Il linguaggio assembly – Parte III

Architettura
degli Elaboratori e delle Reti

Alberto Borghese
Università degli Studi di Milano
Dipartimento di Scienze dell'Informazione
email: borghese@dsi.unimi.it

Sommario

- Introduzione a SPIM
- Direttive
- Chiamate di sistema (system call)
- Chiamate a procedura

Il simulatore SPIM

- **SPIM: A MIPS R2000/R3000 Simulator : PCSPIM version 6.3**
- http://www.cs.wisc.edu/~larus/spim.html
- Piattaforme:
  - Unix or Linux system
  - Microsoft Windows (Windows 95, 98, NT, 2000)
  - Microsoft DOS

Esempio:

```
int main(int argc, char* argv[])
 { int i,n,t = 0; for (i=0;i<n;i++) t +=i*i; return(0); }
```


Codice assembly

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

.data
.align 2
str:
        .asciiz "La soma da 0 a N vale "
.text
.globl main

main:

        li      $t1, 2    # N interi di cui calcolare la somma dei
                                  # quadrati
        li      $t6, 0    # t6 e' indice di ciclo indice di ciclo
        li      $t8, 0    # t8 contiene la somma dei quadrati
Loop:
        mul     $t7, $t6 $t6        # t7 = t6 x t6
        addu    $t8, $t8, $t7       # t9 = t8 + t7
        addi    $t6, $t6, 1
        ble     $t6, $t1, Loop      # if t6 < N stay in loop

        la      $a0, str
        li      $v0, 4              #print
        syscall

        li      $v0, 1              #print
        add     $a0, $t8, $zero
        syscall

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

Sommario

- Introduzione a SPIM
- Direttive
- Chiamate di sistema (system call)
- Chiamate a procedura

Direttive

- Le direttive (data layout directives) danno delle indicazioni all'assemblatore sul contenuto di un file (istruzioni, strutture dati, ecc.)
- Sintatticamente le direttive iniziano tutte con il carattere "."

Direttive

`.data <addr>`

- Gli elementi successivi sono memorizzati nel segmento dati a partire dall'indirizzo **`addr, facoltativo.`**

`.asciiz str`

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

`.byte b1,…,bn`

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

Direttive

`.word w1, ..,wn`

- Memorizza gli **n** valori su 32-bit **`w1, .., wn`** in parole 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** byte nel segmento dati

SEGMENTO TESTO

`.text <addr>`

- Memorizza gli elementi successivi nel segmento testo dell'utente a partire dall'indirizzo **`addr.`** (Questi elementi possono essere solo istruzioni o parole).

Direttive

`.globl sym`

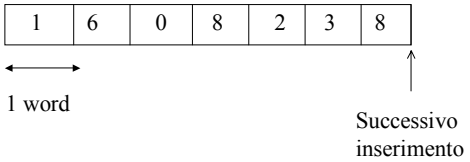
- Dichiara **sym** come etichetta globale (ad essa è possibile fare riferimento da altri file)

`.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`**, e **`.double`** fino a quando compare la successiva direttiva **`.data`**

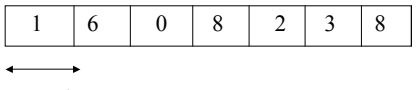
.word: esempio

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



.word + etichetta: esempio

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



⇒ array rappresenta l'indirizzo del primo elemento

Codice assembly

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

.data
.align 2
str:
        .asciiz "La soma da 0 a N vale "
.text
.globl main

main:

        li      $t1, 2    # N interi di cui calcolare la somma dei
                                  # quadrati
        li      $t6, 0    # t6 e' indice di ciclo indice di ciclo
        li      $t8, 0    # t8 contiene la somma dei quadrati
Loop:
        mul     $t7, $t6 $t6        # t7 = t6 x t6
        addu    $t8, $t8, $t7       # t9 = t8 + t7
        addi    $t6, $t6, 1
        ble     $t6, $t1, Loop      # if t6 < N stay in loop

        la      $a0, str
        li      $v0, 4              #print
        syscall

        li      $v0, 1              #print
        add     $a0, $t8, $zero
        syscall

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

Sommario

- Introduzione a SPIM
- Direttive
- Chiamate di sistema (system call)
- Chiamate a procedura

System call

- Sono disponibili delle **chiamate di sistema (system call)** predefinite che implementano particolari servizi (ad esempio: stampa a video)
- Ogni system call ha:
  - un codice
  - degli argomenti (opzionali)
  - dei valori di ritorno (opzionali)

System call

- **`print_int:`** stampa sulla console il numero intero che le viene passato come argomento;
- **`print_float:`** stampa sulla console il numero in virgola mobile con singola precisione che le viene passato come argomento;
- **`print_double:`** stampa sulla console il numero in virgola mobile con doppia precisione che le viene passato come argomento;
- **`print_string:`** stampa sulla console la stringa che le è stata passata come argomento terminandola con il carattere **`Null`**;

System call

- **`read_int:`** legge una linea in ingresso fino al carattere a capo incluso (i caratteri che seguono il numero sono ignorati);
- **`read_float:`** leggono una linea in ingresso fino al carattere a capo incluso (i caratteri che seguono il numero sono ignorati);
- **`read_double:`** leggono una linea in ingresso fino al carattere a capo incluso (i caratteri che seguono il numero sono ignorati);
- **`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)`**;

System call

- **sbrk** restituisce il puntatore (indirizzo) ad un blocco di memoria;
- **exit** interrompe l'esecuzione di un programma;

System call

Nome	Codice ($v0)	Argomenti	Risultato
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		

System call

- 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 - $f15** nel caso di valori in virgola mobile)
  - Eseguire **syscall**
  - L'eventuale **valore di ritorno** è caricato nel registro **$v0 ($f0)**

Esempio

```
#Programma che stampa: la risposta è 5
      .data
str:  .asciiz "la risposta è "
      .text
      .globl main

Main:
      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

```
#Programma che stampa "Dammi un intero: "
# e che legge un intero
       .data
prompt:.asciiz "Dammi un intero: "
       .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

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


Esempio, fattoriale

```
# Programma che calcola il fattoriale iterativamente
       .data
prompt: .asciiz "Inserisci un numero intero"
output: .ascii "Il fattoriale 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 l'intero e lo carica in $v0
```

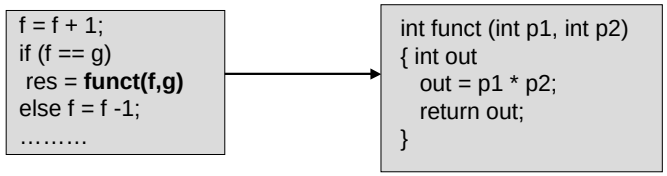

Esempio, fattoriale (calcolo)

```
# calcola il fattoriale

       li $t0, 1    # inizializzo $t0 contatore cicli
       li $t1, 1    # inizializzo $t1 accumulatore
                    # che conterrà il risultato n!
loop:
       mul $t1, $t1, $t0   # $t1 ← $t1 * $t0
       addi $t0, $t0, 1    # incremento $t0 contatore cicli
       ble  $t0, $v0, loop # se $t0 ≤ $v0 go to loop
```


Esempio, fattoriale (stampa risultato)

```
# stampa il risultato

       move $a0, $v0       # $a0 ← $v0
       li $v0, 1           # $v0 ← codice della print_int
       syscall             # stampa l'intero in input

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

       li $v0, 1           # $v0 ← codice della print_int
       move $a0, $t1       # $a0 ← n!
       syscall             # stampa n!

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

Chiamata a procedura: esempio



Chiamata a procedura

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

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

Chiamata a procedura

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

Allocazione dei registri

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

Chiamata a procedura

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

Riassumendo

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

Problemi

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

utilizzo dello stack

Lo stack

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

Gestione dello stack nel MIPS

- Lo stack cresce **da indirizzi di memoria alti verso indirizzi bassi**
- Il registro **$sp** contiene l'indirizzo della prima locazione libera in cima allo stack.
- L'inserimento di un dato nello stack **(operazione di push)** avviene **decrementando $sp** per allocare lo spazio e si esegue sw per inserire il dato.
- Il prelevamento di un dato dallo stack **(operazione di pop)** avviene **incrementando $sp** (per eliminare il dato) e riducendo quindi la dimensione dello stack.

Gestione dello stack nel MIPS

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

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


Gestione dello stack nel MIPS

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

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

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

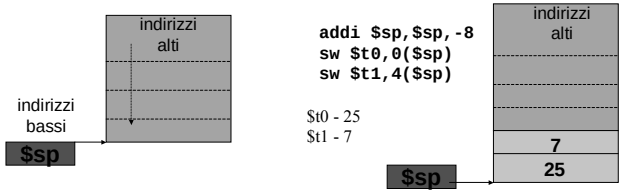
Gestione dello stack nel MIPS

- Per inserire elementi nello stack

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

- Per recuperare elementi dallo stack

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



Lo stack

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

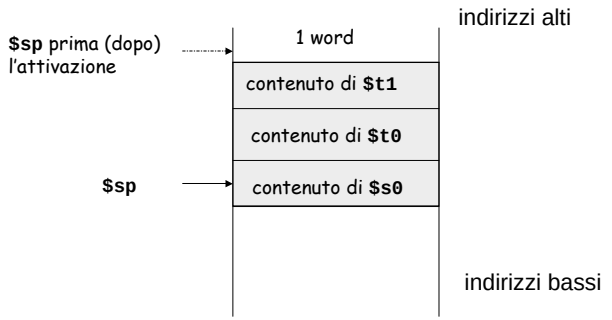
Esempio

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

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


Esempio

```
# g,h,i e j associati a $a0, …, $a3;
# f associata ad $s0
# il calcolo di f richiede 3 registi: $s0, $t0, $t1
# necessario salvare i 3 registri nello stack


example:
  addi $sp,$sp,-12      # alloca nello stack
                        # lo spazio per i 3 registri
  sw $t1, 8($sp)        # salvataggio di $t1
  sw $t0, 4($sp)        # salvataggio di $t0
  sw $s0, 0($sp)        # salvataggio di $s0
```


Esempio (cont.)



Esempio (cont.)

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

add $v0, $s0, $zero        # restituisce f copiandolo
                           # nel reg. di ritorno $v0

# ripristino del vecchio contenuto dei registri
# estraendoli dallo stack
lw $s0, 0($sp)             # ripristino di $s0
lw $t0, 4($sp)             # ripristino di $t0
lw $t1, 8($sp)             # ripristino di $t1

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

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


Lo stack

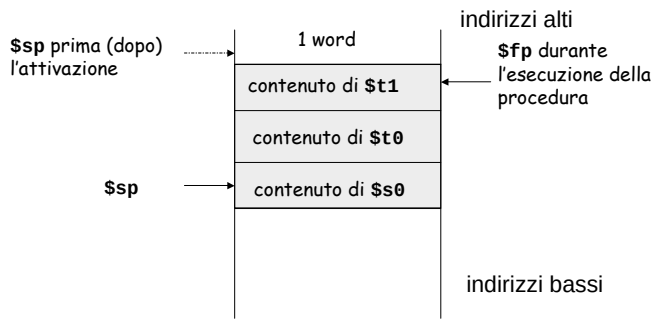
- Per evitare di salvare inutilmente il contenuto dei registri, i registri sono divisi in due classi:
  - **registri temporanei**: **$t0, …, $t9 ($f4, .. $f11, $f16, .., $f19)** il cui contenuto ***non è salvato*** dal chiamato nello stack;
  - **registri non-temporanei**: **$s0, …, $s8 ($f20, …, $f31)** il cui contenuto è ***salvato*** nello stack ***se utilizzati dal chiamato***.
- Nell'esempio precedente: dato che il chiamante non si aspetta che **$t0** e **$t1** siano preservati durante la chiamata a procedura, si possono eliminare due store e due load.
  E' necessario salvare e ripristinare **$s0** perché il chiamante ha bisogno del suo valore originale.
- Si può ovviare anche a ciò, se si elimina l'utilizzo di $s0:

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

può diventare:

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


Il frame pointer



Uso dei registri: convenzioni

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

Uso dei registri: convenzioni

Registri usati per le operazioni floating point

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

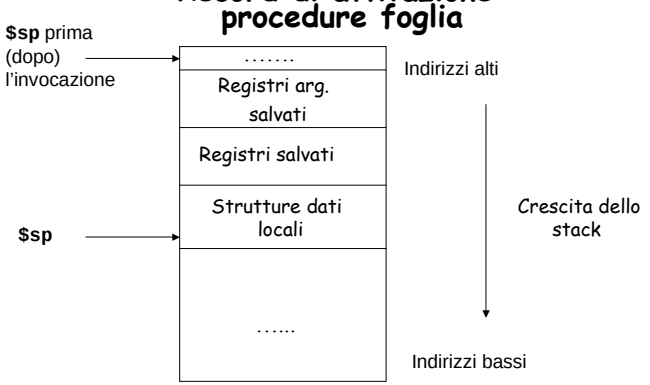
Procedure foglia

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

Procedure foglia

- Nel caso di procedure foglia, il **chiamato** alloca nello stack:
  - I registri non temporanei che vuole utilizzare **(`$s0-$s8`)**
  - Strutture dati locali (es: array, matrici) e variabili locali della procedura che non stanno nei registri.
- Lo stack pointer **`$sp`** è aggiornato per tener conto del numero di registri memorizzati nello stack; alla fine i registri vengono ripristinati e lo stack pointer riaggiornato.

Record di attivazione: procedure foglia



Record di attivazione

- Una procedura è eseguita in uno spazio *privato* detto record di attivazione
  - area di memoria dove vengono allocate le variabili locali della procedura e i parametri
- Il programmatore assembly deve provvedere esplicitamente ad allocare/cedere lo spazio necessario (*frame di chiamata a procedura*) per:
  - Mantenere i valori passati come parametri alla procedura;
  - Salvare i registriche una procedura potrebbe modificare ma che al chiamante servono inalterati.
  - Fornire spazio per le variabili locali alla procedura.
- Quando sono permesse chiamate di procedura annidate, i record di attivazione sono allocati e rimossi come gli elementi di uno stack

Salvataggio dell'ambiente

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

Convenzioni per il salvataggio dell'ambiente

- Convenzione **del MIPS**
  - per ottimizzare il numero di accessi alla memoria, il chiamante e il chiamato salvano solo i registri di un particolare gruppo
  - il **chiamante**, se vuole che siano preservati, salva i registri di **temporanei `$t0-$t9 ($f4-$f11, $f16-$f19)`**.
  - il **chiamato** salva sullo stack **`$ra;`** se li usa, i registri **di variabile** $s0-$s8 ($f20-$f31), se utilizzati; i registri argomento $a0-$a3 ($f12-$f15), se servono dopo la chiamata a procedura; ed eventuali argomenti aggiuntivi **e s**trutture dati locali (es: array, matrici) e variabili locali.

Struttura di una procedura

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