Università degli Studi di Milano
Dipartimento di Informatica "Giovanni Degli Antoni"
Corso di Laurea Triennale in Informatica

Architettura degli Elaboratori II
Laboratorio

Procedure 2/2: Procedure annidate e ricorsive

1

Procedure «foglia»

- Scenario più semplice: `main` chiama la procedura `funct` che, senza chiamare a sua volta altre procedure, termina e restituisce il controllo al `main`

main

```
f = f + 1;

if (f == g)
        res = funct(f,g);

else
        f = f -1;

print(res)
```

funct

```
int funct (int p1, int p2){

        int out;
        out = p1 * p2;
        return out;
}
```

- Una procedura che non ne chiama un'altra al suo interno è detta procedura *foglia*

  Motivo: si può rappresentare l'esecuzione del programma con un albero: i nodi sono le procedure invocate. La radice è il main. Un nodo B è figlio di A se A invoca B. In questo albero, le procedure che non invocano altre procedure sono foglie.

3

Procedure non «foglia»

- Una procedura che può invocarne un'altra durante la sua esecuzione non è una procedura foglia, ha annidata al suo interno un'altra procedura:

main

```
f = f + 1;

if (f == g)
          res = funct(f,g);

else
          f = f -1;

print(res)
```

funct

```
int funct (int p1, int p2){

          int out,x;
          x = p1 * p2;
          out = funct2(x)
          return out;
}
```

funct2

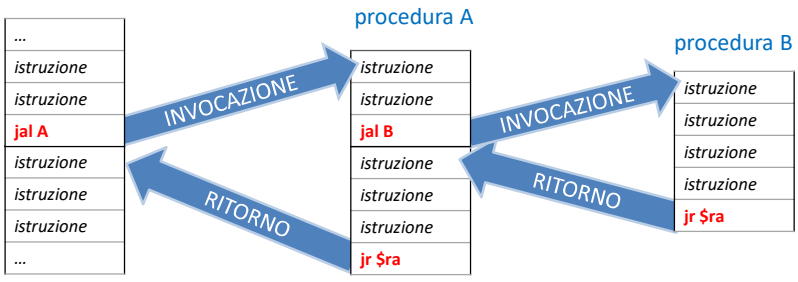
```
int funct2 (int p1){

          return p1^2;
}
```

- Se una procedura contiene una chiamata ad un'altra procedura dovrà effettuare delle operazioni per (1) garantire la non-alterazione dei registri opportuni (2) consentire una restituzione del controllo consistente con l'annidamento delle chiamate.
- *Ricordiamo: in assembly la modularizzazione in procedure è un'assunzione concettuale sulla struttura e sul significato del codice. Nella pratica, ogni «blocco» di istruzioni condivide lo stesso register file e aree di memoria*

4

Invocazione di procedura annidate



5

Convenzione sull'uso dei registri da parte delle procedure

Registro:	$0	$1	$2	$3	$4	$5	$6	$7
Sinonimo:	$r0	$at	$v0	$v1	$a0	$a1	$a2	$a3
Registro:	$8	$9	$10	$11	$12	$13	$14	$15
Sinonimo:	$t0	$t1	$t2	$t3	$t4	$t5	$t6	$t7
Registro:	$16	$17	$18	$19	$20	$21	$22	$23
Sinonimo:	$s0	$s1	$s2	$s3	$s4	$s5	$s6	$s7
Registro:	$24	$25	$26	$27	$28	$29	$30	$31
Sinonimo:	$t8	$t9	$k0	$k1	$fp	$sp	$s8	$ra

deve rimanere invariato dopo la chiamata

può essere modificato dalla procedura

6

Local variables (in Go)

```
func pippo() {
   /* vengono dichiarate (e allocate) nuove variabili locali */
   var a, b, c int

   a = 10
   b = 20
   c = a + b
   fmt.Printf ("value of a = %d, b = %d and c = %d\n", a, b, c)

   /* alla fine della funzione, tutte le variabili locali
      vengono automaticamente deallocate */
}
```

7

Local variables (in Go)

```
func pippo() {
   var a , b , c int
   a = 10
   b = 20
   c = a + b
   if a%2 == 0 {
        var d , e , f int
        ...
        for j := 7; j <= 9; j++ {
            k := j+3
            fmt.Println(k)
        }
   } else {
        pippo := 6
        …
   }
   …
}
```

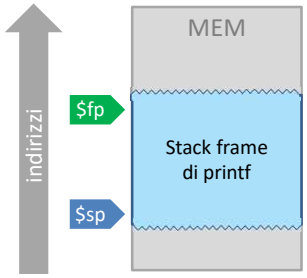
Nuove variabili locali sono aggiunte in vari punti dell'esecuzione di una funzione (compreso il main)

8

Record di attivazione – Stack frame

- Una procedura ha bisogno di usare la memoria
  - Per memorizzare le sue variabili locali
  - Per memorizzare la copia dei registri da preservare

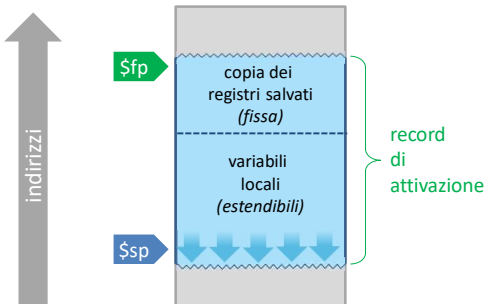
- Dedichiamo ad ogni procedura in esecuzione una sua area di memoria *sullo stack*, detta record di attivazione o stack frame
- MIPS riserva due registri per indirizzare lo stack frame della procedura attualmente in esecuzione:
  da `$sp` (stack pointer)
  a `$fp` (frame pointer)
  compresi!



9

Il record di attivazione di una funzione

- Il record di attivazione di una procedura memorizza
  - La copia dei registri da preservare per il chiamante
  - Le variabili locali (attaverso push e pop, come normale)

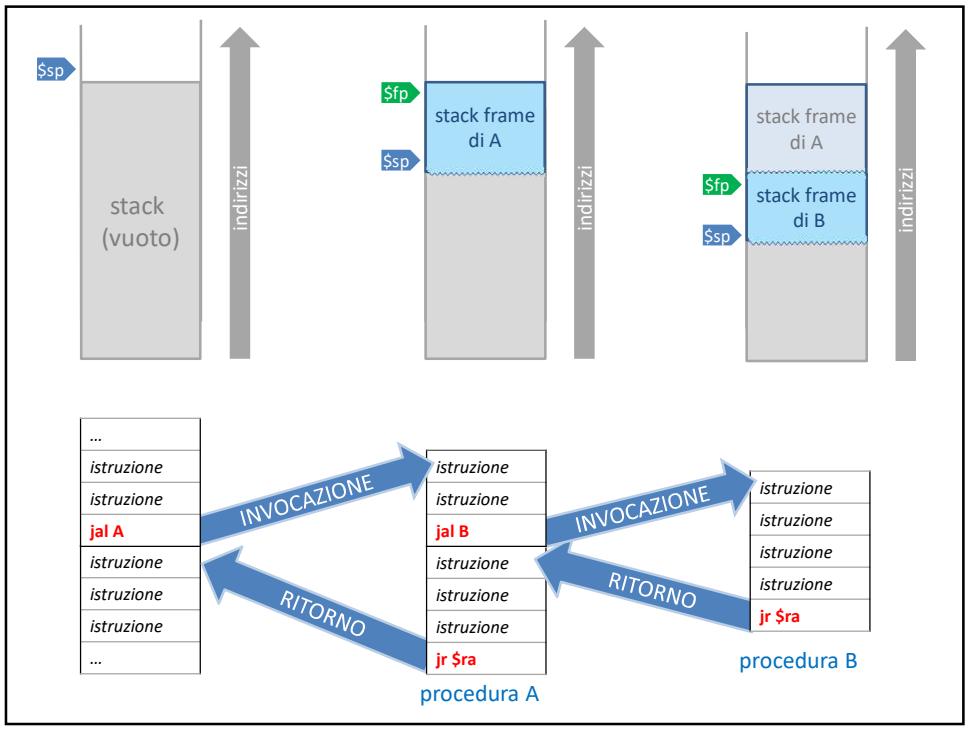


10

Allocazione e deallocazione degli stack frame

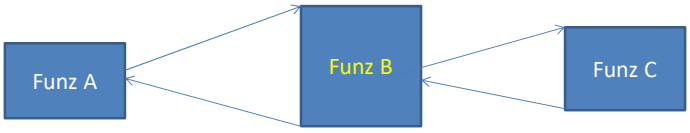
- I record di attivazione si impilano (LIFO) in memoria sullo stack
- quando una procedura viene invocata, un nuovo record di attivazione viene impilato nello stack
  - sotto al precedente
  - modificando i registri $sp e $fp
- quando una procedura termina, il suo record di attivazione (che è sempre quello in cima allo stack) viene rimosso
  - modificando i registri $sp e $fp
  - nota: non è necessario «pulire la memoria» sovrascrivendola con valori 0 semplicemente, l'area dello stack verrà riutilizzata dalle prossime procedure o variabili locali

11



12

Problemi per le procedure non «foglia»



La funzione B ha alcuni problemi da risolvere…
Problema 1:

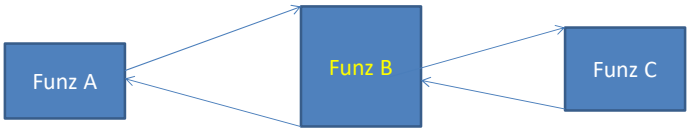
- Se B usa registri $t questi vengono (potenzialmente) distrutti da C, lecitamente ☹
- Se B usa registri $s (in scrittura), contravviene al «contratto» con A ☹
- Quali registri deve usare B?

Problema 2:

- Quando B usa la JAL per invocare C, sovrascrive il $ra
- Al momento di tornare ad A, non ha più l'indirizzo di ritorno originale!

13

Problemi per le procedure non «foglia»



Soluzione ad entrambi i problemi:

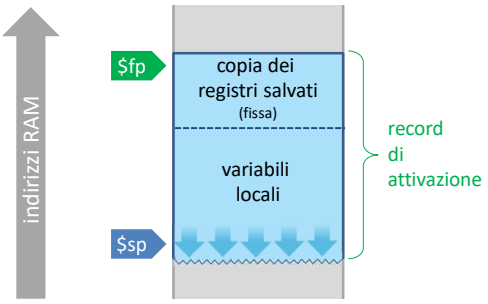
Prima di usare i registri $s (ma anche $ra, e gli altri che non possono cambiare)
Funz B li memorizza nel proprio RECORD DI ATTIVAZIONE

La stessa strategia vale anche per $ra e $sp

14

Differenze fra $fp (inizio) e $sp (fine del record di attivaz)?

- $sp può variare nel corso della procedura
  - scende quando il record di attivazione si espande per ospitare nuove variabili locali
  - semplicemente, indentifica la fine dello stack
- $fp invece *non cambia* durante l'esecuzione della procedura
- $fp è comodo per tener traccia di dove sono stati salvati i registri e dove sono state alloggiate le variabili locali



15

Manuale per scrivere una procedura
(versione completa, per non-foglie)

preambolo

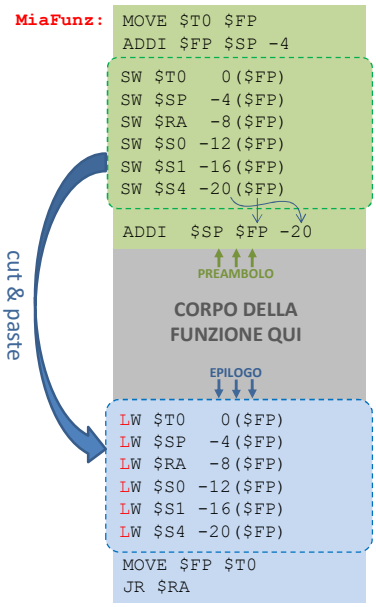
1. Aggiornare $fp al primo indirizzo utilizzabile nello stack, cioè $sp-4
   - Ma prima, fare una copia del suo valore iniziale in un registro temporaneo (es $t0)
2. salvare nello stack una copia del valore iniziale di tutti i registri necessari alla funzione e cioè:
   - **$fp** (il suo valore originale, non quello modificato al passo 1, la cui copia è in $t0)
   - **$sp** ed **$ra** (che viene sovrascritto dalla jal usata per invocare sotto-funzioni)
   - se serve: gli **$s0**..**$s7** che verranno usati nel corpo
   - come: memorizzandoli con delle store-word a indirizzo $fp, $fp-4, $fp-8 , $fp-12 …
3. aggiornare il valore di $sp all'ultimo indirizzo usato dello stack
4. implementare la procedura!
   - leggere gli eventuali input da $a0 .. $a3
   - utilizzare liberamente $t0..$t9, e/o $s0..$s7 solo se sono stati salvati nel passo 2
   - Invocare liberamente sottofunzioni
   - se servono nuove variabili locali, ingrandire il record di attivazione (decrementando $sp)
   - scrivere l'eventuale output in $v0 .. $v1

epilogo

5. ripristinare tutti i registri salvati sul frame stack nel passo 2
   - come: con altrettante load-word agli stessi indirizzi usati nel passo 2
   - nota: ripristinando **$sp** e **$fp** il frame buffer della funzione viene scartato, tornando quello precedente
   - nota: il $ra ripristinato viene usato al passo successivo
6. restituire il controllo al chiamante
   - con una jump-register a **$ra**

17

Guida pratica per funzioni non-foglia



Copia temporanea del Frame Pointer *iniziale* (in $T0).
Perché il *nuovo* record di attivazione comincia subito dopo il vecchio.

I valori dei registri *iniziali* sono salvati (in qualsiasi ordine) nel (nuovo) record di attivazione.
Compreso lo stack pointer SP, il Return Address RA, e anche FP stesso (sotto forma di T0)

Aggiornamento dello SP
(che punta sempre all'ultimo elemento occupato dello stack)

La funzione può
- usare i registri S solo se sono stati salvati (qui: S0, S1, S4).
- invocare altre funzioni (quindi usando RA),
- allocare variabili «locali» nello stack (quindi usando SP).
- usare i registri T, A e V ma sapendo che non vengono mantenuti dopo l'invocazione di eventuali altre funzioni

Ripristino del valore iniziale di tutti i registri salvati, compreso lo SP (flush dello stack)

… e compreso il FP
Ritorno al chiamante (usando il RA appena ripristinato)

19

Esercizi

1) Scrivere una procedura che converta in maiuscolo una stringa in input.
Suggerimento: usare SB e LB (StoreByte e LoadByte) per accedere ai singoli caratteri.
Cosa succede per la stringa “Hello World?”

2) Adattare le funzione per convertire solo le lettere minuscole tramite un ulteriore procedura “MaiuscolizzaLettera” che agisca su una sola lettera alla volta data in input.

21

Ricorsione

- La risoluzione di un problema P è costruita sulla base della risoluzione di un sottoproblema di P
- Esempio: il fattoriale di n $$n! = \prod_{k=1}^{n} k = n \prod_{k=1}^{n-1} k = n \times (n-1)!$$
- il fattoriale di n è uguale a n moltiplicato per il fattoriale di $n-1$, ma, quando $n = 0$ è uguale a 1. Quindi, una definizione ricorsiva è:

$$n! = \begin{cases} n \times (n-1)! & \text{if } n > 0 \\ 1 & \text{if } n = 0. \end{cases}$$

22

Ricorsione

- Applico la regola in cascata

$$n! = \begin{cases} n \times (n-1)! & \text{if } n > 0 \\ 1 & \text{if } n = 0. \end{cases}$$

$$\begin{aligned} 4! &= 4 \times (3)! \\ 3! &= 3 \times (2)! \\ 2! &= 2 \times (1)! \\ 1! &= 1 \times (0)! \\ 0! &= 1 \\ &= 4 \times 3 \times 2 \times 1 \times 1 \end{aligned}$$

28

Funzioni ricorsive

- Una funzione che invoca se stessa
- La definizione ricorsiva…

$$n! = \begin{cases} n \times (n-1)! & \text{if } n > 0 \\ 1 & \text{if } n = 0. \end{cases}$$

può essere convertita facilmente in una funzione ricorsiva in un programma ad alto livelo (qui: il Go)

```
func fattoriale (n int) int {
    if n == 0 {
        return 1  // caso base
    } else {
        return n * fattoriale(n-1) // caso ricorsivo
    }
}
```

- Le funzioni ricorsive sono casi di funzioni (evidentemente) non foglie e a basso livello vanno interpretate come tali (vedi esercizi)

29

Altro esempio: fibonacci

- E' una successione di numeri naturali $F_0, F_1, F_2,$
- Comincia con i due elementi 1 e 1,
  e ogni elemento successivo è la somma dei due precedenti
  1,1,2,3,5,8,13,21 …
- Definizione ricorsiva:

$$F_i = \begin{cases} 1 & \text{se } i = 0 \text{ o } i = 1 \\ F_{i-1} + F_{i-2} & \text{altrimenti} \end{cases}$$

- Suo calcolo (ricorsivo) in Go:

```
func Fibonacci (i int) int {
    if i <= 1 {
        return 1
    }
    return Fibonacci (i-1) + Fibonacci (i-2)
}
```

30

Funzioni ricorsive

- Funzione (o procedura) ricorsiva: è una procedura che per risolvere il problema P invoca se stessa per risolvere un sotto-problema di P
- Una procedura ricorsiva quindi non è mai una procedura foglia (perché, per definzione, invoca se stessa)
- Una procedura ricorsiva è:
  - Un callee: deve salvare i registri callee-saved ($s0, …, $ra, $fp)
  - Un caller: deve salvare i registri caller-saved ($t0, …, $a0, …, $v0, $v1)
- Tip: deve esserci sempre un «caso base»:
  una situazione in cui la funzione NON invoca se stessa.
  Altrimenti: la funzione continua ad invocare se stessa…
  fino ad esaurimento dello stack (stack overflow)

31