System Calls, Register Spilling

UNIVERSITÀ DEGLI STUDI DI MILANO
matteo.re@unimi.it
http://homes.di.unimi.it/re/arch2-lab-2015-2016.html

Esercizio 3.1

- Eseguire il seguente codice assembly e descrivere quello che fa.

```
.data
v: .byte 2,0,0,0,4,0,0,0
array: .byte 2,0,0,0,3,0,0,0,5,0,0,0,7,0,0,0,11,0,0,0,13,0,0,0,17,0,0,0,19,0,0,0
```

```
.text
main:
la $s1, array
la $s2, v
lw $t0, 0($s2)
addi $t0, $t0, -1

mulo $t0, $t0, 4
add $t1, $s1, $t0

lw $t2, 0($t1)
addi $t2, $t2, 1
```

```
lw $t0, 4($s2)
addi  $t0,  $t0,  -1

mulo $t0, $t0, 4
add  $t3,  $s1,  $t0

lw $t4, 0($t3)
addi $t4, $t4, -1


sw $t2, 0($t3)
sw $t4, 0($t1)

jr $ra
```

Esercizio 3.1

- Eseguire il seguente codice assembly e descrivere quello che fa.

```
.data
v: .byte 2,0,0,0,4,0,0,0
array: .byte 2,0,0,0,3,0,0,0,5,0,0,0,7,0,0,0,11,0,0,0,13,0,0,0,17,0,0,0,19,0,0,0
```

```
.text
main:
la $s1, array # carico gli indirizzi
la $s2, v
lw $t0, 0($s2) # load valore 2
addi $t0, $t0, -1 # sottraggo 1 perchè indirizzerà il secondo elemento
mulo $t0, $t0, 4 # offset (in bytes)
add $t1, $s1, $t0 # indirizzo secondo elemento array (base + offset)
lw $t2, 0($t1) # load secondo elemento array
addi $t2, $t2, 1 # incremento secondo elemento array
```

```
lw $t0, 4($s2) # load valore 4
addi $t0, $t0, -1 # sottraggo 1 perché indirizzerà il quarto elemento
mulo $t0, $t0, 4 # offset (in bytes)
add $t3, $s1, $t0 # indirizzo quarto elemento array (base + offset)
lw $t4, 0($t3) # load quarto elemento array
addi $t4, $t4, -1 # decremento quarto elemento array


sw $t2, 0($t3)  # store in posizioni scambiate
sw $t4, 0($t1)

jr $ra
```

In pseudocodice:

```
temp = array[v[0]-1];
array[v[0]-1]          =
array[v[1]-1] - 1;
array[v[1]-1]  =  temp  +
1;
```

System Calls

- Come possiamo fare I/O in assembly?
- **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 numerico univoco.
- Come si invoca una system call che ha codice **K**?
  - Caricare **K** nel registro `$v0`;
  - caricare gli argomenti nei registri `$a0, $a1, $a2, $a3, $f12` (opzionale);
  - eseguire istruzione `syscall`;
  - leggere eventuali valori di ritorno nei registri `$v0, $f0` (opzionale).

System Calls

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		

sbrk, varia dinamicamente la dimensione del segmento dati modificando il *valore di break* assegnato dal sistema operativo (il primo indirizzo oltre il limite dell'area dati). E' usata quando, ad esempio, il programma invoca una `malloc`

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;
  - mostri all'utente il risultato (messaggio su terminale).

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;
  - 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
```

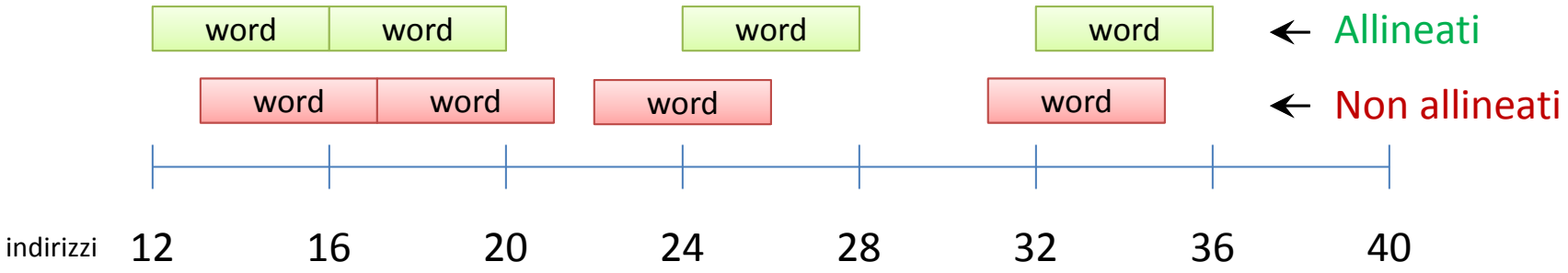
Esercizio 3.3

- 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 in memoria;
  - mostri all'utente i due numeri (messaggio su terminale).

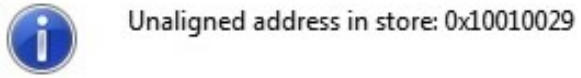
Allineamento dati

- Accesso di memoria allineato:
  - ogni dato ha dimensione n bytes;
  - l'indirizzo di ogni dato è multiplo di n;
  - n è una potenza di 2.

- Nel nostro caso:
  - un dato corrisponde a una word e ha dimensione di 32 bit (n = 4);
  - l'indirizzo di una word deve essere multiplo di 4.



- Le istruzioni di `sw` e `lw` richiedono di operare con accesso allineato con parole da 32 bit, quindi se specifichiamo un indirizzo **non** multiplo di 4:



Allineamento dati

- 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.

```
.data
string: .asciiz "stringa di prova"
array: .byte 5,0,0,0,13,0,0,0
```

```
User data segment [10000000]..[10040000]
[10000000]..[1000ffff]  00000000
[10010000]    1769108595  0543254382  1881172324  1635151730    s t r i n g a   d i   p r o v a
[10010010]    0000000005  0000000000  0000000000  0000000000    . . . . . . . . . . . . . . . .
[10010020]..[1003ffff]  00000000
```

```
.data
string: .asciiz "stringa di prova"
.align 2
array: .byte 5,0,0,0,13,0,0,0
```

```
User data segment [10000000]..[10040000]
[10000000]..[1000ffff]  00000000
[10010000]    1769108595  0543254382  1881172324  1635151730    s t r i n g a   d i   p r o v a
[10010010]    0000000005  0000000000  0000000000  0000000000    . . . . . . . . . . . . . . . .
[10010020]..[1003ffff]  00000000
```

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.

Uso dello Stack

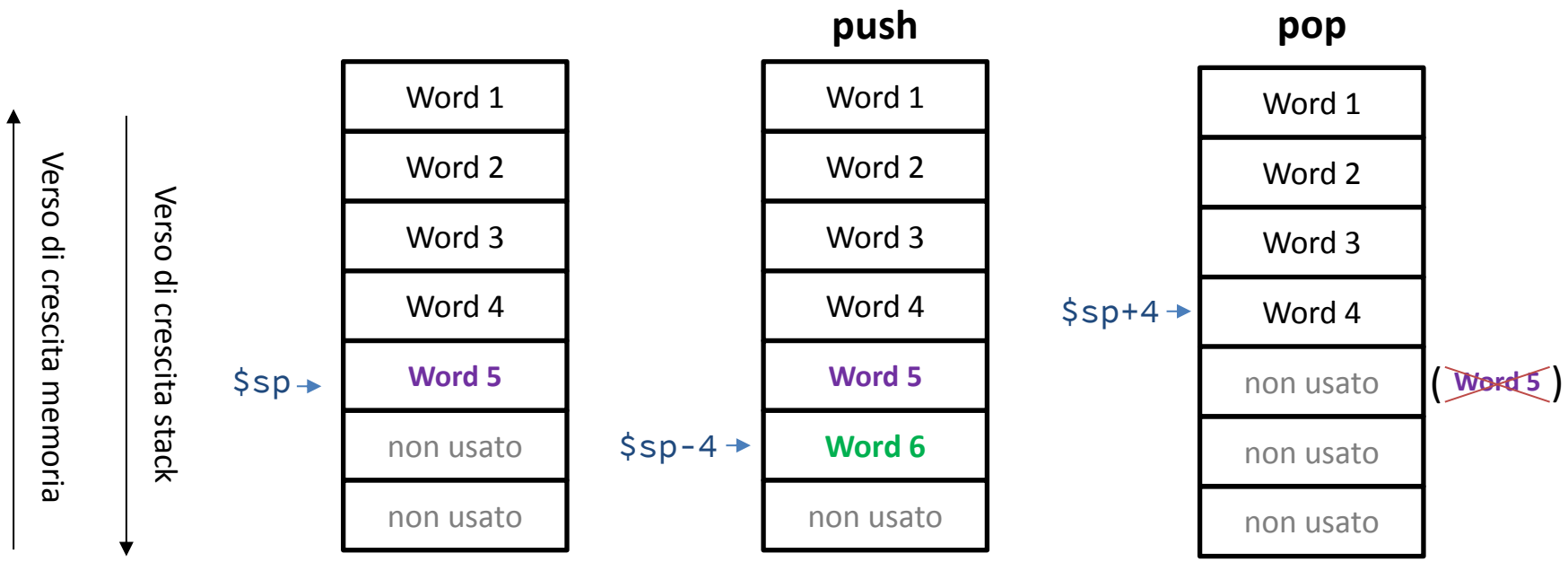
- Il registro $sp (stack pointer) contiene l'indirizzo della parola che sta in cima allo stack (l'ultima locazione in uso, la prima che sarà utilizzata).

- Per aggiungere un **dato** (push):

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

- Per consumare un **dato** (pop):

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



Esercizio 3.4

- 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 si utilizzi la pseudo-istruzione `mul`.

Esercizio 3.4

- 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 si utilizzi la pseudo-istruzione `mul`.

```
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
```

Esercizio 3.5

- Si modifichi il codice dell'esercizio 3.1 in modo che:
  - chieda all'utente di inserire i due valori in v e gli otto in array;
  - effettui le operazioni di 3.1;
  - stampi un messaggio con i valori scambiati in array (prima e dopo la modifica).

Esercizio 3.6

- Si scriva il codice assembly che:
  - inizializzi il segmento dati con un array di 13 interi scelti a piacere;
  - chieda all'utente di inserire tre interi `a`, `b` e `c`;
  - se `c=0`, scambia l'`a`-esimo elemento dell'array con il `b`-esimo;
  - se `c=1`, sovrascrive il `b`-esimo elemento dell'array con il valore dell'`a`-esimo elemento;
  - se `c=-1`, sovrascrive l'`a`-esimo elemento dell'array con il valore del `b`-esimo elemento;
  - se `c` ha un altro valore stampi la stringa `«comando non riconosciuto»` e non effettui modifiche all'array.