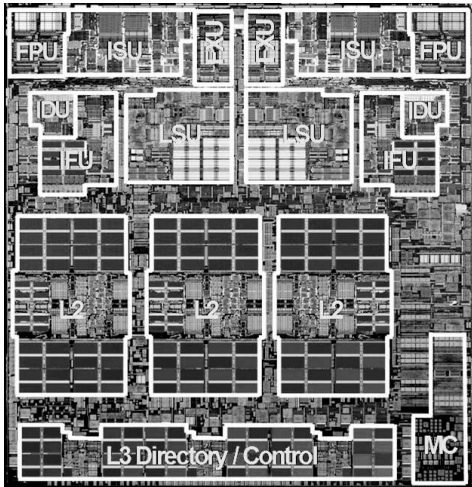




Università degli Studi di Cassino



Corso di Calcolatori Elettronici I

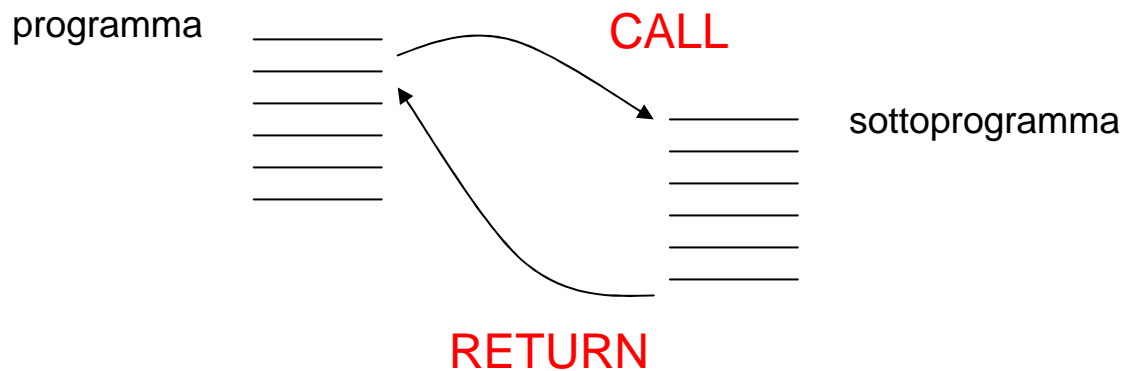
Gestione dei sottoprogrammi

Anno Accademico 2007/2008

Francesco Tortorella

Gestione dei sottoprogrammi

temporaneo passaggio del controllo dal programma in esecuzione ad un sottoprogramma



Subroutine linkage { **CALL**: viene salvato l'indirizzo di ritorno e quindi si effettua il salto
RETURN: viene recuperato l'indirizzo di ritorno e si effettua il salto

Esempio di chiamata di funzione

```
void main()  
{  
    int x=3,y=4,z;  
    z=sum(x,y);  
}
```

```
int sum(int a, int b)  
{  
    int c;  
    c=a+b;  
    return c;  
}
```

Tecniche per il Subroutine Linkage

- **Link Register**
 - accesso ad un registro interno
 - gestione semplice e veloce
 - rende efficiente il caso di chiamata non innestata
- **Stack**
 - accesso a registri in memoria centrale
 - gestione più complessa (problema dello stack overflow)
 - risolve automaticamente il problema delle chiamate innestate

Tecniche per il passaggio dei parametri

- **Registri interni**
 - semplice e veloce
 - efficiente solo nel caso di pochi argomenti e procedure non innestate (caso più frequente ?)
- **Stack**
 - accesso a registri in memoria centrale
 - gestione più complessa (problema dello stack overflow)
 - robusto rispetto al problema del numero degli argomenti e delle chiamate innestate

MIPS: Convenzioni sui registri

- CalleR: funzione chiamante
- CalleE: funzione chiamata, sottoprogramma
- Quando termina l'esecuzione del sottoprogramma, il chiamante deve sapere quali registri potrebbero essere stati modificati e quali invece è garantito siano rimasti inalterati.
- **Convenzione sui registri**: un insieme di regole che definiscono quali registri devono rimanere inalterati dopo una chiamata a sottoprogramma e quali possono invece essere modificati.

Convenzioni sui registri: registri da non modificare

- **\$0**: Nessuna modifica.
- **\$s0-\$s7**: Da ripristinare se modificati. Se il sottoprogramma modifica uno di questi registri, è tenuto a ripristinarlo prima del ritorno al chiamante.
- **\$sp**: Da ripristinare se modificato. Dopo l'esecuzione di un `jal`, lo stack pointer deve puntare allo stesso indirizzo cui puntava prima dell'esecuzione. Diversamente, il chiamante non potrebbe accedere correttamente a dati eventualmente presenti sullo stack.
- Nota – I nomi dei registri da **S**alvare iniziano con **S**

Convenzioni sui registri: registri modificabili

- **\$ra**: modificabile. La stessa istruzione jal modifica il registro. Dovrà essere salvato sullo stack a cura del chiamante in caso di chiamate innestate.
- **\$v0-\$v1**: modificabili. Al ritorno dal sottoprogramma contengono i valori restituiti al chiamante.
- **\$a0-\$a3**: modificabili. Il chiamante è tenuto a salvarli nel caso debba accedervi dopo la chiamata.
- **\$t0-\$t9**: modificabili. Il chiamante è tenuto a salvarli nel caso debba accedervi dopo la chiamata.

Calling conventions

Riguardano le scelte operate dal chiamante relative a:

- Salvataggio dell'indirizzo di ritorno
- Salvataggio dei parametri
- Numero dei parametri
- Tipo dei parametri
- Ordine dei parametri

Leaving conventions

Riguardano le scelte relative a:

- Passaggio dei parametri in uscita
- Rispristino del contesto antecedente alla chiamata

Convenzioni per il MIPS

- Invocazione di un sottoprogramma con l'istruzione `jal`. Indirizzo di ritorno salvato in `$ra`.
- Ritorno al chiamante con l'istruzione `jr $ra`.
- Parametri effettivi passati nei registri `$a0–$a3`.
- Valori restituiti nei registri `$v0–$v1`.
- Rispetto delle convenzioni sui registri.

MIPS: Attivazione di un sottoprogramma

- **Caller**

- Preparazione dei parametri effettivi nei registri $\$a^*$
- Salvataggio dell'indirizzo di ritorno e salto

- **Callee**

- Prelievo dei parametri
- Esecuzione delle istruzioni
- Preparazione dei valori restituiti nei registri $\$v^*$
- Ritorno

- Altre istruzioni

Esempio di chiamata di funzione

```
void main()  
{  
    int x=3,y=4,z;  
    z=sum(x,y);  
}
```

```
int sum(int a, int b)  
{  
    int c;  
    c=a+b;  
    return c;  
}
```

```

# Esempio di chiamata a funzione
    .data
x:    .word 3
y:    .word 4
z:    .space 4

    .text
main: lw $a0, x      # x è in $a0
      lw $a1, y      # y è in $a1
      jal sum
      sw $v0, z      # salva la somma in z

      li $v0, 10     # Codice di chiamata al sistema per la
      syscall        # fine del programma

sum:  add $t2,$a0,$a1 # Esegue la somma
      move $v0,$t2   # Salva il risultato
      jr $ra         # Ritorna al chiamante

```

```
# Esempio di chiamata a funzione
```

```
.data
```

```
x: .word 3
```

```
y: .word 4
```

```
z: .space 4
```

Preparazione
parametri effettivi

```
.text
```

```
main: lw $a0, x # x è in $a0
```

```
lw $a1, y # y è in $a1
```

```
jal sum
```

```
sw $v0, z # salva la somma in z
```

Salto

Prelievo
parametri effettivi

```
li $v0, 10 # Codice di chiamata al sistema per la
```

```
syscall # fine del programma
```

```
sum: add $t2, $a0, $a1 # Esegue la somma
```

```
move $v0, $t2 # Salva il risultato
```

```
jr $ra # Ritorna al chiamante
```

Preparazione del
valore restituito

Ritorno

Esempio di chiamata innestata

```
void main()  
{  
    int x=3,y=4,z;  
    z=sum_square(x,y);  
}
```


```
int sum_square(int a, int b)  
{  
    int as,bs,c;  
    as=mply(a,a);  
    bs=mply(b,b);  
    c=as+bs;  
    return c;  
}
```

```
int mply(s,t)  
{  
    int w;  
    w=s*t;  
    return w;  
}
```

```

# Esempio di chiamata innestata a funzione
    .data
x:    .word 3
y:    .word 4
z:    .space 4

    .text
main: lw $a0, x      # x è in $a0
      lw $a1, y      # y è in $a1
      jal sumq
      sw $v0, z      # salva la somma in z
      li $v0, 10     # Codice di chiamata al sistema per la
      syscall       # fine del programma

sumq: 

mply: mul  $t3,$a0,$a1  # Esegue il prodotto
      move $v0,$t3     # Salva il risultato
      jr  $ra         # Ritorna al chiamante

```

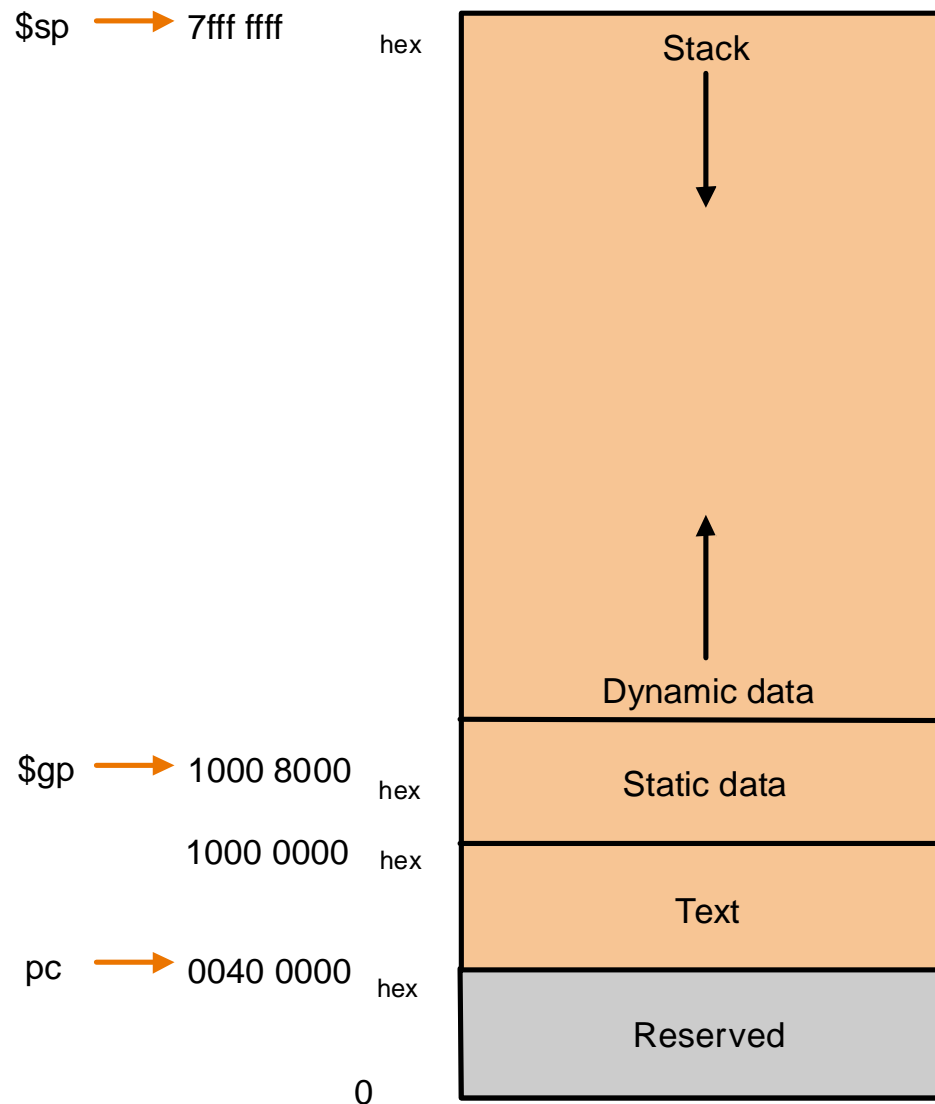

Problemi con le funzioni innestate

- La funzione `sumq` deve invocare una seconda funzione e ciò comporta:
 - Preparazione dei parametri effettivi in `$a0` `$a1`
 - Distruzione di uno dei parametri ricevuti da `main`.
 - Salvataggio dell'indirizzo di ritorno in `$ra` e salto
 - Distruzione dell'indirizzo di ritorno verso `main`.
- Come impedire la perdita dei parametri effettivi ricevuti da `main` e l'indirizzo di ritorno verso `main` ?
 - Necessario salvare in memoria tali informazioni.

Problemi con le funzioni innestate

- In generale, potrebbe essere necessario salvare anche altri dati → necessario uno strumento ad hoc.
- All'atto dell'esecuzione di un programma, sono allocate in memoria tre aree diverse per la gestione dei dati:
 - **Area dati statici**: Variabili definite nel programma che cessano di esistere al termine dell'esecuzione del programma (es. Variabili globali in C)
 - **Heap**: Variabili allocate dinamicamente (es. con malloc o new)
 - **Stack**: spazio utilizzabile dai sottoprogrammi durante l'esecuzione (in particolare per salvare il contenuto dei registri).

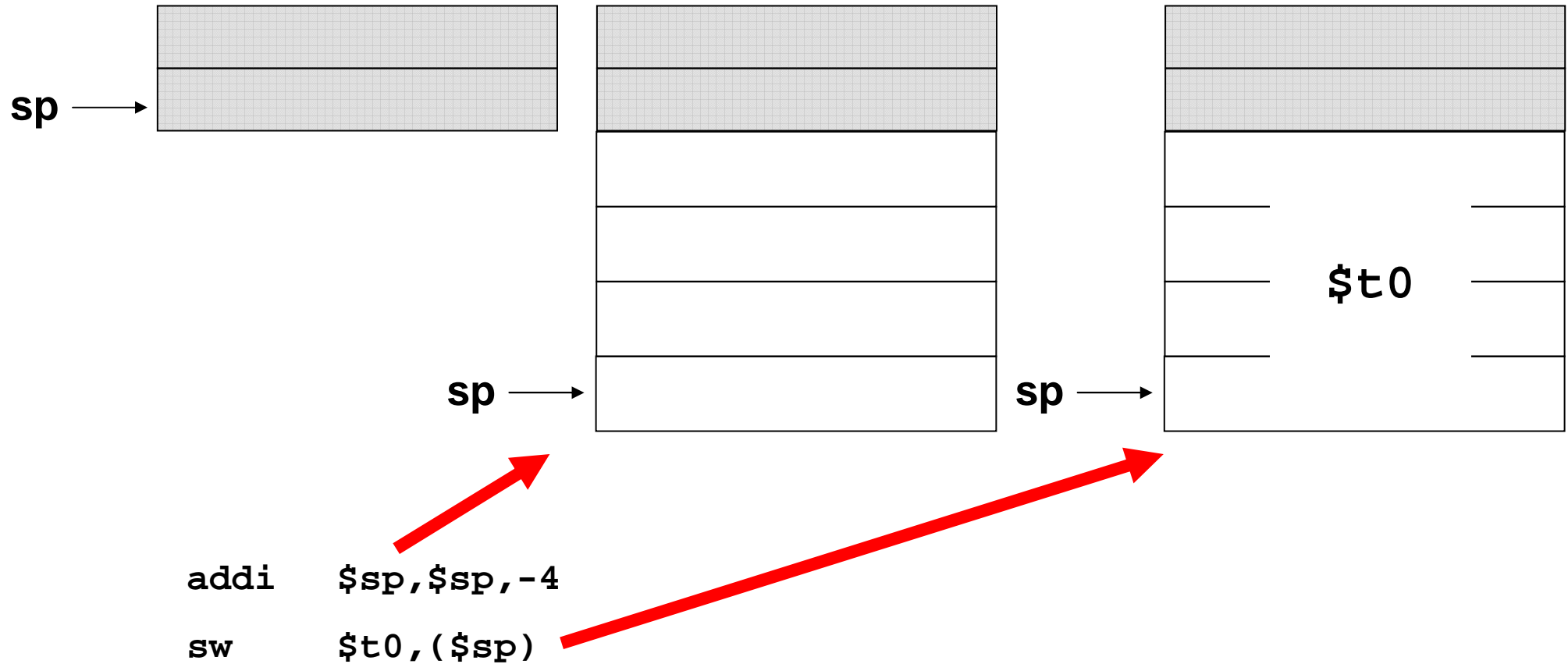
Allocazione di memoria del MIPS



Lo stack

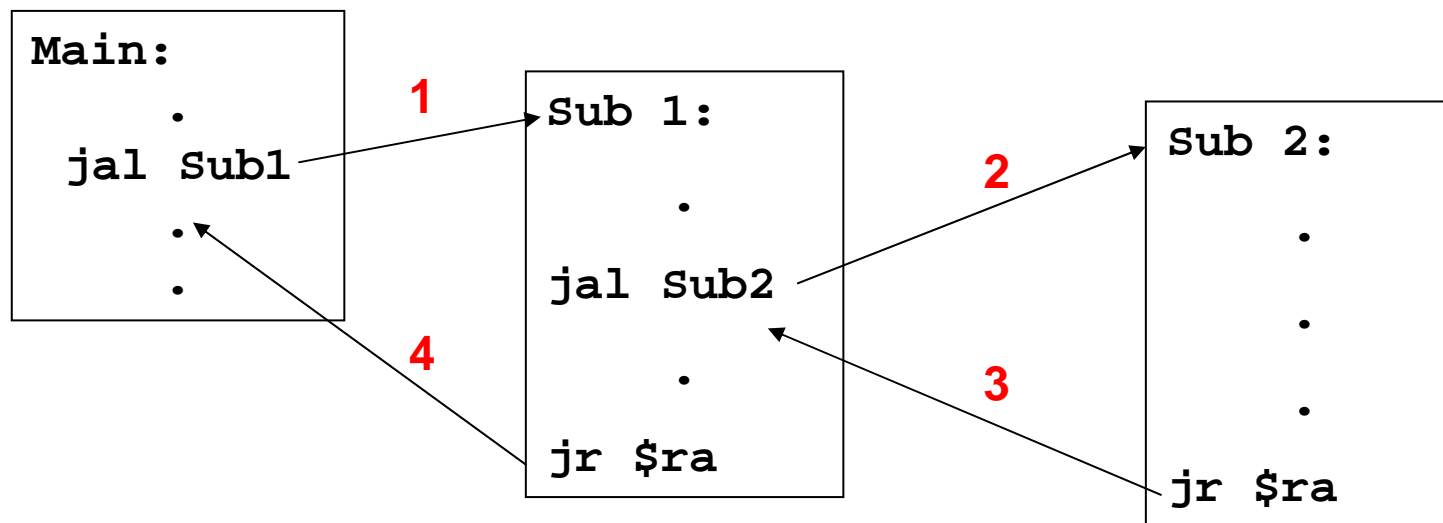
- Il registro $\$sp$ (stack pointer) punta sempre all'ultimo spazio usato nello stack.
- L'area di stack si espande per valori decrescenti di $\$sp$.
- Per allocare uno spazio sullo stack è quindi necessario decrementare $\$sp$ del numero di byte necessari e quindi riempire l'area allocata con i dati.

Lo stack



Lo stack

- E' una area di memoria gestita in modalit  LIFO (Last In First Out)
- E' coerente con la sequenza di attivazione e disattivazione dei sottoprogrammi innestati



Impiego dello stack nelle chiamate innestate

```
sumq:  addi $sp,$sp,-8  # alloca spazio per 2 word sullo stack
        sw $a1,4($sp)   # salva il secondo par. effettivo
        sw $ra,($sp)    # salva l'indirizzo di ritorno

        move $a1,$a0    # prepara i par. effettivi 1a chiamata
        jal mply        # invoca mply
        move $t0,$v0    # salva il valore restituito

        lw $a0,4($sp)   # prepara i par. effettivi 2a chiamata
        move $a1,$a0    # prepara i par. effettivi 2a chiamata
        jal mply        # invoca mply
        add $v0,$t0,$v0 # calcola il valore da restituire

        lw $ra,($sp)    # ripristina l'indirizzo di ritorno
        addi $sp,$sp,8  # dealloca l'area sullo stack
        jr $ra         # Ritorna al chiamante
```

Prologo & Epilogo

- **Prologo**

- Insieme di operazioni compiute dal sottoprogramma, prima dell'esecuzione delle istruzioni proprie, per salvare i valori di registri eventualmente modificati durante l'esecuzione.

- **Epilogo**

- Operazioni compiute dal sottoprogramma, prima del ritorno al chiamante, necessarie per ripristinare il contesto corretto.

Prologo & Epilogo

PROLOGO

```
sumq:  addi $sp,$sp,-8  # alloca spazio per 2 word sullo stack  
       sw  $a1,4($sp)  # salva il secondo par. effettivo  
       sw  $ra,($sp)   # salva l'indirizzo di ritorno
```

```
       move $a1,$a0    # prepara i par. effettivi 1a chiamata  
       jal  mply       # invoca mply  
       move $t0,$v0   # salva il valore restituito
```

```
       lw  $a0,4($sp)  # prepara i par. effettivi 2a chiamata  
       move $a1,$a0   # prepara i par. effettivi 2a chiamata  
       jal  mply       # invoca mply  
       add  $v0,$t0,$v0 # calcola il valore da restituire
```

```
       lw  $ra,($sp)   # ripristina l'indirizzo di ritorno  
       addi $sp,$sp,8  # dealloca l'area sullo stack  
       jr  $ra        # Ritorna al chiamante
```

EPILOGO

Prologo & Epilogo

chiamante

istruzioni precedenti
salvataggio dei parametri
salvataggio dell'indirizzo di ritorno
salto
ripristino
istruzioni seguenti

prologo
istruzioni della procedura
epilogo
salto

chiamato