La chiamata delle procedure

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

Chiamata a procedura: esempio

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

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

Chiamata a procedura

Ci sono due ***attori:***
- Procedura chiamante.
- Procedura chiamata.

- La procedura **chiamante** deve eseguire le seguenti operazioni:
  - Predisporre i parametri di ingresso della procedura in un posto accessibile alla procedura
  - Trasferire il controllo alla procedura

Chiamata a procedura

- La procedura **chiamata** deve eseguire le seguenti operazioni:
  - Allocare lo spazio di memoria necessario alla memorizzazione dei dati e alla sua esecuzione (record di attivazione)
  - Eseguire il compito richiesto
  - Memorizzare il risultato in un luogo accessibile al chiamante
  - Restituire il controllo al chiamante

Allocazione dei registri

- Convenzioni per l'allocazione dei registri per le chiamate a procedura:
  - **`$a0-$a3 ($f12-$f15)`** registri **argomento** usati dal chiamante per il passaggio dei parametri
    - Se i parametri sono più di 4 si passano mediante la memoria (stack)
  - **`$v0,$v1 ($f0, …, $f3)`** registri **valore** sono usati dalla procedura per memorizzare i valori di ritorno
  - **`$ra`** **(return address)** registro di ritorno per memorizzare l'indirizzo della prima istruzione del chiamante da eseguire al termine della procedura

Chiamata a procedura

- Necessaria un'istruzione apposita che cambia il flusso di esecuzione (salta alla procedura) e salva l'indirizzo di ritorno (istruzione successiva alla chiamata di procedura): **`jal`** **(jump and link)**
- **`jal Indirizzo_Procedura`**
  - Salta all'indirizzo con etichetta **`Indirizzo_Procedura`** e memorizza il valore corrente del Program Counter (indirizzo dell'istruzione successiva ***PC+4***) in **`$ra`**
- La procedura come ultima istruzione esegue **`jr $ra`** per effettuare il salto all'indirizzo di ritorno della procedura.

Riassumendo

- Il programma **chiamante** deve:
  - Mettere i valori dei parametri da passare alla procedura nei registri **`$a0-$a3`**
  - Utilizzare l'istruzione **`jal address`** per saltare alla procedura e salvare il valore di **`(PC+4)`** nel registro **`$ra`**
- La procedura **chiamata** deve:
  - Eseguire il compito richiesto
  - Memorizzare il risultato nei registri **`$v0,$v1`**
  - Restituire il controllo al chiamante con l'istruzione **`jr $ra`**

Problemi

- Mantenere i valori passati come parametri alla procedura.
- Una procedura può avere bisogno di più registri rispetto ai 4 a disposizione per i parametri e ai 2 per la restituzione dei valori.
- Salvare i registri che una procedura potrebbe modificare, ma che il programma chiamante ha bisogno di mantenere inalterati.
- Fornire lo spazio necessario per le variabili locali alla procedura.
- Gestione di procedure annidate (procedure che richiamano al loro interno altre procedure) e procedure ricorsive (procedure che invocano dei 'cloni' di se stesse).

utilizzo dello stack

Lo stack

- Lo stack (pila) è una struttura dati costituita da una coda LIFO (last-in-first-out)
- I dati sono inseriti nello stack con l'operazione ***push***
- I dati sono prelevati dallo stack con l'operazione ***pop***
- E' necessario un puntatore al ***top*** dello stack per salvare i registri che servono al programma chiamato.
- Il registro **$sp (stack pointer o puntatore allo stack)** contiene l'indirizzo del top dello stack e viene aggiornato ogni volta che viene inserito o estratto il valore di un registro.

Gestione dello stack nel 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.
- L'inserimento di un dato nello stack **(operazione di push)** avviene **decrementando $sp** per allocare lo spazio e si esegue sw per inserire il dato.
- Il prelevamento di un dato dallo stack **(operazione di pop)** avviene **incrementando $sp** (per eliminare il dato) e riducendo quindi la dimensione dello stack.

Gestione dello stack nel MIPS

- Tutto lo spazio in stack di cui ha bisogno una procedura (**record di attivazione**) viene *esplicitamente* allocato dal programmatore in una sola volta, all'inizio della procedura
- Lo spazio nello stack viene allocato **sottraendo** a **$sp** il numero di byte necessari:
  - Es:

```
addi $sp,$sp,-24   #alloca 24 byte nello stack
```


Gestione dello stack nel MIPS

- Al rientro da una procedura il record di attivazione viene rimosso dalla procedura (deallocato) incrementando **$sp** della stessa quantità di cui lo si era decrementato alla chiamata
  - Es:

```
addi $sp, $sp,24   #dealloca 24 byte nello stack
```

- È necessario liberare lo spazio allocato per evitare di riempire tutta la memoria

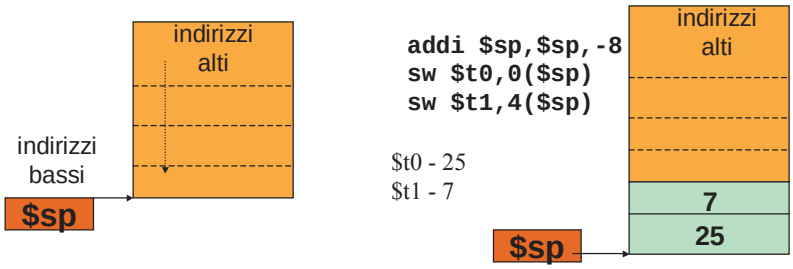
Gestione dello stack nel MIPS

- Per inserire elementi nello stack

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

- Per recuperare elementi dallo stack

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



Lo stack

- Quando si chiama una procedura ***i registri utilizzati dal chiamato*** vanno:
  - salvati nello stack
  - il loro contenuto va ripristinato alla fine dell'esecuzione della procedura

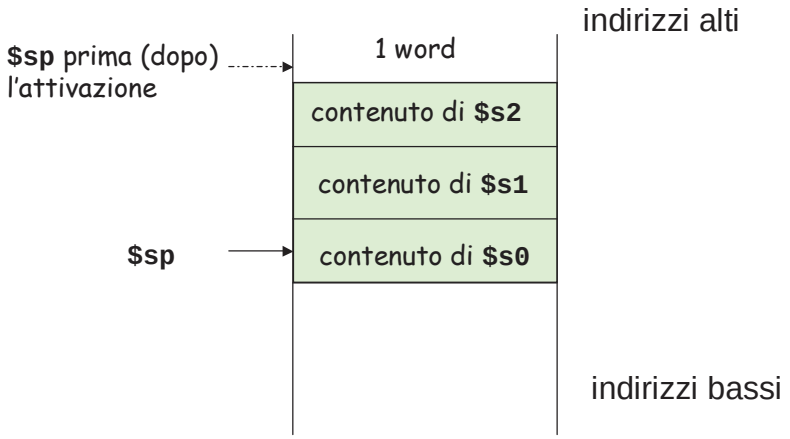
Esempio

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

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


Esempio

```
# g,h,i e j associati a $a0, …, $a3;
# f associata ad $s0
# il calcolo di f richiede 3 registi: $s0, $s1, $s2
# 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.)



Esempio (cont.)

```
add $s0, $a0, $a1          # $t0 ← g + h
add $s1, $a2, $a3          # $t1 ← i + j
sub $s2, $t0, $t1          # f ← $t0 - $t1

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 $s0
lw $s1, 4($sp)             # ripristino di $t0
lw $s0, 8($sp)             # ripristino di $t1

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

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

Lo stack

- Per evitare di salvare inutilmente il contenuto dei registri, i registri sono divisi in due classi:
  - **registri temporanei**:
    `$t0, …, $t9 ($f4, .. $f11, $f16, .., $f19)`
    il cui contenuto ***non è salvato*** dal chiamato nello stack (viene salvato dal chiamante se serve);
  - **registri non-temporanei**:
    `$s0, …, $s8 ($f20, …, $f31)`
    il cui contenuto è ***salvato*** nello stack ***se utilizzati dal chiamato.***

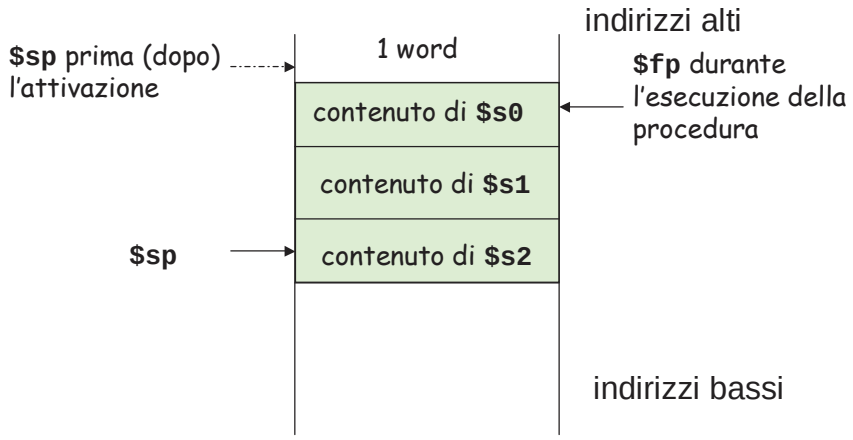
- Si può eliminare il salvataggio dei registri $s0 e $s1, utilizzando al loro posto i registri $t, ed eliminare l'utilizzo del registro $s2:

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

può diventare:

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


Il frame pointer



Uso dei registri: convenzioni

Nome	Numero	Utilizzo
$zero	0	costante zero
$at	1	riservato per l'assemblatore
$v0-$v1	2-3	valori di ritorno di una procedura
$a0-$a3	4-7	argomenti di una procedura
$t0-$t7	8-15	registri temporanei (non salvati)
$s0-$s7	16-23	registri salvati
$t8-$t9	24-25	registri temporanei (non salvati)
$k0-$k1	26-27	gestione delle eccezioni
$gp	28	puntatore alla global area (dati)
$sp	29	stack pointer
$s8	30	registro salvato (fp)
$ra	31	indirizzo di ritorno

Uso dei registri: convenzioni

Registri usati per le operazioni floating point

Nome	Utilizzo
$f0-$f3	valori di ritorno di una procedura
$f4-$f11	registri temporanei (non salvati)
$f12-$f15	argomenti di una procedura
$f16-$f19	registri temporanei (non salvati)
$f20-$f31	registri salvati

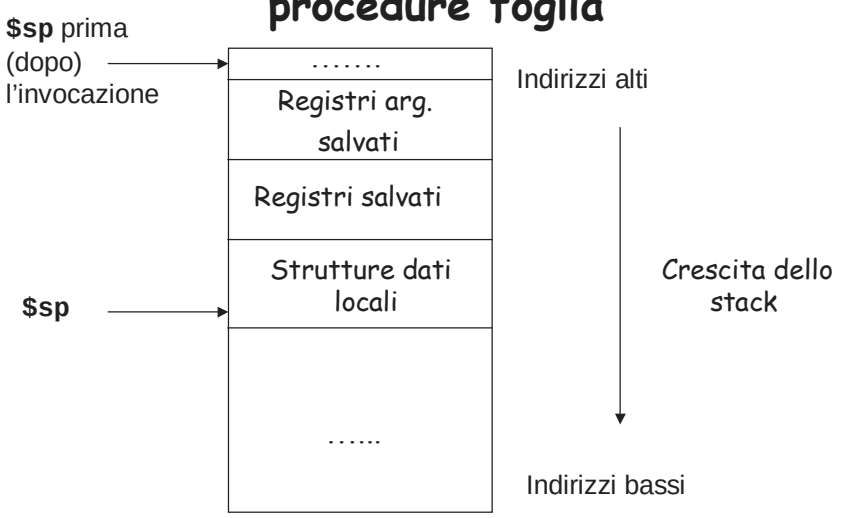
Procedure foglia

- Procedura **foglia** è una procedura che ***non*** ha annidate al suo interno chiamate ad altre procedure
  - **non serve che salvi `$ra` (perché nessuno altro lo modifica)**
- Nel caso di procedure foglia, il **chiamante** salva nello stack:
  - I registri temporanei di cui vuole salvare il contenuto di cui ha bisogno dopo la chiamata (**`$t0-$t9`,...)**
  - I registri argomento (**`$a0-$a3`,...)** nel caso in cui il loro contenuto debba essere preservato.
  - Eventuali argomenti aggiuntivi oltre a quelli che possono essere contenuti nei registri **`$a0-$a3`**.

Procedure foglia

- Nel caso di procedure foglia, il **chiamato** alloca 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.
  - I registri argomento della procedura ($ra, $fp).
- Lo stack pointer **`$sp`** è aggiornato per tener conto del numero di registri memorizzati nello stack; alla fine i registri vengono ripristinati e lo stack pointer riaggiornato.

Record di attivazione: procedure foglia



Record di attivazione

- Una procedura è eseguita in uno spazio *privato* detto record di attivazione
  - 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 (*frame di chiamata a procedura*) 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 come gli elementi di uno stack

Salvataggio dell'ambiente

- L'esecuzione di una procedura non deve interferire con l'ambiente chiamante
- I registri usati dal chiamante devono essere salvati per poter essere ripristinati al rientro dalla procedura
- Esistono delle ***convenzioni*** (regole) per farlo

Convenzioni per il salvataggio dell'ambiente

- Convenzione **del MIPS**
  - per ottimizzare il numero di accessi alla memoria, il chiamante e il chiamato salvano solo i registri di un particolare gruppo
  - il **chiamante**, *se vuole che siano preservati*, salva i registri di **temporanei `$t0-$t9 ($f4-$f11, $f16-$f19)`**, ed i registri **argomento `$a0-$a3`**
  - il **chiamato** salva sullo stack **`$ra` e `$fp`** ed i registri **di variabile** $s0-$s8 ($f20-$f31), *se utilizzati*, ed eventuali argomenti aggiuntivi **e** strutture dati locali (es: array, matrici) e variabili locali.

Struttura di una procedura

- Ogni procedura ha:
  - un prologo
    - Salvataggio dell'ambiente
  - un corpo
    - Esecuzione della procedura vera e propria
  - un epilogo
    - Ripristino dell'ambiente

Struttura di una procedura

- Ogni procedura ha:
  - **un prologo**
    - Acquisire le risorse necessarie per memorizzare i dati interni alla procedura ed il salvataggio dei registri.
    - Salvataggio dei registri di interesse.
  - **un corpo**
    - Esecuzione della procedura vera e propria
  - **un 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.

Storia della chiamata

Main:

- Fare spazio nello stack per memorizzare i registri di eventuale interesse nello stack (registri $t, registri $a) e memorizzarne il contenuto in stack.
- Mettere i parametri della procedura nei registri $a0, $a1, $a2, $a3 ed eventualmente in stack.
- Trasferire il controllo alla procedura (definizione di un nome-etichetta per la procedura, es: **`proc_name:`**)

Procedura:

- Fare spazio nello stack per memorizzare i dati locali.
- Salvataggio dei registri di interesse nello stack (registri $s, $ra, $fp, se vengono utilizzati dalla procedura internamente).

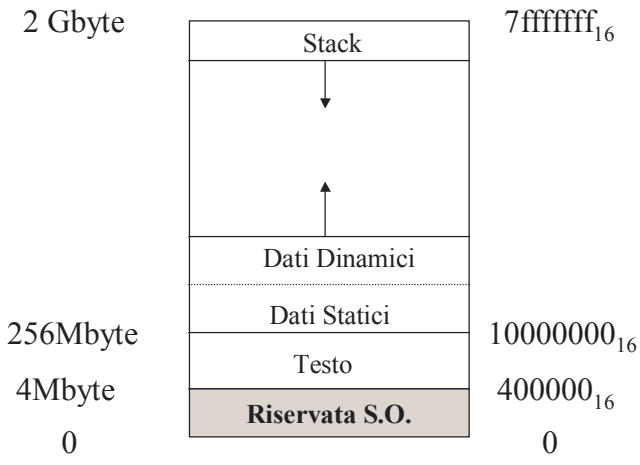
Prologo – record di attivazione

- Determinazione della dimensione del record di attivazione
- Per determinare la dimensione del record di attivazione si deve stimare lo spazio per:

  - registri interi da salvare.
  - registri per argomenti.
  - registri per variabili locali.

NB I registri $t e $a vengono salvati in stack dal chiamante, non fanno parte del record di attivazione.

Organizzazione logica della memoria



Prologo – creazione spazio in stack

- Allocazione dello spazio sullo stack: aggiornare il valore di **$sp:**

```
addi $sp,$sp,-dim_record_attivaz

# lo stack pointer viene decrementato
# della dimensione prevista per la procedura
```

Prologo – salvataggio registri

- Salvataggio dei registri per i quali è stato allocato spazio nello stack:

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

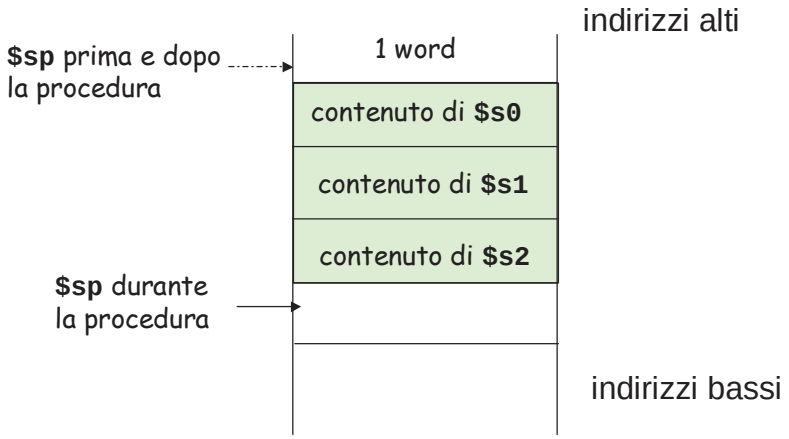
  - **N (N ≥ 4)** viene incrementato di 4 ad ogni salvataggio

Esempio di salvataggio dei registri

- Record di attivazione: 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
```

Esempio di salvataggio dei registri (cont.)



Corpo della procedura

- Stesura delle istruzioni per l'esecuzione delle funzionalità previste dalla procedura

Epilogo

- Ripristino dei registri di interesse dallo stack (i registri $s e $f ed i registri $ra e $fp, se vengono utilizzati dalla procedura internamente).
- Restituzione dei parametri della procedura (dai registri $v0, $v1 e dallo stack).
- Eliminare lo spazio dello stack in cui sono stati memorizzati i dati locali.
- Trasferire il controllo al programma chiamante.

Epilogo della procedura

- Ripristino dei registri salvati:
  `lw reg, dim_record_attivaz - N($sp)`
- Rimozione dello spazio allocato sullo stack:
  `addi $sp,$sp,dim_record_attivaz.`

Cosa manca?

- Registro $ra.
- Restituzione del controllo al chiamante:
  ```
  jr $ra
  ```
- Si ritorna all'istruzione successiva a quella che ha chiamato la procedura.

Procedura chiamante

- Salva i registri $t di cui vuole preservare il contenuto.
- Copia eventuali argomenti in numero superiore a quattro nello stack (oltre a quelli contenuti nei registri **$a0-$a3**)
- Esegue:
  ```
  jal  proc_name
  ```

Esempio 1

```
# Programma che stampa una stringa mediante
# procedura print
        .data
str:    .asciiz "benvenuti in xSPIM\n "
        .text
        .globl main
main:   la $a0, str             # $a0 ← ind. stringa da stampare
        li $v0, 16              # $v0 ← valore 16 da preservare
        jal print
        li $v0, 10              # $v0 ← codice della exit
        syscall                 # esce dal programma

print:  addi $sp, $sp,-4        # allocazione dello stack
        sw $v0, 0($sp)          # salvo $v0 modificato da print
        li $v0, 4               # $v0 ← codice di print_string
        syscall                 # stampa della stringa
        lw $v0, 0($sp)          # ripristina il reg. $v0
        addi $sp,$sp,4          # deallocazione dello stack
        jr $ra
```


Esempio 2

```
# Programma che copia tramite una procedura gli interi positivi
# contenuti nell'array (elem) in un secondo array (pos_el)
```

- Inizializzazione
- Leggi gli interi
- Individua gli interi positivi
- Copia gli interi positivi
- Stampa gli interi positivi

Esempio 2 – Versione in linguaggio C

```
main()
{       int i, k = 0, N=10, elem[10], pos_el[10];

        elem = leggi_interi(N);
        for (i=0; i<N;i++)
                if (elem[i] > 0)
                {       pos_el[k] = elem[i]; k++;
                }
        printf("Il numero di interi positivi e' %d",k);
        for (i=0; i<k; i++)
                printf(" %d",pos_el[i]);
        exit(0);
}
```


Esempio 2 – estrazione di numeri positivi

```
# Programma che copia tramite una procedura gli interi positivi
# contenuti nell'array (elem) in un secondo array (pos_el)
        .data
elem:   .space 40               # alloca 40 byte per array elem
pos_el: .space 40               # alloca 40 byte per array pos_el
prompt: .asciiz "Inserire il successivo elemento \n"
msg:    .asciiz "Gli elementi positivi sono \n"
        .text
        .globl main

main:   la $s0, elem            # $s0 ← indirizzo di elem.
        la $s1, pos_el          # $s1 ← indirizzo di pos_el
        li $s2, 40              # $s2 ← numero di elementi (in byte).

# Ciclo di lettura dei 10 numeri interi
        move $t0, $s0           # indice di ciclo
        add $t1, $t0, $s2       # $t1 ← posizione finale di elem

loop:   li $v0, 4               # $v0 ← codice della print_string
        la $a0, prompt          # $a0 ←  indirizzo della stringa
        syscall                 # stampa la stringa prompt
```

Esempio 2 (cont.)

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

       sw $v0, 0($t0)         # memorizza l'intero in elem
       addi $t0, $t0, 4
       bne $t0, $t1, loop

# Fine della lettura, ora prepara gli argomenti per la proc.
       move $a0, $s0          # $a0 ← ind. array elem.
       move $a1, $s1          # $a1 ← ind. array pos_el
       move $a2, $s2          # $a2 ← dim. in byte dell'array

# Chiamata della procedura cp_pos
       jal cp_pos             # restituisce il numero di byte
                              # occupati dagli interi pos in $v0
```


Esempio 2 (cont.)

```
  # ciclo di stampa degli elementi positivi
  # Gli indirizzi in cui e' racchiuso pos_elem vanno da $s2 a $t2
        add $t2, $s1, $v0      # $t2 ← ind. fine ciclo pos_el
        move $t0, $s1          # $t0 ← ind. di ciclo

  loop_w: beq $t2, $t0, exit_main
        li $v0, 1              # $v0 ← codice della print_integer
        lw $a0, 0($t0)         # $a0 ← intero da stampare
        syscall                # stampa dell'intero
        addi $t0, $t0, 4
        j loop_w

  exit_main:     li $v0, 10     # $v0 ← codice della exit
                 syscall        # esce dal programma
```

Esempio 2 (procedure)

```
# Questa procedura esamina l'array elem ($a0) contenente $a2/4 elementi
# Resituisce in $v0, il numero di byte occupati dagli interi positivi e
# in pos_el gli interi positivi
cp_pos: addi $sp, $sp, -16     # allocazione dello stack
        sw $s0, 0($sp)
        sw $s1, 4($sp)
        sw $s2, 8($sp)
        sw $s3, 12($sp)
        move $s0, $a0          # $s0 ← ind. ciclo su elem
        add $s1, $a0, $a2      # $s1 ← ind. di fine ciclo su elem
        move $s2, $a1          # $s2 ← ind. ciclo su pos_el
next:   beq $s1, $s0, exit     # se esaminato tutti gli elementi di elem
                               #       salta alla fine del ciclo (exit).
        lw $s3, 0($s0)         # carica l'elemento da elem
        addi $s0, $s0, 4       # posiziona sull'elemento succ. di elem
        ble $s3, $zero, next   # se $s3 ≤ 0 va all'elemento succ. di elem
        sw $s3, 0($s2)         # Memorizzo il numero in pos_elem
        addi $s2, $s2, 4       # posiziona sull'elemento succ. di pos_el
        j next                 # esamina l'elemento successivo di elem
exit:   sub $v0, $s2, $a1      # salvo lo spazio in byte dei pos_el
        lw $s0, 0($sp)
        lw $s1, 4($sp)
        lw $s2, 8($sp)
        lw $s3, 12($sp)
        addi $sp,$sp,12        # deallocazione dello stack
        jr $ra                 # restituisce il controllo al chiamante
```


Esercizi

- Estrazione di numeri pari da un vettore di N numeri interi.
- Scrivere un algoritmo di bubble-sort o quick-sort.
- Ricerca di una stringa (e.g. "nome") all'interno di un testo.