Tutorato Architettura degli Elaboratori 03

Alberto Paparella[1]

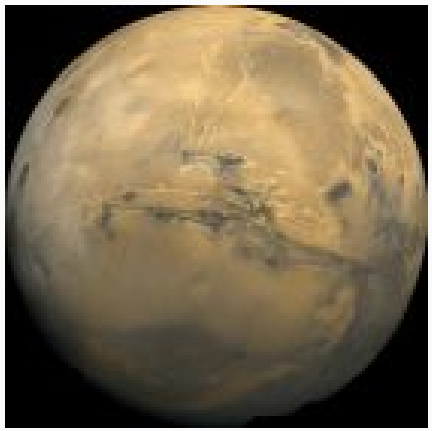
3 Aprile 2025

[1]Dipartimento di Matematica e Informatica, Università degli studi di Ferrara

Introduzione al Simulatore del Set di Istruzioni MIPS “MARS”

Ambiente Integrato di Sviluppo (IDE)

- Per sviluppare un codice MIPS assembly, il progettista può avvalersi di un ambiente integrato di sviluppo che lo aiuti a scrivere, testare, compilare ed eseguire il codice.
- L'ambiente di sviluppo che useremo nelle prossime esercitazioni è **MARS**, sviluppato alla Missouri State University per scopi didattici.
- Website: `https://dpetersanderson.github.io/`



Il Simulatore MARS

- **M**ips **A**ssembler and **R**untime **S**imulator.
- E' un simulatore del set di istruzioni MIPS sviluppato in ambito accademico.
- Serve per sviluppare, simulare e fare il debugging di codice Assembler del MIPS.
- Ultima versione scaricabile da: `https://dpetersanderson.github.io/Mars4_5.jar`.
- **Importante: richiede la Java Virtual Machine!**

Il Simulatore MARS

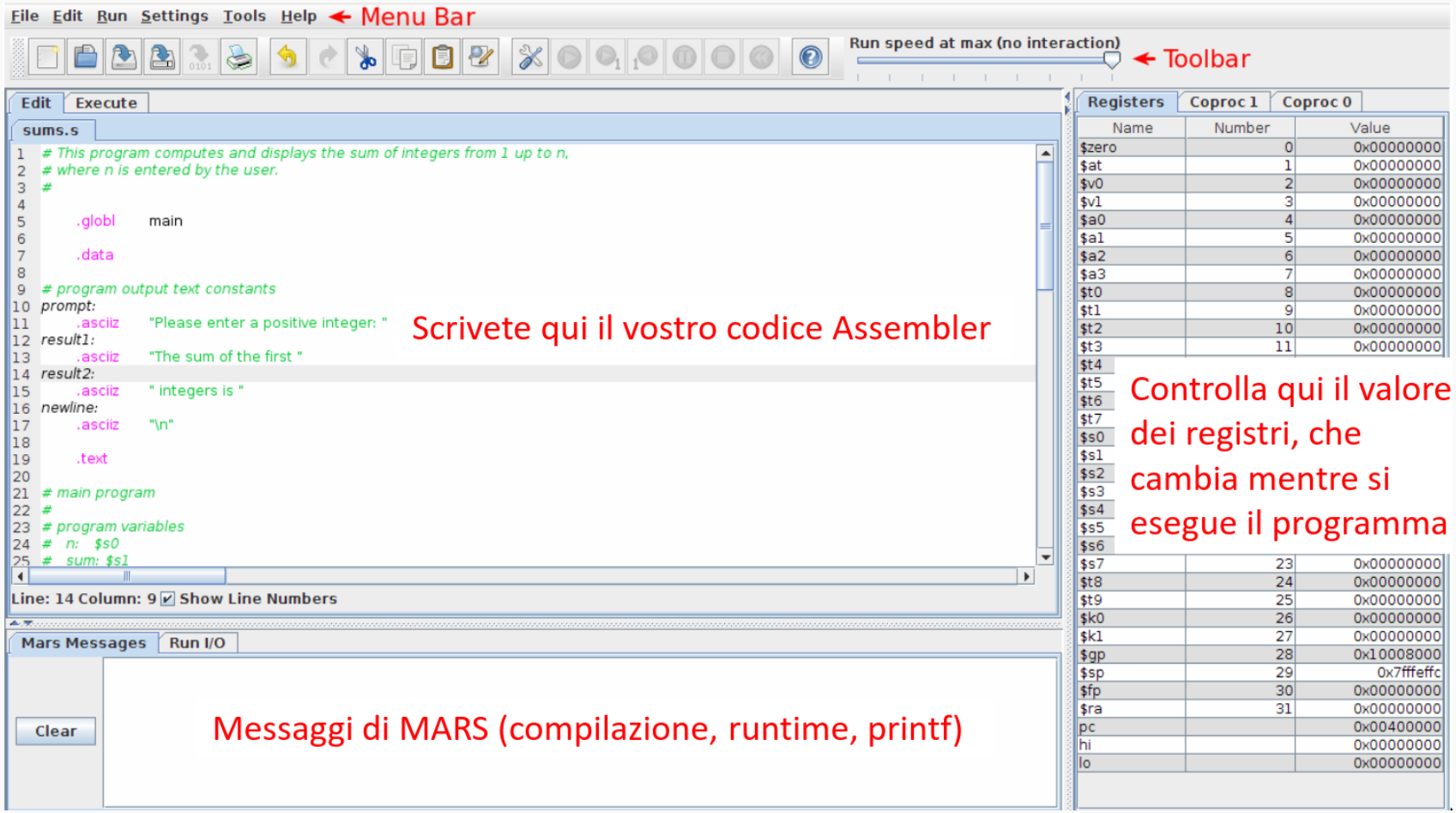
- Se stai usando **Windows**, installa la Java Virtual Machine se non l'hai ancora fatto (`http://www.oracle.com/technetwork/java/javase/downloads/index.html`), dopodichè fai doppio clic sull'eseguibile Jar `Mars4_5` per aprirlo.
- Se stai usando **Linux**, installa **Java SDK** usando il tuo gestore dei pacchetti (**aptitude** per **Debian**/**Ubuntu**, **pacman** per **Arch**, ...); per esempio, su **Debian**/**Ubuntu** puoi lanciare il comando:

  ```
  sudo apt install default-jre
  ```

  Poi lancia Mars con il comando:

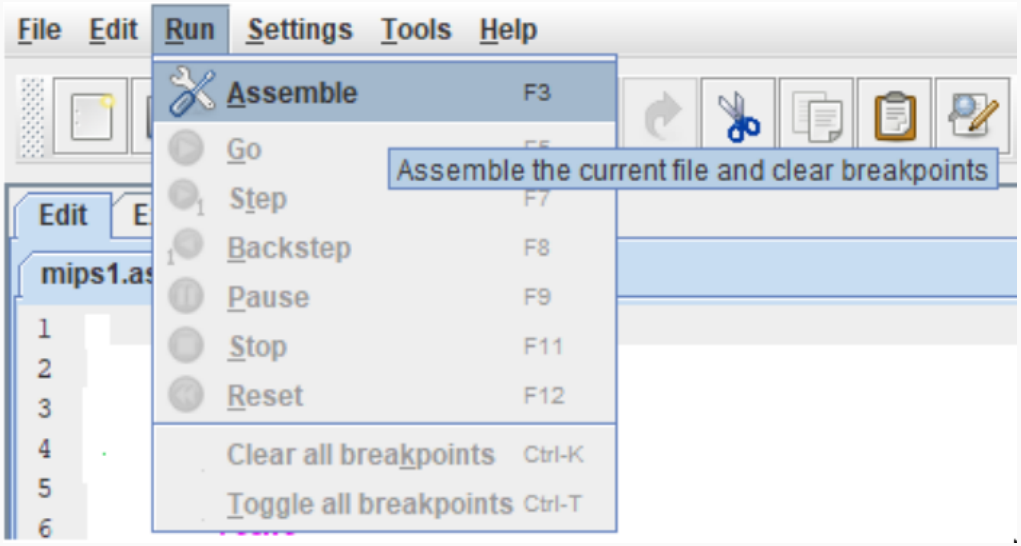
  ```
  java -jar Mars4_5.jar
  ```

Il Simulatore MARS

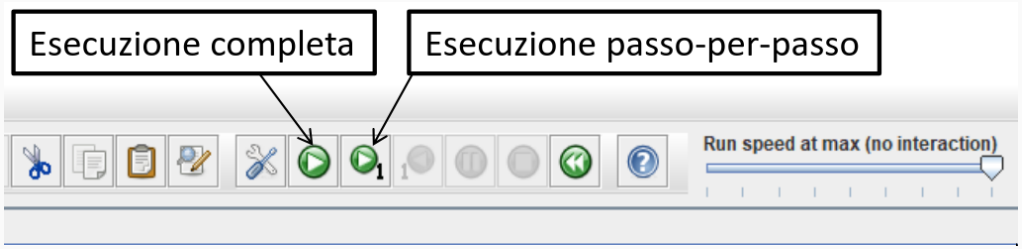


Il Simulatore MARS

- Per compilare, clicca su **Run** e poi **Assemble**:



- Per eseguire, clicca su uno dei due tasti:



Hello, World!

- Scriviamo il nostro primo programma in Assembler:

```
1  .data
2      MyMessage:   .asciiz "Hello, World!\n"
3
4  .text
5      Main:
6          li   $v0, 4
7          la   $a0, MyMessage
8          syscall
9      Exit:
10         li   $v0, 10
11         syscall
```

- Il programma dovrebbe stampare il messaggio:

  ```
  Hello, World!
  ```

  nella finestra di dialogo in basso.

Hello, World!

```
1 .data
2     MyMessage:   .asciiz "Hello, World!\n"
```

- `.data` delimita l'inizio del segmento data, ovvero il contenitore dei dati statici nel file oggetto.
- `.asciiz` indica che la stringa tra virgolette è ASCII e terminata da `NULL` (byte 0).
- "MyMessage" è una **label** o etichetta.

Hello, World!

```
1 .text
2     Main:
3         li   $v0, 4
4         la   $a0, MyMessage
5         syscall
6     Exit:
7         li   $v0, 10
8         syscall
```

- `.text` delimita l'inizio del segmento text, ovvero il contenitore delle istruzioni del file oggetto.
- `Main` è in questo caso il nostro punto di ingresso.
- Carichiamo il servizio "print string" (**4**) con la load immidiate (**li**).
- Con la "load address" (**la**) carichiamo in $a0 l'indirizzo della stringa.
- Syscall **10** per la terminazione del programma.
- "Main" e "Exit" sono **label** o etichette.

Tutorial sulle funzionalità di base del simulatore MARS - Parte 1

Registri

Registri	`regdef.h`	Utilizzo
$0		Permanentemente settato a $0x0$.
$at		Riservato per l'assemblatore.
$2..3	v0-v1	Usati per la valutazione di espressioni e per contenere il risultato di funzioni di tipo intero. Anche usato per passare il link statico quando si chiamano procedure innestate.
$4..7	a0-a3	Usati per passare i primi 4 argomenti attuali (parole di tipo intero); valori non preservati fra chiamate a procedura.
$8..15	t0-t7	Registri temporanei usati per la valutazione di espressioni; i loro valori non sono preservati fra chiamate a procedura.
$16..23	s0-s7	Registri salvati: preservare i valori fra chiamate a procedura.
$24..25	t8-t9	Come t0-t7.
$26..27	k0-k1	Riservati per il kernel del sistema operativo.
$28	gp	Contiene il puntatore globale (global pointer).
$29	sp	Contiene il puntatore allo stack (stack pointer).
$30	fp	Contiene il puntatore al frame (frame pointer) se necessario; altrimenti un registro salvato (come s0-s7).
$31	ra	Contiene il return address, per la valutazione di espressioni.

Table 1: In ASM abbiamo accesso diretto ai 32 registri a 32 bit del MIPS.

Direttive all'Assemblatore

Direttiva	Descrizione
`.text`	Inizio del blocco di istruzioni.
`.globl x`	Indica che la label `x` è accessibile da un altro file.
`.data`	Inizio del blocco dei dati statici.
`.eqv $nome, $reg`	Permette di usare `$nome` per riferirci a `$reg`.
`.macro` e `end_macro`	Definisce una macro.

Table 2: Le direttive all'Assemblatore forniscono informazioni utili all'Assembler per gestire l'organizzazione del codice.

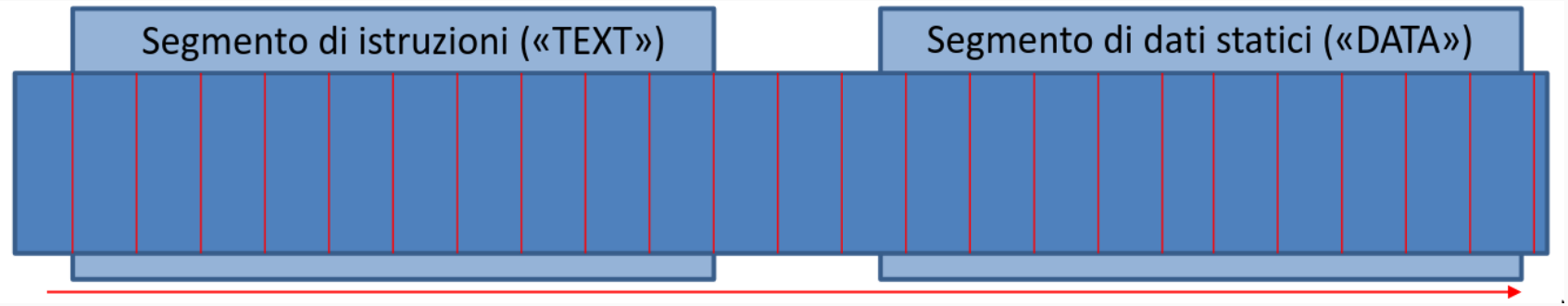
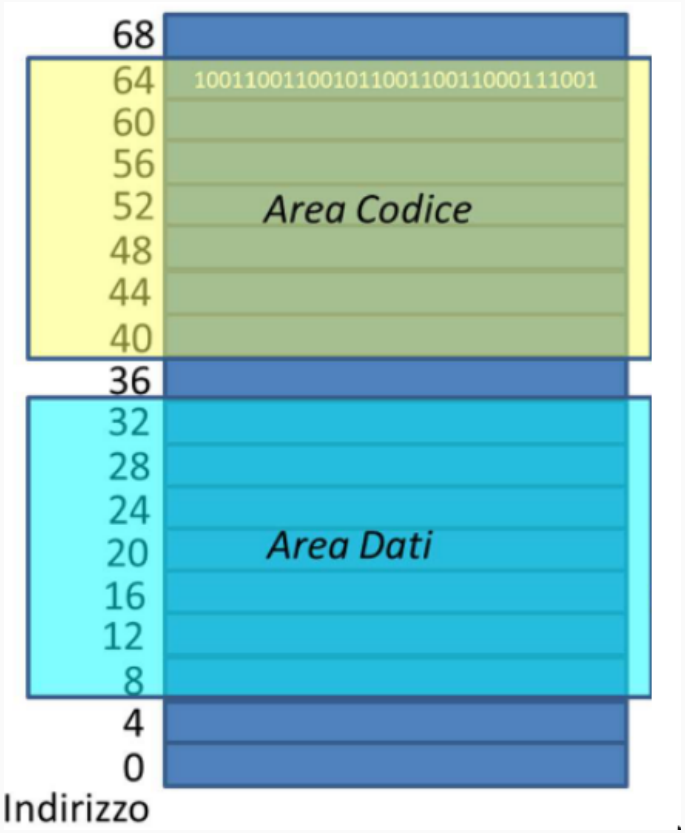


Figure 1: Array Lineare di Memoria: è diviso in segmenti (TEXT, DATA, ma anche STACK, HEAP, . . .).

Direttive Principali

```
1 .data
2     # allocare qui le variabili in memoria dati
3
4 .text
5     # scrivere qui il codice della memoria istruzioni
```



Equivalenze

- Migliorano la leggibilità del codice
- L'utilizzo è a totale discrezione del programmatore

```
1 .text
2     addi    $t0, $t0, 1
3     addi    $t1, $t1, 2
4     add     $t2, $t0, $t1
```

Listing 1: Esempio di codice senza l'utilizzo di equivalenze.

```
1 .eqv op1, $t0
2 .eqv op2, $t1
3 .eqv risultato, $t2
4
5 .text
6     addi op1, op1, 1
7     addi op2, op2, 2
8     add risultato, op1, op2
```

Listing 2: Esempio di codice con l'utilizzo di equivalenze.

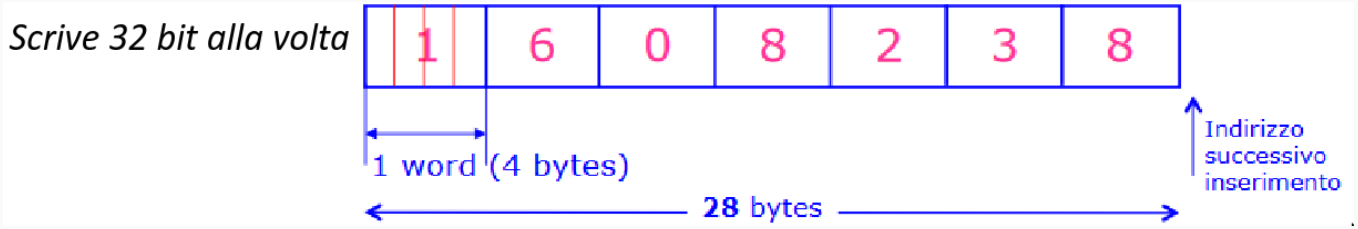
Allocazione Statica di Memoria

Direttiva	Descrizione
`.byte` $b_1, \ldots, b_n$	Alloca n quantità a 8 bit in byte successivi in memoria.
`.half` $h_1, \ldots, h_n$	Alloca n quantità a 16 bit in halfword successive in memoria.
`.word` $w_1, \ldots, w_n$	Alloca n quantità a 32 bit in word successive in memoria.
`.float` $f_1, \ldots, f_n$	Alloca n valori floating point a singola precisione in locazioni successive in memoria.
`.double` $d_1, \ldots, d_n$	Alloca n valori floating point a doppia precisione in locazioni successive in memoria.
`.asciiz` *str*	Alloca la stringa *str* in memoria, terminata con il valore 0.
`.space` n	Alloca n byte, senza inizializzazione.

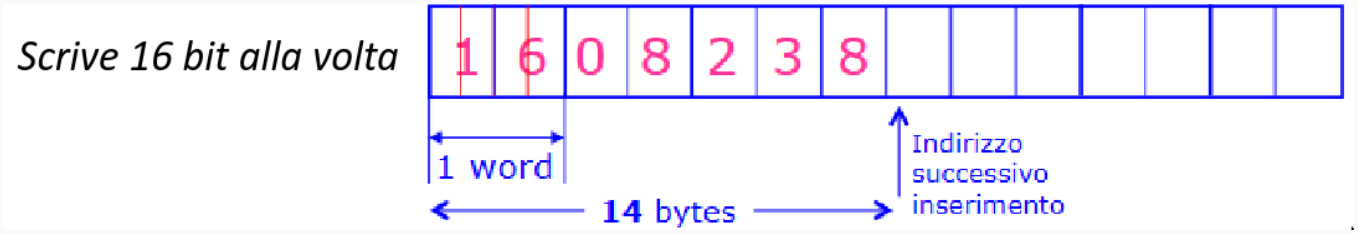
Table 3: Direttive di allocazione di memoria. All'interno del segmento **.data** possiamo definire dati statici in questi modi.

Esempi

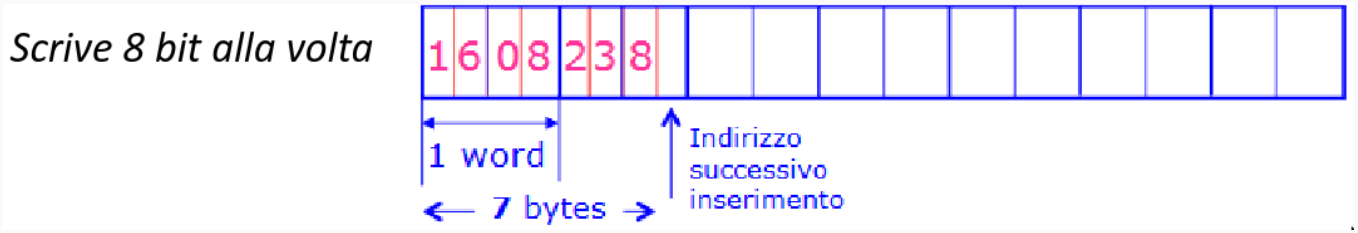
```
1  .word    1, 6, 0, 8, 2, 3, 8
```




```
1  .half    1, 6, 0, 8, 2, 3, 8
```

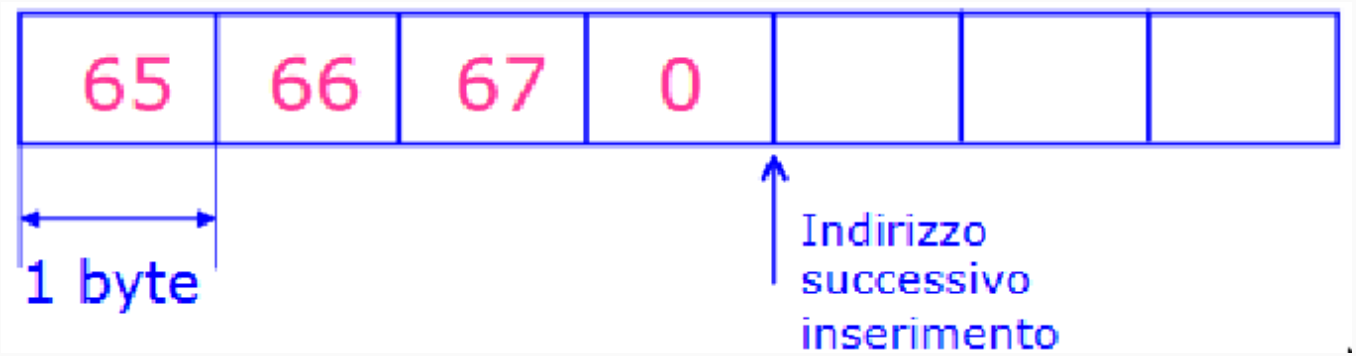



```
1  .byte    1, 6, 0, 8, 2, 3, 8
```



Esempi

```
1 .asciiz "ABC"
```

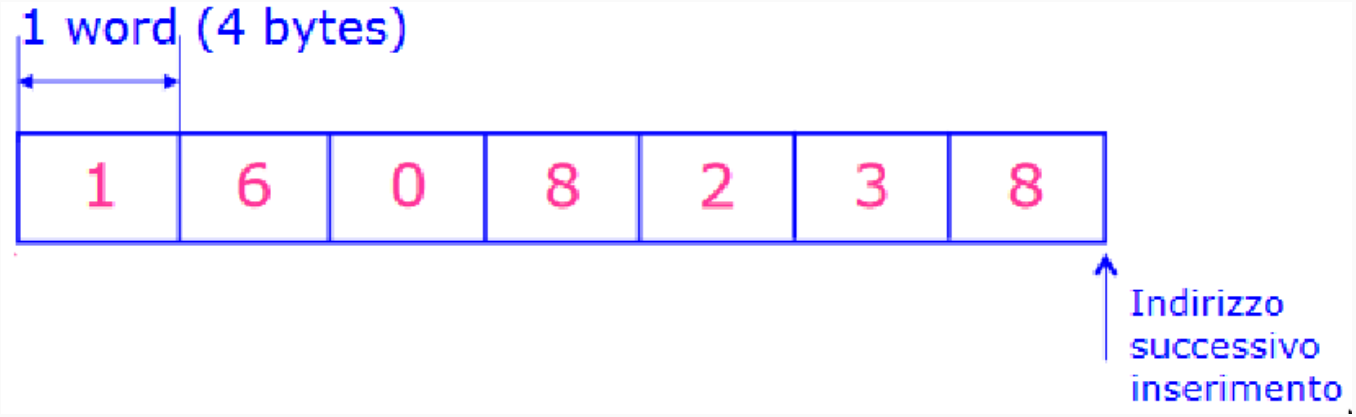


E' equivalente a:

```
1 .byte    65, 66, 67, 0
```

Esempi

```
1 .word    1, 6, 0, 8, 2, 3, 8
```



- Si tratta della allocazione statica di un array di interi.
- Come accedere agli elementi dell'array?
- Ricordandosi l'esatto indirizzo di memoria di ogni elemento? (**Spoiler: ovviamente no.**)

Esempi

- **Soluzione**: uso un identificatore!

```
1 array:   .word    1, 6, 0, 8, 2, 3, 8
```

Identificatori

- Un **identificatore** è un **nome** associato ad una particolare posizione del programma ASM, come l'indirizzo di un'istruzione o di un dato.
  - E.g., "main", oppure "forloop", oppure "exitcode, ...
  - E.g., "A" associato ad una variable di x byte
- Ogni istruzione o dato si trova in un particolare indirizzo di memoria. Un identificatore ci permette di fare riferimento ad una particolare posizione senza sapere il suo indirizzo in memoria.

Etichette o Label

- Un'**etichetta** *introduce* un identificatore e lo associa al punto del programma in cui si trova.
- Un'etichetta consiste in un identificatore seguito dal simbolo ":".
  - E.g., "main:", "forloop:", "exitcode:", ...
  - E.g., "A: word 15" indica l'etichetta di una variabile di 4 byte inizializzata al valore 15.
- L'identificatore introdotto può avere visibilità **locale** o **globale**. Le etichette sono locali per default.
- L'uso della direttiva ".globl" rende l'etichetta globale.
- Un'etichetta locale può essere referenziata solo dall'interno del file in cui è definita. Un'etichetta globale può essere referenziata anche da file diversi.

Riferimenti

- Un identificatore può essere **usato** in un programma Assembler per fare riferimento alla posizione in memoria associata all'identificatore stesso.

```
1 Forloop:
2     ...(istruzioni)...
3     ...(istruzioni)...
4     jump Forloop
```

- E' sufficiente **una sola etichetta** anche per dati che occupano più byte; ogni byte può essere referenziato tramite uno scostamento (calcolato in byte) all'indirizzo base.

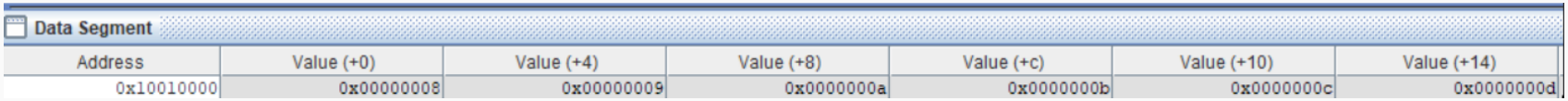
```
1 Array: .word   10, 2, 33, 42, 51   # Istanzia un array di 5
      interi inizializzati
```

Listing 3: Il secondo elemento dell'array si può referenziare con "Array+4".

Esercitiamoci con MARS

```
1 .data
2     a:   .word    8
3     b:   .word    9
4     c:   .word    10, 11, 12, 13
```

- Dopo il comando "Assemble":



Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)
0x10010000	0x00000008	0x00000009	0x0000000a	0x0000000b	0x0000000c	0x0000000d

Figure 2: Layout di memoria.

- Endianess nascosta dal debugger
- Memorizzazione di un intero ogni 4 byte, in ordine di dichiarazione

Esercitiamoci con MARS

```
1 .data
2     a:   .half    8
3     b:   .half    9
4     c:   .half    10, 11, 12, 13
```

- Dopo il comando "Assemble":

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)
0x10010000	0x00090008	0x000b000a	0x000d000c	0x00000000

- Memoria progressivamente riempita ad indirizzi crescenti
- Il debugger visualizza i valori memorizzati usando l'ipotesi di **little endianess**: scrivere ".half 8" significa posizionare "0x08" nel byte di indirizzo più basso
- Le successive half-word sono memorizzate di seguito ognuna in 16 bit

Esercitiamoci con MARS

```
1 .data
2     a:   .byte    8
3     b:   .byte    9
4     c:   .byte    10, 11, 12, 13
```

- Dopo il comando "Assemble":

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)
0x10010000	0x0b0a0908	0x00000d0c	0x00000000

- L'ordine di dichiarazione determina la posizione in memoria, dall'indirizzo più basso a quello più alto
- Il debugger visualizza i valori memorizzati usando l'ipotesi di **little endianess**: nella prima parola, ad indirizzi crescenti, troviamo quindi "0x08", "0x09", "0x0a", "0x0b", che il debugger interpreta come "0x0b0a0908"

Esercitiamoci con MARS

```
1 .data
2     a:            .byte    8
3     Stringa:      .asciiz  "AB"
4     b:            .byte    9
5     c:            .byte    10, 11, 12, 13
```

- Dopo il comando "Assemble":

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)
0x10010000	0x00424108	0x0c0b0a09	0x0000000d	0x00000000

- L'ordine di dichiarazione determina la posizione in memoria, dall'indirizzo più basso a quello più alto
- Il debugger visualizza i valori memorizzati usando l'ipotesi di **little endianess**, ed i caratteri vengono memorizzati secondo la codifica ASCII: nella prima parola, ad indirizzi crescenti, troviamo "0x08", "0x41" ("A"), "0x42" ("B"), "0x00" (terminatore), che il debugger interpreta come "0x00424108"

Esercitiamoci con MARS

```
1 .data
2     a:              .byte    8
3     Stringa:        .ascii   "AB"
4     b:              .byte    9
5     c:              .byte    10, 11, 12, 13
```

- Dopo il comando "Assemble":

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)
0x10010000	0x09424108	0x0d0c0b0a	0x00000000

- Con ".asciiz": nella prima parola, ad indirizzi crescenti, troviamo "0x08", "0x41", "0x42", "0x00", che il debugger interpreta come "0x00424109"
- Con ".ascii": scompaiono i due "00" dalla posizione più significativa, ed abbiamo subito "0x09"