Procedure/Funzioni

- **Molto utili nei linguaggi ad alto livello**
  - astrazione - riusabilità - strutturazione/modularità
- **Chi "gestisce" le chiamate?**
  - il compilatore e il supporto a runtime, nei linguaggi ad alto livello
  - il programmatore in Assembly
- **Passi necessari**
  - passare i parametri al chiamato
  - salvare l'indirizzo di ritorno e passare il controllo alla procedura
  - allocare spazio per variabili locali
  - eseguire il corpo della procedura
  - passare i risultati al chiamante
  - riportare il controllo al punto di partenza

Indirizzo di ritorno

- **Registri riservati da MIPS alle chiamate di procedura:**
  - `$a0,$a1,$a2,$a3` (per il **passaggio** di parametri)
  - `$v0,$v1` (per la **restituzione** dei valori)
  - `$ra` (per **ritornare** al punto di origine del chiamante)
- **E' necessario *salvare* l'indirizzo di ritorno**

  Istruzione *jump and link* (*J-type*)

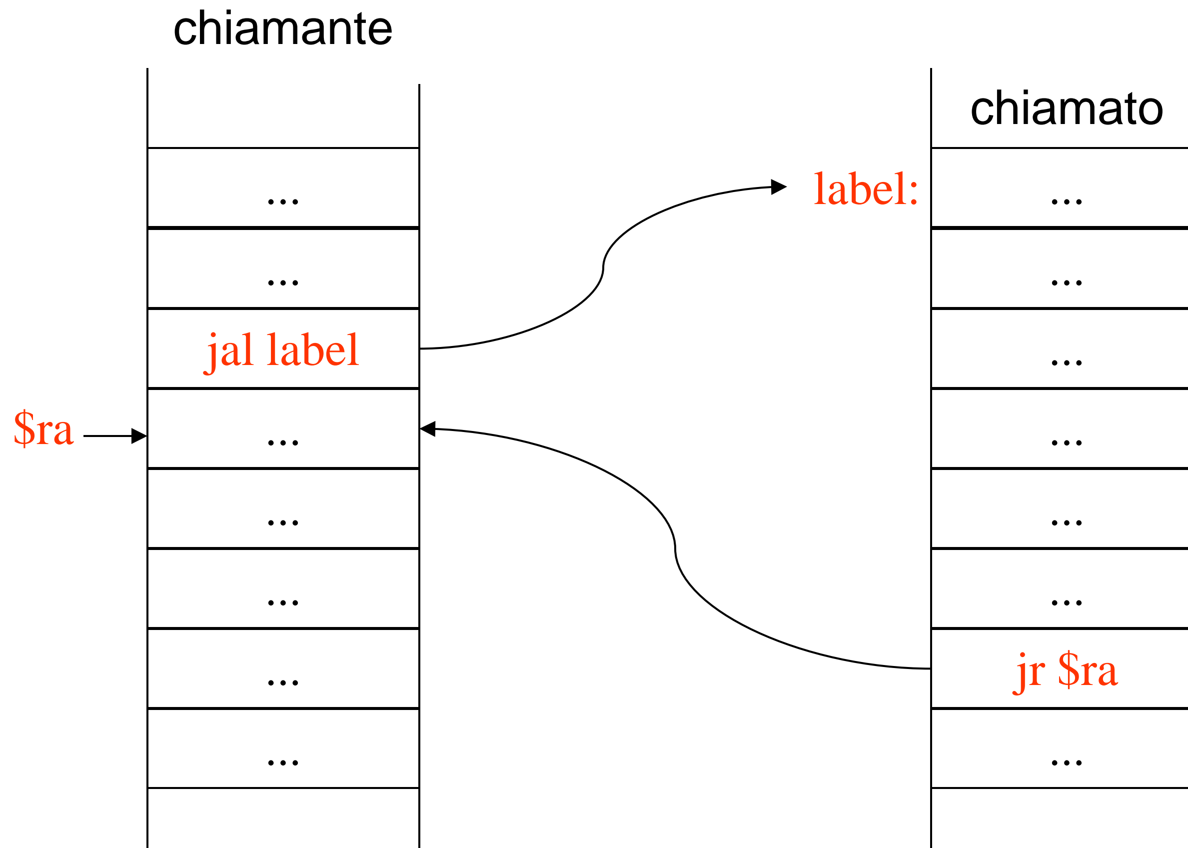
  `jal` *`indirizzo`*

  salta all'istruzione memorizzata ad *indirizzo* e pone in `$ra` l'indirizzo dell'istruzione successiva a `jal` *`indirizzo`* (indirizzo di ritorno).
- **Istruzione per il ritorno (*R-type*)**

  `jr $ra`

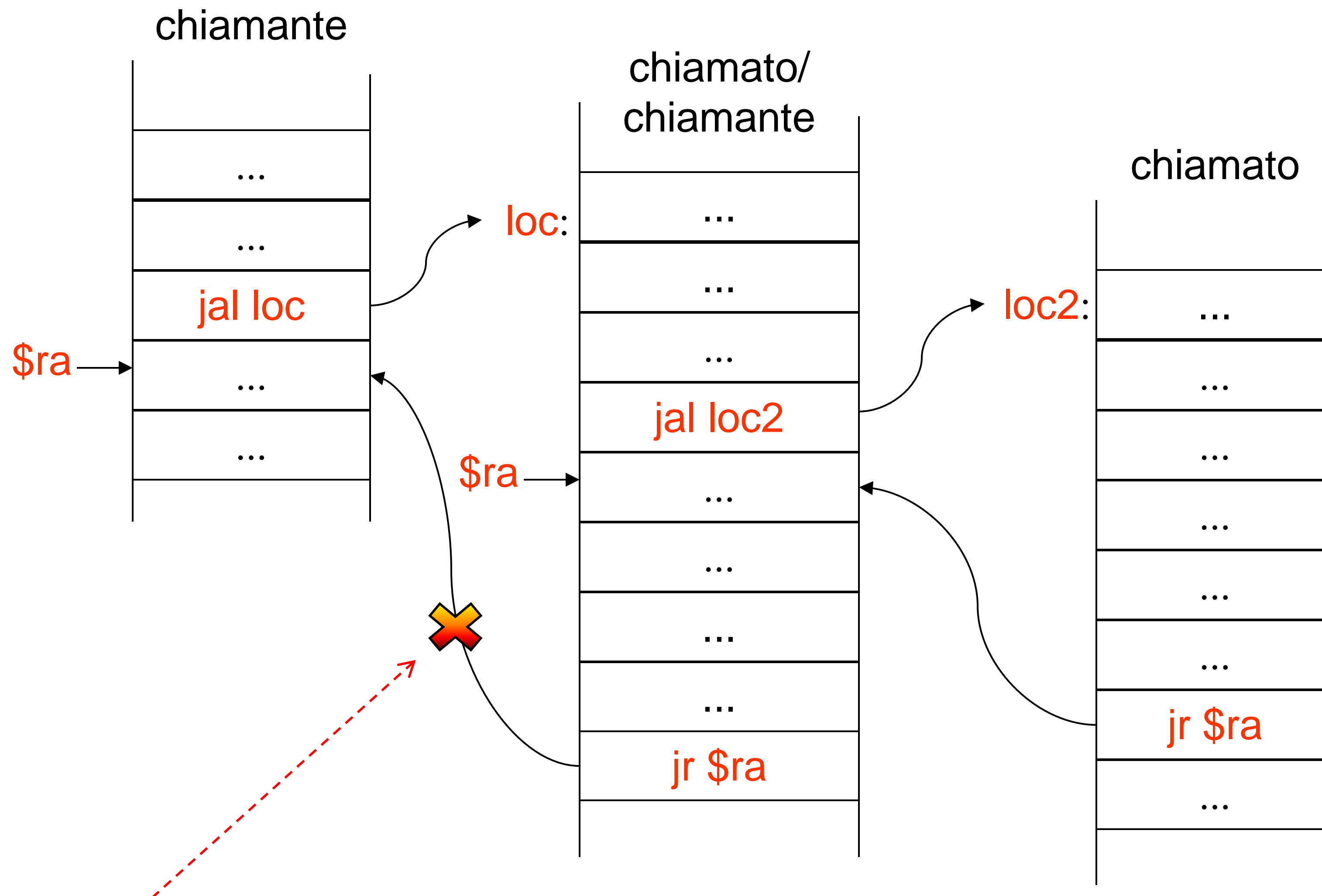
  esegue l'istruzione il cui *indirizzo* è contenuto in `$ra`
- **L'indirizzo dell'istruzione corrente è contenuto nel registro riservato PC (Program Counter)**
  - `jal` mette in `$ra` il valore (PC)+4
  - `jr $ra` mette in PC `$ra`

Chiamata di una procedura



E se il chiamato chiama un'altra procedura?

Chiamate annidate



Per ritornare al chiamante devo salvare il contenuto di $ra prima della 'jal loc2' e ripristinarlo prima della jr

Uso della PILA (STACK)

- **Per ogni procedura non ancora terminata esistono dei *dati locali* e un *indirizzo di ritorno* della procedura**
- **Passaggio dei parametri al chiamato:**
  - Si usano i registri `$a0...$a3` (4 registri in tutto)
- **Se ho più di 4 parametri da passare, o più chiamate innestate?**
- **Occorre una *struttura dati* per memorizzare le variabili locali del chiamato, gestire le chiamate annidate e i parametri dal 5^ in poi**

PILA (Stack)

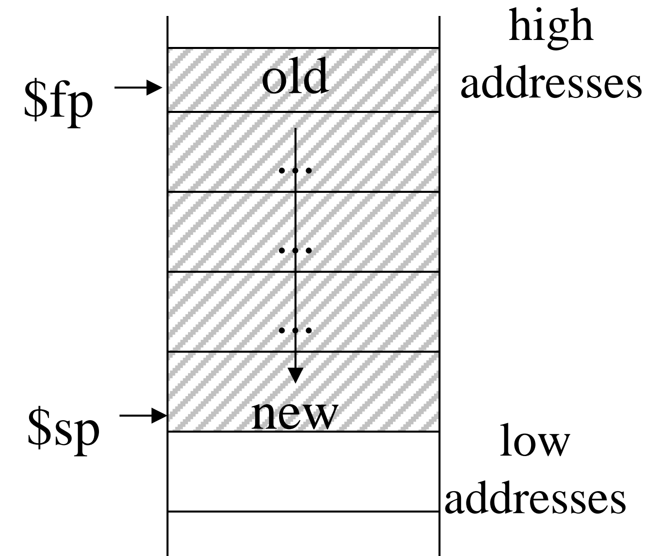
Uso della PILA (STACK)

- **Stack Pointer** `$sp`
  - punta sempre alla cima dello stack
    (= ultima word occupata!)
- **Struttura dati LIFO**
  - **push**:

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

  - **pop**:

```
lw   $t0, 0($sp)
addi $sp, $sp, 4
```



- **Frame Pointer** `$fp`
  - registro che indirizza la prima parola dello stack allocata dalla procedura attualmente in esecuzione

Stack Frame

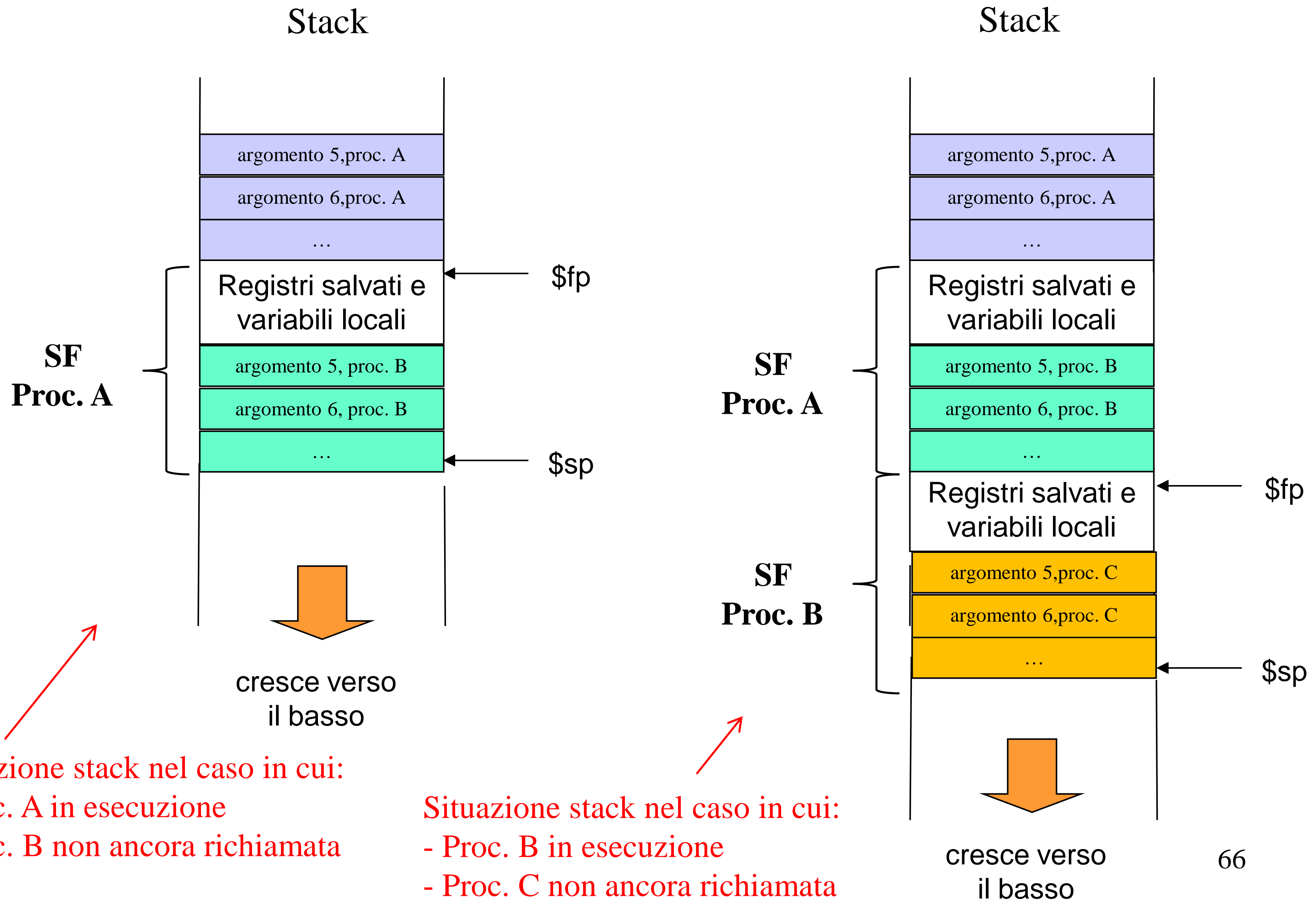
Scopo

- **Memorizzare i dati di una procedura in una struttura singola**
- **Si può accedere ai dati usando `$fp` o `$sp` come puntatore**

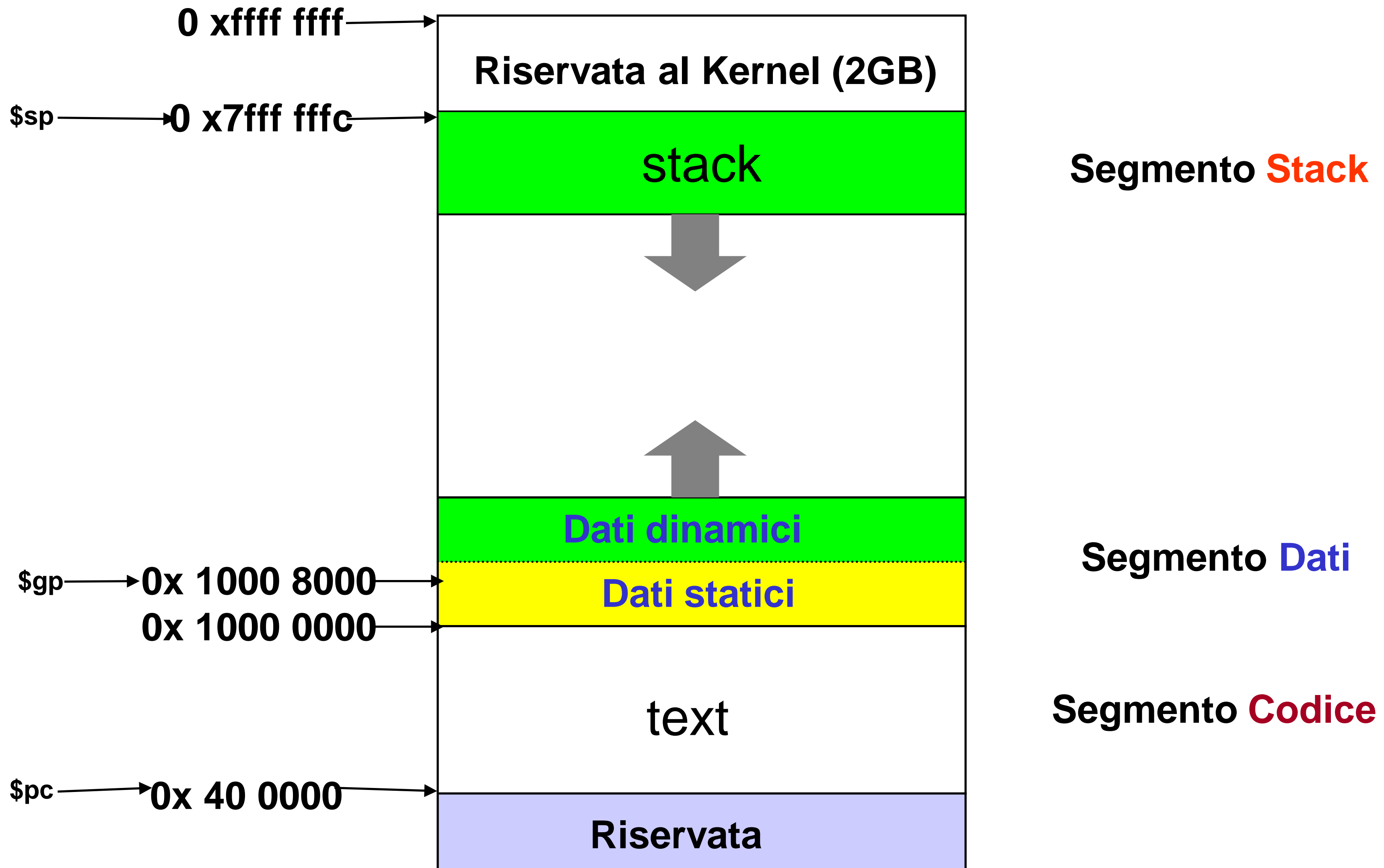
- **Contenuti dello SF**
  - valori di registri salvati
  - variabili locali della procedura
  - argomenti dal 5° in poi per chiamate a procedura

- **E' necessario usare lo SF?**
  - si, per chiamate di procedure complesse
  - no, per programmi semplici
  - I compilatori lo usano!

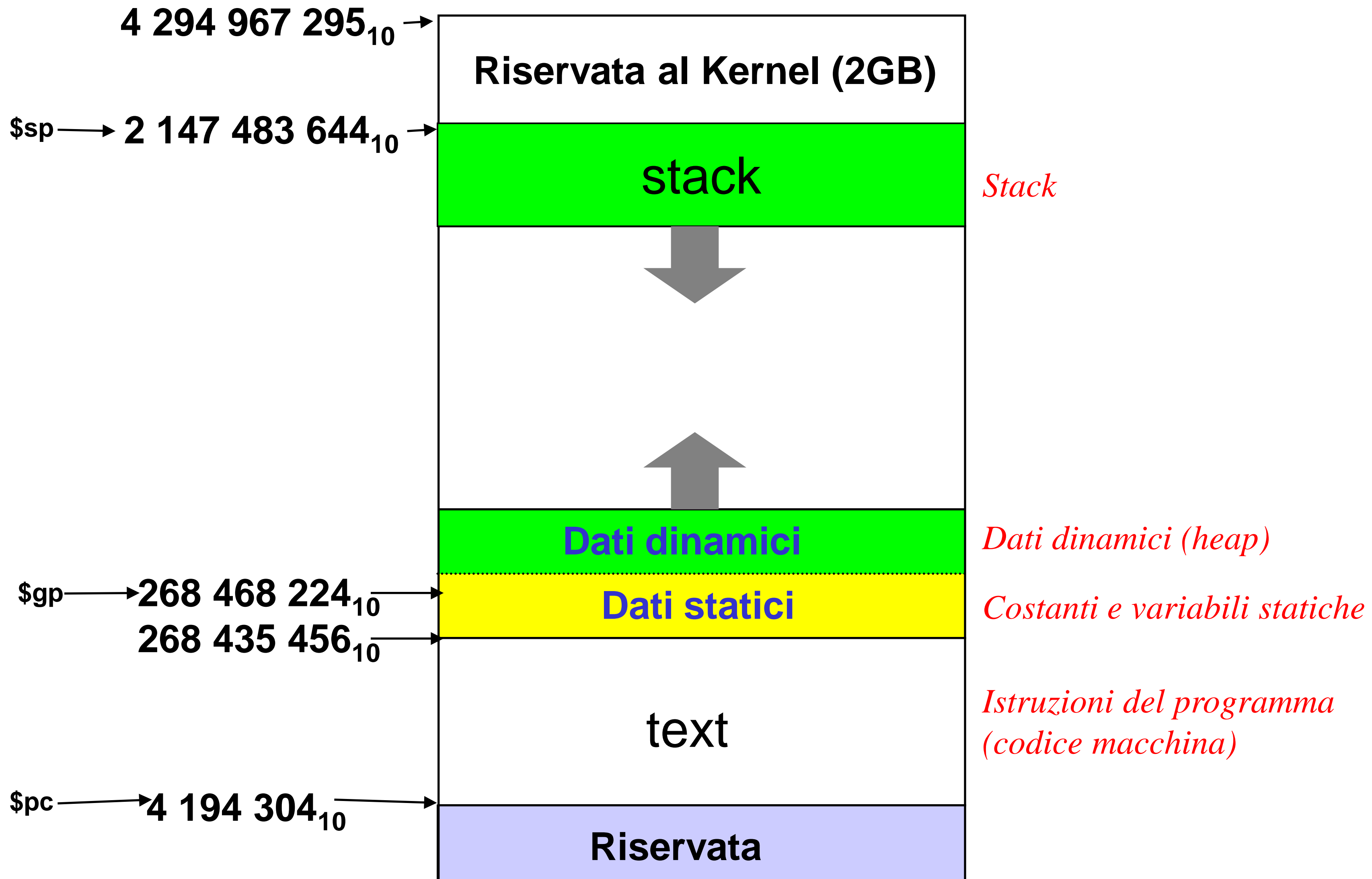
Stack Frame



Organizzazione della memoria



Organizzazione della memoria



Global Pointer

- **Il registro Global Pointer (per convenzione `$gp`) viene utilizzato come puntatore globale caricandovi l'indirizzo 0x10008000 (è automatico in QtSpim), in modo da rendere direttamente accessibili tutti i primi 64KB (l'offset è su 16 bit) dei dati statici (da 0x10000000 a 0x1000ffff, ovvero da $268\ 435\ 456_{10}$ a $268\ 500\ 991_{10}$)**

```
lw $v0, 0($gp)
```

Convenzioni per la *chiamata* di Procedura

- **Il chiamante (prima della chiamata):**
  - salva nel proprio SF i registri **`$a0,…..,$a3,$v0,$v1`** e **`$t0,…,$t9`** **se intende usarli dopo la chiamata** (reg. *caller-saved*): questi sono parte dei dati locali
  - passa gli argomenti se ce ne sono. Per i primi 4 si usano i registri **`$a0,…..,$a3.`** I successivi in pila
  - esegue **`jal`** (salta alla prima istruzione del chiamato e salva in **`$ra`** l'indirizzo di ritorno)

- **Il chiamato (prima di eseguire le proprie istruzioni):**
  - alloca memoria per il proprio SF. Es., se lo stack frame è di 32 byte, **`$sp = $sp - 32`**
  - salva nello SF i registri **`$s0,…,$s7, $fp, $ra`** **se intende alterarli** (reg. *callee-saved*)
  - inizializza il frame pointer, Es. **`$fp = $sp + 28`**

Convenzioni per il *ritorno* da Procedura

- **Il chiamato (dopo avere eseguito le proprie istruzioni):**
  - mette i valori risultato in **`$v0`** e **`$v1`** (se esistono)
  - **ripristina** da SF i registri *callee-saved* che aveva salvato prima dell'esecuzione della procedura (compreso, eventualmente, **`$ra`**)
  - de-alloca lo SF: **`$sp = $sp + 32`**
  - ritorna al programma chiamante: **`jr $ra`**

- **Il chiamante (al ritorno della procedura):**
  - usa i valori risultato in **`$v0`** e **`$v1`** (se esistono)
  - **ripristina** da SF i registri *caller-saved*

Convenzioni sull'uso dei registri

Nome	Numero	Uso	Preservato in una chiamata a procedura?
$zero	0	il valore costante 0	n.a.
$v0-$v1	2-3	valori per risultati di espressior	no
$a0-$a3	4-7	argomenti	no
$t0-$t7	8-15	temporanei	no
$s0-$s7	16-23	salvati	si
$t8-$t9	24-25	altri temporanei	no
$gp	28	global pointer	si
$sp	29	stack pointer	si
$fp	30	frame pointer	si
$ra	31	return address	si

Ottimizzazione delle chiamate

- **Tipi di chiamate di procedura**
  - **non-foglia**
    - procedure che chiamano altre procedure
    - procedure ricorsive
  - **foglia**
    - procedure che non effettuano chiamate
      - che richiedono l’uso della pila per variabili locali
      - che non richiedono uso della pila

Compilare una procedura “foglia” (senza pila)

$s1 → v[], $s2 → k

```
void swap(int v[], int k)
{
   int tmp;
   tmp = v[k];
   v[k] = v[k+1];
   v[k+1] = tmp;
}
```

```
swap:   add $t0,$a1,$a1    } $t0=4k
        add $t0,$t0,$t0    }
        add $t0,$t0,$a0
        lw  $t1,0($t0)
        lw  $t2,4($t0)
        sw  $t2,0($t0)
        sw  $t1,4($t0)
        jr  $ra
```

Caller:

```
add $a0,$s1,$zero
add $a1,$s2,$zero
jal swap
```

8 istruzioni, di cui 4 accessi alla memoria

Compilare una procedura "foglia" (con pila, versione corretta ma inefficiente)

```
              $s1       $s2
void swap(int v[], int k)

{
   int tmp;
   tmp = v[k];
   v[k] = v[k+1];
   v[k+1] = tmp;
}
```

```
swap:   addi $sp,$sp,-4
        sw   $s0,0($sp)
        add  $s0,$a1,$a1     } $s0=4k
        add  $s0,$s0,$s0     }
        add  $s0,$s0,$a0
        lw   $t1,0($s0)
        lw   $t2,4($s0)
        sw   $t2,0($s0)
        sw   $t1,4($s0)
        lw   $s0,0($sp)
        addi $sp,$sp,4
        jr   $ra
```

Rispetto alla versione precedente, si passa da 8 a 12 istruzioni, di cui 6 accessi alla memoria (invece di 4, ovvero +50%).

Caller:

```
add $a0,$s1,$zero
add $a1,$s2,$zero
jal swap
```

Compilare una procedura “non foglia”

```
int square(int a){              square:
   return a*a;                          mul     $v0,$a0,$a0
}                                       jr      $ra

int poly(int x){                poly:   addi    $sp,$sp,-8
   return square(x)+x+1;                sw      $ra,4($sp)
}                                       sw      $a0,0($sp)
                                        jal     square
                                        lw      $a0,0($sp)
                                        add     $v0,$v0,$a0
                                        addi    $v0,$v0,1
                                        lw      $ra,4($sp)
                                        addi    $sp,$sp,8
                                        jr      $ra
```