



Assembly MIPS: le procedure

Proff. A. Borghese, F. Pedersini

Dipartimento di Informatica
Università degli Studi di Milano

Chiamata a procedura: esempio



Procedura chiamante

```
f = f + 1;  
if (f == g)  
    res = mult(f, g)  
else f = f - 1;  
.....
```

Procedura chiamata

```
int mult (int p1, int p2) {  
    int out  
    out = p1 * p2;  
    return out;  
}
```

- ❖ Cosa succede?
 - Cambio di flusso
 - Passaggio informazioni
- ❖ *Come si fa in Assembly?*



Due **attori**: **chiamante** e **chiamata**

- ❖ La procedura **chiamante** (**caller**) deve:
 1. Predisporre i parametri di ingresso in un posto accessibile alla procedura chiamata
 2. Trasferire il controllo alla procedura chiamata

- ❖ La procedura **chiamata** (**callee**) deve:
 1. Allocare lo spazio di memoria necessario all'esecuzione ed alla memorizzazione dei dati: → **record di attivazione**
 2. Eseguire il compito richiesto
 3. Memorizzare il risultato in un luogo accessibile alla procedura chiamante
 4. Restituire il controllo alla procedura chiamante

Chiamata a procedura: **jal**



- ❖ Necessaria un'istruzione apposita che:
 1. **cambia il flusso** di esecuzione (salta alla procedura) e
 2. **salva l'indirizzo di ritorno**
(istruzione successiva alla chiamata a procedura)

jal (jump and link)

Sintassi: **jal Indirizzo_Procedura**

1. Salta all'indirizzo con etichetta **Indirizzo_Procedura**
 2. Memorizza il **valore corrente del Program Counter in \$ra**
(indirizzo dell'istruzione successiva: **\$(Program Counter) + 4**)
- **La procedura chiamata**, come **ultima istruzione**, esegue: **jr \$ra**
per effettuare il salto all'indirizzo di ritorno della procedura



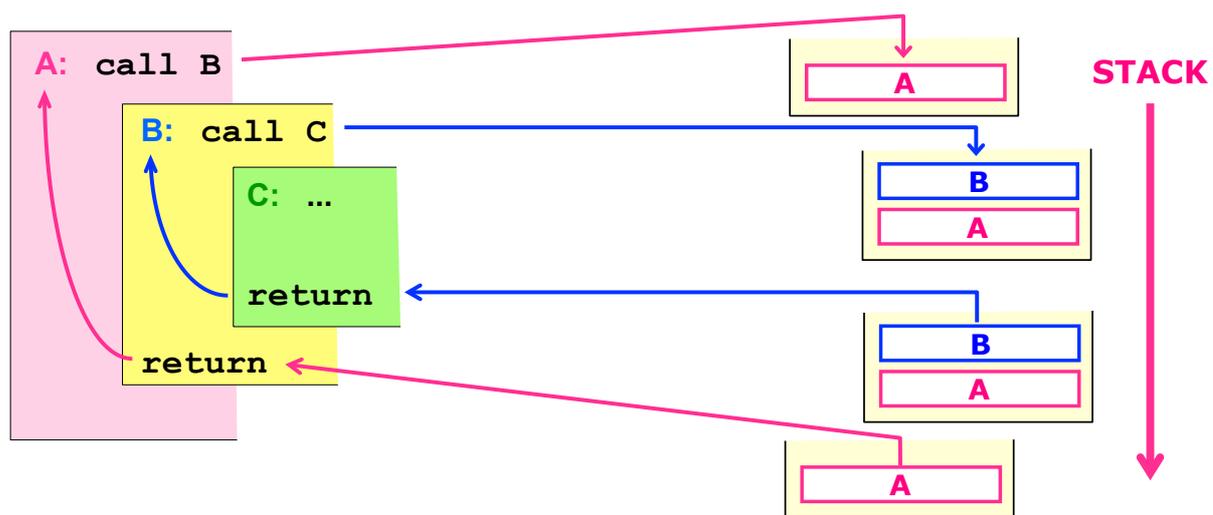
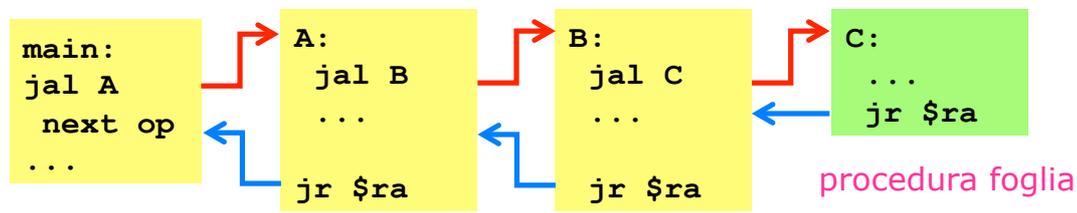
Procedura foglia: è una procedura che non ha annidate al suo interno chiamate ad altre procedure

Non è necessario salvare \$ra, perché nessun altro lo modifica

NON sono procedure foglia:

- Procedure annidate, Procedure recursive
- Procedure che chiamano altre sotto-procedure

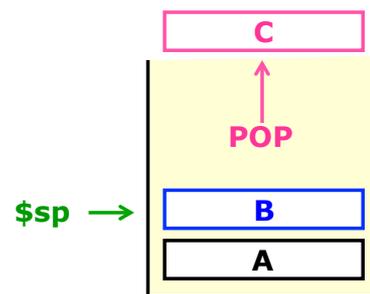
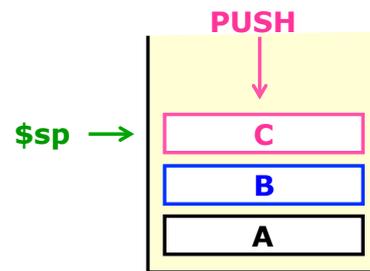
E' necessario salvare \$ra nello stack, perchè la procedura chiamata lo modifica a sua volta!



- ❖ **Gestione automatica** dello STACK:
 - Stack gestito automaticamente dall'architettura, mediante circuiti dedicati nella CPU (es. VAX)
- ❖ **MIPS:** gestione **manuale** dello STACK:
 - Implementata secondo **convenzioni software**



- ❖ Struttura dati: **coda LIFO** (*last-in-first-out*)
- ❖ Inserimento dati nello stack: **PUSH**
- ❖ Prelievo dati dallo stack: **POP**
- ❖ Necessario un puntatore alla cima dello stack per salvare i dati
 - Il registro **\$sp** (**stack pointer** – puntatore allo stack) contiene l'indirizzo della **cima dello stack** (Top of Stack, TOS)
 - **\$sp** deve essere aggiornato ogni volta che viene inserito o estratto un dato dallo stack



Gestione dello **stack** in MIPS



Stack MIPS:

- ❖ Lo stack cresce da indirizzi di memoria **alti verso indirizzi bassi**
- ❖ Il registro **\$sp** contiene l'indirizzo della prima locazione libera in **cima allo stack**
- ❖ Inserimento di un dato nello stack – **PUSH:**
 - Si **decrementa** **\$sp** per allocare lo spazio e
 - Si esegue **sw** per inserire il dato nello stack
- ❖ Prelevamento di un dato dallo stack – **POP:**
 - Si esegue **lw** per recuperare il dato dallo stack
 - Si **incrementa** **\$sp** (per eliminare il dato), riducendo la dimensione dello stack.



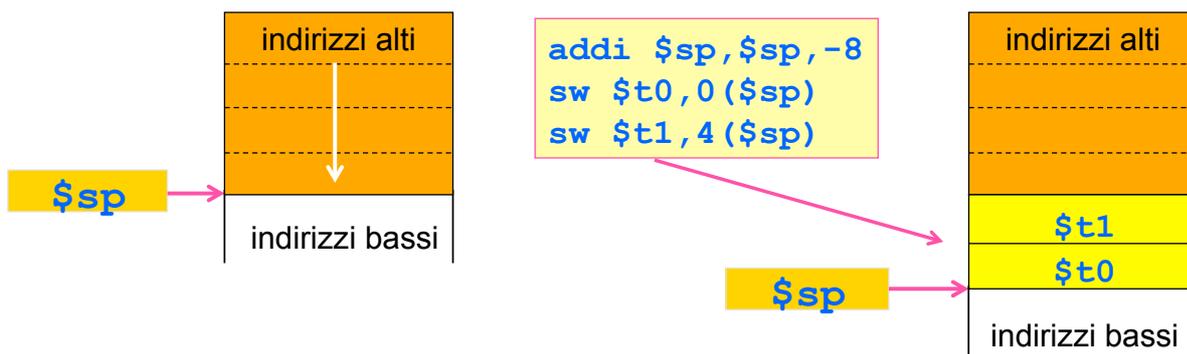
Gestione dello stack MIPS

- ❖ Per inserire elementi nello stack (**push**)

```
sw $t0, offset($sp) # Salvataggio di $t0
```

- ❖ Per recuperare elementi dallo stack (**pop**)

```
lw $t0, offset($sp) # Ripristino di $t0
```



Allocazione dei registri



- ❖ **Convenzioni per l'allocazione dei registri:**

- ❖ **\$ra**

return address

- registro per memorizzare l'indirizzo della **prima istruzione del chiamante**, da eseguire al termine della procedura

- ❖ **\$a0÷\$a3 (\$f12÷\$f15)**

registri argomento

- usati dal chiamante per il passaggio dei parametri
- Se i parametri sono più di 4, si usa la memoria (**stack**)

- ❖ **\$v0, \$v1 (\$f0÷\$f3)**

registri valore

- usati dalla procedura per memorizzare i valori di ritorno



❖ Il programma **chiamante**:

- Inserisce i parametri da passare alla procedura nei **registri argomento**: $\$a0-\$a3$
- Salta alla procedura mediante l'istruzione **jal address** e salvare il valore di **(PC+4)** nel registro $\$ra$

❖ La procedura **chiamata**:

- Esegue il compito richiesto
- Memorizza il risultato nei registri $\$v0, \$v1$
- Restituisce il controllo al chiamante con l'istruzione **jr \$ra**



*La procedura **chiamata** usa gli stessi registri usati dal chiamante.
Al ritorno, la **chiamante** trova registri modificati!!*

❖ **Record di attivazione (MIPS):**

tutto lo **spazio nello stack** di cui ha bisogno una procedura.

- In **Assembly MIPS** lo spazio per memorizzare il record di attivazione viene esplicitamente allocato dal programmatore
- ❖ *Quando si chiama una procedura **i registri utilizzati dal chiamato** vanno:*
 - **salvati nello stack**
 - **ripristinati alla fine** dell'esecuzione della procedura



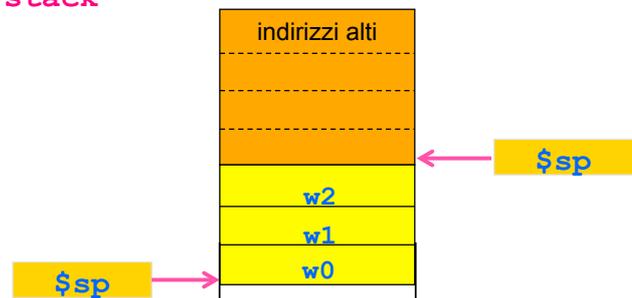
- ❖ **All'inizio della procedura**, il record di attivazione viene **allocato** decrementando \$sp (PUSH)

```
addi $sp, $sp, -12    # alloca 12 byte
                     # nello stack
```

- ❖ **Al termine della procedura**, il record di attivazione viene **rimosso** incrementando \$sp della stessa quantità **(POP)**

```
addi $sp, $sp, 12    # dealloca 12 byte
                     # dallo stack
```

- ❖ Tale spazio esiste soltanto **durante l'esecuzione** della procedura.



Esempio



C:

```
int somma_algebrica (int g, int h, int i, int j)
{
    int f;

    f = (g + h) - (i + j);
    return f;
}
```

Assembly:

```
# g,h,i,j associati a $a0 ... $a3, f associata ad $s0
# Il calcolo di f richiede 3 registri: $s0, $s1, $s2
# E' necessario salvare i 3 registri nello stack
```

Somma_algebrica:

```
addi $sp,$sp,-12    # alloca nello stack
                     # lo spazio per i 3 registri

sw $s0, 8($sp)      # salvataggio di $s0
sw $s1, 4($sp)      # salvataggio di $s1
sw $s2, 0($sp)      # salvataggio di $s2
```

Esempio (cont.)



```
add $s0, $a0, $a1      # $s0 ← g + h
add $s1, $a2, $a3      # $s1 ← i + j
sub $s2, $s0, $s1      # f ← $s0 - $s1

add $v0, $s2, $zero    # restituisce f copiandolo
                       # nel reg. di ritorno $v0

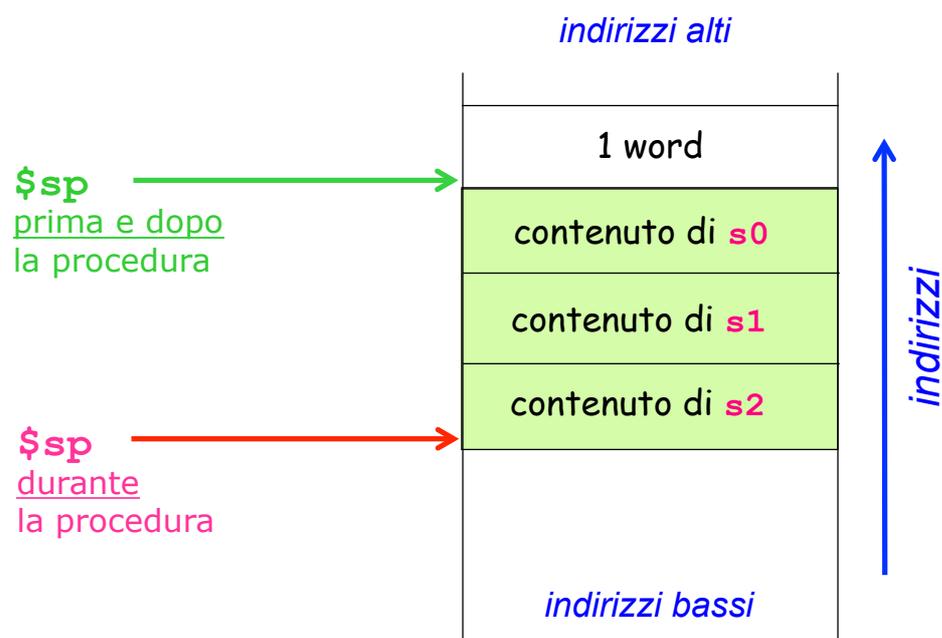
# ripristino del vecchio contenuto dei registri
# estraendoli dallo stack

lw $s2, 0($sp)         # ripristino di $s2
lw $s1, 4($sp)         # ripristino di $s1
lw $s0, 8($sp)         # ripristino di $s0

addiu $sp, $sp, 12    # deallocazione dello stack
                       # per eliminare 3 registri

jr $ra                 # ritorno al prog. chiamante
```

Esempio (cont.)





- ❖ Per evitare di salvare inutilmente i registri, sono divisi in due classi:

- **Registri temporanei:**

$\$t0, \dots, \$t9$ ($\$f4 \dots \$f11, \$f16 \dots \$f19$)

- il cui contenuto **non è salvato dal chiamato** nello stack (viene salvato dal chiamante se serve);

- **Registri non-temporanei:**

$\$s0, \dots, \$s8$ ($\$f20, \dots, \$f31$)

- il cui contenuto è **salvato dal chiamato** nello stack **se utilizzato**

Si può eliminare il salvataggio dei registri $\$s0$ e $\$s1$, utilizzando al loro posto i registri $\$t$, ed eliminare l'utilizzo del registro $\$s2$:

```
sub $s2, $s0, $s1 # f ← $t0 - $t1
```

può diventare:

```
sub $v0, $t0, $t1
```



- ❖ **Tutte le procedure** → il **chiamante salva nello stack**:

- I registri temporanei di cui vuole salvare il contenuto di cui ha bisogno dopo la chiamata ($\$t0-\$t9, \dots$)
- I registri argomento ($\$a0-\$a3, \dots$) nel caso in cui il loro contenuto debba essere preservato.

- ❖ **Procedure foglia** → il **chiamato salva nello stack**:

- I registri non temporanei che vuole utilizzare ($\$s0-\$s8$)
- Strutture dati locali (es: array, matrici) e variabili locali della procedura che non stanno nei registri.

- ❖ **Procedure non-foglia** → il **chiamato salva nello stack anche**:

- I registri argomento della procedura,
- $\$ra$ ($\$fp$)



- ❖ **Record di attivazione: spazio privato della procedura eseguita**
 - area di memoria dove vengono allocate le variabili locali della procedura e i parametri

- ❖ Il programmatore assembly deve provvedere **esplicitamente** ad allocare/cedere lo spazio necessario per:
 - Mantenere i valori passati come parametri alla procedura;
 - Salvare i registri che una procedura potrebbe modificare ma che al chiamante servono inalterati.
 - Fornire spazio per le variabili locali alla procedura.

- ❖ Quando sono permesse chiamate di procedura **annidate**, i **record di attivazione sono allocati e rimossi dallo stack**.

Struttura di una procedura



- ❖ Ogni procedura è composta **3 SEZIONI** consecutive:

- ❖ **Prologo**
 - Acquisire le risorse necessarie per memorizzare i dati interni alla procedura ed il salvataggio dei registri.
 - Salvataggio dei registri di interesse.

- ❖ **Corpo**
 - Esecuzione della procedura vera e propria

- ❖ **Epilogo**
 - Mettere il risultato in un luogo accessibile al programma chiamante.
 - Ripristino dei registri di interesse.
 - Liberare le risorse utilizzate dalla procedura
 - Restituzione del controllo alla procedura chiamante.



- ❖ Determinazione della dimensione del record di attivazione
 - Stimare lo spazio per: registri da salvare, argomenti, variabili locali.
- ❖ Allocazione dello spazio in stack: aggiornamento valore di `$sp`:

```
addi $sp, $sp, -dim_record_attivaz
# lo stack pointer viene decrementato
# della dimensione del record di attivazione
# prevista per la procedura
```
- ❖ Salvataggio dei registri per i quali è stato allocato spazio in stack:

```
sw reg, [dim_record_attivaz-N] ($sp)
```

 - ✦ `N` viene incrementato di 4 ad ogni salvataggio (`N ≥ 4`)

Esempio: record di attivazione: 3 registri → 12 byte

```
addi $sp, $sp, -12
sw $s0, 8($sp) # dim_record_attivazione - 4
sw $s1, 4($sp) # dim_record_attivazione - 8
sw $s2, 0($sp) # dim_record_attivazione - 12
```



- ❖ Restituzione dei parametri della procedura
 - dai registri `$v0, $v1` (e/o stack)
- ❖ Ripristino dei registri dallo stack
 - `$s` e `$f`; `$ra` e `$fp` per procedure non-foglia

```
lw reg, [dim_record_attivaz-N] ($sp)
```
- ❖ Liberare lo spazio nello stack in cui sono stati memorizzati i dati locali.

```
addi $sp, $sp, dim_record_attivaz.
```
- ❖ Trasferire il controllo al programma chiamante:

```
jr $ra
```



Assembly MIPS: Procedure annidate e recursive

Proff. A. Borghese, F. Pedersini

Dipartimento di Informatica
Università degli Studi di Milano

Procedure annidate



Procedure **annidate** = procedure "**non foglia**":
procedure che richiamano altre procedure

- ❖ Devono salvare nello stack:
 - l'indirizzo di ritorno (MIPS: registro 31: **\$ra**)
 - gli argomenti di input alla procedura: **\$a0÷\$a3**
 - ✦ Se la procedura chiamata richiede parametri, i registri **\$a0÷\$a3** potrebbero venire riscritti.
 - Il **record di attivazione** (es. variabili locali non temporanee)

Procedure annidate: esempio

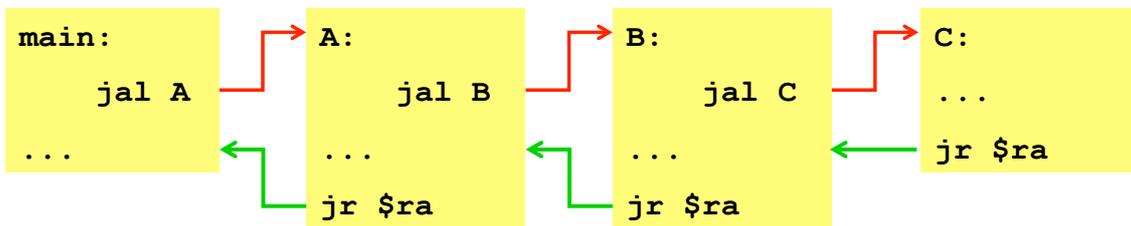


```

main:    ...
         li $a0, 3
         jal A
ra_M:   sw $v0, 0($t0)
         ...
A:      addi $sp,$sp,-32
         sw $a0 8($sp)
         ...
         li $a0, 24
         jal B
ra_A:   sw $v0,4($t2)
         add $a0, $s0, $a0
         ...
         addi
         $sp,$sp,32
         jr $ra

B:      addi $sp,$sp,-24
         ...
         li $a0, 68
         jal C
ra_B:   sw $v0,0($t3)
         ...
         jr $ra

C:      addi $sp,$sp,-12
         ...
         move $a0, $s4
         sw $v0, 4($t2)
         ...
         jr $ra
    
```



Procedure annidate



```

main:
  ...
  li $a0,3
  jal A
ra_M: sw $v0, 0($t0)
  ...

A:    addi $sp,$sp,-32
      sw $a0 8($sp)
      ...
      li $a0, 24
      jal B
ra_A: sw $v0,4($t2)
      add $a0, $s0, $a0
      ...
      addi $sp,$sp,32
      jr $ra
    
```

\$a0: 3

\$ra: ra_M

\$a0: 24

\$ra: ra_A



Procedure annidate (non-foglia)

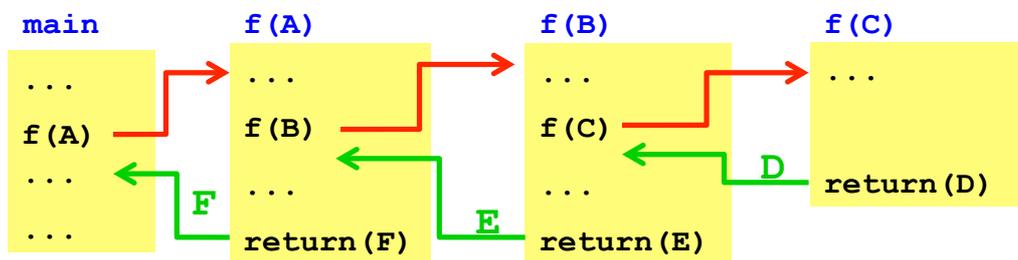


Da gestire esplicitamente in procedure annidate:

1. Modificare e poi ripristinare i **valori-argomento** (contenuti nei registri: **a0÷a3**)
 - ❖ **Soluzione: salvo \$a0÷\$a3 nello stack**
 1. **Prima** della chiamata (**jal**), salvo **\$a0÷\$a3** nello stack (**PUSH**)
 2. Chiamata/e a procedura: **jal...**
 3. **Dopo** tutte le chiamate, recupero **\$a0÷\$a3** dallo stack (**POP**)
2. Ripristinare l'**indirizzo di ritorno** dopo le chiamate (contenuto del registro: **ra**)
 - ❖ **Soluzione: salvo \$ra nello stack:**
 1. **Prima** della chiamata (**jal**), salvo **\$ra** nello stack (**PUSH**)
 2. Chiamata/e a procedura: **jal...**
 3. **Dopo** tutte le chiamate, recupero **\$ra** dallo stack (**POP**)



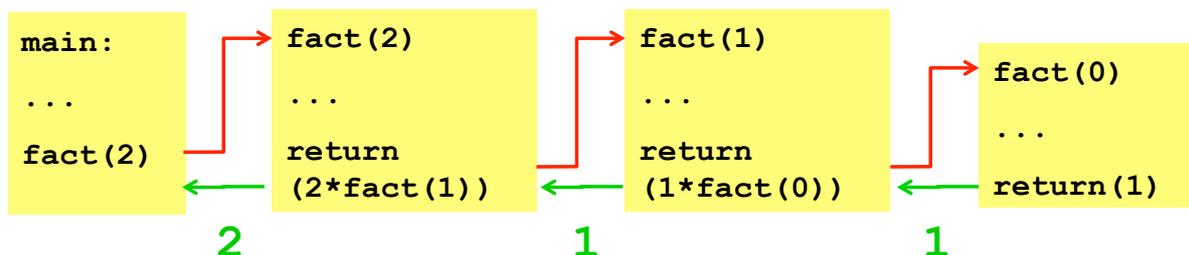
- ❖ **Procedure recursive:**
procedure annidate, che contengono **chiamate a sé stesse**
- ❖ **Problema** con **record di attivazione:**
tutte le variabili locali (registri) sono **condivise tra le istanze di chiamata**
- ❖ **Soluzione:** salvare nello stack:
 - L'indirizzo di ritorno
 - I parametri di input della procedura
- ❖ ...e in più: **tutto il record di attivazione.**
 - **Le variabili di lavoro** (anche i risultati intermedi, se a cavallo della chiamata!)



Fattoriale: esempio di calcolo



- ❖ Fattoriale: $n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot (n-1) \cdot n$
- ❖ in forma recursiva: $n! = n \cdot (n-1)!$ (ponendo: $0! = 1$)
- ❖ La procedura: **fact(n)** viene invocata **n+1 volte**
 - **n+1 record di attivazione** della stessa procedura:





codice C:

```
main(int argc, char *argv[])
{
    int n;
    printf("Inserire un numero intero ");
    scanf("%d", &n);
    printf("Fattoriale: %d\n", fact(n));
}

int fact(int m)
{
    if (m > 0)
        return( m * fact(m-1) );
    else
        return(1);
}
```

Fattoriale (implementazione Assembly)



```
# Programma per il calcolo ricorsivo di n!
#
    .data

prompt: .ascii "Inserire un numero intero:"
output: .ascii "Fattoriale:"

    .text
    .globl main

main:

# Lettura dell'intero di cui si calcola il fattoriale
#
    li $v0, 4      # $v0 ← codice della print_string
    la $a0, prompt # $a0 ← indirizzo della stringa
    syscall        # stampa della stringa

    li $v0, 5      # $v0 ← codice della read_int
    syscall        # legge l'intero n e lo carica in $v0
```



```
# Calcolo del fattoriale
add $a0, $v0, $zero      # $a0 ← n
jal fact                 # chiama fact(n)
add $s0, $v0, $zero     # $s0 ← n!

# Stampa del risultato
li $v0, 4                # $v0 ← codice della print_string
la $a0, output           # $a0 ← indirizzo della stringa
syscall                  # stampa della stringa di output

add $a0, $s0, $zero     # $a0 ← n!
li $v0, 1                # $v0 ← codice della print_int
syscall                  # stampa n!

# Termine del programma
li $v0, 10               # $v0 ← codice della exit
syscall                  # esce dal programma
```



```
# Procedura fact: calcolo del fattoriale di n
# argomenti: n (in a0) restituisce: n! (in v0)

fact:
    addi $sp, $sp, -8      # alloca stack
    sw   $ra, 4($sp)      # salvo return address
    sw   $a0, 0($sp)      # salvo l'argomento n

    bgt  $a0, $zero, core # se n > 0 salta a core
    li   $v0, 1           # $v0 ← 1
    j    end

core:
    addi $a0, $a0, -1     # decremento n → (n-1)
    jal  fact             # chiama fact(n-1) in $v0

    lw   $a0, 0($sp)      # ripristino n in $a0
    mul  $v0, $a0, $v0    # ritorno n * fact (n-1)

end:
    lw   $ra, 4($sp)      # ripristino return address
    addi $sp, $sp, 8      # dealloca stack
    jr   $ra              # ritorno al chiamante
```



```
fact:
    addi $sp, $sp, -8           # alloca stack
    sw   $ra, 4($sp)          # salvo return address
    sw   $a0, 0($sp)          # salvo l'argomento n

    bgt  $a0, $zero, core     # se n > 0 salta a core
    li   $v0, 1                # $v0 ← 1
    j    end

core:
    addi $a0, $a0, -1          # decremento n → (n-1)
    jal  fact                  # chiama fact(n-1) in $v0

    lw   $a0, 0($sp)          # ripristino n in $a0
    mul  $v0, $a0, $v0        # ritorno n * fact (n-1)

end:
    lw   $ra, 4($sp)          # ripristino return address
    addi $sp, $sp, 8           # dealloca stack
    jr   $ra                  # ritorno al chiamante
```



- ❖ Gli elementi interessati dalla ricorsione sono:
 - i registri `$a0` e `$ra`
- ❖ L'esecuzione è suddivisa in:
 - **ciclo superiore**
da inizio procedura a: `jal fact`
 - **ciclo inferiore**
da dopo: `jal fact` a fine procedura
- ❖ **A "cavallo" di `jal fact` posso trovare i registri modificati dalle altre chiamate!**
 - Se voglio evitare sorprese, li devo salvare nello stack
 - Se utilizzo risorse **solo prima** o **solo dopo `jal`** (non a cavallo), non c'è bisogno di salvarle.
- ❖ Cosa fa lo stack?
 - lo **stack cresce nel ciclo superiore** e **decrece nel ciclo inferiore**.

Calcolo serie di Fibonacci (implem. C)



Serie di Fibonacci: $fib(n) = fib(n-1) + fib(n-2)$
 $fib(0) = 0, fib(1) = 1$

Codice C:

```
main(int argc, char *argv[])
{
    int n;
    printf("Inserire un intero\n");
    scanf("%d", &n);
    printf("Numero di Fibonacci di %d = %d\n", n, fib(n) );
}

int fib(int m)
{
    if (m == 1)
        return(1);
    else if (m == 0)
        return(0);
    else
        return( fib(m-1) + fib(m-2) );
}
```

Fibonacci (Assembly – main)



```
# Programma per il calcolo ricorsivo del Numero di
# Fibonacci.

        .data

prompt:  .ascii "Inserire un numero intero: \n"
output:  .ascii "Numero di Fibonacci: "

        .text
        .globl main

main:
        li $v0, 4
        la $a0, prompt
        syscall          # stampa la stringa

        li $v0, 5
        syscall          # legge l'intero
```



```
# calcola fibonacci(n)
move $a0, $v0
move $a1, $0
jal fib

# salva il valore restituito da fib in t0
move $a1, $v0

# stampa il risultato
li $v0, 4
la $a0, output
syscall

move $a0, $a1
li $v0, 1
syscall

li $v0, 10 # esci
syscall
```

Fibonacci – procedura: *fib()*



```
# Fibonacci. Legge la somma parziale in a1 e in a0 il numero.
# Restituisce $v0 = fib($a0-1) + fib($a0-2).

fib:  addi $sp, $sp, -12
      sw  $ra, 8($sp)
      sw  $a0, 4($sp)      # salva i parametri in input
      sw  $a1, 0($sp)      # salva il risultato della precedente fib()

      li  $t0, 1
      bgt $a0, $t0, core   # se n>1, continua, altrimenti restituisci n
      add $v0, $a0, $zero  # n = 0 or n = 1.
      j   return

core:  addi $a0, $a0, -1    # n → n-1
      jal fib              # chiama fib(n-1)
      add $a1, $v0, $zero  # salva fib(n-1) in a1
      addi $a0, $a0, -1    # $a0 diventa: n-2
      jal fib              # esegue fib(n-2)
      add $v0, $v0, $a1    # somma fib(n-1) e fib(n-2)

return: lw $ra, 8($sp)
        lw $a0, 4($sp)
        lw $a1, 0($sp)
        addi $sp, $sp, 12
        jr $ra
```