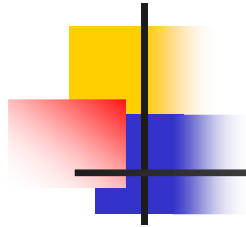


**Università degli Studi di Roma  
"La Sapienza"**

---

# **Architettura degli elaboratori II**

## **Funzioni**

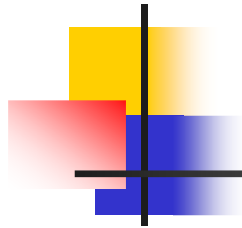


# Indice degli argomenti

---

## ■ Programmazione assembly

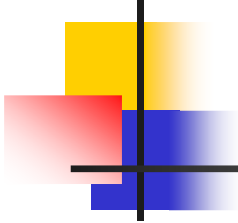
- Definizione e controllo di sottoprogrammi
- Definizione di funzione
- Chiamate a funzioni
- Convenzioni riguardanti l'uso dei registri
- Funzioni ricorsive
  - Lo Stack



# Funzioni

---

- Una funzione (o routine) è una sequenza di istruzioni che esegue un compito definito e che viene usata *come se fosse una unità di codice*.
  - Una funzione è una astrazione presente in (quasi) tutti i linguaggi ad alto livello.
  - Assegna un *nome* ad una operazione complessa (fatta da più istruzioni)
  - Maggiore leggibilità del codice.
  - Riutilizzo del codice
  - L'intero programma assembly viene visto come un insieme di funzioni
- *Non è supportata dall'assembly; quest'ultimo fornisce solo le istruzioni e i registri che consentono di realizzare delle funzioni. E un insieme di convenzioni.*



# Chiamata a funzione e ritorno

---

- **Chiamata a funzione:**

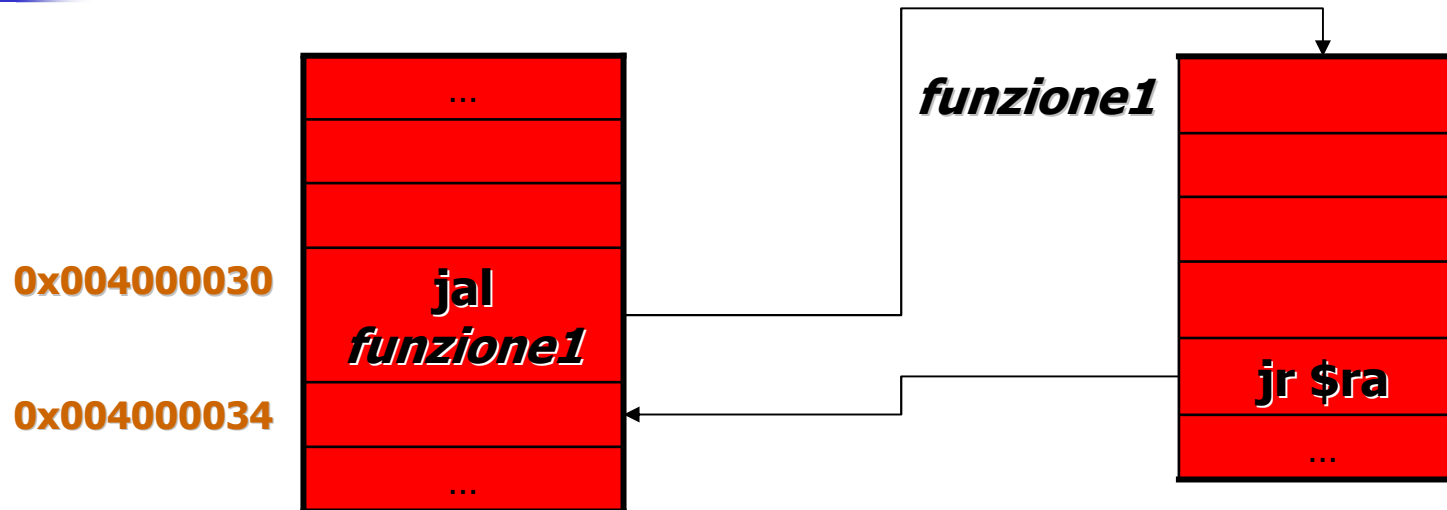
- Per chiamare una funzione si usa l'istruzione  
***jal function-name*** (***jal = jump and link***)
- Memorizza l'indirizzo dell'istruzione che *segue* il **jal** nel registro ***\$ra*** e salta all'etichetta ***function-name***

- **Ritorno da funzione:**

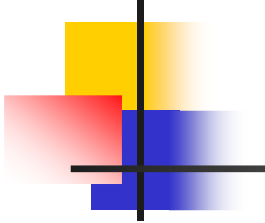
- Per tornare da una funzione si usa l'istruzione  
***jr \$ra*** (***jr = jump register***)
- Salta all'indirizzo contenuto nel registro ***\$ra***
- *All'occorrenza si può anche usare un altro registro*

# Chiamata a funzione e ritorno

## -ESECUZIONE -



1. Vengono eseguite le istruzioni della funzione chiamante **f** che **precedono** la chiamata (**jal**) a *funzione1*
2. **Chiamata:** l'istruzione **jal** salva l'indirizzo della prossima istruzione (da eseguire al ritorno) in **\$ra** e salta all'inizio della funzione chiamata
3. Vengono eseguite tutte le istruzioni di **funzione1** fino alla **jr** conclusiva
4. **Ritorno:** l'istruzione **jr** salta alla prima istruzione che segue la chiamata
5. Vengono eseguite le rimanenti istruzioni della funzione chiamante



# Esempio Funzione

|           |              |                              |
|-----------|--------------|------------------------------|
|           | <b>main:</b> | <code>la \$a0, str</code>    |
| Funzione  |              | <code>jal length</code>      |
| Chiamante |              | <code>sw \$v0, result</code> |

*Argomenti in \$a0-\$a3*

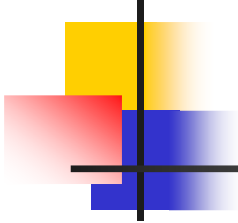
*Chiamata a funzione*

*Risultati in \$v0-\$v1*

|          |                  |                                 |
|----------|------------------|---------------------------------|
|          | <b>length:</b>   | <code>move \$t0, \$a0</code>    |
|          |                  | <code>li \$t1, 0</code>         |
| Funzione | <b>nextChar:</b> | <code>lbu \$t2, (\$t0)</code>   |
| Chiamata |                  | <code>beqz \$t2, end</code>     |
|          |                  | <code>addu \$t0, \$t0, 1</code> |
|          |                  | <code>addu \$t1, \$t1, 1</code> |
|          |                  | <code>b nextChar</code>         |
|          | <b>end:</b>      | <code>move \$v0, \$t1</code>    |
|          |                  | <code>jr \$ra</code>            |

*Parametro: \$a0*

*Valore di ritorno: \$v0*



# Funzione

---

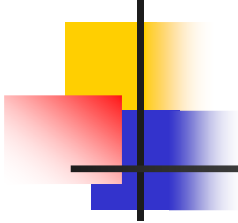
- Una funzione può essere:
- **Foglia:** Se non chiama altre funzioni al suo interno
- **Non-foglia:** Se è sia chiamante che chiamata

```
main:      lw $a0,x  
  
           lw $a1,y  
  
           jal adder  
  
           sw $v0,z
```

FUNZIONE NON FOGLIA

```
adder:  
  
           add $v0,$a0,$a1  
  
           jr $ra
```

FUNZIONE FOGLIA

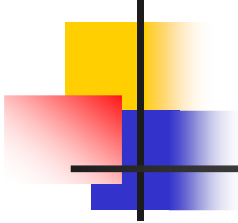


# **Chiamata a funzione e ritorno**

## **-PROBLEMI GENERALI -**

---

1. Il registro \$ra non è sufficiente se la funzione chiamata è a sua volta un chiamante (non foglia)
2. Desidero passare vari parametri alla funzione chiamata e ottenere un risultato
3. Mi fa comodo definire una convenzione per stabilire quali registri vengono preservati dopo una chiamata e quali no
4. La funzione può avere bisogno di variabili locali se non bastano i registri stabiliti.



# Convenzioni: REGISTRI

---

**Registri usati per passare parametri e risultati:**

**\$a0-\$a3** argomenti di una funzione

**\$v0-\$v1** risultato di una funzione (32/64 bit)

*E se ho bisogno di più byte?*

**Registri preservati tra le chiamate**

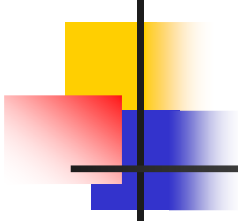
**\$t0-\$t9** e **\$ra** non vengono preservati tra le chiamate

*E se ho bisogno di preservarli?*

**\$s0-\$s7** vengono preservati tra le chiamate

*Come si fanno a preservare se li uso?*

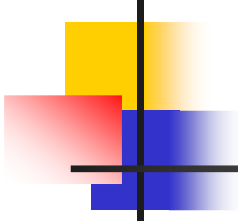
*E se non mi bastano i registri che ci sono per scrivere una funzione particolarmente complessa?*



# ACTIVATION FRAME

---

- **Ogni funzione necessita di uno spazio di memoria per**
  - Memorizzare i valori passati alla funzione come argomenti
  - Salvare i registri che una funzione può modificare, ma che il chiamante si aspetta che vengano preservati
  - Fornire spazio per le variabili locali della procedura (se non ho abbastanza registri da utilizzare)
- ***Questo spazio viene chiamato  
frame di attivazione della funzione***



# Frame di attivazione statico

```
f:  sw $ra,fFrame
    li $a0,5
    li $t0,12
    move $a0,$t0
    jal p
    lw $ra,fFrame
    jr $ra
```

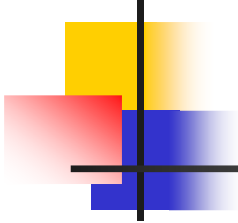
```
p:  sw $ra,pFrame
    addi $s0,$a0,1
    sw $s0,pFrame+4
    jal q
    lw $s0,pFrame+4
    addi $v0,$s0,3
    lw $ra,pFrame
    jr $ra
```

```
q:  li $s0,1
    move $s1,$s0
    lw $s0,qFrame
    lw $s1,qFrame+4
    jr $ra
```

```
.data
```

```
fFrame: .space 4 # spazio per $ra
```

```
pFrame: .space 8 # spazio per $ra e $s0
```



# Frame di attivazione e funz. ricorsive

```
fact:    sw $ra, factFrame
         ble $a0, 1, base
         sw $a0, factFrame+4    # salva n
         subu $a0, $a0, 1
         jal fact               # fact(n-1)
         lw $t0, factFrame+4    # carica n
         mul $v0, $t0, $v0      # n*(n-1)!
         j end

base:    li $v0, 1              # caso base

end:     lw $ra, factFrame
         jr $ra
```

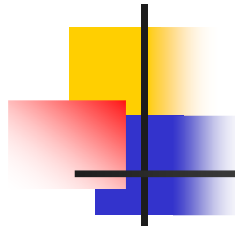
```
.data
```

```
factFrame: .space 8    # spazio per $ra e n
```

- ❖ Un singolo frame per funzione *non è sufficiente* nel caso di funzioni *ricorsive*
- ❖ Da qui il nome: “*frame di attivazione*” perché è necessario un frame per ogni *attivazione* (chiamata) di funzione

*Come faccio ad allocare del nuovo spazio in memoria ogni volta che chiamo una funzione?*

**uso lo STACK**



# STACK

---

- **Last-in, first-out (LIFO)**

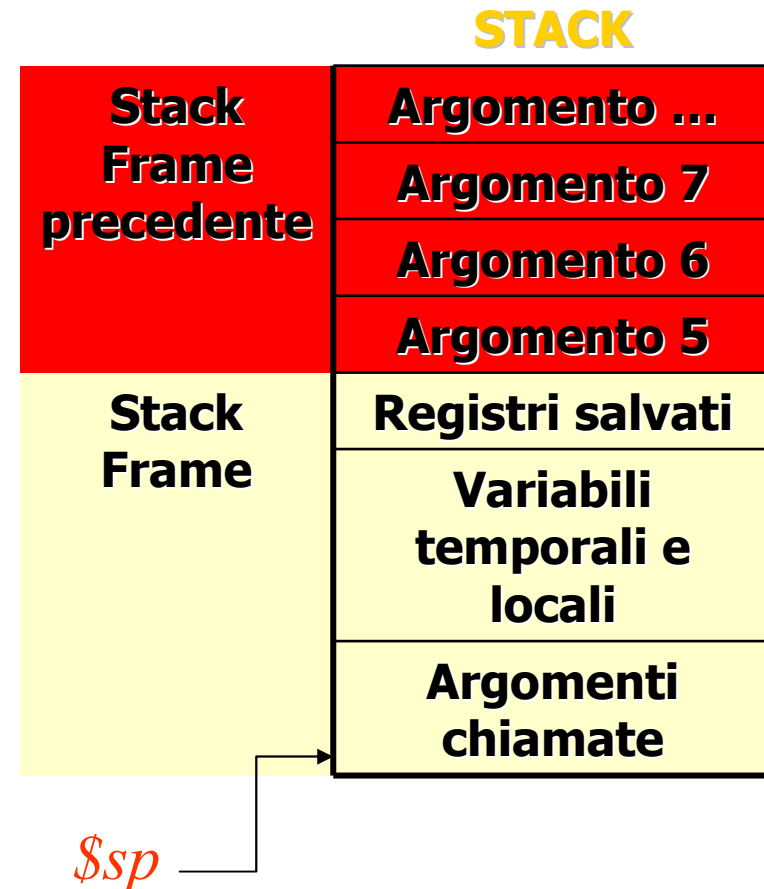
- I frame di attivazione di funzione vengono creati e distrutti seguendo un modello last-in, first-out
- È quindi possibile memorizzarli in uno stack

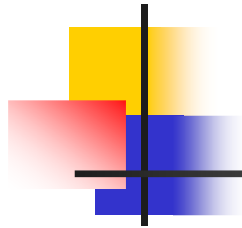
- **Operazioni sullo stack:**

- ***push***: inserisce un frame nello stack, inserendolo in ultima posizione
- ***pop***: rimuove l'ultimo frame dallo stack
- ***read/write***: nel frame corrente (ultimo elemento) dello stack

# STACK FRAME

- **Uno stack frame contiene:**
  - Registri salvati
  - Variabili locali e temporanee
  - Argomenti per le funzioni chiamate dalla funzione (massimo numero richiesto; si conta dal quinto)
- **È accessibile da:**
  - Stack pointer *\$sp*
  - Lo stack cresce verso gli indirizzi più bassi della memoria (Per procedere devo sottrarre)
- ***NB La dimensione del frame deve essere un multiplo di 8 byte***

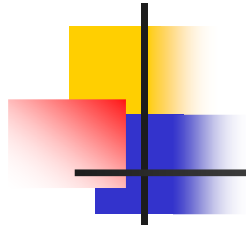




## Convenzioni: Funzioni non foglia

- Alloco nello stack uno spazio sufficiente a contenere tutti i registri che devo salvare, le variabili locali e il *numero max* di argomenti.
- Salvo i registri `$s0-$s7` che intendo usare e `$ra`
  - Se la funzione chiamata richiede più di 4 argomenti li metto nello stack.
  - Se voglio preservare i registri `$t0-$t9, $a0-$a3, $v0-$v1` devo salvarli prima della chiamata a funzione e ripristinarli dopo.
- Ripristino i registri salvati `$ra, $s0-$s7`.
- Libero lo spazio sullo stack allocato all'inizio

```
f: ...  
    subu $sp, $sp, 16  
    sw $s0, 12($sp)  
    sw $s1, 8($sp)  
    sw $ra, 4($sp)  
    # 3 var locali (word)  
    # 0 ulteriori argomenti  
    jal p  
    lw $ra, 4($sp)  
    lw $s1, 8($sp)  
    lw $s0, 12($sp)  
    addu $sp, $sp, 16  
    ...  
    jr $ra
```



# Fattoriale (Ricorsivo)

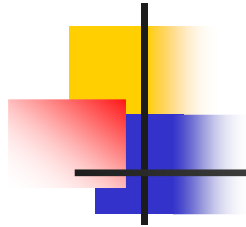
```
.data
n:      .word 7
result: .word 0

.text

main:
    lw $a0,n
    jal fact #fact(n)
    sw $v0,result

fact:  ble $a0,1,base
        subu $sp,$sp,8
        sw $ra,4($sp)
        sw $a0,0($sp) #salva n
        subu $a0,$a0,1
        jal fact      #fact(n-1)
        lw $t0,0($sp) #carica n
        mul $v0,$t0,$v0 #n*(n-1)!
        lw $ra,4($sp)
        addu $sp,$sp,8
        jr $ra

base:  li $v0,1      #caso base
        jr $ra
```



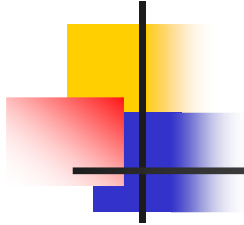
# Fattoriale (Iterativo)

- ❖ Non bisogna usare lo stack
- ❖ Il risultato è molto più efficiente sia in spazio che in tempo

*NB I compilatori dei linguaggi ad alto livello sono in genere abbastanza intelligenti da fare traduzioni iterative anche di funzioni ricorsive (quando è ragionevole).*

```
fact:  ble $a0,1,base    #$a0 n
        move $v0,$a0      #$v0 risultato
loop:  subu $a0,$a0,1
        mul $v0,$v0,$a0    #n* (n-1) * (n-2) ...
        bgt $a0,2,loop
        jr $ra
base:  li $v0,1           #caso base
        jr $ra
```

# Fattoriale (Precalcolato)



- ❖ Il risultato occupa più spazio ma calcola in tempo costante i risultati più richiesti

*N.B.: i compilatori dei linguaggi ad alto livello non fanno questo genere di ottimizzazioni perché non sanno se ne vale la pena. Le macchine virtuali sì.*

```
fact:  ble $a0,10,base      # $a0 = n
        move $v0,$a0        # $v0 = risultato
loop:  subu $a0,$a0,1
        mul $v0,$v0,$a0      # n*(n-1)*(n-2)...
        bgt $a0,2,loop
        jr $ra
base:  mul $a0,$a0,4          # caso precalcolato
        lw $v0,table($a0)
        jr $ra
```

.data

```
table: .word 1,1,2,6,24,120,720,5040,40320,362880,3628800
```