


## 03: LABORATORIO 02

ISTRUZIONI DI LETTURA E SCRITTURA  
DELLA MEMORIA, ARRAY  
REGISTER SPILLING  
DIRETTIVE ASSEMBLY  
I/O

I. Frosio

# SOMMARIO

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

25 March 2011

# 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]**

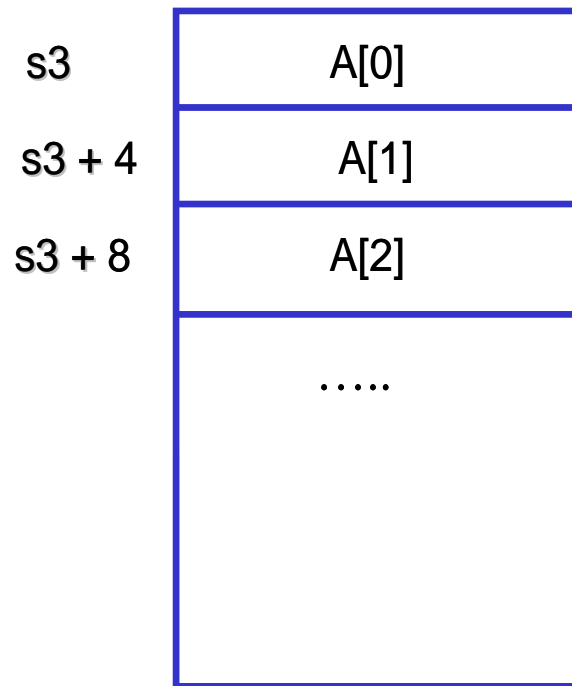
- L'istruzione prende il contenuto del registro \$s1 e lo copia nella parola che ha inizio in \$s2+100. Il registro resta inalterato.

## 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)

25 March 2011

The screenshot shows the PCSpim MIPS simulator interface. At the top, it displays the register file with values for registers R0 through R31. Below this is the assembly code window, which shows the following instructions:

```
[0x00400020] 0x0000000c syscall ; 192: syscall # syscall 10 (exit)
[0x00400024] 0x8d100020 lw $16, 32($8) ; 3: lw $s0, 32($t0) # $s0 = A[8]
[0x00400028] 0x8d310000 lw $17, 0($9) ; 4: lw $s1, 0($t1) # $s1 = h
[0x0040002c] 0x02118020 add $16, $16, $17 ; 6: add $s0, $s0, $s1 # $s0 += $s1, $s0 = A[8]+h
[0x00400030] 0xad100030 sw $16, 48($8) ; 8: sw $s0, 48($t0) # A[12] = $s0
```

Below the assembly code, the memory layout is shown, including DATA, STACK, and KERNEL DATA sections. The bottom part of the window shows the following instructions:

```
[0x0040000c] 0x00041080 sll $2, $4, 2 ; 186: sll $v0 $a0 2
[0x00400010] 0x00c23021 addu $6, $6, $2 ; 187: addu $a2 $a2 $v0
[0x00400014] 0x0c100009 jal 0x00400024 [main] ; 188: jal main
[0x00400018] 0x00000000 nop ; 189: nop
[0x00400024] 0x8d100020 lw $16, 32($8) ; 3: lw $s0, 32($t0) # $s0 = A[8]
```

An orange arrow points to the following exception message:

```
Exception occurred at PC=0x00400024
Bad address in data/stack read: 0x00000020
```

The status bar at the bottom of the window displays: PC=0x80000180 EPC=0x00400024 Cause=0x0000001c



## 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 `0x10000000` è 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 inizializzino “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

25 March 2011

The screenshot shows the PCSpim MIPS simulator interface. The assembly code window displays the following instructions:

```
[0x00400030] 0x22130324 addi $19, $16, 804 ; 7: addi $s3, $s0, 804 # i in $s0+201, address
[0x00400034] 0x20080003 addi $8, $0, 3 ; 10: addi $t0, $zero, 3 # $t0=3
[0x00400038] 0xae680000 sw $8, 0($19) ; 11: sw $t0, 0($s3) # i=$t0=3
[0x0040003c] 0x20080002 addi $8, $0, 2 ; 12: addi $t0, $zero, 2 # $t0=2
[0x00400040] 0xae480000 sw $8, 0($18) ; 13: sw $t0, 0($s2) # C=$t0=2
[0x00400044] 0x20080004 addi $8, $0, 4 ; 16: addi $t0, $zero, 4 # $t0 = 4
[0x00400048] 0x8e690000 lw $9, 0($19) ; 17: lw $t1, 0($s3) # $t1 = i
[0x0040004c] 0x01090018 mult $8, $9 ; 18: mult $t0, $t1 # lo = $t0 * $t1 = i * 4
```

The memory dump window shows the following data:

```
DATA
[0x10000000]...[0x10000320] 0x00000000
[0x10000320] 0x00000002 0x00000003 0x00000000 0x00000000
[0x10000330]...[0x10040000] 0x00000000
```

The stack window shows:


```
STACK
[0x7ffff58c] 0x00000000
```

An orange arrow points to the memory dump with the text "Si osservino i dati in memoria...".

## 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: .ascii "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 address 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

25 March 2011

```
PCSpim
File Simulator Window Help
PC = 00400054  EPC = 00000000  Cause = 00000000  BadVAddr= 00000000
Status = 3000ff10  HI = 00000000  LO = 00000000
General Registers
R0 (r0) = 00000000  R8 (t0) = 1001000c  R16 (s0) = 00000000  R24 (t8) = 00000000
R1 (at) = 00000000  R9 (t1) = 0000000c  R17 (s1) = 00000000  R25 (t9) = 00000000
R2 (v0) = 00000000  R10 (t2) = 00000000  R18 (s2) = 00000000  R26 (k0) = 00000000
R3 (v1) = 00000000  R11 (t3) = 00000000  R19 (s3) = 00000000  R27 (k1) = 00000000
R4 (a0) = 00000000  R12 (t4) = 00000000  R20 (s4) = 00000000  R28 (gp) = 10008000

[0x00400034] 0x21080004 addi $8, $8, 4 ; 13: addi $t0, 4 # Now $t0 contains the
[0x00400038] 0xad090000 sw $9, 0($8) ; 14: sw $t1, 0($t0) # array[1] = $t1 = 4
[0x0040003c] 0x20090008 addi $9, $0, 8 ; 16: addi $t1, $zero, 8 # $t1 = 8
[0x00400040] 0x21080004 addi $8, $8, 4 ; 17: addi $t0, 4 # Now $t0 contains the
[0x00400044] 0xad090000 sw $9, 0($8) ; 18: sw $t1, 0($t0) # array[2] = $t1 = 8
[0x00400048] 0x2009000c addi $9, $0, 12 ; 20: addi $t1, $zero, 12 # $t1 = 12
[0x0040004c] 0x21080004 addi $8, $8, 4 ; 21: addi $t0, 4 # Now $t0 contains the
[0x00400050] 0xad090000 sw $9, 0($8) ; 22: sw $t1, 0($t0) # array[3] = $t1 = 12

DATA
[0x10000000]...[0x10010004] 0x00000000
[0x10010004]...[0x10020008] 0x00000004 0x00000008 0x0000000c
[0x10020008]...[0x1003000c] 0x00000000
[0x1003000c]...[0x10040010] 0x00000000

STACK
[0x7ffff494] 0x00000000 0x00000000 0x7fffffe1

[0x00400038] 0xad090000 sw $9, 0($8) ; 14: sw $t1, 0($t0) # array[1] = $t1 = 4
[0x0040003c] 0x20090008 addi $9, $0, 8 ; 16: addi $t1, $zero, 8 # $t1 = 8
[0x00400040] 0x21080004 addi $8, $8, 4 ; 17: addi $t0, 4 # Now $t0 contains the
[0x00400044] 0xad090000 sw $9, 0($8) ; 18: sw $t1, 0($t0) # array[2] = $t1 = 8
[0x00400048] 0x2009000c addi $9, $0, 12 ; 20: addi $t1, $zero, 12 # $t1 = 12
[0x0040004c] 0x21080004 addi $8, $8, 4 ; 21: addi $t0, 4 # Now $t0 contains the
[0x00400050] 0xad090000 sw $9, 0($8) ; 22: sw $t1, 0($t0) # array[3] = $t1 = 12


For Help, press F1 PC=0x00400054 EPC=0x00000000 Cause=0x00000000
8:54 AM 3/24/2011
```

← 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”...

| <b>Service</b> | <b>Code</b> | <b>Arguments</b> | <b>Result</b>                |
|----------------|-------------|------------------|------------------------------|
| 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)

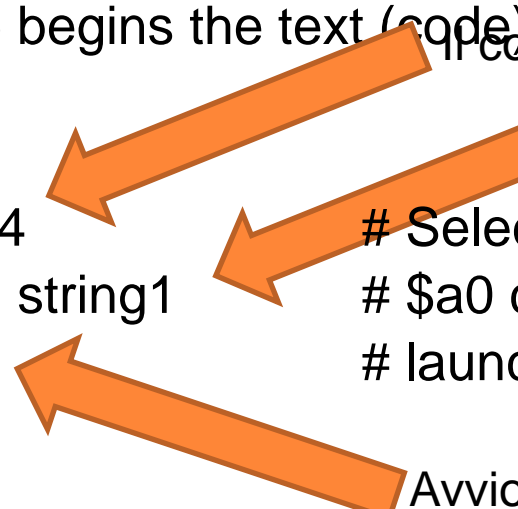
25 March 2011

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
- Il codice del servizio va nel registro \$v0  
Argomenti per la funzione  
# Select print\_string  
# \$a0 contains the address of string1  
# launch print\_string



# 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` ← **Usò della direttiva `align`. per mantenere i dati allineati a 4 byte.**
- `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`

## 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).

## Per un'implementazione efficiente:

- 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 calcolare 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 registro \$t0 dovrà contenere  $1*3$ ,  $2*3$  e  $3*3$  ed essere sommato a \$s0 per avere la somma totale.
- Come effettuare la moltiplicazione, visto che abbiamo bisogno di due registri quali operandi della moltiplicazione? => \$s0 deve essere salvato in memoria e ripristinato (register spilling).