

# Utilizzo della memoria (Indirizzamento, lettura, scrittura) Direttive Assembler in SPIM

[nicola.basilico@unimi.it](mailto:nicola.basilico@unimi.it)

<http://homes.di.unimi.it/basilico/teaching/>

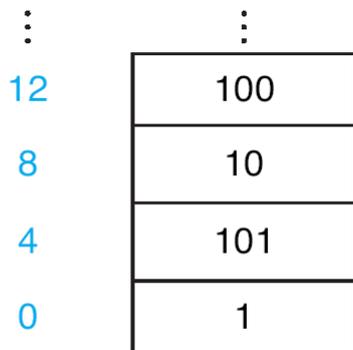
# Organizzazione della memoria

- Cosa contiene la memoria?
  - Istruzioni da eseguire
  - Strutture dati su cui operare
- Come è organizzata?
  - Array uni-dimensionale di elementi dette *parole*
  - Ogni parola è univocamente associata ad un *indirizzo* (come l'indice di un array)



# Organizzazione della memoria

- In generale, la dimensione della parola di memoria non coincide con la dimensione dei registri nella CPU.
- La parola è l'unità base dei trasferimenti tra memoria e registri (*load* e *store* operano per parole di memoria, tipicamente il trasferimento di una parola avviene in un singolo ciclo di clock del bus).
- In MIPS (e quindi in SPIM) una parola è composta da 32 bit (4 bytes)



Il singolo byte è un elemento di memoria spesso ricorrente

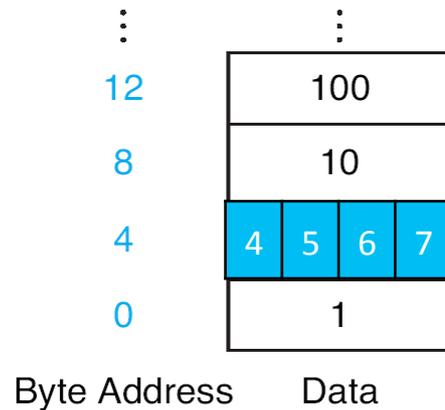
Costruiamo lo spazio indirizzi in modo che ci permetta di indirizzare ognuno dei 4 bytes che compongono una parola: **gli indirizzi di due parole consecutive differiscono di 4** (problema allineamento)

Byte Address      Data

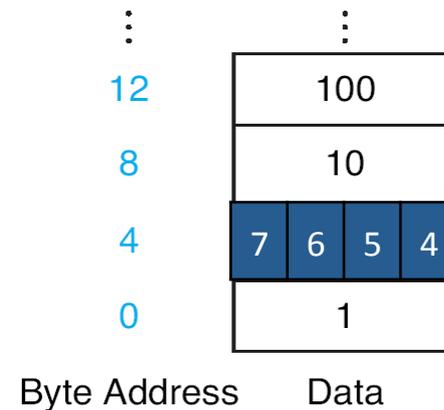
# Indirizzamento del singolo byte

- L'indirizzo che indica la parola indica anche un byte di quella parola. Quale? Due convenzioni:

Indirizzo indica byte più a sinistra  
(big end): Big Endian



Indirizzo indica byte più a destra  
(little end): Little Endian

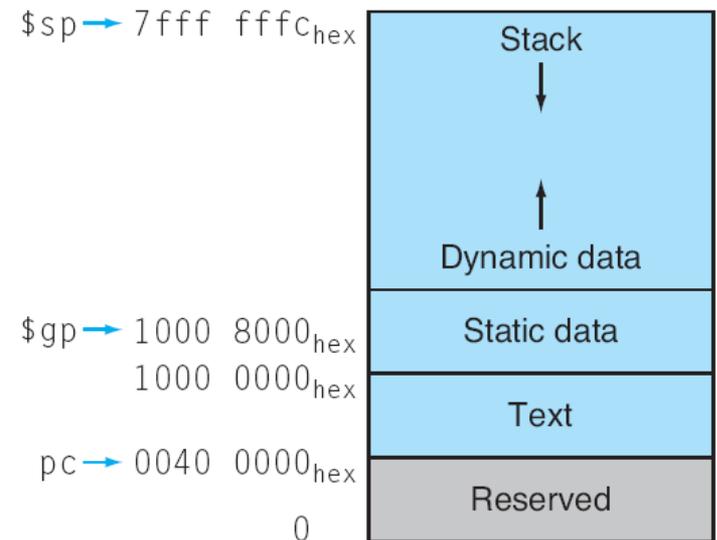


- MIPS adotta la convenzione **Big Endian**

# Utilizzo della memoria

In MIPS la memoria viene divisa in:

- **Segmento testo:** contiene le **istruzioni** del programma
- **Segmento dati:**
  - **dati statici:** contiene dati la cui dimensione è conosciuta a *compile time* e la cui durata coincide con quella del programma (*e.g., variabili statiche, costanti, etc.*)
  - **dati dinamici:** contiene dati per i quali lo spazio è allocato dinamicamente a *runtime* su richiesta del programma stesso (*e.g., liste dinamiche, etc.*)
- **Stack:** contiene dati dinamici organizzati secondo una coda LIFO (Last In, First Out) (*e.g., parametri di una procedura, valori di ritorno, etc.*)



# Accesso alla memoria in Assembly

- Lettura dalla memoria (Load Word):

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

- Scrittura verso la memoria (Store Word):

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

- Memoria indirizzata come vettore (indirizzo base + offset identificano la locazione dove scrivere/leggere una parola).

# Vettori

- Si consideri un vettore  $v$  dove ogni elemento  $v[i]$  è una parola di memoria (32 bit)
- Obiettivo: leggere/scrivere  $v[i]$
- Gli array sono memorizzati in modo sequenziale:
  - $brv$ : registro base di  $v$ , è anche l'indirizzo di  $v[0]$
  - L'elemento  $i$ -esimo ha indirizzo  $brv + 4*i$

# Esercizio 2.1

- Si scriva il codice Assembly che effettui:

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

- Si supponga che
  - La variabile  $h$  sia memorizzata all'indirizzo contenuto in  $\$s1$
  - Il base address di  $A$  sia nel registro  $\$s2$

# Esercizio 2.1

- Si scriva il codice Assembly che effettui:

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

- Si supponga che
  - La variabile  $h$  sia memorizzata all'indirizzo contenuto in  $\$s1$
  - Il base address di  $A$  sia nel registro  $\$s2$

```
lw $t0, 0($s1) # $t0 ← h
lw $t1, 32($s2) # $t1 ← A[8]
add $t0, $t1, $t0 # $t0 ← $t1 + $t0
sw $t0, 48($s2) # A[12] ← $t0
```

# Inizializzazione esplicita degli indirizzi

- Perché il codice funzioni `$s1` e `$s2` devono contenere indirizzi validi
- Pseudo-istruzione «**load address**»:

`la $s1, 0x10000000`      `# $t0 = &h`



32 bit

<code>la \$s1, 0x10000000</code>	<code># \$t0 ← &amp;h</code>
<code>la \$s2, 0x10000004</code>	<code># \$t1 ← A</code>
<code>lw \$t0, 0(\$s1)</code>	<code># \$t0 ← h</code>
<code>lw \$t1, 32(\$s2)</code>	<code># \$t1 ← A[8]</code>
<code>add \$t0, \$t1, \$t0</code>	<code># \$t0 ← \$t1 + \$t0</code>
<code>sw \$t0, 48(\$s2)</code>	<code># A[12] ← \$t0</code>

# Esercizio 2.2

- Dato il seguente codice C dove il base address di  $A$  è contenuto nel registro  $\$s3$ :

```
for (i=0; i<N; i+=2)  
    g = h + A[i];
```

- Specificare le istruzioni di indirizzamento dei vari elementi di  $A$  ad ogni iterazione del ciclo.

# Esercizio 2.2

```
for (i=0; i<N; i+=2)  
    g = h + A[i];
```

- Prima iterazione

```
lw $t0, 0($s3)
```

- Successive iterazioni

```
addi $s3, $s3, 8  
lw $t0, 0($s3)
```

# Esercizio 2.3

- Si scriva il codice Assembly che effettui:

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

- Inizializzazione dei registri indirizzi:
  - I vettori **A** e **B** contengono 100 elementi, ogni elemento è un intero a 32 bit
  - Variabili **C** e **i** sono interi a 32 bit
- Inizializzazione dei valori dati in memoria: **i=3**, **C=2**, **B[i]=10**

# Esercizio 2.3

# Inizializzazione registri indirizzi

```
la $s0, 0x10000000    # ind. A in $s0
addi $s1, $s0, 400    # ind. B in $s1←$s0+100
addi $s2, $s0, 800    # ind. C in $s2←$s0+200
addi $s3, $s0, 804    # ind. i in $s3←$s0+201
```

# Inizializzazione valori (i=3, C=2)

```
addi $t0, $zero, 3    # $t0←3
sw $t0, 0($s3)        # i←$t0
addi $t0, $zero, 2    # $t0←2
sw $t0, 0($s2)        # C←$t0
```

# Inizializzazione valori (B[i]=10)

```
addi $t0, $zero, 4    # $t0 ← 4
lw $t1, 0($s3)        # $t1 ← i
mult $t0, $t1 # lo ← i * 4
mflo $t0              # $t0 ← lo
add $t1, $s1, $t0     # $t1 ← $s1 + $t0
addi $t2, $zero, 10   # $t2 ← 10
sw $t2, 0($t1)        # B[i] ← $t2
```

# calcolo espressione

```
lw $t0, 0($t1) # $t0 ← B[i]
lw $t1, 0($s2)  # $t1 ← C
add $t0, $t0, $t1 # $t0 ← $t0 + $t1
addi $t0, $t0, 5 # $t0 ← $t0 + 5
sw $t0, 396($s0) # A[99] ← $t0
```

# Esercizio 2.4

- Si scriva il codice Assembly che effettui

$$A[c-1] = c*(B[A[c]] + c)/A[2*c-1]$$

- Inizializzazione dei registri indirizzi:
  - I vettori **A** e **B** contengono 4 elementi, ogni elemento è un intero a 32 bit
  - Variabile **c** è intero a 32 bit

- Inizializzazione dei valori dati in memoria:  $c=2$ ,

$A[0]=-1$	$B[0]=-1$
$A[1]=-1$	$B[1]=6$
$A[2]=1$	$B[2]=-1$
$A[3]=4$	$B[3]=-1$

# Esercizio 2.4

# Inizializzazione registri indirizzi

```
la $s0, 0x10000000      # A in $s0
addi $s1, $s0, 16      # B in $s0+4
addi $s2, $s0, 32      # &c in $s0+8
```

# Inizializzazione valori (c=2)

```
addi $t0, $zero, 2      # $t0=2
sw $t0, 0($s2) # c=$t0
```

# Inizializzazione vettori A e B

```
addi $t0, $zero, -1     # $t0 = -1
sw $t0, 0($s0) # A[0] = -1
sw $t0, 4($s0) # A[1] = -1
sw $t0, 0($s1) # B[0] = -1
sw $t0, 8($s1) # B[2] = -1
sw $t0, 12($s1) # B[3] = -1
addi $t0, $zero, 1      # $t0 = 1
sw $t0, 8($s0) # A[2] = 1
addi $t0, $zero, 4      # $t0 = 4
sw $t0, 12($s0) # A[3] = 1
addi $t0, $zero, 6      # $t0 = 6
sw $t0, 4($s1) # B[1] = 1
```

# calcolo  $A[c-1] = c \cdot (B[A[c]] + c) / A[2 \cdot c - 1]$

```
lw $t0, 0($s2) # $t0 = c
addi $t1, $zero, 4 # $t1 = 4
mult $t0, $t1 # lo = $t0 * $t1 = c * 4
mflo $t2 # $t2=c*4, offset A[c]
add $t2, $s0, $t2 # $t2 = $s0+$t2, ind. A[c]
lw $t3, 0($t2) # $t3 = A[c]
mult $t1, $t3 # lo = 4 * A[c]
mflo $t3 # $t3=4*A[c], offset B[A[c]]
add $t3, $s1, $t3 # $t3=$s1+$t3, ind. B[A[c]]
lw $t2, 0($t3) # $t2 = B[A[c]]
add $t2, $t0, $t2 # $t2 = B[A[c]] + c
mult $t0, $t2 # lo = c * (B[A[c]] + c)
mflo $t2 # $t2 = c * (B[A[c]] + c)
addi $t3, $zero, 2 # $t3 = 2
mult $t0, $t3 # lo = 2 * c
mflo $t3 # $t3 = 2 * c
addi $t3, $t3, -1 # $t3 = 2 * c - 1
mult $t1, $t3 # lo = 4 * (2 * c - 1)
mflo $t3 # $t3=4*(2*c-1), offset A[2*c-1]
add $t3, $s0, $t3 # $t3 = $s0+$t3, ind. A[2*c-1]
addi $t0, $t0, -1 # $t0 = c - 1
mult $t0, $t1 # lo = (c-1) * 4
mflo $t0 # $t0=(c-1) * 4, offset A[c-1]
add $t1, $s0, $t0 # $t1 = $s0+$t0, ind. A[c-1]
lw $t0, 0($t3) # $t0 = A[2*c-1]
div $t2, $t0 # lo = c*(B[A[c]] + c)/A[2*c-1]
mflo $t2 # $t2 = c*(B[A[c]] + c)/A[2*c-1]
sw $t2, 0($t1) # A[c-1] = c*(B[A[c]] + c)/A[2*c-1]
jr $ra
```

# Direttive Assembler

- Le operazioni di indirizzamento degli indirizzi e dei valori nella memoria possono essere effettuate mediante direttive Assembler
- Cosa è una direttiva Assembler? Una «meta-istruzione» che fornisce ad Assembler informazioni operazionali su come trattare il codice Assembly dato in input
- Con una direttiva possiamo qualificare parti del codice: e.g., indicare che una porzione di codice è l'inizializzazione del segmento dati, mentre un'altra rappresenta l'elenco di istruzioni da memorizzare nel segmento testo
- Una direttiva è specificata dal suo nome preceduto da «.»

# Direttive Assembler

- `.ascii str` memorizza la stringa `str` in memoria (aggiungendo terminatore di fine stringa)
- `.byte b1, ..., bn` memorizza gli `n` valori in `n` bytes successivi di memoria
- `.data` specifica che gli elementi indicati di seguito sono memorizzati nel segmento dati
- `.space n` alloca `n` byte di spazio nel segmento corrente (deve essere `data`)
- `.text` :specifica che gli elementi indicati di seguito sono memorizzati nel segmento testo (devono essere istruzioni)

# Esercizio 2.1 (con direttive)

- Le variabili vengono identificate mediante le corrispondenti **label**

```
main:
```

```
.data
```

```
A: .space 16
```

```
h: .space 1
```

```
.text
```

```
la $s1, h           # $t0 = &h
```

```
la $s2, A           # $t1 = A
```

```
lw $t0, 0($s1)       # $t0 = h
```

```
lw $t1, 32($s2)      # $t1 = A[8]
```

```
add $t0, $t1, $t0    # $t0 = $t1 + $t0
```

```
sw $t0, 48($s2)      # A[12] = $t0
```

# Esercizio 2.5

- Mediante le direttive assembler, si allochi la memoria per un array di dimensione 4 inizializzato in memoria come segue:  $A[0]=0$ ,  $A[1]=4$ ,  $A[2]=8$ ,  $A[3]=12$

# Esercizio 2.5

- Mediante le direttive assembler, si allochi la memoria per un array di dimensione 4 inizializzato in memoria come segue:  $A[0]=0$ ,  $A[1]=4$ ,  $A[2]=8$ ,  $A[3]=12$

```
main:
```

```
.data
```

```
A: .space 16 # Alloca 16 bytes per A
```

```
.text
```

```
la $t0, A # Scrive base address di A in $t0
```

```
addi $t1, $zero, 0 # $t1 = 0
```

```
sw $t1, 0($t0) # A[0] = 0
```

```
addi $t1, $zero, 4 # $t1 = 4
```

```
addi $t0, 4 # indirizzo di A[1]
```

```
sw $t1, 0($t0) # A[1] = 4
```

```
addi $t1, $zero, 8 # $t1 = 8
```

```
addi $t0, 4 # indirizzo di A[2]
```

```
sw $t1, 0($t0) # A[2] = 8
```

```
addi $t1, $zero, 12 # $t1 = 12
```

```
addi $t0, 4 # indirizzo di A[3]
```

```
sw $t1, 0($t0) # A[3] = 12
```

# Esercizio 2.5

- Con la direttiva `.byte`

```
main:  
  
.data  
array: .byte 0,0,0,0,4,0,0,0,8,0,0,0,12,0,0,0  
  
.text  
jr $ra
```

array: .byte 0,0,0,0, 4,0,0,0, 8,0,0,0, 12,0,0,0

Least Significant Byte

Most Significant Byte

- Quale valore avremmo avuto in `A[1]` con questa direttiva?

```
array: .byte 0,0,0,0,0,4,0,0,8,0,0,0,12,0,0,0
```