



Università degli Studi di Milano
Dipartimento di Informatica "Giovanni Degli Antoni"
Corso di Laurea Triennale in Informatica

Architettura degli Elaboratori II

Laboratorio

Procedure (Lezione 1/2)

Procedure

- Programmando ad alto livello, vogliamo spezzare il programma in unità funzionali dette **procedure**
 - o anche (nei vari linguaggi) **functions, routines, subroutines, subprograms...**
- Esempi:
 - procedura che volge in maiuscolo una data stringa,
 - procedura calcola l'interesse cumulato di una certa somma di denaro,
 - procedura che legge il nome dell'utente da tastiera
 - procedura che verifica una password...
- le procedure vengono **invoke** all'occorrenza, ogni volta che sia necessario
 - dal programma principale,
 - oppure, da un'altra procedura

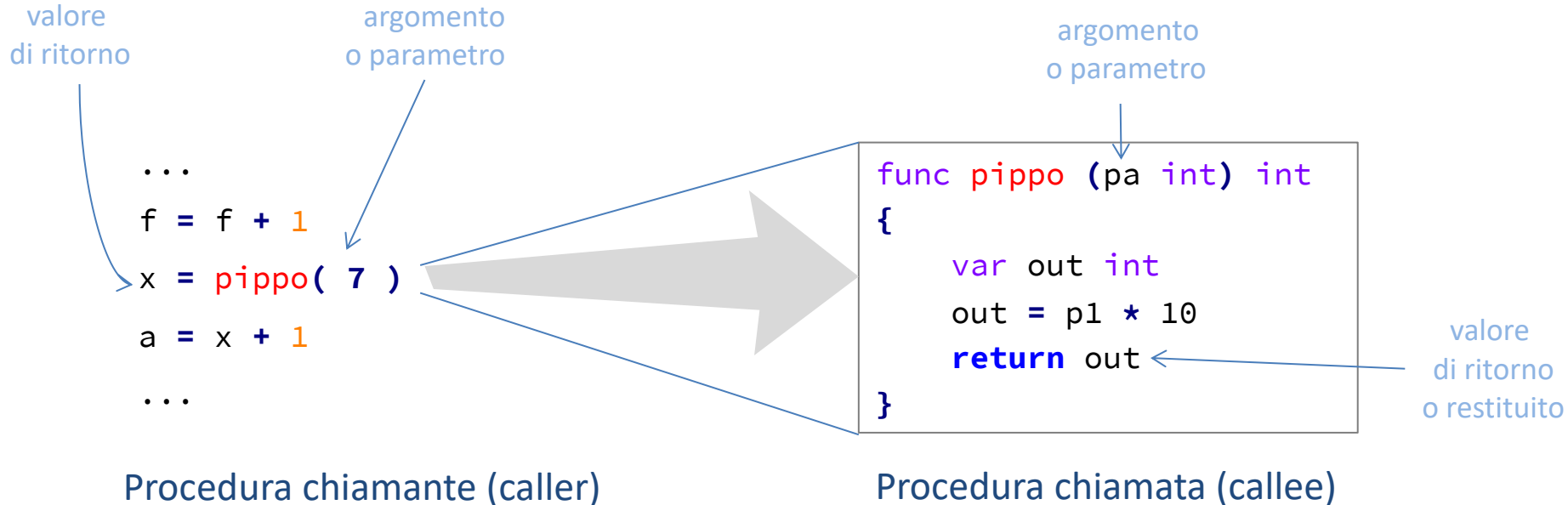
Procedure

- Una procedura è un brano di codice che risolve un sotto-problema specifico, e deve funzionare a prescindere dal contesto nel quale cui viene “invocata”

Implementazione e uso delle procedure

- Chi implementa una procedura (chi scrive il suo codice) è spesso una persona diversa da chi la usa (chi scrive un programma che la invoca)
 - per esempio: il primo è l'autore di una libreria di programma, il secondo è l'utente di questa libreria
 - per sono membri diversi di un team di sviluppo
- I linguaggi ad alto livello forniscono meccanismi con cui i due programmatori possono coordinarsi
 - ad esempio, per specificare...
 - ...quali dati la procedura prenda in input (se alcuno)
 - ...quali dati restituisca in output (se alcuno)
- a basso livello, si adottano invece una serie di convenzioni
 - che sta al programmatore (o al compilatore) rispettare
 - le vediamo in questa lezione

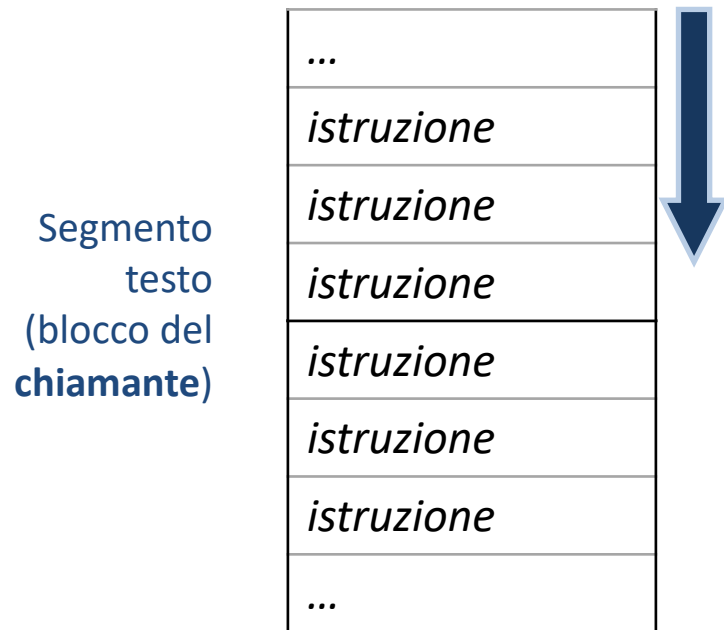
Chiamata a procedura ad alto livello (es: in Go)



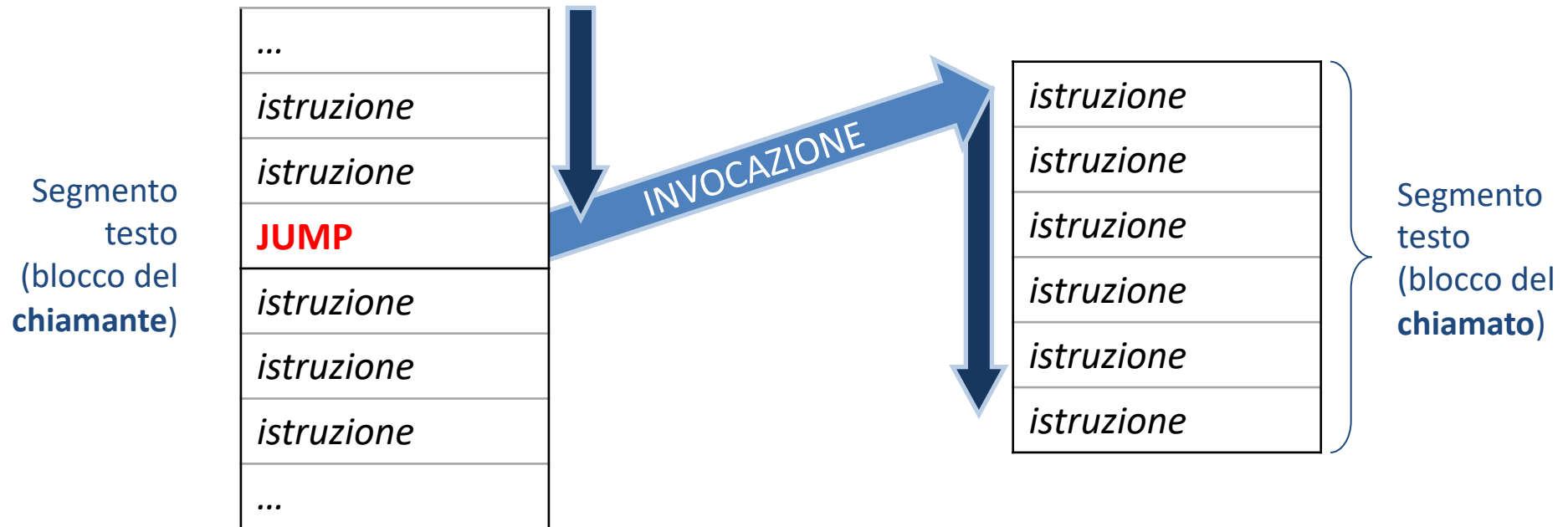
Caller e callee interagiscono attraverso:

- passaggio di **parametri** di input (dal caller al callee)
- **ritorno di valori** di output (dal callee al caller)

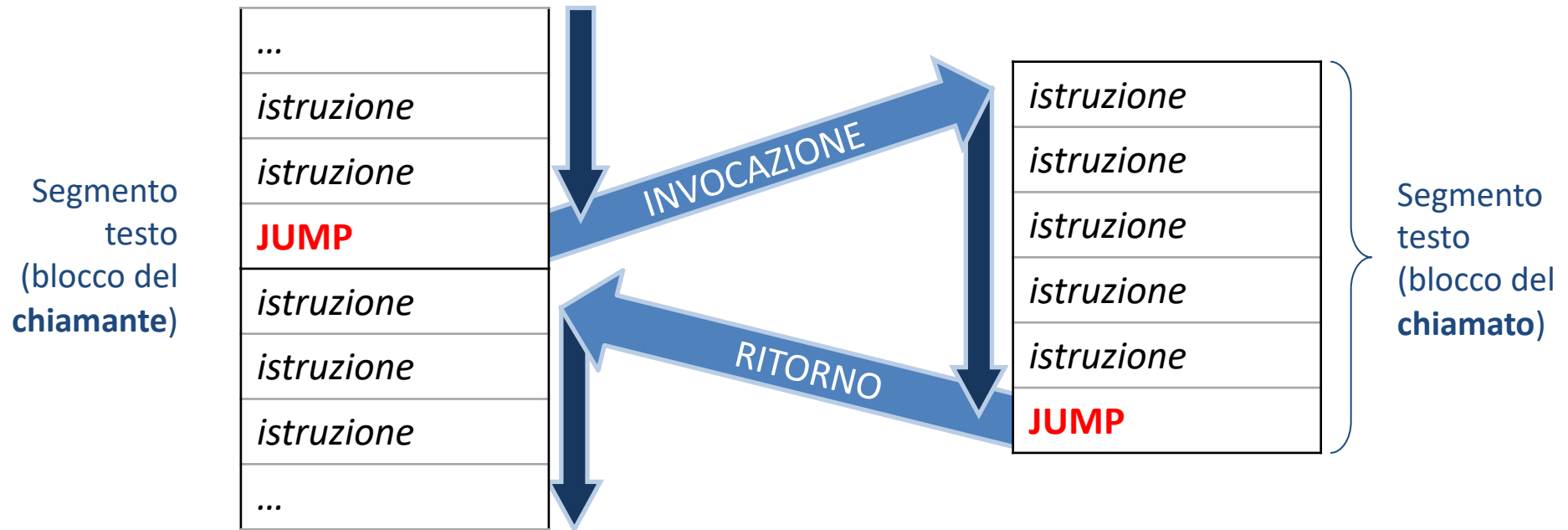
Chiamata a procedura a basso livello




Chiamata a procedura a basso livello



Chiamata a procedura a basso livello




Esempio di procedura nel segmento testo

 **main:**
Questa label indica
l'indirizzo della prima
istruzione del main


<i>istruzione main 1</i>
<i>istruzione main 2</i>
<i>istruzione main 3</i>
<i>istruzione main 4</i>
<i>main invoca printf</i>
<i>istruzione main 6</i>
<i>istruzione main 7</i>
<i>istruzione main ...</i>

⋮

 **printf:**
Questa label indica
l'indirizzo della prima
istruzione della
procedura printf


<i>istruzione printf 1</i>
<i>istruzione printf 2</i>
<i>istruzione printf 3</i>
<i>istruzione printf ...</i>

Esempio di procedura nel segmento testo

 **main:**
Questa label indica l'indirizzo della prima istruzione del main

<i>istruzione main 1</i>
<i>istruzione main 2</i>
<i>istruzione main 3</i>
<i>istruzione main 4</i>
<i>main invoca printf</i>
<i>istruzione main 6</i>
<i>istruzione main 7</i>
<i>istruzione main ...</i>


⋮

 **printf:**
Questa label indica l'indirizzo della prima istruzione della procedura printf

<i>istruzione printf 1</i>
<i>istruzione printf 2</i>
<i>istruzione printf 3</i>
<i>istruzione printf ...</i>


- **Invocare printf** significa eseguire un salto non condizionato all'indirizzo della prima istruzione della procedura printf

Esempio di procedura nel segmento testo

 **main:**
Questa label indica l'indirizzo della prima istruzione del main

<i>istruzione main 1</i>
<i>istruzione main 2</i>
<i>istruzione main 3</i>
<i>istruzione main 4</i>
<i>main invoca printf</i>
<i>istruzione main 6</i>
<i>istruzione main 7</i>
<i>istruzione main ...</i>

⋮

 **printf:**
Questa label indica l'indirizzo della prima istruzione della procedura printf

<i>istruzione printf 1</i>
<i>istruzione printf 2</i>
<i>istruzione printf 3</i>
<i>istruzione printf ...</i>


- **Invocare printf** significa eseguire un **salto non condizionato all'indirizzo della prima istruzione della procedura printf**

ATTENZIONE!

- Non basta fare `j printf` perché devo ricordare che quando printf termina si deve fare un altro salto non condizionato per riprendere il flusso di esecuzione del chiamante


 **da qui**

Esempio di procedura nel segmento testo

 **main:**
Questa label indica l'indirizzo della prima istruzione del main

<i>istruzione main 1</i>
<i>istruzione main 2</i>
<i>istruzione main 3</i>
<i>istruzione main 4</i>
<i>main invoca printf</i>
<i>istruzione main 6</i>
<i>istruzione main 7</i>
<i>istruzione main ...</i>

⋮

 **printf:**
Questa label indica l'indirizzo della prima istruzione della procedura printf

<i>istruzione printf 1</i>
<i>istruzione printf 2</i>
<i>istruzione printf 3</i>
<i>istruzione printf ...</i>

- **Invocare printf** significa eseguire un **salto non condizionato all'indirizzo della prima istruzione della procedura printf**

ATTENZIONE!

- Non basta fare `j printf` perché devo ricordare che quando printf termina si deve fare un altro salto non condizionato per riprendere il flusso di esecuzione del chiamante

 **da qui**

- **L'indirizzo a cui inizia printf** è unico
- **L'indirizzo di ritorno** è diverso ad ogni invocazione: **non è unico!**
- Oltre al salto a printf devo anche salvare l'indirizzo a cui riprendere l'esecuzione al termine della procedura

JAL: Jump and Link

Registro:	\$0	\$1	\$2	\$3	\$4	\$5	\$6	\$7
Sinonimo:	\$r0	\$at	\$v0	\$v1	\$a0	\$a1	\$a2	\$a3
Registro:	\$8	\$9	\$10	\$11	\$12	\$13	\$14	\$15
Sinonimo:	\$t0	\$t1	\$t2	\$t3	\$t4	\$t5	\$t6	\$t7
Registro:	\$16	\$17	\$18	\$19	\$20	\$21	\$22	\$23
Sinonimo:	\$s0	\$s1	\$s2	\$s3	\$s4	\$s5	\$s6	\$s7
Registro:	\$24	\$25	\$26	\$27	\$28	\$29	\$30	\$31
Sinonimo:	\$t8	\$t9	\$k0	\$k1	\$gp	\$sp	\$s8	\$ra



RETURN ADDRESS

A quale indirizzo saltare per tornare al chiamante?

- Non può essere un indirizzo fissato (es, indicato da una label) perché la procedura può essere invocata da qualsiasi riga del programma!
 - anche da più parti dello stesso programma
- In MIPS dedichiamo un registro a memorizzare l'indirizzo di ritorno:
 - `$ra` : «Return Address»
 - è il \$31, ma non è necessario saperlo o ricordarselo, basta chiamarlo col suo sinonimo
- Le istruzioni jump hanno una variante «and link» che, prima di sovrascrivere il PC, salvano in `$ra` il valore PC+4 :
 - `jal <indirizzo>` : Jump-and-link
 - `jalr <registro>` : Jump-and-link register
- Il codice invocante salta all'indirizzo di partenza della procedura con `jal` (o `jalr`)
 - Come per qualsiasi jump, usare le label indentificare l'indirizzo di arrivo
- La procedura restituisce il controllo al chiamante con jump register:
`jr $ra`

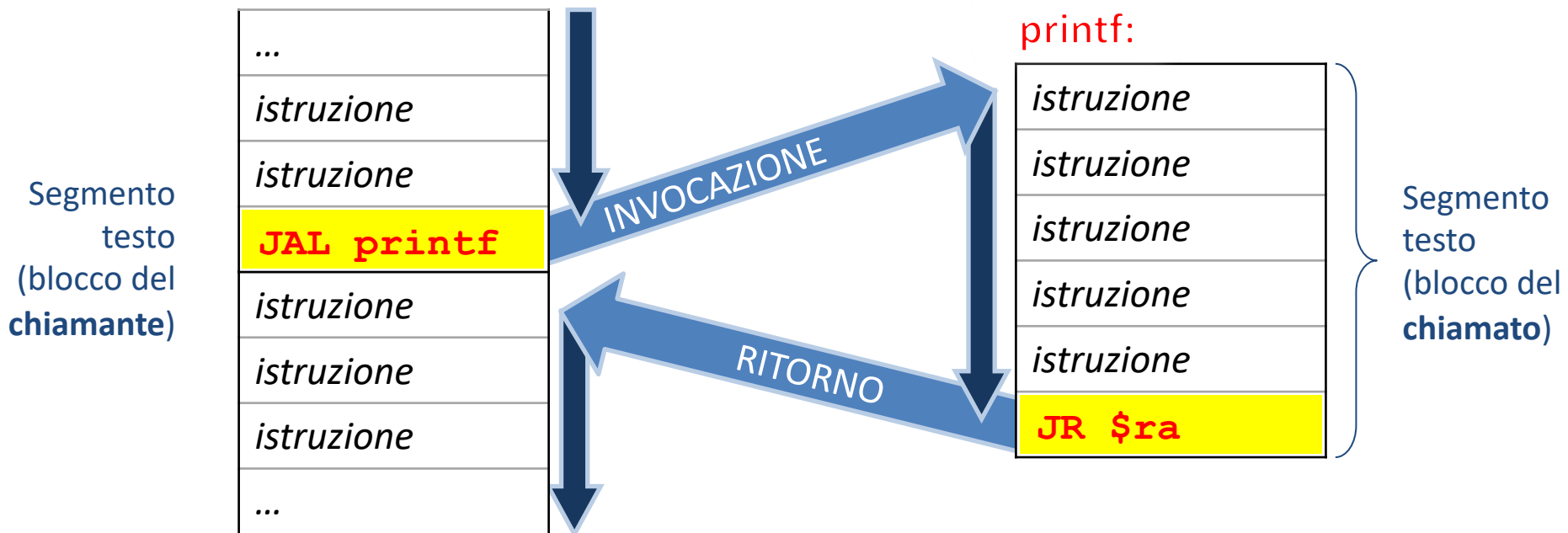
Salti di invocazione e ritorno

Salto di invocazione:

- Indicare l'inizio della procedura con una etichetta
- Saltare con una **jal** a quell'etichetta

Salto di ritorno

- Saltare con una **jr** al Return address



Progetti multi-file e linking

- Un progetto multi-file è costituito da più di un file sorgente
- L'assembler traduce separatamente ogni file sorgente da linguaggio assembly a linguaggio macchina
 - producendo un codice in linguaggio macchina
- Linking: i codici prodotti vengono assemblati (linked) in un unico codice
 - Semplicemente, concatenando le loro parti “text” fra loro, e le loro parti “data” fra loro
 - Nota: gli indirizzi finali a cui sono memorizzati istruzioni e dati divengono dunque noti solo dopo il linking
 - Solo dopo il linking, le nostre etichette (nell'assembly) vengono convertite in indirizzi (nel linguaggio macchina)

Progetti multi-file e linking

- E' possibile condividere etichette fra file diversi
 - Cioè, consentire ad un file sorgente di usare un'etichetta definita in un file diverso dello stesso progetto
 - Bisogna però dichiarare che queste etichette sono “globali”, inserendo, nel file che le definisce, l'apposita direttiva:

```
.globl etichetta
```

- Nota: etichette diverse usate da file diversi possono condividere uno stesso nome (per es: “loop”, “fine”, “else”, etc), se non sono globali

Progetti multi-file e procedure

- Un caso classico è porre una procedura o un insieme di procedure in un file separato di un progetto multifile
 - Analogo ad una libreria ad alto livello
 - Per poter essere invocate dagli altri file, le etichette di inizio delle procedure devono essere definite come globali
- Un unico file separato conterrà la funzione “main”
 - da cui per convenzione inizia l’esecuzione

Progetti multi-file in MARS

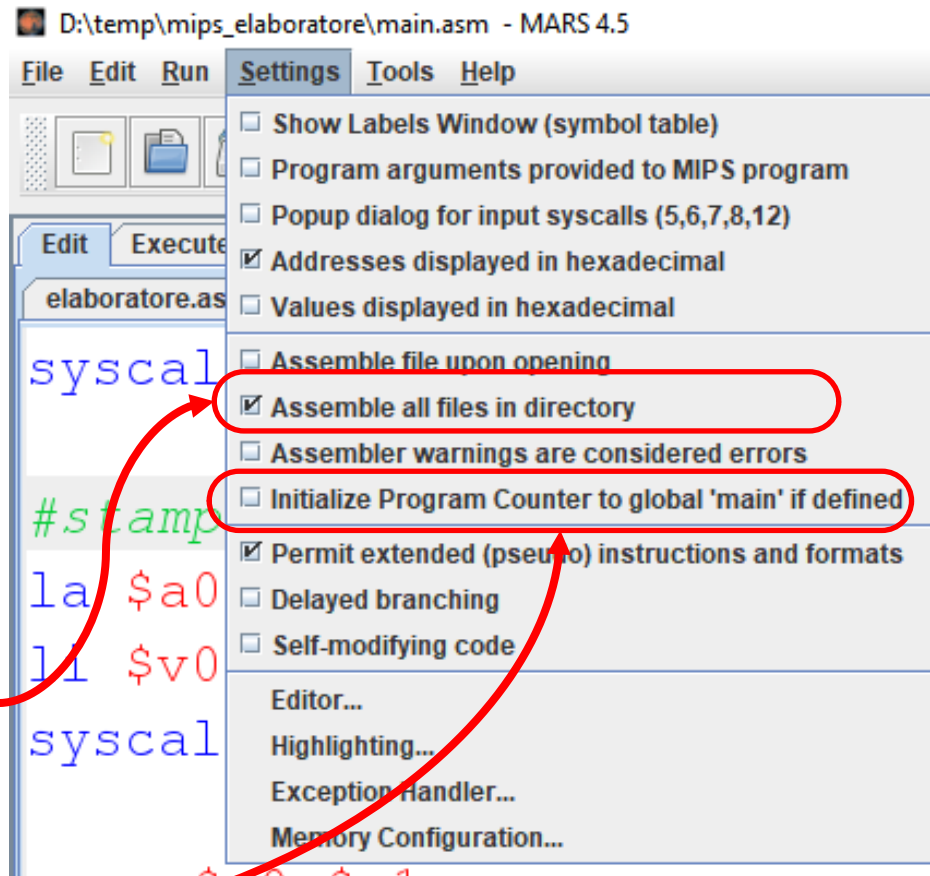
- In MARS, per definire un progetto multi-file è necessario ...

1. porre tutti i suoi sorgenti in un'apposita cartella del file system

2. abilitare questa opzione

3. abilitare questa opzione

4. e dichiarare anche l'etichetta main come "globale")



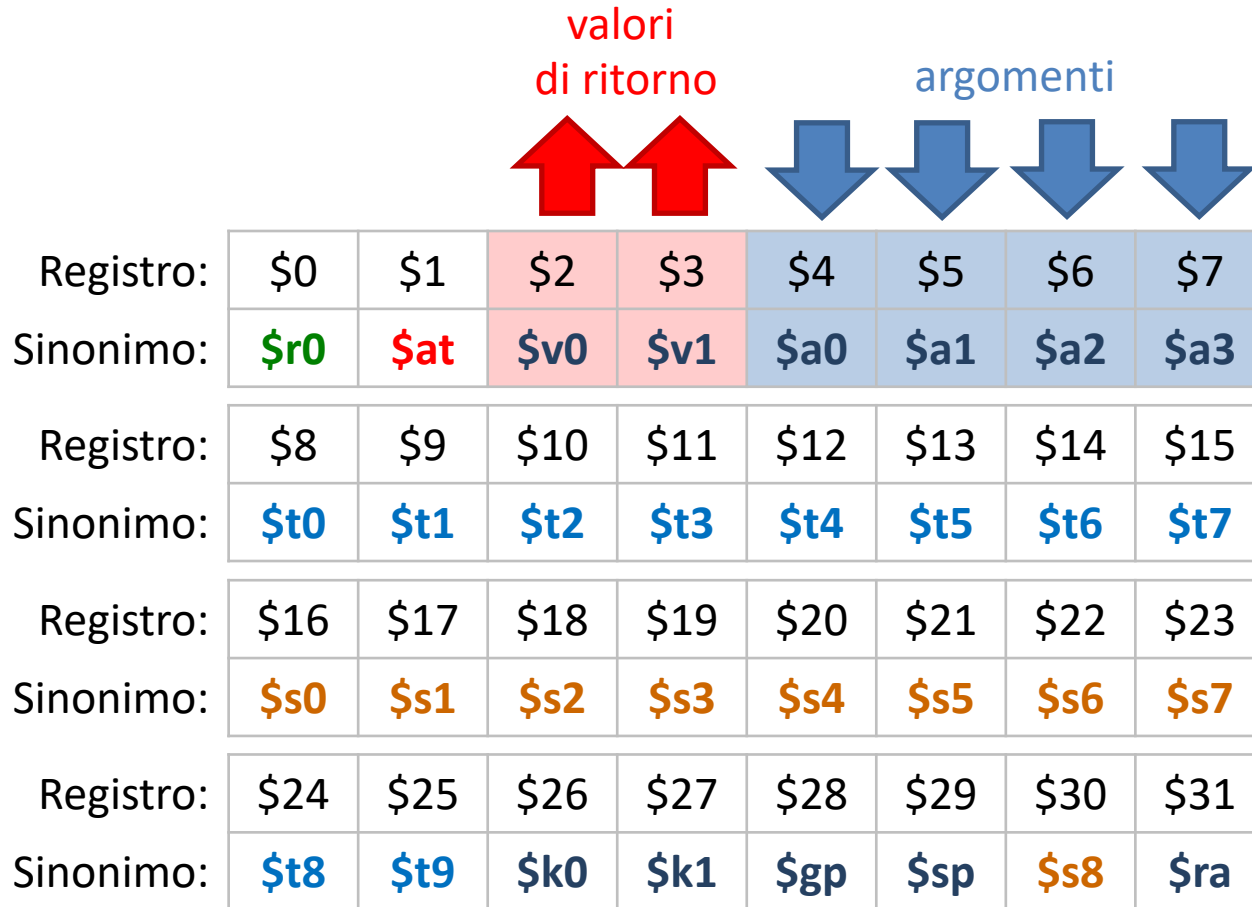
Come il chiamante comunica alla procedura i «parametri»

- Molte procedure si aspettano degli input
 - es: una procedura che volge in maiuscolo una stringa deve sapere l'indirizzo della stringa su cui lavorare
- Ad alto livello, questi sono gli **argomenti** (o i **parametri**) della procedura
- In MIPS, dedichiamo alcuni registri a memorizzare questi argomenti: **\$a0**, **\$a1**, **\$a2**, **\$a3** (a = argomento)
- Convenzione: (che sta ai programmatori / compilatori / studenti rispettare)
 - Il chiamante mette i valori dei parametri in **\$a0..\$a3** prima di invocare la procedura (quelli necessari)
 - La procedura assumerà di trovare gli input necessari in **\$a0..\$a3**


Come la procedura comunica al chiamante i suoi «valori di ritorno»

- Molte procedure restituiscono degli output
 - es: una procedura che calcola l'interesse cumulato deve comunicare al chiamante questo valore
- Ad alto livello, questo è il **valore di ritorno** della procedura (uno o più)
- In MIPS, dedichiamo alcuni registri a memorizzare i valori di ritorno: **\$v0**, **\$v1** (v = valore di ritorno)
- Convenzione: (che sta a noi rispettare)
 - Prima di restituire il controllo, la procedura mette in **\$v0** (e/o **\$v1**) il valore/i da restituire
 - Al ritorno il caller assume di avere in **\$v0** (e/o **\$v1**) il valore/i restituito/i dalla procedura

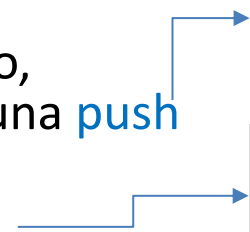
Input e output di una procedura



Come facciamo se abbiamo bisogno di più di 4 argomenti?

- Si usa lo stack
 - Dal 5to argomento il poi:
 - Il chiamante: prima di invocare, mette l'argomento nello stack con una **push**
 - Il chiamato: per prima cosa, prende l'argomento dallo stack, con una **pop**
- 
- ```
addi $sp $sp -4
sw $** ($sp)
```
- ```
lw $** ($sp)
addi $sp $sp 4
```
- Nota: chiamante e chiamato devono sapere che questo è il caso per ogni data funzione
 - Quindi, quanti argomenti servono oltre i primi 4
 - Lo stack viene compromesso irrimediabilmente se entrambi la procedura e il codice che la invoca non rispettano questa convenzione (viene fatta una push o, peggio, una pop di troppo)
 - **NOTA:** in altri ISA, questo è il modo convenzionale di passare *tutti* gli argomenti

Come facciamo se abbiamo bisogno di più di 2 valori di ritorno?

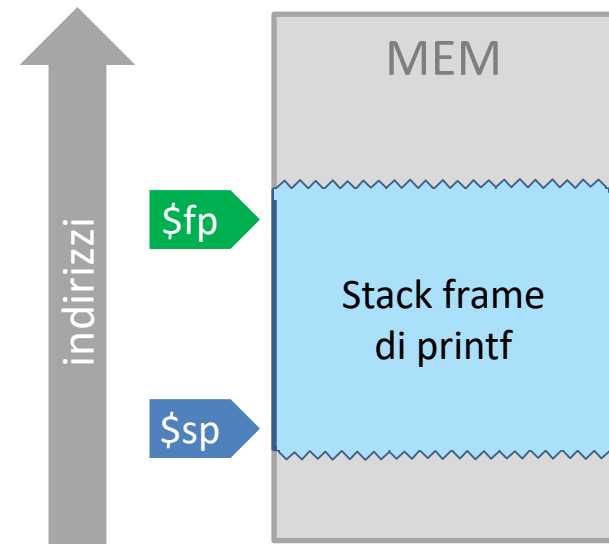
- Si usa lo stack
 - Dal 3zo valore di ritorno il poi:
 - Il chiamato: prima di restituire il controllo, mette il valore prodotto nello stack con una **push**
 - Il chiamante: dopo l'invocazione, prende il valore dallo stack, con una **pop**
- 
- ```
addi $sp $sp -4
sw $** ($sp)

lw $** ($sp)
addi $sp $sp 4
```
- Nota: chiamante e chiamato devono sapere che questo è il caso per ogni data funzione
    - Quindi, quanti valori vengono restituiti oltre i primi due
    - Lo stack viene compromesso irrimediabilmente se entrambi la procedura e il codice che la invoca non rispettano questa convenzione (viene fatta una push o, peggio, una pop di troppo)
  - **NOTA:** in altri ISA, questo è il modo convenzionale di restituire *tutti* i valori



# Record di attivazione – Stack frame

- Spesso una **procedura** ha bisogno di usare la memoria
  - esempio: memorizzare quelle che ad alto livello sono le sue **variabili locali**
  - Sono dati dinamici: vanno tenuti in memoria solo durante l'esecuzione della procedura, ma non più al suo termine
- Dedichiamo ad ogni procedura in esecuzione una sua area di memoria sullo stack, detta **record di attivazione** o **stack frame**
- MIPS riserva due registri per indirizzare lo **stack frame** della procedura attualmente in esecuzione: da **\$sp** (stack pointer) a **\$fp** (frame pointer) compresi

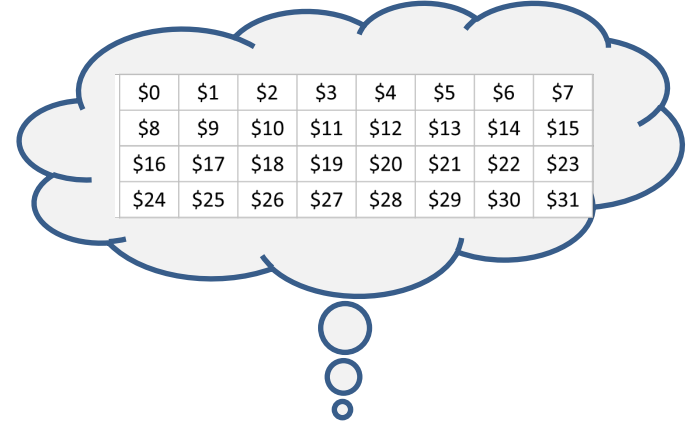
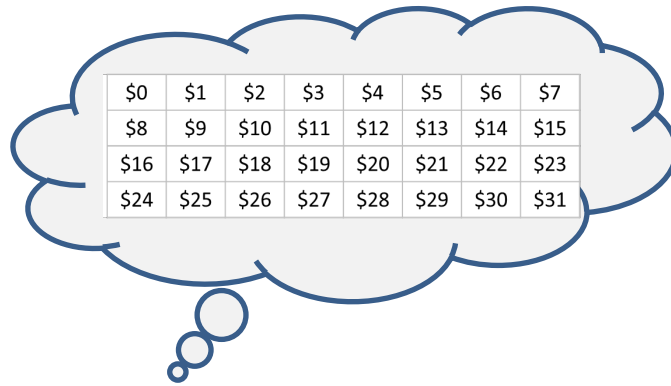


# Allocazione dei frame

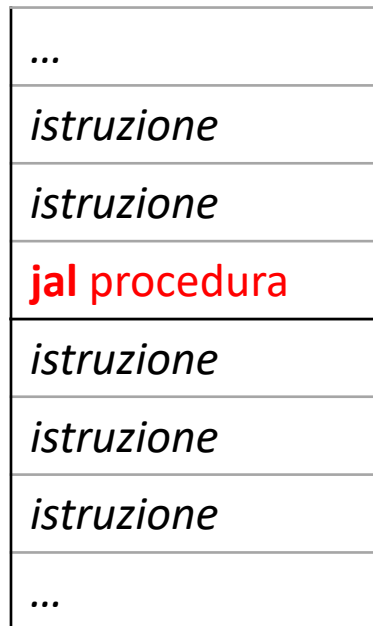
Situazione: una procedura A invoca un'altra procedura B

- Durante l'esecuzione di B, il record di attivazione di A va ancora mantenuto in memoria:
  - A non è ancora terminato
  - quando B restituisce il controllo ad A, A dovrà ancora lavorare con le variabili che sono nel suo frame
- Osserviamo che la catena di chiamate e ritorni da procedura segue un ordine LIFO: l'ultima procedura ad essere stata invocata (last in) è la prima a restituire il controllo al chiamante (first out)
- Soluzione: i record di attivazione si **impilano** in memoria sullo **stack**
  - quando una procedura viene invocata un nuovo record di attivazione viene impilato nello stack
  - l'ultimo record di attivazione impilato viene rimosso dallo stack quando una procedura termina

# Problema: i registri sono gli stessi!

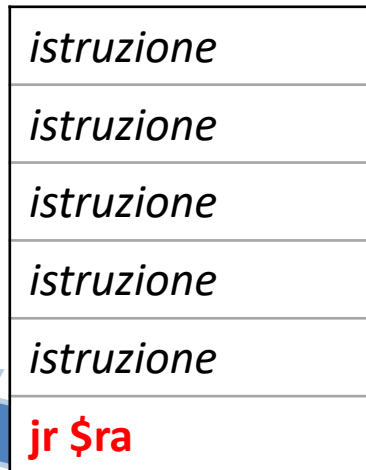


Segmento  
testo  
(blocco del  
chiamante)



INVOCAZIONE

RITORNO



Segmento  
testo  
(blocco del  
chiamato)

# Registri \$s e \$t

- Problema: i registri sono usati tanto dalla procedura quanto dal chiamante
  - Quindi, dopo una chiamata ad una procedura, il chiamante rischia di trovare i registri che stava utilizzando completamente cambiati («sporcati»)
- Ogni linguaggio assembly usa delle convenzioni per consentire a chiamante e chiamato di usare i registri senza interferire un con l'altro
- In MIPS, adottiamo una convenzione:
  - gli otto registri **\$s0 .. \$s7** (s = save) devono essere **preservati** dalla procedura: quando la procedura restituisce il controllo, il chiamante deve trovare in questi registri gli stessi valori che avevano al momento dell'invocazione
  - i dieci registri **\$t0 .. \$t9** (t = temp) possono invece essere modificati da una procedura: il chiamante sa che invocare una procedura potrebbe modificare («sporcare») questi registri

# Convenzione sull'uso dei registri da parte delle procedure

|           |      |      |      |      |      |      |      |      |
|-----------|------|------|------|------|------|------|------|------|
| Registro: | \$0  | \$1  | \$2  | \$3  | \$4  | \$5  | \$6  | \$7  |
| Sinonimo: | \$r0 | \$at | \$v0 | \$v1 | \$a0 | \$a1 | \$a2 | \$a3 |
| Registro: | \$8  | \$9  | \$10 | \$11 | \$12 | \$13 | \$14 | \$15 |
| Sinonimo: | \$t0 | \$t1 | \$t2 | \$t3 | \$t4 | \$t5 | \$t6 | \$t7 |
| Registro: | \$16 | \$17 | \$18 | \$19 | \$20 | \$21 | \$22 | \$23 |
| Sinonimo: | \$s0 | \$s1 | \$s2 | \$s3 | \$s4 | \$s5 | \$s6 | \$s7 |
| Registro: | \$24 | \$25 | \$26 | \$27 | \$28 | \$29 | \$30 | \$31 |
| Sinonimo: | \$t8 | \$t9 | \$k0 | \$k1 | \$fp | \$sp | \$s8 | \$ra |



deve rimanere  
invariato



può essere modificato  
dalla procedura

# Anche detti: registri caller-saved e callee-saved

## Caller-saved

*«salvati dal chiamante»*

*sono i registri rispetto a cui **non** vige una convenzione di preservazione attraverso chiamate a procedura*

**\$t0 ... \$t9, \$a0 ... \$a3, \$v0, \$v1**

Un callee è libero di sovrascrivere questi registri

Se il chiamante vuole essere sicuro di non perderne i loro valori deve salvarli sullo stack prima della chiamata a procedura

Esempio: `main` ha un dato importante nel registro `$t0`, prima di invocare `f` salva `$t0` sullo stack, una volta riacquisito il controllo lo ripristina.

## Callee-saved

*«salvati dal chiamato»*

*sono i registri rispetto cui la convenzione esige che vengano preservati attraverso chiamate a procedura*

**\$s0 ... \$s9 \$ra \$sp \$fb**

un callee non può sovrascrivere questi registri, il chiamante si aspetta che restino invariati dopo la chiamata a procedura.

Se il chiamato li vuole usare, deve prima salvarli sullo stack per poi ripristinarli una volta terminato, prima della JR \$RA

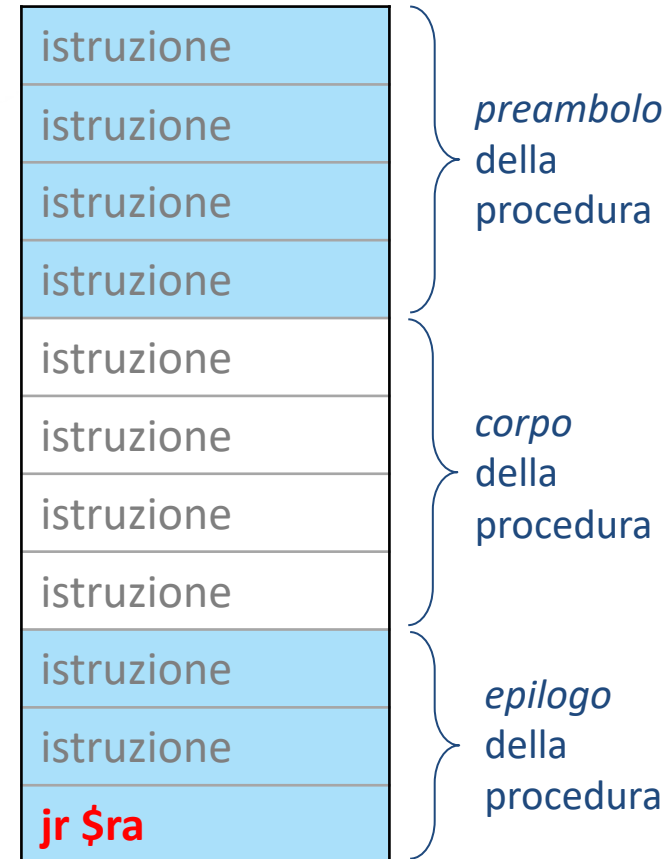
Esempio: `main` ha un dato importante nel registro `$s1` e invoca `f`; `f` salva `$s1` sullo stack prima di utilizzarlo, una volta terminato lo ripristina.

# Registri \$s e \$t: per la procedura

La procedura può rispettare la convenzione attraverso diversi modi

- Modo 1: non scrivere mai i registri \$s
  - usare quindi solo i registri \$t
- Modo 2: salvare i registri \$s nello stack
  - Prima di scrivere su un dato registro \$s, (ad esempio, nel «*preambolo*» della proc.) o cmq salvarne una copia con una **push** nello stack
  - Poi, usare questi registri come normale
  - prima di restituire il controllo al chiamante, (ad esempio, nel «*epilogo*» della proc, subito prima della *jr \$ra* finale) ripristinare il valore originale di questi registri con una **pop**
  - Nota: dal punto di vista del chiamante, lo stack rimane inalterato

miaFunz:



# Conclusione: piccolo manuale per invocare una procedura

1. Se necessario, salvare una copia dei registri \$t
  - con copie nei registri \$s, oppure sullo stack, con delle «push»
  - vale anche per \$a0..\$a3, \$v0 e \$v1
2. Caricare i parametri della procedura (se previsti)
  - I primi 4: in \$a0.. \$a3
  - Gli eventuali successivi, con delle push nello stack
3. Invocare la procedura: jal <label>
4. Leggere i valori restituiti (se previsti)
  - I primi 2, da \$v0 e \$v1
  - Gli eventuali successivi, con delle pop dallo stack
5. Se necessario, ripristinare il valore dei registri salvati nel passo 1
  - Con copie dai registri \$s, oppure dallo stack, con delle «pop»



# Conclusione: piccolo manuale per scrivere una procedura

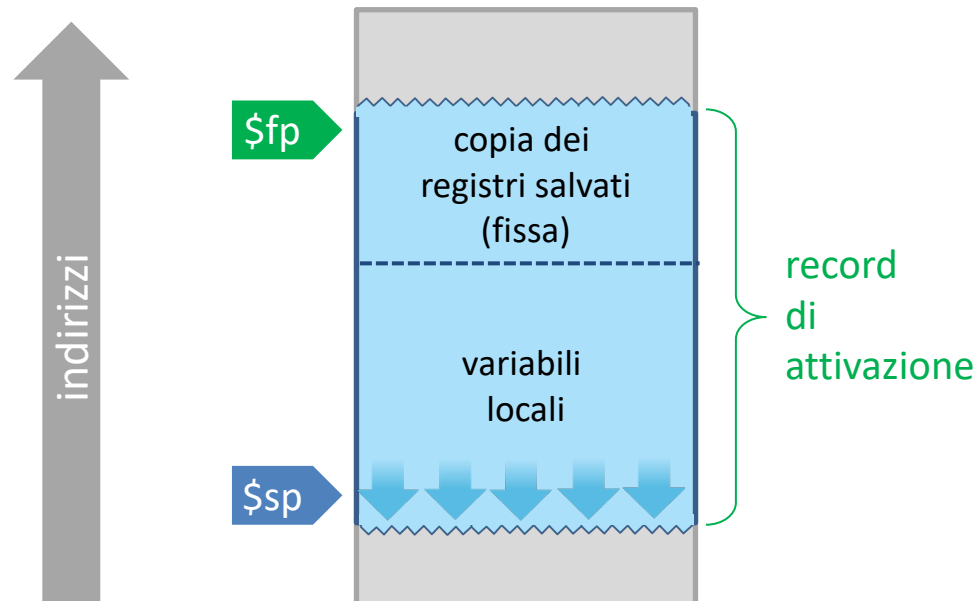
1. «preambolo» :  
salvare una copia dei registri \$s che si intende usare nello stack frame
  - Con delle push sullo stack
2. «corpo» : implementare la procedura (scrivere codice)
  - leggendo gli eventuali input da \$a0 .. \$a3  
oltre al 4to: con delle pop dallo stack
  - scrivendo liberamente su \$t0 .. \$t9
  - scrivendo su \$s0 .. \$s8 che siano stati salvati precedentemente
  - scrivendo l'eventuale output in \$v0 (e/o \$v1),  
e, oltre al secondo, con delle push sullo stack
3. «epilogo» :  
ripristinare tutti i registri salvati nel passo 1
  - con delle pop dallo stack
4. restituire il controllo al chiamante
  - jr \$ra

**NOTA:** se la procedura ne invoca un'altra, la situazione si complica, va aggiunta un ulteriore passo.  
Vedi prossima lezione.

# A cosa serve \$fp?

## Potrei usare solo \$sp?

- \$sp può variare nel corso della procedura: viene decrementato quando una nuova procedura aggiunge il suo record di attivazione sullo stack
- \$fp invece non cambia durante l'esecuzione della procedura
- \$fp può essere comodo per tener traccia di dove sono stati salvati i registri

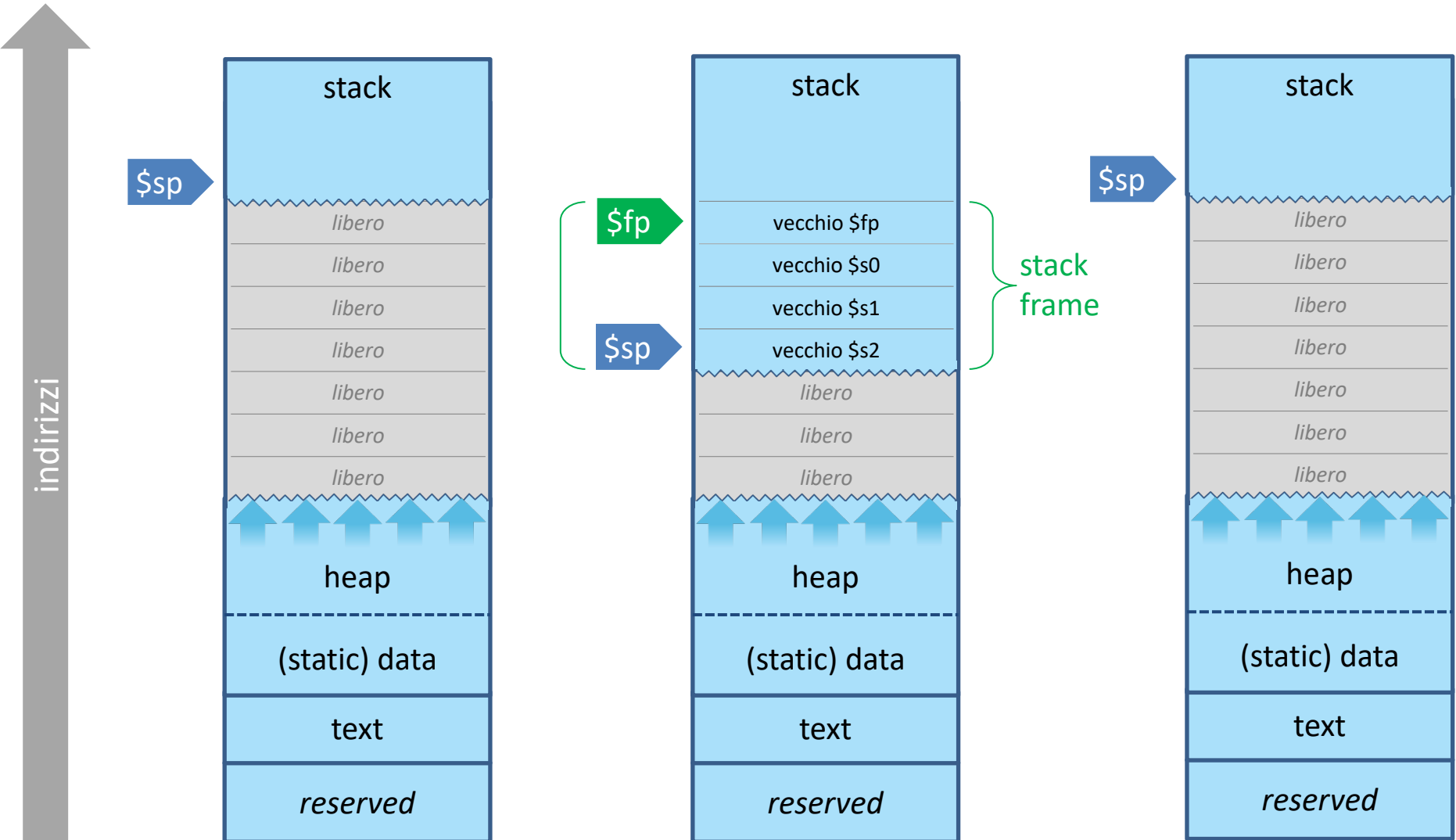


# Esempio

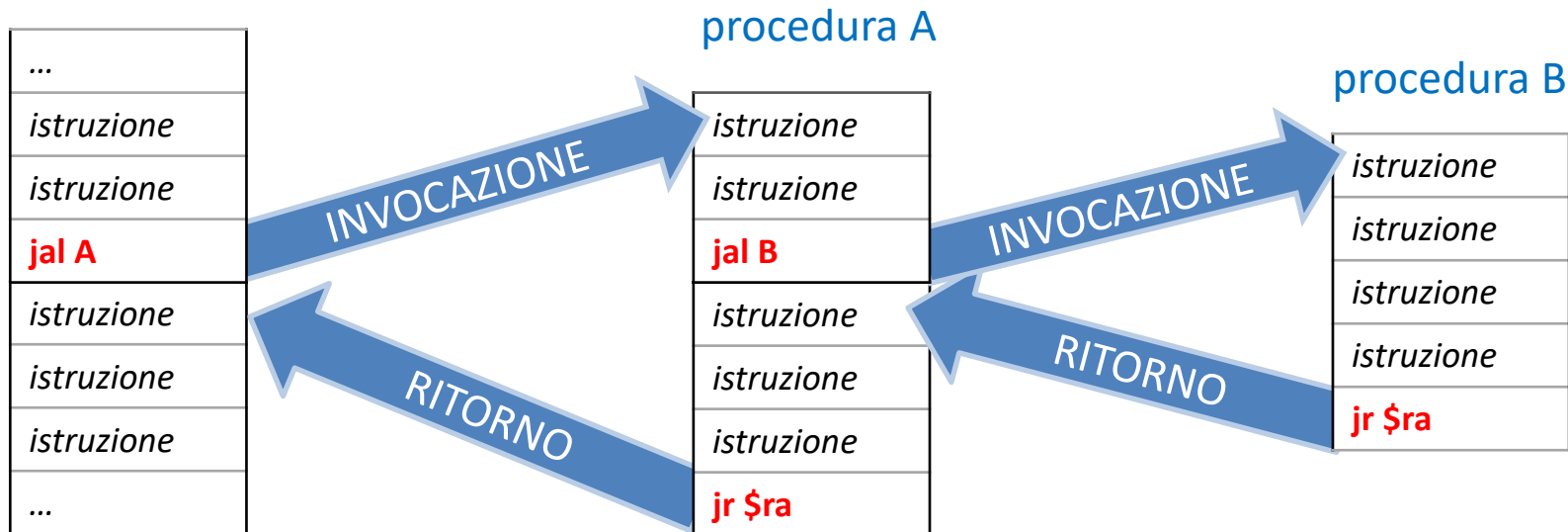
Prima della chiamata

Durante la procedura

Dopo il ritorno



# Prossima lezione: invocazioni di procedura annidate





Università degli Studi di Milano  
Dipartimento di Informatica "Giovanni Degli Antoni"  
Corso di Laurea Triennale in Informatica

# Architettura degli Elaboratori II

## Laboratorio