

Assembly MIPS:

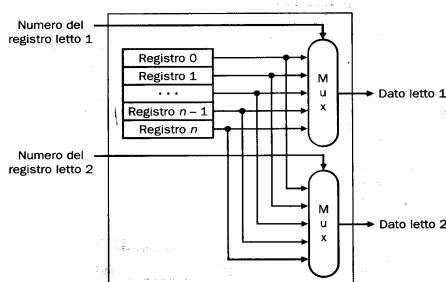
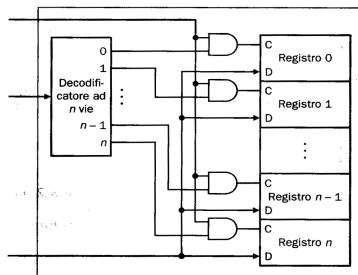
- Istruzioni native e Pseudo-istruzioni
- Gestione di costanti in MIPS

F. Pedersini

Dipartimento di Informatica
Università degli Studi di Milano

Uso dei registri: convenzioni

Il Register File di MIPS32: 32 registri "general purpose"



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



PSEUDO-ISTRUZIONI

Per comodità in ogni Assembly vengono definite alcune pseudoistruzioni

- Significato intuitivo
- Non hanno un corrispondente 1 a 1 con le istruzioni dell'ISA (native)

❖ Vantaggi:

- **Parziale standardizzazione** del linguaggio Assembly
 - ✦ La stessa pseudo-istruzione viene tradotta in modi differenti, per architetture (ISA) differenti
- Rappresentazione **più compatta ed intuitiva** di istruzioni Assembly comunemente utilizzate
 - ✦ La traduzione della pseudo-istruzione nelle istruzioni equivalenti viene attuata automaticamente dall' Assembler

Esempi:



Pseudoistruzione:

Codice MIPS nativo:

<pre>move \$t0, \$t1 # t0 ← t1</pre>	→	<pre>add \$t0, \$zero, \$t1</pre>
<pre>mul \$s0, \$t1, \$t2 # s0 ← t1*t2</pre>	→	<pre>mult \$t1, \$t2 mflo \$s0</pre>
<pre>div \$s0, \$t1, \$t2 # s0 ← t1/t2</pre>	→	<pre>div \$t1, \$t2 mflo \$s0</pre>
<pre>muli \$s0, \$t1, 4 # s0 ← t1*4</pre>	→	<pre>addi \$at, \$zero, 4 mult \$t1, \$at mflo \$s0</pre>
<pre>lw \$t2, \$t3(\$a0) # t2 ← Mem[\$a0+\$t3]</pre>	→	<pre>add \$at, \$a0, \$t3 lw \$t2, 0(\$at)</pre>
<pre>bgti \$t2, 5, Label # branch if \$t2 > 5 # → if not(\$t2 < 6)</pre>	→	<pre>slti \$at, \$t2, 6 beq \$at, \$zero, Label</pre>



Utilizzo di costanti in Assembly MIPS:

Spesso in un programma si ricorre all'uso di costanti:

- incremento di registri (± 1 , $+4$, ...), azzeramento di un registro, ...

In Assembly MIPS sono disponibili 3 opzioni:

- le costanti risiedono **in memoria** e vengono caricate nel registro con **lw**
- utilizzo di **registri speciali** (es: \$zero)
- utilizzare istruzioni di **tipo I**, in cui un operando (l'immediato) è una costante: utilizzo di modalità di **indirizzamento immediato**

Esempi:

```
add $s0, $zero, $zero # $s0 ← 0 (azzeramento registro)
addi $s0, $s0, 4      # $s0 ← $s0 + 4 (sign-extended)
addi $s0, $s0, -4     # $s0 ← $s0 - 4 (sign-extended)
slti $t0, $s2, 10     # $t0 ← 1 if $s2 < 10
```

Gestione di costanti in MIPS



- ❖ Le istruzioni di **tipo I** consentono di rappresentare costanti espresse **su 16 bit**:

	Unsigned:	Signed:
range di rappresentazione:	0 ÷ 65535₁₀	-32768₁₀ ÷ 32767₁₀
in esadecimale:	0x0000 ÷ 0xFFFF	

- ❖ Se 16 bit non sono sufficienti per rappresentare la costante, è necessario fare due passi:

1. si utilizza l'istruzione **lui** (*load upper immediate*):

- carica i 16 bit ALTI della costante nei **16-bit ALTI** di un **registro**.
- I **16-bit** meno significativi (**BASSI**) del registro sono posti a **zero**.

- una successiva istruzione (ad esempio **ori** o **addi**) specifica i rimanenti **16 bit meno significativi** della costante.

- ❖ Il registro **\$at** (**=\$1**) è riservato all'assemblatore per creare costanti su 32 bit.



Istruzione: **load upper immediate** **lui rt, immval**

- carica i 16 bit del campo **immediato** nei 16 bit più significativi del registro **rt**.
- I rimanenti **16 bit meno significativi** del registro **rt** sono posti a 0.

Nome campo	op	rs	rt	indirizzo			
Dimensione	6-bit	5-bit	5-bit	16-bit			
lui \$at, 61	001111	00000	00001	0000	0000	0011	1101

Usata per la pseudo-istruzione **load immediate (li)**:

li rdest, immval

- carica il valore **imm** nel registro **rdest**.

Assegnamento di costanti a 32 bit



ESEMPIO: si vuole assegnare la costante $4.000.000_{10}$ a **\$s0**:

Pseudo-istruzione: **li \$s0, 4000000** # carica 4000000 in \$s0

In binario, su 32 bit:

4000000₁₀ → 0000 0000 0011 1101 0000 1001 0000 000₂

In esadecimale:

4000000₁₀ → **0x 003D 0900**

In decimale:

4000000₁₀ = **61 * 2¹⁶ + 2304**

Quindi (in decimale):

lui \$s0, 61 # 61 = 0000 0000 0011 1101

valore di **\$s0**: 0000 0000 0011 1101 0000 0000 0000 0000

addiu \$s0, \$s0, 2304 # 2304 = 0000 1001 0000 0000

valore di **\$s0**: 0000 0000 0011 1101 0000 1001 0000 0000 (4000000₁₀)

oppure in esadecimale:

lui \$s0, 0x3D # 0x3D = 0000 0000 0011 1101

addiu \$s0, \$s0, 0x900 # 0x900 = 0000 1001 0000 0000



ESEMPIO: si consideri la costante: 118345_{10}

- decimale: **118345** = $65536 + 52809 = 1 \times 2^{16} + 52809$
- esadecimale: **0x1CE49**
- binario: **0000 0000 0000 0001 1100 1110 0100 1001**
 16-bit più significativi 16-bit meno significativi

HI: 0000 0000 0000 0001 corrispondenti al valore 1_{10}

LO: 1100 1110 0100 1001 corrispondenti al valore 52809_{10}

↙ **Pseudoistruzione: li, load immediate**

Quindi: **li \$t1, 118345**

- L'assemblatore la sostituisce con le seguenti istruzioni in linguaggio macchina:

----- in notazione decimale -----

lui \$at, 1

ori \$t1, \$at, 52809

----- oppure (esadecimale) -----

lui \$at, 1 # \$at: 0000 0000 0000 000**1** 0000 0000 0000 0000

ori \$t1, \$at, 0xCE49 # \$t1: 0000 0000 0000 000**1 1100 1110 0100 1001**

Esercizi di riepilogo



Tradurre il seguente frammento di codice: a) in Assembly MIPS nativo e b) in linguaggio macchina (specificando ampiezza in bit e valore dei campi delle istruzioni).

bgei \$t1, 10, -16 # branch if greater than or equal to 10

divi \$s2, \$s0, 8⁸ # divide by immediate

Si traduca il seguente frammento di codice Assembly MIPS in linguaggio macchina, in formato esadecimale, calcolando prima i valori esadecimali **N1** e **N2** che permettono di saltare esattamente all'indirizzo indicato in ciascun commento.

0x12345678: **beq \$s0, \$s4, N1** # salta a: 0x12345660

j N2 # salta a: 0x10203040



Assembly MIPS:

- Gestione degli array
- Strutture di controllo

Istruzione MIPS: **lw** (load word)



In **MIPS** le istruzioni **lw/sw** hanno tre argomenti:

- **registro destinazione**, in cui caricare (da cui prelevare) la parola letta da (da scrivere in) memoria.
 - una costante o **spiazzamento (offset)**
 - registro base (**base register**), che contiene il valore dell'indirizzo base (**base address**) da sommare alla costante.
- ❖ L'indirizzo della parola di memoria da caricare nel registro destinazione è ottenuto come somma della costante e del contenuto del registro base.
- ❖ L'indirizzo è espresso in byte.

Esempio:

```
lw $s1, 100($s2) # $s1 ← M[$s2+100]
```

Al registro destinazione **\$s1** è assegnato il valore contenuto all'indirizzo: **\$s2 + 100** della memoria dati.

```
sw $s1, 100($s2) # M[$s2 + 100] ← $s1
```

Alla locazione di memoria di indirizzo: **\$s2 + 100** è assegnato il valore contenuto nel registro **\$s1**



Gestione di array di parole (32 bit / 4 byte):

L'elemento i -esimo $A[i]$ si trova all'indirizzo di memoria: $br + 4*i$

br è il registro base; i è l'indice ad alto livello

- ❖ Il registro base punta all'inizio dell'array: primo byte del primo elemento
- ❖ Spiazzamento: $4i$ per selezionare l'elemento i -esimo dell'array

Elem. array	parola (word)				Indirizzo	Cella
$A[0] :$	0	1	2	3	$\$s3 \rightarrow$	$A[0]$
$A[1] :$	4	5	6	7	$\$s3 + 4 \rightarrow$	$A[1]$
$A[2] :$	8	9	10	11	$\$s3 + 8 \rightarrow$	$A[2]$
....						
$A[k-2] :$...
$A[k-1] :$	2^{k-4}	2^{k-3}	2^{k-2}	2^{k-1}	$\$s3 + 4i \rightarrow$	$A[i]$
						...

Array: accesso con indice costante



Esempio: accesso ad array con indice costante

Codice C: $A[12] = h + A[8];$

Si suppone che:

- la variabile h sia associata al registro $\$s2$
- l'indirizzo del primo elemento dell'array (*base address*) sia contenuto nel registro $\$s3$ ($A[0]$)

Codice MIPS:

```
lw  $t0, 32($s3)      # $t0 ← M[$s3 + 32]
add $t0, $s2, $t0     # $t0 ← $s2 + $t0
sw  $t0, 48($s3)     # M[$s3 + 48] ← $t0
```



Esempio: accesso ad array con indice variabile

Codice C: $g = h + A[i]$

❖ Si supponga che:

- le variabili g , h , i siano associate rispettivamente ai registri:

$\$s1$, $\$s2$, ed $\$s4$

- l'indirizzo del primo elemento dell'array (*base address*) sia contenuto nel registro $\$s3$

g	$\$s1$
h	$\$s2$
i	$\$s4$
Array A[: base address	$\$s3$

Array: esempio *indirizzamento*



...segue: accesso ad array con indice variabile

L'elemento **i-esimo** dell'array, $A[i]$ si trova nella locazione di memoria di indirizzo: $(\$s3 + 4*i)$

❖ Caricamento dell'indirizzo di $A[i]$ nel registro temporaneo $\$t1$:

```
mul $t1, $s4, 4      # $t1 ← 4 * i
add $t1, $t1, $s3    # $t1 ← address of A[i]
                    # ora $t1 contiene: ($s3 + 4*i)
```

❖ Per trasferire $A[i]$ nel registro temporaneo $\$t0$:

```
lw $t0, 0($t1)      # $t0 ← A[i]
```

❖ Per sommare h e $A[i]$ e mettere il risultato in g :

```
add $s1, $s2, $t0   # g = h + A[i]
```



Strutture di controllo

- ❖ Alterano l'ordine di esecuzione delle istruzioni
 - La prossima istruzione da eseguire non è l'istruzione successiva all'istruzione corrente
- ❖ Permettono di eseguire **cicli** e valutare **condizioni**
 - In assembly le strutture di controllo sono molto semplici e primitive

Costrutti:

- ❖ **if ... then ... else**
- ❖ **do ... while, while**
- ❖ **for**
- ❖ **switch**

Istruzioni di salto condizionato e incondizionato



Che cosa abbiamo a disposizione nel Set Istruzioni MIPS32 ?

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

- **beq** (*branch on equal*)
- **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 (jump)**: il salto va sempre eseguito.

- **j** (*jump*)
- **jr** (*jump register*)
- **jal** (*jump and link*)

```
j L1                     # go to address L1
jr $reg                   # go to address contained in $reg
jal L1                    # go to L1 but, before, save address of
                          # next instruction in $ra
```



Codice C:

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

- ❖ Si suppongano le variabili *f, g, h, i* e *j* associate rispettivamente ai registri: *\$s0, \$s1, \$s2, \$s3* e *\$s4*
- ❖ La condizione viene trasformata in **codice C** in:

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

Codice MIPS:

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



Codice C:

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

- 7

- Si suppongano le variabili *f, g, h, i* e *j* associate rispettivamente ai registri *\$s0, \$s1, \$s2, \$s3* e *\$s4*

<i>f</i>	<i>\$s0</i>
<i>g</i>	<i>\$s1</i>
<i>h</i>	<i>\$s2</i>
<i>i</i>	<i>\$s3</i>
<i>j</i>	<i>\$s4</i>

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
End:
    ...                # (skipped if i = j)
```



Costrutto: `do ... while ()`

Codice C:

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

- ❖ `g` e `h` siano associate a `$s1` e `$s2`,
`i` e `j` associate a `$s3` e `$s4`,
`$s5` contenga il *base address* di `A`

<code>g</code>	<code>\$s1</code>
<code>h</code>	<code>\$s2</code>
<code>i</code>	<code>\$s3</code>
<code>j</code>	<code>\$s4</code>
<i>base address di A[]</i>	<code>\$s5</code>

- ❖ `A[i]`: array con indice variabile
⇒ devo **moltiplicare `i` per 4** ad ogni iterazione
del ciclo, per indirizzare il vettore `A`.



Codice C modificato (sfruttando `goto`):

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

<code>g</code>	<code>\$s1</code>
<code>h</code>	<code>\$s2</code>
<code>i</code>	<code>\$s3</code>
<code>j</code>	<code>\$s4</code>
<i>base address A[]</i>	<code>\$s5</code>

Codice MIPS:

```
Ciclo:  muli $t1, $s3, 4      # $t1 ← 4 * i
        add $t1, $t1, $s5   # $t1 ← address 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, Ciclo # goto Ciclo if i≠h
```



Costrutto: `WHILE() { ... }`

Codice C:

```
while (A[i] == k) {
    i = i + j;
}
```

Codice C modificato:

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

Fine:  ...
```

i, j, k associate a: `$s3, $s4, $s5`
`$s6` contenga il *base address* di `A[]`



<i>i</i>	<code>\$s3</code>
<i>j</i>	<code>\$s4</code>
<i>k</i>	<code>\$s5</code>
<i>base address A[]</i>	<code>\$s6</code>



Codice C modificato:

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

Fine:
```

Associazioni
 variabili → registri:

i, j, k → `$s3, $s4, $s5`
`A[0]` → `$s6`

Codice MIPS:

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

Fine:  ...
```




Condizioni di *disuguaglianza*

- ❖ MIPS32: **branch** a disposizione solo per condizioni di 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
- ❖ Istruzione **slt**:
slt \$s1, \$s2, \$s3 # set if less than
 - Assegna il valore \$s1 = 1 se \$s2 < \$s3;
altrimenti assegna \$s1 = 0
- ❖ Con **slt**, **beq** e **bne** si possono implementare tutti i test sui valori di due variabili: (=, ≠, <, ≤, >, ≥)

Esempio



Codice C:

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

Codice C modificato:

```
if (i < j)    t = 1;
else        t = 0;

if (t == 0)
    goto Else;
k = i + j;
goto Exit;
Else:
    k = i - j;
Exit:
```

Codice MIPS32:

#\$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:
    ...
```

**Ciclo FOR:****Codice C:**

```

int i;
int sum = 0;
for(i=0; i<10; i++) {
    sum = sum + A[i];
}

```

- ❖ **sum** associata a **\$s1**,
i associata a **\$t0**,
\$s5 contenga il *base address* di **A**



i	\$t0
sum	\$s1
<i>base address A[]</i>	\$s5

- ❖ **A[i]**: array con indice variabile
⇒ devo **moltiplicare i per 4** ad ogni iterazione del ciclo per indirizzare il vettore **A**.

Struttura: do ... while (repeat)**Codice C modificato** (sfruttando **if/goto**):

```

sum = 0; i = 0;
FOR:   if !(i < 10) goto END;
        sum = sum + A[i];
        i = i+1;
        goto FOR;
END:   ...

```

Codice MIPS:

```

add $s1, $zero, $zero      # sum ← 0
add $t0, $zero, $zero      # i ← 0
add $t1, $zero, $s5        # $t1 ← address of A[0]
FOR:   slti $at, $t0, 10     # if i<10 → $at ← 1
        beq $at, $zero, END  # if $at==0 → goto END
        lw  $t2, 0($t1)      # $t2 ← A[i]
        add $s1, $s1, $t2    # sum ← sum + A[i]
        addi $t1, $t1, 4     # addr. A[i] ← addr. A[i+1]
        addi $t0, $t0, 1     # i ← i + 1
        j  FOR
END:   ...

```



Codice C:

```
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;
}
```

Implementabile in due modi:

1. Mediante una serie di:

if-then-else

2. Mediante una *jump address table*

- Tabella che contiene una serie di indirizzi a istruzioni alternative



❖ Codice C alternativo:

```
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,g,h,i,j,k
  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: ...
    
```

f	\$s0
g	\$s1
h	\$s2
i	\$s3
j	\$s4
k	\$s5



Metodo alternativo:

Jump address table

utilizzo il valore della variabile di switch **k** per calcolare l'indirizzo di salto:

k	Byte address: $\$t4 + 4k$	A[k]
0	$\$t4 \rightarrow$	indirizzo di L0 :
1	$\$t4 + 4 \rightarrow$	indirizzo di L1 :
2	$\$t4 + 8 \rightarrow$	indirizzo di L2 :
3	$\$t4 + 12 \rightarrow$	indirizzo di L3 :

A[k]

indirizzo L0 :
indirizzo L1 :
indirizzo L2 :
indirizzo L3 :

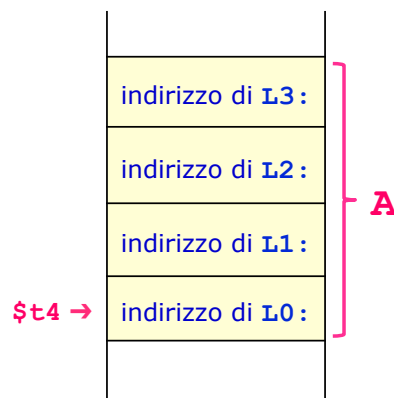


```

# $s0, ..., $s5 contengono f,g,h,i,j,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
bne $t3, $zero, Exit
slti $t3, $s5, 4
beq $t3, $zero, Exit
# case vero e proprio
muli $t1, $s5, 4
add $t1, $t4, $t1
lw $t0, 0($t1)
jr $t0 # jump to 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: ...
    
```

Memoria dati



Assembly MIPS:

- L'ambiente di simulazione
- Direttive di Assembler
- System Calls



Cosa simulano SPIM/MARS?

MARS: MIPS Assembler and Runtime Simulator

- Una **CPU MIPS 32**
- La sua **memoria**
- Un dispositivo minimale di **I/O** (terminale alfanumerico)
- **Sistema Operativo** minimale (per gestione di I/O e memoria)

Abbiamo bisogno di strumenti per:

- Interfaccia di programmazione (load/save, debugging, monitoring)
- Accesso alle risorse (MEM, I/O) attraverso il sistema operativo

Strumenti:

- ❖ **Ambiente di simulazione (memoria, Assembler, ...)**
- ❖ **Direttive**
- ❖ **System calls**

MARS: MIPS Assembler and Runtime Simulator



**MARS:
console**

The screenshot displays the MARS application window. The assembly code in the main editor is as follows:

```

1 # Compute first twelve Fibonacci numbers and put in array, then print
2 .data
3 fibs: .word 0 : 12      # "array" of 12 words to contain fib values
4 size: .word 12         # size of "array"
5 .text
6 la $t0, fibs           # load address of array
7 la $t5, size           # load address of size variable
8 lw $t5, 0($t5)         # load array size
9 li $t2, 1              # 1 is first and second Fib. number
10 add.d $f0, $f2, $f4
11 sw $t2, 0($t0)         # F[0] = 1
12 sw $t2, 4($t0)         # F[1] = F[0] = 1
13 addi $t1, $t5, -2      # Counter for loop, will execute (size-2) times
14loop: lw $t3, 0($t0)    # Get value from array F[n]
15      lw $t4, 4($t0)    # Get value from array F[n+1]
16      add $t2, $t3, $t4  # $t2 = F[n] + F[n+1]
17      sw $t2, 8($t0)    # Store F[n+2] = F[n] + F[n+1] in array
18      addi $t0, $t0, 4   # increment address of Fib. number source
19      addi $t1, $t1, -1  # decrement loop counter
20      bgtz $t1, loop    # repeat if not finished yet.
21      la $a0, fibs      # first argument for print (array)
22      add $a1, $zero, $t5 # second argument for print (size)
23      jal print         # call print routine.
    
```

The registers window on the right shows the following state:

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x00000000
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$t0	8	0x00000000
\$t1	9	0x00000000
\$t2	10	0x00000000
\$t3	11	0x00000000
\$t4	12	0x00000000
\$t5	13	0x00000000
\$t6	14	0x00000000
\$t7	15	0x00000000
\$s0	16	0x00000000
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t8	24	0x00000000
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x10008000
\$sp	29	0x7ffffcfc
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000



Organizzazione memoria MIPS32

La memoria associata un programma è divisa in **tre parti**:

Segmento testo: contiene le **istruzioni del programma**

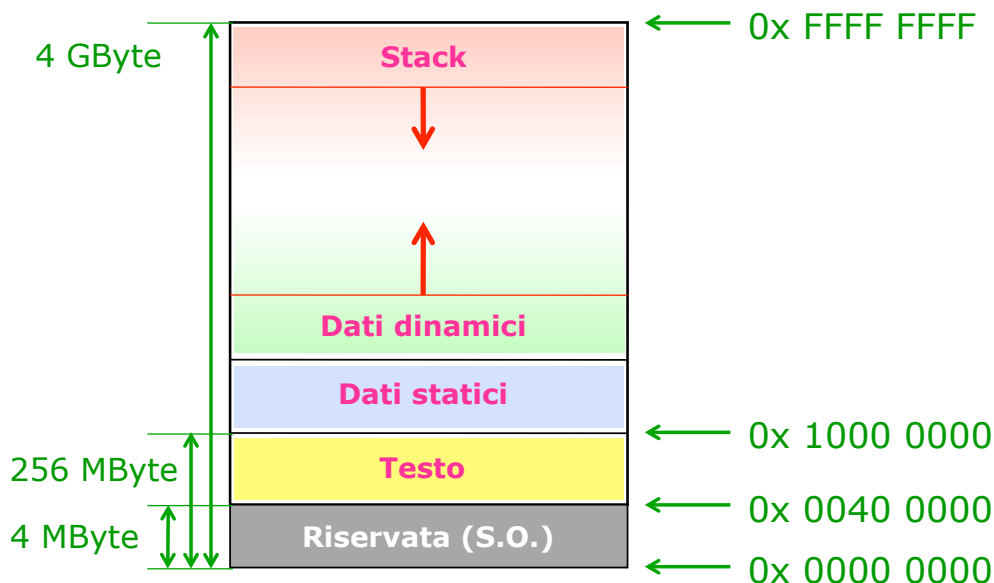
Segmento dati: contiene i **dati di elaborazione**. Ulteriormente suddiviso in:

- **dati statici**: contiene dati già presenti in memoria (o almeno lo spazio che occupano lo è) al momento della compilazione, e che ci rimangono per tutto il tempo di esecuzione del programma.
- **dati dinamici**: contiene dati ai quali lo spazio è **allocato dinamicamente** al momento dell'esecuzione del programma su richiesta del programma stesso.

Segmento stack: contiene lo **stack** allocato automaticamente da un programma durante l'esecuzione.



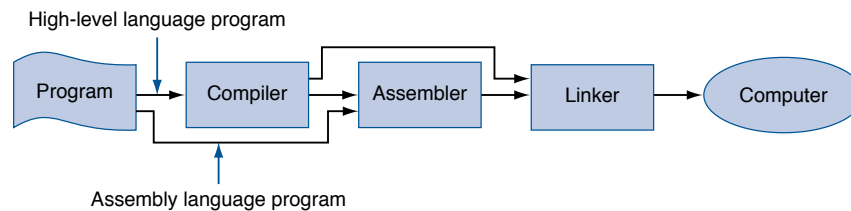
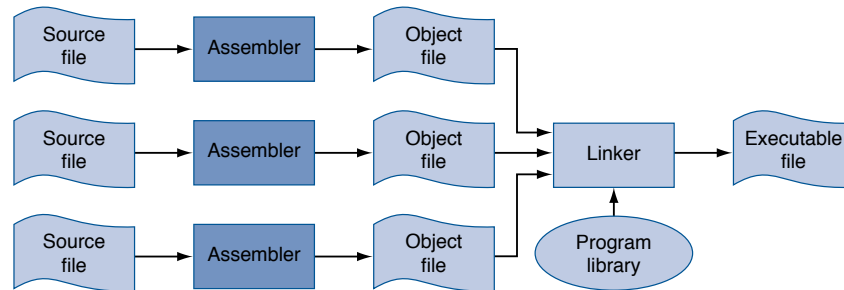
Organizzazione della Memoria di MIPS32





Assembler: traduttore da Assembly a Object (Assembly nativo)

- ❖ Pseudoistruzioni → istruzioni Assembly native
- ❖ Simboli (labels) → indirizzi effettivi (eventualmente rilocabili)



Assembly:

```

.text
.align 2
.globl main
main:
    subu    $sp, $sp, 32
    sw     $ra, 20($sp)
    sd     $a0, 32($sp)
    sw     $0, 24($sp)
    sw     $0, 28($sp)
loop:
    lw     $t6, 28($sp)
    mul   $t7, $t6, $t6
    lw     $t8, 24($sp)
    addu  $t9, $t8, $t7
    sw     $t9, 24($sp)
    addu  $t0, $t6, 1
    sw     $t0, 28($sp)
    ble   $t0, 100, loop
    la    $a0, str
    lw    $a1, 24($sp)
    jal  printf
    move $v0, $0
    lw   $ra, 20($sp)
    addu $sp, $sp, 32
    jr   $ra

.data
.align 0
str:
.asciiz "The sum from 0 .. 100 is %d\n"
    
```

ASSEMBLER

Assembly nativo:

```

addiu    $29, $29, -32
sw       $31, 20($29)
sw       $4, 32($29)
sw       $5, 36($29)
sw       $0, 24($29)
sw       $0, 28($29)
lw       $14, 28($29)
lw       $24, 24($29)
multu   $14, $14
addiu   $8, $14, 1
slti   $1, $8, 101
sw     $8, 28($29)
mflo   $15
addu   $25, $24, $15
bne   $1, $0, -9
sw     $25, 24($29)
lui   $4, 4096
lw     $5, 24($29)
jal   1048812
addiu  $4, $4, 1072
lw     $31, 20($29)
addiu  $29, $29, 32
jr     $31
move   $2, $0
    
```




- ❖ **Direttive**
- ❖ Chiamate di sistema (**system call**)
- ❖ Esempi

Rif. bibliografico:

Patterson, Hennessy

Appendix A – Assemblers, Linkers, and the SPIM Simulator

(sulla pagina Web del corso: [MIPS Instruction Set Reference](#))

Direttive MIPS



Le direttive danno alcune indicazioni all'assemblatore:

- su come/dove memorizzare il programma e i dati in memoria

Sintassi: le direttive iniziano con il carattere "."

Classificazione:

- direttive relative al **segmento dati**
- direttive relative al **segmento testo**

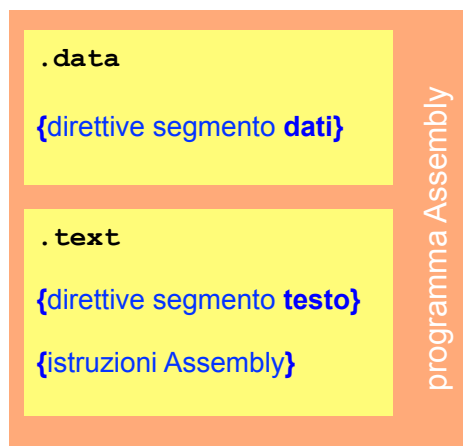
Struttura programma Assembly:

1. segmento dati:

- inizia con la direttiva: **.data**
- contiene dichiarazioni di variabili e costanti

2. segmento testo:

- inizia con la direttiva: **.text**
- contiene il programma





Direttive – segmento dati:

- Definiscono regole di allocazione di dati in memoria

`.data <addr>`

- Gli elementi successivi sono memorizzati nel segmento dati a partire dall'indirizzo `addr` (facoltativo)

`.ascii str`

- Memorizza la stringa `str` terminandola con il carattere **Null (0x00)**
- `.ascii str` ha lo stesso effetto, ma non aggiunge alla fine il carattere **Null**

`.byte b1, ..., bn`

- Memorizza gli `n` valori `b1, ..., bn` in **bytes consecutivi** di memoria

`.word w1, ..., wn`

- Memorizza gli `n` valori su 32-bit `w1, ..., wn` in parole (**words**) **consecutive** di memoria

`.half h1, ..., hn`

- Memorizza gli `n` valori su 16-bit `h1, ..., hn` in **halfword (mezze parole) consecutive** di memoria

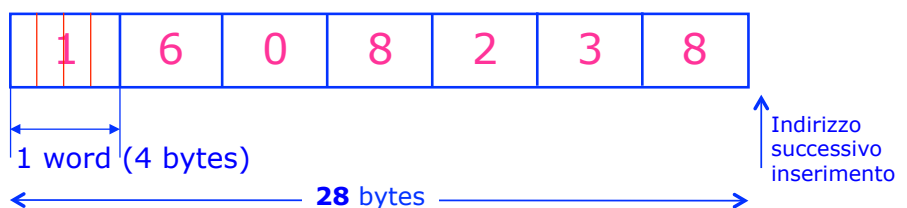
`.space n`

- Alloca uno **spazio** pari ad `n bytes` nel segmento dati

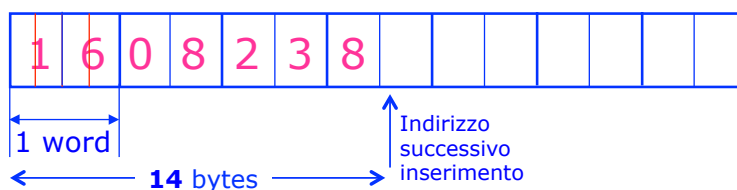
Allineamento: esempio



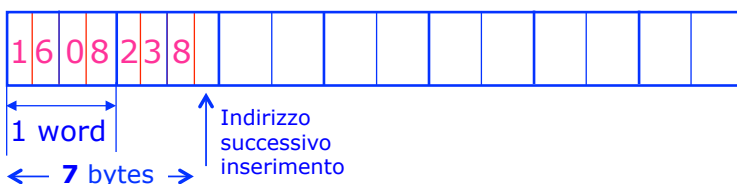
`.word 1, 6, 0, 8, 2, 3, 8`



`.half 1, 6, 0, 8, 2, 3, 8`

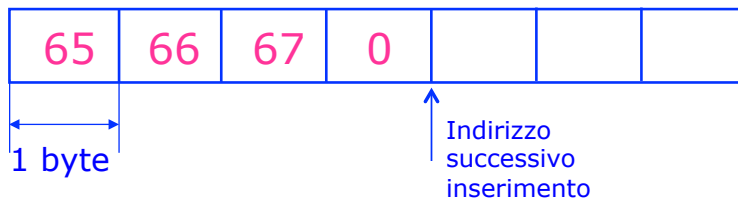


`.byte 1, 6, 0, 8, 2, 3, 8`





`.ascii "ABC"`

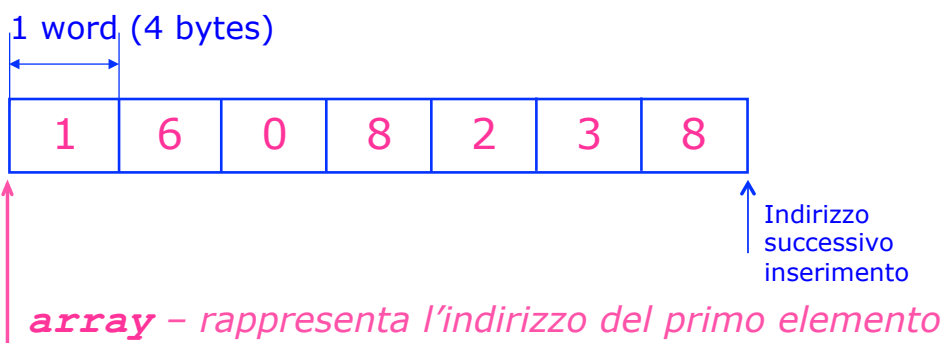


È equivalente a:

`.byte 65, 66, 67, 0`



`array: .word 1, 6, 0, 8, 2, 3, 8`





(continua) ... direttive: segmento **dati**

.align N

Allinea il dato successivo a blocchi di **2^N byte**: ad esempio:

.align 2 = .word allinea alla parola il valore successivo
.align 1 = .half allinea alla mezza parola il valore successivo
.align 0 elimina l'allineamento automatico delle direttive
.half, .word, .float, .double, fino a quando compare la successiva direttiva **.data**

Direttive – segmento testo:

.text <addr>

- Memorizza gli elementi successivi nel segmento testo dell'utente a partire dall'indirizzo **addr**. Questi elementi sono tipicamente istruzioni.

.globl sym

- Dichiarare **sym** come etichetta globale (ad essa è possibile fare riferimento da altri file)



- ❖ Direttive
- ❖ Chiamate di sistema (**system call**)
- ❖ Esempi



- ❖ Sono disponibili delle **chiamate di sistema (system call)** predefinite, che implementano i seguenti servizi:
 - 1. Accesso** alle periferiche di **I/O** (video (OUT) e tastiera (IN))
 - 2. Allocazione dinamica** di memoria dati
 - 3. Ritorno al S.O.** a fine programma

- ❖ Ogni **system call** ha:
 - un **codice**
 - eventuali **argomenti** (0, 1 o 2)
 - eventuali **valori di ritorno** (0 o 1)



- 1: print_int:**
 - stampa sulla console il numero **intero** che le viene passato come argomento;

- 2: print_float:**
 - stampa sulla console il numero in virgola mobile con **singola precisione (IEEE 754)** che le viene passato come argomento;

- 3: print_double:**
 - stampa sulla console il numero in virgola mobile con **doppia precisione (IEEE 754)** che le viene passato come argomento;

- 4: print_string:**
 - stampa sulla console la stringa che le è stata passata come argomento terminandola con il carattere **Null (0)**;



5: **read_int:**

- legge una linea in ingresso fino al carattere a capo incluso (i caratteri che seguono il numero sono ignorati) e la interpreta come **numero intero (32 bit)**;

6: **read_float:**

- leggono una linea in ingresso fino al carattere a capo incluso (i caratteri che seguono il numero sono ignorati) e la interpreta come **numero in virgola mobile – singola precisione (32 bit)**;

7: **read_double:**

- leggono una linea in ingresso fino al carattere a capo incluso (i caratteri che seguono il numero sono ignorati) e la interpreta come **numero in virgola mobile – doppia precisione (64 bit)**;

8: **read_string:**

- legge una stringa di caratteri di **lunghezza \$a1** da una linea in ingresso, scrivendoli in un **buffer (\$a0)** e terminando la stringa con il carattere **Null** (se ci sono meno caratteri sulla linea corrente, li legge fino al carattere a capo incluso e termina la stringa con il carattere **Null**);



9: **sbrk**

- alloca in memoria dati un blocco di memoria della dimensione richiesta (**n bytes**), restituendo l'indirizzo di inizio blocco;

10: **exit**

- interrompe l'esecuzione del programma;

Elenco **syscall** MIPS32:

Nome	Codice (in \$v0)	Argomenti in:	Risultato in:
print_int	1	\$a0	
print_float	2	\$f12	
print_double	3	\$f12	
print_string	4	\$a0	
read_int	5		\$v0
read_float	6		\$f0
read_double	7		\$f0
read_string	8	\$a0, \$a1	
sbrk	9	\$a0	\$v0
exit	10		

**Chiamata di una *syscall***

- ❖ Per richiedere un servizio ad una chiamata di sistema (**syscall**) occorre:
 - Caricare il **codice** della **syscall** nel registro **\$v0**
 - Caricare gli **argomenti** nei registri **\$a0 ÷ \$a3**
 - ✦ oppure nei registri **\$f12 ÷ \$f15**, nel caso di valori in virgola mobile
 - Chiamare la procedura con l'istruzione nativa: **syscall**
 - L'eventuale **valore di ritorno** è caricato nel registro **\$v0**
 - ✦ **\$f0**, nel caso di valori in virgola mobile



- ❖ Direttive
- ❖ Chiamate di sistema (system call)
- ❖ Esempi

Esempio



```

# Programma (banale) che stampa: "Dammi un intero: "
# e legge un intero, quindi lo ristampa
# sulla console.

.data
prompt: .ascii "Dammi un intero: "
str: .ascii "Il numero è "

.text
.globl main

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

      li $v0, 5       # $v0 ← codice della read_int
      syscall          # legge un intero e lo carica in $v0

      ...

```

Pseudoistruzioni:

li: load immediate

la: load address



```
...  
  
li $v0, 4      # $v0 ← codice della print_string  
la $a0, str    # $a0 ← indirizzo della stringa  
syscall       # stampa della stringa  
  
li $v0, 1      # $v0 ← codice della print_integer  
li $a0, 5      # $a0 ← intero da stampare  
syscall       # stampa dell'intero  
  
li $v0, 10     # $v0 ← codice della exit  
syscall       # esce dal programma
```



Esempio:
calcolo somma dei quadrati da 1 a n (con n=10):

Codice C:

```
int main(int argc, char* argv[])  
{  
    int i, n=10, t=0;  
    for ( i=1; i<=n; i++ )  
        t = t + i*i;  
    printf("La somma vale %d \n", t );  
    return(0);  
}
```



```
# Programma che somma i quadrati dei primi N numeri
# N e' memorizzato in $t1

        .data
str:    .asciiz "La somma vale "
        .text
        .globl main

main:   li      $t1, 10          # N interi di cui calcolare la somma quadrati
        li      $t6, 1          # $t6 e' l'indice di ciclo
        li      $s1, 0          # $s1 conterra` la somma dei quadrati
loop:   mult    $t6, $t6        # lo ← $t6 * $t6
        mflo    $t7              # $t7 ← lo
        addu    $s1, $s1, $t7    # $s1 ← $s1 + $t7
        addi    $t6, $t6, 1      # i = i+1
        slt     $t0, $t1, $t6    # if NOT (i > N) ...
        beq     $t0, $zero, loop # ... then stay in loop

        la      $a0, str        # syscall: print_string
        li      $v0, 4
        syscall

        li      $v0, 1          # syscall: print_int
        add     $a0, $s1, $zero
        syscall

        li      $v0, 10         # syscall: exit
        syscall                # esce dal programma
```

n	\$t1
i	\$t6
t	\$s1



Assembly MIPS: Le procedure



Procedura chiamante

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

flusso di
elaborazione

Procedura chiamata

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

Cosa succede alla chiamata
(ed al ritorno)?

1. Cambio di flusso
2. Passaggio informazioni: (argomenti →) , (valori ritorno ←)

Come si fa in Assembly?



Due attori: chiamante e chiamata

La procedura **chiamante** (**caller**) deve:

1. Predisporre i parametri di ingresso in un posto accessibile alla procedura chiamata
2. Trasferire il controllo alla procedura chiamata

La procedura **chiamata** (**callee**) deve:

1. Eseguire il compito richiesto
2. Memorizzare il risultato in un luogo accessibile alla procedura chiamante
3. Restituire il controllo alla procedura chiamante (→ da dove è stata chiamata)



Chiamata a procedura: jal

È necessaria un'istruzione che:

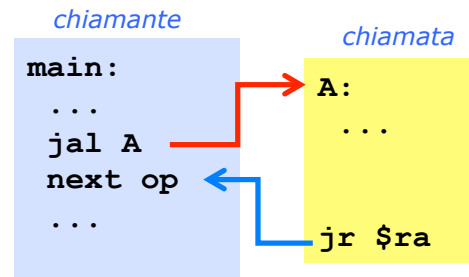
1. **cambia il flusso** di esecuzione (salta alla procedura) e
2. **salva l'indirizzo di ritorno** (istruzione successiva alla chiamata a procedura)

→ speciale istruzione di salto:

jal (jump and link)

Sintassi: **jal Indirizzo_Procedura**

1. Salta all'indirizzo con etichetta **Indirizzo_Procedura**
 2. Memorizza il **valore corrente del Program Counter** in **\$ra**
→ indirizzo dell'istruzione successiva: **$\$(PC) + 4$**
- **La procedura chiamata**, come **ultima istruzione**, esegue: **jr \$ra**
\$ra per effettuare il salto all'indirizzo di ritorno della procedura



Procedure foglia



Procedure ANNIDATE:

procedure che sono sia **chiamate** che **chiamanti**

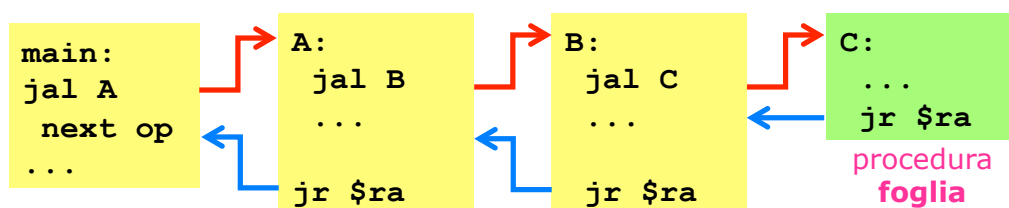
Procedura foglia: è una procedura che non contiene al suo interno chiamate ad altre procedure → è solo chiamata, non chiamante

Non è necessario salvare **\$ra**, perché nessun altro lo modifica

NON sono procedure foglia:

- Procedure che **chiamano** altre sotto-procedure
- Procedure **annidate**, Procedure **recursive**

E' necessario salvare **\$ra** perchè la procedura chiamata lo modifica a sua volta!



Procedure annidate: esempio



```
main:  ...
      li $a0, 3
      jal A
ra_M:  sw $v0, 0($t0)
      ...

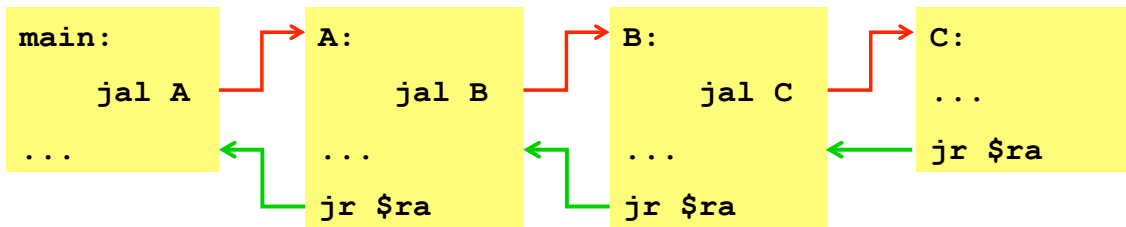
-----

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

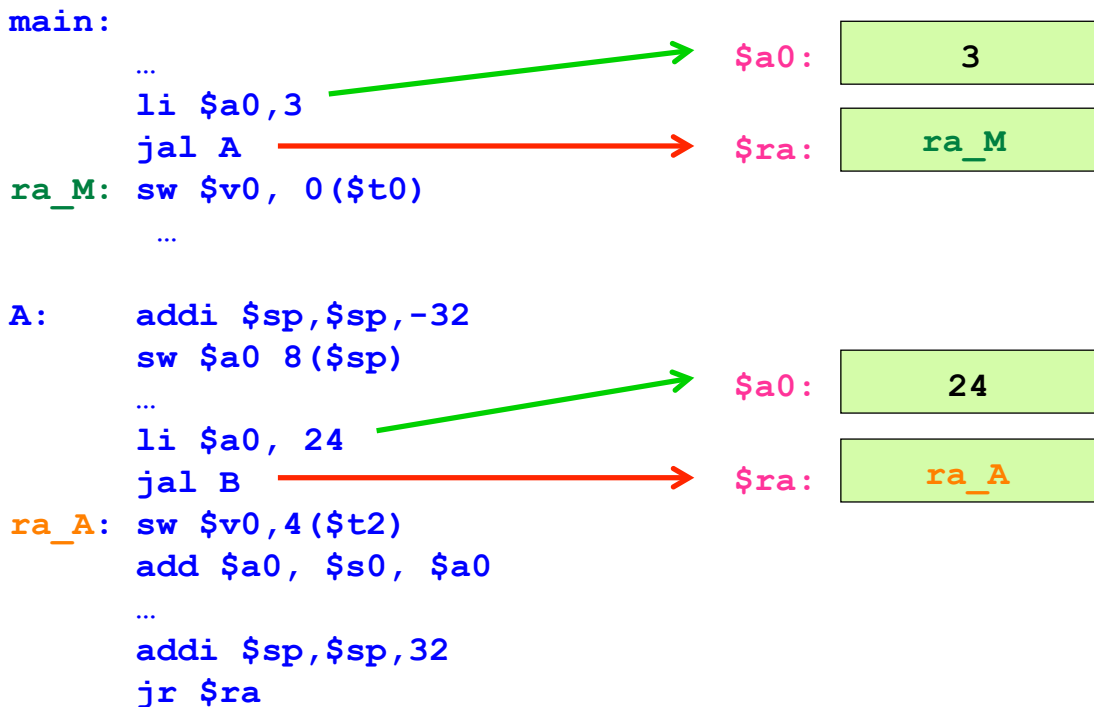
```
B:     addi $sp,$sp,-24
      ...
      li $a0, 68
      jal C
ra_B:  sw $v0,0($t3)
      ...
      jr $ra

-----

C:     addi $sp,$sp,-12
      ...
      move $a0, $s4
      sw $v0, 4($t2)
      ...
      jr $ra
```



Procedure annidate





```

B:  addi $sp,$sp,-24
    ...
    li $a0, 68
    jal C
ra_B: sw $v0,0($t3)
    ...
    jr $ra

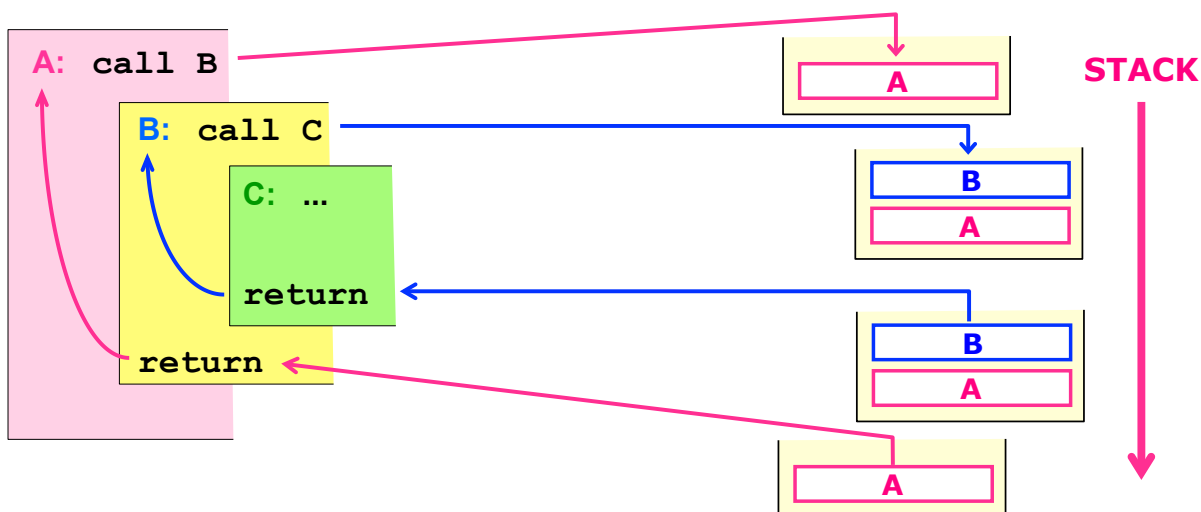
C:  addi $sp,$sp,-12
    ...
    add $a0, $s4, $zero
    sw $v0, 4($t2)
    ...
    jr $ra
    
```

\$a0: 68

\$ra: ra_B

\$a0: \$s4

\$ra: ra_B

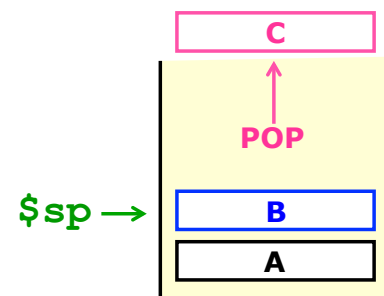
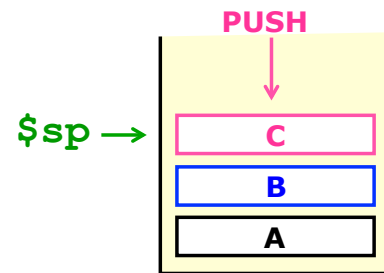


È necessario un meccanismo di memorizzazione degli indirizzi di ritorno a **catasta (stack)**:

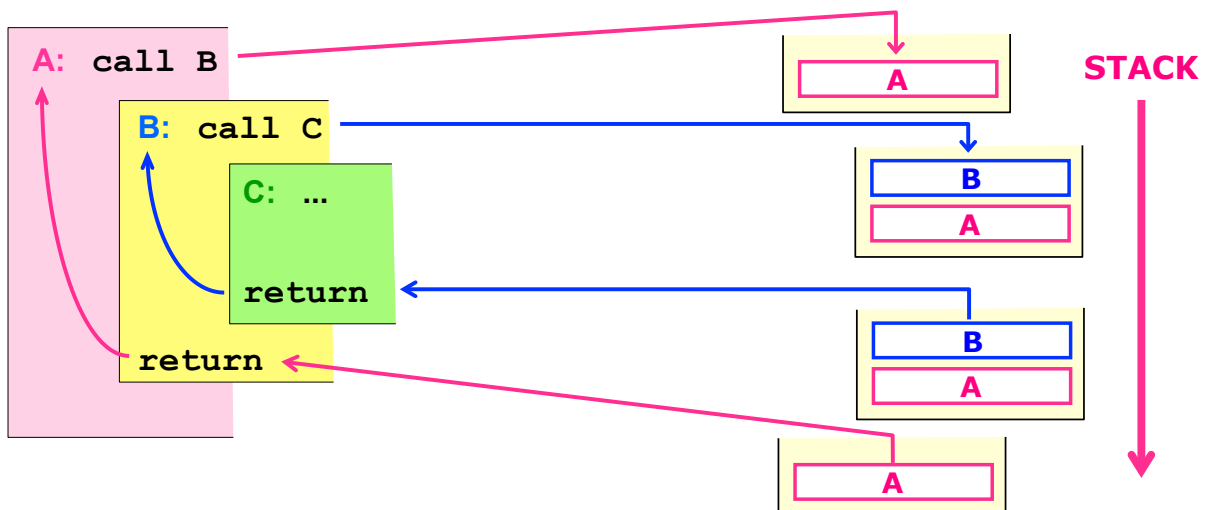
- ❖ Gli indirizzi possono essere solo: **inseriti (PUSH)** o **estratti (POP)**
- ❖ L'ultimo indirizzo inserito è il primo ad essere estratto (**Last In, First Out – LIFO**)

Lo stack:

- ❖ Struttura dati: **coda LIFO** (*last-in-first-out*)
- ❖ Inserimento dati nello stack: **PUSH**
- ❖ Prelievo dati dallo stack: **POP**
- ❖ Necessario un puntatore alla cima dello stack per salvare i dati
 - Il registro **\$sp** (**stack pointer** – puntatore allo stack) contiene l'indirizzo della **cima dello stack** (Top of Stack, TOS)
 - **\$sp** deve essere aggiornato ogni volta che viene inserito o estratto un dato dallo stack



Gestione dello stack per ritorno da procedure

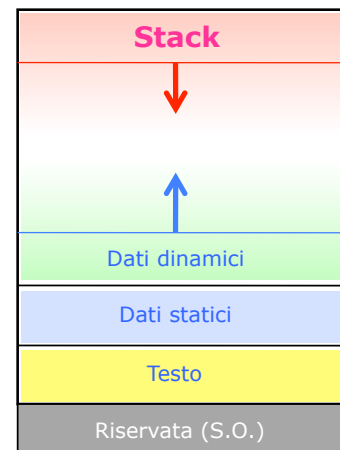


- ❖ Gestione **automatica** dello STACK:
 - Stack gestito automaticamente dall'architettura, mediante circuiti dedicati nella CPU (es. **DEC VAX**)
- ❖ **MIPS**: gestione **manuale** dello STACK:
 - Implementata secondo **convenzioni software**



Lo stack in MIPS:

- ❖ Ospitato nella zona più alta della memoria
- ❖ Lo stack cresce in memoria **da indirizzi alti verso indirizzi bassi**
- ❖ Il registro $\$sp$ contiene l'indirizzo della prima locazione libera in **cima allo stack**
- ❖ Inserimento di un dato nello stack – PUSH:
 - Si **decrementa** $\$sp$ per allocare lo spazio per una parola e
 - Si esegue **sw** per inserire il dato nello stack
- ❖ Estrazione di un dato dallo stack – POP:
 - Si esegue **lw** per recuperare la parola dallo stack
 - Si **incrementa** $\$sp$ (per eliminare il dato), riducendo la dimensione dello stack.

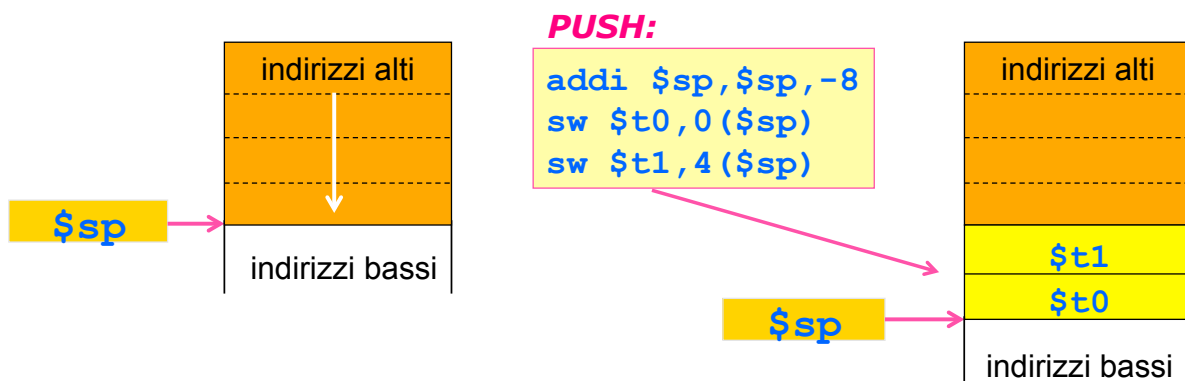


PUSH: per **inserire** elementi nello stack

```
addi $sp, $sp, -4    # Spostamento cima dello stack
sw $t0, offset($sp) # Salvataggio di $t0
```

POP: per **recuperare** elementi dallo stack

```
lw $t0, offset($sp) # Recupero di $t0
addi $sp, $sp, +4   # Ripristino cima dello stack
```



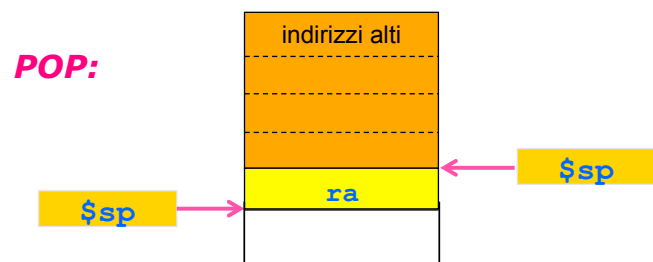


- ❖ **All'inizio della procedura**, il record di attivazione viene **allocato, decrementando \$sp** → **PUSH**

```
addi $sp, $sp, -4    # alloca 4 byte nello stack
sw $ra, 0($sp)      # salva $ra
```

- ❖ **Al termine dalla procedura**, il record di attivazione viene **rimosso, incrementando \$sp** della stessa quantità → **POP**

```
lw $ra, 0($sp)      # recupera $ra
addi $sp, $sp, +4   # dealloca 4 byte
```



Convenzioni per l'allocazione dei registri:

- ❖ **\$ra** return address
 - registro per memorizzare l'indirizzo della **prima istruzione del chiamante**, da eseguire al termine della procedura
- ❖ **\$a0÷\$a3 (\$f12÷\$f15)** registri argomento
 - usati dal chiamante per il passaggio dei parametri
 - Se i parametri sono più di 4, si usa la memoria (**stack**)
- ❖ **\$v0, \$v1 (\$f0÷\$f3)** registri valore
 - usati dalla procedura per memorizzare i valori di ritorno



La procedura *chiamata* usa gli *stessi registri usati dal chiamante*.
Al ritorno, la *chiamante* trova *registri modificati!!*

- ❖ **Record di attivazione (MIPS):**
tutto lo **spazio nello stack** di cui ha bisogno una procedura.
 - **ASSEMBLY:** lo spazio per memorizzare il record di attivazione viene esplicitamente allocato dal programmatore
- ❖ **Quando si chiama una procedura i registri utilizzati dal chiamato vanno:**
 - salvati nello stack
 - ripristinati alla fine dell'esecuzione della procedura



Sintesi convenzioni di chiamata a procedura:

- ❖ **Il programma chiamante:**
 - Inserisce i parametri da passare alla procedura nei **registri argomento: \$a0-\$a3**
 - Salta alla procedura mediante l'istruzione **jal address** e salvare il valore di **(PC+4)** nel registro **\$ra**
- ❖ **La procedura chiamata:**
 - Esegue il compito richiesto
 - Memorizza il risultato nei registri **\$v0, \$v1**
 - Restituisce il controllo al chiamante con l'istruzione **jr \$ra**

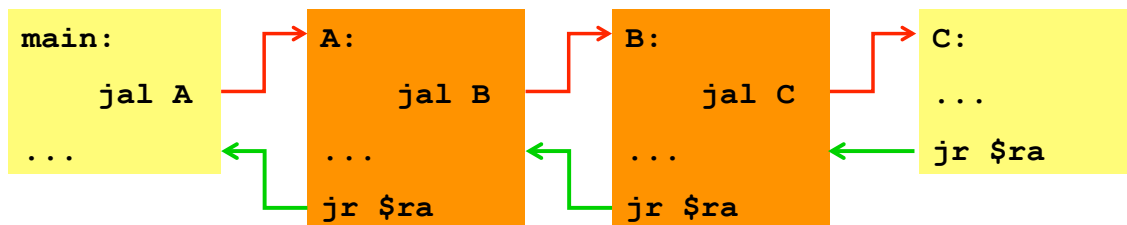


Procedure **annidate** = procedure "**non foglia**":

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**:
(indirizzo di ritorno, variabili locali non temporanee, ecc.)



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**)



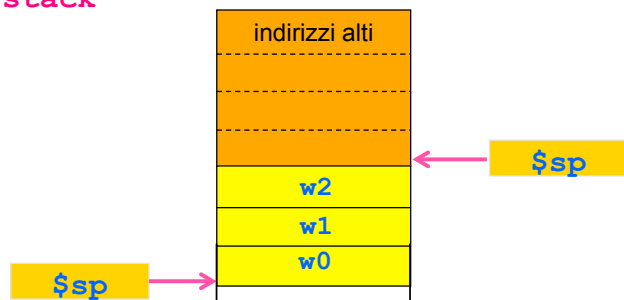
- ❖ **All'inizio della procedura**, il record di attivazione viene **allocato decrementando \$sp (PUSH)**

```
addi $sp, $sp, -12 # alloca 12 byte
                    # nello stack
```

- ❖ **Al termine della procedura**, il record di attivazione viene **rimosso incrementando \$sp della stessa quantità (POP)**

```
addi $sp, $sp, +12 # dealloca 12 byte
                    # dallo stack
```

- ❖ Tale spazio esiste soltanto **durante l'esecuzione** della procedura.



Esempio



Codice C:

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

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

g	\$a0
h	\$a1
i	\$a2
j	\$a3

Assembly:

```
# g,h,i,j associati a $a0 ... $a3, f associata ad $s0
# Il calcolo di f richiede 3 registri: $s0, $s1, $s2
# E' 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
```



```

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

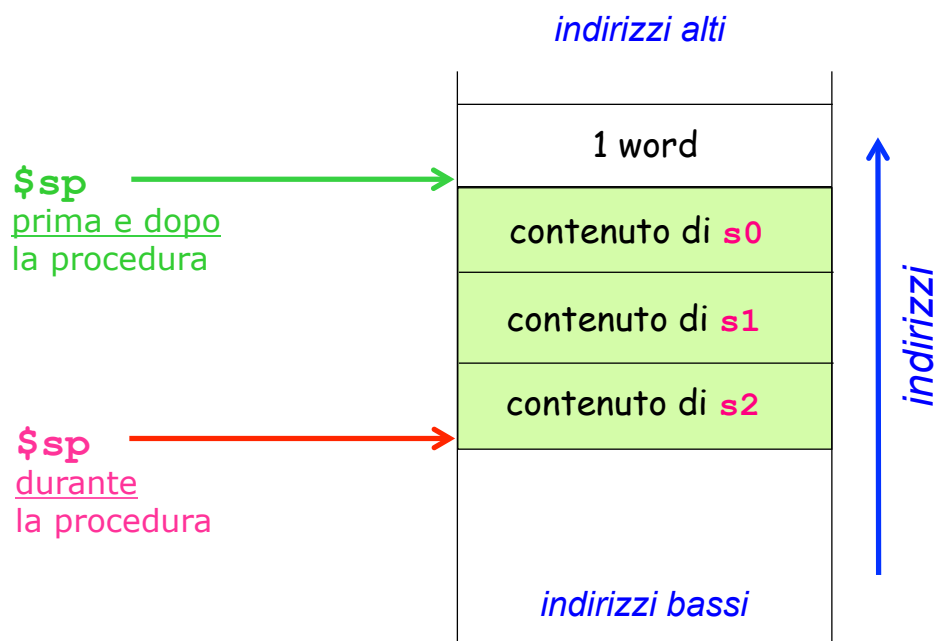
add $v0, $s2, $zero    # restituisce f copiandolo
                       # nel reg. di ritorno $v0

# ripristino del vecchio contenuto dei registri
# estraendoli dallo stack

lw $s2, 0($sp)         # ripristino di $s2
lw $s1, 4($sp)         # ripristino di $s1
lw $s0, 8($sp)         # ripristino di $s0

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

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





Convenzioni d'uso dei Registri

per evitare di salvare inutilmente i registri, sono divisi in due classi:

Registri temporanei:

$\$t0, \dots, \$t9$ ($\$f4 \dots \$f11, \$f16 \dots \$f19$)

- il cui contenuto **non è salvato dal chiamato** nello stack (viene salvato dal chiamante se serve);

Registri non-temporanei:

$\$s0, \dots, \$s8$ ($\$f20, \dots, \$f31$)

- il cui contenuto è **salvato dal chiamato** nello stack **se utilizzato**

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
```

Sintesi



- ❖ **Tutte le procedure** → il **chiamante** salva nello stack:
 - I **registri temporanei** di cui vuole salvare il contenuto di cui ha bisogno dopo la chiamata ($\$t0-\$t9, \dots$)
 - I **registri argomento** ($\$a0-\$a3, \dots$) nel caso in cui il loro contenuto debba essere preservato.
- ❖ **Procedure foglia** → il **chiamato** salva 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.
- ❖ **Procedure non-foglia** → il **chiamato** salva nello stack anche:
 - I **registri argomento** della procedura,
 - $\$ra$ ($\$fp$)



Record di attivazione: spazio **privato di ogni istanza** di una procedura

- ❖ Ospitato nello **STACK**, utilizzato per memorizzare:
 - **Return Address (\$ra)** (tranne le procedure FOGLIA)
 - Valori precedenti dei registri (variabili locali del chiamante)
 - Valori di registri locali, da proteggere (variabili locali)
- ❖ Il programmatore Assembly deve provvedere **esplicitamente** ad allocare/cedere lo spazio nello **stack**
- ❖ Nelle chiamate di procedura **annidate**, i **record di attivazione** vengono **via via allocati e rimossi** dallo stack.

Struttura di una procedura



Ogni procedura è in genere composta dalle **3 SEZIONI** consecutive:

1. Prologo

- **Acquisire spazio nello STACK** per memorizzare i dati interni alla procedura ed il salvataggio dei registri
- **Salvataggio** dei registri di interesse (**record di attivazione**)

2. Corpo

- Esecuzione della procedura vera e propria
- Porre il/i risultato/i dove se li aspetta il programma chiamante (**registri valore**)

3. Epilogo

- **Ripristino** dei registri di interesse
- **Liberare lo spazio nello STACK** allocato nel prologo
- Restituzione del controllo alla procedura chiamante (**jr \$ra**)



- ❖ Determinazione della dimensione del record di attivazione
 - Stimare lo spazio per: **\$ra**, registri da salvare, argomenti, variabili locali

- ❖ Allocazione dello spazio in stack: aggiornamento valore di **\$sp**:

```
addi $sp, $sp, -dim_record_attivaz
# lo stack pointer viene decrementato
# della dimensione del record di attivazione
```

- ❖ Salvataggio dei registri di interesse nello stack:

```
sw reg, [dim_record_attivaz-N]($sp)
N viene incrementato di 4 ad ogni salvataggio (N ≥ 4)
```

Esempio: record di attivazione: 3 registri → **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
```



Struttura epilogo:

- ❖ Restituzione dei parametri della procedura
 - dai registri valore (**\$v0**, **\$v1** e/o stack)
- ❖ Ripristino dei registri dallo stack
 - registri di interesse e **\$ra** per procedure non-foglia

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

- ❖ Liberare lo spazio nello stack in cui sono stati memorizzati i dati locali.

```
addi $sp, $sp, dim_record_attivaz.
```

- ❖ Trasferire il controllo al programma chiamante:

```
jr $ra
```




Esempio: programma che, tramite una procedura, copia solo gli interi positivi contenuti nell'array (`elem`) in un secondo array (`pos_el`)

Convenzioni passaggio informazioni:

Argomenti:

- ❖ la procedura si aspetta:
 - posizione inizio array `elem` → `$a0`
 - posizione inizio array `pos_el` → `$a1`
 - dimensione (in byte) array `elem` → `$a2`

Valori di ritorno:

- ❖ il chiamante si aspetta:
 - dimensione array `pos_el` → `$v0`

❖ Algoritmo:

1. Inizializzazione
2. Leggi gli interi
3. Individua gli interi positivi
4. Copia gli interi positivi
5. Stampa gli interi positivi

Esempio – 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 – 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 prossimo 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
     add $t0, $s0, $zero # $t0 ← 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 (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.
     add $a0, $s0, $zero # $a0 ← ind. array "elem"
     add $a1, $s1, $zero # $a1 ← ind. array "pos_el"
     add $a2, $s2, $zero # $a2 ← dim. in byte dell'array "elem"

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



```

# Stampa num. elementi positivi
add $t1, $v0, $zero    # $t1 ← n. elementi pos_el
add $t2, $s1, $v0     # $t2 ← indirizzo fine array pos_el
add $t0, $s1, $zero   # $t0 ← indirizzo base array pos_el

li $v0, 4              # $v0 ← codice della print_string
la $a0, msg            # $a0 ← indirizzo della stringa "msg"
syscall                # print_string: stringa "msg"

li $v0, 1              # $v0 ← codice della print_int
add $a0, $t1, $zero   # $a0 ← N
syscall                # print_int

# Ciclo di stampa degli elementi positivi
loop_w:
    beq $t2, $t0, exit
    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:  li $v0, 10      # $v0 ← codice della exit
      syscall         # esce dal programma
    
```



```

# 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)

      add $s0, $a0, $zero    # $s0 ← ind. inizio array elem
      add $s1, $a0, $a2     # $s1 ← ind. fine array elem
      add $s2, $a1, $zero   # $s2 ← ind. inizio array pos_el
    
```

Spazio privato della funzione `cp_pos`:

registri: `$s0, $s1, $s2, $s3`

→ li salvo nello **STACK**



```
next:  beq $s1, $s0, exit_f  # 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_f: sub $v0, $s2, $a1    # salvo lo spazio in byte dei pos_el
        divi $v0, $v0, 4    # ...e lo divido per 4 (ottenendo N)

        lw $s0, 0($sp)
        lw $s1, 4($sp)
        lw $s2, 8($sp)
        lw $s3, 12($sp)
        addi $sp, $sp, +16  # deallocazione dello stack

        jr $ra              # restituisce il controllo al chiamante
```



Assembly MIPS: procedure recursive



Procedura ricorsiva:
procedura annidata, che **contiene chiamate a se stessa.**

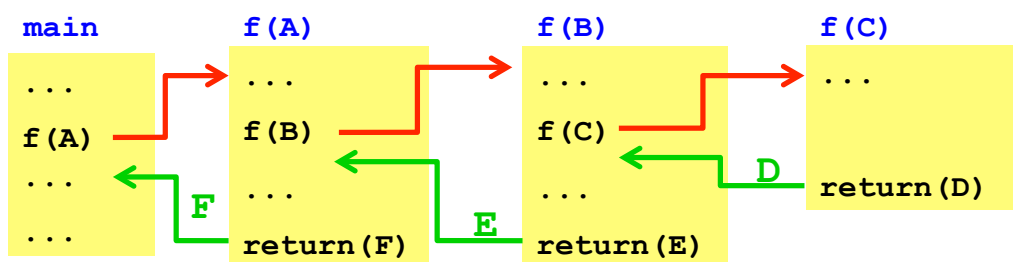
Problema:
tutte le variabili locali (registri) sono condivise tra le istanze di chiamata

Soluzione: salvare nello stack:

- L'indirizzo di ritorno
- I parametri di input della procedura

...e in più: le variabili di lavoro del record di attivazione

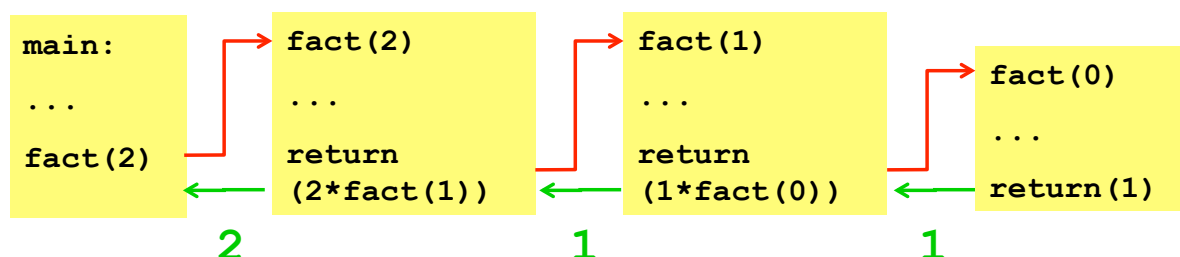
- **Le variabili di lavoro** (anche i risultati intermedi, se a cavallo della chiamata!)



Fattoriale: esempio di calcolo



- ❖ Fattoriale: $n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot (n-1) \cdot n$
- ❖ in forma ricorsiva: $n! = n \cdot (n-1)!$ (ponendo: $0! = 1$)
- ❖ La procedura: **fact(n)** viene invocata **n+1** volte
 - n+1 record di attivazione della stessa procedura:





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: .ascii "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
```



```

# 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
    
```



```

# 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
    
```



```
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 **decrece nel ciclo inferiore**.



Serie di Fibonacci: $fib(n) = fib(n-1) + fib(n-2)$
 $fib(0) = 0, fib(1) = 1$

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) );
}
```



```
# Programma per il calcolo ricorsivo del Numero di
# Fibonacci.

        .data

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

        .text
        .globl main

main:

        li $v0, 4
        la $a0, prompt
        syscall          # stampa la stringa

        li $v0, 5
        syscall          # legge l'intero

        ...
```



```

...

# calcola fibonacci(n)
move $a0, $v0
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(n): legge in $a0 il numero n; restituisce in $v0 fib(n)
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  # se n=0 o n=1: fib(n)=n.
      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
      # non recupero $a0 perché preservato!
      addi $a0, $a0, -1    # $a0 diventa: n-2
      jal fib              # esegue fib(n-2)
      # non recupero $a0 perché preservato!
      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

```



```
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(...);
}
```



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

.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
```



```

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
    
```



```

# 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
    
```



```

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
    
```



```

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 $t2, 0($t0)         # 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
    
```