



Università degli Studi di Milano  
Dipartimento di Informatica "Giovanni Degli Antoni"  
Corso di Laurea Triennale in Informatica

# Architettura degli Elaboratori II

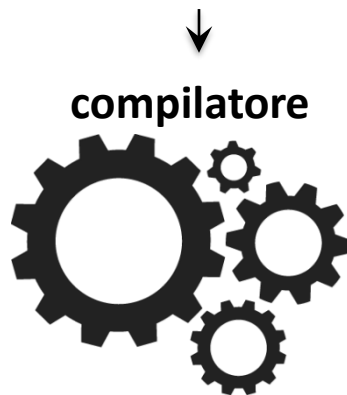
## Laboratorio

# Progettare e assemblare software in MIPS

# Introduzione

*Linguaggio di alto livello*

```
int main()  
{  
    cout << "Hello world!" << endl;  
    return 0;  
}
```



*Assembly*

```
multi $2, $5, 4  
add $2, $4, $2  
lw $15, 0($2)  
lw $16, 4($2)  
sw $16, 0($2)  
sw $15, 4($2)  
jr $31
```



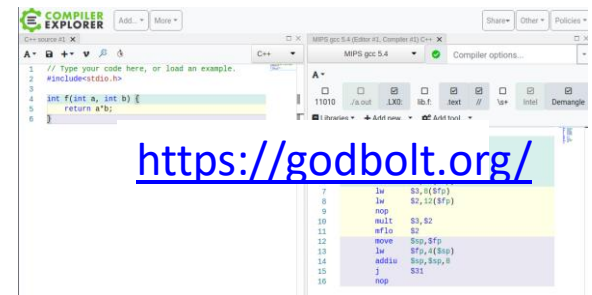
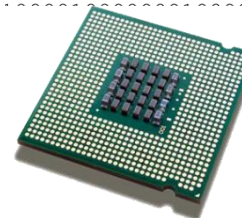
**Assembler + linker**



*Linguaggio macchina*

```
000011111111001000001000  
0010000011111111111101  
1011100100000000000110  
0000010000010000000000  
1010000000000000000110  
0001000000000000011011
```

**hardware**

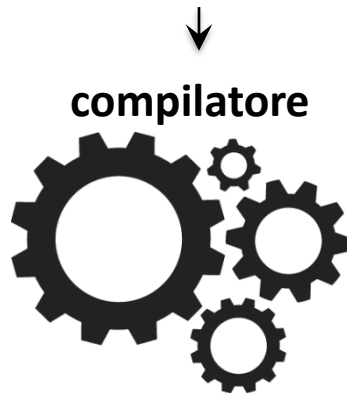


*Livello più basso (vicino all'hardware)  
dove poter programmare le istruzioni  
di un elaboratore*

# Introduzione

*Linguaggio di alto livello*

```
int main()  
{  
    cout << "Hello world!" << endl;  
    return 0;  
}
```



*Assembly*

```
multi $2, $5, 4  
add $2, $4, $2  
lw $15, 0($2)  
lw $16, 4($2)  
sw $16, 0($2)  
sw $15, 4($2)  
jr $31
```

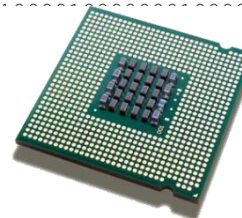


**Assembler + linker**



*Linguaggio macchina*

```
000011111111001000001000  
0010000011111111111101  
10111001000000000000110  
00000100000100000000000  
10100000000000000000110  
00010000000000000011011
```



*Livello più basso (vicino all'hardware)  
dove poter programmare le istruzioni  
di un elaboratore*

Laboratorio di Architettura 2

# Assembly

È la rappresentazione simbolica del linguaggio macchina di un elaboratore.

```
add $t0 $s2 $s3
```

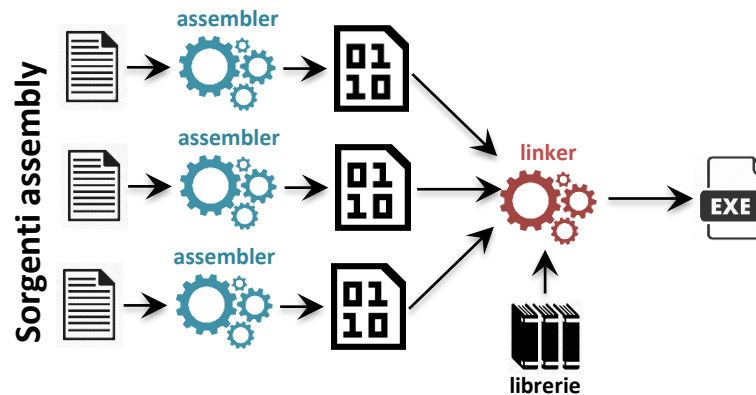
Binary: 00000010010100110100000000100000

Hex: 0x02534020

 [MIPS instruction converter](#)

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL 000000	\$s2 10010	\$s3 10011	\$t0 01000	0 00000	ADD 100000	
6	5	5	5	5	6	

- Dà alle istruzioni una forma *human-readable* e permette di usare **label** per referenziare con un nome parole di memoria che contengono istruzioni o dati.



- Programmi coinvolti:
  - assembler**: «traduce» le istruzioni assembly (da un **file sorgente**) nelle corrispondenti istruzioni macchina in formato binario (in un **file oggetto**);
  - linker**: combina i files oggetto e le librerie in un **file eseguibile** dove la «destinazione» di ogni label è determinata.

# Assembly

- Il codice Assembly può essere il risultato di due processi:
  - *target language* del compilatore che traduce un programma in linguaggio di alto livello (C, Pascal, ...) nell'equivalente assembly;
  - *linguaggio di programmazione* usato da un programmatore.
- Assembly è stato l'approccio principale con cui scrivere i programmi per i primi computer.
- Oggi la complessità dei programmi, la disponibilità di compilatori sempre migliori e di memoria rendono conveniente programmare in linguaggi di alto livello.
- Assembly come linguaggio di programmazione è adatto in certi casi particolari:
  - ottimizzare le performance (anche in termini di prevedibilità) e spazio occupato da un programma (ad es., sistemi embedded);
  - eredità di certi sistemi vecchi, ma ancora in uso, dove Assembly rappresenta l'unico modo conveniente per scrivere programmi;
  - rendere più efficienti certe istruzioni che hanno una semantica di basso livello.

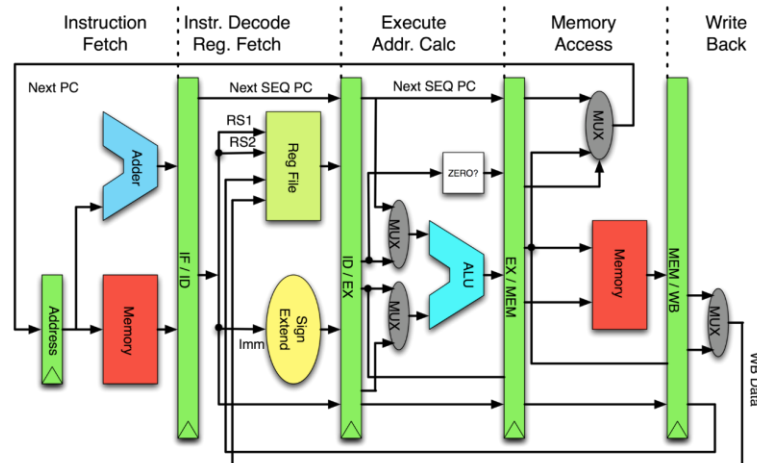
# MIPS



- In questo laboratorio lavoreremo con **MIPS** (Multiprocessor without Interlocked Pipeline Stages): un'ISA di tipo RISC
- Nasce a metà anni '80 come architettura *general purpose*;
- Inizialmente è un progetto accademico (Stanford), poco dopo diventa commerciale
- Oggi è impiegata prevalentemente nell'ambito dei *sistemi embedded*

# MIPS

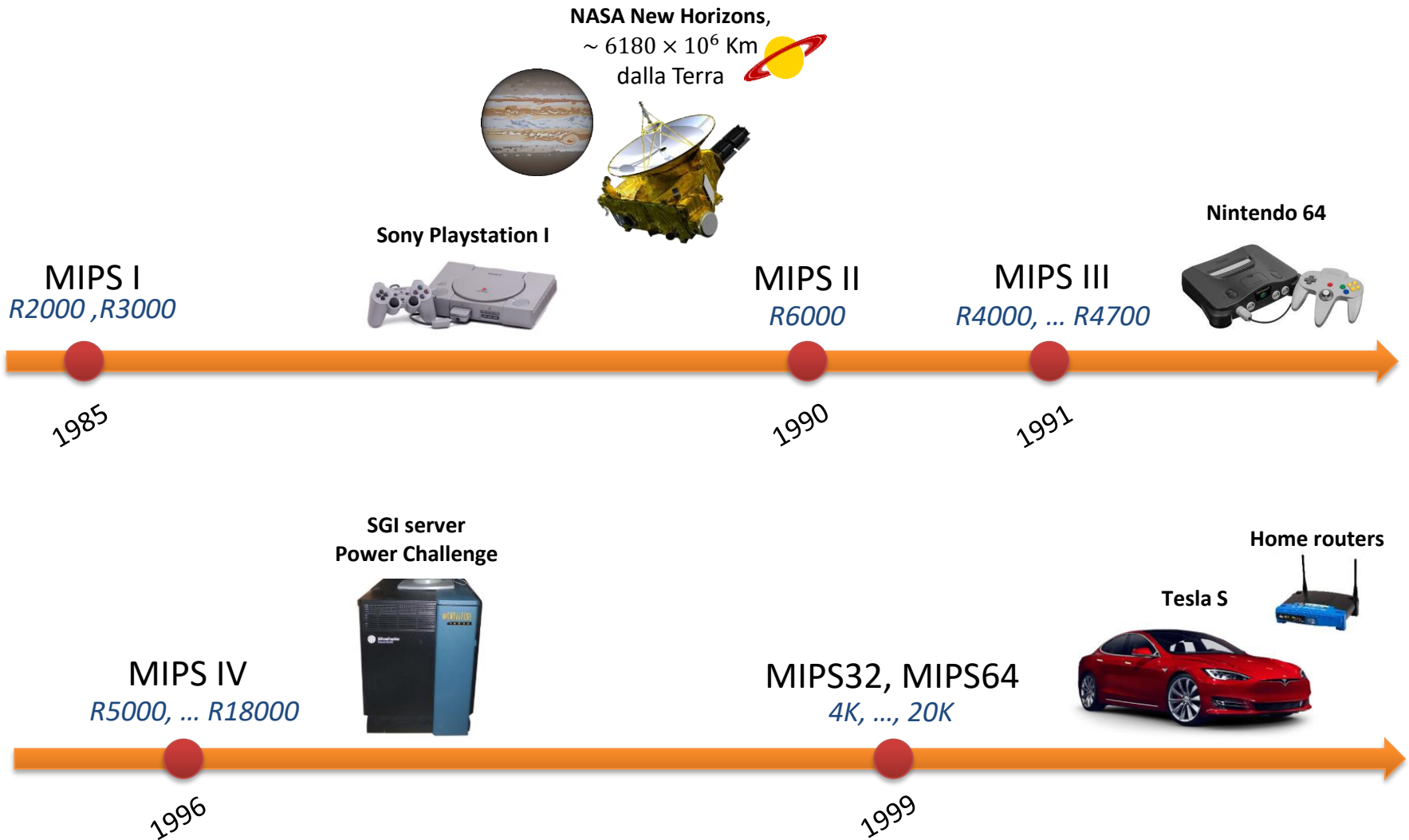
- La maggior parte dei corsi accademici di architetture adotta MIPS, perché?



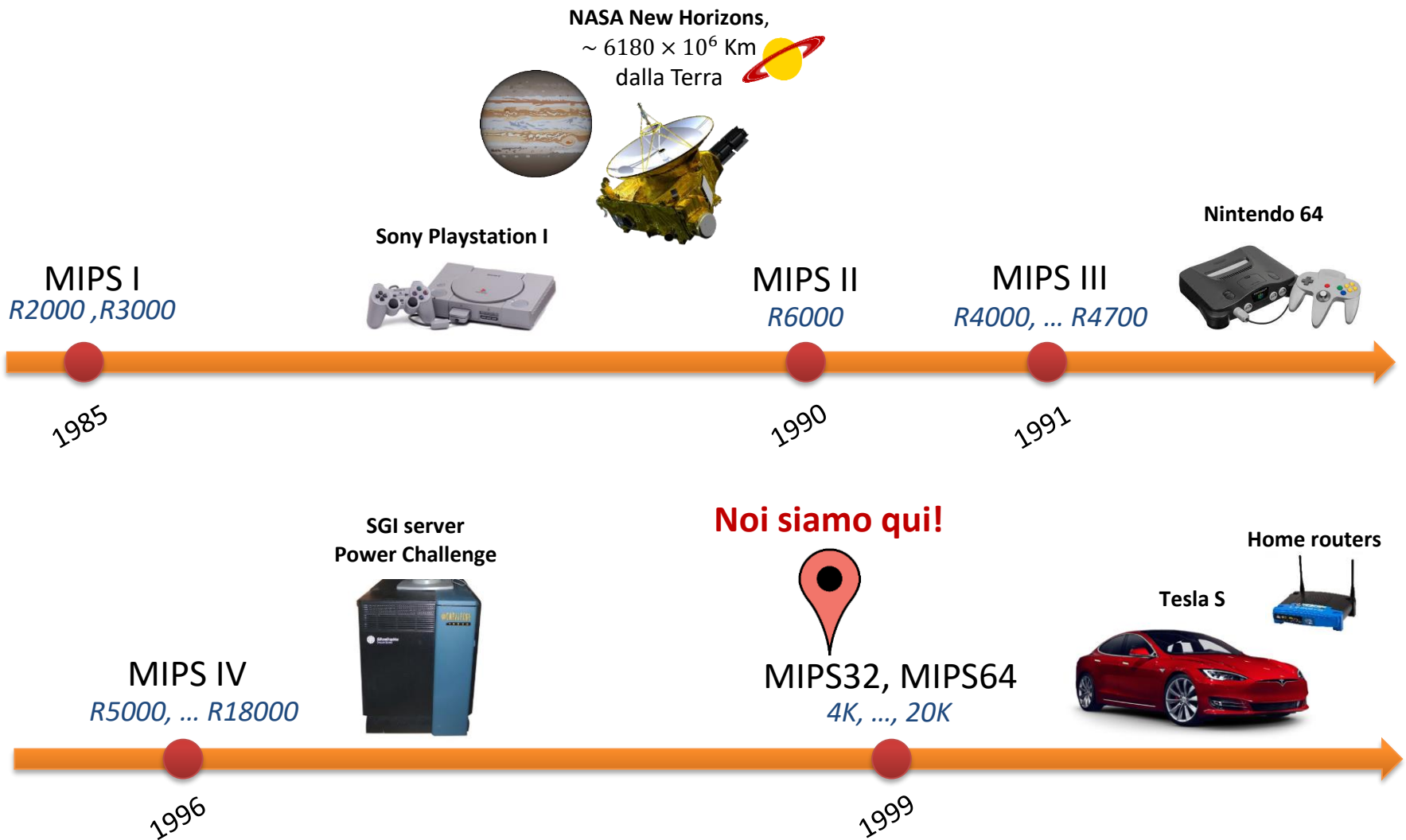
- È una prima e lineare implementazione del concetto di pipeline
- È costruita su una semplice assunzione: ogni stadio della pipeline deve terminare in un ciclo di clock, ogni stadio non necessita di attendere il completamento degli altri (interlock)
- (Oggi l'assunzione è rilassata per avere istruzioni come moltiplicazione e divisione, ma il nome è rimasto lo stesso)*



# MIPS: passato e presente

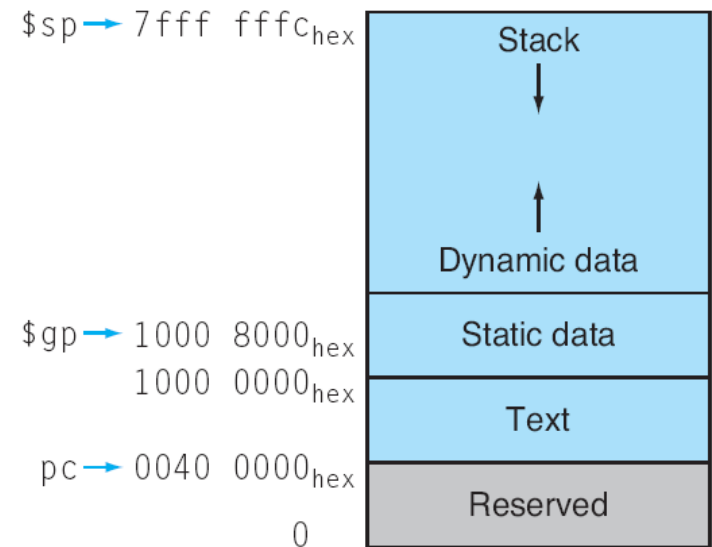


# MIPS: passato e presente



# Il programma in memoria (in MIPS)

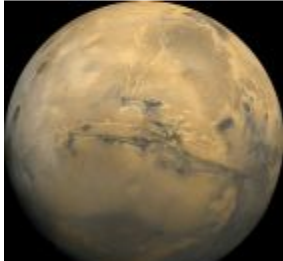
- **Segmento testo:** contiene le **istruzioni** del programma.
- **Segmento dati:**
  - **dati statici:** contiene dati la cui dimensione è conosciuta a *compile time* e la cui durata coincide con quella del programma (*e.g., variabili statiche, costanti, etc.*);
  - **dati dinamici:** contiene dati 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.*).



# MARS



Missouri State  
UNIVERSITY



## ***MARS (MIPS Assembler and Runtime Simulator)***

### ***An IDE for MIPS Assembly Language Programming***

MARS is a lightweight interactive development environment (IDE) for programming in MIPS assembly language, intended for educational-level use with Patterson and Hennessy's *Computer Organization and Design*.

- È un emulatore di una CPU che obbedisce alle convenzioni MIPS32
- Perché usare un emulatore e non la macchina vera?
  - Usiamo tutti la stessa ISA indipendentemente dal calcolatore reale.
  - Ci offre una serie di strumenti che rendono la programmazione più comoda.
  - Maschera certi aspetti reali a cui non saremmo interessati (es., delays).
- Disponibile a questo URL <http://courses.missouristate.edu/KenVollmar/MARS/index.htm>

# MARS (interfaccia)

File Edit Run Settings Tools Help

Run speed at max (no interaction)

Edit Execute

Text Segment

Bkpt	Address	Code	Basic	Source
<input type="checkbox"/>	0x00400000	0x24020004	addiu \$2,\$0,4	7: li \$v0, 4
<input type="checkbox"/>	0x00400004	0x3c011001	lui \$1,4097	8: la \$a0, hello
<input type="checkbox"/>	0x00400008	0x34240000	ori \$4,\$1,0	
<input type="checkbox"/>	0x0040000c	0x0000000c	syscall	9: syscall
<input type="checkbox"/>	0x00400010	0x2402000a	addiu \$2,\$0,10	11: li \$v0, 10
<input type="checkbox"/>	0x00400014	0x0000000c	syscall	12: syscall

Memoria (Text e Data)

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)
0x10010000	1818576906	539783020	1819438935	663908	0
0x10010020	0	0	0	0	0
0x10010040	0	0	0	0	0
0x10010060	0	0	0	0	0

Mars Messages Run I/O

Assemble: assembling /home/nbas/git/archlab/arch2lab/session\_01/00-hello/hello.asm

Assemble: operation completed successfully.

Go: running hello.asm

Go: execution completed successfully.

Clear

Registers Coproc 1 Coproc 0

Name	Number	Value
\$zero	0	0
\$at	1	268500992
\$v0	2	10
\$v1	3	0
\$a0	4	268500992
\$a1	5	0
\$a2	6	0
\$a3	7	0
\$t0	8	0
\$t1	9	0
\$t2	10	0
\$t3	11	0
\$t4	12	0
\$t5	13	0
\$t6	14	0
\$t7	15	0
\$s0	16	0
\$s1	17	0
\$s2	18	0
\$s3	19	0
\$s4	20	0
\$s5	21	0
\$s6	22	0
\$s7	23	0
\$t8	24	0
\$t9	25	0
\$k0	26	0
\$k1	27	0
\$gp	28	268468224
\$sp	29	2147479548
\$fp	30	0
\$ra	31	0
pc		4194328
hi		0
lo		0

Logs e console (I/O)

Banco Registri

# MARS (Registri)

Registers	Coproc 1	Coproc 0
Name	Number	Value
\$zero	0	0
\$at	1	268500992
\$v0	2	10
\$v1	3	0
\$a0	4	268500992
\$a1	5	0
\$a2	6	0
\$a3	7	0
\$t0	8	0
\$t1	9	0
\$t2	10	0
\$t3	11	0
\$t4	12	0
\$t5	13	0
\$t6	14	0
\$t7	15	0
\$s0	16	0
\$s1	17	0
\$s2	18	0
\$s3	19	0
\$s4	20	0
\$s5	21	0
\$s6	22	0
\$s7	23	0
\$t8	24	0
\$t9	25	0
\$k0	26	0
\$k1	27	0
\$gp	28	268468224
\$sp	29	2147479548
\$fp	30	0
\$ra	31	0
pc		4194328
hi		0
lo		0

- 32 registri a 32bit per operazioni su interi ( **\$0..\$31** ).
- 32 registri a 32 bit per operazioni in virgola mobile sul coprocessore 1 (**\$FP0..\$FP31** ).
- registri speciali a 32bit:
  - il **Program Counter (PC)** l'indirizzo della prossima istruzione da eseguire;
  - **hi** e **lo** usati nella moltiplicazione e nella divisione;
  - **EPC, Cause, BadVAddr, Status** (coprocessore 0) vengono usati nella gestione delle eccezioni.
- I registri general-purpose sono chiamati col nome dato dalla convenzione MIPS e numerati da 0 a 31
- Il loro valore è ispezionabile nel formato esadecimale o decimale

# Richiamo di istruzioni aritmetiche (somma, sottrazione)

- Convenzioni di notazione:
  - Identificativo con iniziale minuscola: deve essere un registro o un valore immediato (intero con segno su 16 bit);
  - Identificativo con iniziale «\$»: deve essere un registro.

`add $s1, $s2, s3`      #  $\$s1 = \$s2 + s3$ , rileva overflow

`sub $s1, $s2, s3`      #  $\$s1 = \$s2 - s3$ , rileva overflow

`addi $s1, $s2, 13`      #  $\$s1 = \$s2 + 13$ , rileva overflow

`addu $s1, $s2, s3`      #  $\$s1 = \$s2 + s3$ , unsigned, non rileva overflow

`subu $s1, $s2, s3`      #  $\$s1 = \$s2 - s3$ , unsigned, non rileva overflow

`addui $s1, $s2, 27`      #  $\$s1 = \$s2 + 27$ , unsigned, non rileva overflow

# Istruzioni: moltiplicazione

- Due istruzioni:
  - `mult $rs $rt`
  - `multu $rs $rt` # unsigned
- Il registro destinazione è **implicito**.
- Il risultato della moltiplicazione viene posto sempre in due registri dedicati di una parola (special purpose) denominati **hi (High order word)** e **lo (Low order word)**.
- La moltiplicazione di due numeri rappresentabili con 32 bit può dare come risultato un numero non rappresentabile in 32 bit.



# Istruzioni: moltiplicazione

- Il risultato della moltiplicazione si preleva dal registro **hi** e dal registro **lo** utilizzando le due istruzioni:

– **mfhi \$rd** # move from hi

- sposta il contenuto del registro **hi** nel registro **rd**;

– **mflo \$rd** # move from lo

- sposta il contenuto del registro **lo** nel registro **rd**.

Test sull'overflow

Risultato del prodotto

# Operazioni aritmetiche: divisione

`div $s2, $s3`      `# $s2 / $s3, divisione intera`

- Il risultato della divisione intera va in:
  - Lo: `$s2 / $s3` [quoziente];
  - Hi: `$s2 mod $s3` [resto].
- Il risultato va quindi prelevato dai registri Hi e Lo utilizzando ancora la `mfhi` e la `mflo`.

# Istruzioni: pseudo-istruzioni

- Le pseudoistruzioni sono un modo compatto ed intuitivo di specificare un insieme di istruzioni.
- La traduzione della pseudoistruzione nelle istruzioni equivalenti è attuata automaticamente dall'assemblatore.

## Esempi:

- `move $t0, $t1` # pseudo istruzione
  - `add $t0, $zero, $t1` # (in alternativa) addi \$t0, \$t1, 0
- `mul $s0, $t1, $t2` # pseudo istruzione
  - `mult $t1, $t2`
  - `mflo $s0`
- `div $s0, $t1, $t2` # pseudo istruzione
  - `div $t1, $t2`
  - `mflo $s0`

# Primo programma in Assembly

Indirizzamento, lettura  
e scrittura della memoria

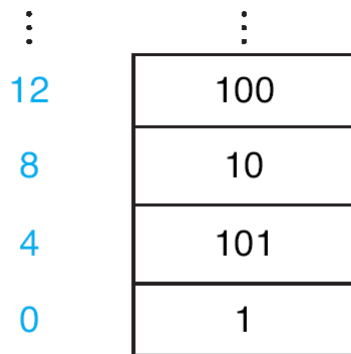
# Organizzazione della memoria

- 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



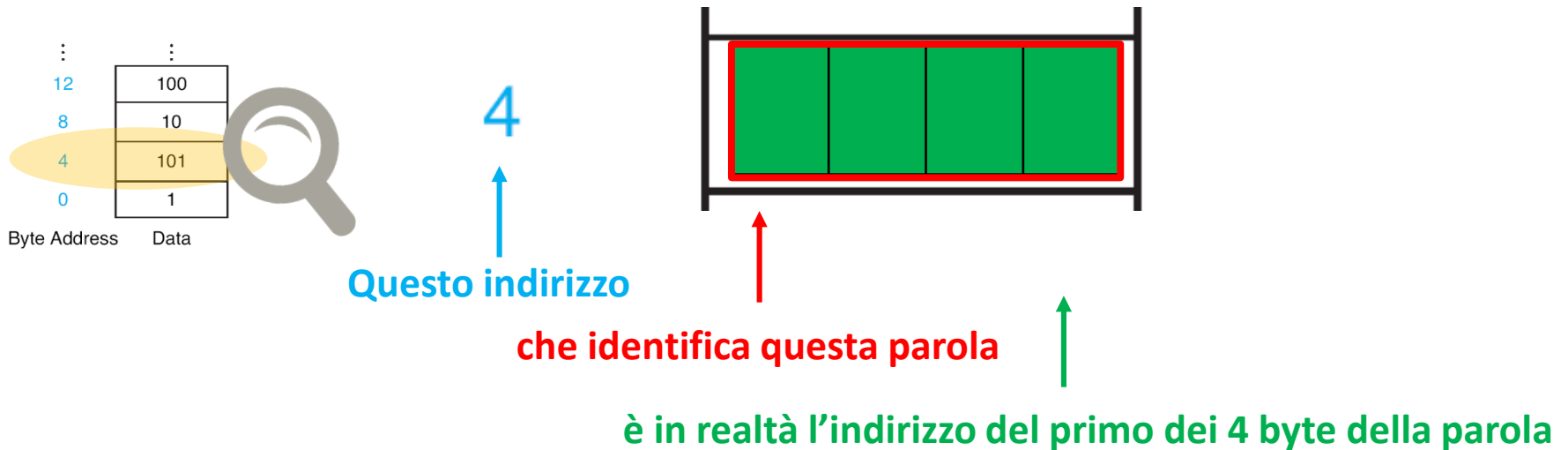
Byte Address      Data

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



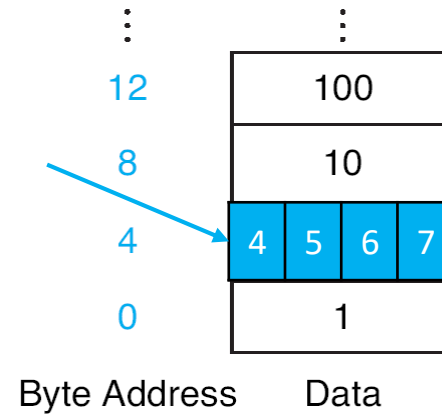
- Ma, tra i 4, quale è il **primo byte**? La risposta sta nell'ordine dei byte: la **endianness**



# Endianness

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

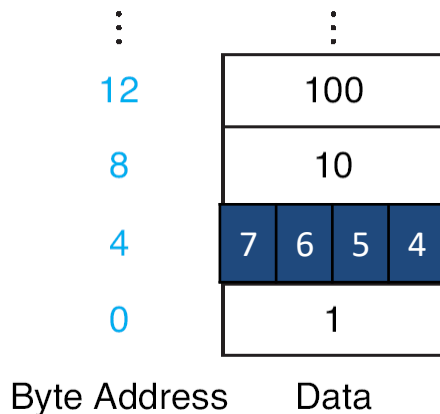
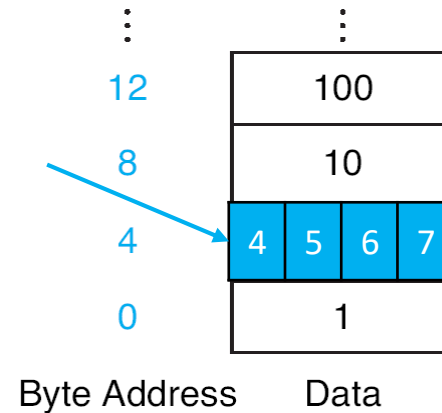
**Big endian:** il primo byte è quello **più** significativo (quello più a **sinistra**, **big end**)



# 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**)

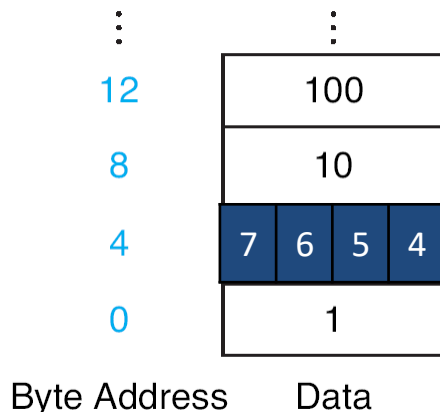
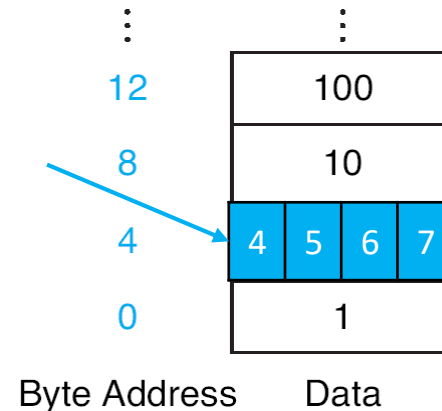


**Little endian:** il primo byte è quello **meno** significativo (quello più a **destra**, **little end**)

# 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 emulatore MARS (e anche SPIM) eredita la endianness della macchina su cui è eseguito

# 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

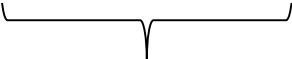
- Si consideri un vettore  $v$  dove ogni elemento  $v[i]$  è una parola di memoria (32 bit).
- Obiettivo: leggere/scrivere  $v[i]$  (elemento alla posizione  $i$  nell'array).
- Gli array sono memorizzati in modo sequenziale:
  - $b$ : registro base di  $v$ , è anche l'indirizzo di  $v[0]$ ;
  - l'elemento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

# 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  
addi $s2, $zero, 0x10000004
```

  
32 bit

`# $s1 = &h`


`# $s2 = 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    # $s1 = &h  
addi $s2, $zero, 0x10000004    # $s2 = 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    # $s1 = &h  
addi $s2, $zero, 0x10000004    # $s2 = 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?

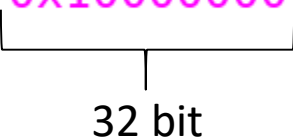
0x00400000	0x3c011000	lui \$1, 0x00001000
0x00400004	0x34210000	ori \$1, \$1, 0x00000000
0x00400008	0x00018820	add \$17, \$0, \$1
0x0040000c	0x3c011000	lui \$1, 0x00001000
0x00400010	0x34210004	ori \$1, \$1, 0x00000004
0x00400014	0x00019020	add \$18, \$0, \$1



# Inizializzazione esplicita degli indirizzi

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

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



A bracket is drawn under the hexadecimal value 0x10000000, with a vertical line extending from the center of the bracket to the text "32 bit" below it.

32 bit

la \$s1, 0x10000000	# \$s1 <- &h
la \$s2, 0x10000004	# \$s2 <- 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*

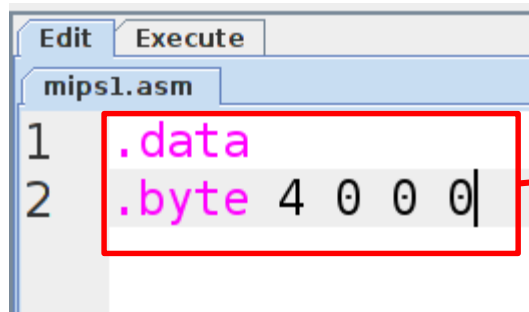


# Direttive Assembler

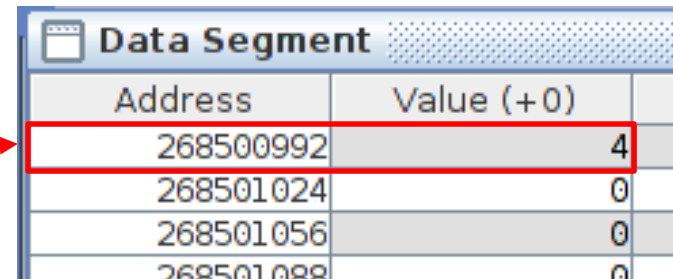
- `.data` specifica che ciò che segue nel file sorgente è il segmento dati: vengono specificati gli elementi presenti in tale segmento (stringe, array, etc ...).
- `.text` specifica che ciò che segue nel file sorgente è il segmento testo
- **STRINGA:** `.asciiz "stringa_di_esempio"` memorizza la stringa "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).

# La direttiva `.byte` e la endianness

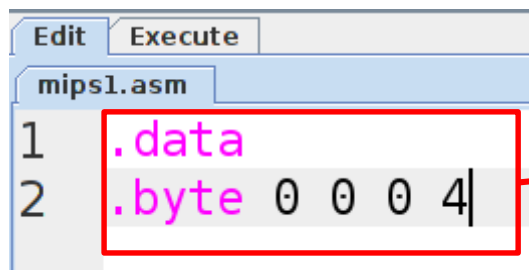
- Testiamo la endianness della macchina su cui stiamo lavorando (*in Linux: comando «lscpu», proprietà «Byte order»*):
- Cerchiamo di allocare la costante 4 in una **parola** di memoria usando la direttiva `byte` che permette di inserire parole di memoria specificando il valore di ogni singolo byte che la compone:



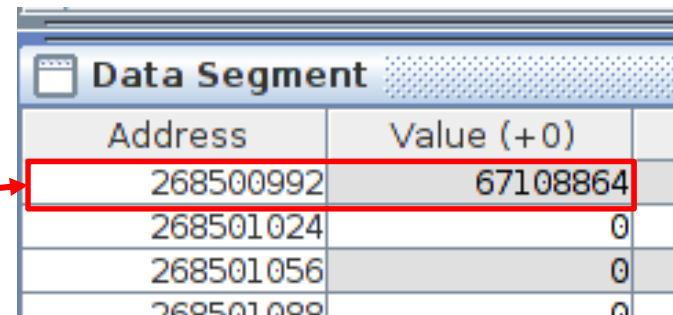
```
1 .data
2 .byte 4 0 0 0
```



Address	Value (+0)
268500992	4
268501024	0
268501056	0
268501088	0



```
1 .data
2 .byte 0 0 0 4
```



Address	Value (+0)
268500992	67108864
268501024	0
268501056	0
268501088	0

# La direttiva `.byte` e la endianness

`.data`  
A: `.byte` 0 0 0 0 4 0 0 0 8 0 0 0 12 0 0 0

Least Significant Byte

Most Significant Byte

A + 12	0	0	0	12
A + 8	0	0	0	8
A + 4	0	0	0	4
A + 0	0	0	0	0

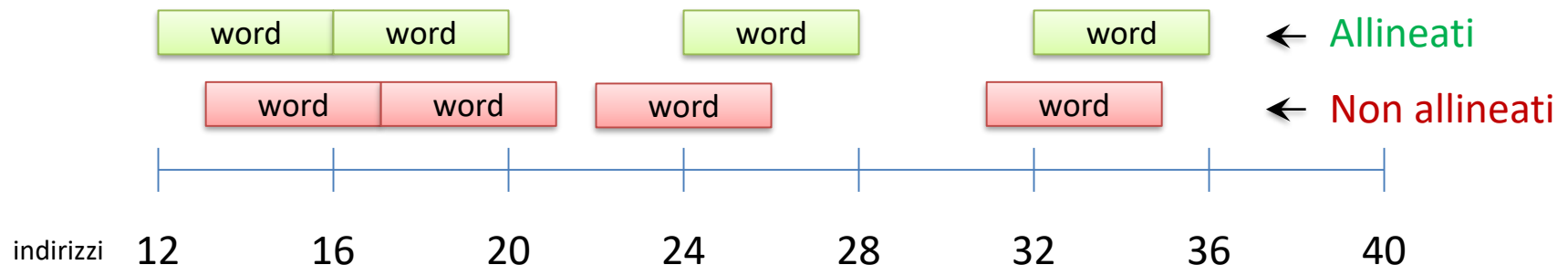
Byte Address      Data

**Attenzione!**

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, ...).

# Allineamento dati

- L'accesso a memoria si dice allineato su  $n$  byte se:
  - ogni dato ha dimensione  $n$  byte
  - $n$  è una potenza di 2
  - l'indirizzo di ogni dato è multiplo di  $n$
- Nel nostro caso:
  - un dato è una word che ha dimensione 4 byte, quindi  $n=4$
  - ( $4 = 2^2$ )
  - l'indirizzo di ogni word deve essere multiplo di 4



# Esempio

```
.data
string: .ascii "Ciao"
A: .space 8
```

```
.text
.globl main
```

```
main:
```

```
la $t0, A
li $t1, 5
sw $t1 0($t0)
```

# Esempio

Il segmento dati inizia qui  
(indirizzo **0x10010000**), i dati  
che seguono sono allocati in  
modo sequenziale



```
.data
string: .asciiz "Ciao"
A: .space 8

.text
.globl main

main:

la $t0, A
li $t1, 5
sw $t1 0($t0)
```



# Esempio

Il segmento dati inizia qui  
(indirizzo **0x10010000**), i dati  
che seguono sono allocati in  
modo sequenziale

```
.data
string: .asciiz "Ciao"
A: .space 8

.text
.globl main

main:

la $t0, A
li $t1, 5
sw $t1 0($t0)
```

La stringa «Ciao» verrà  
quindi allocata a partire  
dall'inizio del segmento:

Indirizzo	Valore
0x10010000	c
0x10010001	i
0x10010002	a
0x10010003	o
0x10010004	\0
0x10010005	prima word di A
0x10010009	seconda word di A

# Esempio

Il segmento dati inizia qui  
(indirizzo **0x10010000**), i dati  
che seguono sono allocati in  
modo sequenziale

```
.data
string: .asciiz "Ciao"
A: .space 8

.text
.globl main

main:

la $t0, A
li $t1, 5
sw $t1 0($t0)
```

La stringa «Ciao» verrà  
quindi allocata a partire  
dall'inizio del segmento:

Indirizzo	Valore
0x10010000	c
0x10010001	i
0x10010002	a
0x10010003	o
0x10010004	\0
0x10010005	prima word di A
0x10010009	seconda word di A

Proseguendo nel segmento dati  
incontriamo **A**, la prima posizione  
disponibile per allocarlo è nel byte  
all'indirizzo **0x10010005**



Convertendo in base 10 si  
osserva che non è multiplo di 4  
 $(0x10010005)_{16} = (268500997)_{10}$

# Esempio

Il segmento dati inizia qui (indirizzo **0x10010000**), i dati che seguono sono allocati in modo sequenziale

```
.data
string: .asciiz "Ciao"
A: .space 8
```

La stringa «Ciao» verrà quindi allocata a partire dall'inizio del segmento:

```
.text
.globl main

main:

la $t0, A
li $t1, 5
sw $t1, 0($t0)
```

Indirizzo	Valore
0x10010000	c
0x10010001	i
0x10010002	a
0x10010003	o
0x10010004	\0
0x10010005	prima word di A
0x10010009	seconda word di A

Proseguendo nel segmento dati incontriamo **A**, la prima posizione disponibile per allocarlo è nel byte all'indirizzo **0x10010005**

Convertendo in base 10 si osserva che non è multiplo di 4  
 $(0x10010005)_{16} = (268500997)_{10}$

Cosa succede se tento di accedere ad un indirizzo non allineato con **sw** o **lw** ?

# Esempio

```
.data
string: .asciiz "Ciao"
A: .space 8

.text
.globl main

main:

    la $t0, A
    li $t1, 5
    sw $t1 0($t0)
```

Go: running mips1.asm

Error in D:\Jacopo Essenziale\MEGA\MIPS\_Stuff\mars\mips1.asm line 8: Runtime exception at 0x0040000c: store address not aligned on word boundary 0x10010005

Go: execution terminated with errors.

Le istruzioni di **sw** e **lw** richiedono di operare con accesso allineato con parole da 32 bit, quindi se specifichiamo un indirizzo **non** multiplo di 4 in MARS otteniamo un errore.

# Esempio

```
string: .data
       .ascii "Ciao"
       .align 2
       A: .space 8

       .text
       .globl main

main:

       la $t0, A
       li $t1, 5
       sw $t1 0($t0)
```

Aggiungendo la direttiva di allineamento viene lasciato spazio libero per mantenere l'allineamento

Ora A viene allocato all'indirizzo  
 $(0x10010008)_{16} = (67125250)_{10}$   
che è multiplo di 4

Indirizzo	Valore
0x10010000	c
0x10010001	i
0x10010002	a
0x10010003	o
0x10010004	\0
0x10010005	
0x10010006	
0x10010007	
0x10010008	prima word di A
0x1001000C	seconda word di A



Università degli Studi di Milano  
Dipartimento di Informatica "Giovanni Degli Antoni"  
Corso di Laurea Triennale in Informatica

# Architettura degli Elaboratori II

## Laboratorio