



Università degli Studi di Milano
Corso di Laurea in Informatica, A.A. 2017-2018

Indirizzamento, lettura e scrittura della memoria



[Homepage del corso](#)

Turno A

Nicola Basilico

Dipartimento di Informatica
Via Comelico 39/41 - 20135 Milano (MI)

Ufficio S242

nicola.basilico@unimi.it

+39 02.503.16294

Turno B

Jacopo Essenziale

Dipartimento di Informatica
Via Celoria 20 - 20133 Milano (MI)

AILab

jacopo.essenziale@unimi.it

+39 02.503.14010

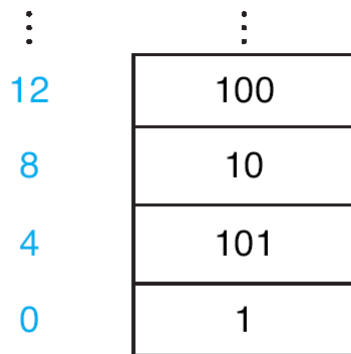
Organizzazione della memoria

- Cosa contiene la memoria?
 - Le istruzioni da eseguire
 - Le 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 (ma nel MIPS sì)
- La parola è l'unità base dei trasferimenti tra memoria e registri (*load word* e *store word* operano per parole di memoria)
- In MIPS (e quindi anche nel simulatore MARS) una parola è composta da 32 bit e cioè 4 byte



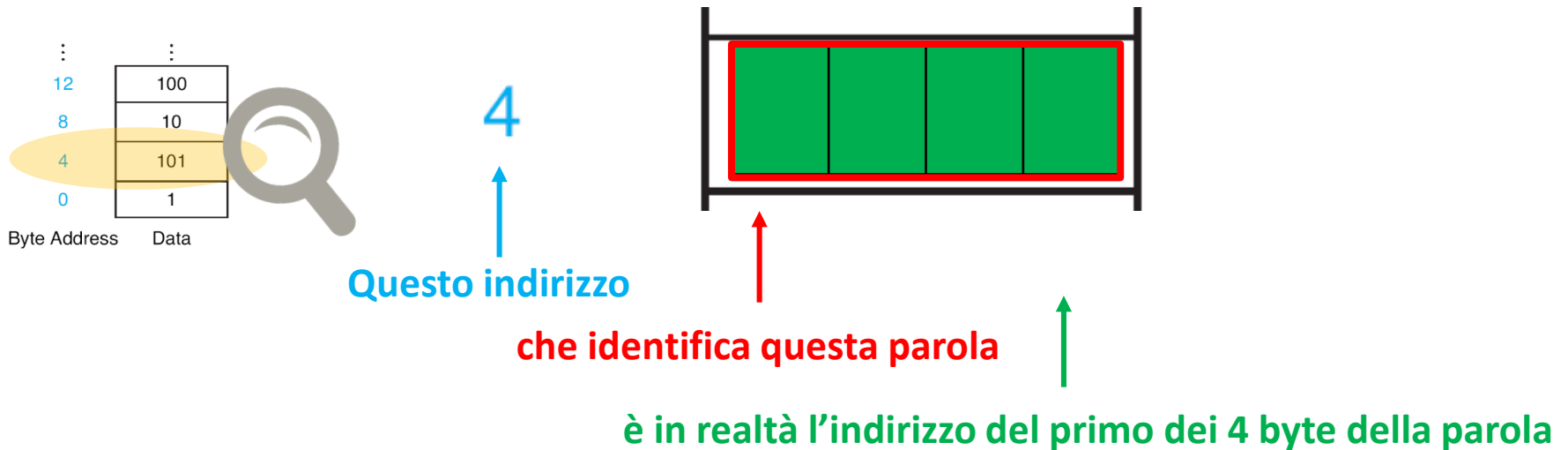
Byte Address Data

Il singolo byte è un elemento di memoria spesso ricorrente

Costruiamo lo spazio degli indirizzi in modo che ci permetta di indirizzare ognuno dei 4 byte che compongono una parola: **gli indirizzi di due parole consecutive differiscono di 4**

Endianness

- L'indirizzo di una parola di memoria è in realtà l'indirizzo di uno dei 4 byte che compongono quella parola

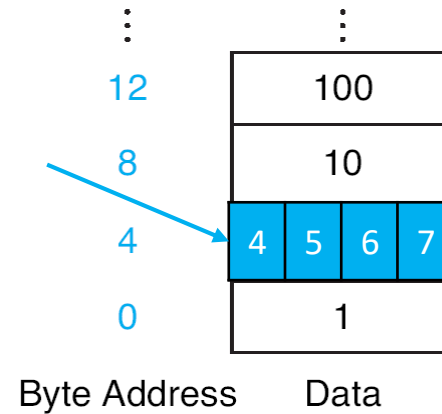


- Ma, tra i 4, quale è il **primo byte**? La risposta sta nell'ordine dei byte: la **endianness**

Endianness

- La **endianness** stabilisce l'ordine dei byte (quindi chi è il **primo** e chi l'ultimo)

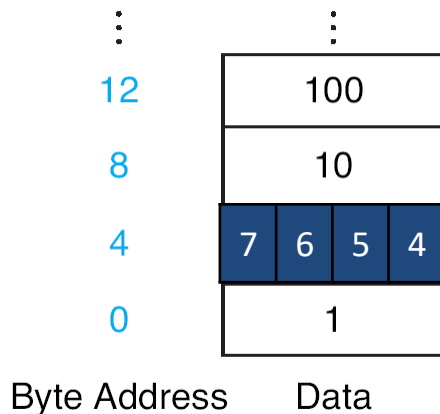
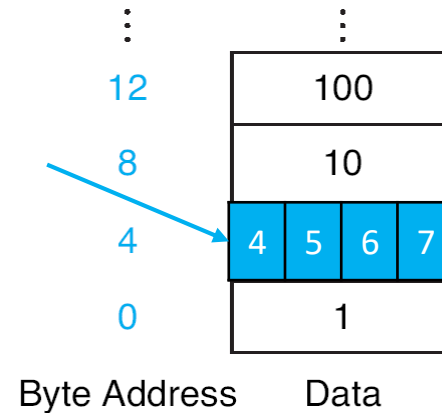
Big endian: il primo byte è quello **più** significativo (quello più a **sinistra**, **big end**)



Endianness

- La **endianness** stabilisce l'ordine dei byte (quindi chi è il **primo** e chi l'ultimo)

Big endian: il primo byte è quello **più** significativo (quello più a **sinistra**, **big end**)

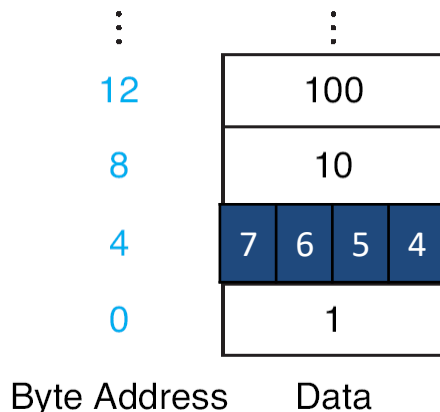
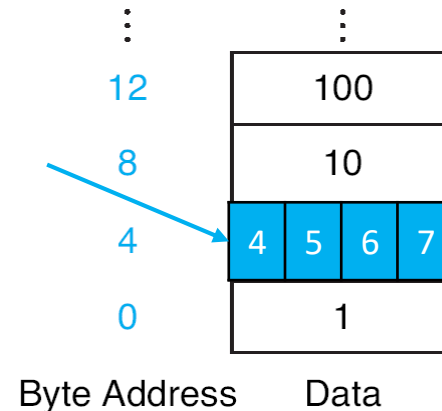


Little endian: il primo byte è quello **meno** significativo (quello più a **destra**, **little end**)

Endianness

- La **endianness** stabilisce l'ordine dei byte (quindi chi è il **primo** e chi l'ultimo)

Big endian: il primo byte è quello **più** significativo (quello più a **sinistra**, **big** end)

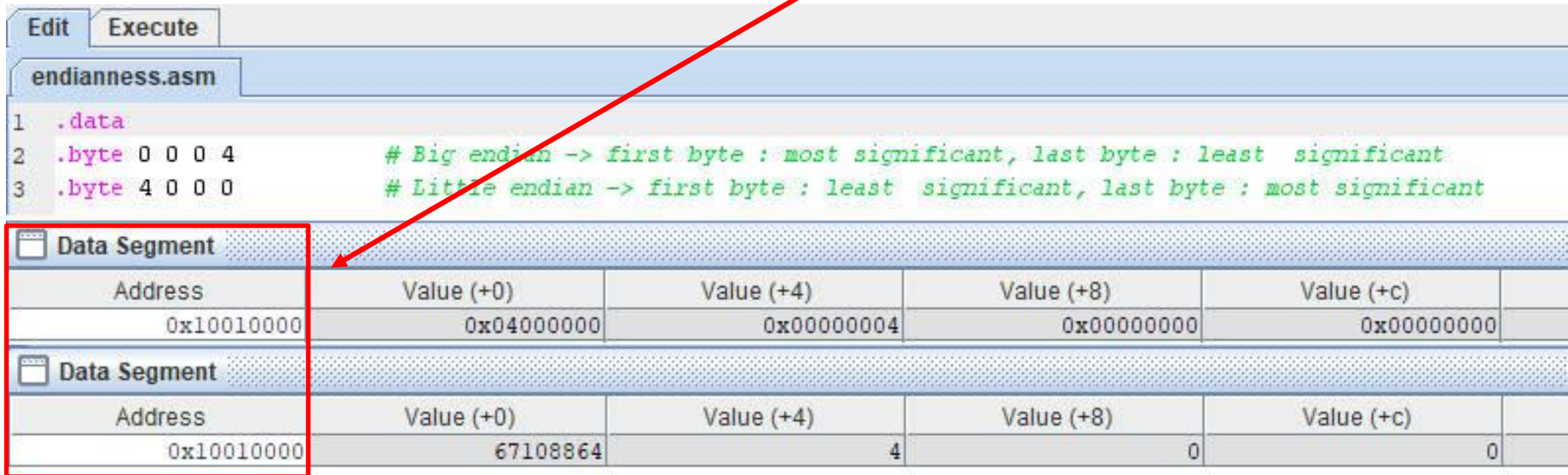


Little endian: il primo byte è quello **meno** significativo (quello più a **destra**, **little** end)

- MIPS è una architettura **Big Endian**, ma ...
- ... il nostro simulatore MARS (e anche SPIM) simula la endianness della macchina su cui è eseguito

Endianness

- Testiamo la endianness della macchina su cui stiamo lavorando:
- Cerchiamo di allocare la costante 4 in una **parola** di memoria:



The screenshot shows a debugger window with the file `endianness.asm` open. The assembly code defines a data segment with two byte sequences. Below the code, the memory dump shows the 'Data Segment' starting at address `0x10010000`. The first table shows the memory layout for the first sequence, and the second table shows the memory layout for the second sequence.

```
1 .data
2 .byte 0 0 0 4      # Big endian -> first byte : most significant, last byte : least significant
3 .byte 4 0 0 0      # Little endian -> first byte : least significant, last byte : most significant
```

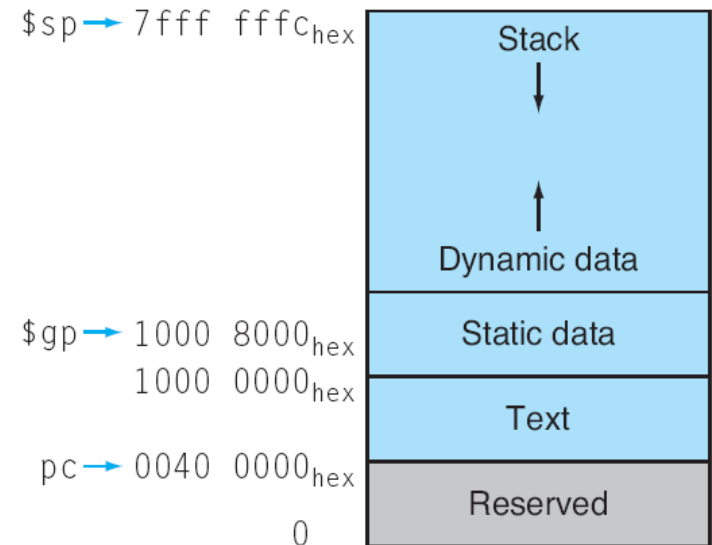
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)
0x10010000	0x04000000	0x00000004	0x00000000	0x00000000

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)
0x10010000	67108864	4	0	0

Utilizzo della memoria (richiamo)

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
```

- La memoria viene indirizzata come un vettore: indirizzo base + offset identificano la locazione della parola da scrivere o leggere

Vettori

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

Esercizio 2.1

- Nome del file sorgente: *arraysum.asm*
- Si assuma che:
 - una variabile *h* sia memorizzata all'indirizzo contenuto in *\$s1*;
 - Il base address di un array *A* sia contenuto nel registro *\$s2*.
- Si scriva, senza eseguirlo, il codice Assembly che effettui:
$$A[12] = h + A[8];$$

Esercizio 2.1

- Nome del file sorgente: *arraysum.asm*
- Si assuma che:
 - una variabile *h* sia memorizzata all'indirizzo contenuto in *\$s1*;
 - Il base address di un array *A* sia contenuto nel registro *\$s2*.
- Si scriva, senza eseguirlo, il codice Assembly che effettui:

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


<i>lw \$t0, 0(\$s1)</i>	<i># \$t0 <- h</i>
<i>lw \$t1, 32(\$s2)</i>	<i># \$t1 <- A[8]</i>
<i>add \$t0, \$t1, \$t0</i>	<i># \$t0 <- \$t1 + \$t0</i>
<i>sw \$t0, 48(\$s2)</i>	<i># A[12] <- \$t0</i>

Inizializzazione esplicita degli indirizzi

- Come fare a caricare degli indirizzi nei registri? Obiettivo:
 - caricare in `$s1` l'indirizzo `0x10000000` (per es. indirizzo di `h`)
 - caricare in `$s2` l'indirizzo `0x10000004` (per es. base address di `A`)

- Soluzione?


```
addi $s1, $zero, 0x10000000    # $t0 = &h
addi $s2, $zero, 0x10000004    # $t1 = A
```



Inizializzazione esplicita degli indirizzi

- Come fare a caricare degli indirizzi nei registri? Obiettivo:
 - caricare in `$s1` l'indirizzo `0x10000000` (per es. indirizzo di `h`)
 - caricare in `$s2` l'indirizzo `0x10000004` (per es. base address di `A`)
- Soluzione?

```
addi $s1, $zero, 0x10000000    # $t0 = &h  
addi $s2, $zero, 0x10000004    # $t1 = A
```




32 bit
- No! Il **valore «immediato»** in `addi` deve essere un intero (con segno, in C2) su 16 bit! (Un'istruzione richiede 32 bit nel suo complesso)
- Cosa succede se assembliamo queste due istruzioni in MARS?

Inizializzazione esplicita degli indirizzi

- Come fare a caricare degli indirizzi nei registri? Obiettivo:
 - caricare in `$s1` l'indirizzo `0x10000000` (per es. indirizzo di `h`)
 - caricare in `$s2` l'indirizzo `0x10000004` (per es. base address di `A`)
- Soluzione?

```
addi $s1, $zero, 0x10000000    # $t0 = &h  
addi $s2, $zero, 0x10000004    # $t1 = A
```




32 bit
- No! Il **valore «immediato»** in `addi` deve essere un intero (con segno, in C2) su 16 bit! (Un'istruzione richiede 32 bit nel suo complesso)
- Cosa succede se assembliamo queste due istruzioni in MARS?

0x00400000	0x3c011000	lui \$1,0x00001000
0x00400004	0x34210000	ori \$1,\$1,0x00000000
0x00400008	0x00018820	add \$17,\$0,\$1
0x0040000c	0x3c011000	lui \$1,0x00001000
0x00400010	0x34210004	ori \$1,\$1,0x00000004
0x00400014	0x00019020	add \$18,\$0,\$1

Inizializzazione esplicita degli indirizzi

- Metodo più comodo: usare la pseudo-istruzione «**load address**»:

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



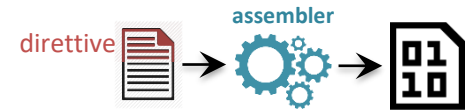
A bracket is drawn under the hexadecimal value 0x10000000, with a vertical line extending downwards to the text "32 bit".

32 bit

```
la $s1, 0x10000000      # $t0 <- &h
la $s2, 0x10000004      # $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
```

Direttive Assembler

- È possibile rappresentare un indirizzo in modo simbolico? Ad esempio scrivendo **A** invece che **0x10000004**? Sì, attraverso le **direttive assembler** (e le label)
- 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. Per esempio indicare che una porzione di codice è il segmento dati, mentre un'altra è il segmento testo (l'elenco di istruzioni)
- Una direttiva è specificata dal suo nome preceduto da « . »
- In MARS tutte le direttive sono visibili sotto *help* ➔ *directives*



Direttive Assembler

- `.data` specifica che ciò che segue nel file sorgente è il segmento dati: vengono specificati gli elementi presenti in tale segmento (stringe, array, etc ...).
- `.text` specifica che ciò che segue nel file sorgente è il segmento testo
- `STRINGA: .asciiz "stringa_di_esempio"` memorizza la stringa "stringa_di_esempio" in memoria (aggiungendo terminatore di fine stringa), il suo indirizzo è referenziato con la label "STRINGA" (significa che potremo scrivere "STRINGA" anzichè l'indirizzo in formato numerico).
- `A: .byte b1, ..., bn` memorizza gli `n` valori in `n` bytes successivi di memoria, la label `A` rappresenta il base address della sequenza (indirizzo della parola con i primi quattro bytes).
- `A: .space n` alloca `n` byte di spazio nel segmento corrente (deve essere data), la label `A` rappresenta il base address (indirizzo della parola con i primi quattro degli `n` bytes).

Esercizio 2.1 (con direttive e label)

```
.data
A: .space 52
h: .space 4

.text
.globl main

main:

    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
    jr $ra
```

- Questo modo di inizializzare gli indirizzi è più intuitivo di quello esplicito visto precedentemente.

Esercizio 2.2

- Nome del file sorgente: *array4.asm*
- 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 2.2

- Nome del file sorgente: *array4.asm*
- Mediante le direttive assembler, si allochi la memoria per un array di dimensione 4 inizializzato in memoria come segue: $A[0]=0$, $A[1]=4$, $A[2]=8$, $A[3]=12$.

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

.text
.globl main

main:

    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, $t0, 4     # indirizzo di A[1]
    sw $t1, 0($t0)       # A[1] = 4
    addi $t1, $zero, 8   # $t1 = 8
    addi $t0, $t0, 4     # indirizzo di A[2]
    sw $t1, 0($t0)       # A[2] = 8
    addi $t1, $zero, 12  # $t1 = 12
    addi $t0, $t0, 4     # indirizzo di A[3]
    sw $t1, 0($t0)       # A[3] = 12
    jr $ra
```

Esercizio 2.2

- Con la direttiva `.byte`

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

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
```

La direttiva `.byte`

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

Least Significant Byte Most Significant Byte



⋮	⋮
Array+12	0 0 0 12
Array+ 8	0 0 0 8
Array+ 4	0 0 0 4
Array+ 0	0 0 0 0
Byte Address	Data

I valori vengono scritti secondo il *byte order* della macchina: la famiglia di architetture x86 è Little Endian (Intel Core i7, AMD Phenom II, FX, ...).

Esercizio 2.3

- Nome del file sorgente: *rwmemoria.asm*
- 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 2.4

- Nome del file sorgente: *rwmemoria2.asm*
- 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: $c=2$,

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



Università degli Studi di Milano
Laboratorio di Architettura degli Elaboratori II
Corso di Laurea in Informatica, A.A. 2017-2018

Nicola Basilico

Dipartimento di Informatica

Via Comelico 39/41 - 20135 Milano (MI)

Ufficio S242

nicola.basilico@unimi.it

+39 02.503.16294

Hanno contribuito alla realizzazione di queste slides:

- Iuri Frosio
- Jacopo Essenziale