


Il Linguaggio Assembly: Controllo del flusso e procedure

Prof. Alberto Borghese
Dipartimento di Scienze dell'Informazione
borghese@dsi.unimi.it

Università degli Studi di Milano


Sommario

Istruzioni MIPS di controllo di flusso.

Le procedure

Lo stack

Le strutture di controllo

Strutture di controllo:

- cicli (for i = 0; i < n; i++)
- condizioni (if …. then …. else ….)
- salto (goto)

- Queste istruzioni (branch & jump):
  - Alterano l'ordine sequenziale di esecuzione delle istruzioni:
    - La prossima istruzione da eseguire non è l'istruzione successiva all'istruzione corrente
  - Permettono di eseguire cicli e condizioni

- In assembly le strutture di controllo sono molto semplici e primitive

Istruzioni di salto condizionato

- Istruzioni di salto: viene caricato un nuovo indirizzo nel contatore di programma (PC) invece dell'indirizzo seguente l'indirizzo di salto secondo l'ordine sequenziale delle istruzioni.

- Istruzioni di ***salto condizionato (conditional branch)*****:** il salto viene eseguito solo se una certa condizione risulta soddisfatta.

- Esempi: **beq** (*branch on equal*) e **bne** (*branch on not equal*)

```
beq r1, r2, L1 # go to L1 if (r1 == r2)
bne r1, r2, L1 # go to L1 if (r1 != r2)
```

Istruzioni di salto incondizionato

- Istruzioni di salto: viene caricato un nuovo indirizzo nel contatore di programma (PC) invece dell'indirizzo seguente l'indirizzo di salto secondo l'ordine sequenziale delle istruzioni.

- Istruzioni di ***salto incondizionato (unconditional jump):*** il salto viene sempre eseguito.
  Esempi: **`j`** (jump) e **`jr`** (jump register) e **`jal`** (jump and link)

```
j   L1          # go to L1
jr  r31         # go to add. contained in r31
jal L1          # go to L1. Save add. of next
                # instruction in reg. ra (ad
                # esempio return address).
```


Esempio if … then

Codice C:

```
if (i==j)
    f=g+h;
```

- Si suppone che le variabili **f, g, h, i** e **j** siano associate rispettivamente ai registri **`$s0, $s1, $s2, $s3 e $s4`**

La condizione viene trasformata in codice C implementabile in Assembly:

```
if (i != j)
      goto Etichetta;
f=g+h;
Etichetta:
```

Codice MIPS:

```
      bne $s3, $s4, Etichetta    # go to Lab1 if i≠j
      add $s0, $s1, $s2          # f=g+h (skipped if i ≠ j)
Etichetta:
```

Esempio if… then … else

Codice C:

```
if (i==j)
        f=g+h;
  else
        f=g-h;
```

- Si suppone che le variabili **f, g, h, i** e **j** siano associate rispettivamente ai registri **`$s0, $s1, $s2, $s3 e $s4`**

Codice MIPS:

```
      bne $s3, $s4, Else # go to Else if i≠j
      add $s0, $s1, $s2  # f=g+h (skipped if i ≠ j
      j   End            # go to End
Else: sub $s0, $s1, $s2  # f=g-h (skipped if i = j)
End:
```


Esempio: do … while (repeate)

Codice C:

```
do
  g = g + A[i];
  i = i + j;
while (i != h)
```

- Si suppone che **g** e **h** siano associate a **`$s1`** e **`$s2,`** i e **j** associate a **`$s3`** e **`$s4`** e che **`$s5`** contenga il *base address* di **A.**
- Si noti che il corpo del ciclo modifica la variabile **i**
  ⇒ devo moltiplicare **i** per **4** ad ogni iterazione del ciclo per indirizzare il vettore A.


Esempio: do … while

Codice C modificato:

```
        i = 0;
Ciclo:  g = g + A[i];
        i = i + j;
        if (i != h) goto Ciclo;
```

g e **h** -> **$s1** e **$s2**
i e **j** -> **$s3** e **$s4**
A[0] -> **$s5**

Codice MIPS:

```
        add $s3, $zero, $zero
Loop:   muli $t1, $s3, 4      # $t1 ← 4 * i
        add $t1, $t1, $s5     # $t1 ← add. of A[i]
        lw $t0, 0($t1)        # $t0 ← A[i]
        add $s1, $s1, $t0     # g ← g + A[i]
        add $s3, $s3, $s4     # i ← i + j
        bne $s3, $s2, Loop    # go to Loop if i ≠ h
```



Esempio: while

Codice C:

```
while (A[i] == k)          Ciclo:   if (A[i] != k) goto Fine;
     i = i + j;                             i = i + j; goto Ciclo;
                           Fine;
```

Si suppone che **i, j** e **k** siano associate a **$s3, $s4, e $s5** e che **$s6** contenga il *base address* di **A**

Codice MIPS:

```
Loop: muli $t1, $s3, 4        # $t1 ← 4 * i
      add  $t1, $t1, $s6      # $t1 ← add. of A[i]
      lw   $t0, 0($t1)        # $t0 ← A[i]
      bne $t0, $s5, Exit      # go to Exit if A[i]≠k
      add $s3, $s3, $s4       # i ← i + j
      j Loop                  # go to Loop
Exit:
```

Strutture di controllo

- Cosa posso fare se il contenuto di un registro è minore o maggiore del contenuto di un altro?
- MIPS mette a disposizione branch solo nel caso uguale o diverso, non maggiore o minore.
- Spesso è utile condizionare l'esecuzione di una istruzione al fatto che una variabile sia minore di una altra:
  - `slt $s1, $s2, $s3  # set on less than`
    - Assegna il valore **1** a `$s1` se `$s2 < $s3;` altrimenti assegna il valore **0**
- Con `slt, beq` e **bne** si possono implementare tutti i test sui valori di due variabili **(=, !=, <, <=, >,>=)**

Esempio

```
if (i < j) then
   k = i + j;
else
   k = i - j;
```

```
   if (i < j)
        t = 1;
   If (t == 0) goto Else;
   k = i + j;
   goto Exit;
Else:   k = i – j;
Exit:
```

```
#$s0 ed $s1 contengono i e j
#$s2 contiene k

      slt $t0, $s0, $s1
      beq $t0, $zero, Else
      add $s2, $s0, $s1
      j Exit
Else: sub $s2, $s0, $s1
Exit:
```

Condizione di disuguaglianza con pseudo-istruzioni (bgt)

```
if (i < j) then
    k = i + j;
else
    k = i - j;
```

```
    if (i < j)
        t = 1;
    If (t == 0) goto Else;
    k = i + j;
    goto Exit;
Else:   k = i – j;
Exit:
```

```
#$s0 ed $s1 contengono i e j
#$s2 contiene k

      bgt $s0, $s1, Else
      add $s2, $s0, $s1
      j Exit
Else: sub $s2, $s0, $s1
Exit:
```


Struttura switch/case

- Può essere implementata mediante una serie di *if-then-else*
- Alternativa: uso di una *jump address table* cioè di una tabella che contiene una serie di indirizzi di istruzioni alternative (espressività maggiore che in linguaggio ad alto livello)

```
switch(k)
 {
   case 0:    f = i + j; break;
   case 1:    f = g + h; break
   case 2:    f = g - h; break;
   case 3:    f = i - j; break;
  default:    break;
}
```


Struttura switch/case

```
      if (k < 0)
            t = 1;
      else
            t = 0;
      if (t == 1)          // k < 0
            goto Exit;
      if (k == 0)          // k >= 0
            goto L0;
      k--; if (k == 0)     // k = 1;
            goto L1;
      k--; if (k == 0)     // k = 2;
            goto L2;
      k--; if (k == 0)     // k = 3;
            goto L3;
      goto Exit;           // k > 3;
```

```
L0: f = i + j;  goto Exit;
L1: f = g + h; goto Exit;
L2: f = g – h; goto Exit;
L3: f = i - j; goto Exit;

Exit:
```

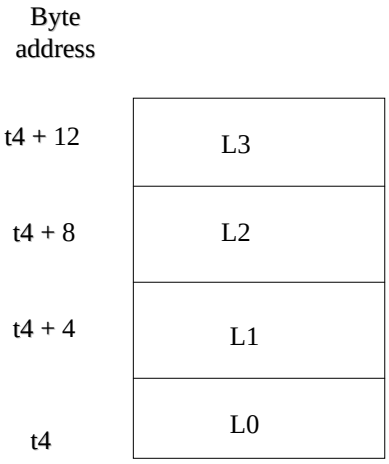


```
#$s0, .., $s5 contengono f,..,k, k variabile di test
#$t2 contiene la costante 4
```

Struttura switch/case

```
      slt $t3, $s5, $zero
      bne $t3, $zero, Exit    # if k<0
                              #case vero e proprio
      beq $s5, $zero, L0
      subi $s5, $s5, 1
      beq $s5, $zero, L1
      subi $s5, $s5, 1
      beq $s5, $zero, L2
      subi $s5, $s5, 1
      beq $s5, $zero, L3
      j Exit;                 # if k>3

L0: add $s0, $s3, $s4
    j Exit
L1: add $s0, $s1, $s2
    j Exit
L2: sub $s0, $s1, $s2
    j Exit
L3: sub $s0, $s3, $s4
Exit:
```

Jump address table




```
#$s0, .., $s5 contengono f,..,k
#$t4 contiene lo start address della jump address table (che si
#            suppone parta da k = 0).

   #verifica prima i limiti (default)
      slt  $t3, $s5, $zero          # if k < 0 exit
      bne  $t3, $zero, Exit
      slti $t3, $s5, 4              # if k >= 4 exit
      beq  $t3, $zero, Exit
   #case vero e proprio
      muli   $t1, $s5, 4            # t1 = k * 4   offset
      add    $t1, $t4, $t1          # t1 += Table_address
      lw     $t0, 0($t1)
      jr     $t0                    # j A[k]
L0: add $s0, $s3, $s4
    j Exit
L1: add $s0, $s1, $s2
    j Exit
L2: sub $s0, $s1, $s2
    j Exit
L3: sub $s0, $s3, $s4
Exit:
```

Struttura switch/case ottimizzata

Esempio

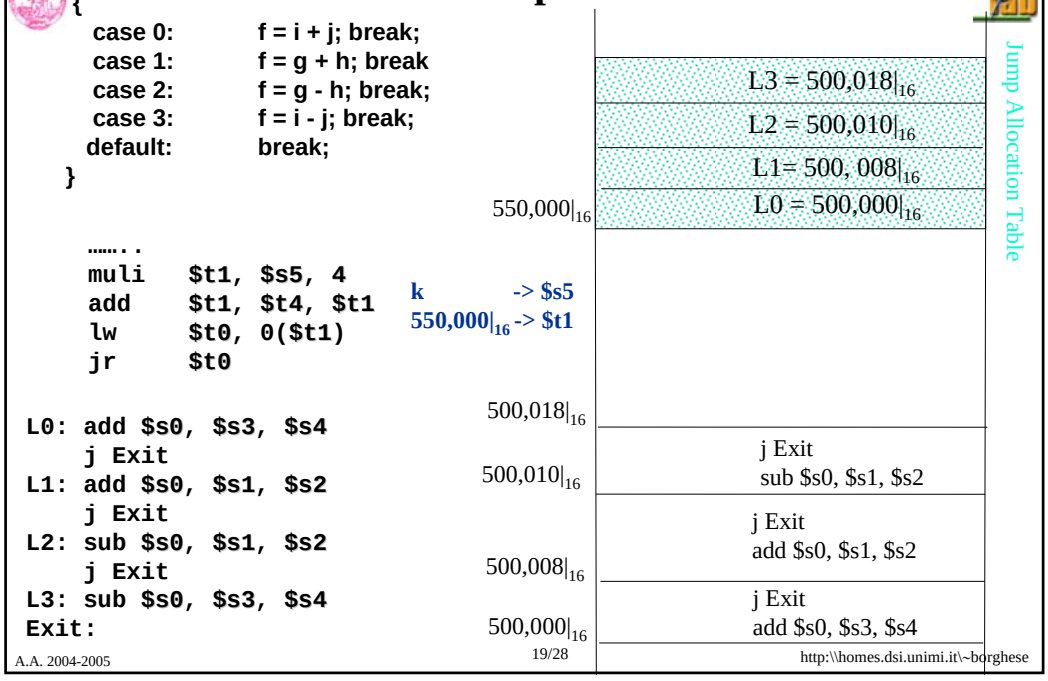
```
switch(k)
{
   case 0:      f = i + j; break;
   case 1:      f = g + h; break
   case 2:      f = g - h; break;
   case 3:      f = i - j; break;
  default:      break;
}
```

```
……..
muli   $t1, $s5, 4
add    $t1, $t4, $t1
lw     $t0, 0($t1)
jr     $t0
```

k -> $s5

$550,000|_{16}$ -> $t1

```
L0: add $s0, $s3, $s4
    j Exit
L1: add $s0, $s1, $s2
    j Exit
L2: sub $s0, $s1, $s2
    j Exit
L3: sub $s0, $s3, $s4
Exit:
```



Sommario

Istruzioni MIPS di controllo di flusso.

Le procedure

Lo stack

Gli attori

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

```
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;
}
```

I due moduli si parlano solamente attraverso i parametri:
-Parametri di input (argomenti della funzione).
-Parametri di output (valori restituiti dalla funzione).

I compiti

- 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
- 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

I registri interessati

- 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

MIPS: Software conventions for Registers

0	zero	constant 0
1	at	reserved for assembler
2	v0	expression evaluation &
3	v1	function results
4	a0	arguments
5	a1	
6	a2	
7	a3	
8	t0	temporary: caller saves
. . .		(callee can clobber)
15	t7	
16	s0	callee saves
. . .		(caller can clobber)
23	s7	
24	t8	temporary (cont'd)
25	t9	
26	k0	reserved for OS kernel
27	k1	
28	gp	Pointer to global area
29	sp	Stack pointer
30	fp	frame pointer (s8)
31	ra	Return Address (HW)

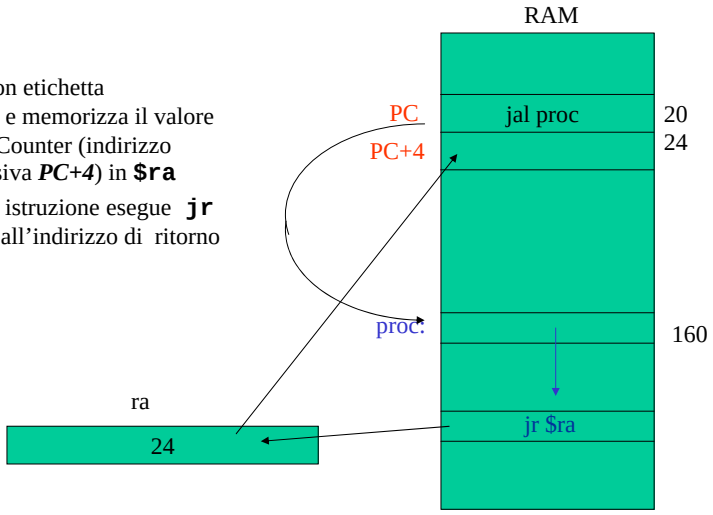
Meccanismo di chiamata

- 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.



La chiamata a procedura

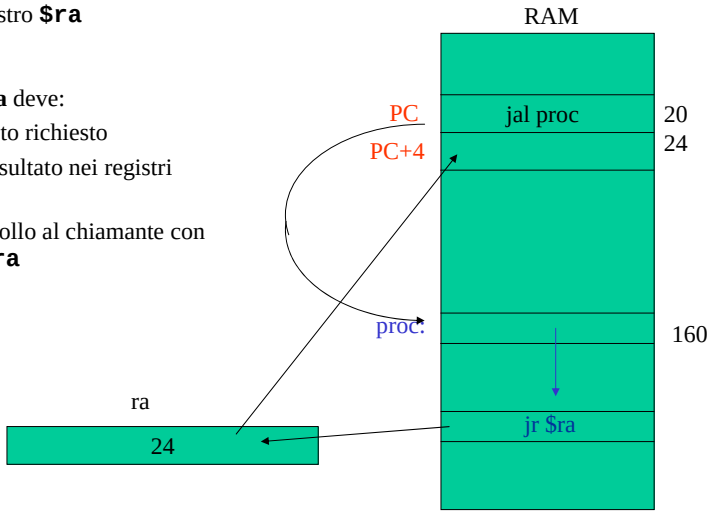
- Il programma **chiamante** deve:
  - Mettere i valori dei parametri da passare alla procedura nei registri **`$a0-$a3`**
  - Utilizzare l'istruzione **`jal address`** per saltare alla procedura e salvare il valore di **`(PC+4)`** nel registro **`$ra`**

•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

- 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

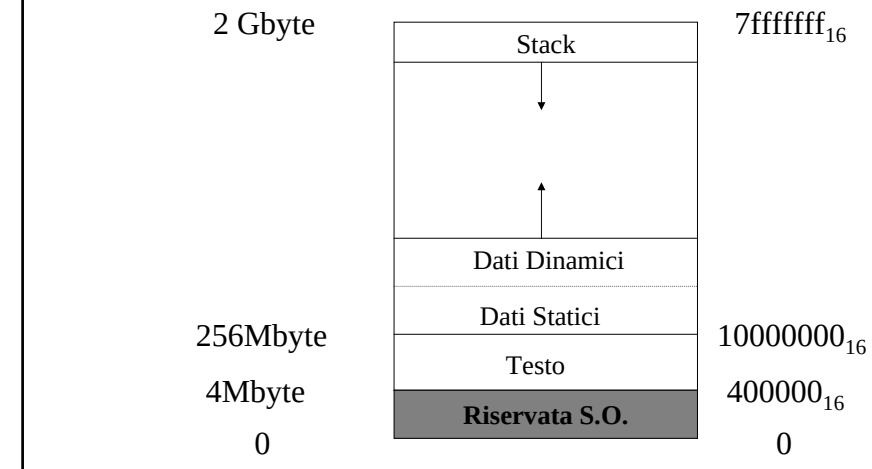
Sommario

Istruzioni MIPS di controllo di flusso.

Le procedure

Lo stack.

Organizzazione logica della memoria

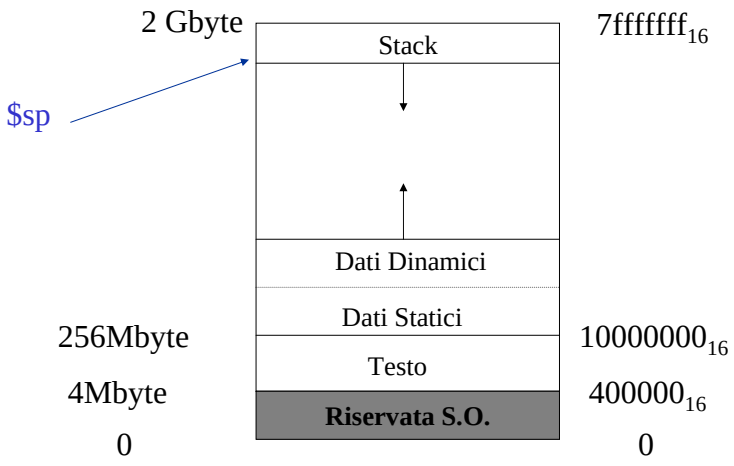


Uso dei registri: registro $sp

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

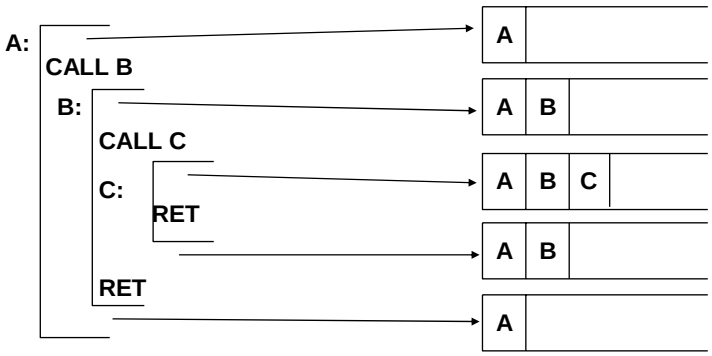
$sp indica l'ultimo indirizzo dell'area di stack.

Organizzazione logica della memoria



Calls: Why Are Stacks So Great?

Stacking of Subroutine Calls & Returns and Environments:



Some machines provide a memory stack as part of the architecture (e.g., VAX)

Sometimes stacks are implemented via software convention (e.g., MIPS)

Gestione dello stack nel MIPS

- Lo stack (pila) è una struttura dati costituita da una coda LIFO (last-in-first-out)
- Lo stack cresce **da indirizzi di memoria alti verso indirizzi bassi**
- Il registro **`$sp`** contiene l'indirizzo dell'ultima locazione utilizzata in cima allo stack.
- L'inserimento di un dato nello stack **(operazione di push)** avviene **decrementando `$sp`** per allocare lo spazio ed eseguendo una sw per inserire il dato.
- Il prelevamento di un dato dallo stack **(operazione di pop)** avviene eseguendo una lw ed **incrementando `$sp`** (per eliminare il dato), riducendo quindi la dimensione dello stack.

- 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 riservato ad una procedura si trova tra i registri $sp e $fp.

Gestione dello stack nel MIPS

- Alla chiamata di 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
```

- 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

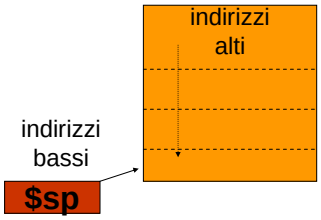
Esempio di gestione dello stack nel MIPS

- Per inserire elementi nello stack

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

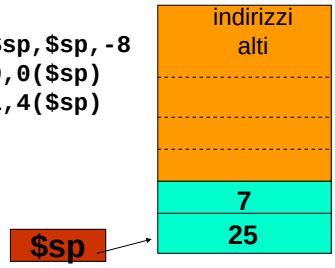
- Per recuperare elementi dallo stack

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




```
addi $sp,$sp,-8
sw $t0,0($sp)
sw $t1,4($sp)
```

$t0 <- 25
$t1 <- 7



Esempio di salvataggio dei registri di variabile

- 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.

Procedura Somma_algebrica

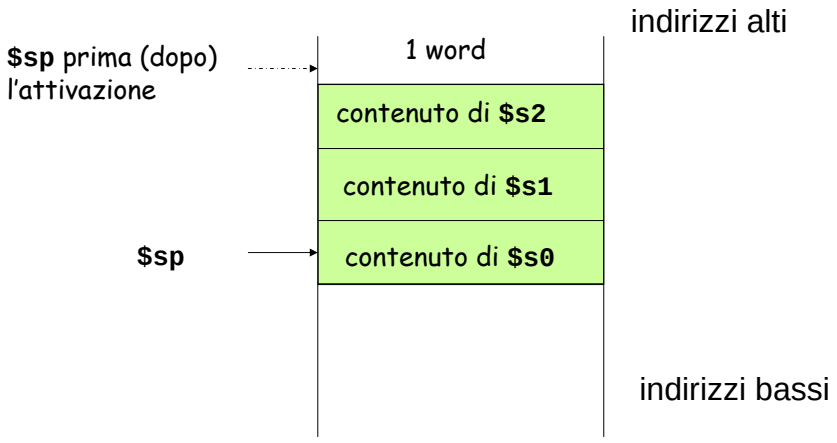
```
int somma_algebrica (int g, int h, int i, int j)
{  int f;
   f = (g + h) - (i + j);
   return f;
}
```

```
# g,h,i e j associati a $a0, …, $a3;
# f associata a $v0
# Supponiamo di utilizzare i 3 registi: $s0, $s1, $s2 nel calcolo ➔è necessario
# salvarne il contenuto (in 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: somma

```
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

  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 estraendolo 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
```


I registri nello stack

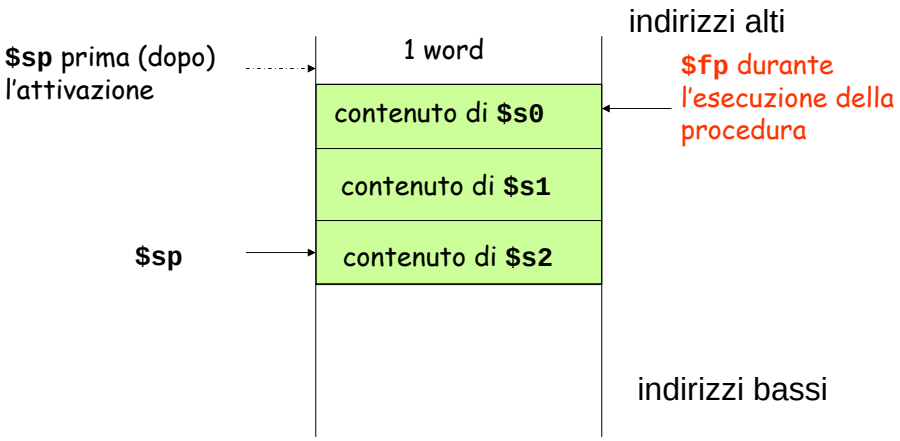


Uso dei registri: registro $fp

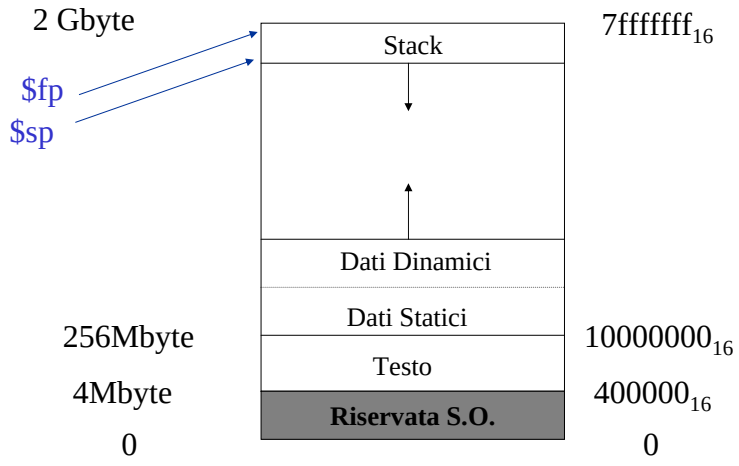
Nome	Numero	Utilizzo
$zero	0	costante zero
$at	1	riservato per l'assemblatore
$v0-$v1	2-3	valori di ritorno di una procedura
$a0-$a3	4-7	argomenti di una procedura
$t0-$t7	8-15	registri temporanei (non salvati)
$s0-$s7	16-23	registri salvati
$t8-$t9	24-25	registri temporanei (non salvati)
$k0-$k1	26-27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

$fp indica l'indirizzo più elevato dello stack in cui sono memorizzati i dati relativi ad una procedura.

Il frame pointer



Organizzazione logica della memoria



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-$t9,...).**
- Nel caso di procedure foglia, il **chiamato** alloca nello stack:
  - I registri non temporanei che vuole utilizzare **`($s0-$s8)`**
  - Strutture dati locali (es: array, matrici) e variabili locali della procedura che non stanno nei registri.

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.

Le stesse operazioni vengono eventualmente eseguite sul $fp.

Sommario

IIstruzioni MIPS di controllo di flusso.

Le procedure

Lo stack