System Calls, Register Spilling, Introduzione al controllo di flusso

nicola.basilico@unimi.it
http://homes.di.unimi.it/basilico/teaching/

Chiamate a Servizi di Sistema

- Una system call permette di utilizzare servizi la cui esecuzione è a carico del sistema operativo (tipicamente operazioni di I/O).

- Ogni servizio è associato ad un codice univoco (un valore intero).

- Come si invoca una System Call?
  - Caricare il codice identificativo nel registro $v0.
  - Caricare gli argomenti necessari nei registri $a0, $a1, $a2, $a3, $f12.
  - Eseguire istruzione syscall.
  - Leggere eventuali valori di ritorno in $v0, $f0.

Chiamate a Servizi di Sistema

Service	System Call Code	Arguments	Result
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = string	
read_int	5		integer (in $v0)
read_float	6		float (in $f0)
read_double	7		double (in $f0)
read_string	8	$a0 = buffer, $a1 = length	
sbrk	9	$a0 = amount	address (in $v0)
exit	10		

Esercizio 3.1

- Si scriva codice assembly che:
  - Chieda all'utente di inserire un intero (messaggio su terminale).
  - Acquisisca un intero da terminale.
  - Calcoli l'intero successivo.
  - Mostri all'utente il risultato (messaggio su terminale).

Esercizio 3.1

- Si scriva codice assembly che:
  - Chieda all'utente di inserire un intero (messaggio su terminale).
  - Acquisisca un intero da terminale.
  - Calcoli l'intero successivo.
  - Mostri all'utente il risultato (messaggio su terminale).

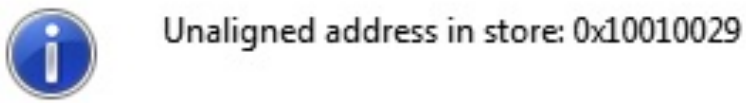
```
main:
.data
string1: .asciiz "Inserire numero intero: "
string2: .asciiz "L'intero successivo è: "

.text
li $v0, 4                   # selezione di print_string (codice = 4)
la $a0, string1             # $a0 = indirizzo di string1
syscall                     # lancio print_string
li $v0, 5                   # Selezione read_int (codice = 5)
syscall                     # lancio read_int
add $t0, $zero, $v0         # leggo risultato $t0 = $v0
li $v0, 4                   # selezione di print_string
la $a0, string2             # $a0 = indirizzo di string2
syscall                     # lancio print_string
addi $t0, $t0, 1            # $t0 += 1
li $v0, 1                   # selezione di print_int (codice = 1)
add $a0,$zero,$t0           # $a0 = $t0
syscall                     # lancio print_int
jr $ra
```

Esercizio 3.2

- Si scriva codice assembly che:
  - Chieda all'utente di inserire un intero (messaggio su terminale).
  - Acquisisca un intero da terminale.
  - Calcoli l'intero successivo.
  - Memorizzi l'intero ed il successivo in un array di dimensione 2, residente in memoria.
  - Mostri all'utente i due numeri (messaggio su terminale).

Problema dell'allineamento



- Le istruzioni di load e store richiedono di operare su **dati allineati**.
- Cosa significa? Un dato è allineato se il suo indirizzo in memoria è multiplo della sua dimensione in bytes.
- Esempio: una parola (4 bytes) è allineata se il suo indirizzo è multiplo di 4.
- Quando allochiamo bytes in un segmento di memoria possiamo chiedere ad assembler di farlo mantenendo un allineamento specificato.
- Direttiva .align n: il prossimo dato va allineato con un indirizzo multiplo di 2^n.

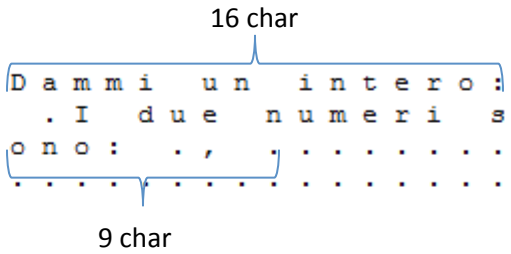
Problema dell'allineamento

```
.data
string1: .asciiz "Dammi un intero: "
string2: .asciiz "I due numeri sono: "
string3: .asciiz ", "

.align 2
array: .space 8
```

- Snapshot della memoria:

```
User data segment [10000000]..[10040000]
[10000000]..[1000ffff]  00000000
[10010000]    1835884868  1853169769  1953392928  0980382309
[10010010]    0541655072  0543520100  1701672302  1931503986
[10010020]    0980381295  0539754528  0000000000  0000000012
[10010030]    0000000013  0000000000  0000000000  0000000000
[10010040]..[1003ffff]  00000000
```



- Se togliessimo il terminatore a string3?

Register Spilling

- In generale, un programma ha un numero di variabili maggiore rispetto al numero di registri: non è possibile avere tutti i dati nel banco registri.
- Tecnica di utilizzo efficiente: mantenere nei registri le variabili usate più frequentemente e il resto in memoria.
- Trasferire variabili poco utilizzate da registri a memoria è **register spilling**.
- L'area di memoria di solito utilizzata per questa operazione è lo stack.

Esercizio 3.3

- Si supponga di poter usare solo i registri $s0 e $t0.

- Si scriva il codice assembly che:
  - calcoli la somma dei primi tre numeri interi positivi (1, 2 e 3), ciascuno moltiplicato per 3;
  - non utilizzi la pseudo-istruzione muli.

Esercizio 3.3

- Si supponga di poter usare solo i registri $s0 e $t0.

- Si scriva il codice assembly che:
  - calcoli la somma dei primi tre numeri interi positivi (1, 2 e 3), ciascuno moltiplicato per 3;
  - non utilizzi la pseudo-istruzione muli.

```
main:

li $s0, 0
addi $sp, $sp, -4          # faccio spazio sullo stack
sw $s0, 0($sp)             # salvo $s0 sullo stack (push)
li $s0, 1
li $t0, 3
mult $s0, $t0
mflo $t0
lw $s0, 0($sp)             # leggo dallo stack
addi $sp, $sp, 4           # aggiorno lo stack pointer (pop)
add $s0, $s0, $t0
```

Controllo di flusso

- Istruzioni (branch e jump) che alterano l'ordine sequenziale di esecuzione delle istruzioni di un programma.

- Usate per tradurre in assembly codice di alto livello che fa uso di strutture di controllo come verifica di condizioni, salti non condizionati o cicli.

- Producono un effetto sull'aggiornamento del program counter: la prossima istruzione da eseguire potrà non essere quella all'indirizzo successivo (+4), ma quella ad un indirizzo specificato.

Unconditional Jump

- **Unconditional:** Il salto viene sempre eseguito.
- Esempi: j (jump) e jr (jump register) e jal (jump and link)

```
j   label   # go to label
jr  $r31    # go to indirizzo contenuto in $r31
jal label   # go to label, salva PC in $ra
```

Conditional Branch

- **conditional:** il salto viene eseguito solo se una certa condizione risulta verificata.
- Esempi: **`beq`** (*branch on equal*) e **`bne`** (*branch on not equal*)

```
beq $rs, rt, L1      # if (rs == rt) go to L1
bne $rs, rt, L1      # if (rs != rt) go to L1
```

If … Then

Codice C:

```
if (i==j)
      f=g+h;
…
```

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

If … Then

Codice C:

```
if (i==j)
      f=g+h;
…
```

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

- Risccriviamo il codice C in una forma equivalente, ma più «vicina» alla sua traduzione Assembly

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

Codice Assembly:

```
bne $s3, $s4, L    # if i ≠ j go to L
add $s0, $s1, $s2
L:
…
```

If … Then … Else

Codice C:

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

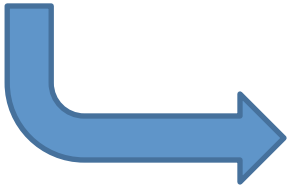
Si supponga che le variabili f, g, h, i e j siano associate rispettivamente ai registri $s0, $s1, $s2, $s3 e $s4

If … Then … Else

Codice C:

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

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



Codice Assembly:

```
bne $s3, $s4, Else
add $s0, $s1, $s2
j End
Else:
sub $s0, $s1, $s2
End:
…
```

Do … While

Codice C:

```
i=0;
do{
      g = g + A[i];
      i = i + j;
}
while (i!=h);
```

Si supponga che:

g e h siano in $s1, $s2

i e j siano in $s3, $s4

A sia in $s5

Do … While

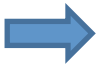
Codice C:

```
i=0;
do{
      g = g + A[i];
      i = i + j;
}
while (i!=h);
```

Si supponga che:
g e h siano in $s1, $s2
i e j siano in $s3, $s4
A sia in $s5

- Riscriviamo il codice C:

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



Codice Assembly:

```
li $s3, 0
Loop:
muli $t1, $s3, 4
add $t1, $t1, $s5
lw $t0, 0($t1)
add $s1, $s1, $t0
add $s3, $s3, $s4
bne $s3, $s2, Loop
```

While

Codice C:

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

Si supponga che:
i e j siano in $s3, $s4
k sia in $s5
A sia in $s6

While

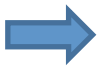
Codice C:

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

Si supponga che:
i e j siano in $s3, $s4
k sia in $s5
A sia in $s6

- Riscriviamo il codice C:

```
Loop:
If (A[i]!=k)
    go to End;
i=i+j;
go to Loop;
```



Codice Assembly:

```
Loop:
muli $t1, $s3, 4
add  $t1, $t1, $s6
lw  $t0, 0($t1)
bne $t0, $s5, End
add $s3, $s3, $s4
j Loop
End:
```

Condizioni di disuguaglianza

- Spesso è utile condizionare l'esecuzione di un'istruzione al fatto che una variabile sia minore di un'altra, istruzione **Set Less Than**.

```
slt $s1, $s2, $s3
```

- Assegna il valore 1 a $s1 se $s2 < $s3 altrimenti assegna il valore 0.
- Con slt, beq e bne si possono implementare tutti i test sui valori di due variabili (=, !=, <, <=, >,>=).

Esempio

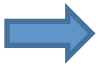
Codice C:

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

Si supponga che:
i e j siano in $s0, $s1
k sia in $s2

- Riscriviamo il codice C:

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



Codice Assembly:

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

Condizioni di disuguaglianza

- slt e beq come nell'esempio precedente possono essere usate tramite pseudo-istruzioni, ad esempio **Branch on Greater Than**.

bgt $s1, s2, label

- Salta a label se $s1 > s2

- Altre pseudo-istruzioni simili:

bge, blt, ble

Il costrutto switch

- Può essere implementato con una serie di if-then-else
- *Alternativa: uso di una jump address table (prossime lezioni)*

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;
}
```

```
if (k < 0)
      t = 1;
else
      t = 0;
if (t == 1) // k < 0
      goto Exit;
t2 = k;
if (t2 == 0) // k = 0
      goto L0;
t2--; if (t2 == 0) // k = 1;
      goto L1;
t2--; if (t2 == 0) // k = 2;
      goto L2;
t2--; if (t2 == 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:
```

Il costrutto switch

- Si supponga che $s0, …, $s5 contengano f,g,h,i,j,k,

Codice Assembly:

```
slt $t3, $s5, $zero
bne $t3, $zero, Exit

beq $s5, $zero, L0

addi $s5, $s5, -1
beq $s5, $zero, L1

addi $s5, $s5, -1
beq $s5, $zero, L2

addi $s5, $s5, -1
beq $s5, $zero, L3
```

```
j Exit;

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:
```