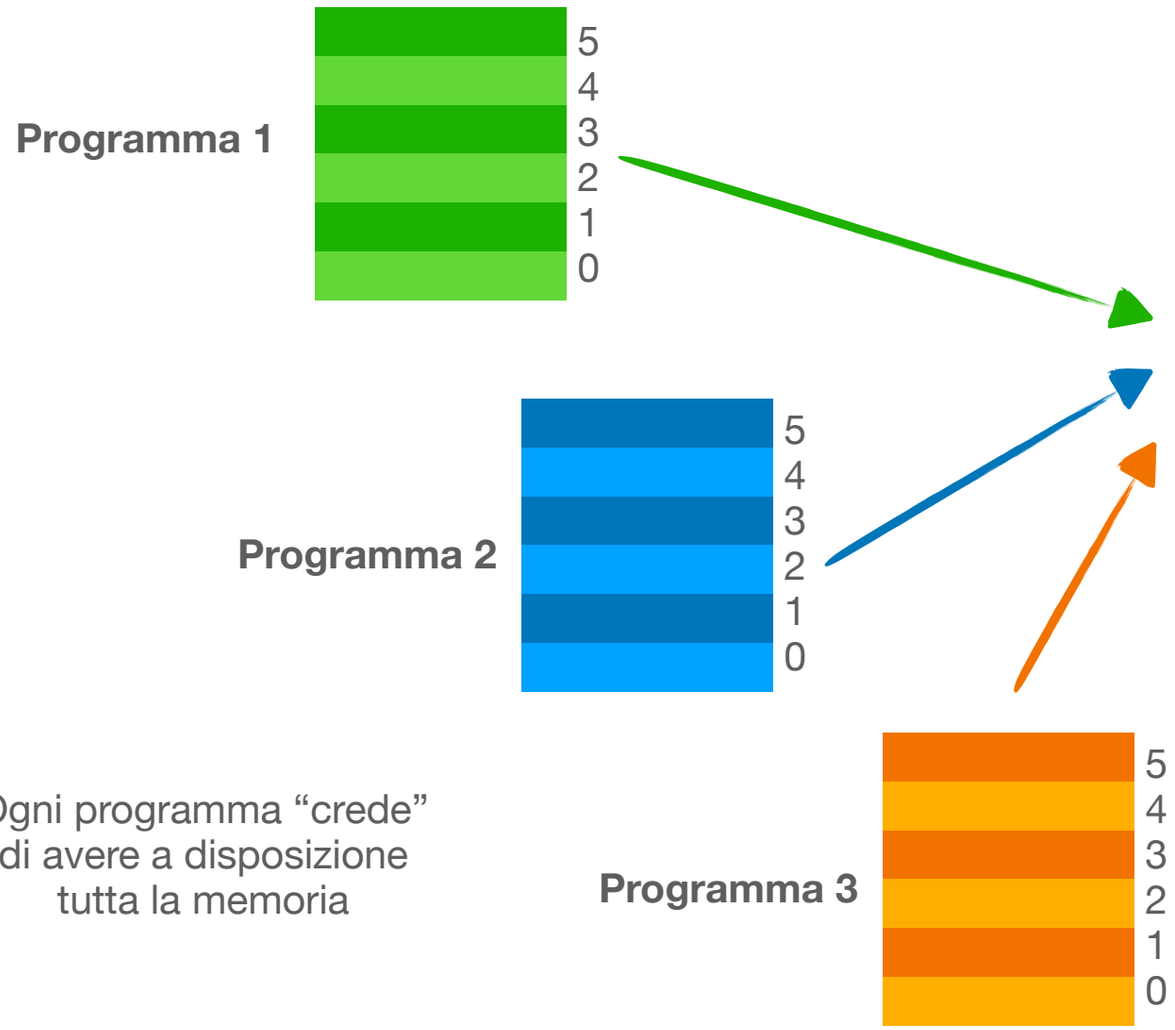
*stiamo facendo molte semplificazioni

Memoria virtuale

Astrazione della memoria

Abbiamo più di un programma in esecuzione nello stesso momento

Come fanno programmi diversi a non “pestarsi i piedi”, leggendo ognuno della memoria dell’altro?



Ogni programma “crede” di avere a disposizione tutta la memoria

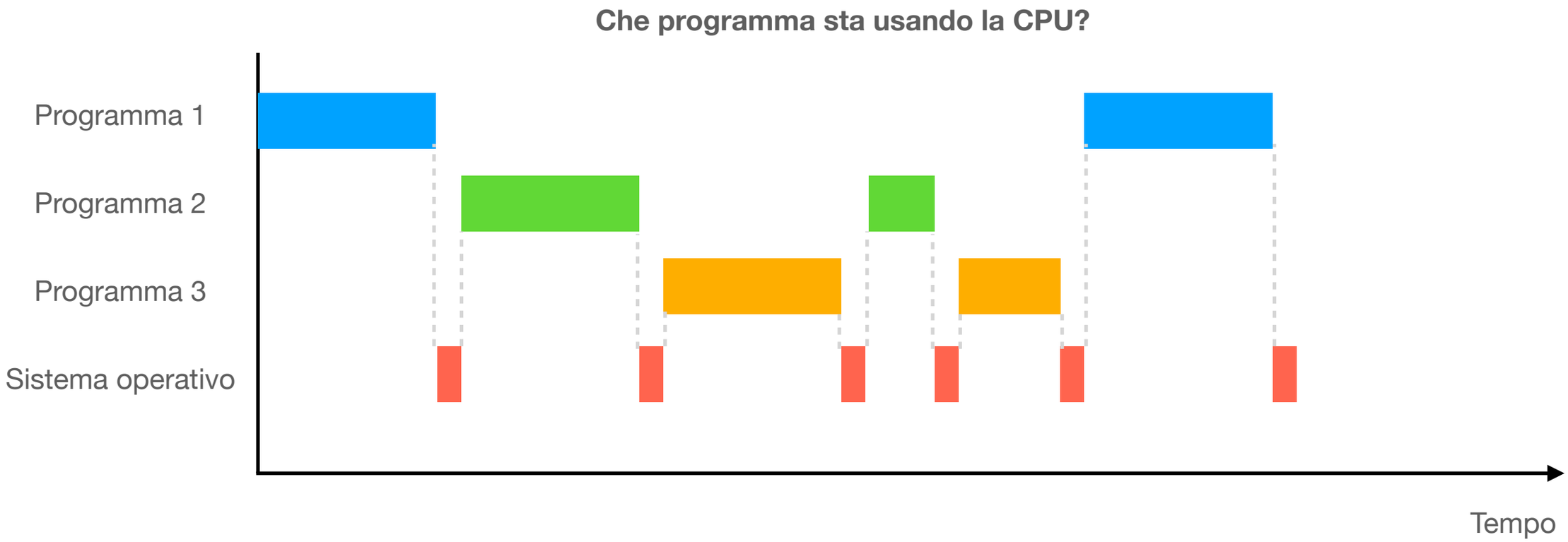
In realtà il sistema operativo (solitamente con supporto dell’hardware) mantiene un mapping tra la memoria “dei programmi" e la vera locazione nella memoria fisica

Processi

E astrazione della CPU

Abbiamo più di un programma in esecuzione nello stesso momento

Come fanno tutti i programmi a eseguire in contemporanea se il processore è uno solo?



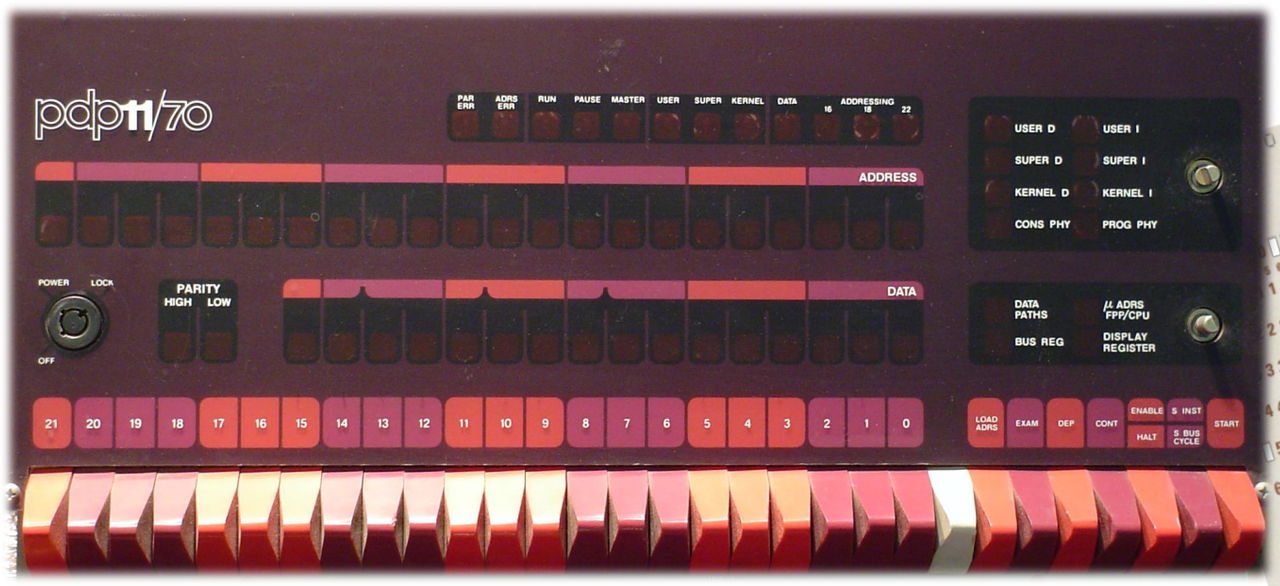
Solo un programma in esecuzione, è il sistema operativo che “cambia” tra un programma e l’altro

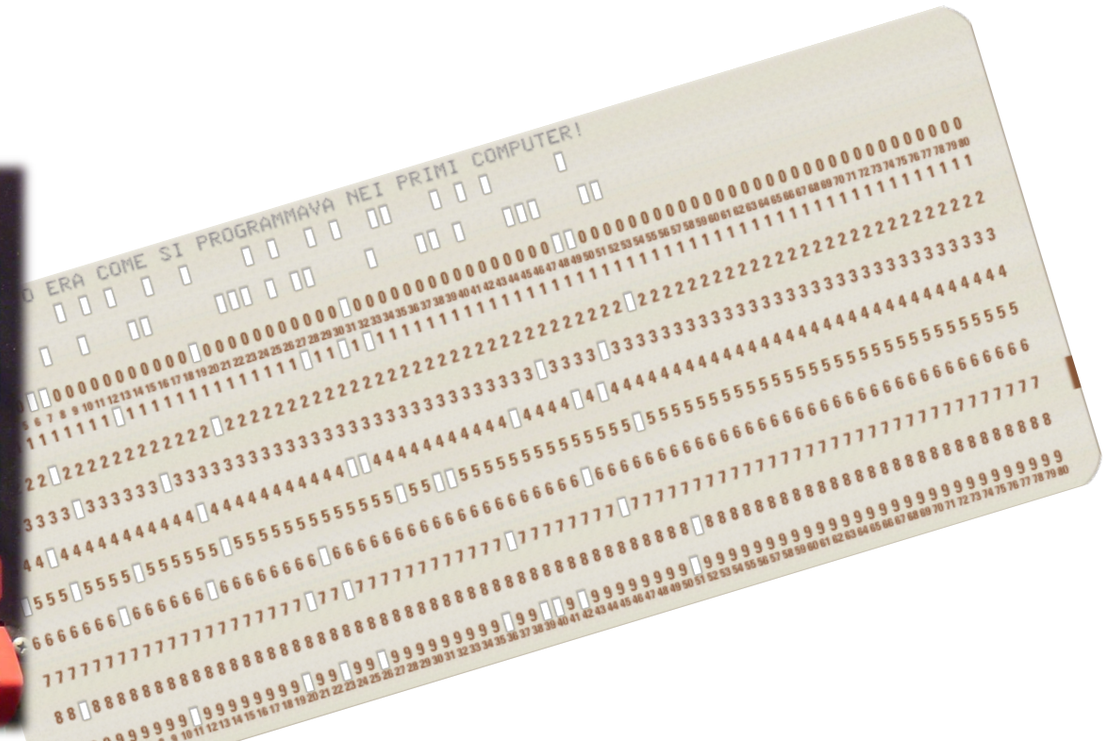
Se questo “cambio” è abbastanza veloce abbiamo l’illusione che tutti i programmi eseguano contemporaneamente

Altre funzionalità del sistema operativo

Caricare i programmi, controllo degli accessi…

- Il sistema operativo fornisce anche molte altre funzionalità:
  - Permette di caricare un programma in memoria, e farne partire l'esecuzione
  - Controlla gli accessi alle risorse permettendo di gestire utenti multipli con privilegi diversi
  - …





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
- Assembly per ARM

Struttura del corso

Sistemi operativi

- Funzionalità e storia dei sistemi operativi
- Unix e l’uso dell’ambiente Unix
- Processi e scheduling
- Memoria virtuale
- Filesystems
- Creazione e gestione di processi in C

Struttura del corso

Concorrenza

- Comunicazione tra processi
- Segnali
- Thread vs processi
- Mutex e semafori
- Problemi della programmazione concorrente
- Creazione di thread in C

Struttura del corso

Reti di calcolatori

- Stack di rete
- Ethernet
- IP
- TCP e UDP
- Tutto quello che stiamo ignorando (e perché è importante)

Struttura del corso

Argomenti avanzati

- *I seguenti argomenti verranno trattati in modo introduttivo e limitatamente ai vincoli di tempo del corso*

- Cache, pipelining, esecuzione out-of-order e jump prediction: come fanno i processori recenti a essere veloci

- Istruzioni vettoriali

- Oltre il singolo processore: tassonomia dei sistemi multiprocessore

- Programmazione su GPU: sfruttare il parallelismo dei processori grafici moderni

Algebra di Boole
e porte logiche

Algebra di Boole

Una brevissima introduzione

- Invece di avere come valori i numeri interi o i numeri reali abbiamo solo due valori: 0 e 1 (o vero e falso)
- Abbiamo tre principali operazioni
  - AND o prodotto logico, indicato come $\wedge$
  - OR o somma logica, indicato come $\vee$
  - NOT o negazione/complementazione, indicato come $\neg$
- Vediamo la semantica delle diverse operazioni introducendo le porte logiche

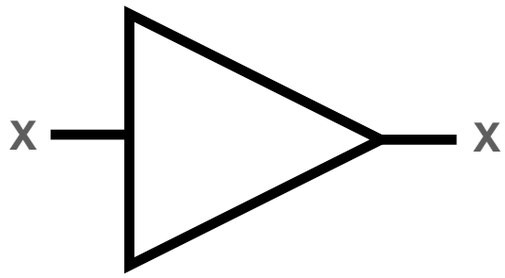
Cosa sono le porte logiche?

I mattoni con cui costruire un computer

- Una porta logica è un'astrazione che facciamo rispetto ai dispositivi fisici
- Possiamo implementare porte logiche usando transistor, relays, condutture, ingranaggi, etc.
- Una porta logica prende in input uno o più valori Booleani (0 o 1) e ritorna uno o più valori Booleani
- Le implementazioni fisiche delle porte logiche hanno molte più complicazioni (e.g., cosa succede se l'input o l'output non sono interpretabili come zero o uno? Quanto tempo ci mette una porta a produrre l'output, etc)

Porte logiche

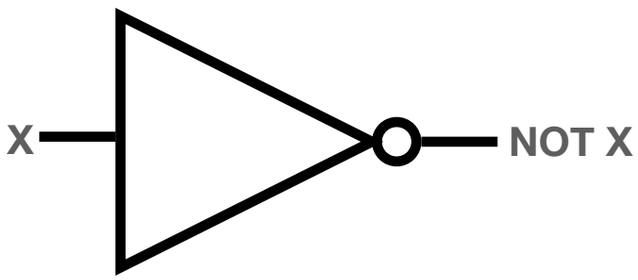
Buffer e Not



Buffer
Rappresenta la funzione identità

Tabella di verità
per ognuna delle combinazioni di input è indicato l'output

Input	Output
X	X
0	0
1	1

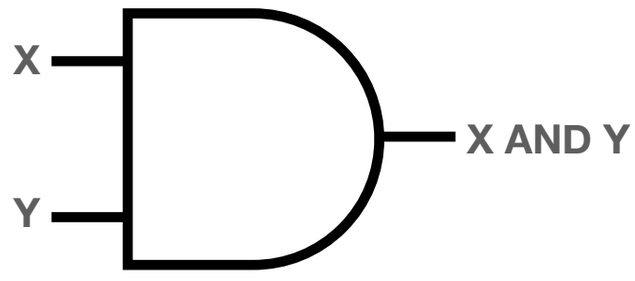


NOT/Inverter
Inverte il valore ricevuto in input

Input	Output
X	NOT X
0	1
1	0

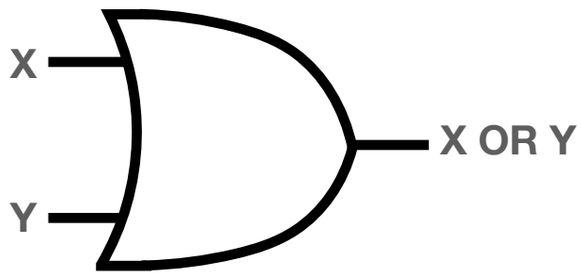
Porte logiche

And e Or



AND
Restituisce 1 solo quando entrambi gli input sono 1

Input		Output
X	Y	X AND Y
0	0	0
0	1	0
1	0	0
1	1	1

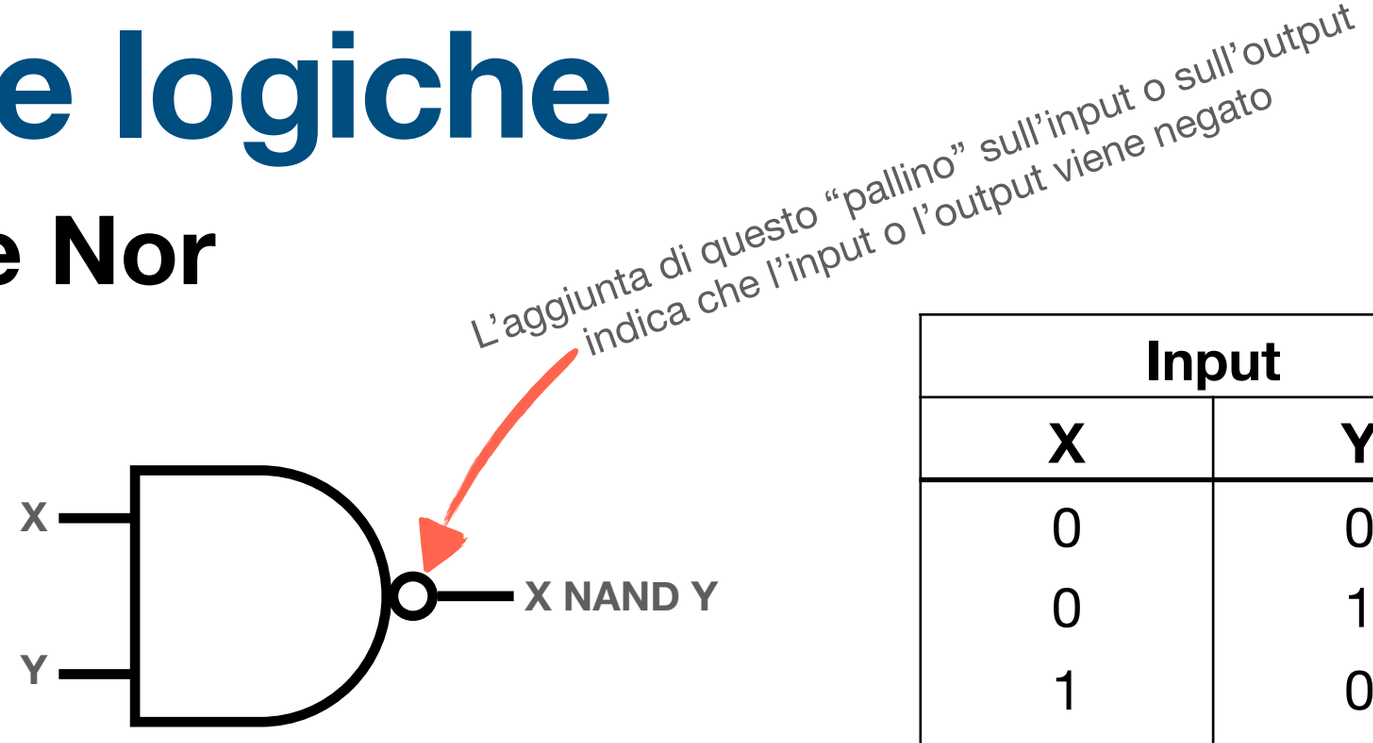


OR
Restituisce 1 quando almeno uno degli input ha valore 1

Input		Output
X	Y	X OR Y
0	0	0
0	1	1
1	0	1
1	1	1

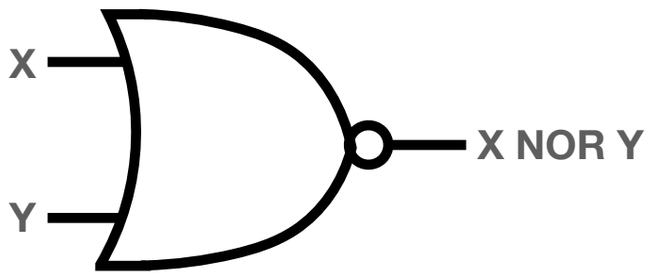
Porte logiche

Nand e Nor



Input		Output
X	Y	X NAND Y
0	0	1
0	1	1
1	0	1
1	1	0

NAND
Inverte il valore ritornato da una porta AND.
Ha output 0 solo quando entrambi gli input hanno valore 1

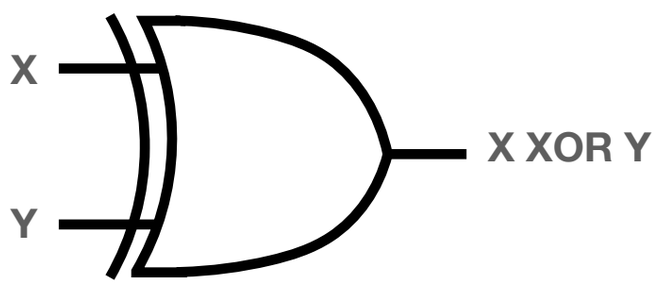


Input		Output
X	Y	X NOR Y
0	0	1
0	1	0
1	0	0
1	1	0

NOR
Inverte il valore ritornato da una porta OR.
Ha output 0 solo quando entrambi gli input hanno valore 1

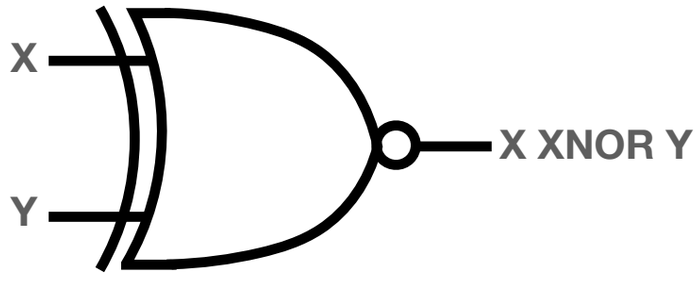
Porte logiche

Xor e Xnor



Input		Output
X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

XOR (OR Esclusivo)
L'output è 1 solamente quando *esattamente* uno degli input ha valore 1



Input		Output
X	Y	X XNOR Y
0	0	1
0	1	0
1	0	0
1	1	1

XNOR
L'output è 0 solamente quando *esattamente* uno degli input ha valore 1 (l'inverso della porta XOR)

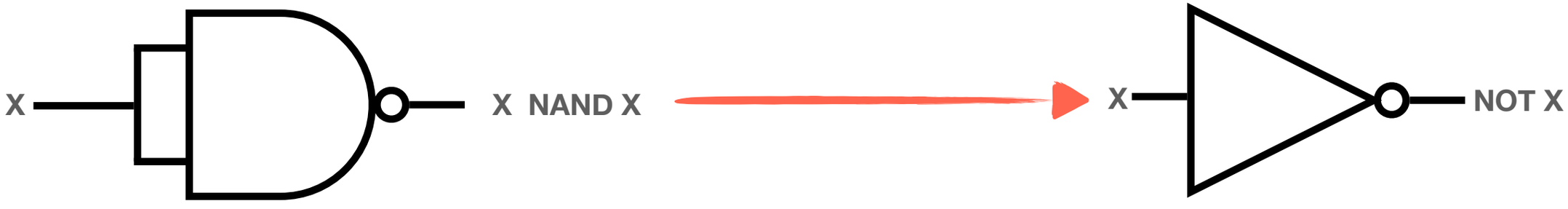
Porte logiche universali

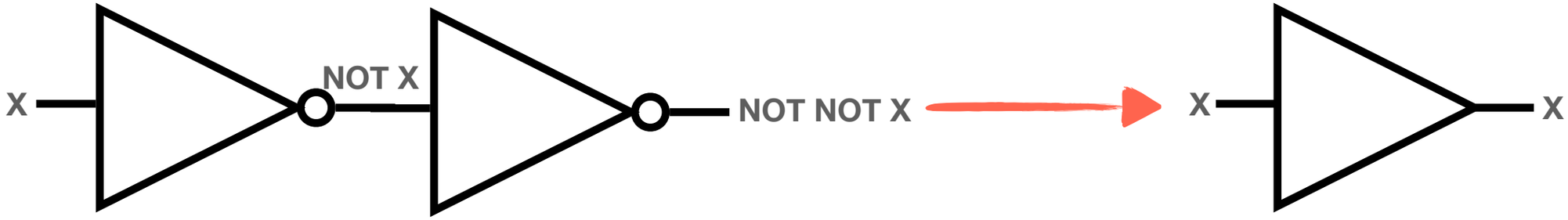
Ci servono davvero tutte le porte logiche?

- Ci si potrebbe chiedere quali porte logiche siano essenziali e quali siano ricostruibili come combinazioni di altre porte logiche
- Sia NAND che NOR sono **universali**, ovvero basta poter implementare un tipo di porta tra NAND o NOR per ricostruire tutte le altre porte logiche
- Mostriamo come ricostruire tutte le porte logiche a partire da NAND

Tutte le porte logiche da NAND

NOT, Buffer e AND





Tutte le porte logiche da NAND

Or e leggi di De Morgan

Questa conversione ci porta a una delle leggi importanti per semplificare insiemi di porte logiche: le **Leggi di De Morgan**

In pratica queste leggi di De Morgan ci dicono che possiamo “portare dentro” la negazione invertendo l’operazione.
Ovvero, AND diventa OR e OR diventa AND:

$$\neg(x \wedge y) = \neg x \vee \neg y \qquad \neg(x \vee y) = \neg x \wedge \neg y$$

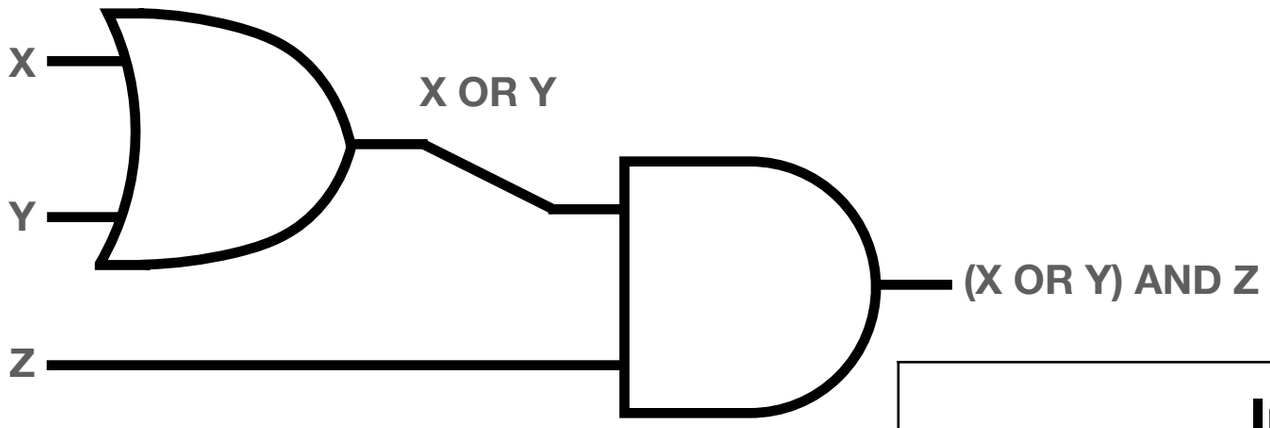
Esercizi

Per prendere la mano

- Abbiamo visto come possiamo usare la porta NAND per costruire ogni altra porta logica…
- …provate con la porta NOR!
- Abbiamo visto porte con uno o due input. Possiamo combinarle per formare porte con tre input. Per esempio per calcolare $(x \vee y) \wedge z$
  - Graficamente come rappresentiamo quella formula?
  - Quale è la sua tabella di verità?

Combinare porte logiche

Un esempio



Il numero di righe della tabella per n input è 2^n (ogni nuovo input raddoppia le configurazioni possibili)

Input			Output	
X	Y	Z	X OR Y	(X OR Y) AND Z
0	0	0	0	0
0	0	1	0	0
0	1	0	1	0
0	1	1	1	1
1	0	0	1	0
1	0	1	1	1
1	1	0	1	0
1	1	1	1	1

Materiale aggiuntivo

Algebra di Boole

proprietà delle algebre di Boole

- $x \wedge 0 = 0$ esistenza di un minimo
- $x \vee 1 = 1$ esistenza di un massimo
- $x \wedge y = y \wedge x$ commutatività dell'AND
- $x \vee y = y \vee x$ commutatività dell'OR
- $(x \wedge y) \wedge z = x \wedge (y \wedge z)$ associatività dell'AND
- $(x \vee y) \vee z = x \vee (y \vee z)$ associatività dell'OR

Algebra di Boole

proprietà delle algebre di Boole

- $x \wedge x = x$ idempotenza dell'AND
- $x \vee x = x$ idempotenza dell'OR
- $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$ proprietà distributiva
- $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$ proprietà distributiva
- $x \wedge \neg x = 0$ e $x \vee \neg x = 1$ esistenza di un complemento
- $x \vee (x \wedge y) = x$ e $x \wedge (x \vee y) = x$ assorbimento