

(Architetture e) Sistemi Operativi

Introduzione alla struttura di un calcolatore

Luca Manzoni

Argomenti

- Rappresentazione dell'informazione: numeri binari, rappresentazione in complemento a due, numeri floating point
- Struttura di base di un calcolatore e ciclo fetch-decode-execute
- Un esempio pratico con l'architettura ARM-v7a:
 - Istruzioni aritmetico-logiche
 - Salti condizionali e non
 - Interagire con la memoria
- Interagire coi dispositivi di input/output: polling vs interrupt, memory-mapped vs port-mapped I/O

Rappresentare l'informazione

Rappresentare l'informazione

Quando si hanno solo due simboli

- Per elaborare l'informazione il computer utilizza solo due simboli, che indichiamo con 0 e 1
- Questo significa che **ogni** informazione deve essere codificata usando solo sequenze di questi simboli:
 - Numeri interi
 - Numeri reali
 - Caratteri
 - Istruzioni
 - ...

Bits and Bytes

Dal singolo valore binario a 8 bit

Il bit rappresenta una singola cifra binaria.

Ovvero può assumere solo due valori, 0 e 1

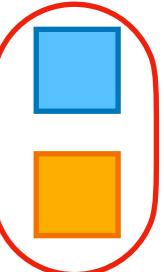


Una sequenza di 8 bit è detta **byte**

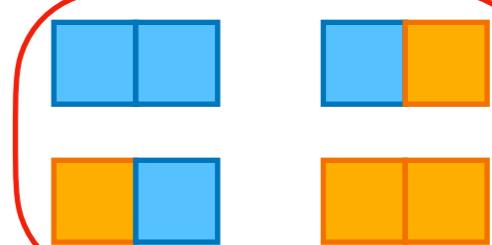
Quante diverse sequenze di 8 bit esistono?

Ogni posizione può assumere 2 valori, vi sono 8 posizioni,
quindi $\underbrace{2 \times 2 \times \dots \times 2}_{8 \text{ volte}} = 2^8 = 256$ possibili sequenze diverse

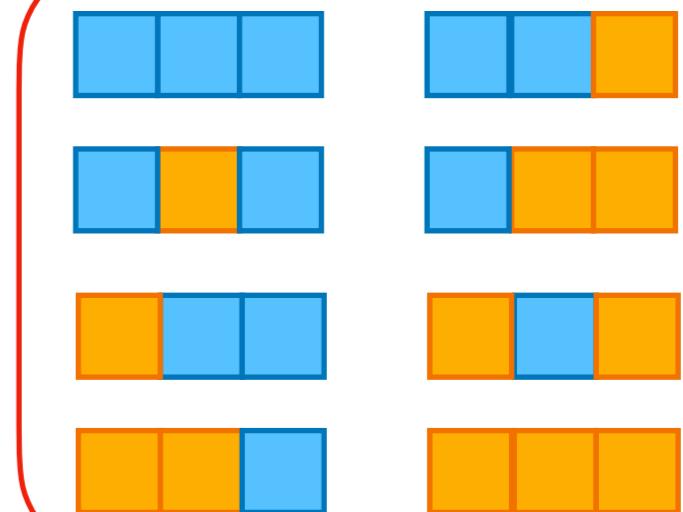
2 sequenze di 1 bit



4 sequenze di 2 bit



8 sequenze di 3 bit



In generale abbiamo 2^n sequenze distinte di n bit

Kilo, Mega, Giga, Tera, ...

Multipli in base 2 e in base 10

- Esiste una possibilità di fraintendimenti quando si parla di Kilobyte, Megabyte, etc
- Tradizionalmente, dato che $2^{10} = 1024$ è molto vicino a $10^3 = 1000$, il prefisso “Kilo” indicava 1024 byte, “Mega” $1024^2 = 2^{20}$ byte, etc
- In altri casi il prefisso “Kilo” indicava 1000 byte, “Mega” $1000^2 = 10^6$ byte, etc
- Per chiarire se si deve moltiplicare per 1000 (“base 10”) o 1024 (“base 2”) esistono nomi diversi...
- ...ma non tutti li usano (e.g., la memoria di solito è in “base 2”, lo storage in “base 10”)

Kilo, Mega, Giga, Tera, ...

Multipli in base 2 e in base 10

Nome	Sigla	Dimensione
Kilobyte	KB	1000 byte
Kibibyte	KiB	1024 byte
Megabyte	MB	1000 KB
Mebibyte	MiB	1024 KiB
Gigabyte	GB	1000 MB
Gibibyte	GiB	1024 MB
Terabyte	TB	1000 GB
Tebibyte	TiB	1024 GiB
Petabyte	PB	1000 TB
Pebibyte	PiB	1024 TiB

Numeri in base 2

Rappresentazione in base 10

Capire come scriviamo i numeri

Proviamo a vedere come interpretiamo un numero:

2508



Dato che usiamo una notazione posizionale
a ogni cifra corrisponde un valore che dipende
dalla posizione in cui si trova

$$2 \times 1000 + 5 \times 100 + 0 \times 10 + 8 \times 1$$



Notiamo una certa struttura nei valori che
moltiplicano le singole cifre... sono
tutte potenze di 10

$$2 \times 10^3 + 5 \times 10^2 + 0 \times 10^1 + 8 \times 10^0$$

In generale il valore denotato da una sequenza

$\langle x_1 x_2 \dots x_n \rangle_{10}$ di n interpretato in base 10 è dato da:

$$\langle x_1 x_2 \dots x_n \rangle_{10} = \sum_{i=1}^n x_i 10^{n-i}$$

Rappresentazione in altre basi

Capire come scriviamo i numeri

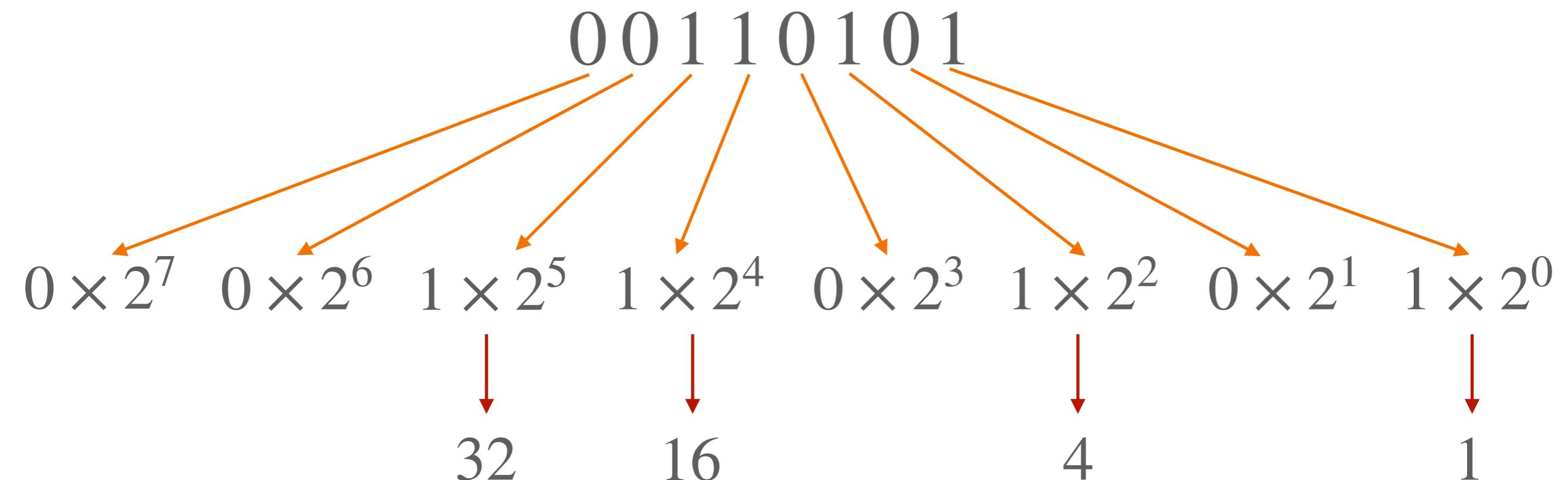
- Usiamo la base 10 perché abbiamo 10 cifre da 0 a 9
- Qualcosa ci limita ad usare esattamente 10 cifre?
- No, possiamo usare un qualunque numero di cifre:
 - Due: 0,1
 - Otto: 0,1,2,3,4,5,6,7
 - Sedici: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
- In generale in base b una sequenza di n cifre viene interpretata come

$$\langle x_1 x_2 \dots x_n \rangle_b = \sum_{i=1}^n x_i b^{n-i}$$



Rappresentazione in base 2

Usare solo due cifre



$$32 + 16 + 4 + 1 = 53$$

Conversione binario-decimale

Alcuni esercizi

	Potenze di 2
0000 0000	$2^0 = 1$
0000 0100	$2^1 = 2$
0000 0011	$2^2 = 4$
1111 1111	$2^3 = 8$
	$2^4 = 16$
	$2^5 = 32$
	$2^6 = 64$
	$2^7 = 128$
0000 0011	$2^1 + 2^0 = 3$
1111 1111	$2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 255$

Ottale e esadecimale

Usare 8 o 16 cifre

In aggiunta ad usare la base 2 altre basi che appaiono sono base 8 (o “ottale”) e base 16 (o “esadecimale”)

Per la base 8 usiamo le cifre 0,1,2,3,4,5,6,7

Per la base 16 ci servono cifre oltre il 9 e, per convenzione, si usano le lettere dell’alfabeto: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

A sarà 10 in base 10, B sarà 11 in base 10, fino a F che sarà 15 in base 10

7BE in esadecimale è

$$7 \times 16^2 + 11 \times 16^1 + 14 \times 16^0$$

↓ ↓ ↓
1792 + 176 + 14
 ↓ ↓
 1982

Somma di numeri in binario

Come in base 10, ma con due cifre

L'algoritmo per la somma di numeri in binario (o in qualsiasi base) rimane lo stesso che si usa per i numeri in base 10:

$$\begin{array}{r} 1101\ 1001 \\ 0010\ 0011 \\ \hline 1111\ 1100 \end{array}$$

Arrighi: $1_2 + 1_2 = 10_2$
Quindi abbiamo 0 col riporto di 1

Similmente è possibile definire la moltiplicazione di numeri in binario, sapendo che:
 $0 \times 0 = 0, 0 \times 1 = 0, 1 \times 0 = 0$, e $1 \times 1 = 1$

Overflow

Utilizzo di un numero finito di bit

- Solitamente in un calcolatore i numeri non sono rappresentati con un numero non limitato di bit ma con un numero *fissato* di bit
- Esempi: 16, 32, 64 bit
- Cosa succede quando una somma ci porterebbe oltre il massimo numero rappresentabile?
- Per esempio $1111\ 1111 + 0000\ 0001$

Overflow

Utilizzo di un numero finito di bit

$$\begin{array}{r} 1111\ 1111 + \\ 0000\ 0001 = \\ \hline 10000\ 0000 \end{array}$$

I primi otto bit sono quelli che teniamo
e abbiamo ottenuto 0

Otteniamo quindi un **overflow** (“straripamento”), dato che il numero di bit che ci servirebbero è superiore a quello che abbiamo

In generale per i numeri senza segno di n bit possiamo rappresentare solo valori tra 0 e $2^n - 1$ e ogni operazione che porta a risultati fuori dal range avrà un risultato errato

Rappresentazione in complemento a due

Numeri negativi

E come rappresentarli

- Generalmente per rappresentare un numero negativo utilizziamo il segno “–”
- Questo però non è disponibile quando dobbiamo rappresentare tutto come sequenza 0 e 1
- Vedremo diversi approcci, ognuno con vantaggi e svantaggi:
 - Segno e modulo
 - Complemento a 1
 - Complemento a 2
 - Eccesso N

Segno e modulo

Un approccio naïve

- Un primo approccio è quello di utilizzare il primo bit di un numero come bit di segno (0 indica un numero positivo e 1 un numero negativo)
- Per esempio 1000 1011 viene interpretato come:
 - 1 in prima posizione indica che il segno è meno
 - 000 1011 viene interpretato come
$$2^3 + 2^1 + 2^0 = 8 + 2 + 1 = 11$$
 - Quindi 1000 1011 viene interpretato come -11

Segno e modulo

I problemi di questo approccio

- Con n bit possiamo rappresentare i valori tra $-2^{n-1} - 1$ e $2^{n-1} - 1$
- L'approccio con segno e modulo ha diversi problemi
- Esistono **due** rappresentazioni del valore 0, una con segno meno e una con segno più:
 - $1000\ 0000 = -0$ e $0000\ 0000 = +0$
- La somma tra numeri positivi e numeri negativi **non** funziona con la stessa procedura che usiamo per numeri senza segno

Complemento a due

La rappresentazione più usata

- Nel complemento a due il bit più significativo in una sequenza di n bit viene interpretato non come moltiplicante 2^{n-1} ma come moltiplicante -2^{n-1}
- In 8 bit il valore più piccolo è quindi rappresentato da 1000 0000, ovvero $-2^7 = -128$
- Sempre in 8 bit il valore più grande è rappresentato da 0111 1111, ovvero $2^6 + 2^5 + \dots + 2^0 = +127$
- In generale usando il complemento a 2 su n bit possiamo rappresentare tutti i numeri da -2^{n-1} a $2^{n-1} - 1$

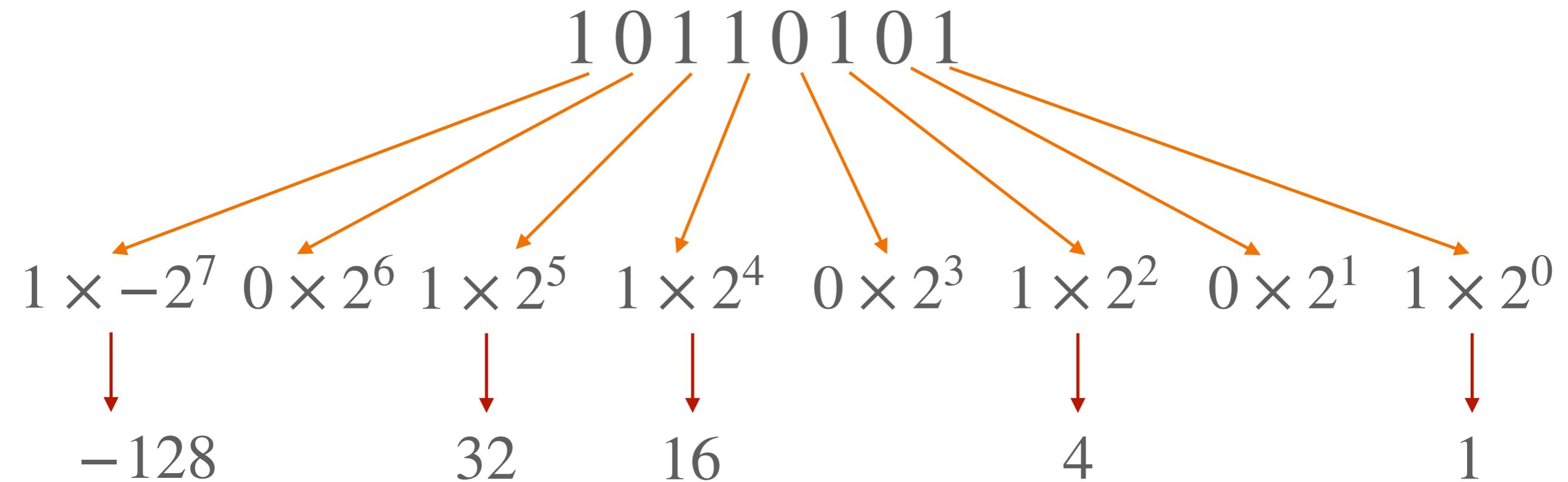
Complemento a due

La rappresentazione più usata

- Abbiamo che 0 ha una **unica** rappresentazione come 0000 0000
- Il primo bit ci indica comunque il segno
- Abbiamo altri vantaggi:
 - I numeri tra 0 e $2^{n-1} - 1$ non cambiano rappresentazione rispetto ai numeri senza segno
 - Possiamo usare gli stessi circuiti che usiamo per la somma di numeri senza segno

Complemento a 2

Esempio



$$-128 + 32 + 16 + 4 + 1 = -128 + 53 = -75$$

Eccesso N

Cambiare da dove si inizia a contare

- Nella notazione a eccesso N si sceglie un valore N e tutti i numeri sono interpretati come numeri binari positivi che rappresentano uno scostamento dal valore -N
- Per esempio se $N = 128$ avremmo:
 - 0000 0000 rappresenta $0 - 128 = -128$
 - 1000 0000 rappresenta $128 - 128 = 0$
 - 1111 1111 rappresenta $255 - 128 = 127$

Numeri floating point

Standard IEEE 754

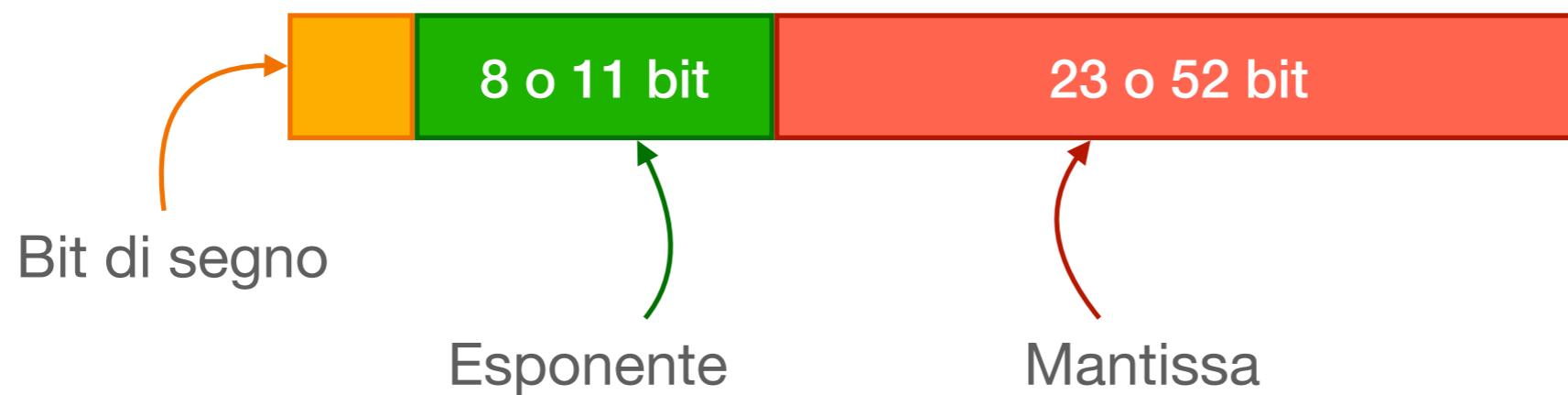
Rappresentazione dei numeri floating point

- In aggiunta ai numeri interi vogliamo poter rappresentare anche numeri reali
- Un computer, anche senza avere limiti di memoria **non** può rappresentare ogni numero reale
- La rappresentazione è necessariamente approssimata
- In generale vogliamo qualcosa di simile ad una notazione scientifica:
 - numero fissato di cifre significative (contenute nella *mantissa*)
 - moltiplicate per una base (10 in notazione scientifica, 2 nei numeri floating point)
 - elevata a un certo *esponente*

Standard IEEE 754

Struttura di un numero floating point

Lo standard IEEE754 prevede numeri a precisione **singola** (32 bit), **doppia** (64 bit) e, meno diffusi, a mezza precisione (16 bit) e quadrupla precisione (128 bit)



Il numero rappresentato è nella forma \pm mantissa $\times 2^{\text{esponente}}$

Quindi il numero di bit della mantissa ci dice quanta precisione abbiamo

Mentre il numero di bit dell'esponente ci dice quanto grandi (o piccoli) sono i numeri che possiamo rappresentare

Infiniti e NaN

Valori che non sono numeri reali

Lo standard IEEE754 prevede la possibilità che alcune operazioni ritornino dei valori che non rappresentano numeri validi, questo con il valore NaN (not a number)

In aggiunta a questo, lo standard prevede anche la possibilità di rappresentare un valore infinito positivo o negativo ($+\infty$ e $-\infty$)

Per rappresentare valori con una precisione ridotta vi è la possibilità di usare dei numeri **denormalizzati**.

Lo standard IEEE 754 per i numeri floating point è abbastanza complesso.
Vedremo facendo esercizi alcune delle sue peculiarità, ma non andremo troppo nei dettagli

Visualizzazione dei numeri floating point: <https://bartaz.github.io/ieee754-visualization/>

Problemi dei numeri floating point

Rappresentazioni inesatte

```
>>> 0.1 + 0.2  
0.30000000000000004
```

Risultato inesatto dovuto alla limitata precisione dei numeri floating point

```
>>> 10**16 - 0.1 == 10**16  
True
```

Risultato inesatto dovuto all'approssimazione nella rappresentazione dei numeri floating point

```
>>> 10**16 - (10**16 + 0.1) == 0  
True  
>>> (10**16 - 10**16) + 0.1 == 0  
False
```

Non associatività delle operazioni

Come fare?

ASCII e Unicode

Rappresentazione di testo

Da ASCII a Unicode

- Per rappresentare il testo si usano comunque sequenze di bit
- Un formato tradizionale è ASCII, che richiede 7 bit per carattere (solitamente con “padding” a 8 bit per usare un intero byte)
- A ogni sequenza di 7 bit è associato un carattere:
 - Alfanumerici: 0,1,2,..., A,B,...,Z, a,b,...z (notare come maiuscole e minuscole siano entrambe presenti)
 - Punteggiatura: ,/,%,{,},...
 - Spazio: “ ”
 - Caratteri di controllo

Rappresentazione di testo

Da ASCII a Unicode

- Con 7 bit si possono rappresentare solo 128 diversi caratteri (inclusi i caratteri di controllo)
- Sufficienti per l'inglese ma non per altre lingue (anche solo per i caratteri accentati in italiano)
- Una serie di estensioni per codifica di caratteri erano state proposte (si veda lo standard ISO/IEC 8859) con molte parti (a seconda della lingua che si voleva usare)
- Più recentemente lo standard unicode permette di codificare buona parte dei simboli usati per la scrittura. Ogni carattere è codificato in un numero variabile di byte da 1 a 4

Architettura del calcolatore

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;
}
```

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

```
#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;
}
```

```
.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

*Notate come le istruzioni siano diverse
a seconda dell'architettura*

```
.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

- 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

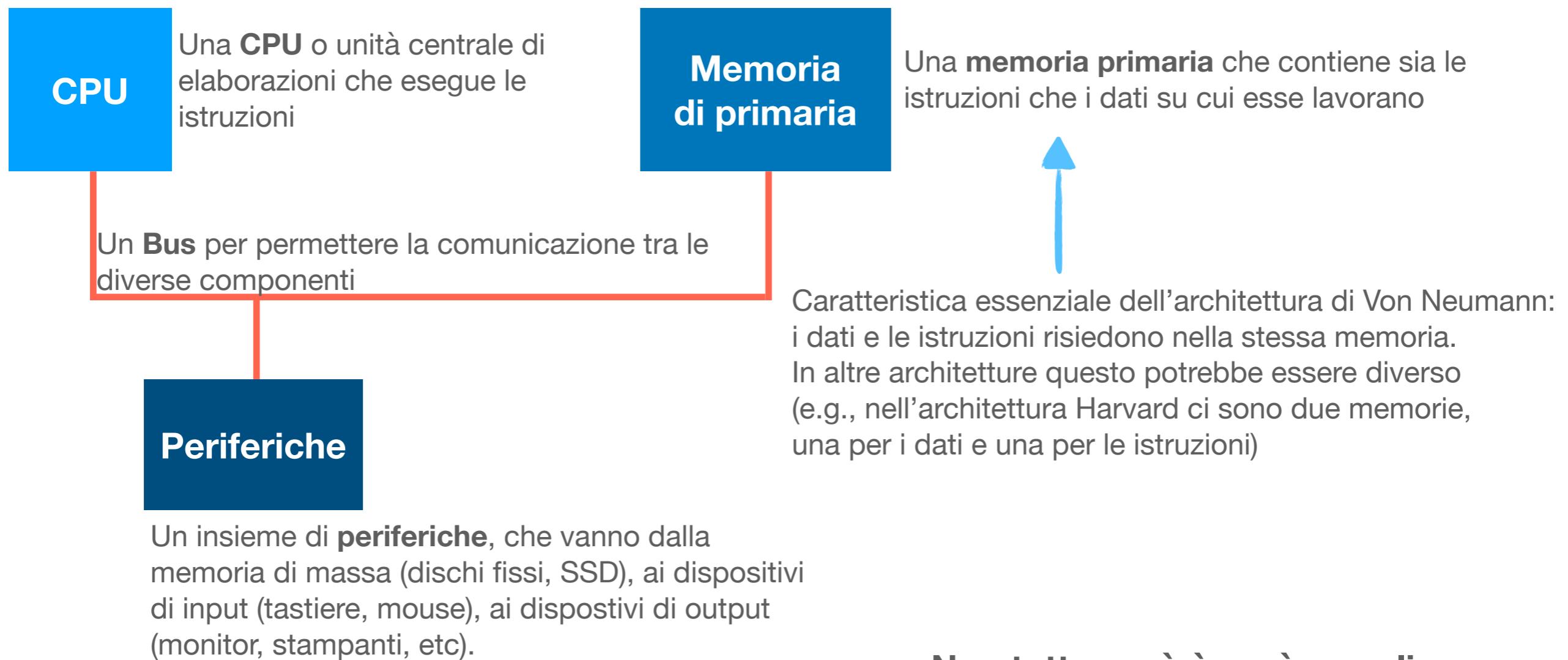
Dal codice macchina all'esecuzione

- 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?

Struttura di base del calcolatore

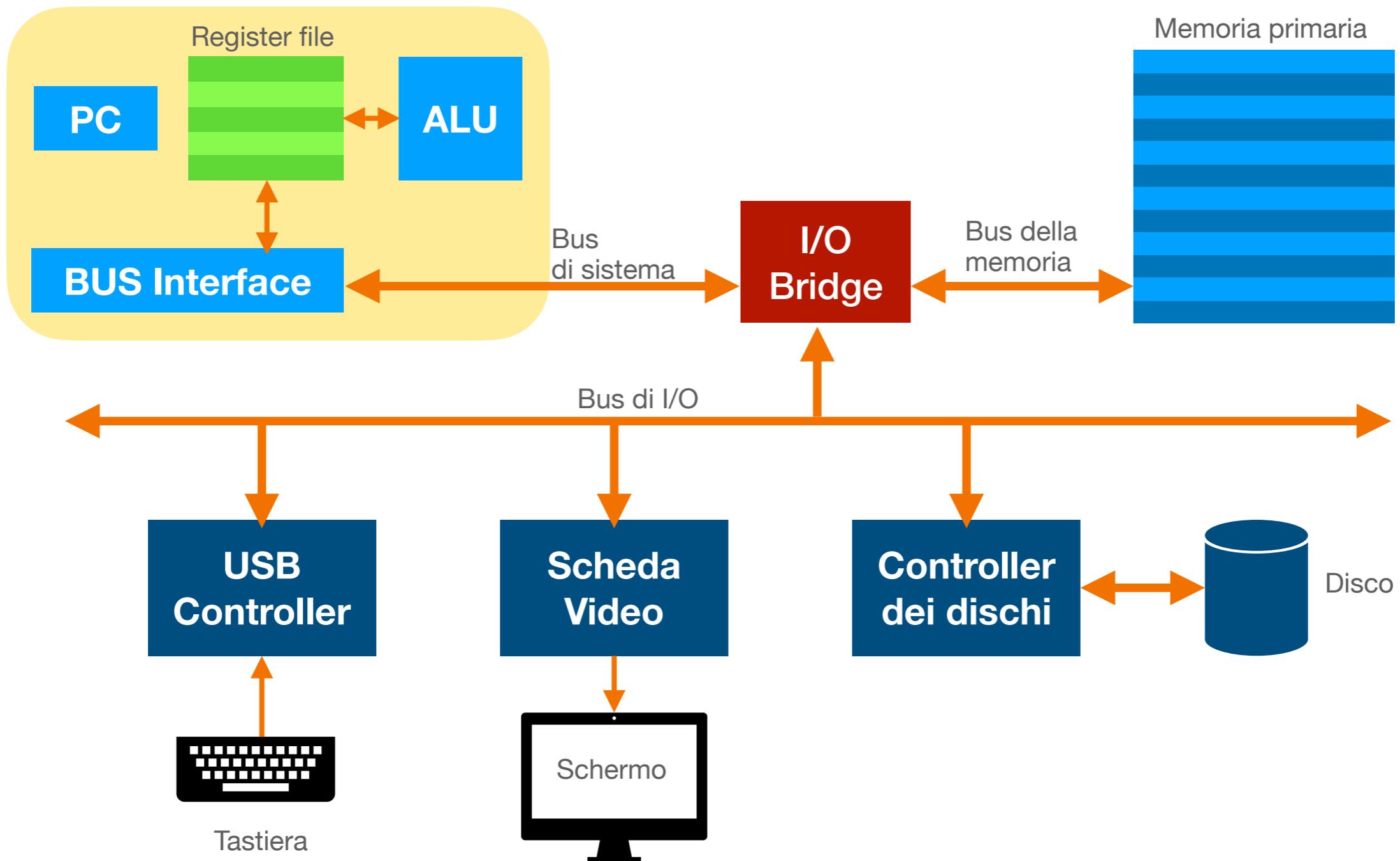
E l'architettura di Von Neumann

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



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



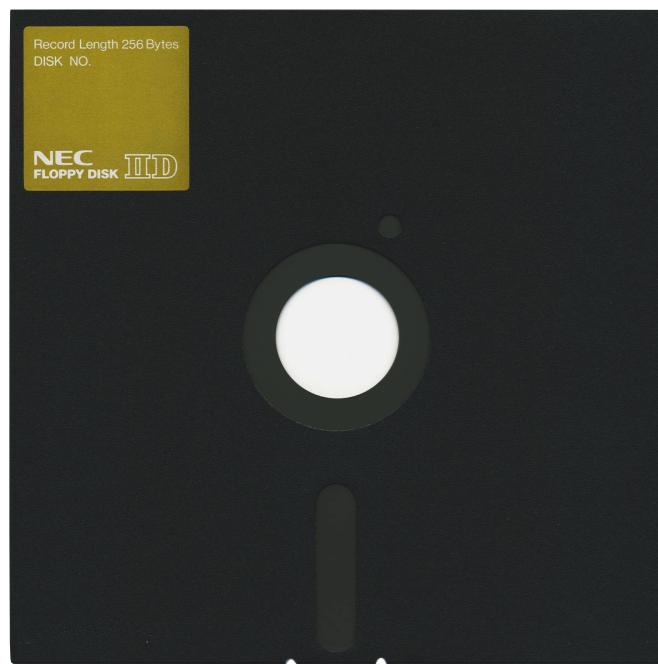
Hard disk

Informazioni memorizzate nelle cariche magnetiche di piatti che girano a migliaia di giri al minuto.
Capienza fino ad alcuni TB



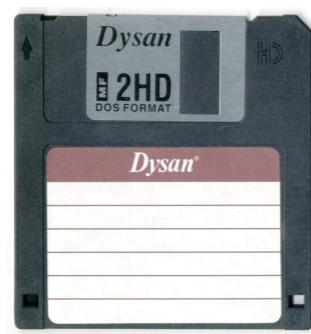
Dischi a stato solido

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

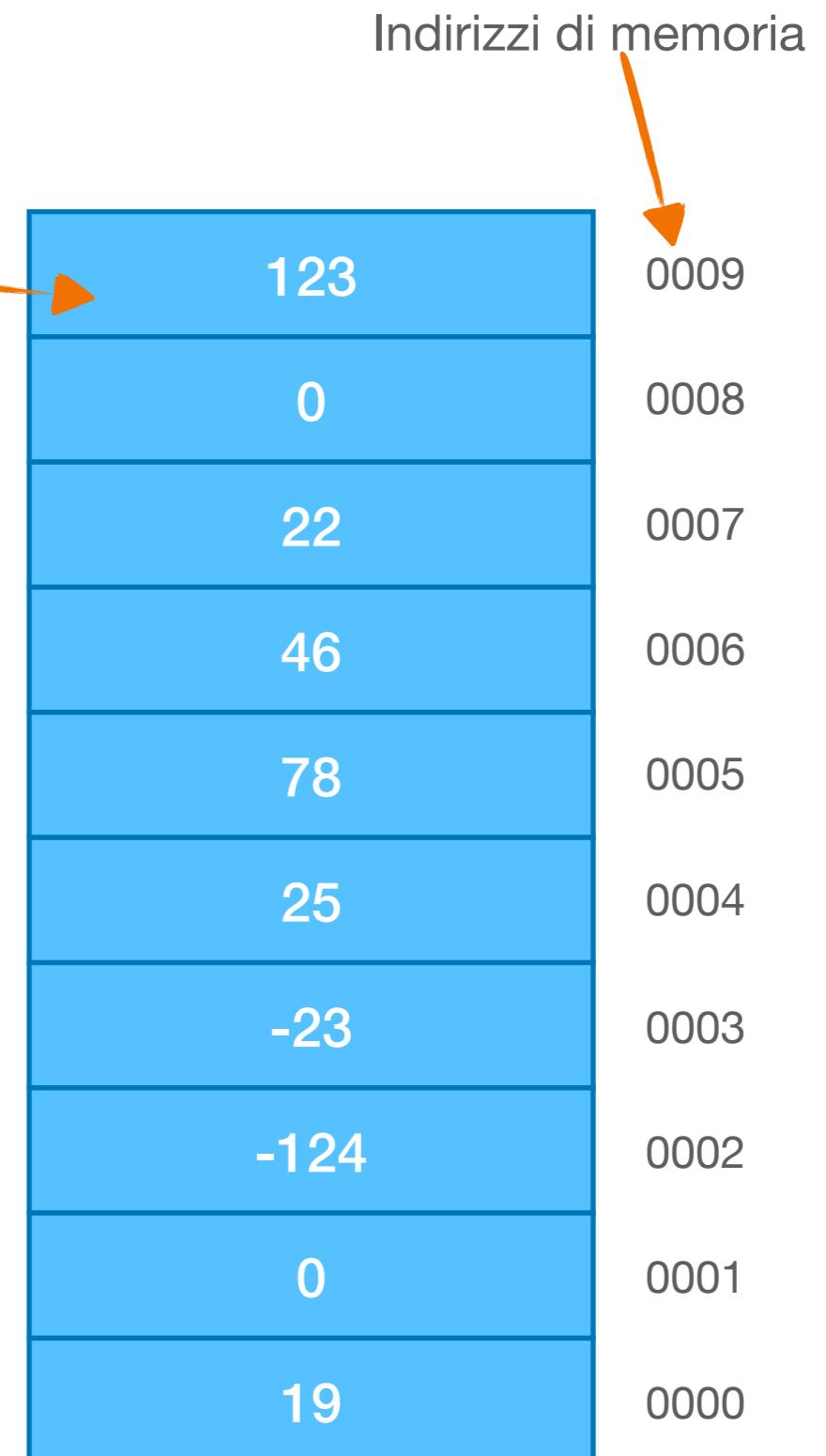
La memoria è organizzata in modo lineare

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

Contenuto della memoria



LA CPU

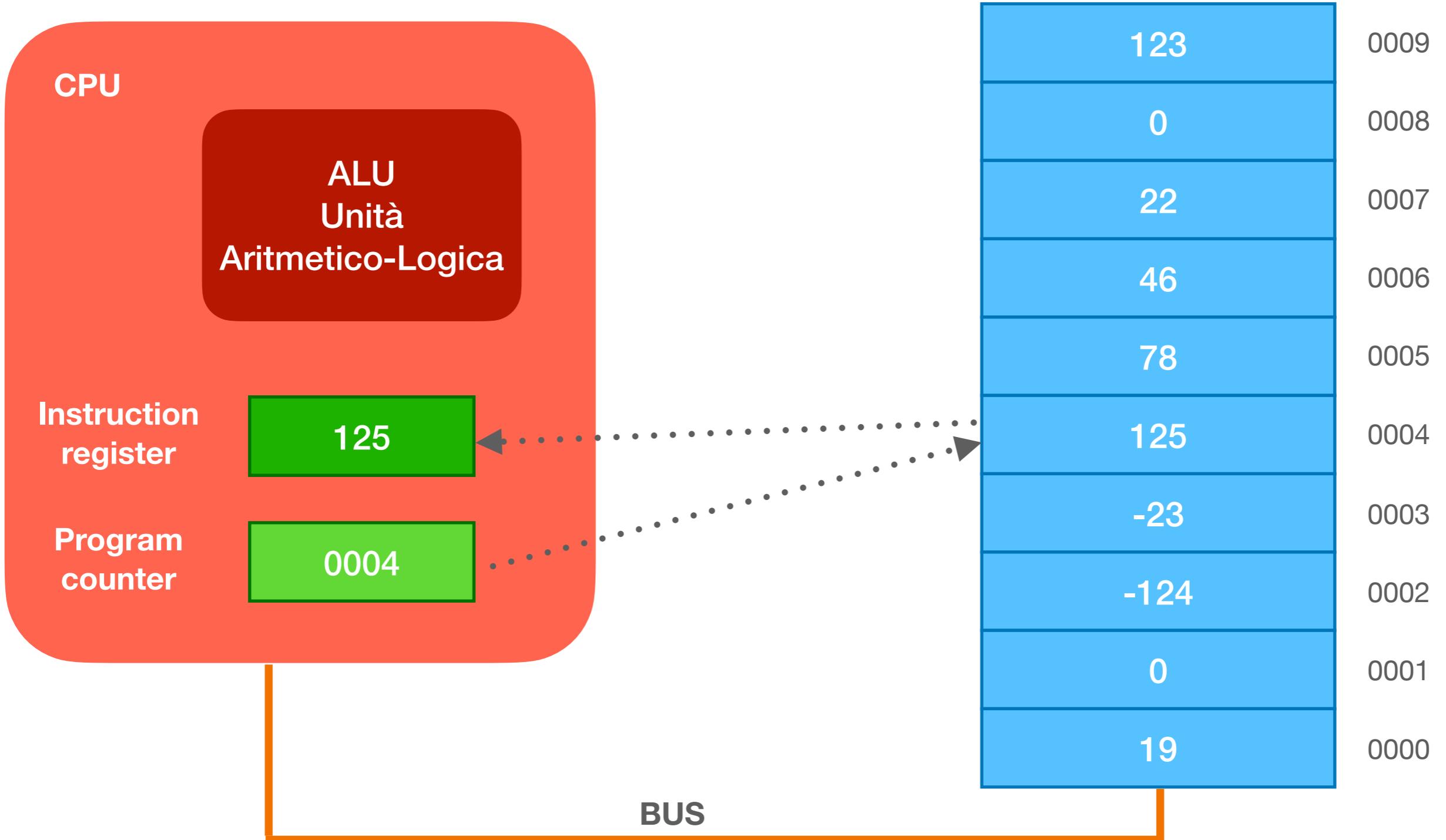
Ciclo di fetch-decode-execute

- 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

Ottenerne una istruzione



Fetch

Ottenerne 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)

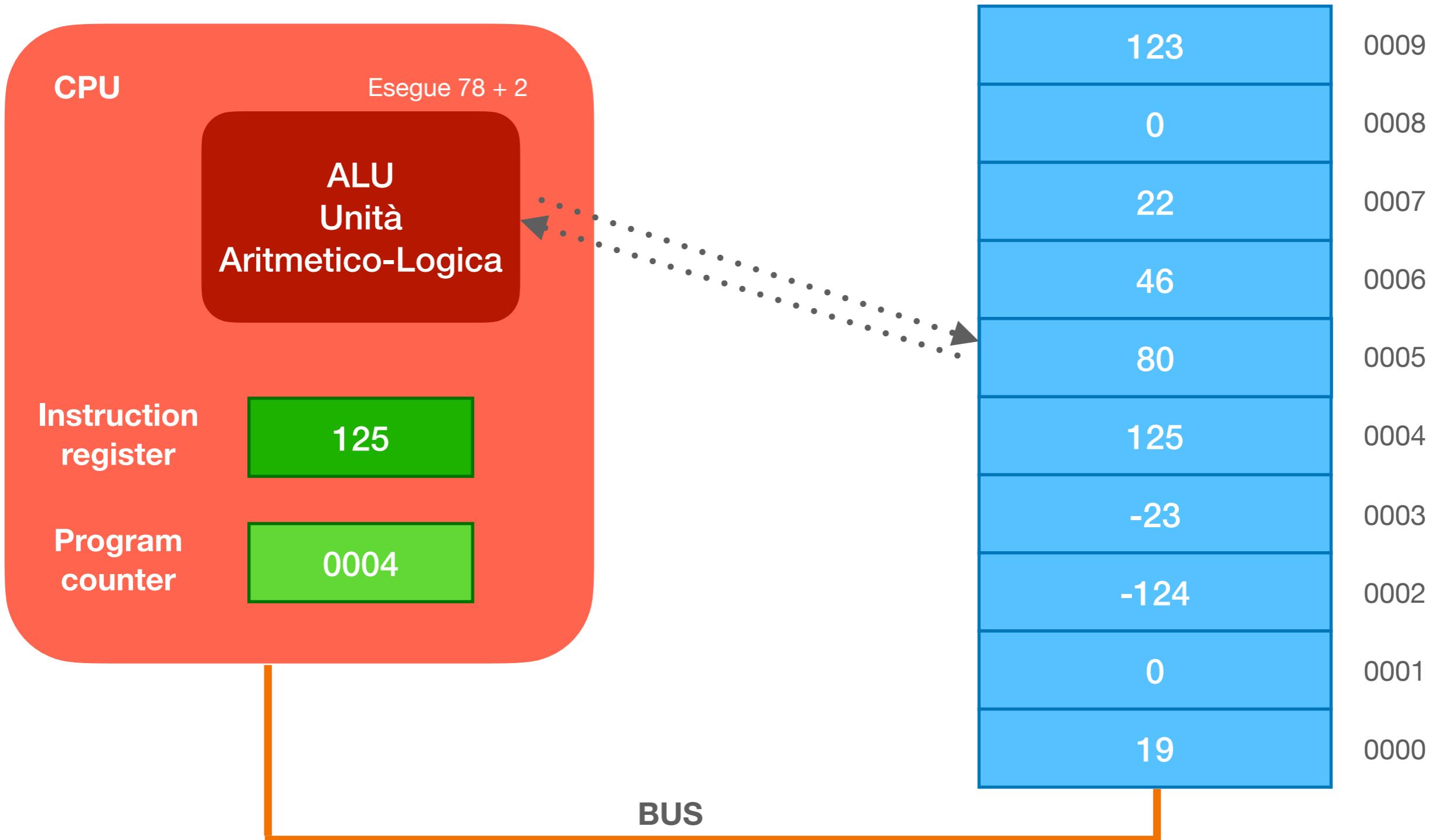
Decode

Cosa chiede l'istruzione?

- Il valore “125” salvato nell’instruction 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

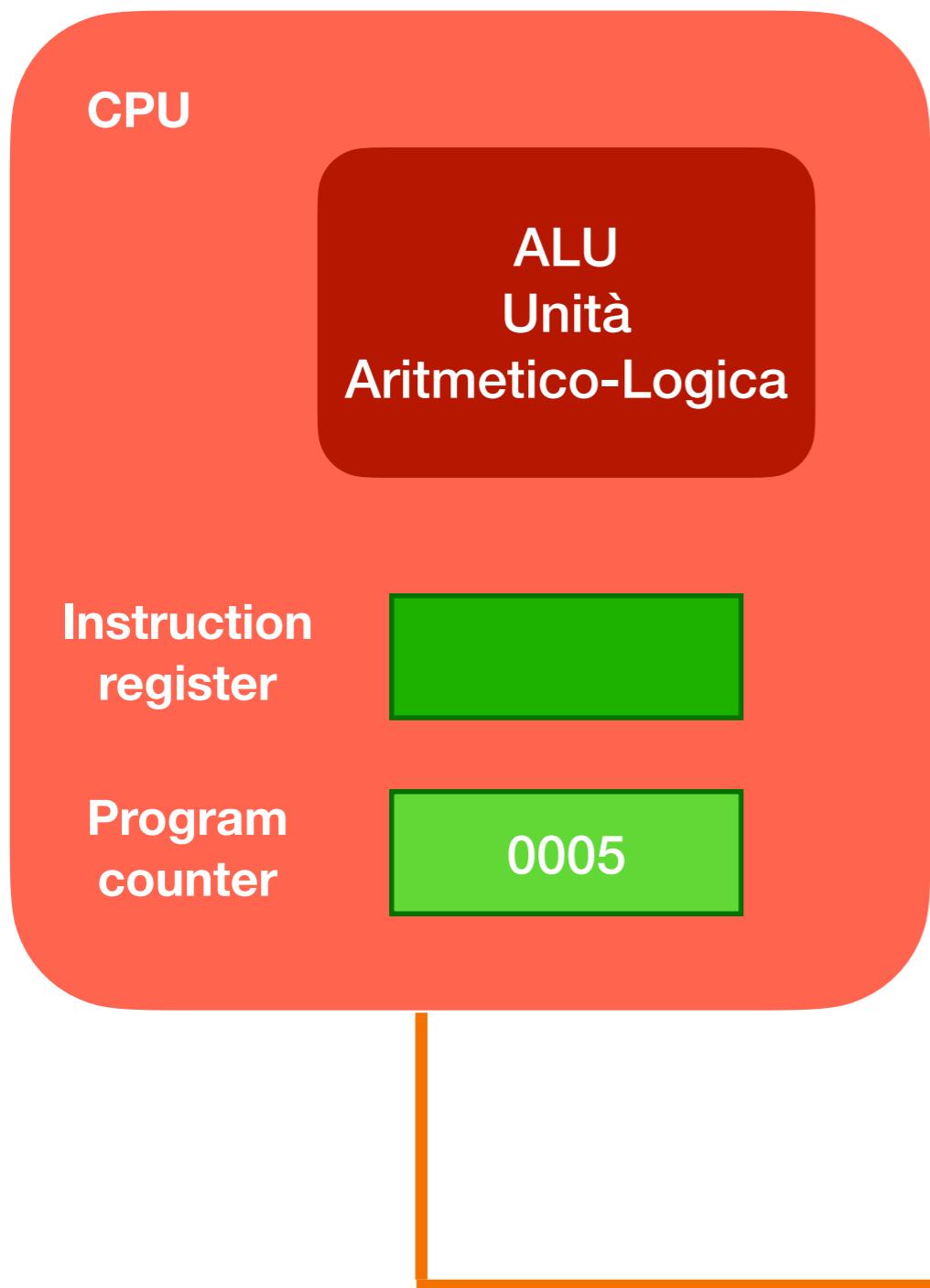
Execute

Eseguire l'operazione



Pronti a ripetere il ciclo?

Passare all'istruzione successiva



Il program counter viene incrementato e il ciclo di fetch, decode e execute continua

0009	123
0008	0
0007	22
0006	46
0005	80
0004	125
0003	-23
0002	-124
0001	0
0000	19

Un esempio pratico: ARM

La storia di ARM

Da Acorn Risc Machine a ARM

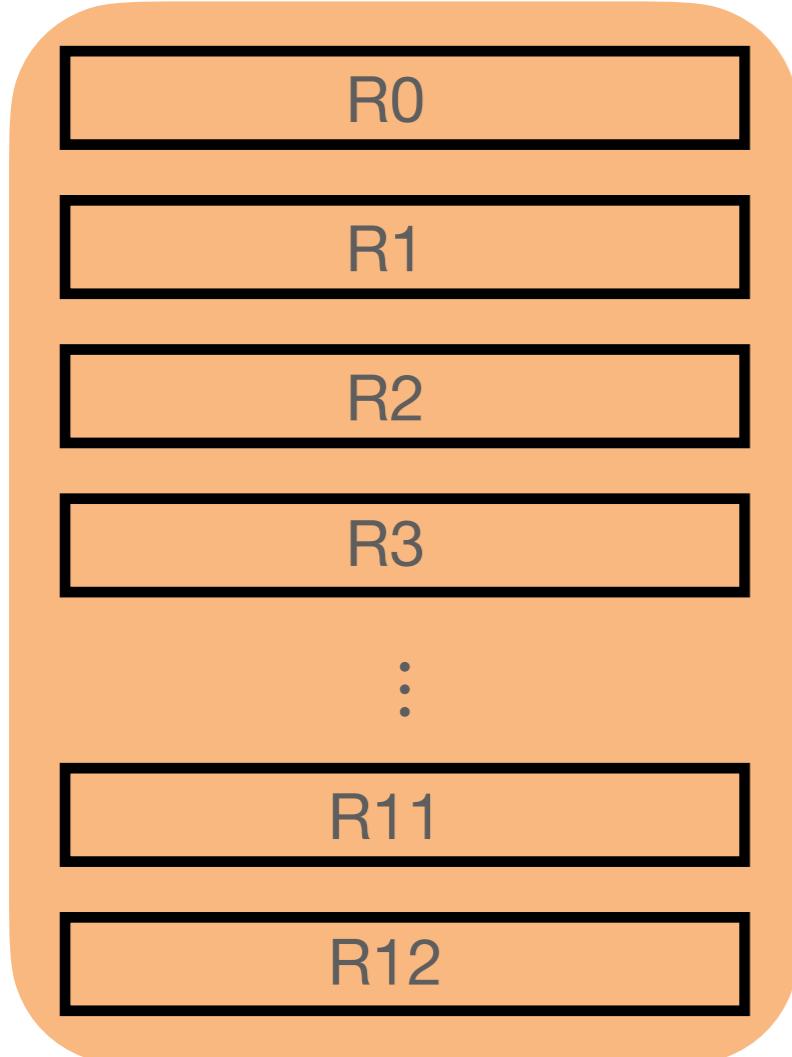
- L'architettura ARM nasce come prodotto della Acorn Computers, società britannica che aveva prodotto il BBC Micro nel 1981
- Dopo il BBC Micro e l'Acorn Electron, la Acorn decise di sviluppare una sua architettura di processori con il progetto *Acorn RISC Machine* (ARM). Nel 1985 l'ARM1 fu il primo processore prodotto
- Nel 1990 la progettazione dei processori fu separata in una diversa compagnia fondata da Acorn, Apple e VLSI Technology
- ARM attualmente non produce direttamente processori ma li progetta e vende i diritti di proprietà intellettuale
- Nel tempo il significato di ARM è passato a “Advanced Risc Machine” per poi essere, semplicemente, ARM

Arm 32 bit

Arm-v7a

- 13 registri per l'uso generale, ognuno di 32 bit
- 1 registro per lo stack pointer
- 1 link register (per le chiamate a subroutine)
- 1 program counter
- 1 program status register (PSR)
- Ogni istruzione richiede 32 bit
- Permette una modalità “ridotta” (Thumb mode) in cui ogni istruzione è a 16 bit (che non vedremo)

Arm: registri



Viene salvato l'indirizzo dello stack
(quello delle chiamate a funzione)

Link register viene usato per
salvare un indirizzo per alcune
operazioni

Contiene l'indirizzo
dell'istruzione da eseguire

Contiene diversi bit che
vengono impostati come
risultato di comparazioni,
operazioni aritmetiche, etc



Nessuno di questi registri ha un utilizzo specifico,
possiamo usarli come vogliamo*

* esistono delle convenzioni quando si lavora con altro codice, ma sono solo convenzioni

CISC vs RISC

CISC

Complex Instruction Set Computers

- Nell'approccio CISC si vuole completare un'operazione nel minor numero di operazioni in assembly
- Questo significa che, per esempio, è possibile nella stessa operazione accedere alla memoria, sommare un valore, scrivere il valore in memoria
- Generalmente le istruzioni richiedono più di un ciclo di clock per essere completate
- Generalmente le istruzioni sono di lunghezza variabile, sono molte e più difficili da decodificare

RISC

Reduced Instruction Set Computers

- Nell'approccio RISC si vuole mantenere basso il numero di istruzioni
- Ogni istruzione compie una cosa sola. Si hanno istruzioni separate per caricare dalla memoria in un registro, sommare un valore e salvare il risultato in memoria
- Si tende ad avere istruzioni che completano in pochi cicli di clock
- Le istruzioni sono di lunghezza fissata, rendendo la decodifica più semplice
- In realtà le architetture RISC sono andate complicandosi e quelle CISC hanno preso molte idee dai RISC

Istruzioni ARM

Assembly

Lavorare con un processore reale

- Le istruzioni, come abbiamo visto, sono codificate come sequenze di bit
- Questa rappresentazione testuale che ha una conversione diretta (o, comunque, semplice) alle istruzioni in codice macchina è detta, come avviamo visto, **assembly**
- Notate però come l'assembly dipenda dall'architettura del processore...
- ...se abbiamo più operazioni e più registri possibili ci aspettiamo di vedere istruzioni diverse

MOV

Spostare valori tra registri

L'istruzione MOV permette di copiare un valore da un registro a un altro o caricare un valore numerico (chiamato immediato) nel registro

Deve essere un registro (e.g., R0-R12)

MOV **dest**, **source**

Può essere:

- Un numero (e.g., #23, #0xAF)
- Un registro (e.g., R3)

Alcuni esempi di istruzione MOV:

MOV R0, #45

Mette il valore 45 nel registro R0

MOV R0, R2

Copia il valore contenuto nel registro R2 nel registro R0

Esercizio: scambiare il valore del registro R1 e R2 usando R0 come registro di appoggio

ADD

Sommare Valori

L'istruzione ADD somma i valori contenuti in due registri o il valore contenuto in un registro e un numero. Il risultato viene poi salvato in un registro



Alcuni esempi di istruzione ADD:

ADD R0, R0, #1

Incrementa il valore contenuto in R0 di 1

ADD R0, R1, R2

Mette in R0 il risultato della somma dei valori contenuti in R1 e R2

Esercizio: Mettere in R1 un valore a scelta. Usando istruzioni ADD mettere in R0 il triplo del valore contenuto in R1

SUB e altre istruzioni

ADC, SBC, RSB, RSC, AND, ORR, EOR, BIC, ORN

L'istruzione SUB sottrae i valori contenuti in due registri o il valore contenuto in un registro e un numero.
Il risultato viene poi salvato in un registro



Altre istruzioni con formato simile:

ADC **dest**, **op1**, **op2** ADD con bit di carry a 1

SBC **dest**, **op1**, **op2** SUB con bit di carry a 1

RSB **dest**, **op1**, **op2** Reverse SUB

RSC **dest**, **op1**, **op2** Reverse SUB con carry a 1

ORR **dest**, **op1**, **op2** OR dei bit di op1 e op2

EOR **dest**, **op1**, **op2** XOR dei bit di op1 e op2

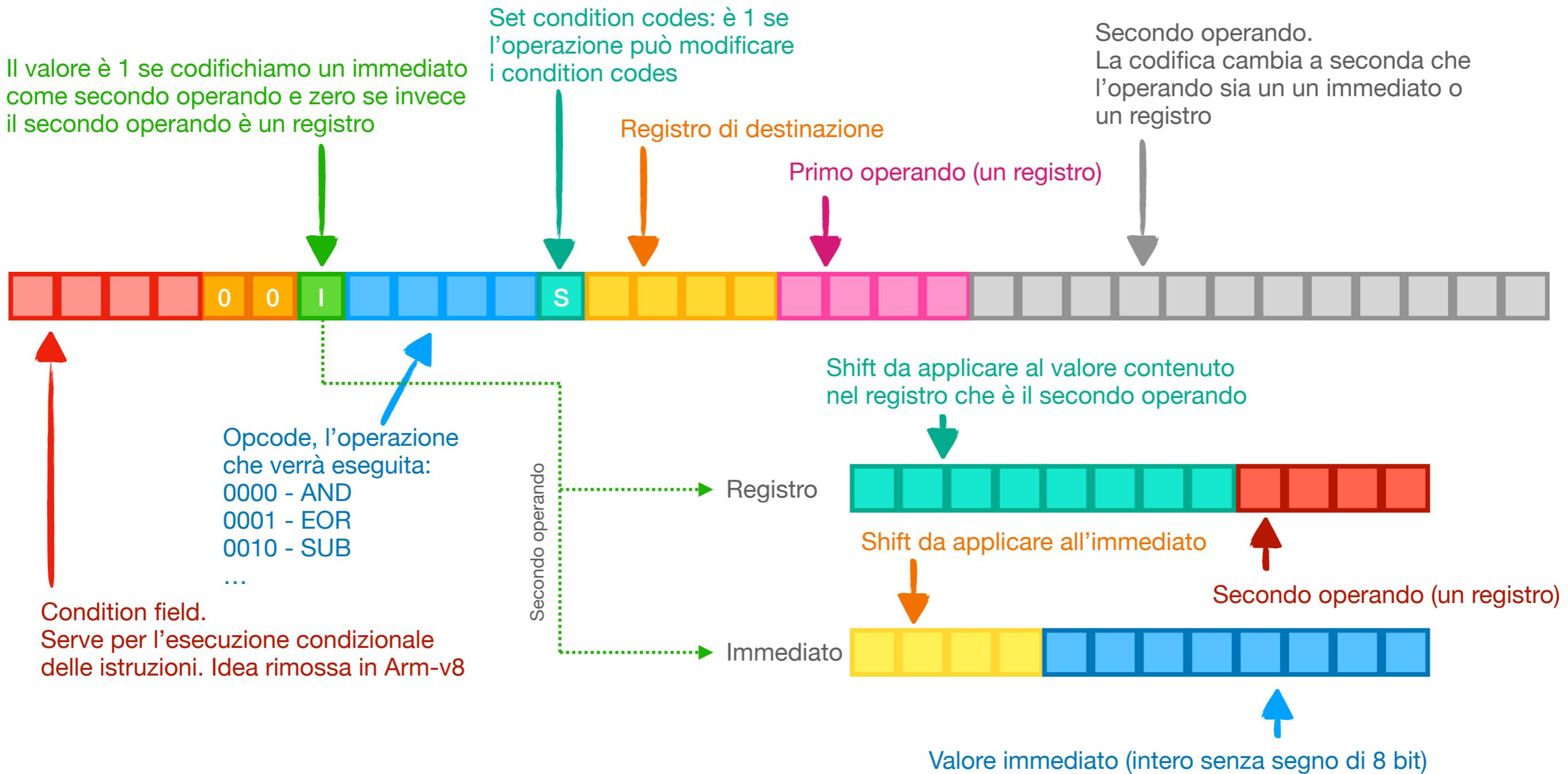
BIC **dest**, **op1**, **op2** Calcola op1 AND NOT op2

ORN **dest**, **op1**, **op2** Calcola op1 OR NOT op2

Codifica delle istruzioni

32 bit per istruzione

Abbiamo detto che ogni istruzione è codificata in 32 bit, vediamo un esempio per le istruzioni cosiddette di “Data Processing” (e.g., ADD, SUB, etc.)



Effettuare scelte

Istruzioni di salto (jump o branch)

- Fino ad ora abbiamo visto come fare operazioni aritmetiche o logiche
- Non possiamo ancora alterare il flusso di esecuzione delle istruzioni
- Per fare questo abbiamo le istruzioni di salto (o branch) che cambiano il valore del program counter. Distinguiamo due categorie:
 - Salto **non condizionato** avviene sempre
 - Salto **condizionato** avviene solo se alcune condizioni sono soddisfatte (si utilizza il program status register)

B, BL ed etichette

Branch e labels

L'istruzione di branch ci permette di saltare ad un'altra istruzione nel codice. Per indicare a quale istruzione dobbiamo saltare usiamo, in assembly, della label.

Deve essere una etichetta
(un nome che precede una istruzione)

B label

BL label

Branch with link

Funziona come branch ma salva il
valore del PC nel link register

etichetta

inizio ADD R0, R0, #1
B inizio

Continuerà a sommare 1 al valore
contenuto nel registro R0

Senza i salti condizionati e usando
solo istruzioni di branch
otterremmo solo cicli infiniti

Codifica:

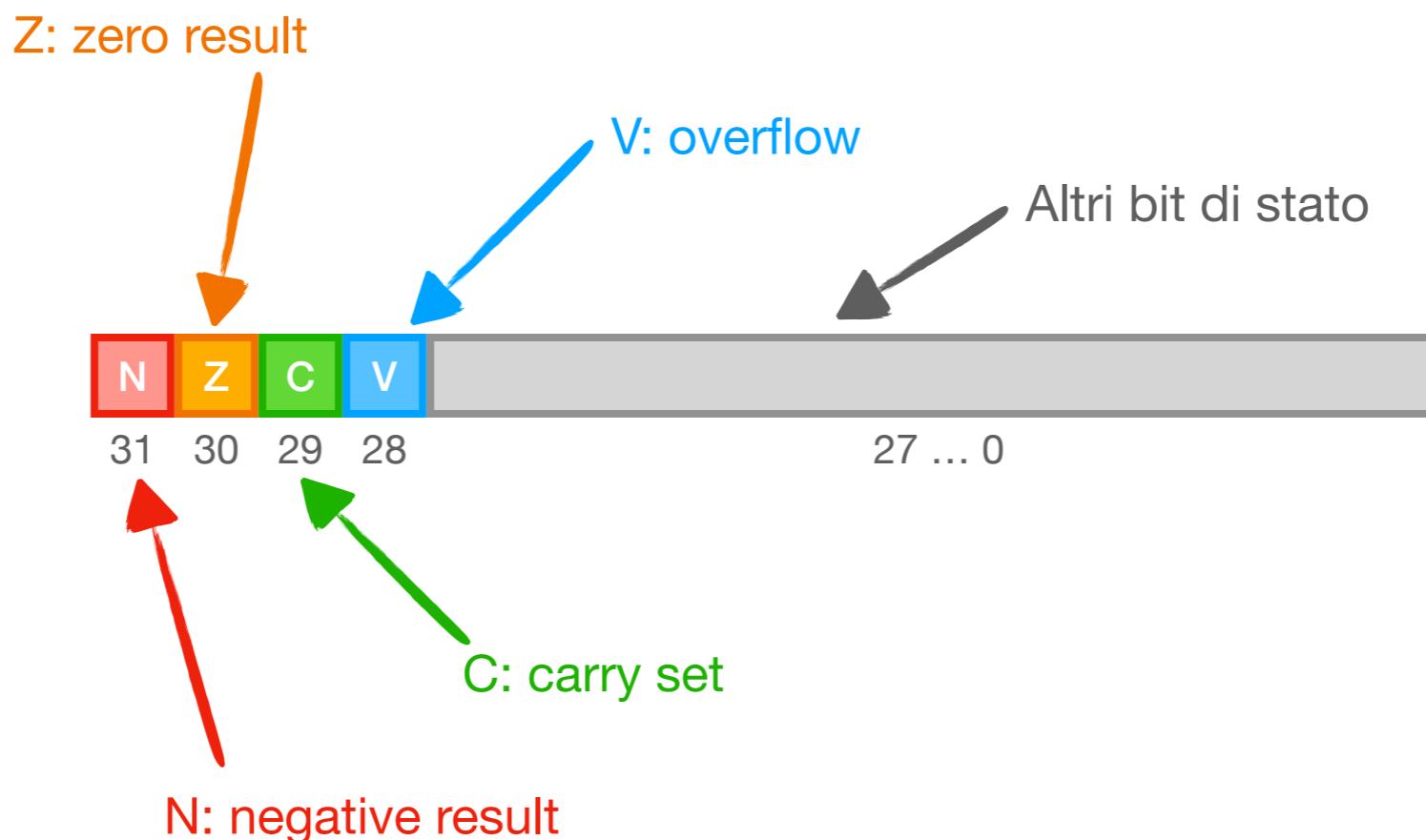
Link bit (distingue Branch da
Branch with link)



Offset: di quanto modificare il program counter
(questo valore viene calcolato dall'assembler)

Program Status Register (PSR)

Alcuni dei bit che ci interessano



Questi bit sono aggiornati quando facciamo una comparazione e diverse configurazioni di questi bit ci forniscono informazioni sul risultato della comparazione

CMP

Comparare valori

L'istruzione CMP ci permette di comparare il valore contenuto in un registro con il valore contenuto in un altro registro o con un immediato. Il risultato della comparazione altererà il PSR

Deve essere un registro (e.g., R0-R12)

Può essere:

- Un numero (e.g., #23, #0xAF)
- Un registro (e.g., R3)

CMP op1, op2

Alcuni esempi di istruzione CMP:

CMP R0, #45

Compara il valore contenuto nel
registro R0 con il valore 45

CMP R0, R2

Compara il valore contenuto nel registro R0
con quello contenuto nel registro R1

Una operazione CMP cambia solo i valori contenuti nel PSR

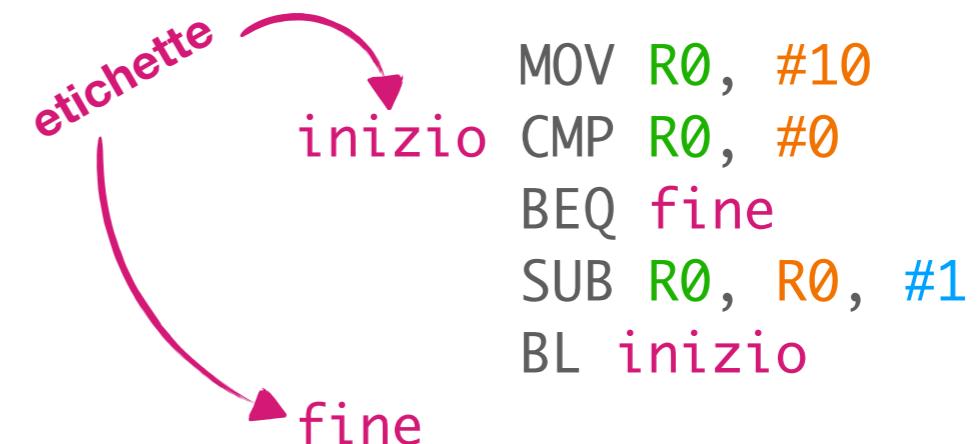
Branch condizionale

BEQ

L'istruzione di branch condizionale ci permette di saltare ad un'altra istruzione nel codice.
Con BEQ questo accade solo se l'ultima comparazione (con CMP) era tra due valori identici

Deve essere una etichetta
(un nome che precede una istruzione)

BEQ **label**



Sottrae uno al valore contenuto in R0
Continua a eseguire l'operazione a meno che
il valore in R0 non sia 0

Codifica:

Cambiano solo i bit nel condition field
(rispetto al normale branch)



Altri tipi di branch condizionale

BNE, BHS, BLO, ...

A seconda dei bit nel condition field si catturano diverse condizioni, corrispondenti a diverse istruzioni in Assembly

	BNE <code>label</code>	I due valori comparati erano diversi
Senza segno	BHS <code>label</code>	Il primo valore era \geq del secondo (considerati senza segno)
	BLO <code>label</code>	Il primo valore era $<$ del secondo (considerati senza segno)
	BHI <code>label</code>	Il primo valore era $>$ del secondo (considerati senza segno)
	BLS <code>label</code>	Il primo valore era \leq del secondo (considerati senza segno)
Con segno	BGE <code>label</code>	Il primo valore era \geq del secondo (considerati con segno)
	BLT <code>label</code>	Il primo valore era $<$ del secondo (considerati con segno)
	BHT <code>label</code>	Il primo valore era $>$ del secondo (considerati con segno)
	BLE <code>label</code>	Il primo valore era \leq del secondo (considerati con segno)

Architetture Load-Store

Accedere alla memoria di lavoro

- Se nonate nessuna delle operazioni precedenti utilizza direttamente sulla memoria di lavoro
- Sono solo operazioni tra registri
- Se vogliamo fare operazioni su valori salvati nella memoria di lavoro dobbiamo prima copiare il valore in un registro
- ARM utilizza operazioni esplicite per interagire con la memoria:
 - LOAD per caricare dalla memoria in un registro
 - STORE per salvare il valore di un registro in memoria

Architetture Load-Store

Indirizzamento

- Per accedere alla memoria ci serve un indirizzo
- Se codificassimo l'indirizzo direttamente nell'istruzione potremmo accedere solo a una frazione della memoria (e.g., se avessimo 24 bit “liberi” per l'indirizzo potremmo accedere solo a 2^{24} diverse locazioni di memoria, solitamente byte)
- Di solito si salva l'indirizzo di memoria in un registro (quindi 32 bit possibili) e si accedere all'indirizzo indicato da quel registro:
 - e.g., “Leggi in R0 il valore contenuto all'indirizzo di memoria contenuto in R1”

LOAD

Caricare valori dalla memoria

L'istruzione LDR ci permette di caricare in un registro un valore in memoria

Dato che le operazioni come ADD, SUB, ... lavorano solo su registri questo è essenziale

Deve essere un registro (e.g., R0-R12)

Un registro che contiene l'indirizzo da cui prendere il valore

LDR dest, [addr]

Alcuni esempi di istruzione LDR:

LDR R0, [R1]

Carica in R0 il valore contenuto all'indirizzo di memoria che è il valore di R1

MOV R0, #0x100

LDR R1, [R0]

Carica in R1 il valore contenuto nell'indirizzo di memoria 0x100 (indicato in esadecimale)

Esercizio: Caricare in R1 il valore contenuto all'indirizzo di memoria 0x200

STORE

Salvare i valori in memoria

L'istruzione STR ci permette di salvare in memoria il valore contenuto in un registro.

Deve essere un registro (e.g., R0-R12)

Un registro che contiene l'indirizzo in cui salvare il valore contenuto in src

STR src, [addr]

Alcuni esempi di istruzione STR:

SDR R0, [R1]

Salva il valore di R0 nell'indirizzo di memoria che è il valore di R1

MOV R0, #0x100

STR R1, [R0]

Salva all'indirizzo 0x100 il valore contenuto nel registro R1

Esercizio: Salvare all'indirizzo di memoria 0x200 il risultato di 12+4

Esercizio

Fare il lavoro del compilatore

Facciamo il lavoro di un compilatore: proviamo a convertire questo ciclo in del codice assembly in modo “plausibile”

```
int x = 5;  
int y = 10;  
while (y > 0) {  
    x = x + y + 3;  
    y = y - 1;  
}
```

Due varianti dell'esercizio:

- x e y sono valori nei registri (e.g., R0 e R1)
- x e y sono da salvare alle locazioni di memoria 0x100 e 0x104

Notate che dobbiamo spezzare questa operazione di somma in due operazioni distinte!

Chiamate a funzione

Chiamate a funzione

Capire come è possibile fare chiamate a funzione

- Come viene convertita una chiamata di funzione in assembly?
- Dobbiamo prima decidere come fare per:
 - Passare gli argomenti
 - Salvare cosa stavamo eseguendo quando chiamiamo una funzione
 - Come salvare lo stato della funzione (e.g., variabili locali)
 - Come ritornare a quello che stavamo eseguendo quando la funzione termina di eseguire

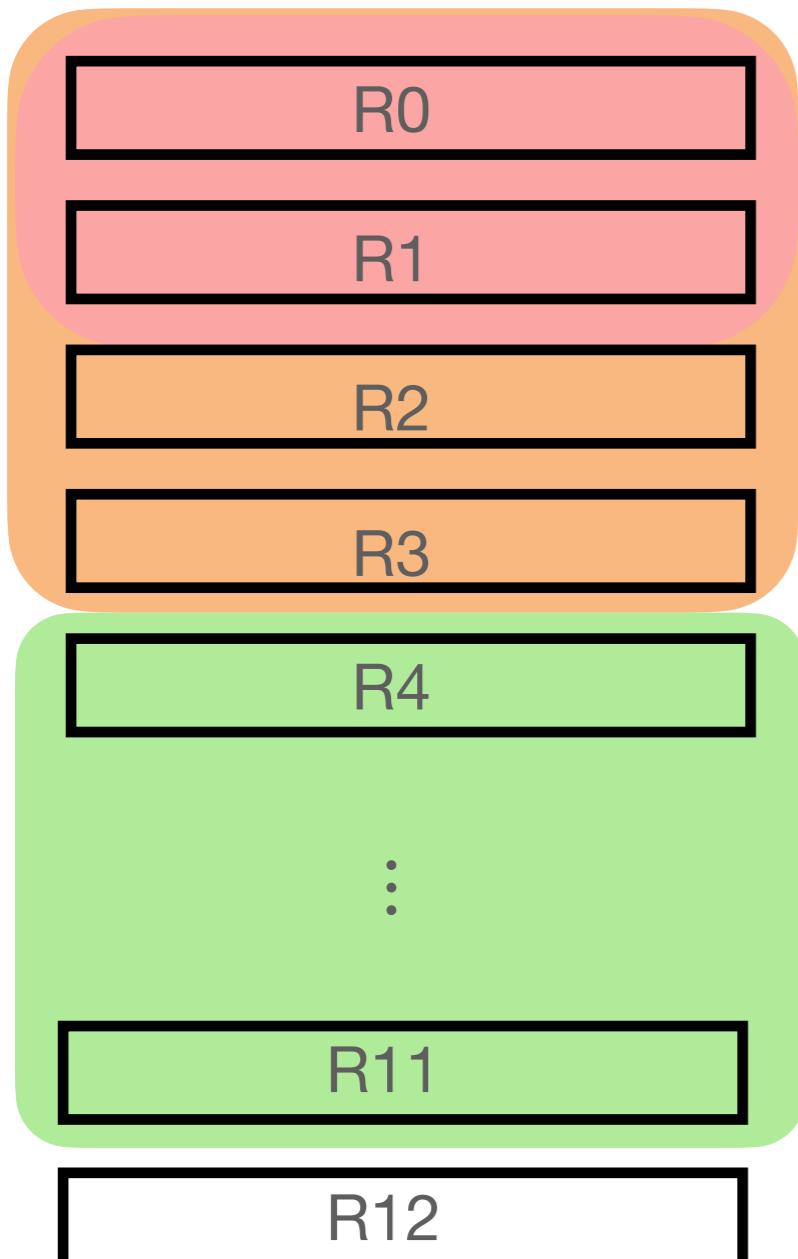
Chiamate a funzione

Capire come è possibile fare chiamate a funzione

- Buona parte di queste funzionalità è fornita dallo stack
- Lo stack inizia ad un indirizzo di memoria fissato (a volte chiamato “top of the stack”)
- Lo stack pointer (registro R13) indica la prima posizione libera sullo stack (o l’ultima occupata, dipende dalla convenzione)
- Ogni volta che vogliamo salvare qualcosa (e.g., variabili locali) effettuiamo uno store alla locazione indicata dallo stack pointer e muoviamo lo stack pointer
- Possiamo far crescere lo stack verso indirizzi più bassi (di solito è così) o più alti

Convenzioni dei registri

Registri usati per il passaggio
dei primi quattro argomenti

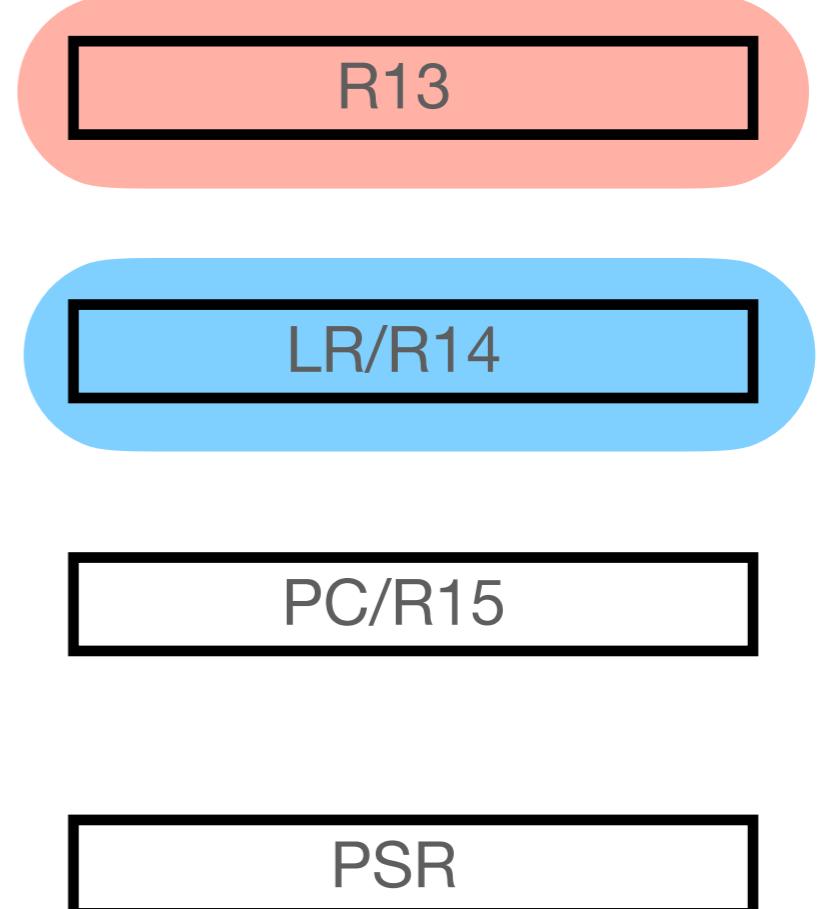


Registri in cui viene salvato il
valore di ritorno della funzione

Usando BL qui verrà salvato
il valore del PC a cui
dobbiamo ritornare una volta
terminata la funzione

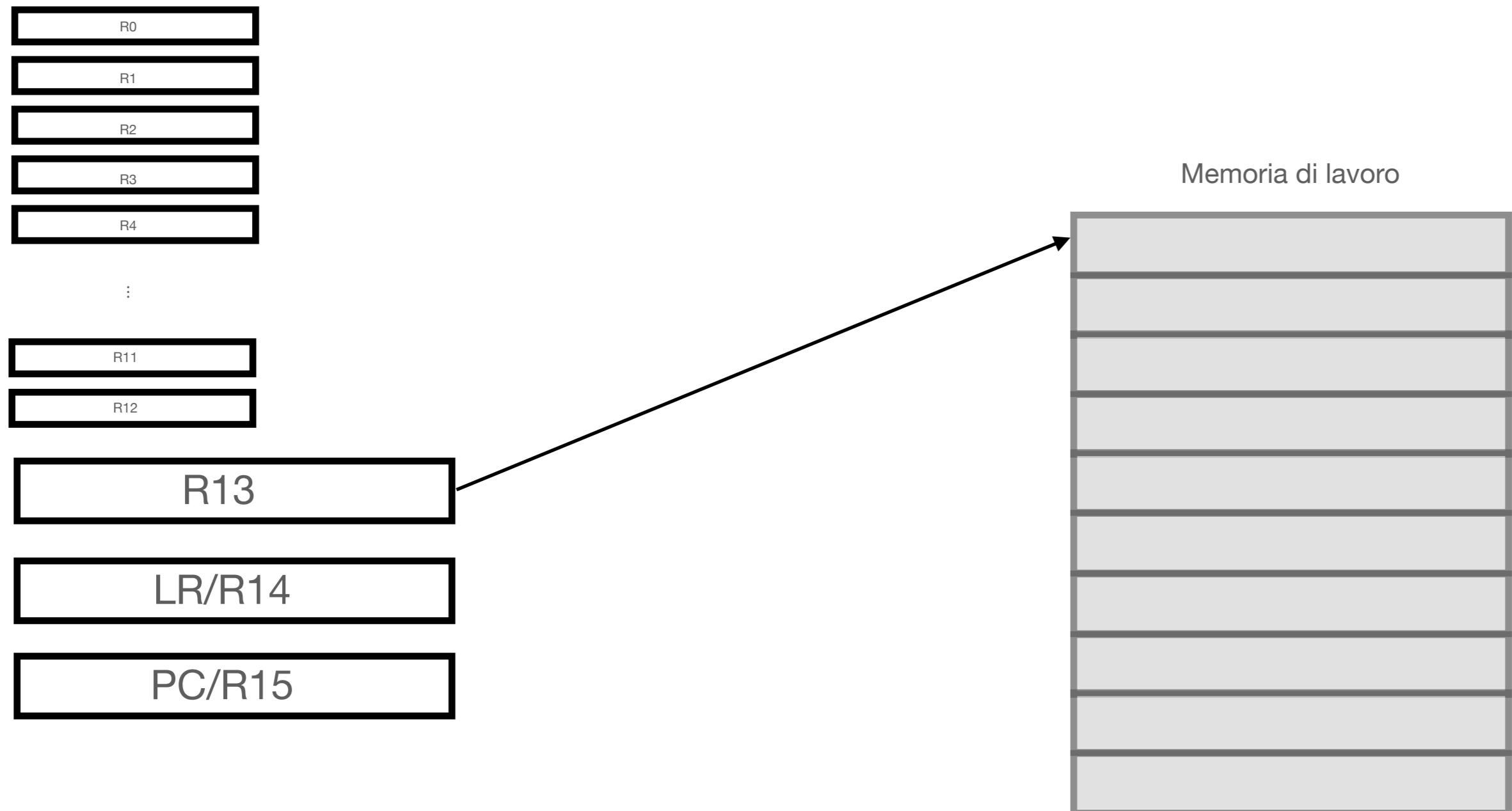
Registri salvati dal chiamante.
Se la funzione *chiamante* vuole
preservare il valore di questi
registri deve salvarli in
memoria e ripristinarli al ritorno
della funzione chiamata

Indirizzo dello stack

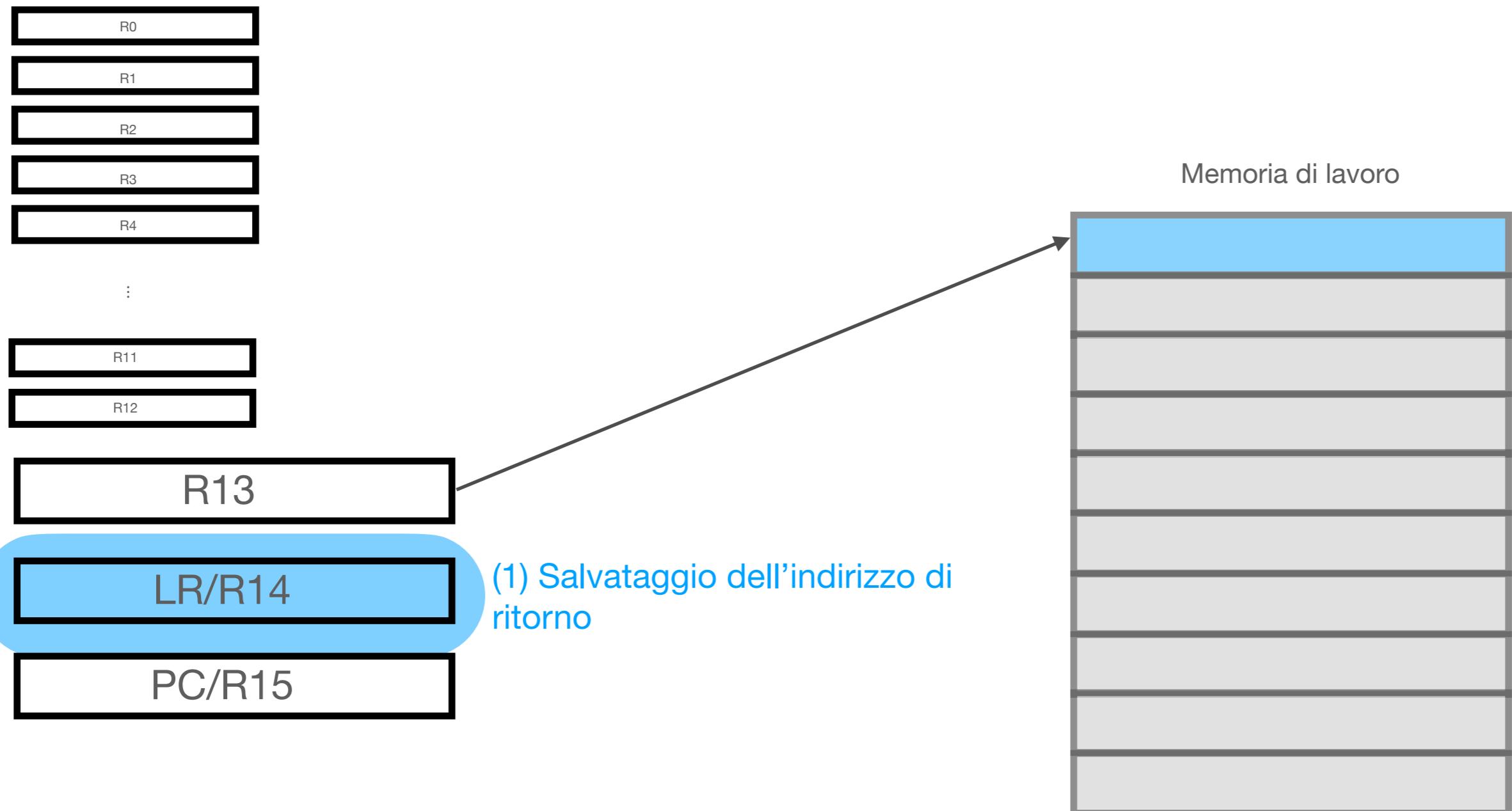


*Gli argomenti successivi devono essere posizionati nello stack

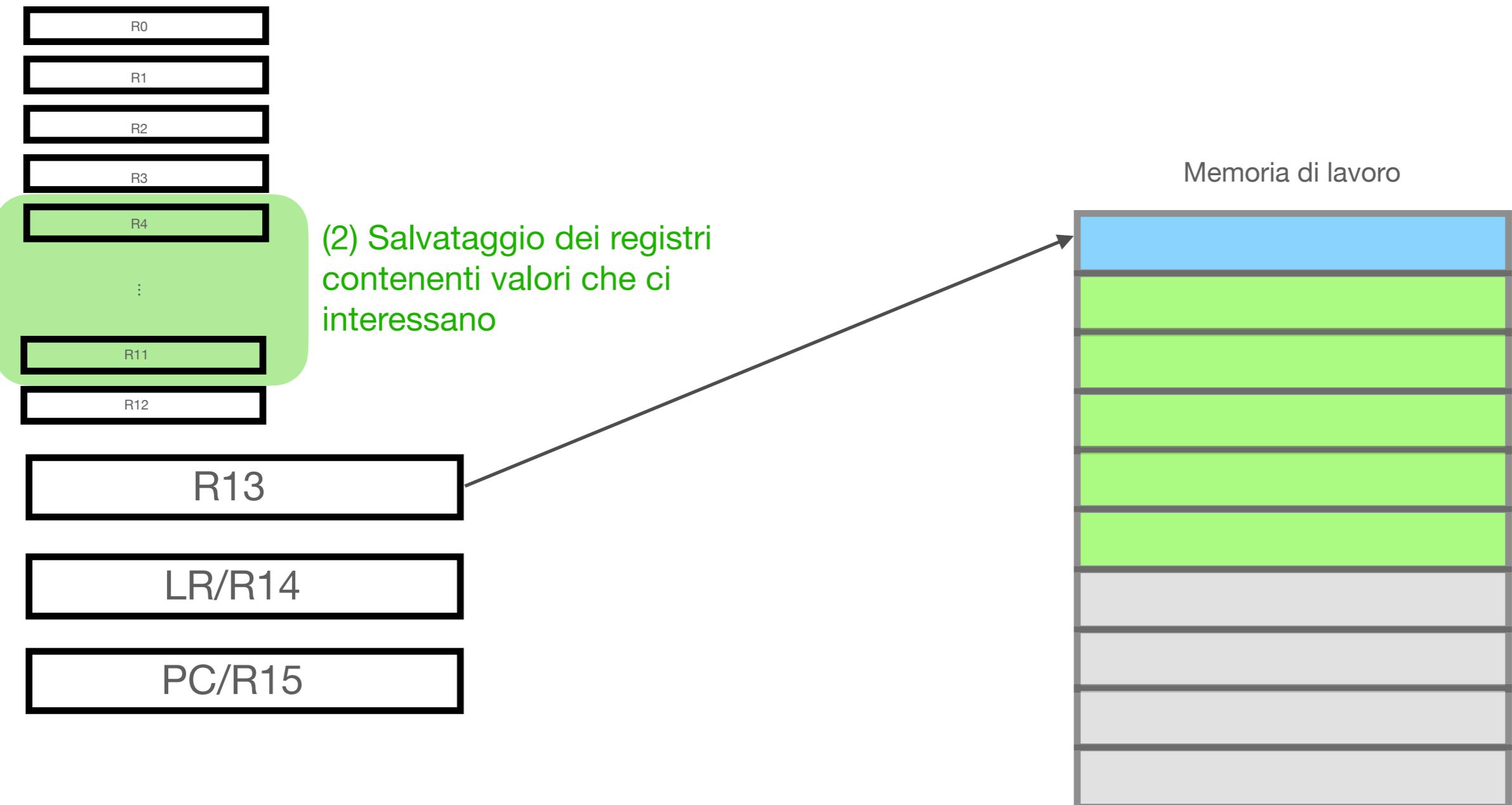
Cosa fare quando si chiama una funzione



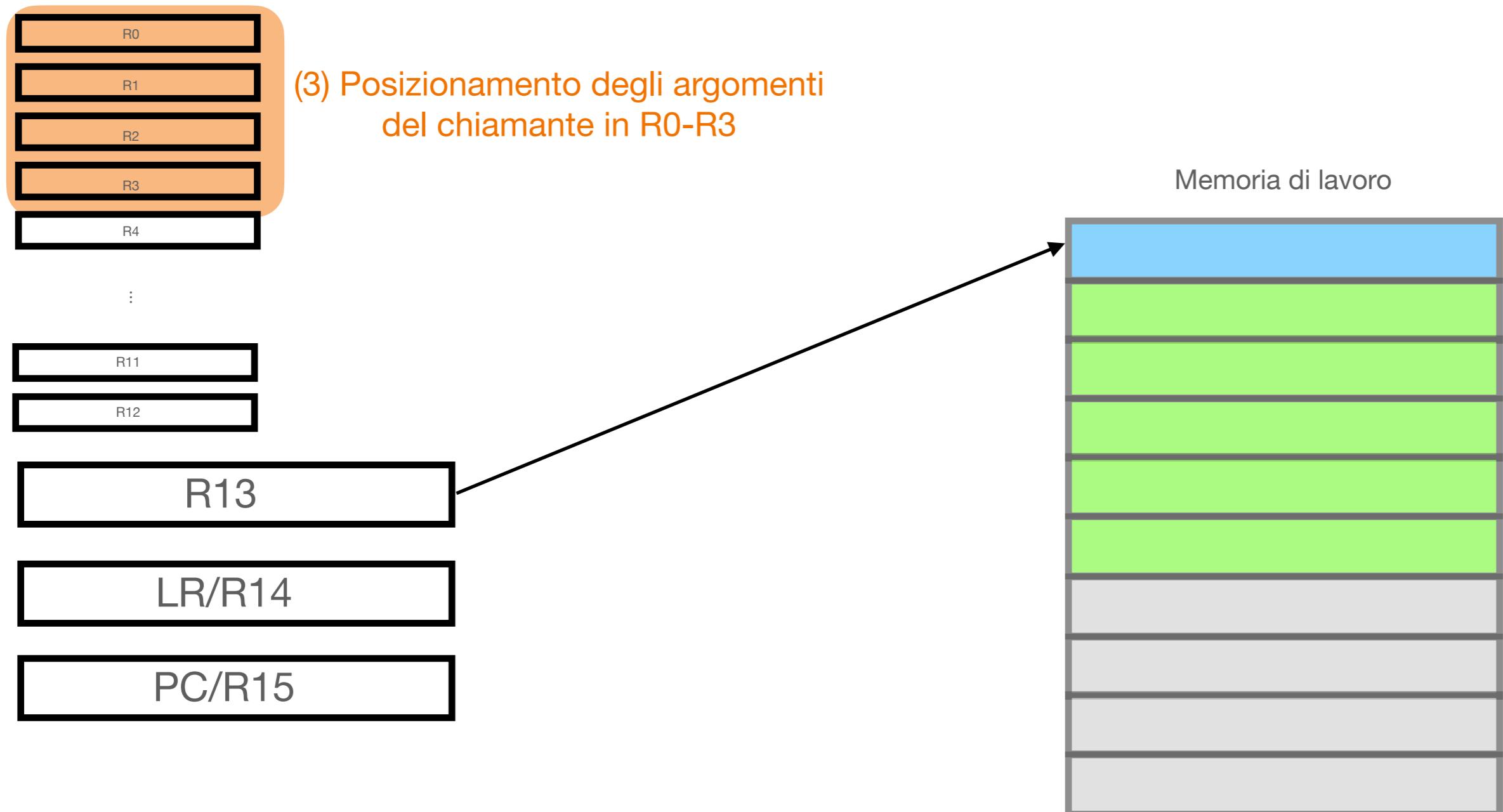
Cosa fare quando si chiama una funzione



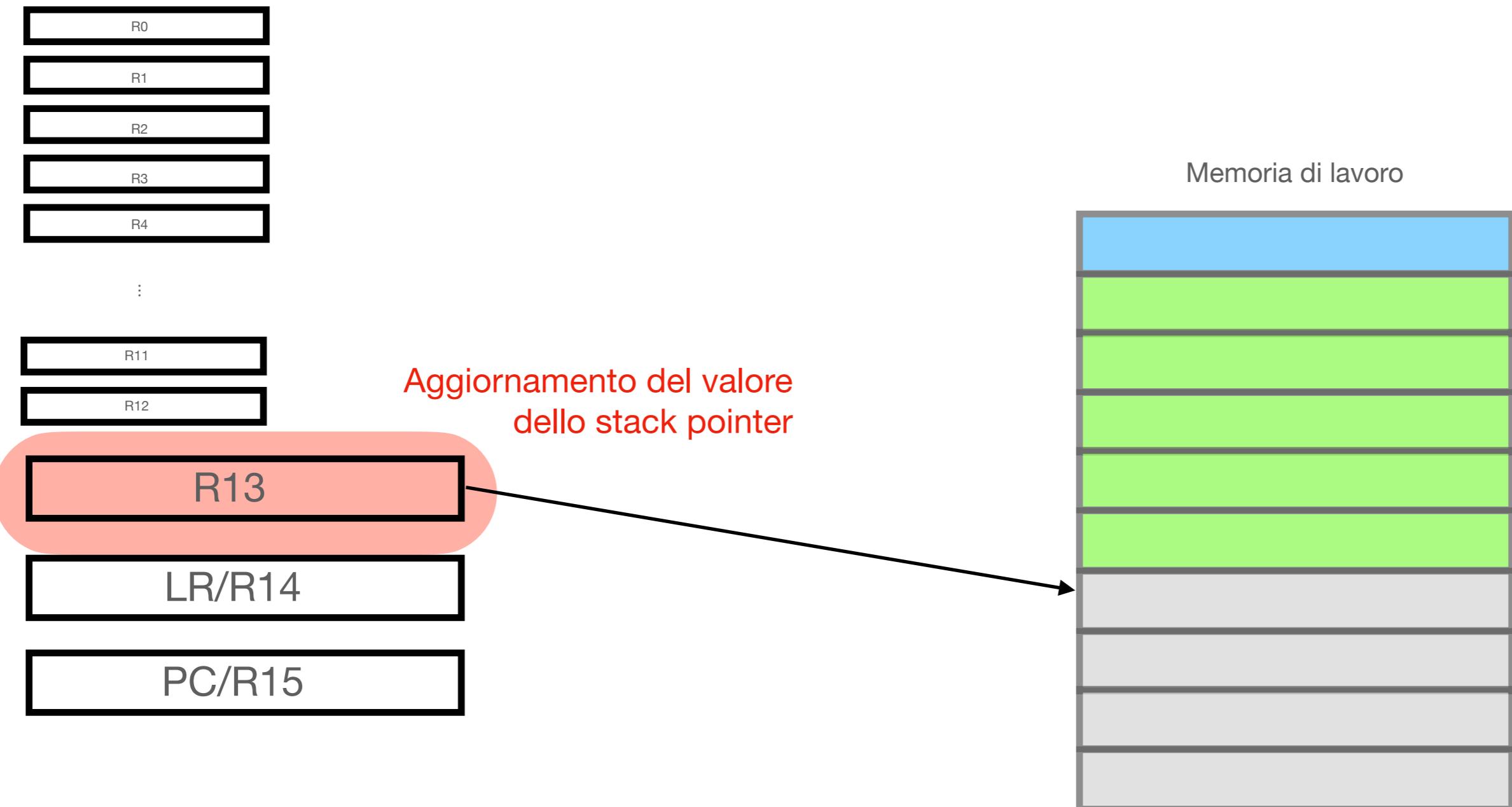
Cosa fare quando si chiama una funzione



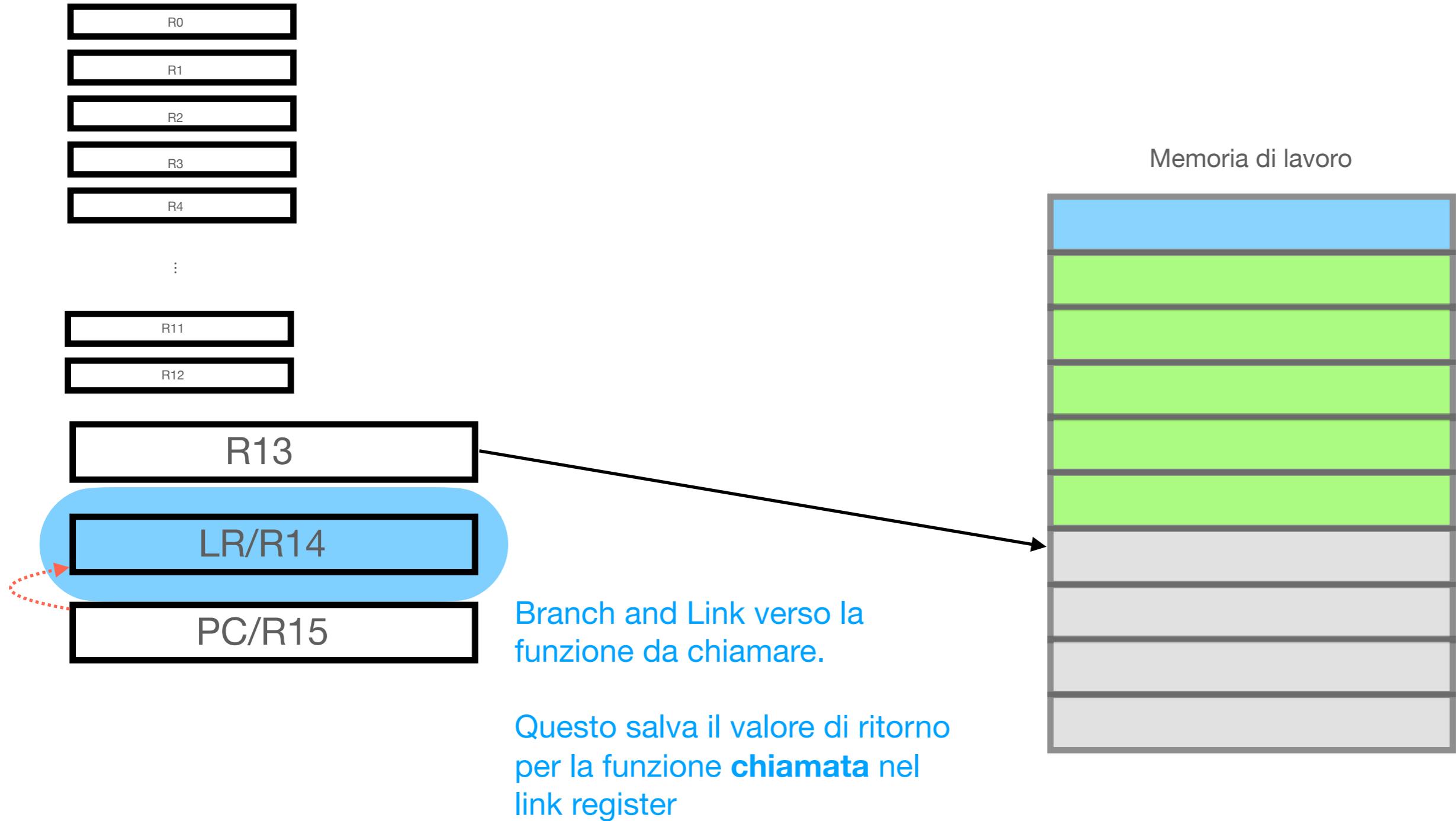
Cosa fare quando si chiama una funzione



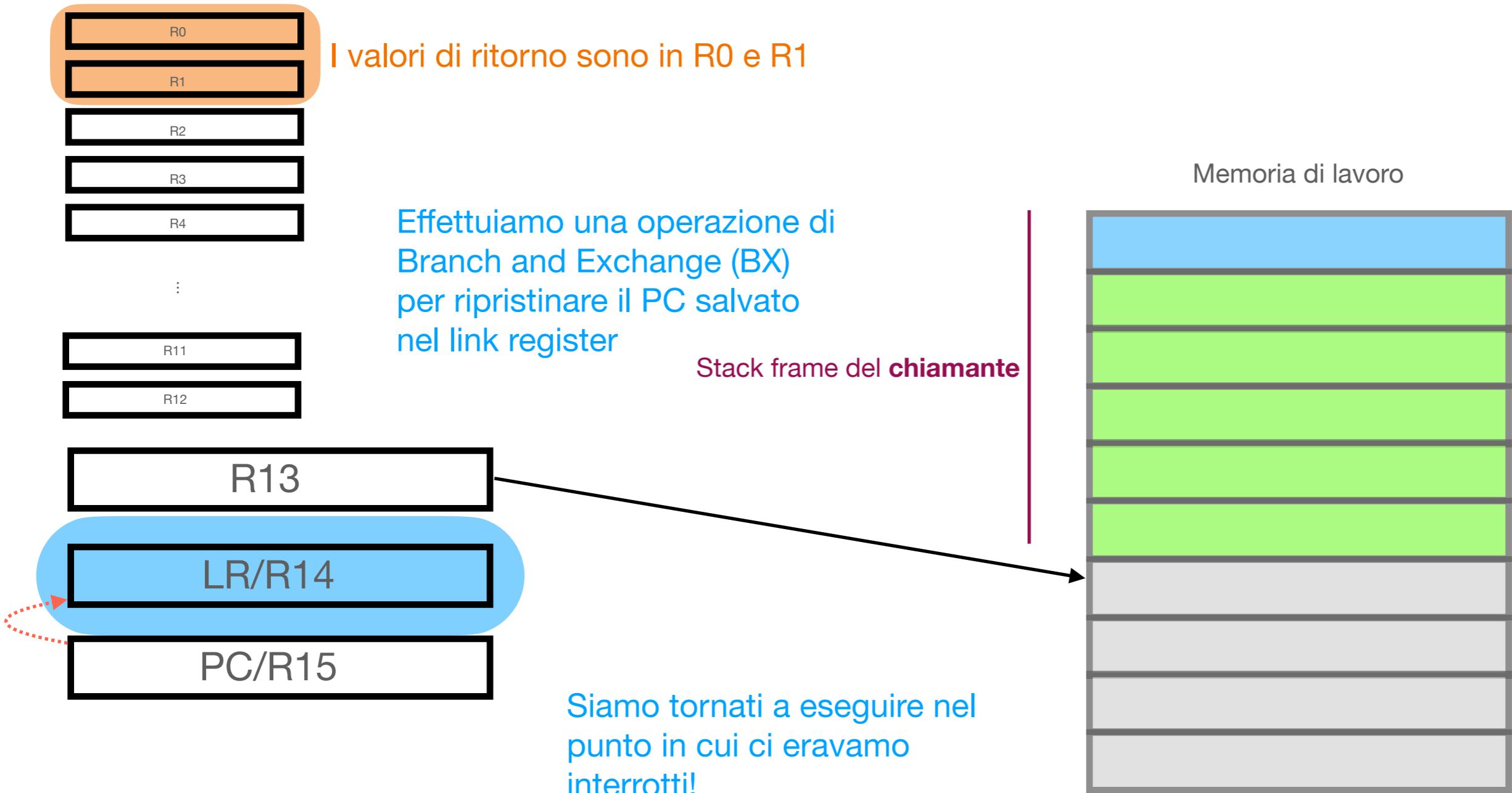
Cosa fare quando si chiama una funzione



Cosa fare quando si chiama una funzione

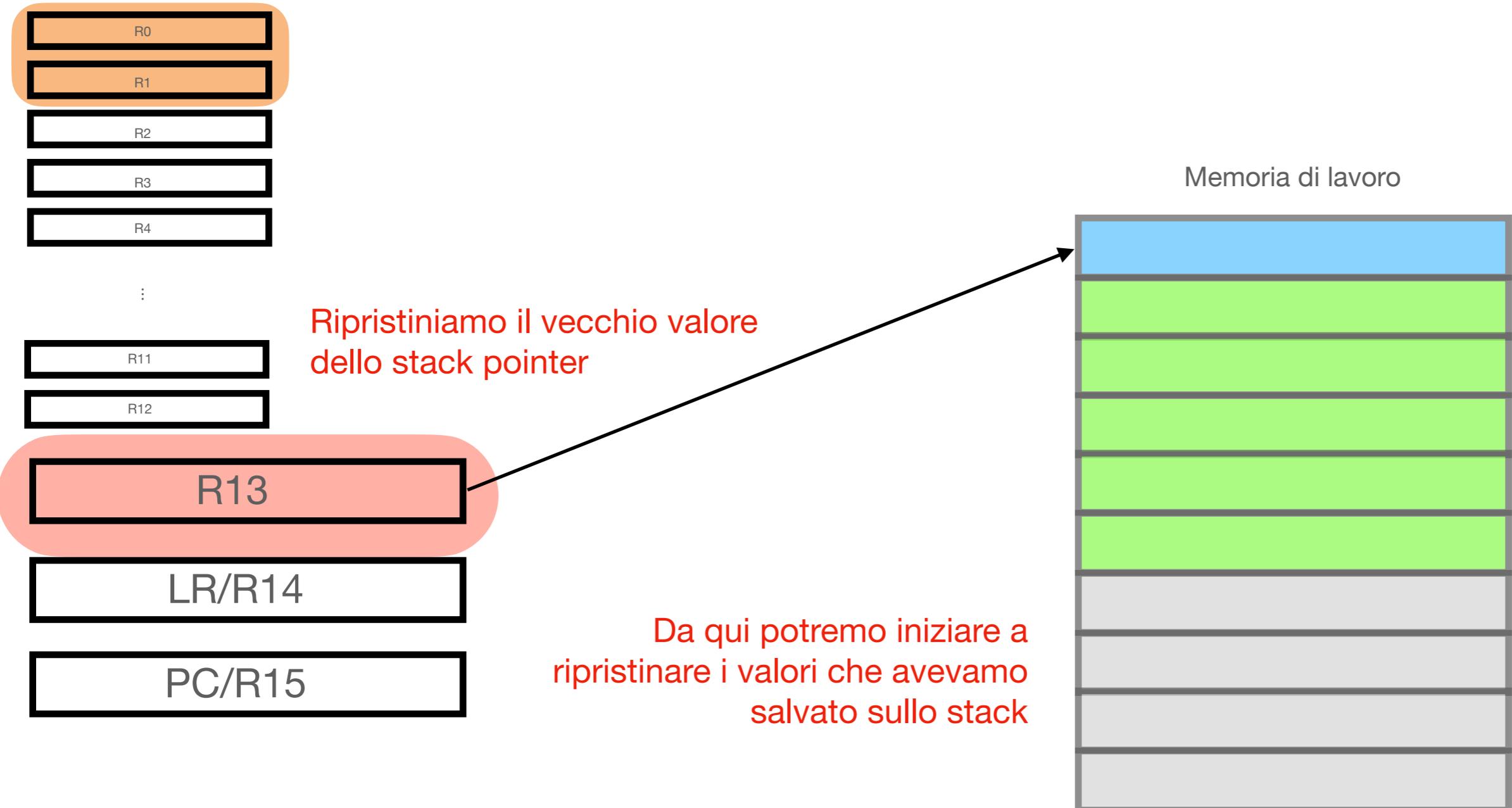


Cosa fare ritornando da una funzione

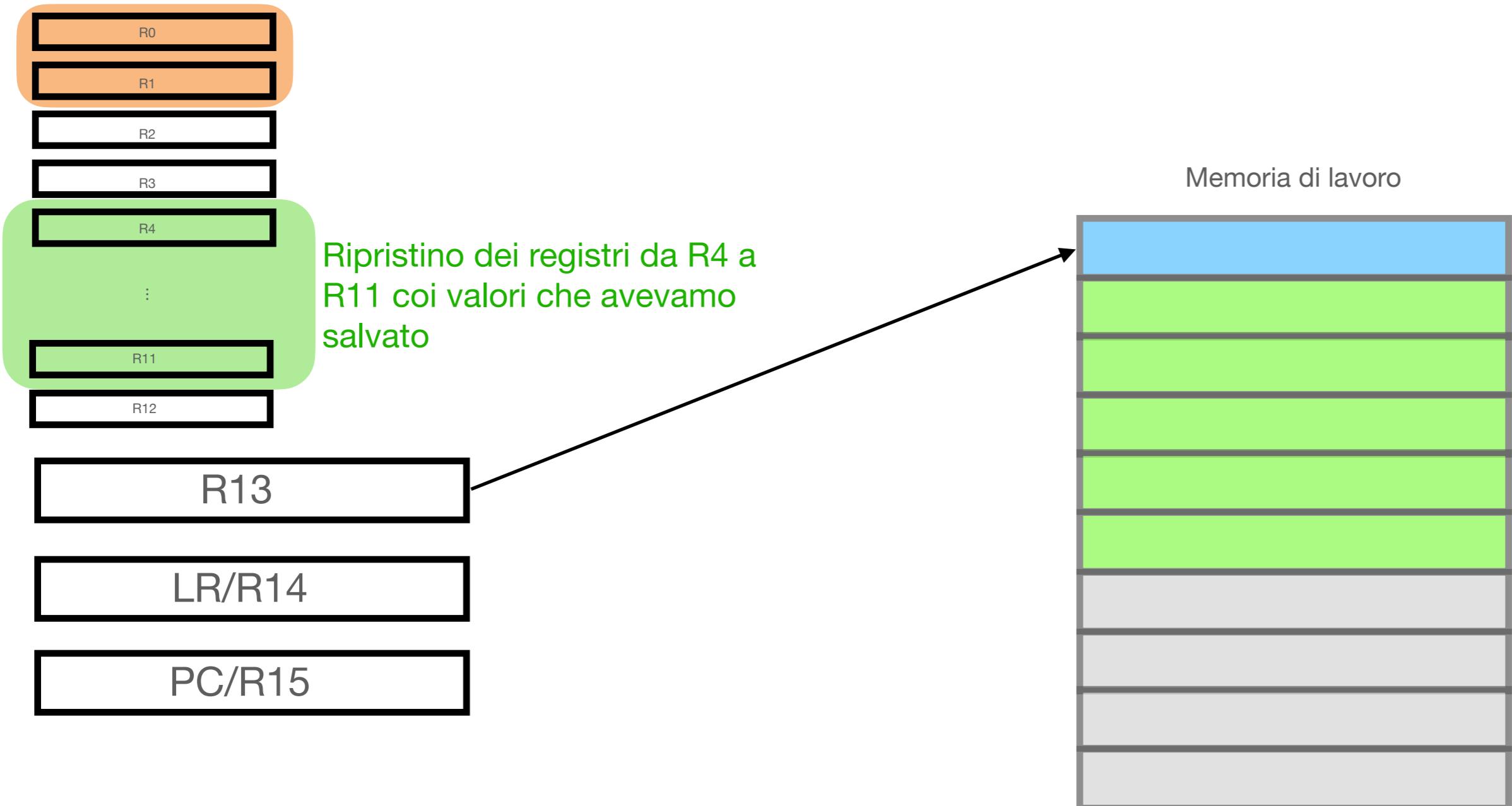


Nota: branch and exchange non è supportato da VISUAL. Possiamo usare MOV R14, R15

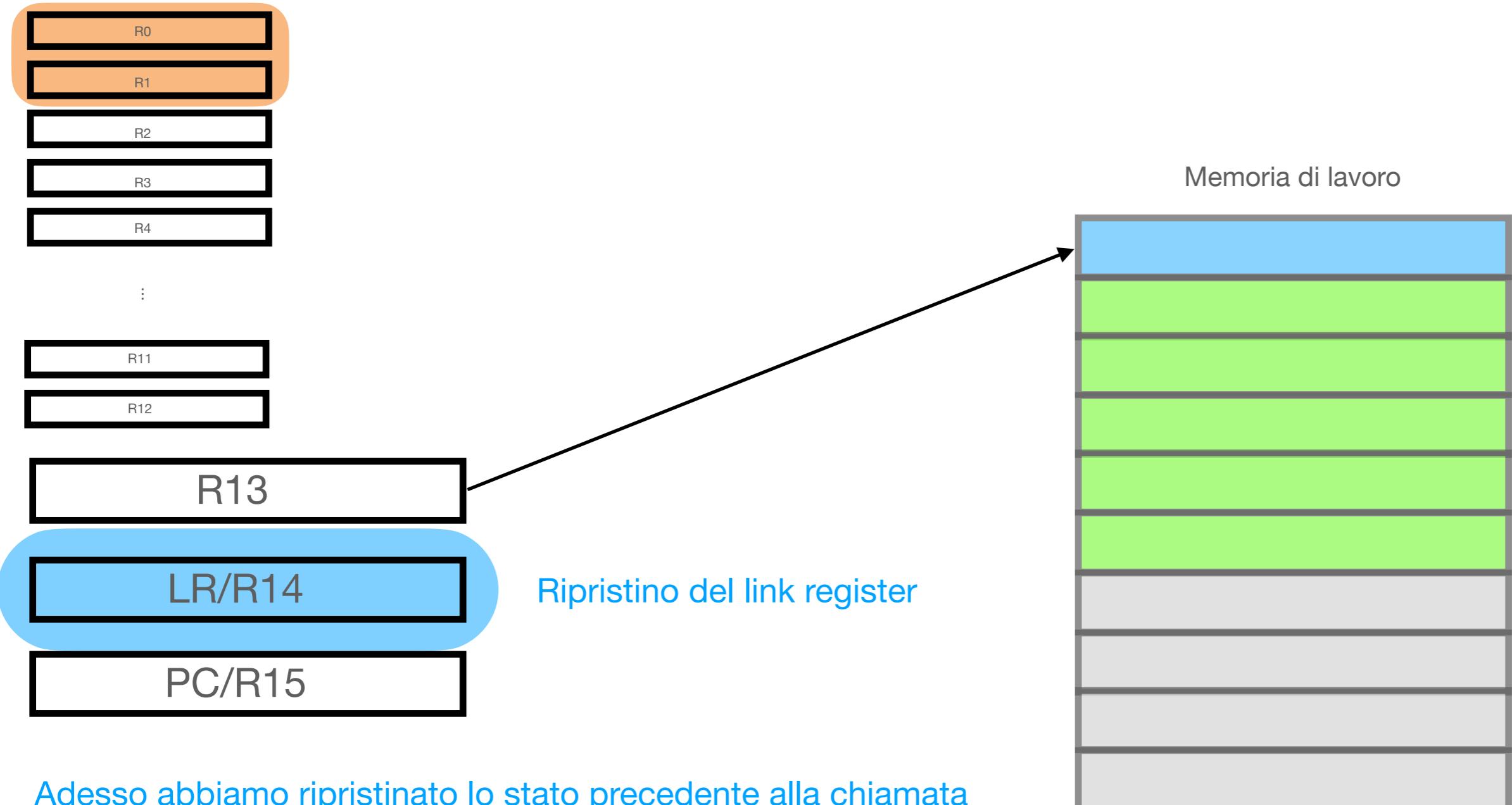
Cosa fare ritornando da una funzione



Cosa fare ritornando da una funzione



Cosa fare ritornando da una funzione



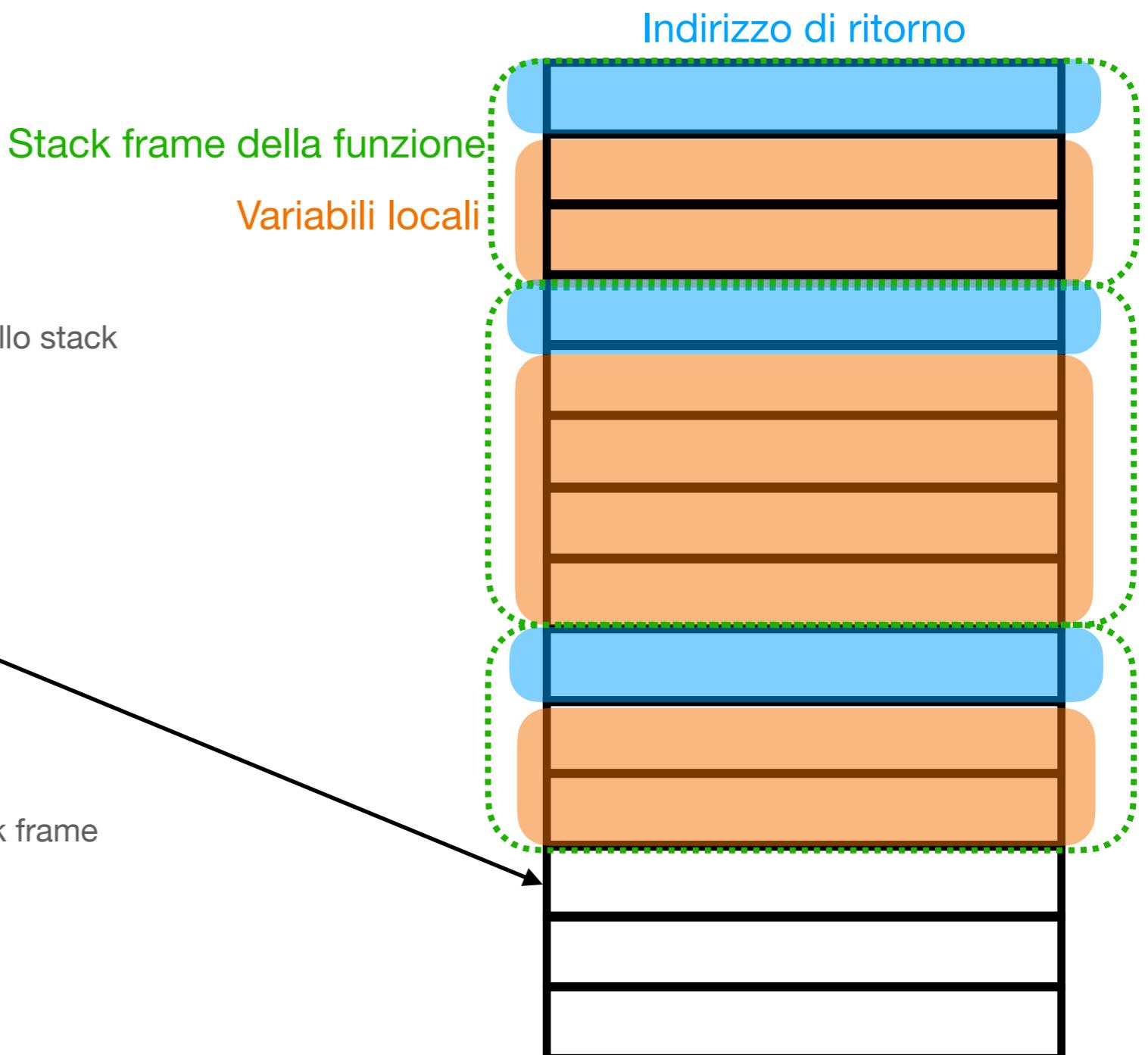
Stack: chiamate innestate

Qui abbiamo quattro chiamate a funzione.

Possiamo osservare i vari stack frame presenti sullo stack

R13

Lo stack pointer indica dove va a iniziare lo stack frame della funzione attualmente in esecuzione



Comunicare con altri dispositivi

Comunicare con l'esterno

Gestione dell'I/O

- Fino ad ora abbiamo visto come far eseguire istruzioni al processore e come salvare e caricare dati dalla memoria
- Non abbiamo visto cosa accade se vogliamo comunicare con altri dispositivi:
 - Come possiamo accedere alla tastiera per ottenere dell'input?
 - Come possiamo inviare dell'output su schermo?
 - Come possiamo accedere ai dispositivi di memorizzazione di massa?

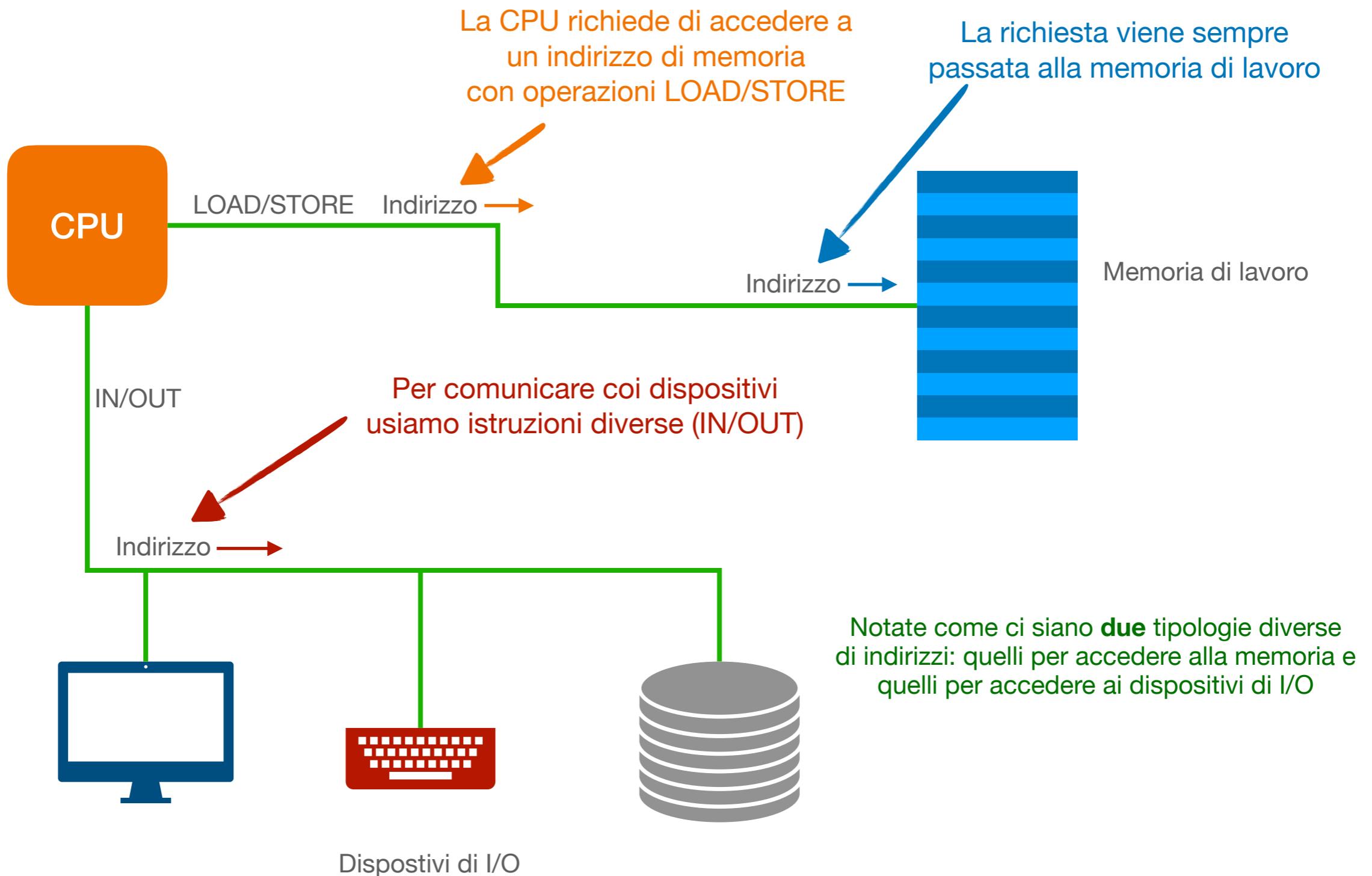
Port mapped I/O

PMIO

- Una prima idea è quella di utilizzare istruzioni speciali per comunicare con i dispositivi di input/output
- Istruzioni simili a dei “load” e “store” ma con il contenuto inviato a degli specifici dispositivi (e.g., su architettura x86 venivano usate le istruzioni “in” e “out”)
- Quindi i dispositivi hanno un sistema di indirizzi diverso da quello della memoria principale
- Questo si chiama **port mapped I/O** (PMIO)

Port mapped I/O

Schema generale



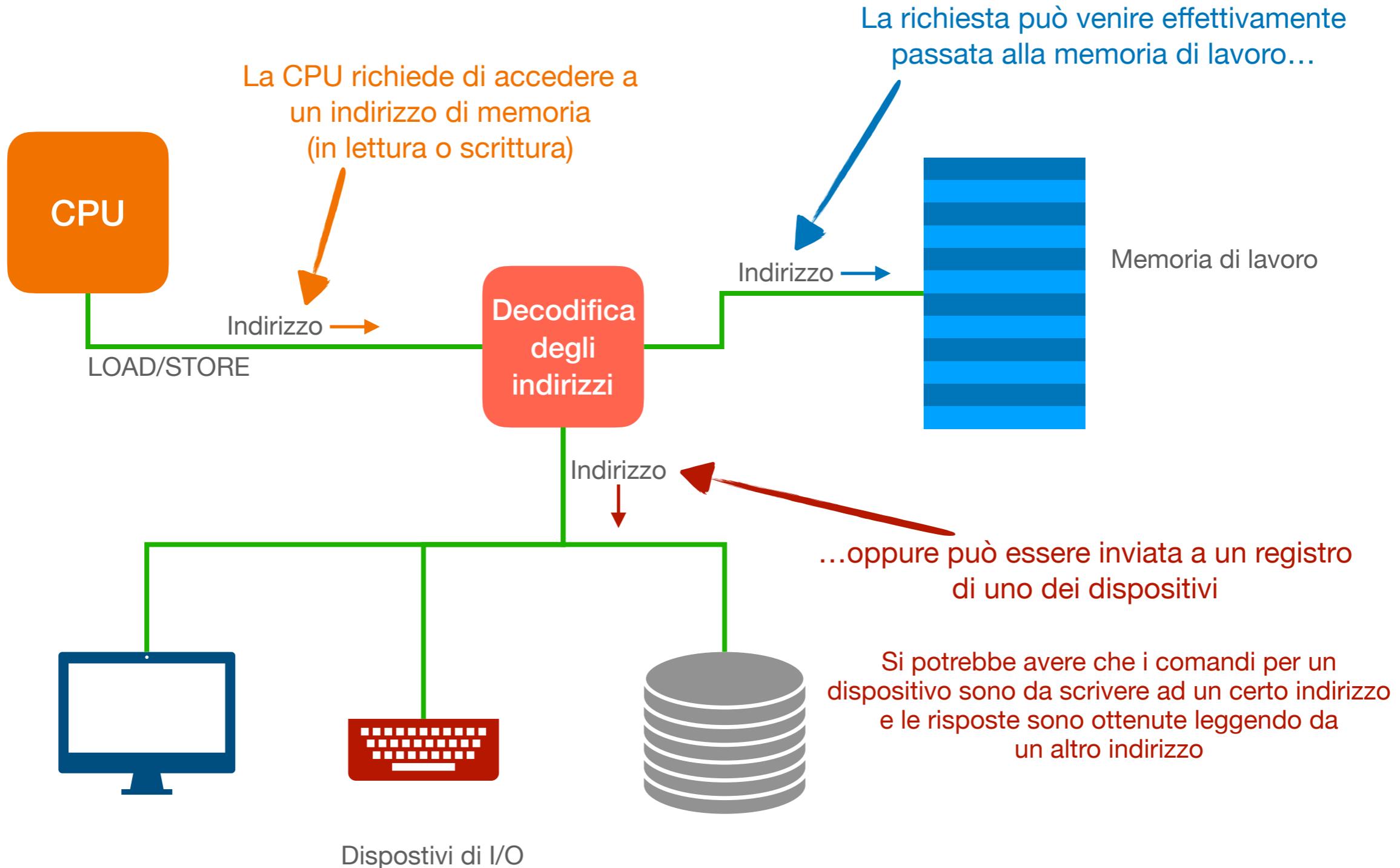
Memory mapped I/O

MMIO

- Una possibile idea è quello di non aggiungere istruzioni specifiche per la gestione dell'I/O
- Si dedicano alcuni degli indirizzi che sarebbero solitamente assegnati alla memoria di lavoro per operazioni di I/O
- Questo permette di usare le normali operazioni di LOAD e STORE per comunicare con i dispositivi di I/O
- Serve avere un componente che decida, dato un indirizzo, se questo deve essere mandato alla memoria di lavoro o inviato a uno dei dispositivi di I/O
- Questo si chiama **memory mapped I/O (MMIO)**

Memory mapped I/O

Schema generale



Polling e Interrupt

Due modi di gestire la comunicazione

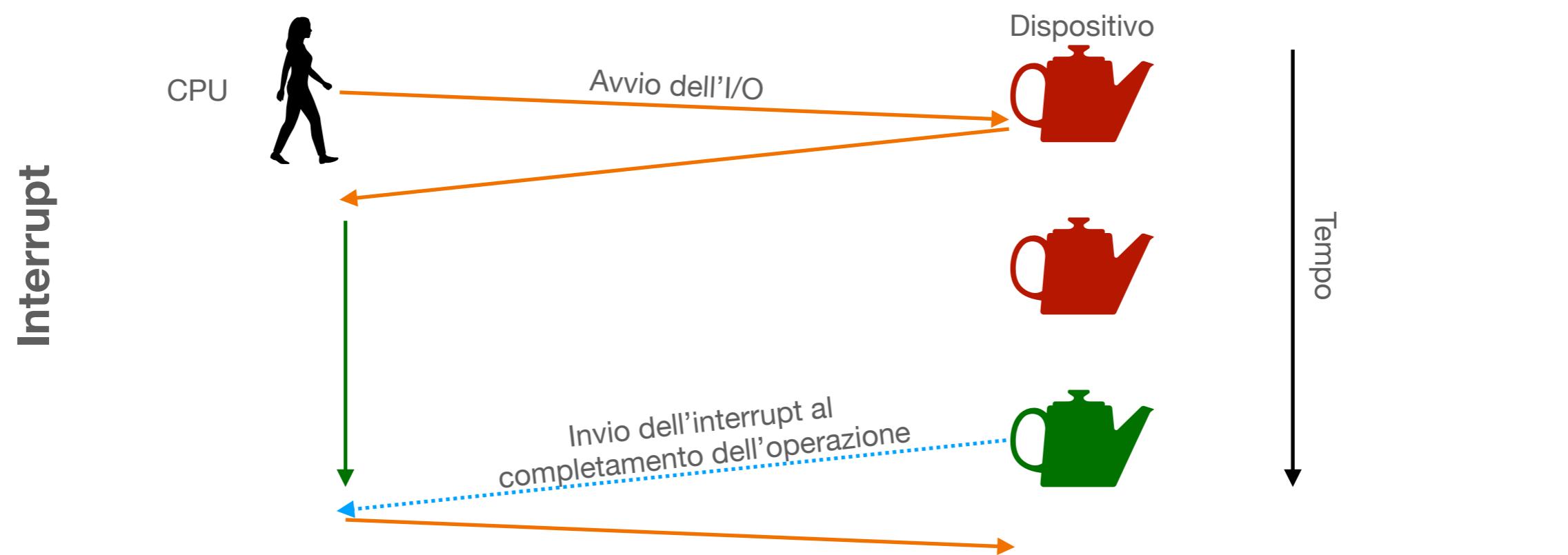
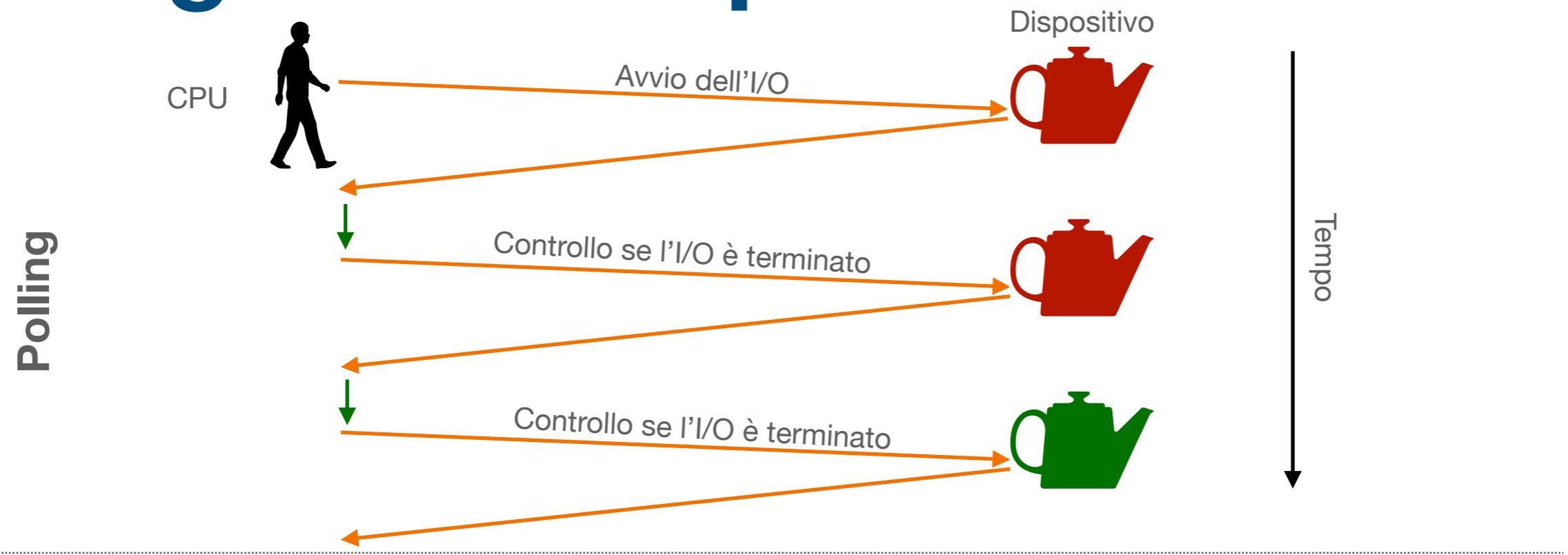
- I dispositivi di I/O sono molto più lenti della CPU nell'eseguire le operazioni
- Come fa la CPU a sapere quando un dispositivo di I/O ha completato una operazione (e.g., scrittura su disco) o hai dei dati pronti (e.g., lettura da tastiera)?
- Una possibilità è avere la CPU che controlla ad intervalli regolari una locazione di memoria (che si riferisce a un dispositivo di I/O) il cui valore indica se il dispositivo ha terminato l'operazione
- Questa procedura si chiama **polling**

Polling e Interrupt

Due modi di gestire la comunicazione

- Il polling può essere dispendioso se il dispositivo la maggior parte delle volte non ha nulla da riportare...
- ...servirebbe che il dispositivo “avvisasse” la CPU quando ha qualcosa di pronto (e.g., operazione completata, nuovo input da leggere, etc.)
- Questo avviene (tramite supporto hardware) con il meccanismo degli **interrupt**.
- Un interrupt “interrompe” la CPU e porta l'esecuzione a un indirizzo prestabilito (potete vederlo come un jump che viene forzato dall'esterno)

Polling vs interrupt



Come funzionano gli interrupt

E l'interrupt vector

- La gestione degli interrupt è molto dipendente dall'hardware
- La struttura base è che quando un interrupt viene ricevuto l'esecuzione al termine dell'istruzione corrente viene interrotta, alcuni registri e il valore del PC sono “salvati”.
- Se implementato, l'*interrupt vectoring* consiste nell'avere una tabella di indirizzi a cui saltare a seconda della tipologia di interrupt (pensate a funzioni diverse a seconda del segnale che si riceve)
- Una volta terminata la gestione dell'interrupt si ripristina l'esecuzione nel punto in cui si era interrotta

Direct Memory Access

DMA

- Se dobbiamo trasferire grandi quantità di dati sia usare polling che via interrupt non è pratico:
 - Nel caso del polling il processore non fa altro che spostare dati tra dispositivo e memoria
 - Nel caso degli interrupt il processore viene continuamente interrotto quando un trasferimento viene completato
 - L'idea è quella di usare un dispositivo aggiuntivo che si occupa solo di gestire l'I/O copiando direttamente i dati in memoria e avvisando il processore quando la copia è completata

Direct Memory Access

DMA

- Questo dispositivo aggiuntivo può accedere direttamente alla memoria senza coinvolgere il processore
- Questo permette al processore di “rimanere libero” durante le operazioni di I/O
- Questo è un meccanismo chiamato **Direct Memory Access** (DMA)
- DMA è indipendente dall’uso di memory mapped o port mapped I/O, può esistere in entrambi i casi

DMA

Schema generale

