

Architetture Degli Elaboratori

Lezione 5

Stack, chiamate a funzione, I/O e interrupt

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
- **D**: in C, chi è deve rispettare queste convenzioni?
- 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
- **D**: cosa scriviamo in LR quando chiamiamo una funzione?

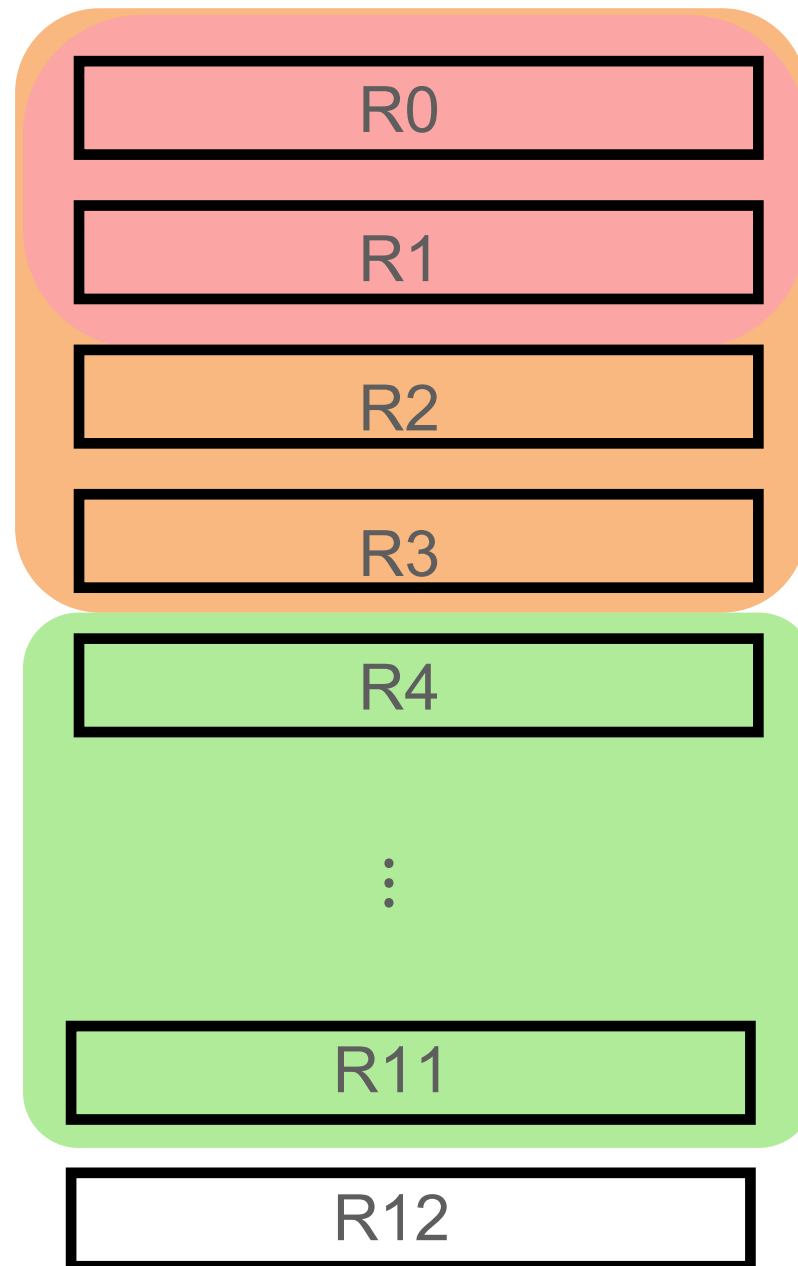
Convenzioni per i registri (2/2)

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

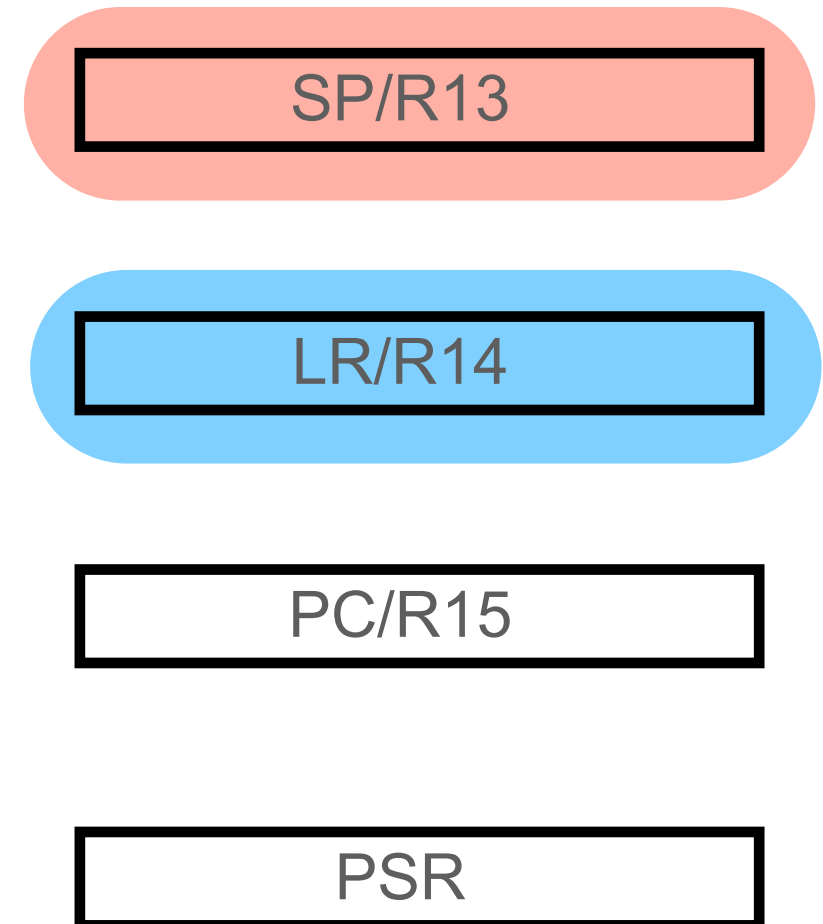


Registri in cui viene salvato il
valore di ritorno della funzione

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

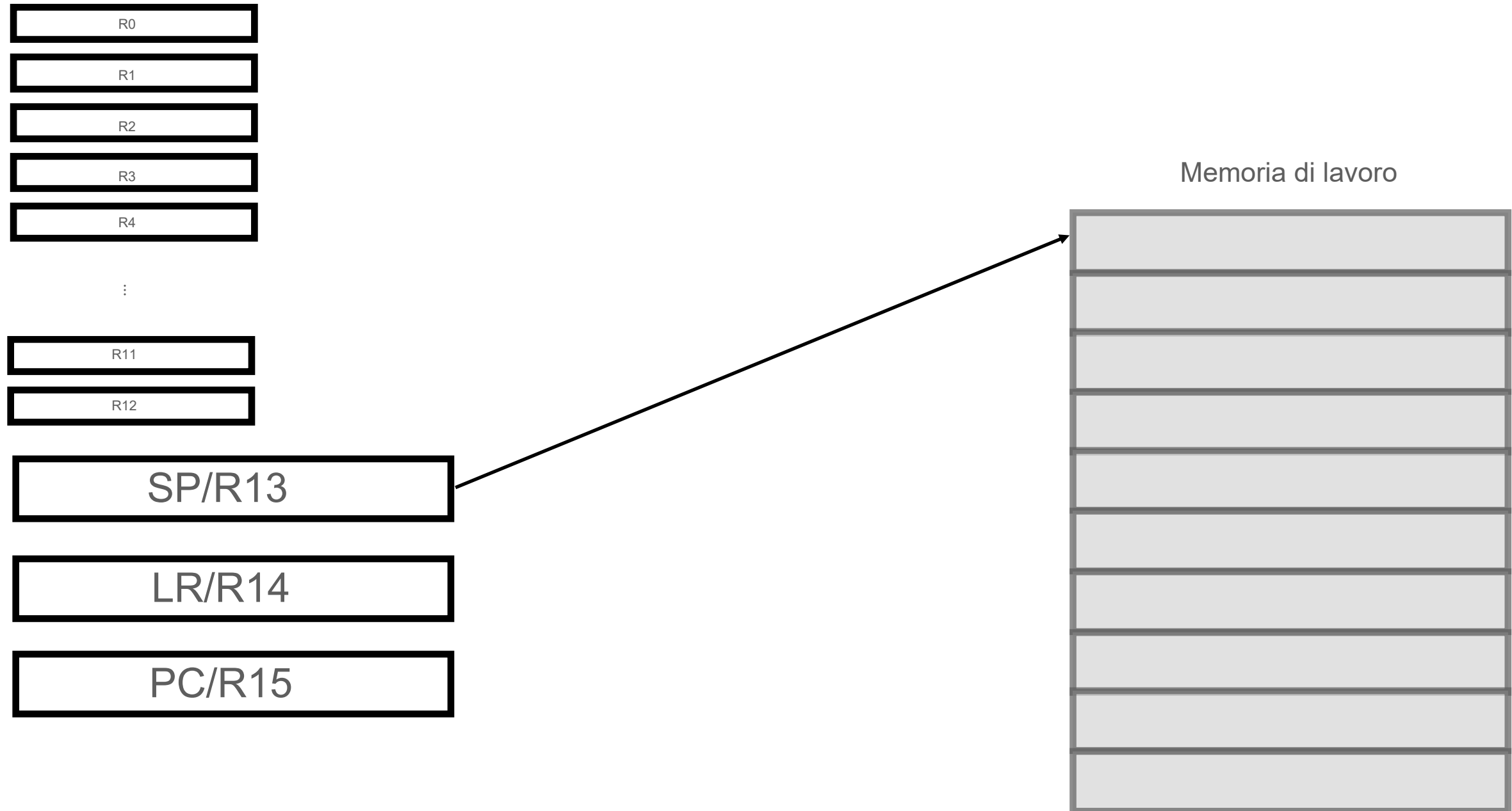
Registri salvati dal chiamante.
Se la funzione *chiamante*
vuole
preservare il valore di questi
registri deve salvarli in
memoria e ripristinarli al ritorno
della funzione chiamata

Indirizzo dello stack

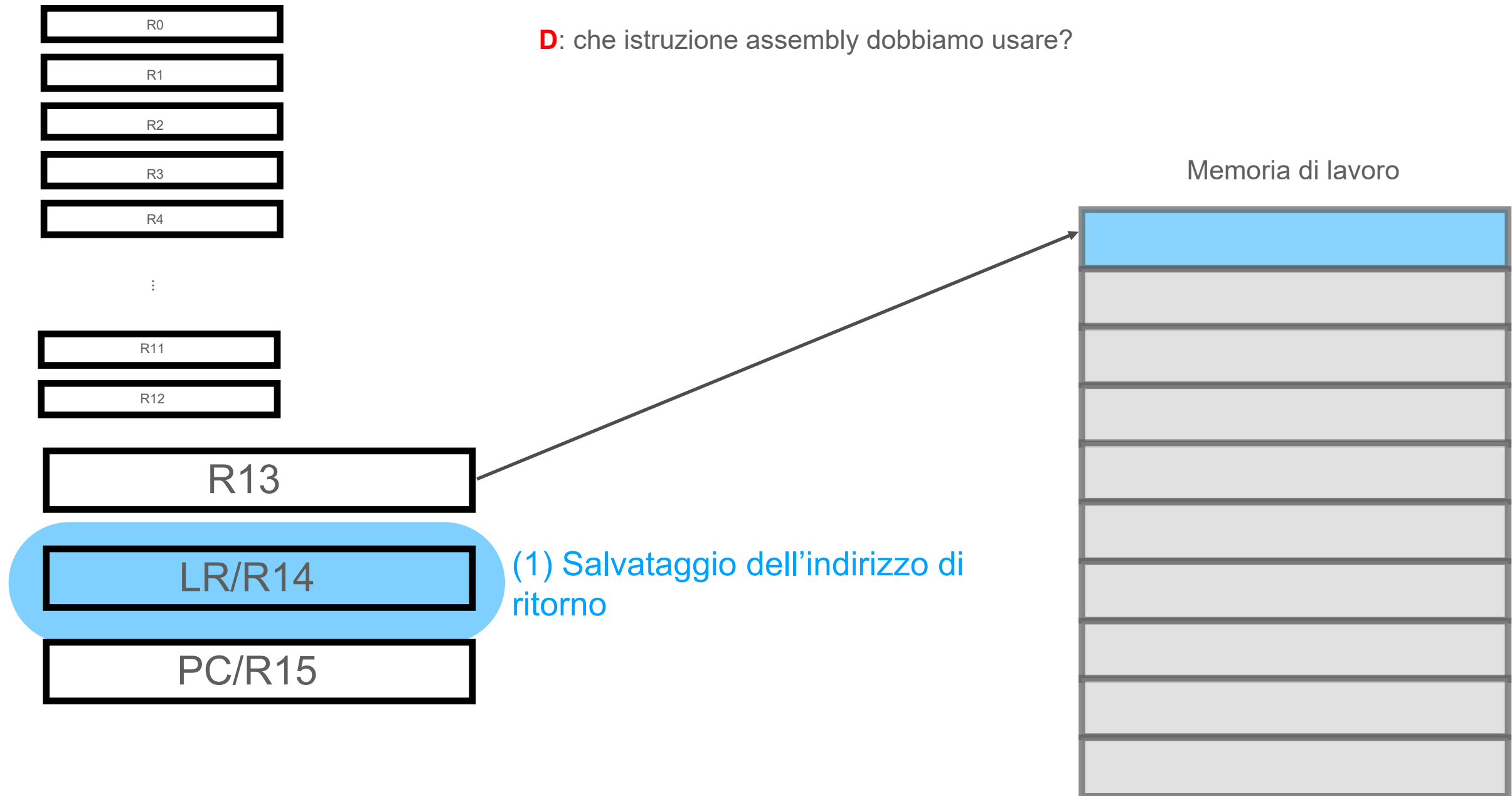


*Gli argomenti successivi devono essere posizionati nello stack

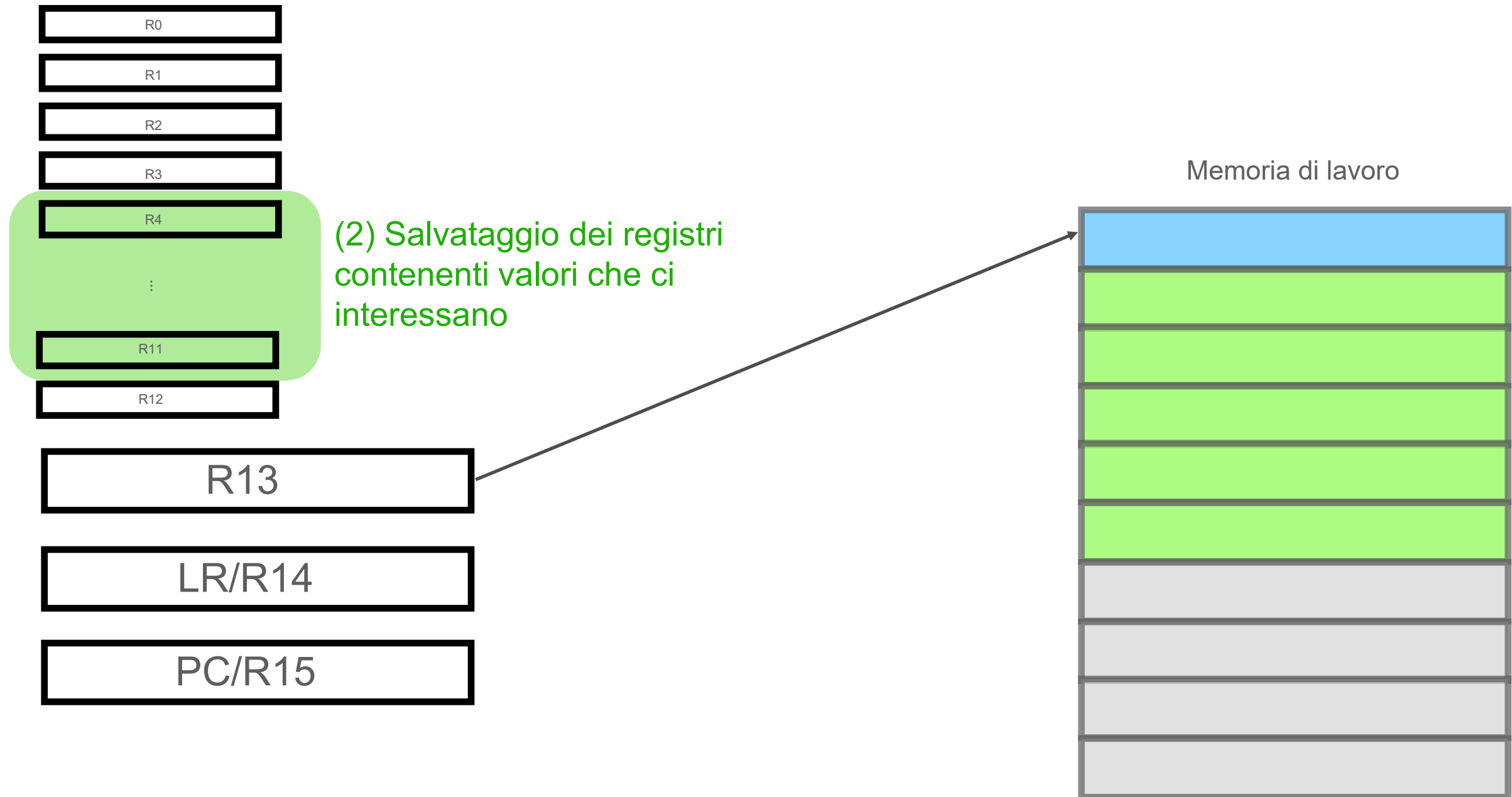
Cosa fare quando si chiama una funzione



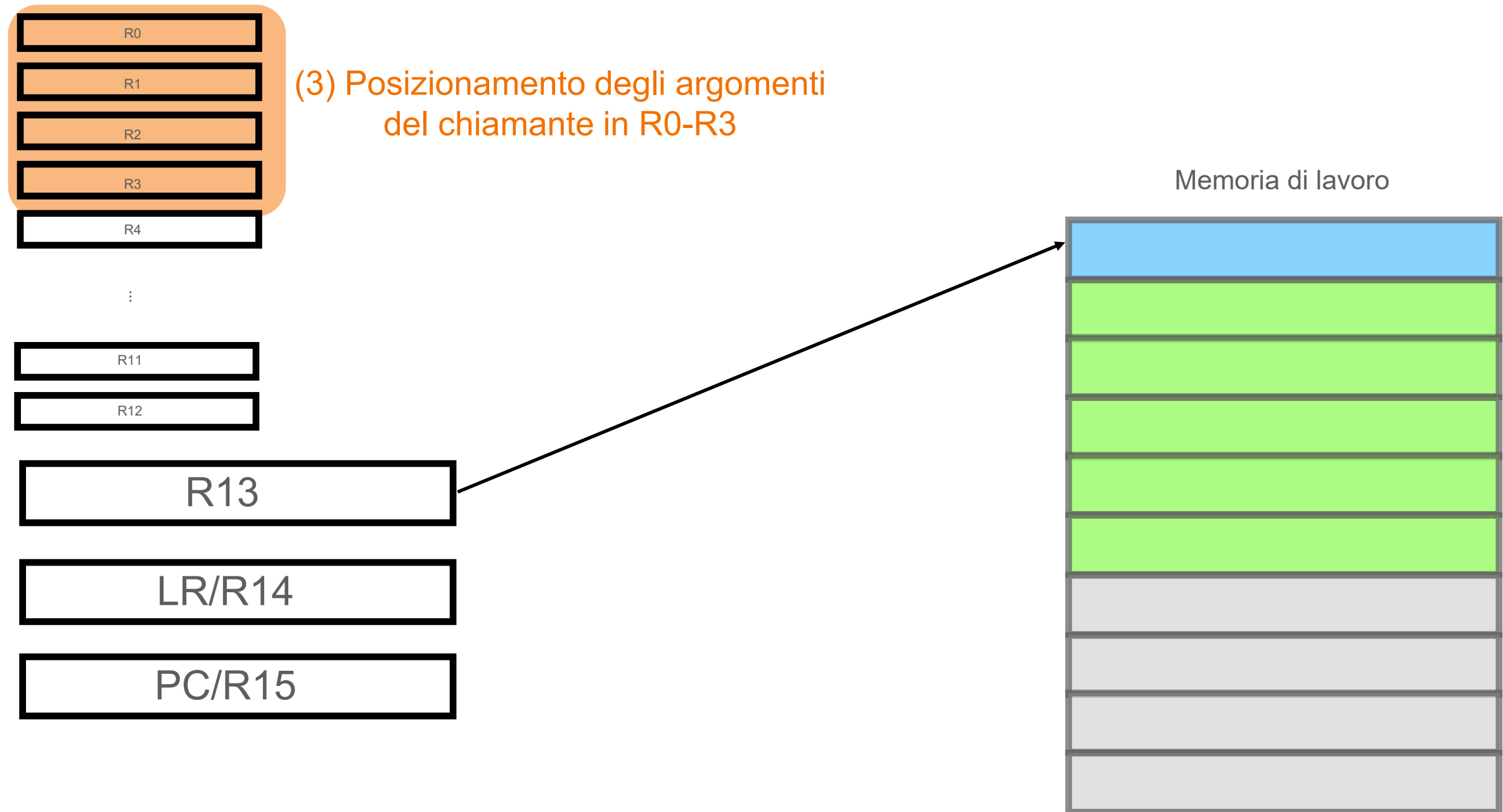
Cosa fare quando si chiama una funzione



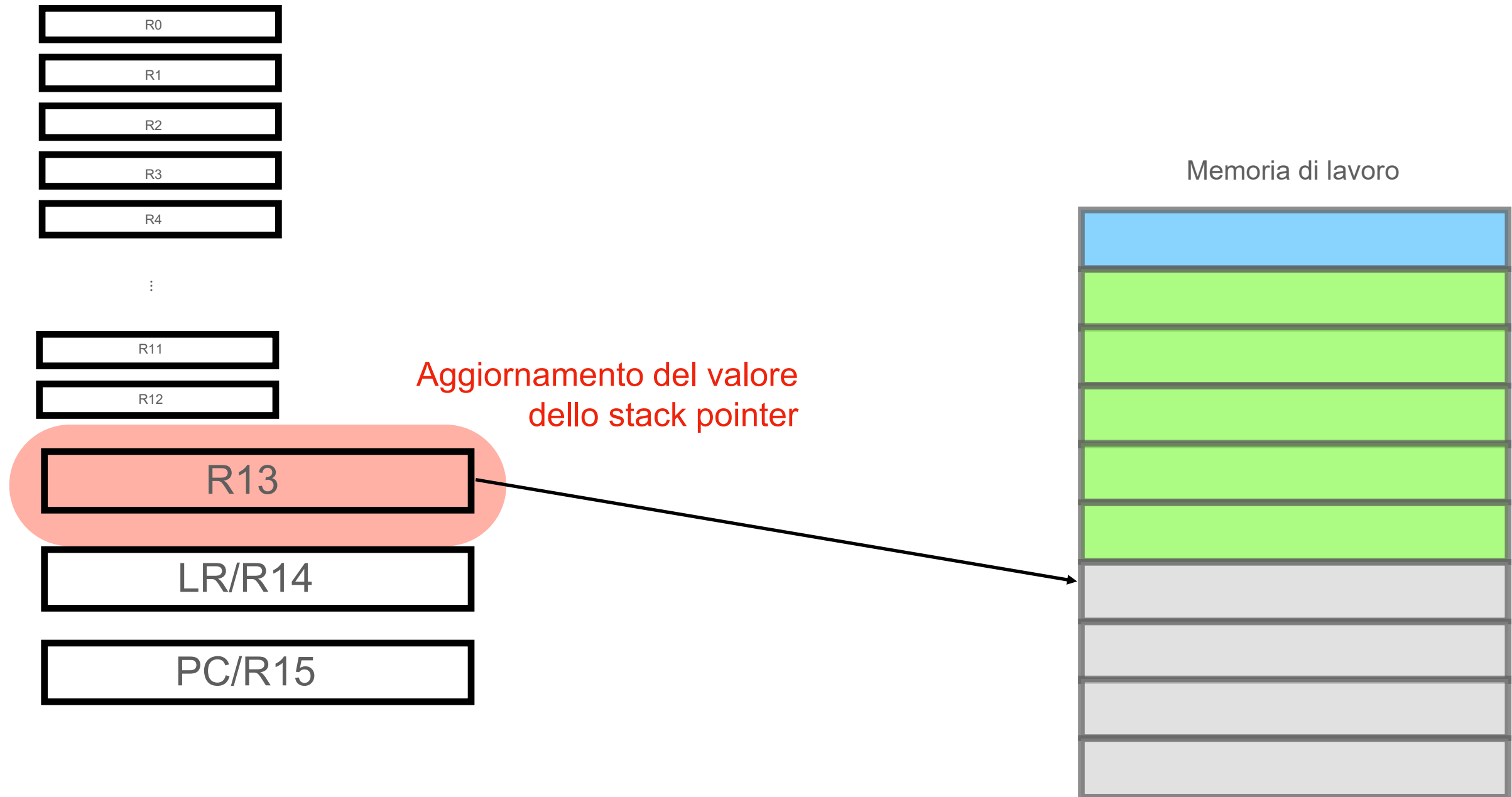
Cosa fare quando si chiama una funzione



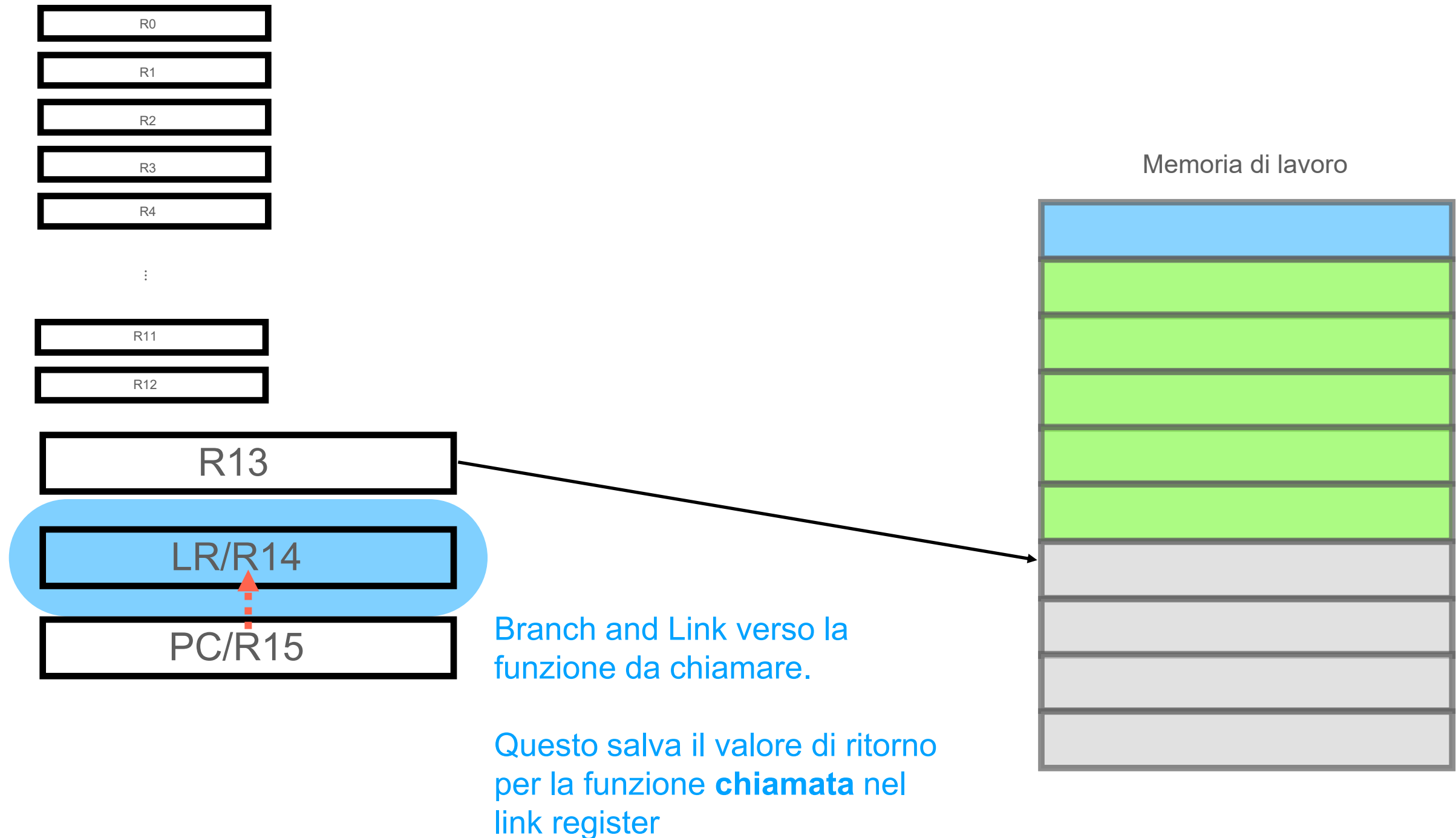
Cosa fare quando si chiama una funzione



Cosa fare quando si chiama una funzione

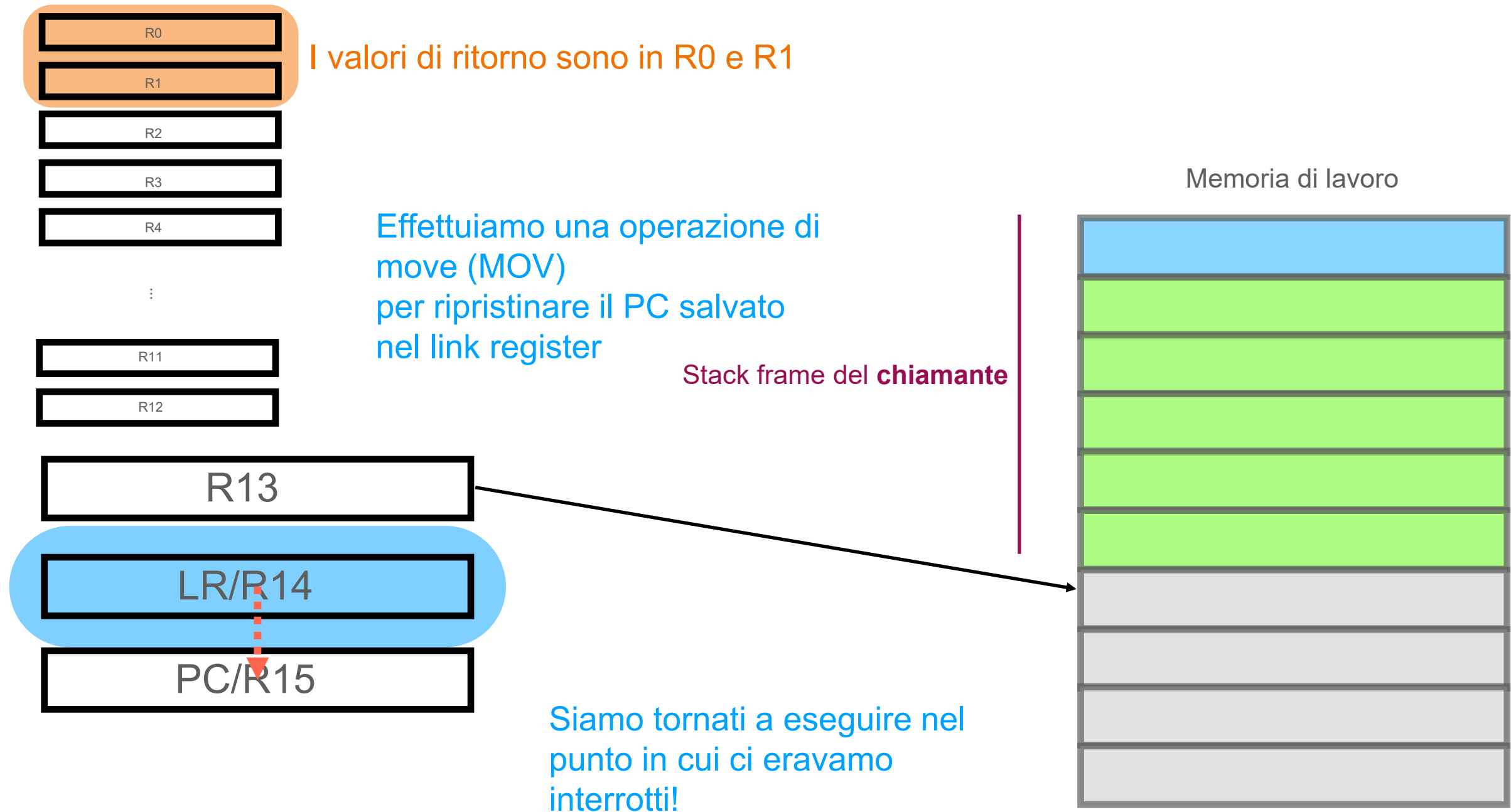


Cosa fare quando si chiama una funzione

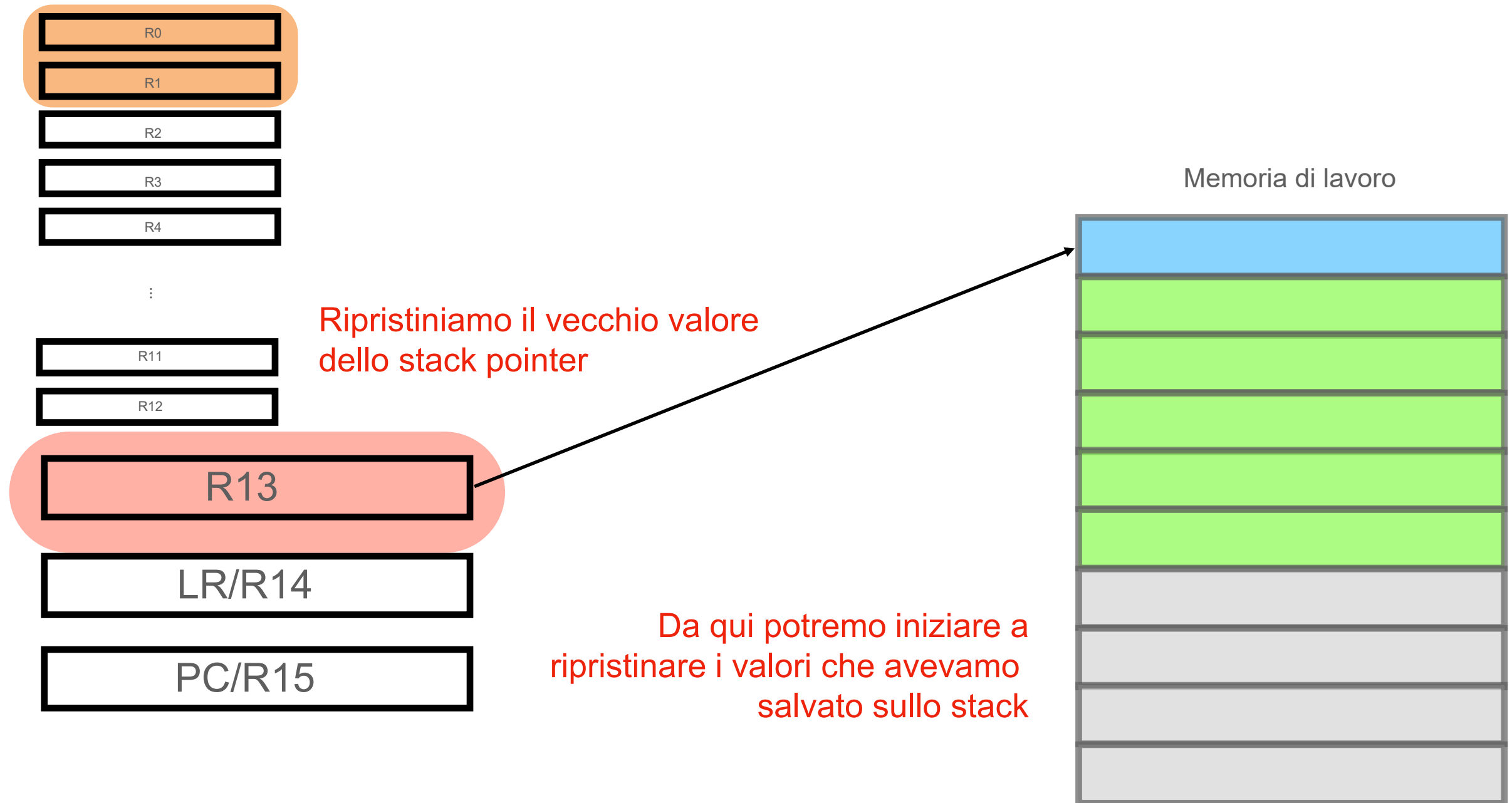


Cosa fare ritornando da una funzione

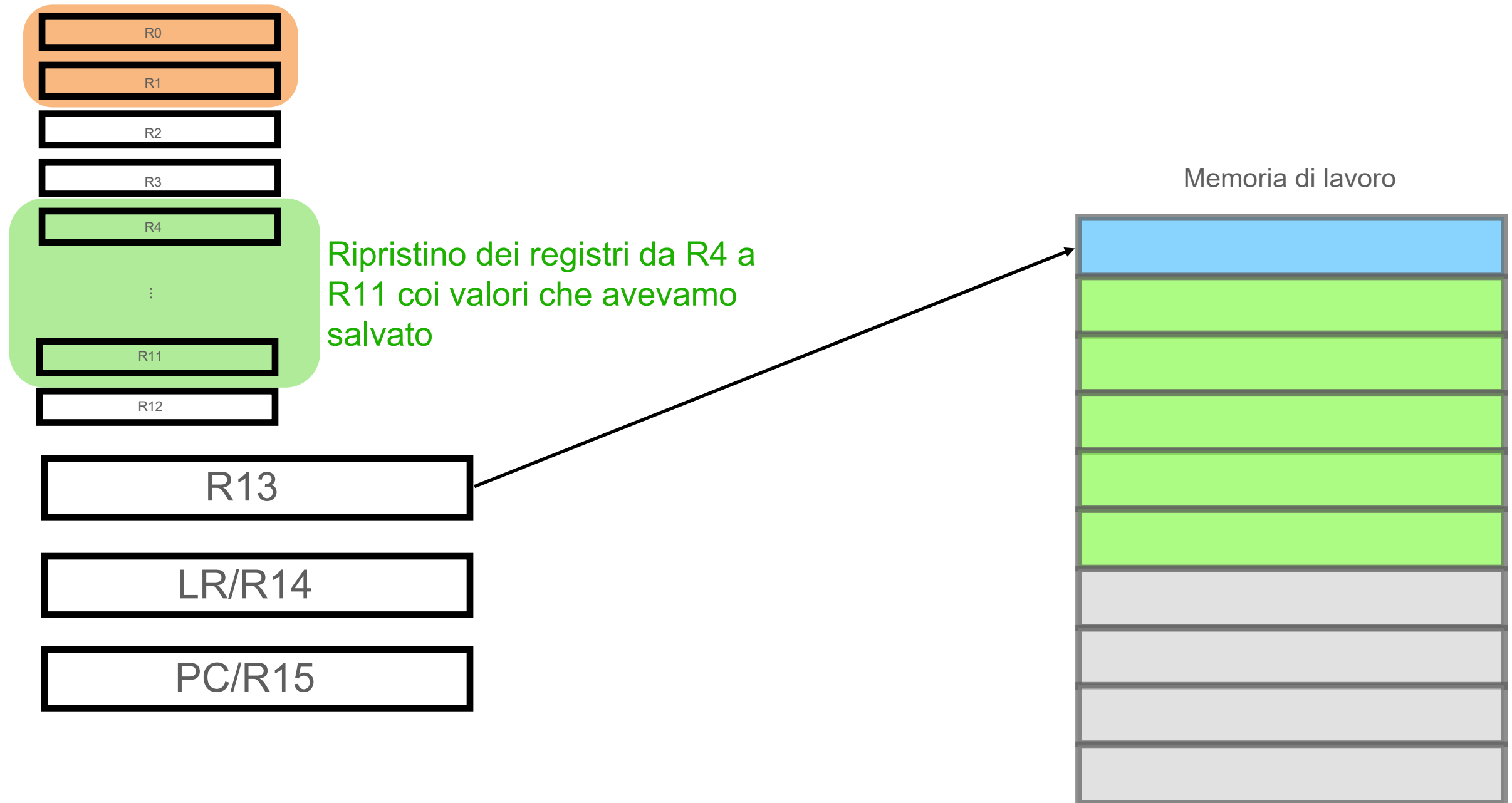
Fine funzione chiamata:



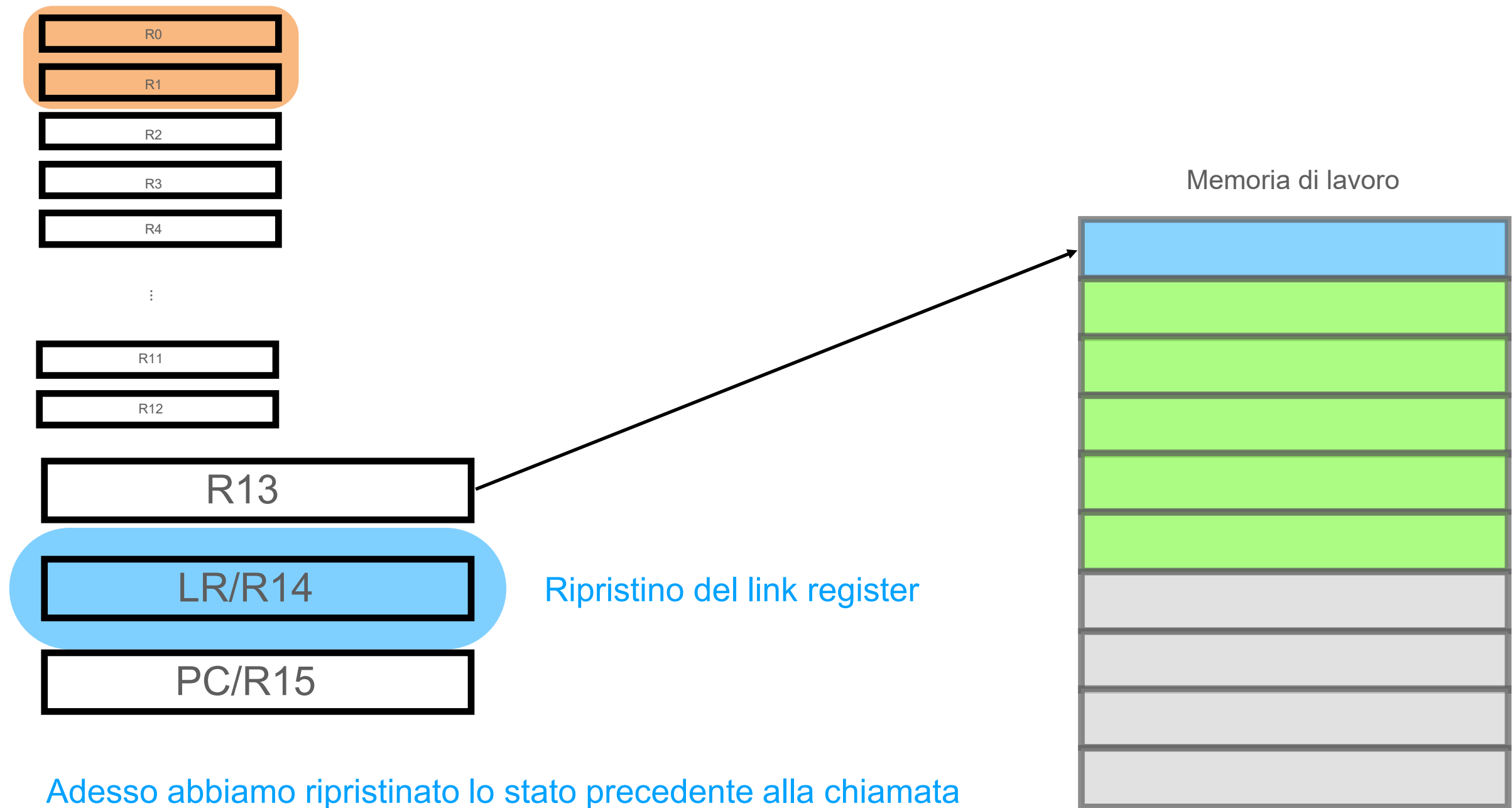
Cosa fare ritornando da una funzione



Cosa fare ritornando da una funzione

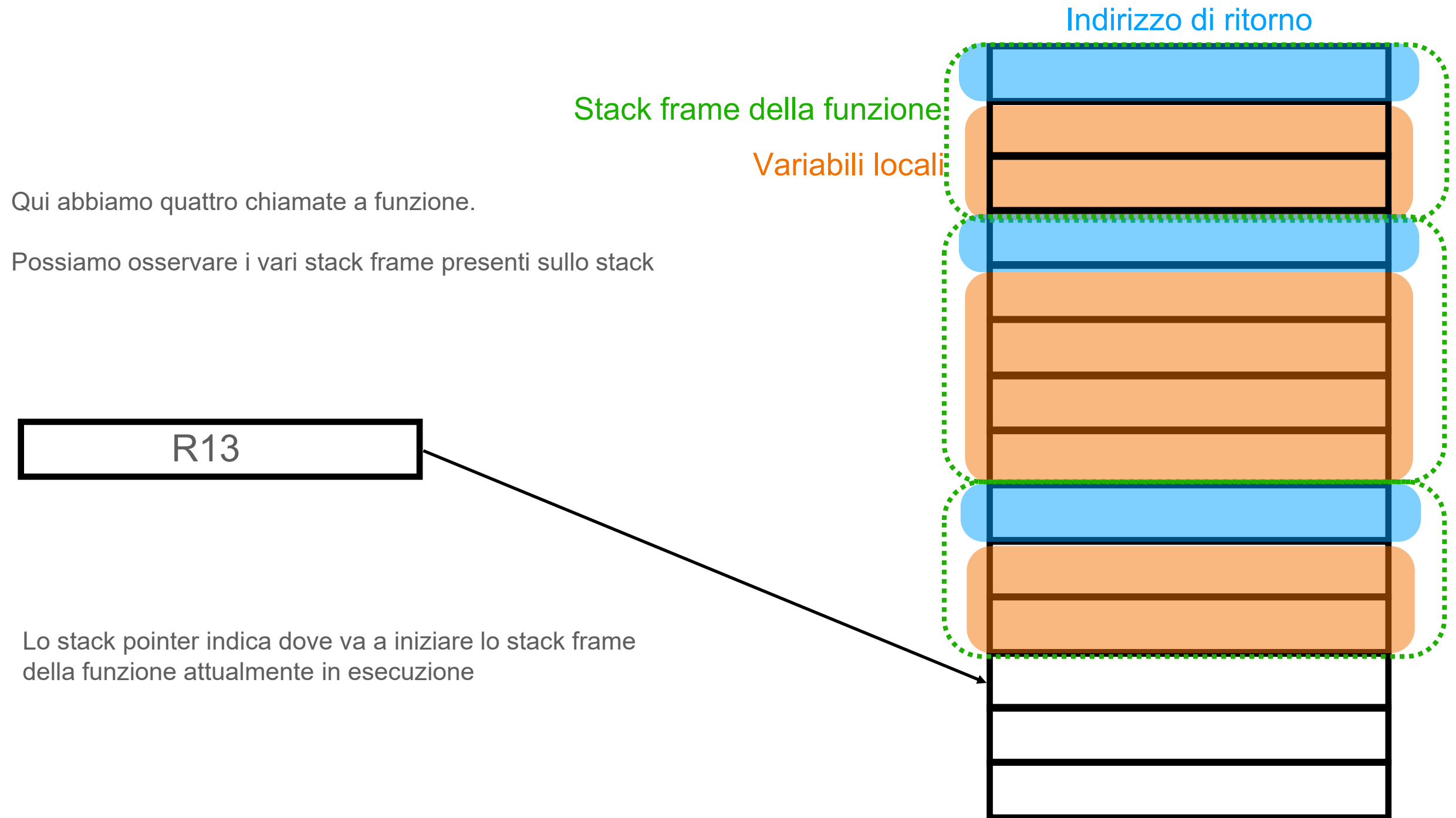


Cosa fare ritornando da una funzione



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

Stack: chiamate innestate



D: recap: cosa ci manca?

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**?

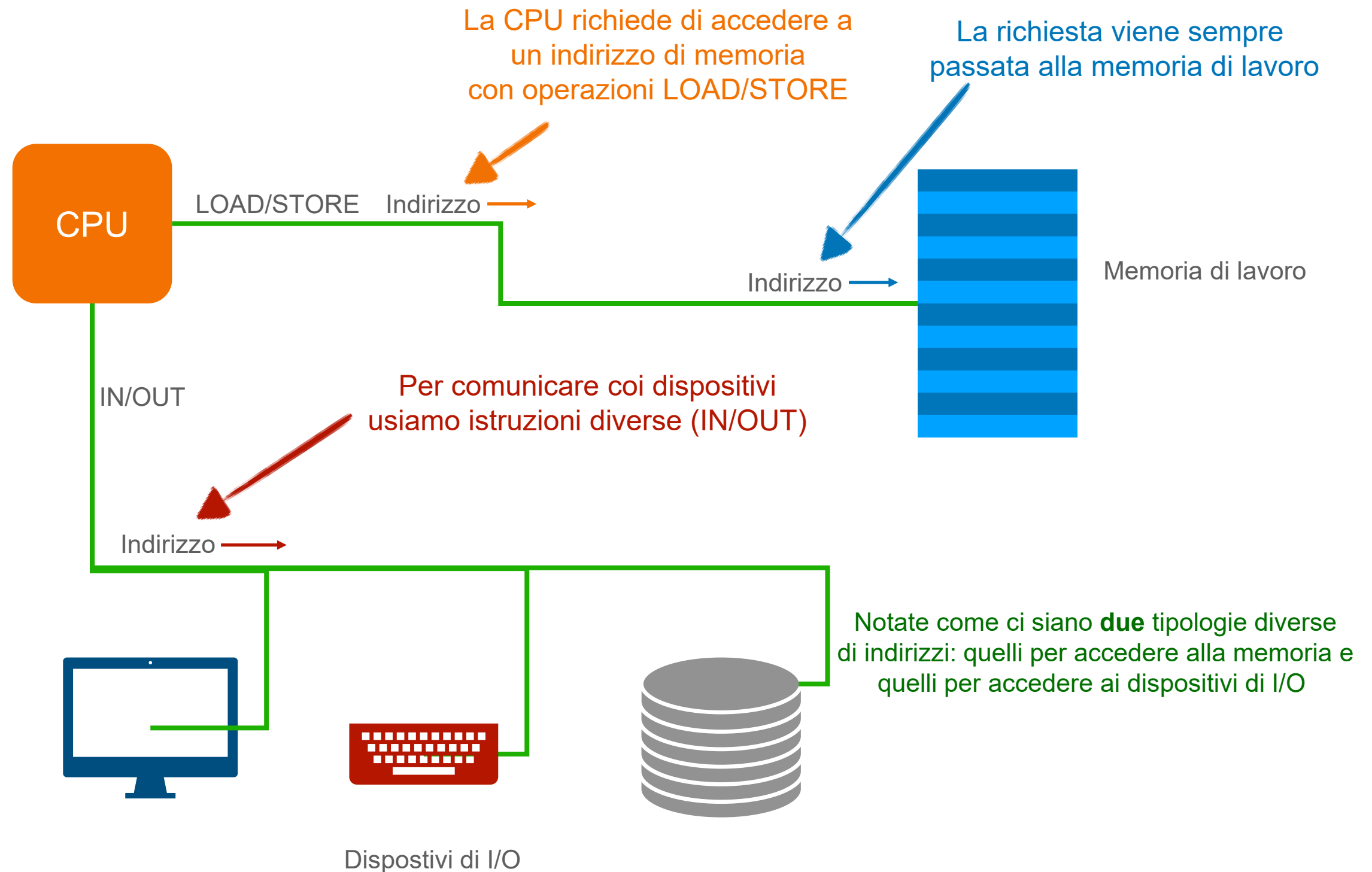
Port mapped I/O

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



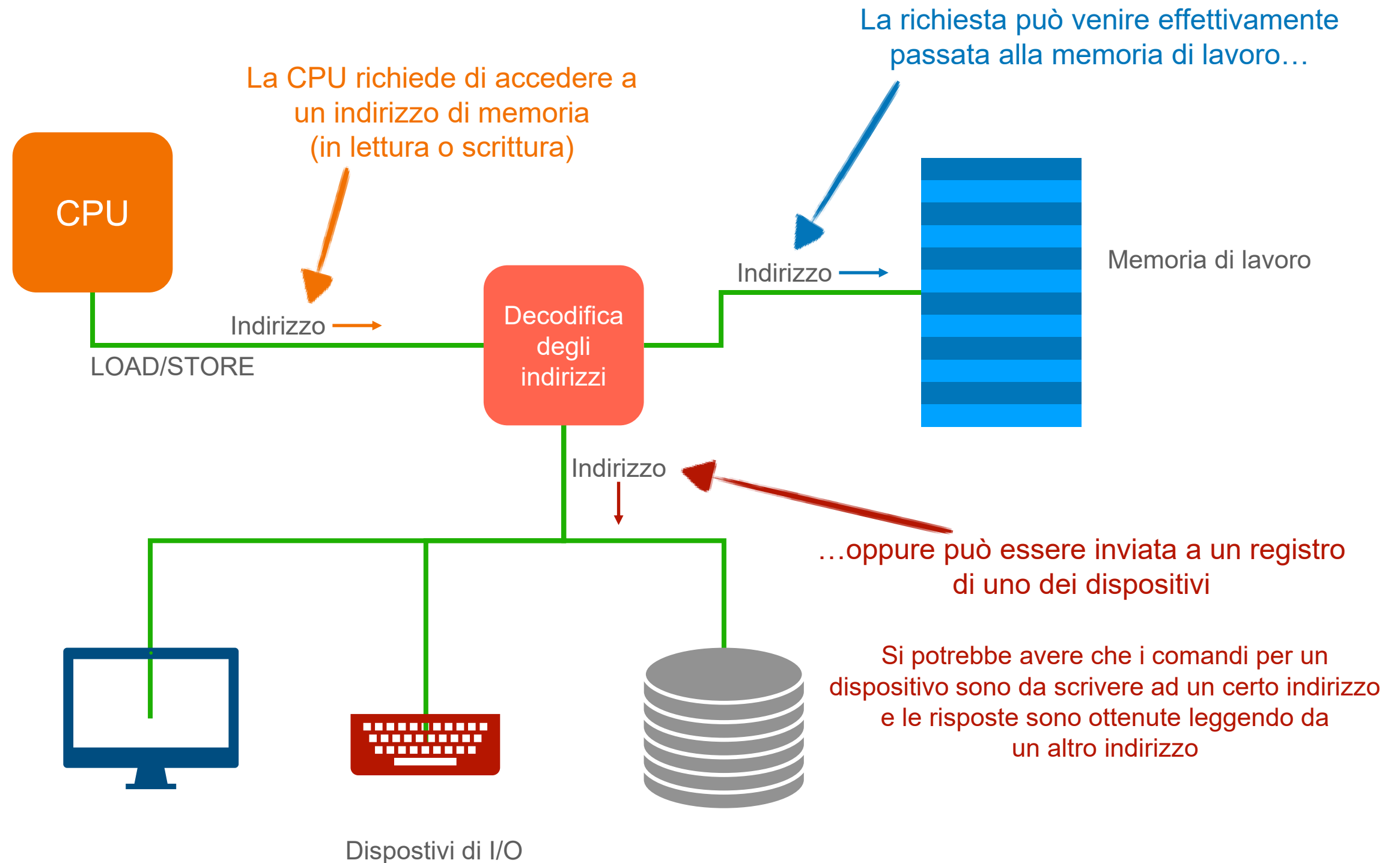
Memory mapped I/O

MMIO

- Una possibile idea è quello di **eliminare** (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**

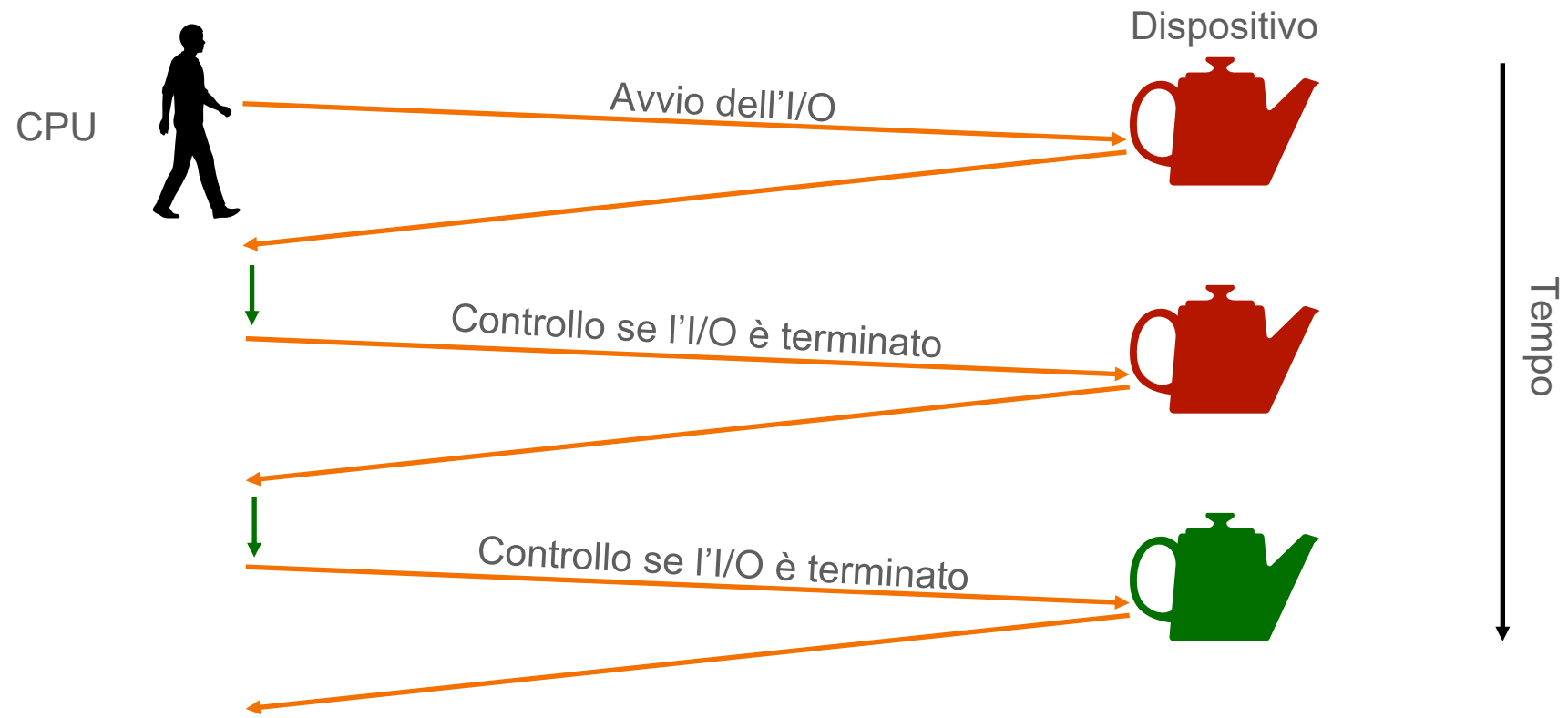
Polling e Interrupt

Due modi di gestire la comunicazione

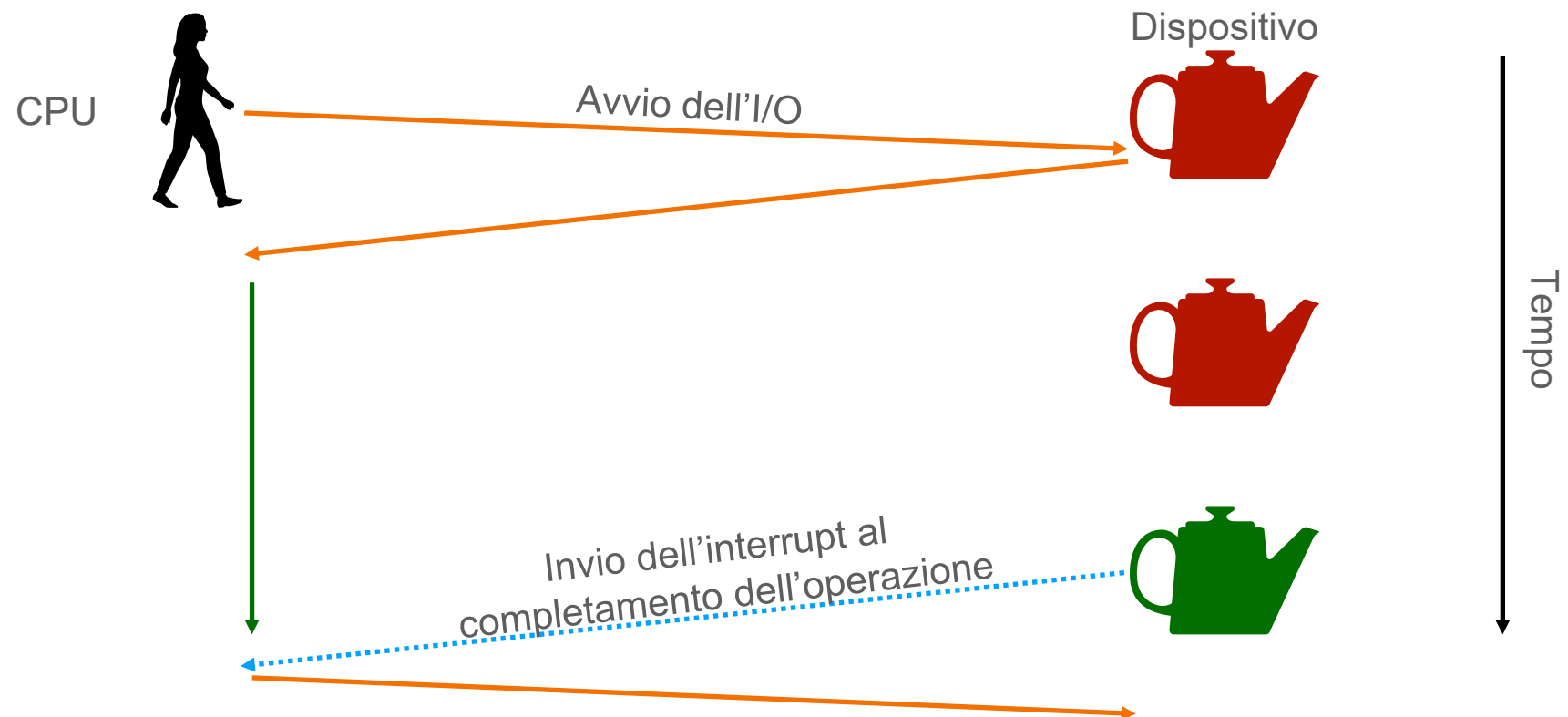
- Il polling può essere dispendioso se il dispositivo la maggior parte delle volte non ha nulla da riportare...
- ...servirebbe che il dispositivo “avvisasse” la CPU quando ha qualcosa di pronto (e.g., operazione completata, nuovo input da leggere, etc.)
- Questo avviene (tramite **supporto hardware**) 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

Polling



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

Polling vs Interrupt

Cosa è meglio?

D: in generale, interrupt è meglio di polling?

Interrupt è meglio quando

- Gli **eventi** sono **irregolari**
- Gli eventi sono **rari**
- La **latenza** è **critica**

→ es. Sensore che segnala un allarme

Polling è meglio quando:

- Gli **eventi** sono **regolari**
- La latenza non è critica
- Il **flusso di esecuzione** deve essere **controllato**

→ es. Sensore termostato

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 e controllare di continuo se ha fatto
 - 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** leggendo e scrivendo **direttamente** in memoria primaria (**RAM**) **grandi moli** di **dati** e avvisando il processore quando la copia è completata

Direct Memory Access

DMA

- Questo dispositivo aggiuntivo (**DMA controller**) può accedere direttamente alla memoria e **fare I/O 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

