Programmazione e Architetture (Modulo B)

Lezione 7
Stack, chiamate a funzione, I/O e interrupt

Luca Manzoni

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 per i registri (1/2)

Cosa salvare e dove

- I primi quattro argomenti della funzione sono salvati nei registri da R0 a R3, la funzione *chiamata* può modificare quei registri a piacimento

- Eventuali argomenti successivi sono salvati sullo stack

- I registri da R4 a R11 possono essere sovrascritti dalla funzione *chiamata*. Se vogliamo preservarne il valore dobbiamo salvarne il contenuto in memoria

- Dobbiamo anche salvare il valore nel Link Register (R14) che ci indica a che indirizzo dobbiamo effettuare il salto una volta terminata la funzione corrente

Convenzioni per i registri (2/2)

Cosa salvare e dove

- I valori di ritorno della funzione chiamata sono salvati nei registri R0 e R1
- Quando si ritorna da una funzione è compito del chiamante ripristinare i tutti i registri salvati, incluso il link register
- Riassumendo:
  - Salvare il valore di tutti i registri che ci interessano sullo stack
  - Chiamare la funzione
  - Al ritorno della funzione ripristinare tutti i registri che abbiamo salvato sullo stack

Convenzioni dei registri

Registri usati per il passaggio dei primi quattro argomenti

R0

R1

R2

R3

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

R4

⋮

R11

R12

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

R13

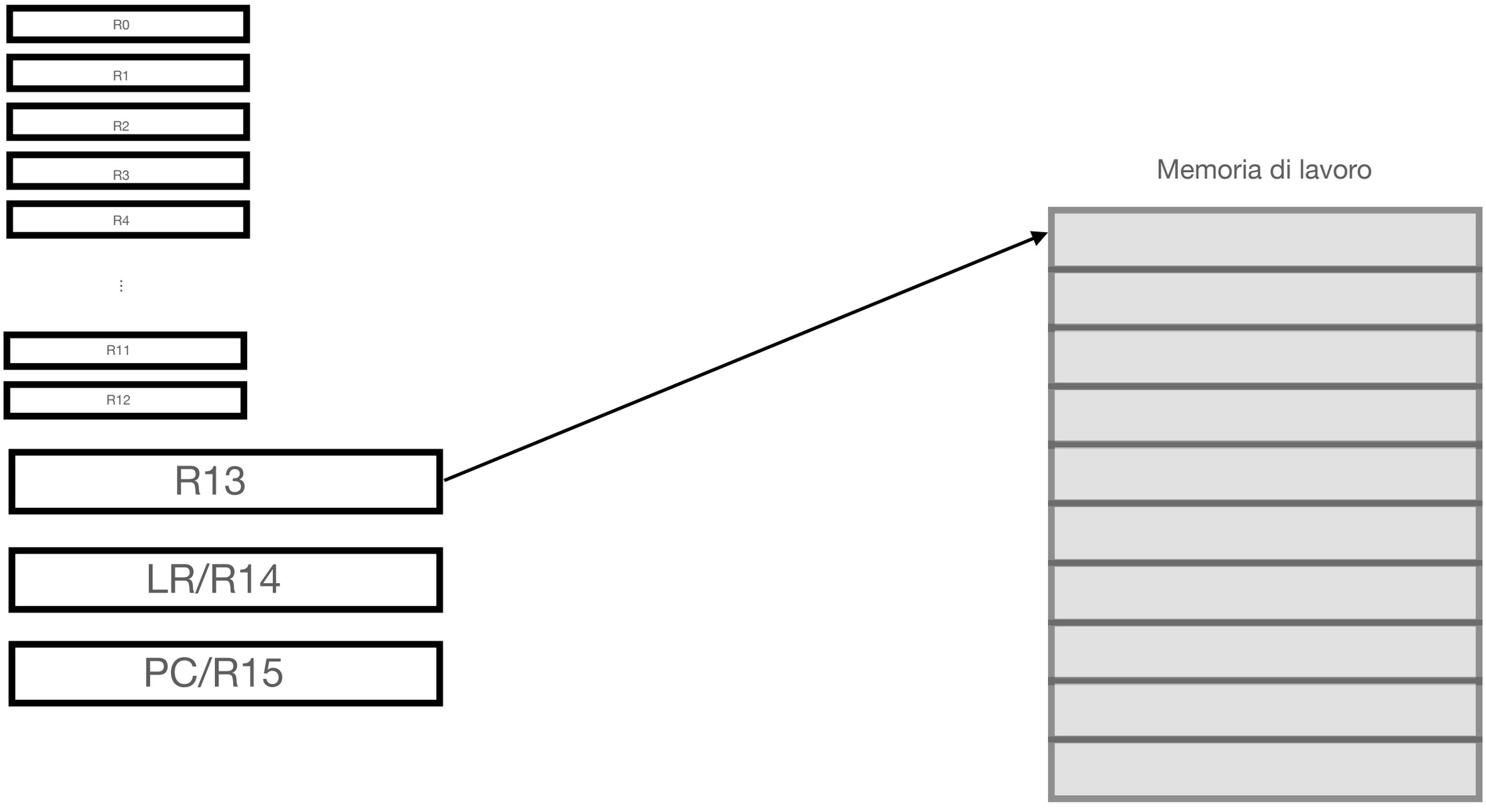
LR/R14

PC/R15

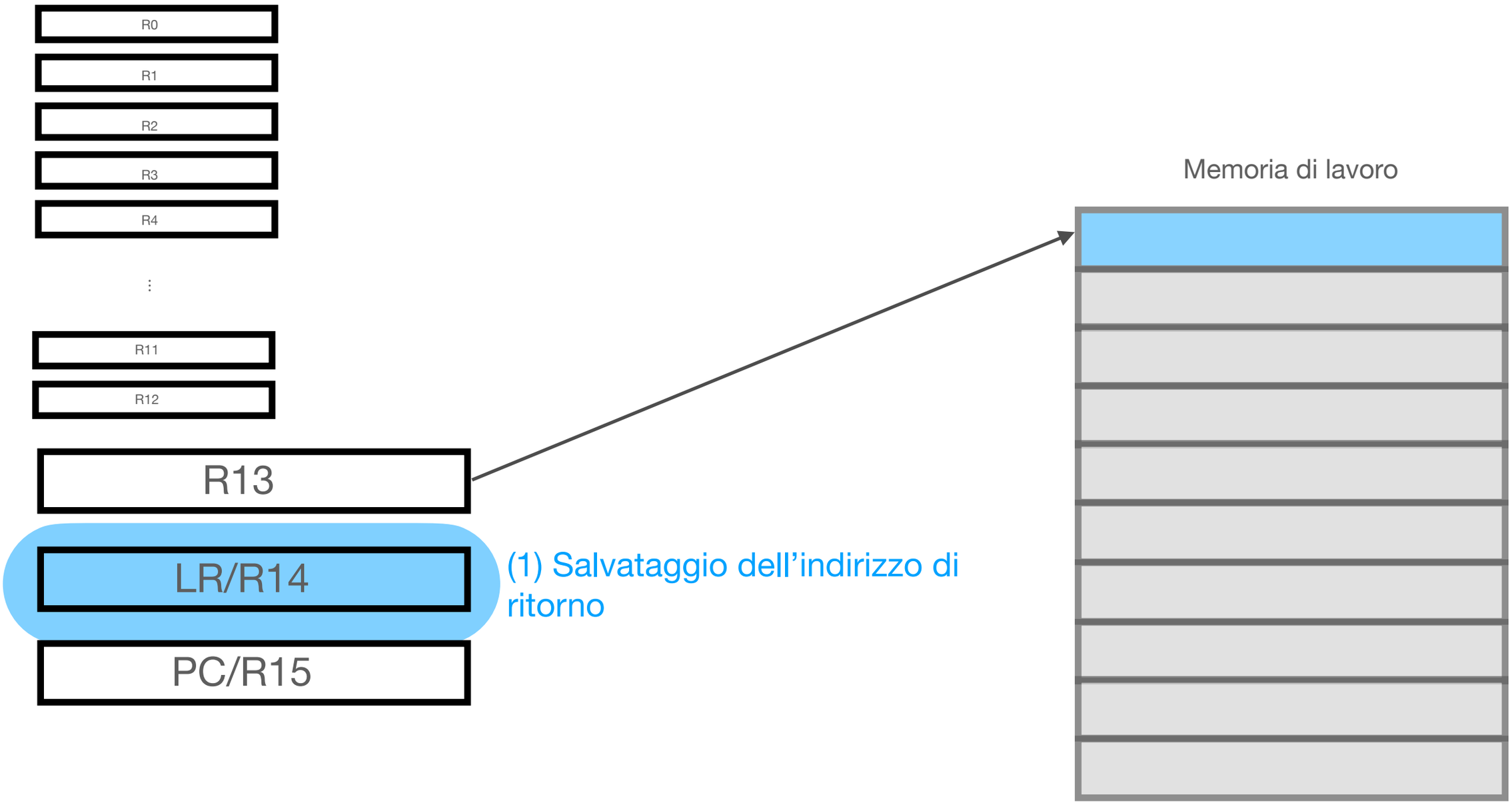
PSR

*Gli argomenti successivi devono essere posizionati nello stack

Cosa fare quando si chiama una funzione

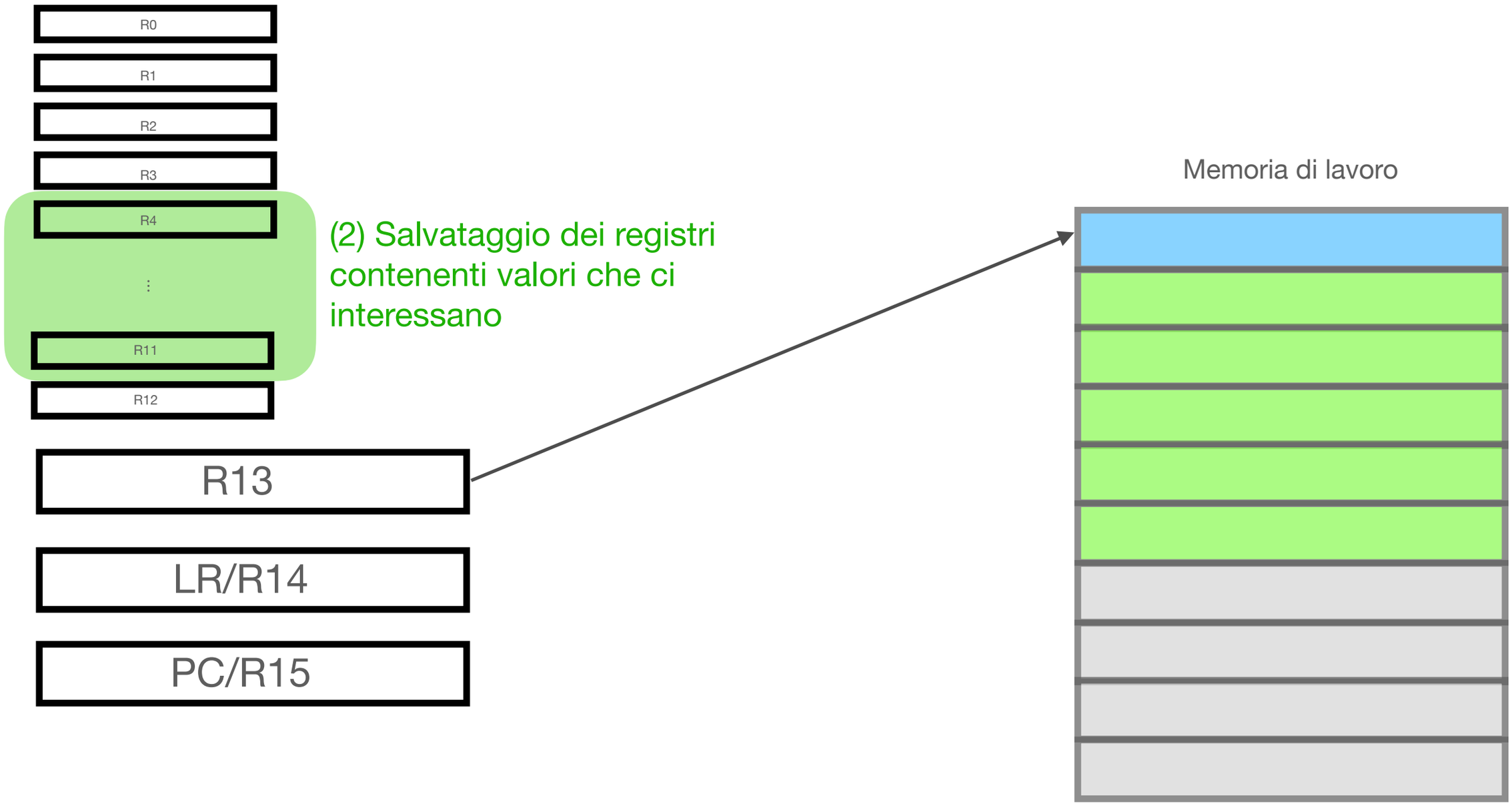


Cosa fare quando si chiama una funzione

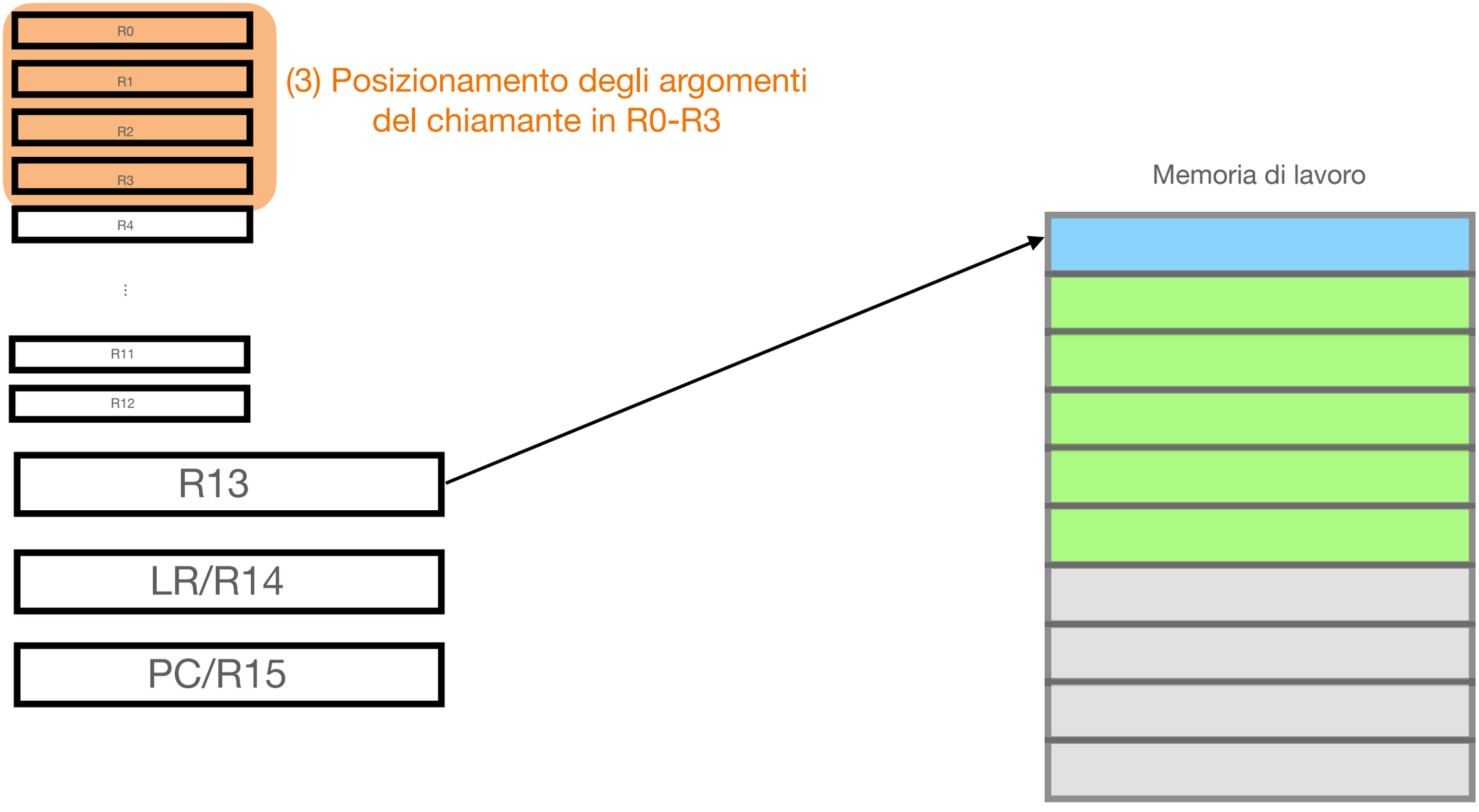


(1) Salvataggio dell’indirizzo di ritorno

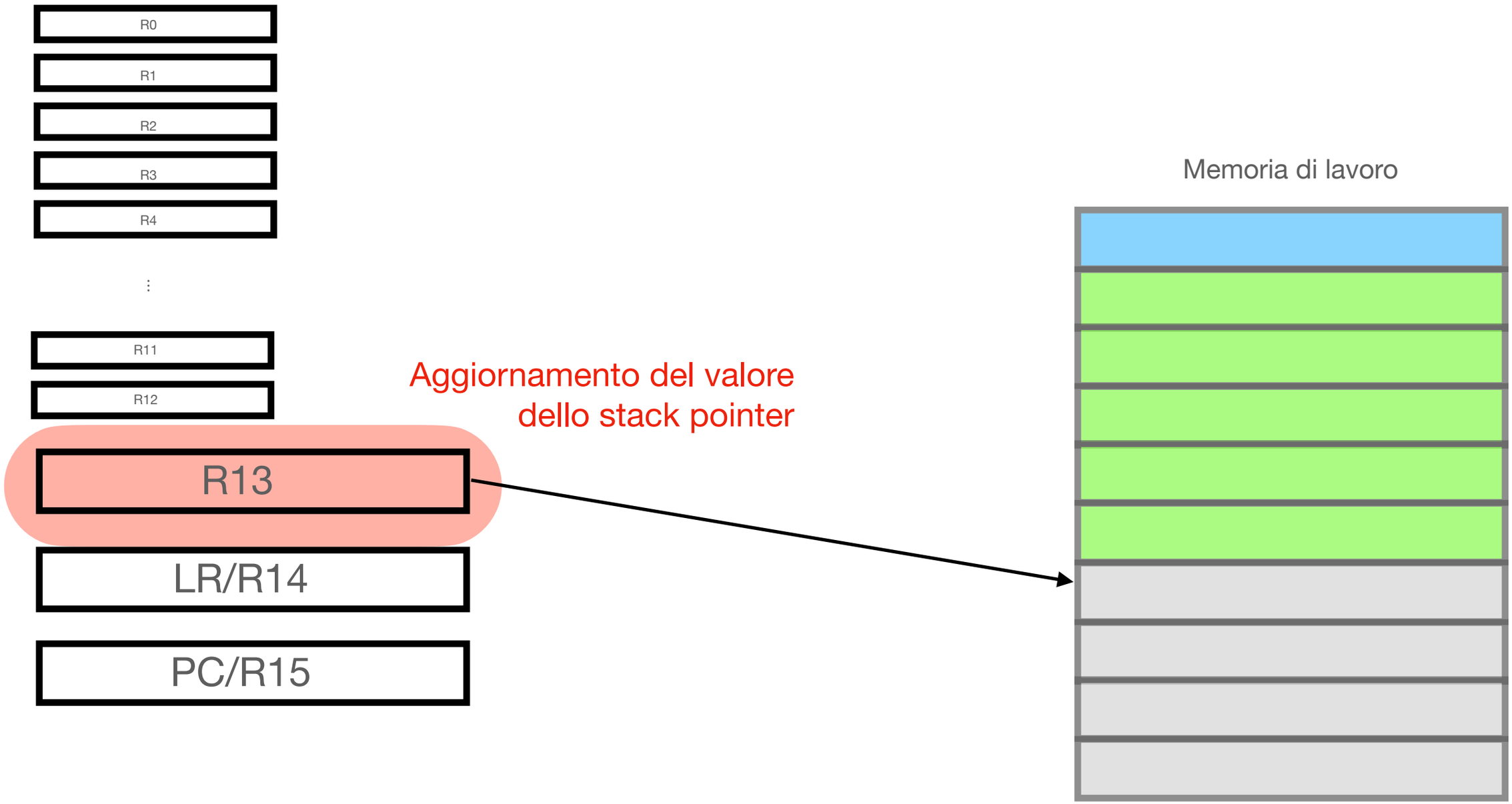
Cosa fare quando si chiama una funzione



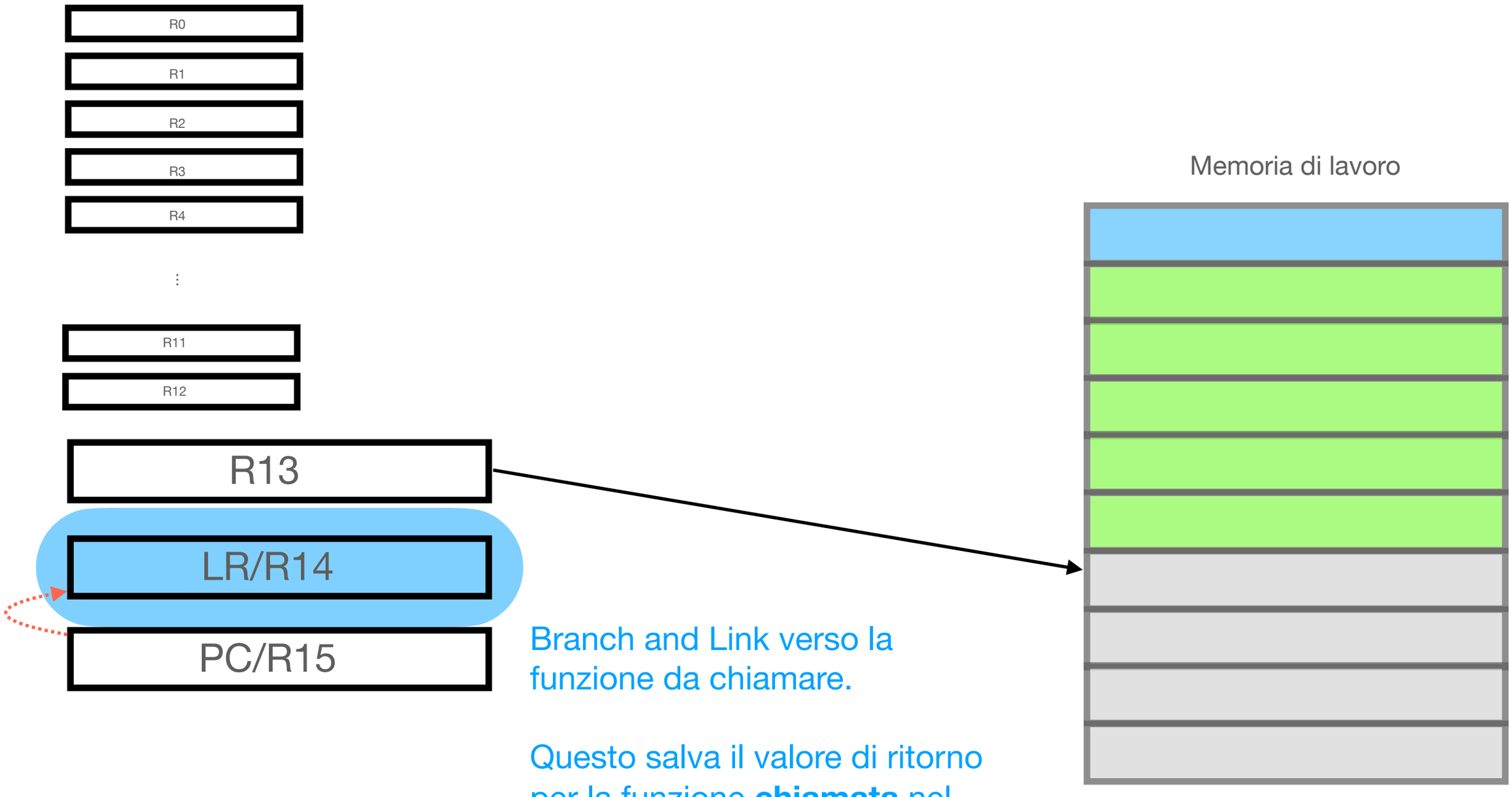
Cosa fare quando si chiama una funzione



Cosa fare quando si chiama una funzione



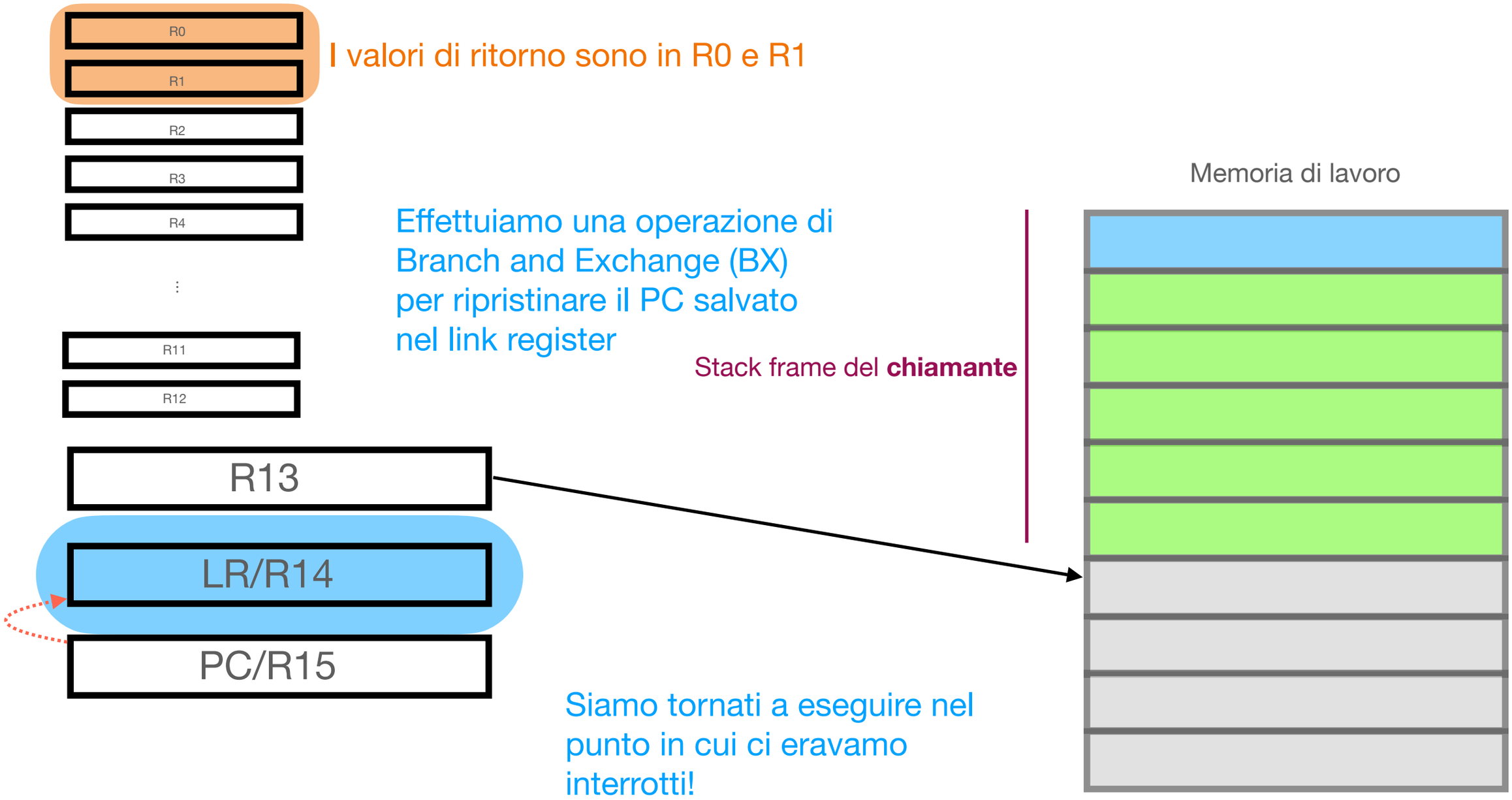
Cosa fare quando si chiama una funzione



Branch and Link verso la funzione da chiamare.

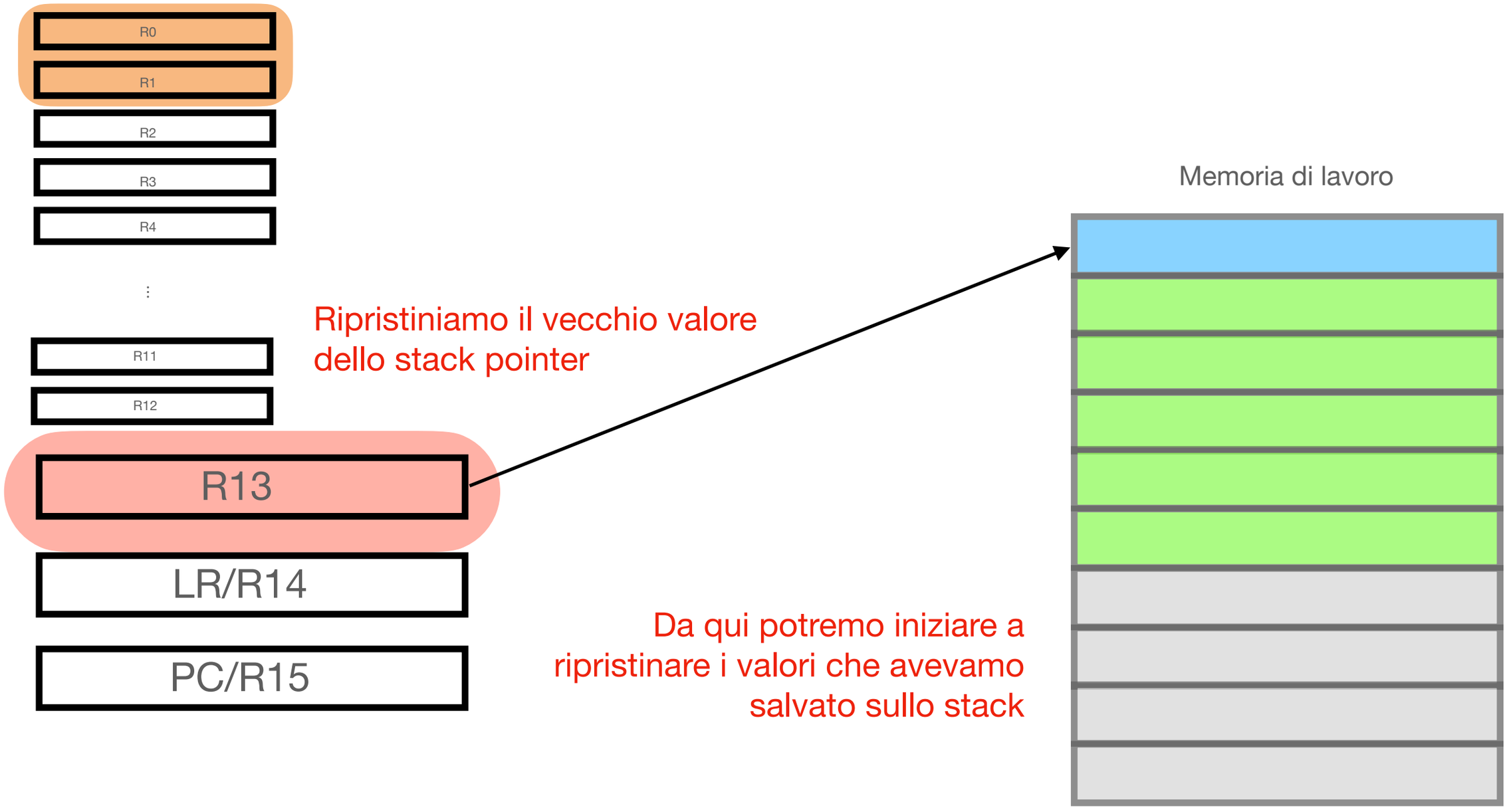
Questo salva il valore di ritorno per la funzione **chiamata** nel link register

Cosa fare ritornando da una funzione

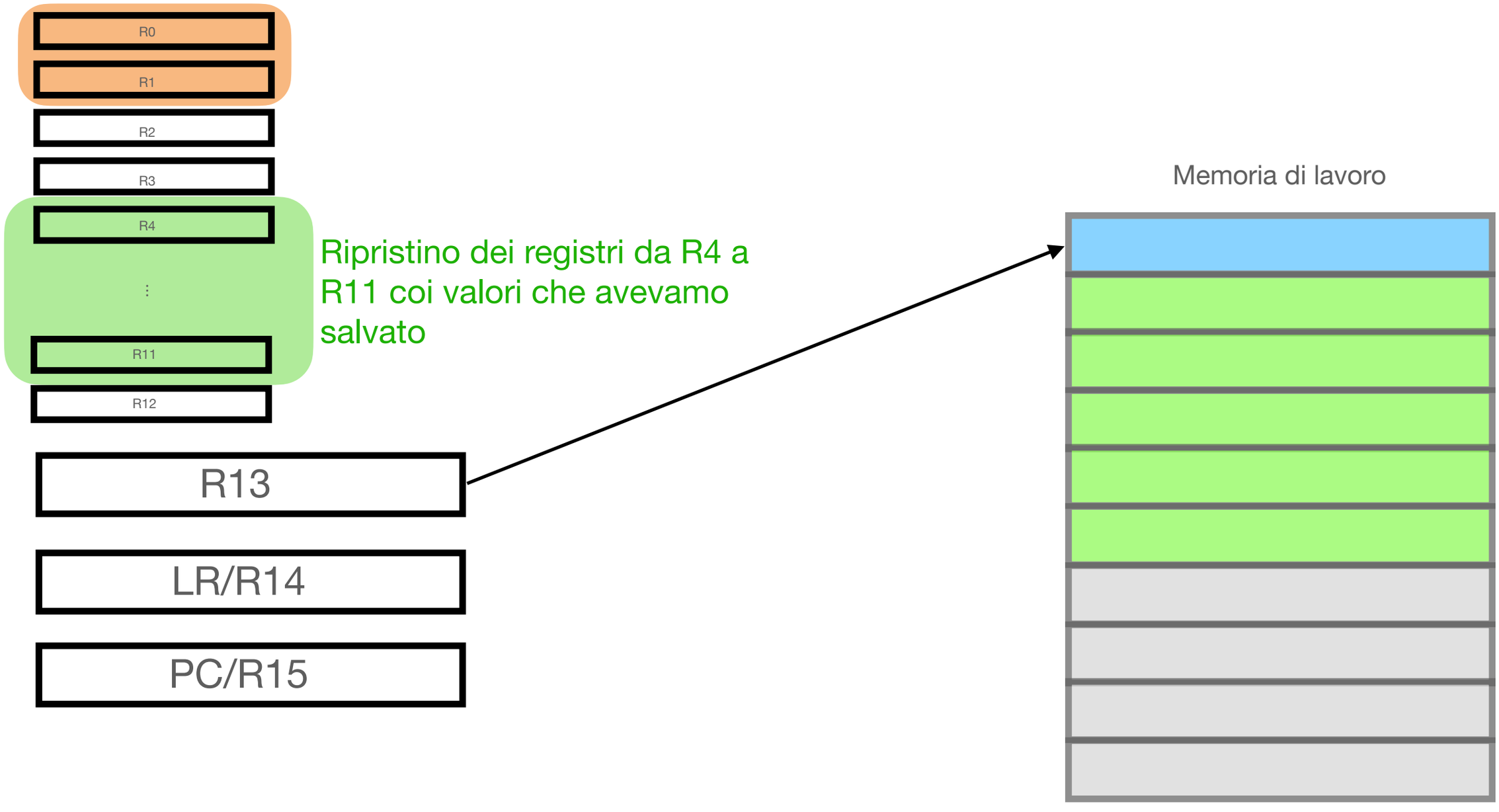


Nota: branch and exchange non è supportato da VisUAL. Possiamo usare MOV R14, R15

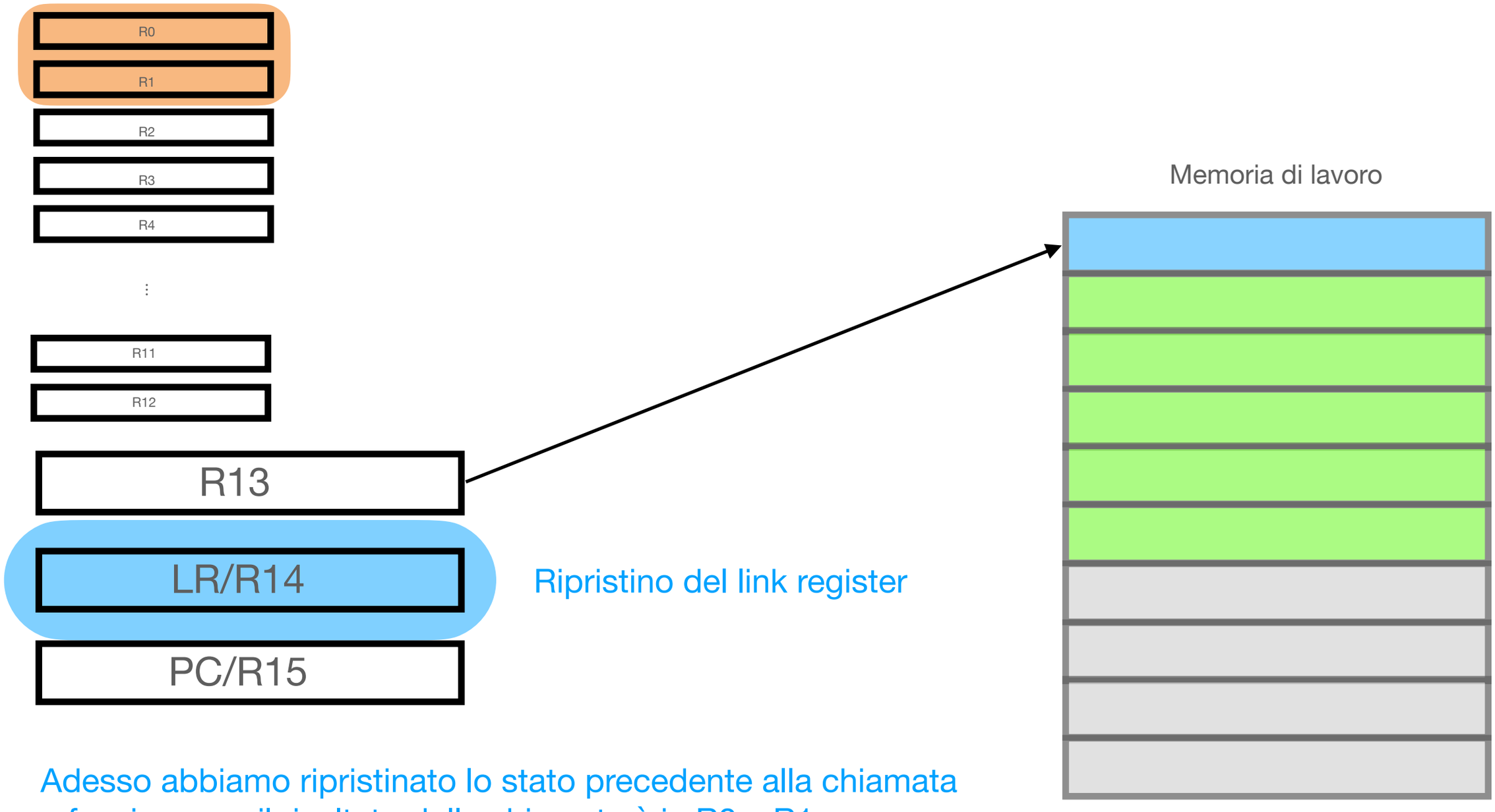
Cosa fare ritornando da una funzione



Cosa fare ritornando da una funzione



Cosa fare ritornando da una funzione



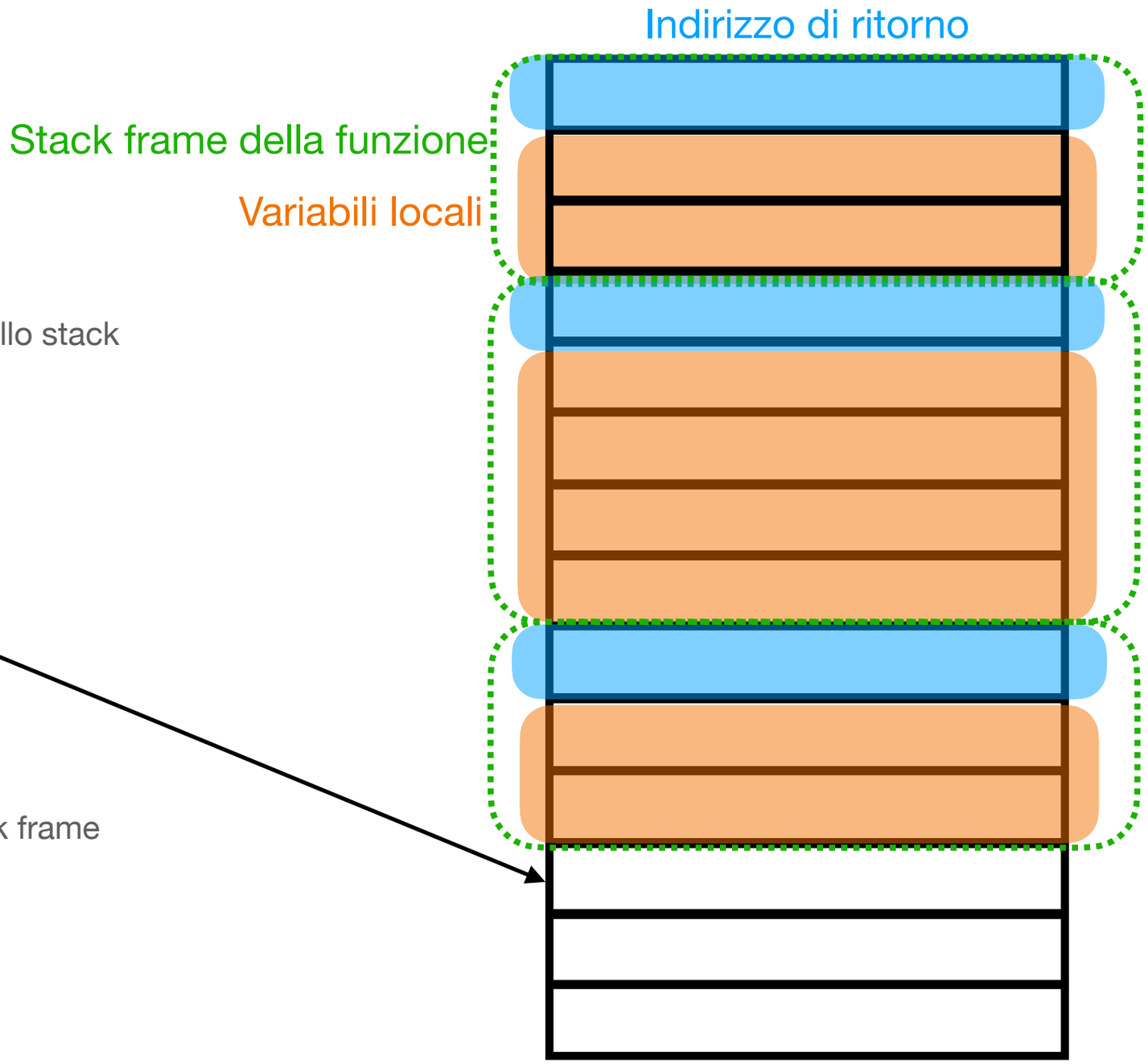
Adesso abbiamo ripristinato lo stato precedente alla chiamata a funziona ma il risultato della chiamata è in R0 e R1

Stack: chiamate innestate

Qui abbiamo quattro chiamate a funzione.

Possiamo osservare i vari stack frame presenti sullo stack

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 dispostivi:

  - 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 dispostivi di input/output

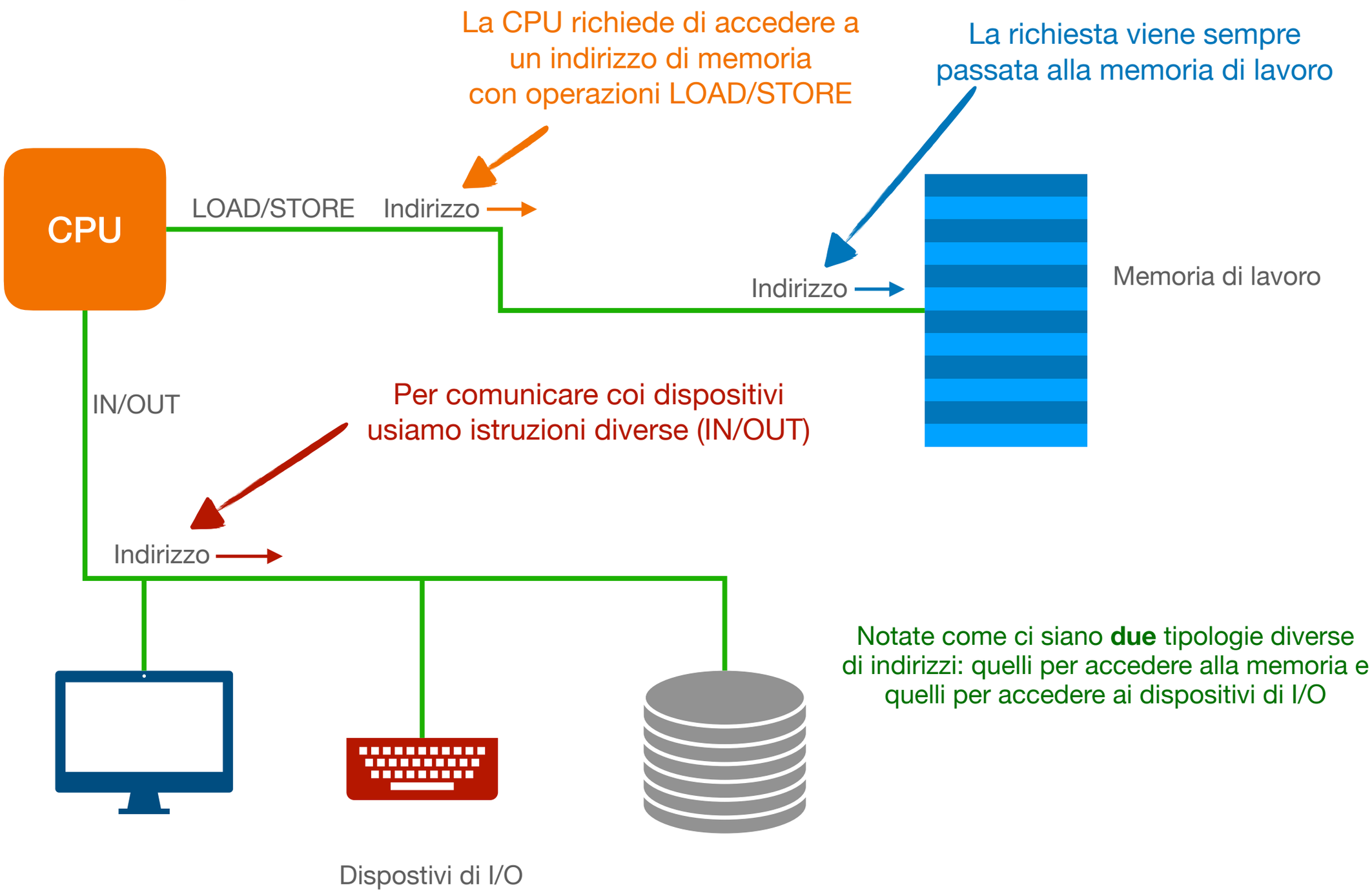
- Istruzioni simili a dei “load” e “store” ma con il contenuto inviato a degli specifici dispositivi (e.g., su architettura x86 venivano usare le istruzioni “in” e “out”)

- 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



Notate come ci siano **due** tipologie diverse di indirizzi: quelli per accedere alla memoria e quelli per accedere ai dispositivi di I/O

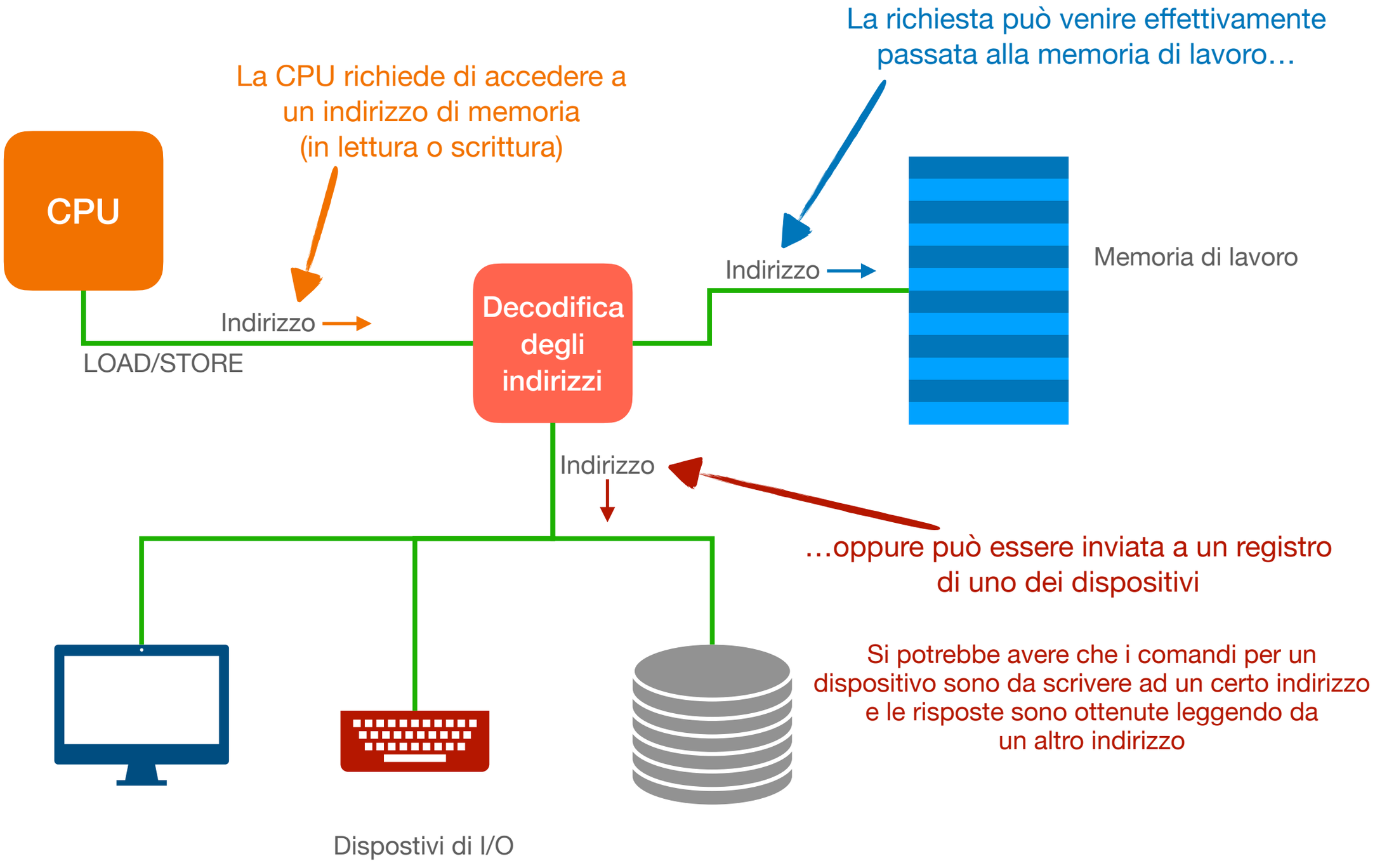
Memory mapped I/O

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

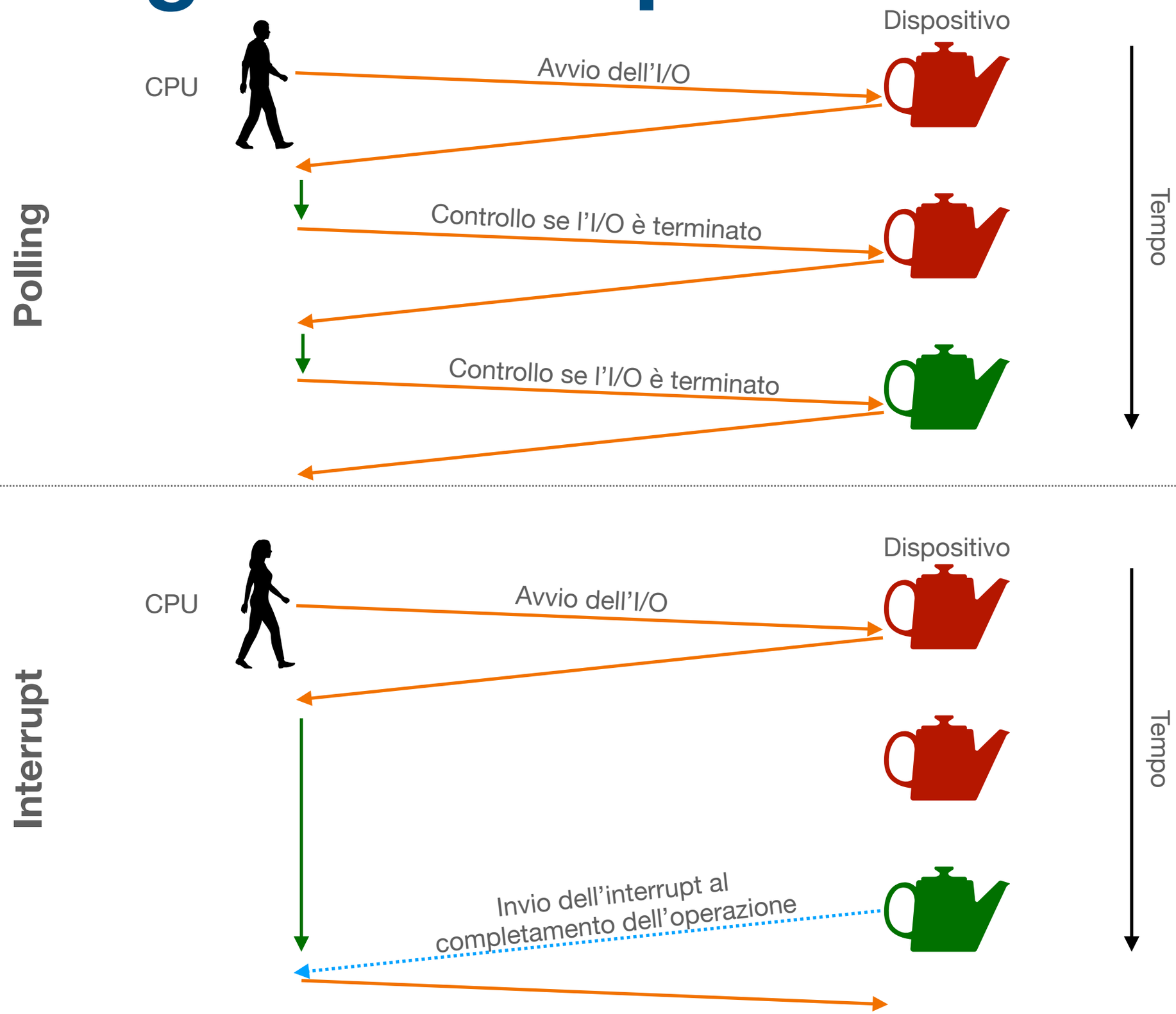
- I dispositivi di I/O sono molto più lenti della CPU nell'eseguire le operazioni
- Come fa la CPU a sapere quando un dispositivo di I/O ha completato una operazione (e.g., scrittura su disco) o hai dei dati pronti (e.g., lettura da tastiera)?
- Una possibilità è avere la CPU che controlla ad intervalli 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