Lezione 15

Assembly V:
Procedure annidate e recursive

Proff. A. Borghese, F. Pedersini

Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano

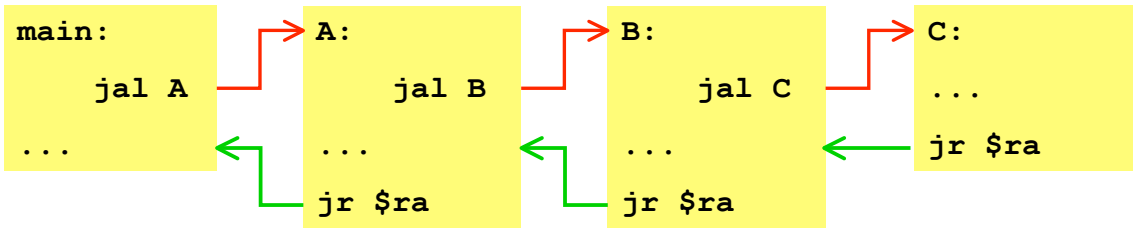
Procedure **annidate** = procedure ***"non foglia":***
procedure che richiamano altre procedure

- 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

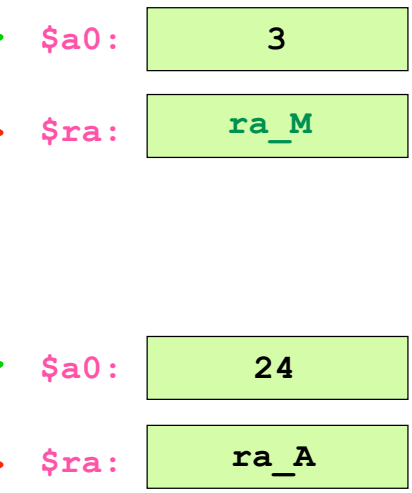
```
main:   ...
        li $a0, 3
        jal A
ra_M:   sw $v0, 0($t0)
        ...
A:      addi $sp,$sp,-32
        sw $a0 8($sp)
        ...
        li $a0, 24
        jal B
ra_A:   sw $v0,4($t2)
        add $a0, $s0, $a0
        ...
        addi
        $sp,$sp,32
        jr $ra
```

```
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
```

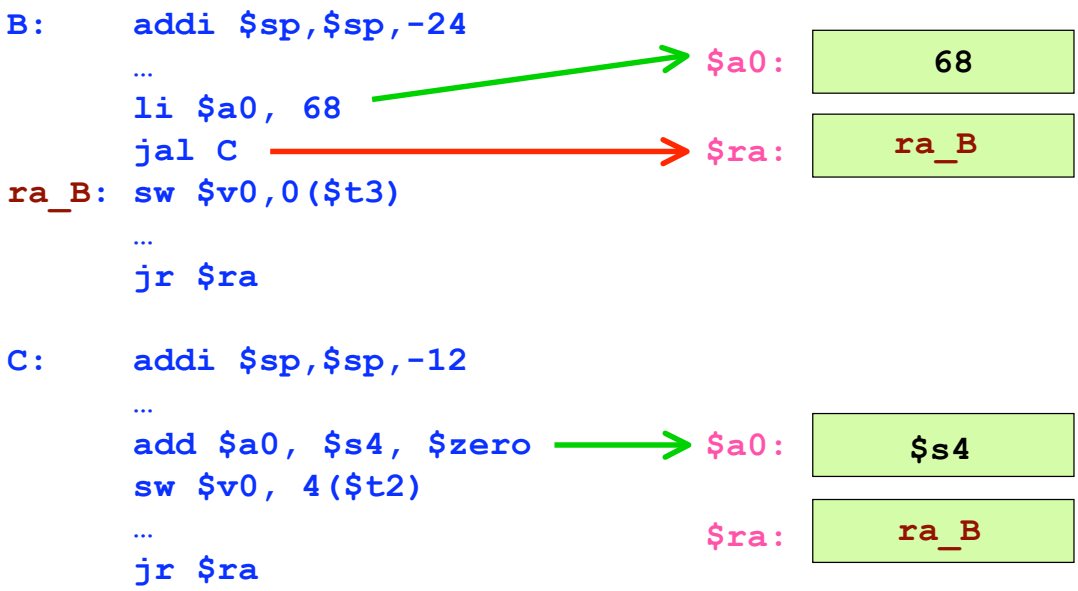


Procedure annidate

```
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
```



Procedure annidate



Procedure annidate (non-foglia)

Da gestire esplicitamente in procedure annidate:

1. Modificare e poi ripristinare i valori-argomento (contenuti nei registri: `a0÷a3`)

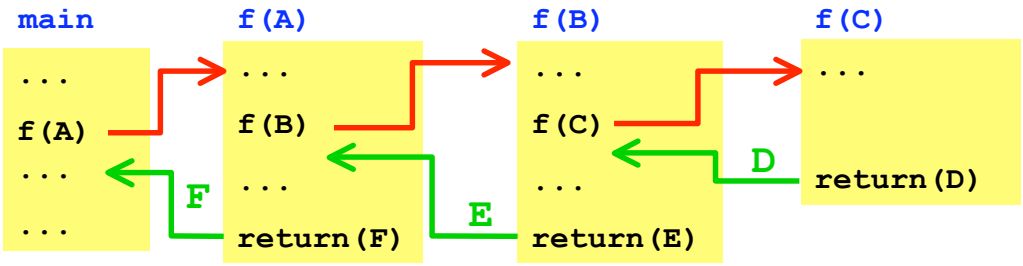
- Soluzione: salvo `$a0÷$a3` nello **stack**
  1. **Prima** della chiamata (`jal`) , salvo `$a0÷$a3` nello stack (PUSH)
  2. Chiamata/e a procedura: `jal...`
  3. **Dopo** tutte le chiamate, recupero `$a0÷$a3` dallo stack (POP)

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)

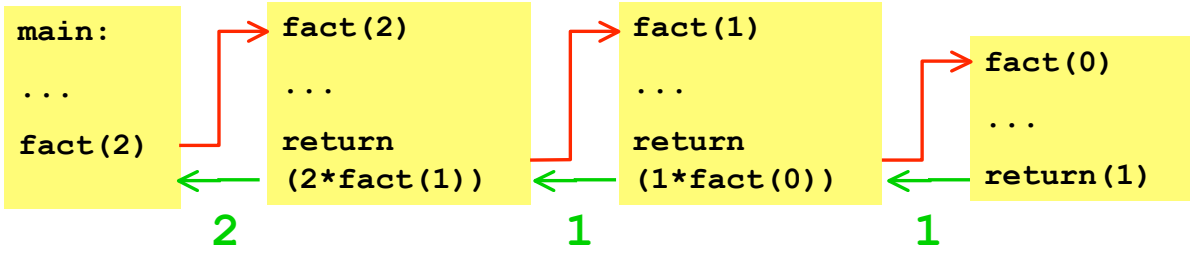
Procedure ricorsive

- **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! = 1·2·3· … (n−1)·n**
- in forma recursiva: **n! = n · (n−1)!** (ponendo: 0! = 1)
- La procedura: `fact(n)` viene invocata `n+1` volte
  - n+1 record di attivazione della stessa procedura:



Procedure ricorsive: Fattoriale

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: .asciiz "Inserire un numero intero:"
output: .asciiz "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
```

Fattoriale (cont.)

```
# 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
```


Fattoriale (procedura)

```
# 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
```

Fattoriale (procedura)

```
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
```


Osservazioni

- 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 decresce nel ciclo inferiore.

Calcolo serie di Fibonacci (implem. C)

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

$fib(0) = 0, \quad fib(1) = 1$

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

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

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

Fibonacci (cont. main)

```
        # 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
```

Esempio - Numeri primi – bozza in C

```
main(int argc, char *argv[])
{
    int primes_test = [1 2 3 5 7 11 13 17 19 23];
    int primes[10], n_primes, n_primes_test = 10, i;

    printf(”Inserire un intero\n"); scanf("%d", &n);
    printf(“I numeri primi sono: “);

    [n_primes primes] = find_primes( n, n_primes_test, primes_test, primes);

    for (i=0; i< n_primes; i++) printf(“%d “,primes[i]);
    exit(0);
}

[int n_primes, int primes[10]] = find_primes(int n,
                                             int n_primes_test,
                                             int primes_test[10],
                                             int primes[10])
{
    int t = n_primes_test, k = 0, l = n;
    while (t > 0)
        [primes[10], t] = find_first_prime( l, n_primes_test,
                                             primes_test[10],
                                             primes[10], t);
    return(…);
}
```


Numeri primi - Codice Assembly

```
        .data
prompt: .asciiz "Inserire un numero intero: "
output: .asciiz "I numeri primi sono: "
primes_test:
        .word 1 2 3 5 7 11 13 17 19 23
primes: .space 40
spazio: .asciiz " “

        .text
        .globl main

# $s0 ($a0) contiene il numero di numeri primi da testare (in byte).
# $s1 ($a1) contiene l’indirizzo del vettore dei numeri primi
# $s2 ($a2) contiene N, il numero primo da scomporre
# $s3 ($a3) contiene l’indirizzo del vettore dei numeri primi di N
# $s4 ($v0) contiene il numero di numeri primi di N (in byte).

main:   li $s0, 40      # Impostato alla lunghezza di primes_test (in byte)
        la $s1, primes_test
        li $v0, 4
        la $a0, prompt
        syscall         # stampa prompt
```

Numeri primi (cont.)

```
        li $v0, 5
        syscall                 # legge l'intero N
        move $s2, $v0

        la $s3, primes

        move $a0, $s0           # prepara la chiamata
        move $a1, $s1           # prepara la chiamata
        move $a2, $s2           # prepara la chiamata
        move $a3, $s3           # prepara la chiamata

        jal find_primes         # chiamata

        move $s4, $v0           # numero di primi di N (in byte)

        # scrivo anche il numero 1 come costituendo di N
        lw $t0, 0($s1)
        add $t1, $s4, $s3
        sw $t0, 0($t1)
        addi $s4, $s4, 4
```


Numeri primi (cont.)

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

        move $t0, $s3
        add $t1, $s4, $t0

Loop_w:
        beq $t0, $t1, fine

        li $v0, 4       # Stampa uno spazio
        la $a0, spazio
        syscall

        lw $a0, 0($t0)
        li $v0, 1
        syscall

        addi $t0, $t0, 4
        j Loop_w
fine:
        li $v0, 10
        syscall
```

Numeri primi (cont.)

```
find_primes:
        addi $sp, $sp, -8
        sw $ra, 0($sp)
        sw $a0, 4($sp)

        move $v0, $zero         # memorizza il numero di primi di N
                                # in numero di byte
        addi $a0, $a0, -4       # elemento di prime_test

# in find_first_prime:
#   a2 viene diviso via via per tutti i primi
#   a0 viene decrementato via via che i primi sono esaminati.

        jal find_first_prime

        lw $ra, 0($sp)
        lw $a0, 4($sp)
        addi $sp, $sp, 8
        jr $ra
```


Numeri primi (cont.)

```
find_first_prime:
       addi $sp, $sp, -4
       sw $ra, 0($sp)

       beq $a0, $zero, fine_1 # il ciclo termina quando ci si posiziona
                              # sul 1. elemento di primes_test (base + 4)
       add $t0, $a1, $a0
       lw $t2, 0($tx0)        # estraggo il numero primo da testare
       addi $a0, $a0, -4      # mi posiziono sul numero primo successivo
       rem $t3, $a2, $t2      # guardo se $t2 e' un divisore di N
       bgtz $t3, oltre        # se non lo e', passo al successivo n. primo
       div $a2, $a2, $t2      # divido N per il numero primo trovato
       add  $t4, $v0, $a3     # $t4 indirizzo su primes in cui scrivere
       sw $t2, 0($t4)         # memorizzo il numero primo trovato
       addi $v0, $v0, 4       # punto al nuovo elemento in primes
oltre:
       jal find_first_prime
fine_1:
       lw $ra, 0($sp)
       addi $sp, $sp, 4
       jr $ra
```