

# Architettura degli Elaboratori Elettronici Esercitazioni

*MARS Nomenclatura*

**Franco Liberati**  
**[liberati@di.uniroma1.it](mailto:liberati@di.uniroma1.it)**

# MARS

*Linguaggio assemblativo*





# MARS

## *Introduzione*

- ❑ Compito principale di un linguaggio assemblativo è quello di consentire una **programmazione immediata**
- ❑ Un **linguaggio assemblativo** (tra cui il MARS):
  - ❑ Utilizza di **parole mnemoniche** per identificare le istruzioni del linguaggio macchina
  - ❑ Aumento del numero di istruzioni disponibili per il programmatore attraverso l'uso di **pseudoistruzioni**
  - ❑ Possibilità di definire **macro** (gruppo di istruzioni)
  - ❑ Possibilità di definire **etichette** (al posto di indirizzi)
  - ❑ Possibilità di aggiungere **commenti**



# MARS

## *Vantaggi e svantaggi*

- ✓ Realizzare sistemi real time
- ✓ Ottimizzare sezioni critiche dal punto di vista della performance di un programma
- ✓ Utilizzare istruzioni particolari del processore altrimenti non utilizzate dai compilatori (ad es., istruzioni MMX)
- ✓ Sviluppare il kernel di un sistema operativo, che necessita di istruzioni particolari per gestire la protezione della memoria
- ✓ Eliminare vincoli dettati dall'espressività dei costrutti di un linguaggio ad alto livello
- ✓ Utilizzare le astrazioni dei linguaggi ad alto livello
- ✗ Complesso
- ✗ Specificare tutti i dettagli implementativi
- ✗ Scarsa leggibilità del codice
- ✗ Scarsa gestione del codice
- ✗ Programmatore è supervisore e supergestore
- ✗ Scarsa produttività
- ✗ Non portabilità del codice
- ✗ Massima efficienza dei compilatori



# MIPS

*Nomenclatura*





# MARS

## Direttive

- ☐ Le **direttive** forniscono informazioni aggiuntive utili all'assemblatore per gestire l'organizzazione del codice
- ☐ Sono identificatori (etichette che iniziano per un carattere alfabetico e sono seguiti da caratteri alfanumerici e il simbolo `_`) che iniziano con un punto





# MARS

## Direttive

- ❑ Esempi di **direttive per il MARS** sono :

**.text**

*indica che le linee successive contengono istruzioni*

**.data**

*indica che le linee successive contengono dati*

**.end**

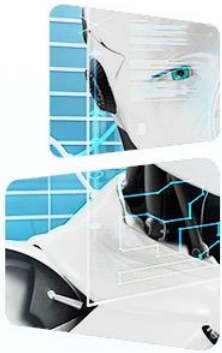
*indica la fine del programma*

**.globl**

*indica funzioni, variabili globali (accessibili da diversi moduli)*

**.macro**

*definisce una macro (un insieme di istruzioni sono descritte da una etichetta)*



# MARS

## Direttive - esempio

```
.text  
.globl main  
main:
```

```
    lw $a0, Size  
    li $a1, 0  
    li $a2, 0  
    li $t2, 4  
  
loop:  
    mul $t1, $a1, $t2  
    lw $a3, Array($t1)  
    add $a2, $a2, $a3  
    add $a1, $a1, 1  
    beq $a1, $a0, store  
    j loop
```

```
store:
```

```
    sw $a2, Result
```

```
.end
```

```
.data
```

```
Array: .word 1, 2, 3, 4, 5
```

```
Size: .word 4
```

```
Result: .word 0
```





# MARS

## Direttive – Tipi di dati

- ❑ Il MARS gestisce 4 tipi di dati:
  - ❑ **interi** con lunghezza ad otto bit (*byte*); a sedici bit (*half*); e a 32bit (*word*);
  - ❑ **reali** in singola precisione a 32bit (*float*) e in doppia precisione a 64bit (*double*)
  - ❑ **stringhe** con terminatore (*asciiz*) e senza terminatore (*ascii*)
  - ❑ **spazi di memoria allocabili** (*space*)



# MARS

## Direttive – Tipi di dati: definizione

### □ Definizione dei tipi del MARS:

<b>.byte</b> b1 , ..., bn	<i>Alloca n quantità a 8 bit (byte) successivi in memoria</i>
<b>.half</b> h1, ..., hn	<i>Alloca n quantità a 16 bit (halfword) successive in memoria</i>
<b>.word</b> w1, ..., wn	<i>Alloca n quantità a 32 bit (word) successive in memoria</i>
<b>.float</b> f1, ..., fn	<i>Alloca n valori a singola precisione (floating point) in locazioni successive di memoria</i>
<b>.double</b> d1, ..., dn	<i>Alloca n valori a doppia precisione (double point) in locazioni successive di memoria</i>
<b>.asciiz</b> str	<i>Alloca la stringa str in memoria, terminata con il valore 0</i>
<b>.space</b> n	<i>Alloca n byte, senza inizializzazione</i>



# MARS

Direttive – Tipi di dati: esempi

- ❑ L'impiego degli operandi con diversi tipi avviene come nel seguente esempio:

**.data**

Val8bit: **.byte** 127

Val16bit: **.half** -33000

Val32bit: **.word** 5678

Array32bit : **.word** 1000202, 52462, 3876865

Stringa: **.asciiz** "Ciao a tutti"

Vettore10byteliberi: **.space** 10



# MARS

## Etichette

- ❑ Una **etichetta** (specificata da un identificatore ovvero una stringa alfanumerica che inizia per un carattere alfabetico) individua un punto del programma in cui si trova, cioè un indirizzo
- ❑ Una etichetta consiste in un identificatore seguito dal simbolo due punti  $\{A, \dots, Z, a, \dots, z\} \{A, \dots, Z, a, \dots, z, 0, \dots, 9\}^* \{:\}$

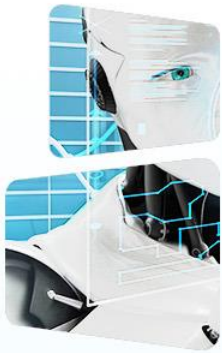
Esempio: main:, loop:, store:, Size:, Array:, Result:, ...



# MARS

## Etichette

- ☐ L'etichetta può avere una **visibilità locale** o una **visibilità globale**
- ☐ Le etichette sono locali; l'uso della direttiva **.globl** rende un'etichetta globale
- ☐ Una etichetta locale può essere referenziata solo dall'interno del file in cui è definita; una etichetta globale può essere referenziata anche da file diversi da quello in cui è definita (*etichette globali*)



# MARS

## Etichette - esempio

```
.text  
.globl main  
main:
```

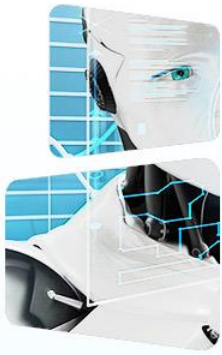
```
loop:
```

```
store:
```

```
.end
```

```
lw $a0, Size  
li $a1, 0  
li $a2, 0  
li $t2, 4  
  
mul $t1, $a1, $t2  
lw $a3, Array($t1)  
add $a2, $a2, $a3  
add $a1, $a1, 1  
beq $a1, $a0, store  
j loop  
  
sw $a2, Result
```

```
.data  
Array: .word 1, 2, 3, 4, 5  
Size: .word 4  
Result: .word 0
```

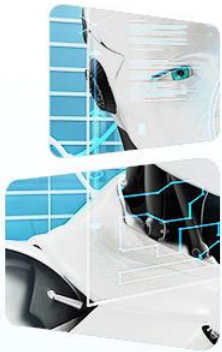


# MARS

Etichetta come indirizzo di operando

- ☐ Una etichetta individua anche una locazione di memoria nella quale sono stipati gli operandi





# MARS

Etichetta come indirizzo di operando

```
.text  
.globl main  
main:
```

```
    lw $a0, Size
```

```
    li $a1, 0
```

```
    li $a2, 0
```

```
    li $t2, 4
```

```
loop:
```

```
    mul $t1, $a1, $t2
```

```
    lw $a3, Array($t1)
```

```
    add $a2, $a2, $a3
```

```
    add $a1, $a1, 1
```

```
    beq $a1, $a0, store
```

```
    j loop
```

```
store:
```

```
    sw $a2, Result
```

```
.end
```

```
.data
```

```
Array: .word 1, 2, 3, 4, 5
```

```
Size: .word 4
```

```
Result: .word 0
```



# MARS

## Registri

- ❑ I registri contengono dati o indirizzi e sono suddivisi in
  - ❑ **registri generali:** possono essere utilizzati in qualunque istruzione, a scelta del programmatore (sebbene esistano delle convenzioni)
  - ❑ **registri speciali:** hanno istruzioni dedicate per essere utilizzati
    - ❖ Esistono istruzioni speciali per gestire i registri speciali:
      - Istruzioni “Branch” e “Jump” per il PC
      - Istruzioni mthi, mtlo, mfhi, mflo per LO ed HI



# MARS

## Registri nel MARS

Nome reale	Alias	Significato
\$0	\$zero	Valore fisso a 0
\$1	\$at	Riservato
\$2-\$3	\$v0-\$v1	Risultati di una funzione
\$4-\$7	\$a0-\$a3	Argomenti di una funzione
\$8-\$15	\$t0-\$t7	Temporanei (non preservati fra chiamate di funzioni)
\$16-\$23	\$s0-\$s7	Temporanei (preservati fra le chiamate di funzioni)
\$24-\$25	\$t8-\$t9	Temporanei (non preservati fra chiamate di funzioni)
\$26-\$27	\$k0-\$k1	Riservate per OS kernel
\$28	\$gp	Pointer to global area
\$29	\$sp	Stack pointer
\$30	\$fp	Frame pointer
\$31	\$ra	Return address

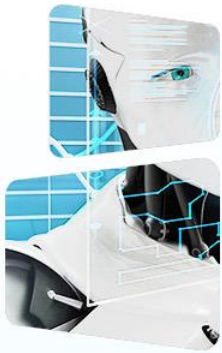


# MARS

## Registri generali

☐ I **registri speciali**: hanno istruzioni dedicate per essere utilizzati

Registro Speciale	Significato
PC	<i>Program counter</i>
HI	<i>Risultato di una moltiplicazione, parte più significativa</i>
LO	<i>Risultato di una moltiplicazione, parte meno significativa</i>
SP	<i>Stack Pointer</i>
RA	<i>Indirizzo di ritorno (usato per memorizzare il valore del PC dopo la chiamata di sub-routine)</i>



# MARS

## Registri

```
.text  
.globl main  
main:
```

```
    lw $a0, Size  
    li $a1, 0  
    li $a2, 0  
    li $t2, 4
```

```
loop:
```

```
    mul $t1, $a1, $t2  
    lw $a3, Array($t1)  
    add $a2, $a2, $a3  
    add $a1, $a1, 1  
    beq $a1, $a0, store  
    j loop
```

```
store:
```

```
    sw $a2, Result
```

```
.end
```

```
.data
```

```
Array: .word 1, 2, 3, 4, 5
```

```
Size: .word 4
```

```
Result: .word 0
```



# MARS

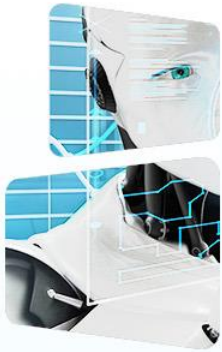
## Istruzioni

- ❑ Una **istruzione** inizia con una parola riservata (keyword) che corrisponde all'OPCODE e continua a seconda della sua sintassi

*Esempio:*

**bne reg1, reg2, address** (*branch if not equal*)

- ❑ Ad ogni istruzione del linguaggio macchina MIPS **corrisponde una** istruzione del linguaggio assembly



# MARS

## Istruzioni

```
.text  
.globl main  
main:
```

```
lw $a0, Size  
li $a1, 0  
li $a2, 0  
li $t2, 4
```

```
loop:
```

```
mul $t1, $a1, $t2  
lw $a3, Array($t1)  
add $a2, $a2, $a3  
add $a1, $a1, 1  
beq $a1, $a0, store  
j loop
```

```
store:
```

```
sw $a2, Result
```

```
.end
```

```
.data
```

```
Array: .word 1, 2, 3, 4, 5
```

```
Size: .word 4
```

```
Result: .word 0
```





# MARS

## Pseudo- Istruzioni

- ❑ Una **pseudoistruzione** è una istruzione fornita dall'assemblatore ma non direttamente implementata

*Esempio:*

**blt reg1, reg2, address** (*branch if less than*)

*diventa:*

**slt \$at, reg1, reg2** (*set less than*)

**bne \$at, \$zero, address** (*branch if not equal*)

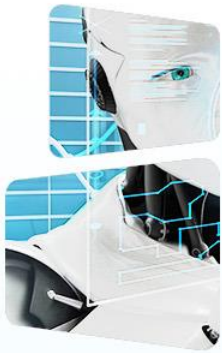
- ❑ In pratica una **pseudo-istruzione consta di due o più istruzioni**



# MARS

## Commenti

- ☐ I **commenti** sono utili per comprendere i singoli passi o l'intero programma (*furono un punto di svolta per la programmazione*)
- ☐ I commenti non sono inclusi nel modulo oggetto
- ☐ Sintassi:  
**#Commento**



# MARS

## Commenti

```
.text
.globl main
main:
```

```
    lw $a0, Size
    li $a1, 0
    li $a2, 0
    li $t2, 4
```

```
loop:
```

```
    mul $t1, $a1, $t2
    lw $a3, Array($t1)
    add $a2, $a2, $a3
    add $a1, $a1, 1
    beq $a1, $a0, store
    j loop
```

```
store:
```

```
    sw $a2, Result
```

```
.end
```

```
.data
```

```
Array: .word 1, 2, 3, 4, 5
```

```
Size: .word 4
```

```
Result: .word 0
```

#conservo valore su cui operare

#sommo

#sommo il valore 1

#se è uguale a zero salto alla parte finale

#salto incondizionato all'inizio



# MARS

## Pre-assemblaggio

- ❑ Prima della fase di assemblaggio a doppia passata si effettua un **pre-assemblaggio** dove accadono queste operazioni:
  - ❑ si risolvono le macro
  - ❑ si risolvono le pseudo istruzioni
  - ❑ si includono eventuali file esterni
  - ❑ si analizzano le direttive

Fine

