03: Laboratorio 02

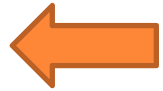
Istruzioni di lettura e scrittura della memoria, Array
Register spilling
Direttive assembly
I/O

I. Frosio

25 March 2011

SOMMARIO

- Istruzioni di lettura e scrittura della memoria, Array
- Direttive assembly, allocazione memoria
- System services, I/O
- Register spilling



MIPS & ISTRUZIONI DI ACCESSO ALLA MEMORIA

- Servono istruzioni per trasferire dati da memoria a registri e viceversa.

  - **lw (load word)**, per trasferire una parola di memoria in un registro della CPU. E' necessario specificare il registro destinazione e l'indirizzo di memoria contenente la parola che si vuole copiare.

  - **sw (store word)**, per trasferire il contenuto di un registro della CPU in una parola di memoria E' necessario specificare il reistro che si intende copiare e la destinazione in memoria della parola copiata.

CARICAMENTO DALLA MEMORIA (LOAD)
SCRITTURA IN MEMORIA (STORE)

- Load (word):

  lw $s1, 100($s2) # $s1 ← M[[$s2] + 100]

- La memoria viene indicata sottoforma di vettore M[].
- L'istruzione prende il contenuto della parola che ha inizio in $s2+100 e la copia nel registro $s1. La memoria resta inalterata.

- Store (word):

  sw $s1, 100($s2) # $s1 ← M[[$s2] + 100]

- L'istruzione prende il contenuto del registro $s1 e lo copia nella parola che ha inizio in $s2+100. Il registro resta inalterato.

Es. 2.1

- Si implementi in Assembly la seguente riga di codice C:

  A[12] = h + A[8];

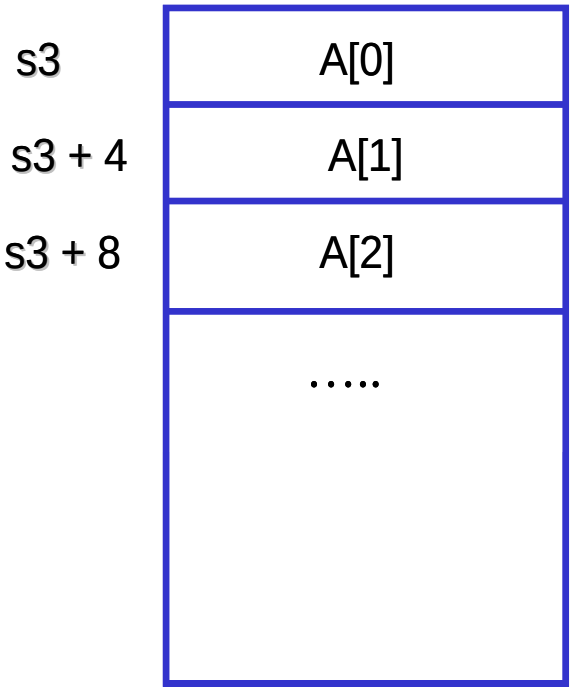
  si supponga che l'indirizzo di base di A sia contenuto nel registro $t0, quello di h in $t1.

- Attenzione: la memoria è indirizzata al singolo byte, ma gli elementi di A sono di 32 bit ciascuno... Quale è l'offset per A[1]?

- Cosa succede se $t0, $t1 non sono inizializzati (pari a 0x00000000 al primo avvio o ad un numero casuale negli altri casi)?

Es. 2.1 – Soluzione (Memorizzazione di un vettore)

- L'elemento numero **i-esimo** di un array si troverà nella locazione `br + 4 * i` dove:
  - **`br`** è il registro base;
  - **`i`** è l'indice ad alto livello;
  - il fattore **`4`** dipende dall'indirizzamento al byte della memoria nel MIPS



A[0]	0	1	2	3
	4	5	6	7
Offset (A[2])	8	9	10	11
	2^k-4	2^k-3	2^k-2	2^k-1

Es. 2.1 – Soluzione (registri non inizializzati)

- main:

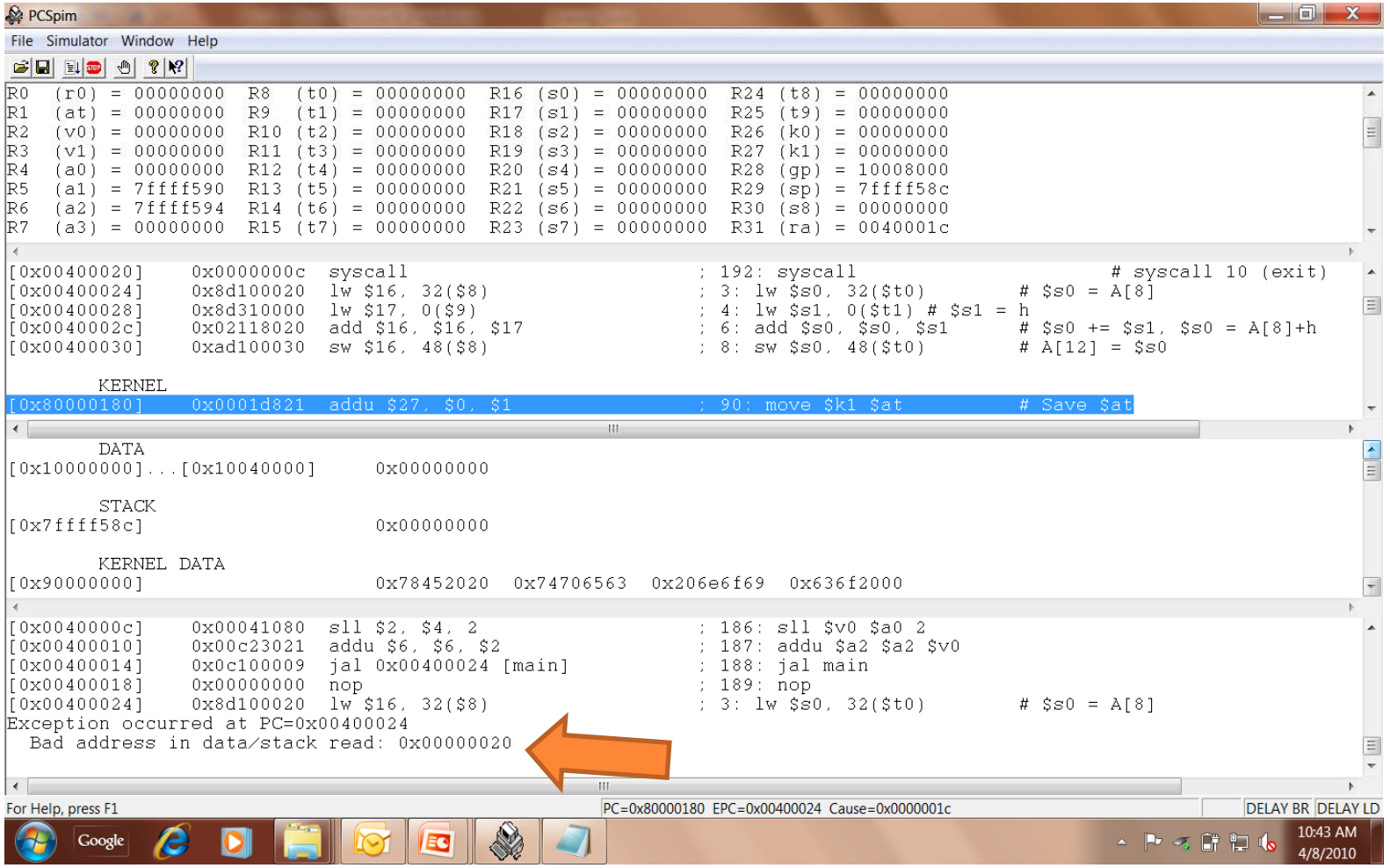
- lw $s0, 32($t0) # $s0 = A[8]
- lw $s1, 0($t1) # $s1 = h

- add $s0, $s0, $s1 # $s0 += $s1, $s0 = A[8]+h

- sw $s0, 48($t0) # A[12] = $s0

- Se osserviamo l'output (Console) ci accorgiamo che viene sollevata un'eccezione [Exception 7, Bad data address] => I registri puntano alla zona di memoria 0x00000000 (e parole successive), dove risiede la porzione di codice.

Es. 2.1 – Soluzione (registri non inizializzati)



Es. 2.1 – Soluzione (registri inizializzati... Da noi)

- main:

- addi $t1, $zero, 0x10000000 # h is at the beginning of the data segment
- addi $t0, $zero, 0x10000004 # A follows h in memory

- lw $s0, 32($t0) # $s0 = A[8]
- lw $s1, 0($t1) # $s1 = h

- add $s0, $s0, $s1 # $s0 += $s1, $s0 = A[8]+h

- sw $s0, 48($t0) # A[12] = $s0

- Questo codice NON FUNZIONA! Quando PCSpim prova a caricare il file .asm, ci avverte che la addi può avere in ingresso numeri al massimo di 16 bit, mentre 0x10000000è a 32 bit.
- L'istruzione del MIPS è a 32 bit, 16 bit sono riservati per la codifica dell'istruzione, restano solo 16 bit per il valore da impostare nel registro!

Es. 2.1 – Soluzione (registri inizializzati… Da noi)

- Pseudo istruzione **la**, serve per caricare un valore a 32 bit in un registro. Prima vengono caricati a 16 bit più significativi (mediante l'istruzione **lui**, load upper immediate), quindi i 16 bit meno significativi (pseudo-istruzione **li**). Esempio:

  la $t1, 0x1a1b3456 # load 0x1a1b3456 into register $t1

- main:

- la $t1, 0x10000000 # h is at the beginning of the data segment
- la $t0, 0x10000004 # A follows h in memory

- lw $s0, 32($t0) # $s0 = A[8]
- lw $s1, 0($t1) # $s1 = h

- add $s0, $s0, $s1 # $s0 += $s1, $s0 = A[8]+h

- sw $s0, 48($t0) # A[12] = $s0

Es. 2.2

- Si traduca la riga di codice:

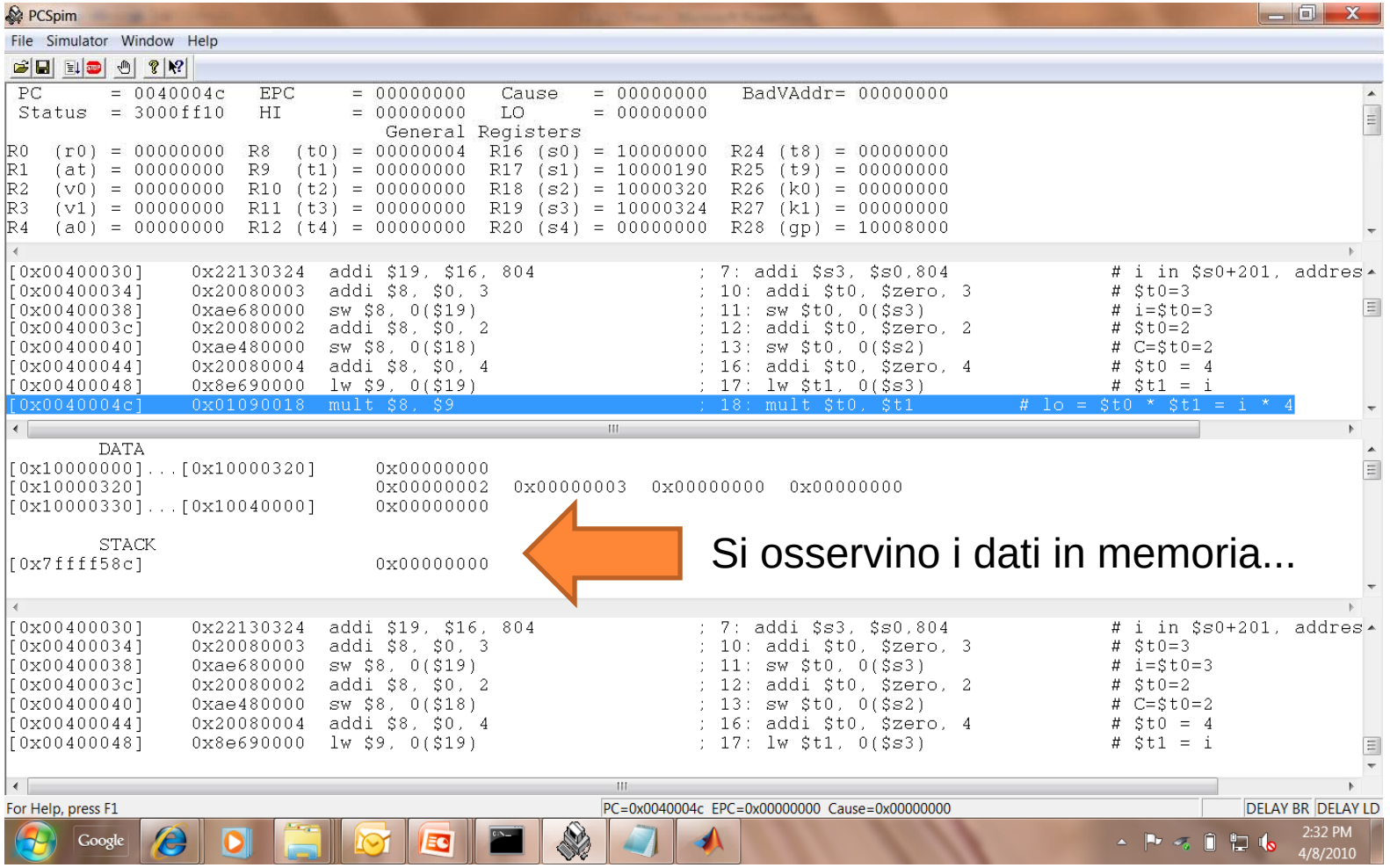
  A[99] = 5 + B[i] + C

  in Assembly. Si inzializzino “casualmente” i registri $s0, $s1, $s2 e $s3 con gli indirizzi di A[0], B[0], C ed i.

Es. 2.2 - Soluzione

```
main:

# Init registers
la $s0,0x10000000 # A[0] in $s0, address of A
addi $s1,$s0,400 # B[0] in $s0+100, address of B
addi $s2,$s0,800 # C in $s0+200,  address of C
addi $s3, $s0,804 # i in $s0+201, address of i

# Init data (i=3, C=2)
addi $t0, $zero, 3        # $t0=3
sw $t0, 0($s3)            # i=$t0=3
addi $t0, $zero, 2        # $t0=2
sw $t0, 0($s2)            # C=$t0=2

# Init data (B[i]=10)
addi $t0, $zero, 4        # $t0 = 4
lw $t1, 0($s3)            # $t1 = i
mult $t0, $t1             # lo = $t0 * $t1 = i * 4
mflo $t0   # $t0 = lo = i*4, offset for B[i]
add $t1, $s1, $t0 # $t1 = $s1+$t0, address of B[i]
addi $t2, $zero, 10                    # $t2 = 10
sw $t2, 0($t1)# B[i] = $t2 = 10

# Make sum
lw $t0, 0($t1)                         # $t0 = B[i]
lw $t1, 0($s2)                         # $t1 = C
add $t0, $t0, $t1                      # $t0 += C
addi $t0, $t0, 5                       # $t0 += 5

# Save sum
sw $t0,396($s0)           # A[99] = $t0
```

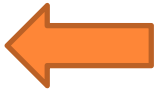
Es. 2.2 - Osservazioni



Es. 2.2 - Osservazioni

- Abbiamo utilizzato delle porzioni di memoria assegnate “a caso”...
- Come è possibile allocare la memoria per contenere dei dati?
- => Direttive assembly (prossime slide).

SOMMARIO

- Istruzioni di lettura e scrittura della memoria, Array
- Direttive assembly, allocazione memoria 
- System services, I/O
- Register spilling

ASSEMBLER DIRECTIVES (1)

.align n : Align the next datum on a 2^n byte boundary. For example, .align 2 aligns the next value on a word boundary. .align 0 turns off automatic alignment of .half, .word, .float, and .double directives until the next .data or .kdata directive.

.ascii str : Store the string in memory, but do not null-terminate it.

.asciiz str : Store the string in memory and null-terminate it.

.byte b1, ..., bn : Store the *n values in successive bytes of memory.*

.data : The following data items should be stored in the data segment. If the optional argument *addr is present, the items are stored beginning at address addr.*

.double d1, ..., dn : Store the *n floating point double precision numbers in successive memory locations.*

.extern sym size : Declare that the datum stored at sym is size bytes large and is a global symbol. This directive enables the assembler to store the datum in a portion of the data segment that is efficiently accessed via register $gp.

.float f1, ..., fn : Store the *n floating point single precision numbers in successive memory locations.*

ASSEMBLER DIRECTIVES (2)

.globl sym : Declare that symbol sym is global and can be referenced from other files.

.half h1, ..., hn : Store the *n 16-bit quantities in successive memory halfwords.*

.kdata : The following data items should be stored in the kernel data segment. If the optional argument *addr is present, the items are stored beginning at address addr.*

.ktext : The next items are put in the kernel text segment. In SPIM, these items may only be instructions or words (see the .word directive below). If the optional argument *addr is present, the items are stored beginning at address addr.*

.space n : Allocate *n bytes of space in the current segment (which must be the data segment in SPIM).*

.text : The next items are put in the user text segment. In SPIM, these items may only be instructions or words (see the .word directive below). If the optional argument *addr is present, the items are stored beginning at address addr.*

.word w1, ..., wn : Store the *n 32-bit quantities in successive memory words.*

ASSEMBLER DIRECTIVES

- .data <segmento dati>
  - Stringa1: .asciiz “Questa è la prima stringa”
  - Array1: .space 16 # Allocazione di un array con 4 elementi
- .text
  - <segmento testo, ovvero codice del programma>
- Le variabili verranno identificate mediante le label corrispondenti (Stringa1, Array1):
- **la $t0, Stringa1 # Carica l’indirizzo di String1 in $t0**

ESERCIZIO 2.3

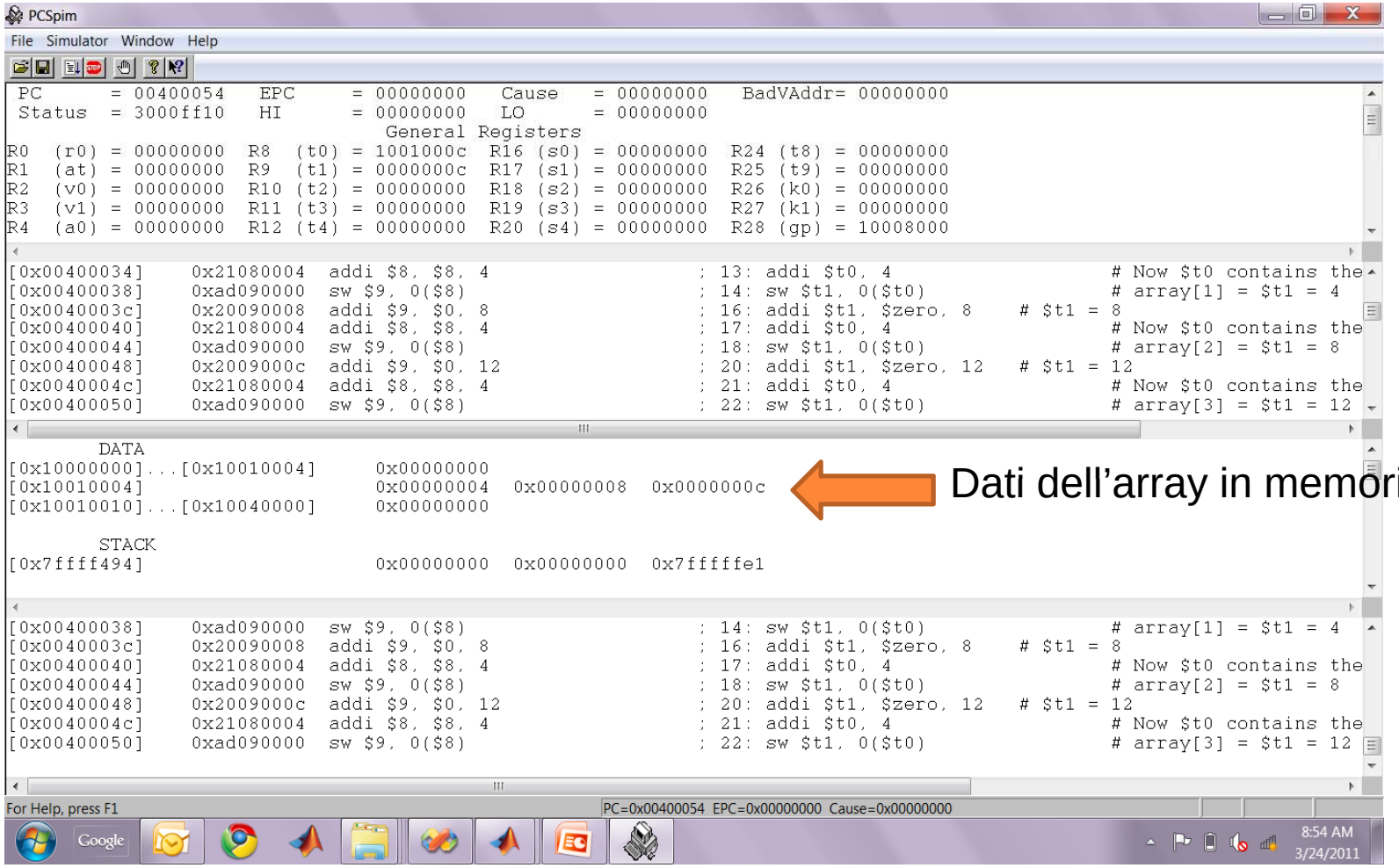
- 2.3a: Mediante le direttive assembly, si allochi la memoria per un array di dimensione 4 e si scriva un programma che riempia l'array con i valori 0, 4, 8 e 12.
- Si analizzi lo stato della memoria prima e dopo l'esecuzione del programma.
- 2.3b: Come si sarebbe potuto ottenere lo stesso risultato usando la direttiva .byte?

- .data <segmento dati>
  - Stringa1: .asciiz "Questa è la prima stringa"
  - Array1: .space 16 # Allocazione di un array con 4 elementi
- .text
  - <segmento testo, ovvero codice del programma>
- **la $t0, Stringa1 # Carica l'indirizzo di String1 in $t0**

ESERCIZIO 2.3A: SOLUZIONE

```
main:
.data
array: .space 16        # Allocate 16 bytes for an array of size 4
.text
la $t0, array           # Save addres of array[0] in $t0
addi $t1, $zero, 0      # $t1 = 0
sw $t1, 0($t0)          # array[0] = $t1 = 0
addi $t1, $zero, 4      # $t1 = 4
addi $t0, 4             # Now $t0 contains the address of  array[1]
sw $t1, 0($t0)          # array[1] = $t1 = 4
addi $t1, $zero, 8      # $t1 = 8
addi $t0, 4             # Now $t0 contains the address of  array[2]
sw $t1, 0($t0)          # array[2] = $t1 = 8
addi $t1, $zero, 12     # $t1 = 12
addi $t0, 4             # Now $t0 contains the address of  array[3]
sw $t1, 0($t0)          # array[3] = $t1 = 12
```

ESERCIZIO 2.3A: SOLUZIONE E OSSERVAZIONI

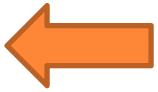


Dati dell'array in memoria

ESERCIZIO 2.3B: SOLUZIONE

- main:
- .data
- array: .byte 0,0,0,0,4,0,0,0,8,0,0,0,12,0,0,0 # Allocate 16 bytes for an array of size 4
- .text
- Oss: attenzione all'ordine con cui sono inseriti i dati nel vettore – i bit meno significativi sono a sinistra nella stringa di 4 byte.

SOMMARIO

- Istruzioni di lettura e scrittura della memoria, Array
- Direttive assembly, allocazione memoria
- System services, I/O 
- Register spilling

SYSTEM SERVICES

Vengono messe a disposizione alcune funzioni “di servizio”...

Service	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		

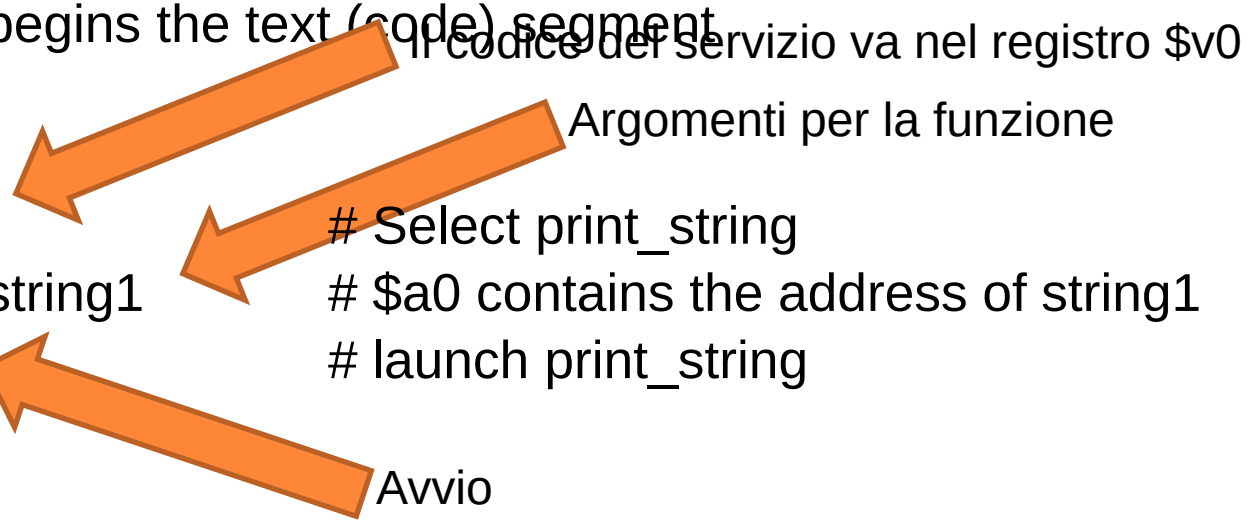
ESEMPIO: ACQUISIZIONE / STAMPA DI UN INTERO (ES. 2.4A)

DATA SEGMENT

- main:
- # The following data go into the data segment
- .data
- string1: .asciiz "Dammi un intero: "
- string2: .asciiz "L'intero è: "

TEXT SEGMENT

- # Here begins the text (code) segment
- .text
- li $v0, 4
- la $a0, string1
- syscall



ESEMPIO: ACQUISIZIONE / STAMPA DI UN INTERO (ES. 2.4B)

```
# The following data go into the data segment
.data
string1: .asciiz "Dammi un intero: "
string2: .asciiz "L'intero successivo è: “
# Here begins the text (code) segment
.text
main:
li $v0, 4              # Select print_string
la $a0, string1        # $a0 contains the address of string1
syscall                # launch print_string
li $v0, 5              # Select read_int
syscall                # launch read_int (in $v0)
add $t0, $zero, $v0    # $t0 = $v0
li $v0, 4              # Select print_string
la $a0, string2        # $a0 contains the address of string2
syscall                # launch print_string
addi $t0, $t0, 1       # $t0 += 1
li $v0, 1              # Select print_int
add $a0,$zero,$t0      # $a0 = $t0
syscall                # launch print_int
```

ESEMPIO: ACQUISIZIONE / STAMPA DI UN INTERO (ES. 2.4C)

- Aggiungiamo la exit per uscire “elegantemente”...

- # The following data go into the data segment
- .data
- string1: .asciiz "Dammi un intero: "
- ... ...
- li $v0, 10 # Select exit
- syscall # Exit

- Si noti come, in questo caso, non vi sono più errori al termine del programma con PCSpim.

ESERCIZIO 2.5

- Si scriva la procedura per l'acquisizione di un numero intero. Si calcoli il successivo del numero acquisito. Si memorizzino l'intero ed il successivo in un array di dimensione 2, residente in memoria. Si mostrino a schermo i due numeri.

- Hint -> si modifichi il testo del codice dell'esercizio 2.4c.

ES. 2.5 - SOLUZIONI & OSSERVAZIONI (1)

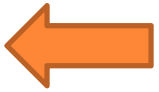

```
.data
string1: .asciiz "Dammi un intero: "
string2: .asciiz "I due numeri sono: "
string3: .asciiz ", “
.align 2
array: .space 8
# Here begins the text (code) segment
.text
main:
li $v0, 4                  # Select print_string
la $a0, string1            # $a0 contains the address of string1
syscall                    # launch print_string
li $v0, 5                  # Select read_int
syscall                    # launch read_int (in $v0)
la $s1, array              # $s1 contains the base address of array
add $t0, $zero, $v0        # $t0 = $v0
sw $t0, 0($s1)             # array[0] = $t0
addi $t0, $t0, 1           # $t0 += 1
```

Uso della direttiva align. per mantenere i dati allineati a 4 byte.

Es. 2.5 - Soluzioni & Osservazioni (2)

```
sw $t0, 4($s1)          # array[1] = $t0
li $v0, 4               # Select print_string
la $a0, string2         # $a0 contains the address of string2
syscall                 # launch print_string
li $v0, 1               # Select print_int
lw $t0, 0($s1)
move $a0, $t0           # $a0 = array[0]
syscall                 # launch print_int
li $v0, 4               # Select print_string
la $a0, string3         # $a0 contains the address of string3
syscall                 # launch print_string
li $v0, 1               # Select print_int
lw $t0, 4($s1)
move $a0, $t0           # $a0 = array[1]
syscall                 # launch print_int
li $v0, 10              # Select exit
syscall                 # Exit
```

SOMMARIO

- Istruzioni di lettura e scrittura della memoria, Array
- Direttive assembly, allocazione memoria
- System services, I/O
- Register spilling 

REGISTER SPILLING

- Il numero di registri è limitato (32);
- Il numero di variabili di un programma è grande a piacere*, in ogni caso può essere maggiore del numero dei registri;
- La memoria viene utilizzata per depositare i dati che non ci stanno nei registri (variabili).

<u>Per un'implementazione efficiente:</u>

- Variabili non utilizzate a breve -> possono risiedere in memoria
- Variabili di prossimo utilizzo -> nei registri
- In questo modo vengono minimizzate le operazioni di trasferimento dati tra registri e memoria

* Si suppone di avere a disposizione una RAM infinita

ESERCIZIO 2.6 - REGISTER SPILLING

- Si supponga che tutti i registri del MIPS non possano essere alterati in quanto contengono dati utili; si abbiano quindi a disposizione i soli registri $s0 e $t0.
- Si scriva il codice che calcola la somma dei primi tre numeri naturali positivi (1, 2 e 3), ciascuno moltiplicato per 3. Non si utilizzi la pseudo-istruzione muli.

- Hint -> viene richiesto di calolare la somma 1*3 + 2*3 + 3*3, effettuando esplicitamente le moltiplicazioni 1*3, 2*3 e 3*3.
- Un registro ($s0) conterrà la somma parziale.
- Il resitro $t0 dovra contenere 1*3, 2*3 e 3*3 ed essere sommato a $s0 per avere la somma totale.
- Come effettuare la moltiplicazion, visto che abbiamo bisogno di due registri quali operandi della moltiplicazione? => $s0 deve essere salvato in memoria e ripristinato (register spilling).