

# Architettura degli Elaboratori

Funzioni, activation record e indirizzi



SAPIENZA  
UNIVERSITÀ DI ROMA

Alessandro Checco

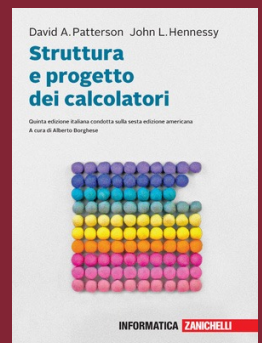
[checco@di.uniroma1.it](mailto:checco@di.uniroma1.it)

Special thanks and credits:

Andrea Sterbini, Iacopo Masi, Claudio di  
Ciccio

[S&PdC]

2.8–2.10, 2.14

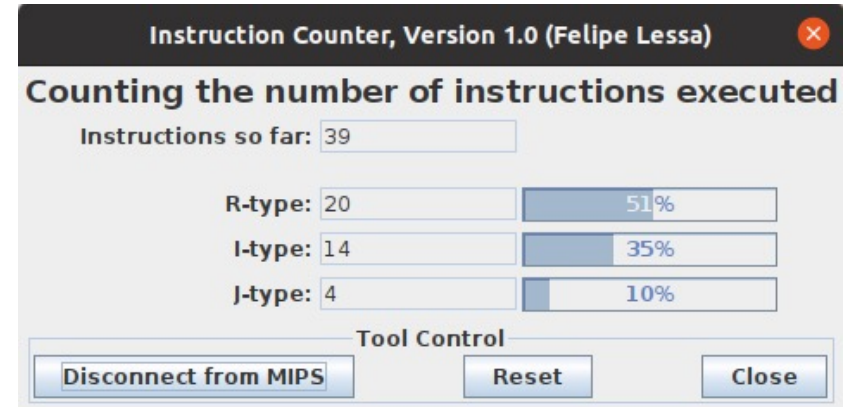
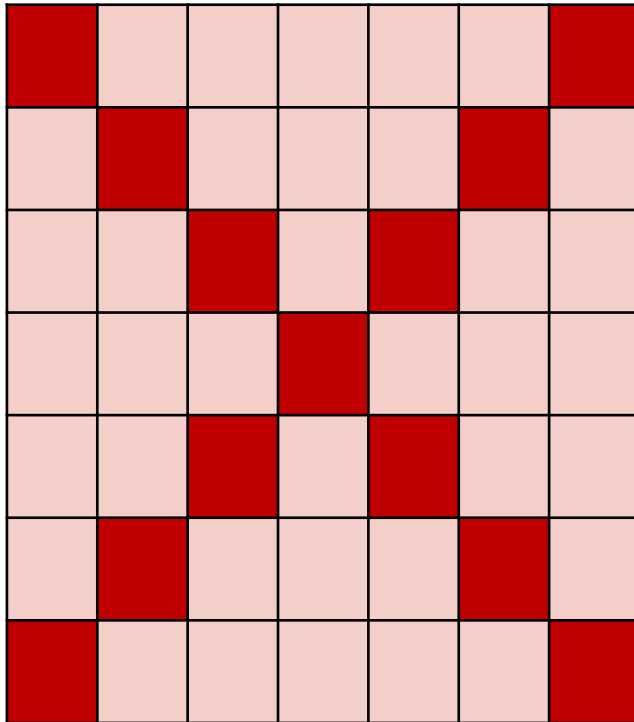


## Esercizio



SAPIENZA  
UNIVERSITÀ DI ROMA

# Somma della diagonale (v. più efficiente)

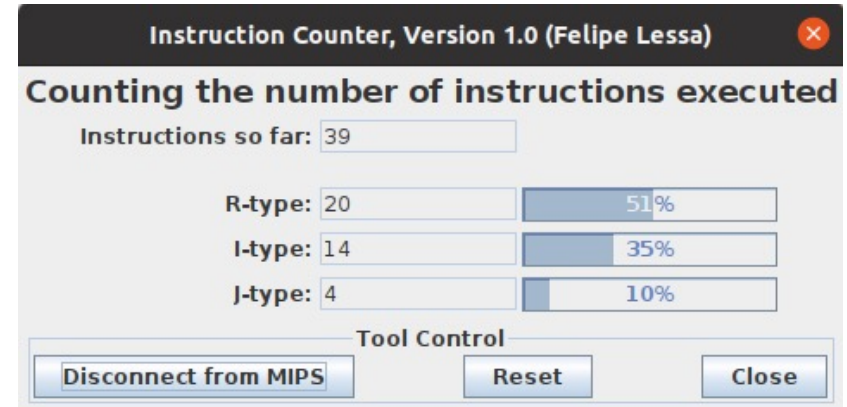
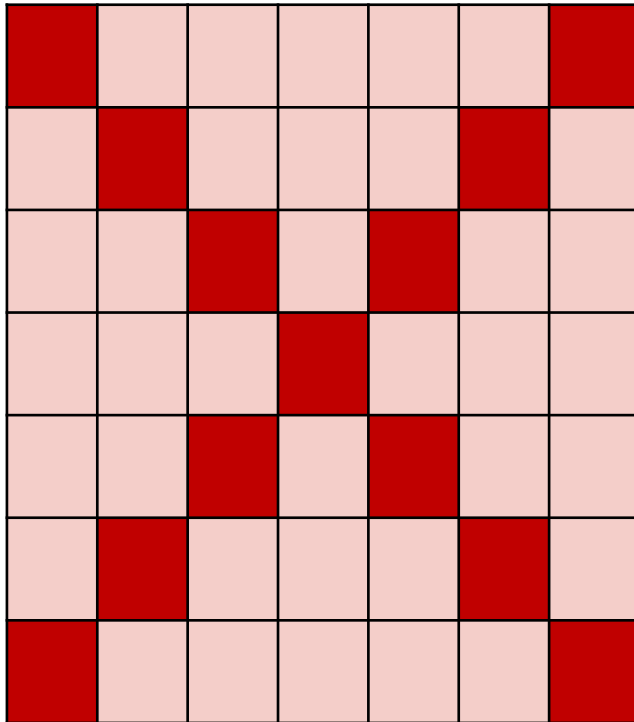


## Esercizio 1:

Scrivere un programma in assembly che, data una matrice bidimensionale quadrata **MATRIX** di word in memoria, di lato **N**, restituisca in **\$s0** il risultato della somma dei valori sulle diagonali principale e secondaria.



# Somma della diagonale (v. più efficiente) / reprise



## Esercizio 2:

Aggiungere istruzioni di stampa e di uscita al programma di cui all'esercizio 1.



---

## Circa gli spazi di indirizzamento e i salti



SAPIENZA  
UNIVERSITÀ DI ROMA

## Indirizzamento MIPS immediato a 32bit

Anche se mantenere tutte le istruzioni a 32-bit semplifica l'hardware, ci sono dei casi in cui è utile passare una costante o un indirizzo a 32bit in un'istruzione.

Il MIPS può indirizzare  $2^{32}$  byte.

Nessuna istruzione offre 32 bit per puntare alle locazioni in memoria disponibili.

I-type	Op 6 bit	Rs 5bit	Rt/d 5bit	Immediate 16 bit
J-type	Op 6 bit	Address 26 bit		

Idea: «spezzare» il caricamento in più istruzioni.



## Load upper immediate (lui) + or immediate (ori)

Qual è il codice assembly MIPS usato per caricare la seguente costante a 32 bit nel registro \$s0?

0000 0000 1111 1111 0000 1001 0000 0000

1

lui \$t0, 255 # 255 è il decimale di 0xFF

La versione in linguaggio macchina di lui \$t0, 255 # \$t0 è il registro 8:

001111	00000	01000	0000 0000 1111 1111
--------	-------	-------	---------------------

Il contenuto del registro \$t0 dopo avere eseguito lui \$t0, 255:

0000 0000 1111 1111	0000 0000 0000 0000
---------------------	---------------------

\$t0
0000 0000 1111 1111 0000 0000 0000 0000

2

ori \$s0, \$s0, 2304 # 2304 in decimale =  
# 0000 1001 0000 0000 in binario

\$t0
0000 0000 1111 1111 0000 1001 0000 0000

## Load upper immediate (lui) + or immediate (ori)

`$s0 = 0000 0000 1111 1111 0000 0000 0000 0000`

`ori $s0, $s0, 2304 # 2304 in decimale è 0000 1001 0000 0000`



# Chiarimenti su indirizzamento nei salti

J

Address	Code	Basic
0x00400000	0x24080000	addiu \$8,\$0,0x00000000
0x00400004	0x08100001	j 0x00400004

Portare il PC a puntare alla locazione in memoria del **text segment** all'indirizzo 0x00400004

4194356	0x014b082a	slt \$1,\$10,\$11	33:	bge \$t2,\$t3,end_col
4194360	0x10200003	beq \$1,\$0,3		
4194364	0x0c10002b	jal 4194476	34:	jal check_white
4194368	0x214a0001	addi \$10,\$10,1	35:	add \$t2,\$t2,1
4194372	0x0810000d	j 4194356	36:	j for_cols

Indirizzo è assoluto

spiazzamento rispetto al PC  
(in word, perché?)

bne \$s0,\$s1,Esci # vai a Esci se \$s0 è diverso da \$s1

4194324	0x1446002e	bne \$2,\$6,46	21:	bne \$v0,\$a2,exit
4194328	0x0c100026	jal 4194456	24:	jal close_file

per salto condizionato l'Indirizzo **non è assoluto**, ma uno scostamento in word relativo al PC

---

## Funzioni, invocazioni e activation record



SAPIENZA  
UNIVERSITÀ DI ROMA

## Funzioni: ingredienti / 2 – Istruzioni necessarie

Per chiamare la funzione/procedura

**jal** *etichetta*          Jump And Link

(simile ad un salto a subroutine ma più limitato)

registra nel registro **\$ra** la posizione dell'istruzione successiva      (**\$ra** ← PC+4)

cambia il PC per iniziare l'esecuzione del corpo della funzione      (PC ← *etichetta*)

Per tornare e continuare l'esecuzione del programma chiamante

**jr** **\$ra**                  Jump to Register

salta all'indirizzo contenuto nel registro indicato      (PC ← **\$ra**)

Come passare valori ALLA funzione (caso semplice)

**\$a0, \$a1, \$a2, \$a3**          4 registri per passare fino a 4 valori a 32 bit o 2 a 64 bit

e restituirli DALLA funzione

**\$v0, \$v1**                  2 registri per restituire fino a 2 valori a 32 bit (o 1 a 64 bit)

In quali problemi incorriamo usando solo **jal** e **jr \$ra** se quando si vogliono implementare funzionalità complesse?

# Riepilogo struttura della memoria

---

Stack pointer viene decrementato, memoria dinamica cresce a partire dalla fine di .data

# Preservare il contenuto dei registri

---

Conviene **preservare** il precedente contenuto dei registri usati dalla funzione e **ripristinarlo**

- **meno vincoli** alla funzione chiamante
- nelle funzioni che chiamano altre funzioni, che perderebbero il contenuto almeno di **\$ra**
- **\$a0, \$a1, \$a2, \$a3** possiamo solo passare 4 argomenti

Le informazioni da preservare hanno un **ciclo di vita caratteristico**, dovuto al **nidificarsi delle chiamate** delle funzioni:

- salvo stato prima di chiamata 1
  - salvo stato prima di chiamata 2
    - ...
  - ripristino stato prima di chiamata 2
- ripristino stato prima di chiamata 1

Questo è il comportamento di una pila (**stack** o LIFO), in cui aggiungere un elemento (**push**) e togliere l'ultimo inserito (**pop**)

Viene realizzata con un vettore di cui si tiene l'**indirizzo dell'ultimo elemento occupato** nel registro **\$sp** (Stack Pointer)

## Esempio chiamata funzioni nidificate

---

- main chiama foo che chiama bar
- foo ha bisogno di 3 registri \$s0, \$s1, \$s2
- bar ha bisogno di 2 registri \$s0, \$s1
- return address? (se ho già nidificato lo perdo)

# Preservare il contenuto dei registri

Conviene **preservare** il precedente contenuto dei registri usati dalla funzione e **ripristinarlo**

- **meno vincoli** alla funzione chiamante
- nelle funzioni che chiamano altre funzioni, che perderebbero il contenuto almeno di **\$ra**
- **\$a0, \$a1, \$a2, \$a3** possiamo solo passare 4 argomenti

Le informazioni da preservare hanno un **ciclo di vita caratteristico**, dovuto al **nidificarsi delle chiamate** delle funzioni: **\$sp** **980**

- vengono salvate le prime
  - ne vengono salvate altre
  - ...
- vengono prima ripristinate le seconde
- e poi vengono ripristinate le prime

Questo è il comportamento di una pila (**stack** o LIFO), in cui aggiungere un elemento (**push**) e togliere l'ultimo inserito (**pop**)

Viene realizzata con un vettore di cui si tiene l'**indirizzo dell'ultimo elemento occupato** nel registro **\$sp** (Stack Pointer)

Address	MEM
1000	xxxx
996	yyyy
992	zzzz
988	tttt
984	ssss
980	rrrr
976	
972	
968	
964	
960	
956	
952	
948	
944	
940	
936	
932	
928	
924	

# Uso dello stack (implementare push e pop)

Lo stack si trova nella parte «alta» della memoria e cresce **verso il basso**

Supponiamo di voler salvare e ripristinare il registro **\$ra**

Come salvare un elemento (push):

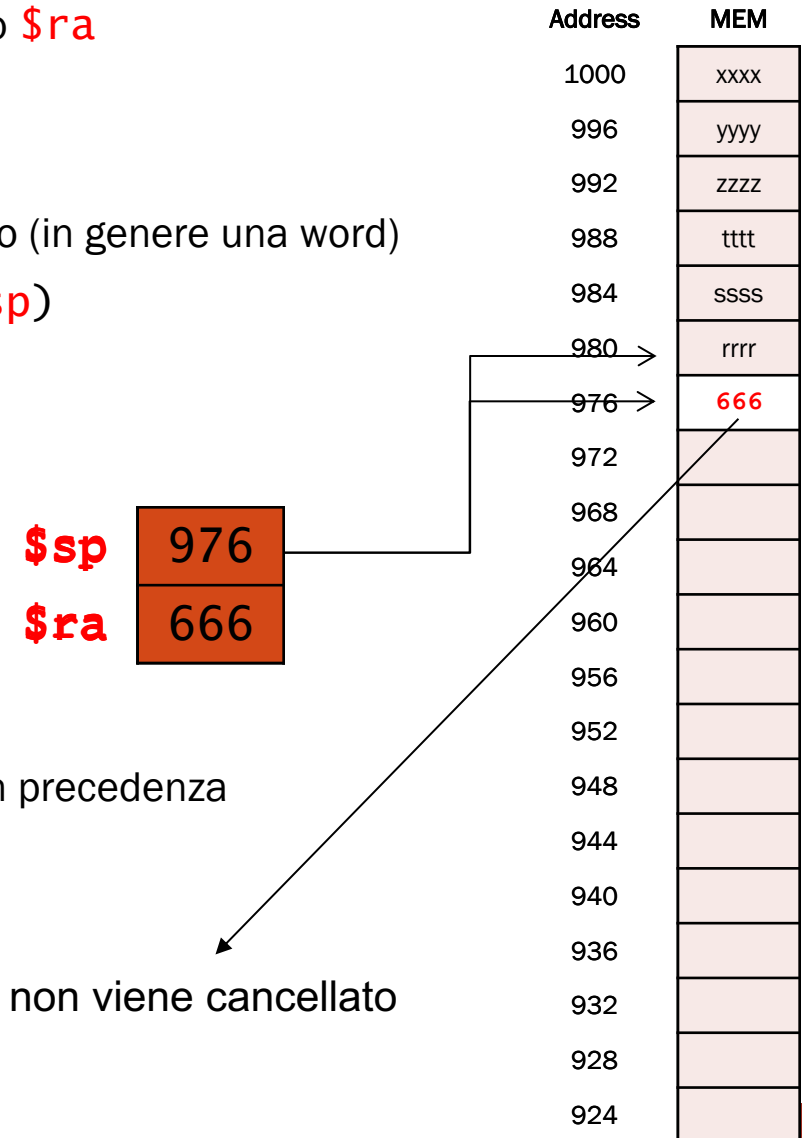
- si decrementa lo **\$sp** della dim. dell'elemento (in genere una word)
- si memorizza l'elemento nella posizione **0(\$sp)**

```
subi    $sp, $sp, 4
sw      $ra, 0($sp)
```

Come recuperare un elemento (pop):

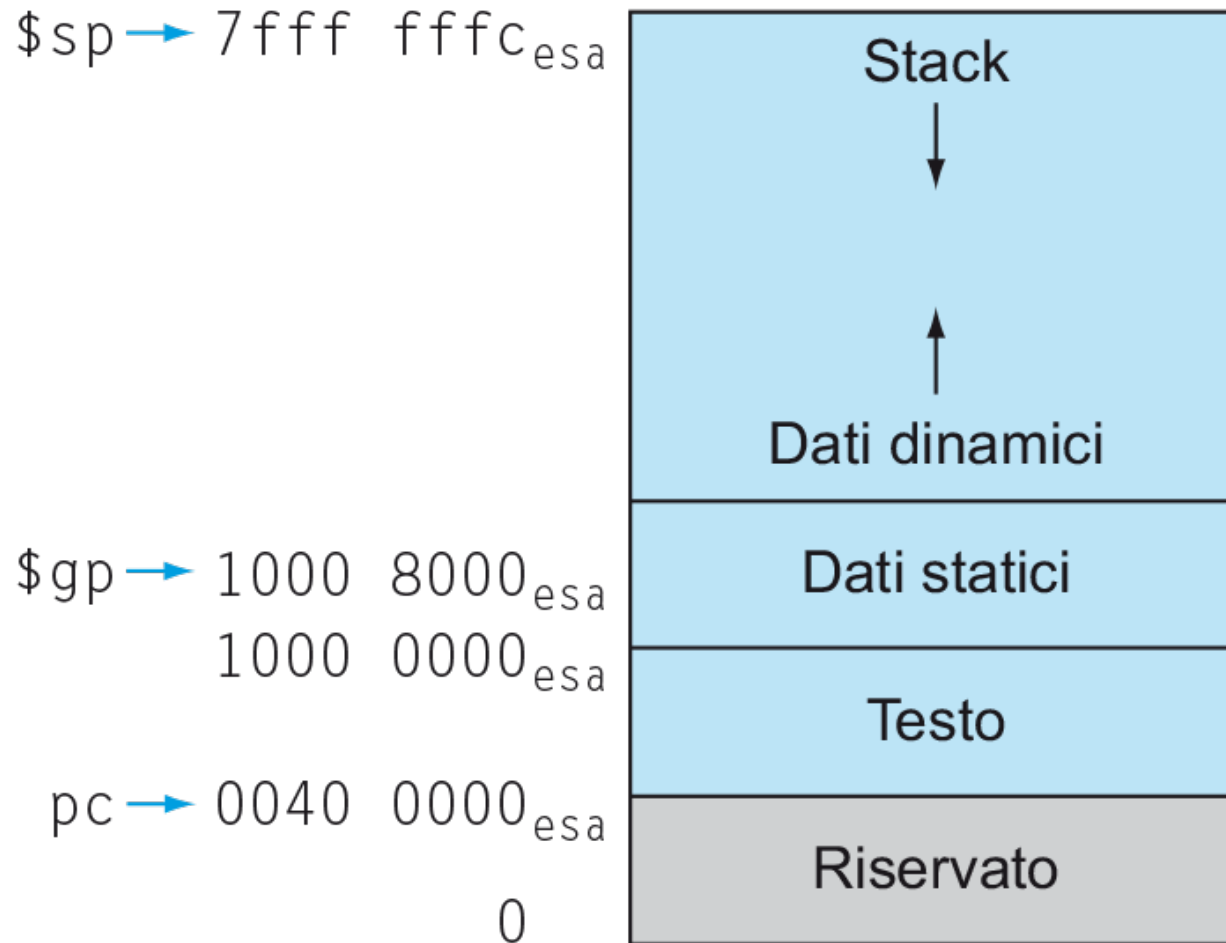
- si legge l'elemento dalla posizione **0(\$sp)**
- si incrementa lo **\$sp** della quantità allocata in precedenza

```
lw      $ra, 0($sp)
addi    $sp, $sp, 4
```





# La memoria



# Uso dello stack in una funzione

---

*funzione:*

All'inizio della funzione:

- **allocare** su stack abbastanza word da contenere i registri da preservare

- **salvare** su stack i registri, ad offset multipli di 4 rispetto a **\$sp**

**NOTA:** conviene allocare tutto lo spazio assieme per avere **offset che restano costanti** durante tutta l'esecuzione della funzione

All'uscita della funzione:

- **ripristinare** da stack i registri salvati, agli stessi offset usati precedentemente

- **disallocare** da stack lo stesso spazio allocato in precedenza

- **tornare** alla funzione chiamante

**addi**     **\$sp, \$sp, -12**

**sw**        **\$ra, 8(\$sp)**

**sw**        **\$a0, 4(\$sp)**

**sw**        **\$a1, 0(\$sp)**

**# corpo della funzione**

**# posso chiamare jal**

**# tranquillamente perché**

**# salvo \$ra**

**lw**        **\$a1, 0(\$sp)**

**lw**        **\$a0, 4(\$sp)**

**lw**        **\$ra, 8(\$sp)**

**addi**     **\$sp, \$sp, 12**

**jr**        **\$ra**

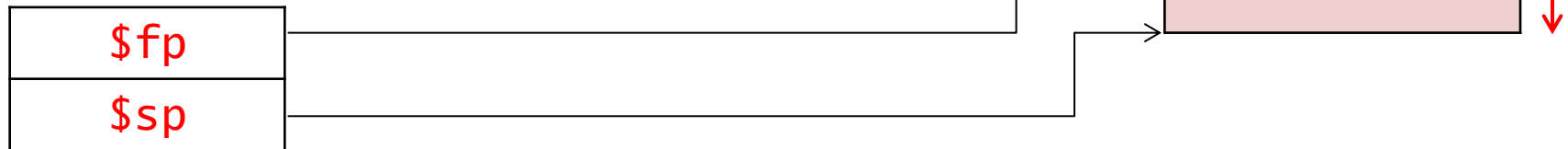
# Record di attivazione (stack frame / activation record)

Lo stack è usata anche per:

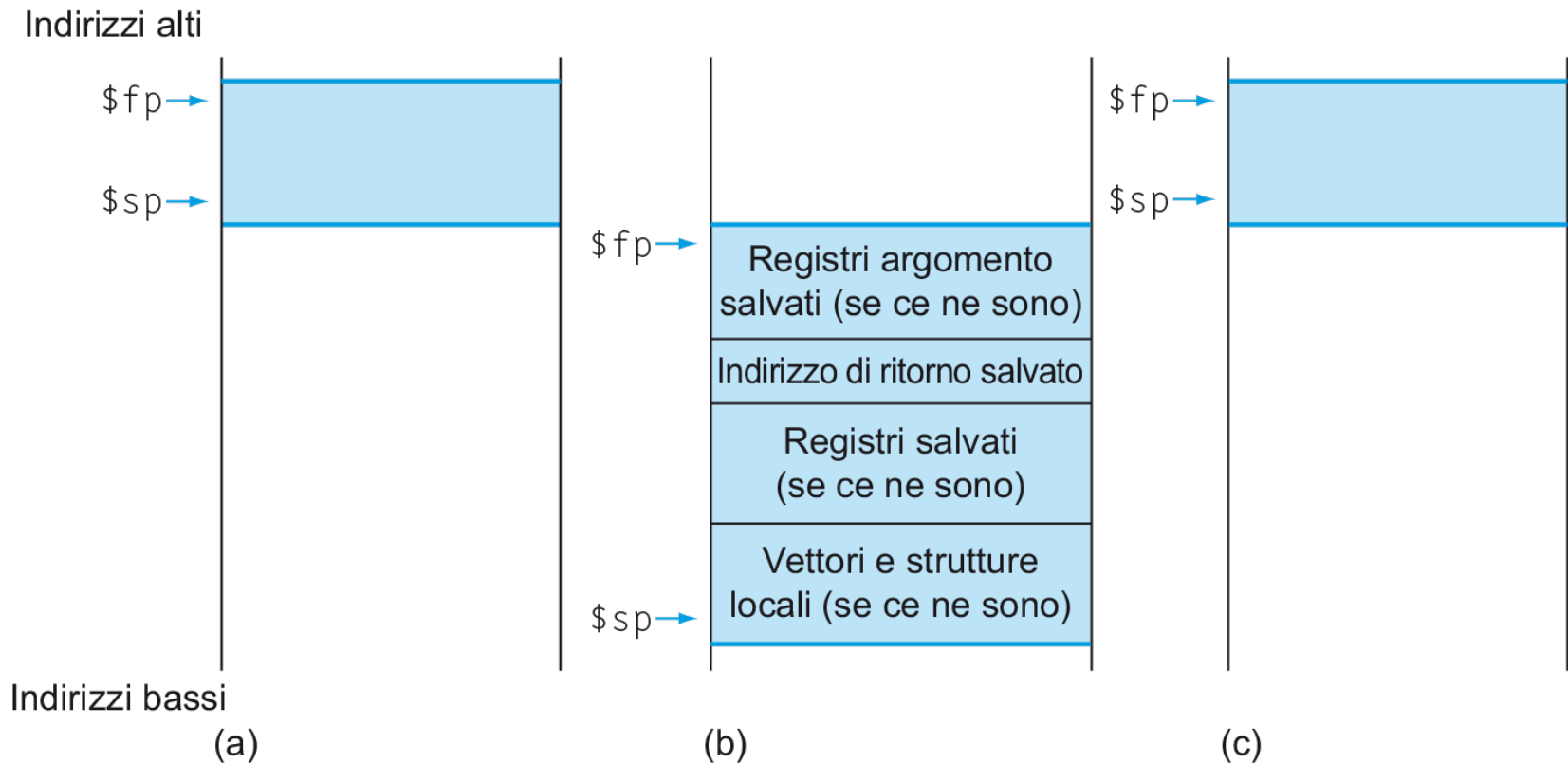
- Comunicare **ulteriori argomenti** oltre \$a0, ..., \$a3
- Comunicare **ulteriori risultati** oltre \$v0, \$v1
- Allocare **variabili locali** alla procedura

Questo blocco prende il nome di **stack frame** o **activation record**)

- Allocated su stack *prima* della chiamata della funzione e rilasciato subito dopo
- **\$sp (stack pointer)** si usa per puntare alla fine del record di attivazione
  - Varia allocando dati dinamicamente
- **\$fp (frame pointer)** si usa per puntare all'inizio del record di attivazione (ridondante ma comodo, non molto usato)
  - Resta fisso durante l'esecuzione della funzione

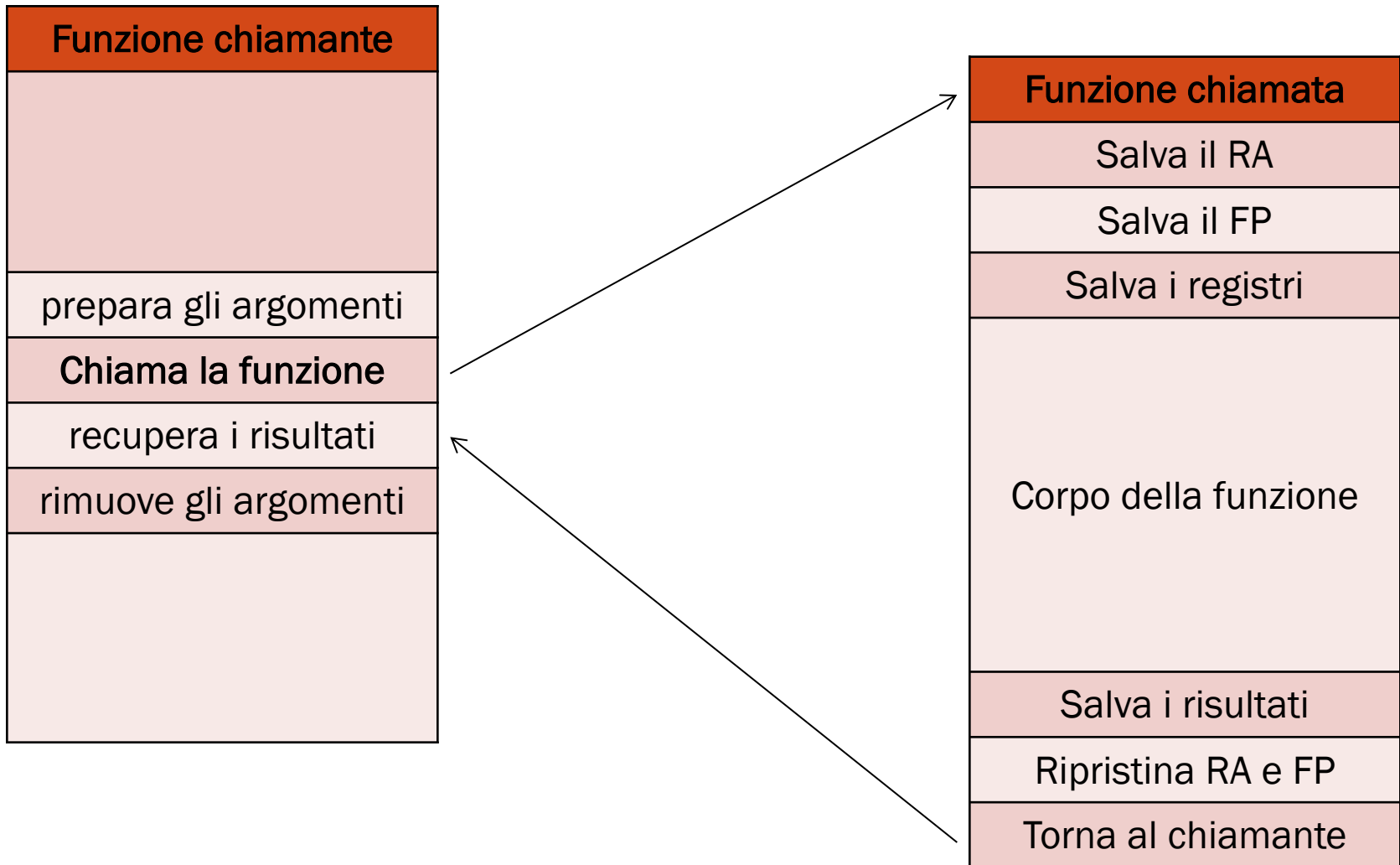


# Frame pointer e stack pointer




# Schema di una chiamata

---




## Esempio: calcolare $f$ in MIPS assembly

avgOfSquareAbsSub


$$f(x, y, w, z) = \frac{(|x| - |y|)^2 + (|w| - |z|)^2}{2}$$

squareAbsSub


$$g(x, y) = (|x| - |y|)^2$$

$$f(x, y, w, z) = \frac{g(x, y) + g(w, z)}{2}$$

# Esempio: calcolare $f$ in MIPS assembly

```
26
27 # function avgOfSquareAbsSub($a0: int, $a1: int, $a2: int, $a3: int): ($v0: int)
28 #   returns ( ( |$a0| - |$a1| ) ^ 2 + ( |$a2| - |$a3| ) ^ 2 ) / 2
29 avgOfSquareAbsSub:
30     subi $sp,$sp,20
31     sw $ra,0($sp)
32     sw $fp,4($sp)
33     sw $a0,8($sp)
34     sw $a1,12($sp)
35     sw $s0,16($sp)
```

perché non salvo \$a2 e \$a3?    non li uso (li leggo ma non li scrivo)  
perché salvo \$ra?            andrei in loop a 45

```
36
37     jal squareAbsSub
38
39     add $s0,$zero,$v0
40
41     add $a0,$zero,$a2
42     add $a1,$zero,$a3
43
44     jal squareAbsSub
45
46     add $s0,$s0,$v0
47     srl $s0,$s0,1
48
49     add $v0,$zero,$s0
50
51     lw $s0,16($sp)
52     lw $a1,12($sp)
53     lw $a0,8($sp)
54     lw $fp,4($sp)
55     lw $ra,0($sp)
56     addi $sp,$sp,20
57
58     jr $ra
```

non salvo \$ra?  
foglia!

```
59
60 # function squareAbsSub($a0: int, $a1: int): ($v0: int)
61 #   returns ( |$a0| - |$a1| ) ^ 2
62 squareAbsSub:
63
64     stack (lo
65           scrivo dopo)
66
67     abs $a0,$a0
68     abs $a1,$a1
69
70     sub $v0,$a0,$a1
71     mul $v0,$v0,$v0
72
73     lw $a1,4($sp)
74     lw $a0,0($sp)
75     addi $sp,$sp,8
76
77     jr $ra
```

# Esempio: calcolare $f$ in MIPS assembly

```
1  .globl main
2
3  .data
4  A:    .word 5
5  B:    .word 7
6  C:    .word 12
7  D:    .word -8
8
9  .text
10 main:
11     lw $a0,A           # Load parameters
12     lw $a1,B
13     lw $a2,C
14     lw $a3,D
15
16     jal avgOfSquareAbsSub # Call avgOfSquareAbsSub
17
18     add $s0,$v0,$zero # Save result in $s0 ($v0 is used later)
19
20     li $v0,1           # System call function selector: print int
21     add $a0,$s0,$zero # System call parameter passage
22     syscall
23
24     li $v0,10          # Exit
25     syscall
26
```

in questo caso buona parte dello stack si poteva omettere,  
ma è buona abitudine salvare tutto ciò che viene scritto  
per rendere il codice riutilizzabile

```
26
27 # function avgOfSquareAbsSub($a0: int, $a1: int, $a2: int, $a3: int): ($v0: int)
28 # returns ( ( |$a0| - |$a1| ) ^ 2 + ( |$a2| - |$a3| ) ^ 2 ) / 2
29 avgOfSquareAbsSub:
30     subi $sp,$sp,20
31     sw $ra,0($sp)
32     sw $fp,4($sp)
33     sw $a0,8($sp)
34     sw $a1,12($sp)
35     sw $s0,16($sp)
36
37     jal squareAbsSub
38
39     add $s0,$zero,$v0
40
41     add $a0,$zero,$a2
42     add $a1,$zero,$a3
43
44     jal squareAbsSub
45
46     add $s0,$s0,$v0
47     srl $s0,$s0,1
48
49     add $v0,$zero,$s0
50
51     lw $s0,16($sp)
52     lw $a1,12($sp)
53     lw $a0,8($sp)
54     lw $fp,4($sp)
55     lw $ra,0($sp)
56     addi $sp,$sp,20
57
58     jr $ra
59
60 # function squareAbsSub($a0: int, $a1: int): ($v0: int)
61 # returns ( |$a0| - |$a1| ) ^ 2
62 squareAbsSub:
63     subi $sp,$sp,8
64     sw $a0,0($sp)
65     sw $a1,4($sp)
66
67     abs $a0,$a0
68     abs $a1,$a1
69
70     sub $v0,$a0,$a1
71     mul $v0,$v0,$v0
72
73     lw $a1,4($sp)
74     lw $a0,0($sp)
75     addi $sp,$sp,8
76
77     jr $ra
```

Esempio in MARS



## Registri: ricapitoliamo

---

Nome	Numero del registro	Utilizzo	Da conservare nella chiamata?
\$zero	0	Costante 0	n.a.
\$v0-\$v1	2–3	Risultati e valutazione di espressioni	no
\$a0-\$a3	4–7	Argomenti	no
\$t0-\$t7	8–15	Variabili temporanee	no
\$s0-\$s7	16–23	Variabili da preservare	sì
\$t8-\$t9	24–25	Altri registri temporanei	no
\$gp	28	Global pointer	sì
\$sp	29	Stack pointer	sì
\$fp	30	Frame pointer	sì
\$ra	31	Indirizzo di ritorno	sì