Le procedure ricorsive
Come eseguire un programma

Architettura
degli Elaboratori e delle Reti,
Turno I

Alberto Borghese
Università degli Studi di Milano
Dipartimento di Scienze dell'Informazione
email: borghese@dsi.unimi.it

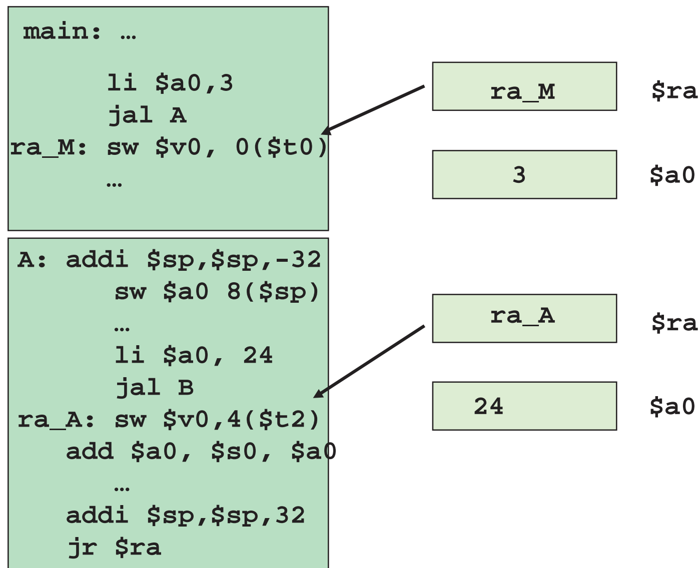
Procedure annidate

- Sono procedure che richiamano al loro interno altre procedure (non sono procedure foglia)
- Devono salvare nello stack un ambiente più ampio
- Rispetto alle procedure foglia, devono essere salvati anche:
  - i parametri di input della procedura ($a0, $a1, $a2, $a3)
  - l'indirizzo di ritorno ($ra)
    - la procedura chiamata all'interno di un'altra riscrive il contenuto di **`$ra`**
    - La procedura chiamata può richiedere dei parametri ed in questo caso i registri $ai potrebbero venire riscritti.

Procedure ricorsive

- Procedure che contengono una chiamata a se stesse al loro interno
- Devono salvare nello stack
  - i parametri di input della procedura
  - l'indirizzo di ritorno
  - eventuali risultati intermedi

Procedure ricorsive & annidate



Procedure ricorsive & annidate

```
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
```

ra_B	$ra
68	$a0
ra_B	$ra
[$s4]	$a0

Fattoriale

```
main(int argc, char *argv[])
{ int n;

  printf(”Inserire un numero intero\n");
  scanf("%d", &n);
  printf(”Fattoriale: %d\n", fact(n));
}

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

Fattoriale (Implementazione assembly)

```
# programma per il calcolo ricorsivo di n!
        .data
prompt: .asciiz "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
```

Fattoriale (cont.)

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

    move $a0, $s0       # $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)

```
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 $v1
    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

L'esecuzione e' suddivisa in un ciclo superiore (dall'etichetta fact: alla jal fact) ed in un ciclo inferiore (dalla prima istruzione dopo la jal fact - lw $a0, 0($sp) – alla fine della procedura.

Gli elementi maggiormente interessati dalla ricorsione sono:
-il registro $a0,
-il registro $ra,
-lo stack, che cresce nel ciclo superiore e decresce nel ciclo inferiore.

Fibonacci

```
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 (cont.)

```
        .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.)

```
# calcola fibonacci(n)

        move $a0, $v0
        move $a1, $0
        jal  fib

# stampa il risultato ed esci
        move $a1, $v0  # salva il valore restituito da fib in t0
        li $v0, 4
        la $a0, output
        syscall

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

        li $v0, 10
        syscall
```

Fibonacci (cont.)

```
# Fibonacci. Legge la somma parziale in a1 e in a0 il numero.
# Restituisce v0 = fib(a0-1)->a1 + fib (a0-2)->v0.
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
                              #chiamata di fib
        li   $t0, 1
        bgt  $a0, $t0, core   # se n> 1, continua,
                              # altrimenti restituisci n
        move $v0,$a0          # n = 0 or n = 1.
        j    return

core:
        addi $a0, $a0,-1      # n → n-1
        jal  fib              # chiama fib(n-1)
        move $a1, $v0         # salva fib(n-1) nello stack (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 - 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 da testare
# $s2 ($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

        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
```

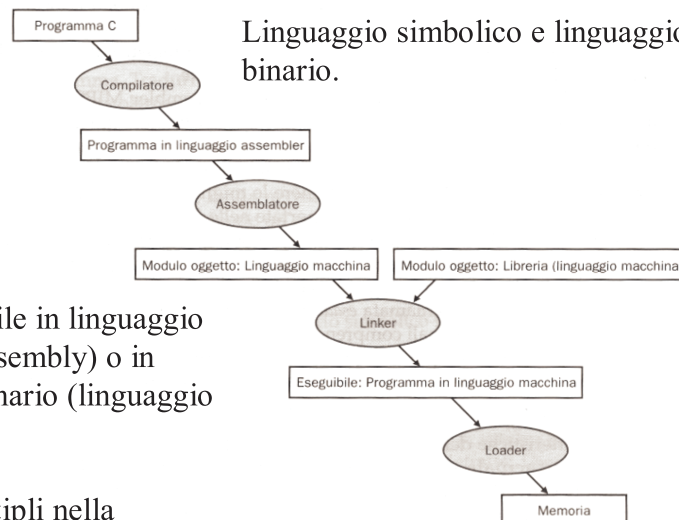
Numeri primi (cont.)

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

        beq $a0, $zero, fine_1    # il ciclo termina quando ci si posiziona sul
                                  # primo elemento di primes_test (cioe' base + 4)
                                  # (primo elemento di prime_test che e' = 1)
        add $t0, $a1, $a0
        lw $t2, 0($tx0)           # estraggo il numero primo da testare
        addi $a0, $a0, -4         # mi posiziono sul numero primo successivo da
                                  # esaminare
        rem $t3, $a2, $t2         # guardo se $t2 e' un divisore di N
        bgtz $t3, oltre           # se non e' un divisore di N, passo al successivo
                                  # numero primo da esaminare
        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
```

Dai simboli ai numeri binari: La catena di generazione di un programma binario

Dai simboli ai numeri binari

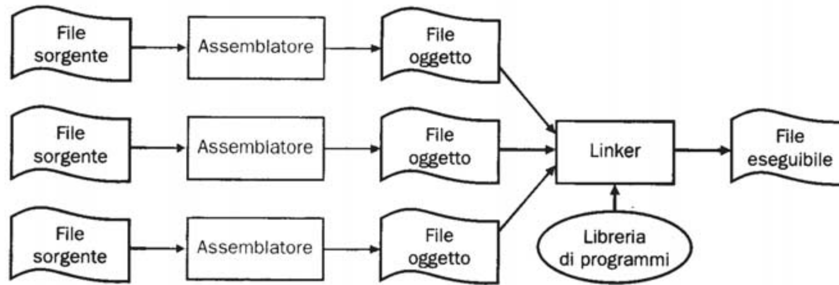


Linguaggio simbolico e linguaggio binario.

ISA esprimibile in linguaggio simbolico (assembly) o in linguaggio binario (linguaggio macchina).

Passaggi multipli nella traduzione.

Il processo di creazione di un programma



Dall'assembly al binario

Calcolo dei primi 100 numeri al quadrato

```
# N e' memorizzato in t1.

.data
.align 2
str:
          .asciiz "La soma da 0 a N vale "
.text
.globl main

main:
          li        $t1, 2
          li        $t6, 0
          li        $t8, 0
Loop:
          mul       $t7, $t6 $t6
          addu      $t8, $t8, $t7
          addi      $t6, $t6, 1
          ble       $t6, $t1, Loop

          la        $a0, str
          li        $v0, 4
          #print
          syscall

          li        $v0, 1
          #print
          add       $a0, $t8, $zero
          syscall

          li $v0, 10
          syscall
```

```
00100111101111011111111111100000
10101111101111110000000000010100
10101111101001000000000000100000
10101111101001010000000000100100
10101111101000000000000000011000
10101111101000000000000000011100
10001111101011100000000000011100
10001111101110000000000000011000
00000001110011100000000000011001
00100101110010000000000000000001
00101001000000010000000001100101
10101111101010000000000000011100
00000000000000001111000000010010
00000011000011111100100000100001
00010100001000001111111111110111
10101111101110010000000000011000
00111100000001000001000000000000
10001111101001010000000000011000
00001100000100000000000011101100
001001001000010000000010000110000
10001111101111110000000000010100
00100111101111010000000000100000
00000011111000000000000000001000
00000000000000000001000000100001
```

L'assemblatore

1) Individua le etichette e gli indirizzi associati (symbol table).

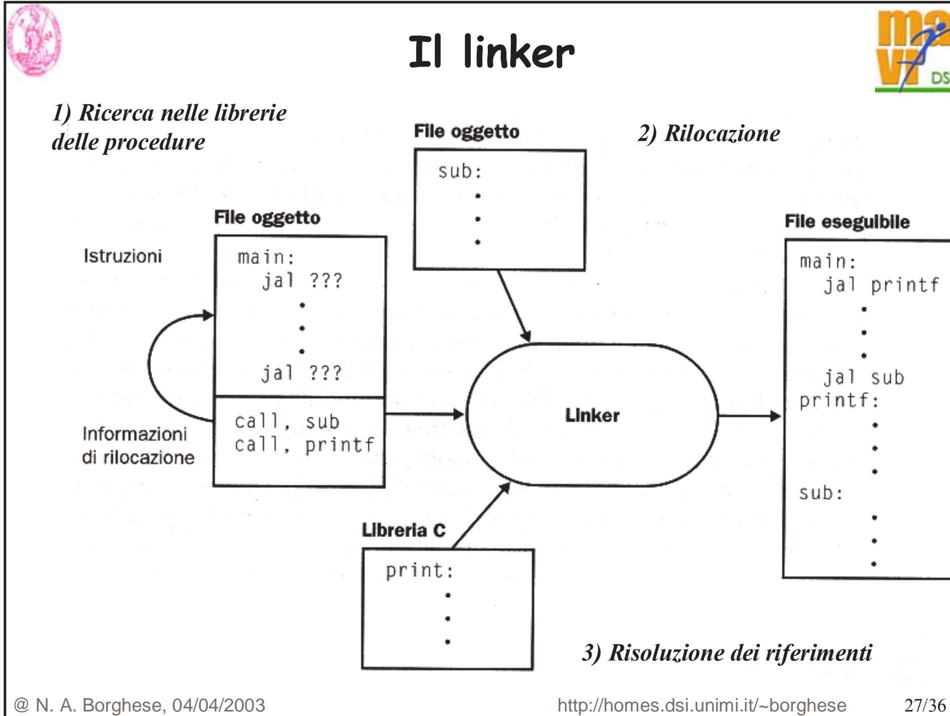
2) Traduzione delle istruzioni con sostituzione degli indirizzi alle etichette.

3) Le etichette non risolte saranno definite in altri moduli.

Componenti del modulo oggetto

1. Intestazione del file oggetto. Dimensione e posizione.
2. Segmento di testo. Codice in linguaggio macchina del sorgente.
3. Segmento dati. Contiene la rappresentazione binaria dei dati statici.
4. Informazioni di rilocazione.
5. Tabella dei simboli.
6. Informazioni per il debugger (se presente).

Il linker



Loader

•Lettura dell’intestazione del file eseguibile per determinare la lunghezza dei segmenti testo e dati.

•Creazione di uno spazio di indirizzamento sufficiente a contenere testo e dati.

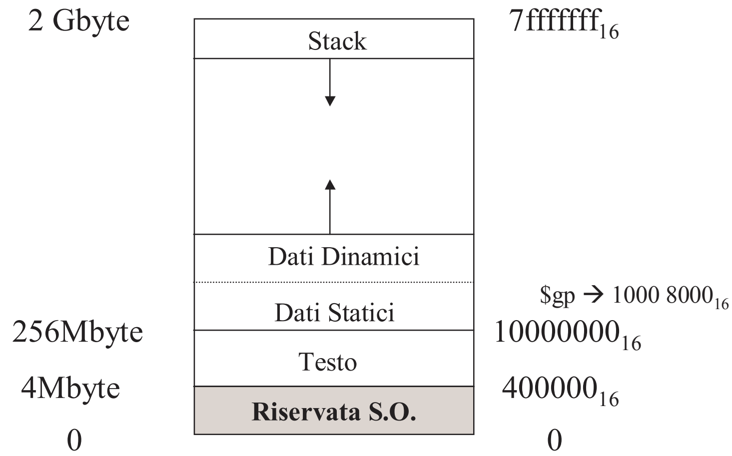
•Copia delle istruzioni dei dati dal file eseguibile in memoria.

•Copia dello stack degli eventuali parametri passati al programma principale.

•Inizializzazione dei registri dell’elaboratore e posizionamento dello stack pointer alla prima posizione libera.

•Salto ad una procedura di inizializzazione (start-up) che copia i parametri nei registri argomento e chiama la procedura principale del programma. La procedura di start-up termina con la chiamata di sistema *exit*.

Organizzazione logica della memoria



I passi del linker

Il linker lavora su due procedure: A e B

Intestazione del file oggetto			
	Nome	Procedura A	
	Dimensione del testo	100_{esa}	
	Dimensione dei dati	20_{esa}	
Segmento di testo	Indirizzo	Istruzione	
	0	lw $a0, 0($gp)	
jal B	4	jal 0	
	…	…	
Segmento di dati	0	(X)	
	…	…	
Informazioni di rilocazione	Indirizzo	Tipo di istruzione	Dipendenza
	0	lw	X
	4	jal	B
Tabella dei simboli	Etichetta	Indirizzo	
	X	-	
	B	-	

I passi del linker: procedura B

Intestazione del file oggetto			
	Nome	Procedura B	
	Dimensione del testo	200_{esa}	
	Dimensione dei dati	30_{esa}	
Segmento di testo	Indirizzo	Istruzione	
	0	lw $a1, 0($gp)	
jal A	4	jal 0	
	...	...	
Segmento di dati	0	(Y)	
...	...		
Informazioni di riallocazione	Indirizzo	Tipo di istruzione	Dipendenza
	0	lw	Y
	4	jal	A
Tabella dei simboli	Etichetta	Indirizzo	
	Y	-	
	A	-	

Risultato

Intestazione del file eseguibile		
	Dimensione del testo	300esa
	Dimensione dei dati	50esa
Segmento di testo	Indirizzo	Istruzione
A:	$0040\ 0000_{esa}$	lw $a0, 8000_{esa}($gp)
jal B	$0040\ 0004_{esa}$	jal $40\ 0100_{esa}$
	...	...
B:	$0040\ 0100_{esa}$	sw $a1, 8020_{esa}($gp)
jal A	$0040\ 0104_{esa}$	jal $40\ 0000_{esa}$
	...	...
Segmento di dati	Indirizzo	
	$1000\ 0000_{esa}$	(X)
	...	...
	$1000\ 0020_{esa}$	(Y)
	...	...

Calcoli degli indirizzi

Segmento testo:

Inizia dopo il segmento riservato al S.O., indirizzo 0x400 000 = 0100 0000 0000 0000 0000 0000 binario = 1x 2^{22} = 4Mbyte.

Procedura A. Inizia subito dopo. Indirizzo 0 della procedura è l'indirizzo 0x400 000.

Procedura B. Inizia dopo la procedura A. Indirizzo 0 della procedura B è: 0x400 000 + 0x100 (dimensione della procedura B) = 0x400 100.

Queste osservazioni consentono di sostituire le etichette di salto.

Calcoli degli indirizzi

Segmento dati:

Inizia dopo il segmento riservato ai dati, indirizzo 0x1000 0000 = 0001 0000 0000 0000 0000 0000 0000 0000 binario = 1x 2^{28} byte = 256Mbyte.

I dati della procedura A vengono scritti immediatamente sopra i 256 Mbyte. Per indirizzarli occorre una procedura in due passi:

- Impostare correttamente il $gp.
- Impostare correttamente l'offset.

Impostazione corretta del $gp

Il $gp deve puntare ad un indirizzo 32kbyte sopra il limite del segmento dati (256Mbyte = 0x1000 0000 byte).

$gp = 0x1000 0000 + 0x8000 = 0x1000 8000 byte.

[0x8000 = 1000 0000 0000 0000 byte = $1x2^{15}$ byte.

Impostazione corretta dell'offset

I dati della procedura A possono essere scritti direttamente sopra il segmento testo. L'indirizzo di partenza dei dati, X, è l'indirizzo 256Mbyte.

Dobbiamo esprimere questo indirizzo in termini relativi al $gp:

X = 0x1000 8000 – 0x8000 = 0x1000 0000.

Per leggere il primo dato in memoria opereremo una:
lw $registro, -0x8000($sp).

-0x8000 = (su 16 bit) 1000 0000 0000 0000 = -2^{15}, in complemento a 2!!