Laboratorio di Architettura degli Elaboratori

- Tutorial sulle funzionalità di base del simulatore MARS
  - ✓ Direttive all'assemblatore

Prof. **Davide Bertozzi** davide.bertozzi@unife.it

I Registri

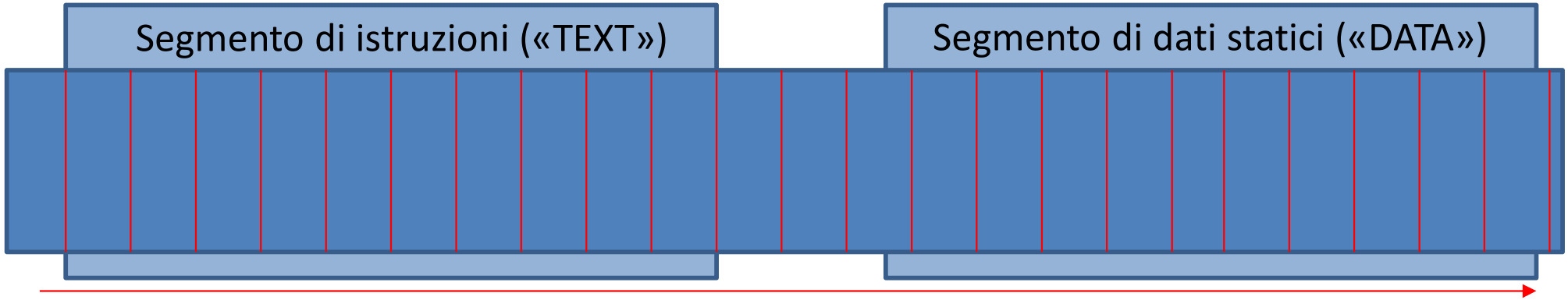
In Assembler abbiamo accesso diretto ai **32 registri a 32 bit del MIPS.**

Register Name	Software Name (from regdef.h)	Use and Linkage
$0		Always has the value 0.
$at		Reserved for the assembler.
$2..$3	v0-v1	Used for expression evaluations and to hold the integer type function results. Also used to pass the static link when calling nested procedures.
$4..$7	a0-a3	Used to pass the first 4 words of integer type actual arguments, their values are not preserved across procedure calls.
$8..$15	t0-t7	Temporary registers used for expression evaluations; their values aren't preserved across procedure calls.
$16..$23	s0-s7	Saved registers. Their values must be preserved across procedure calls.
$24..$25	t8-t9	Temporary registers used for expression evaluations; their values aren't preserved across procedure calls.
$26..27 or $kt0..$kt1	k0-k1	Reserved for the operating system kernel.
$28 or $gp	gp	Contains the global pointer.
$29 or $sp	sp	Contains the stack pointer.
$30 or $fp	fp	Contains the frame pointer (if needed); otherwise a saved register (like s0-s7).
$31	ra	Contains the return address and is used for expression evaluation.

Direttive all'Assemblatore

Forniscono informazioni utili all'Assembler per gestire l'organizzazione del codice. Le direttive iniziano con il punto:

DIRETTIVA	DESCRIZIONE
`.text`	Inizio del bocco 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

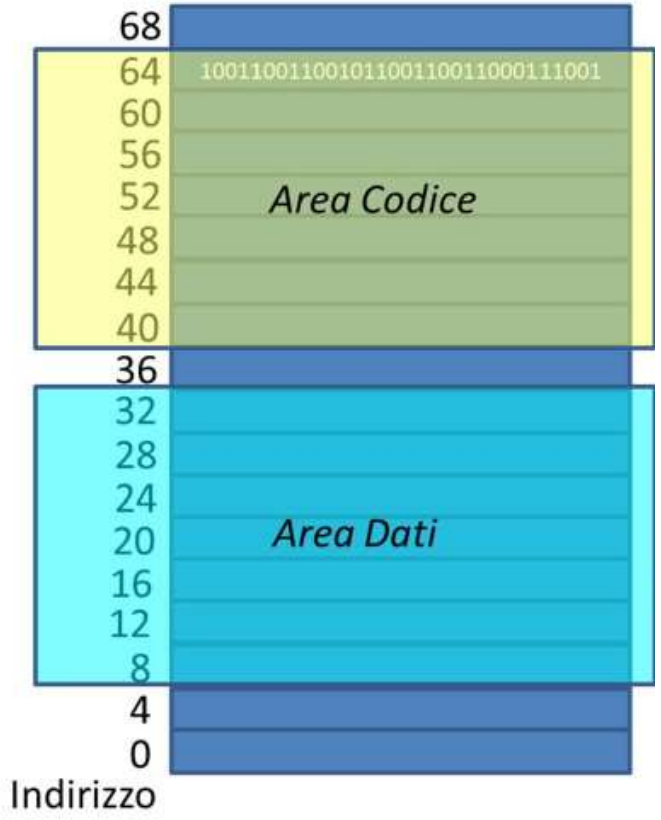


Array Lineare di Memoria: è diviso in segmenti (TEXT, DATA, ma anche STACK, HEAP,..)

Direttive Principali

```
.data
#allocare qui le variabili in memoria dati

.text
#scrivere qui il codice della memoria istruzioni
```



Equivalenze

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

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

Posso riscrivere il codice **in modo da poter utilizzare nomi meglio memorizzabili per i registri?**

Equivalenze

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

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

```
.eqv op1, $t0
.eqv op2, $t1
.eqv risultato, $t2

.text
addi op1, op1, 1
addi op2, op2, 2
add risultato, op1, op2
```

Laboratorio di Architettura degli Elaboratori

- Tutorial sulle funzionalità di base del simulatore MARS
  - ✓ Allocazione dei dati statici

Prof. **Davide Bertozzi** davide.bertozzi@unife.it

Allocazione Statica di Memoria

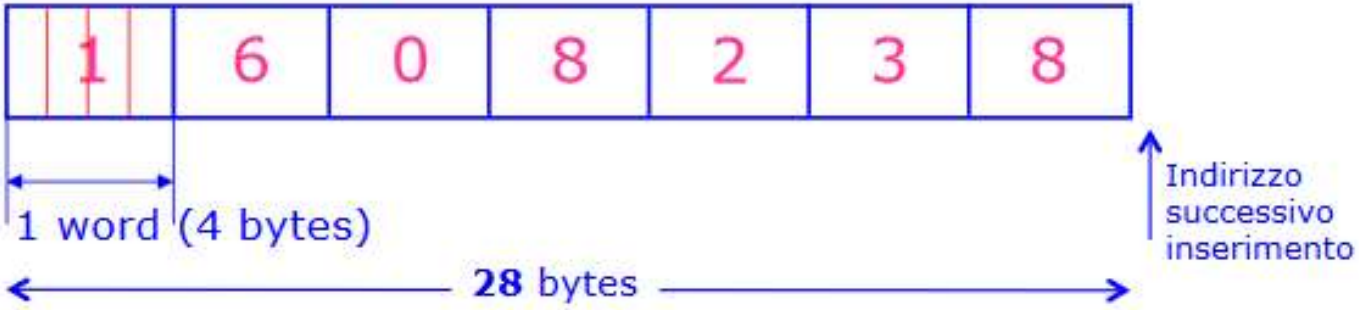
All'interno del segmento **.data** possiamo definire dati statici in questi modi:

- **Direttive di allocazione di memoria:**
  - `.byte` $b_1, \ldots, b_n$
    Alloca *n* quantità a 8 bit in byte successivi in memoria
  - `.half h1, …, hn`
    Alloca *n* quantità a 16 bit in halfword successive in memoria
  - `.word w1, …, wn`
    Alloca *n* quantità a 32 bit in word successive in memoria
  - `.float f1, …, fn`
    Alloca *n* valori floating point a singola precisione in locazioni successive in memoria
  - `.double d1, …, dn`
    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

Esempi

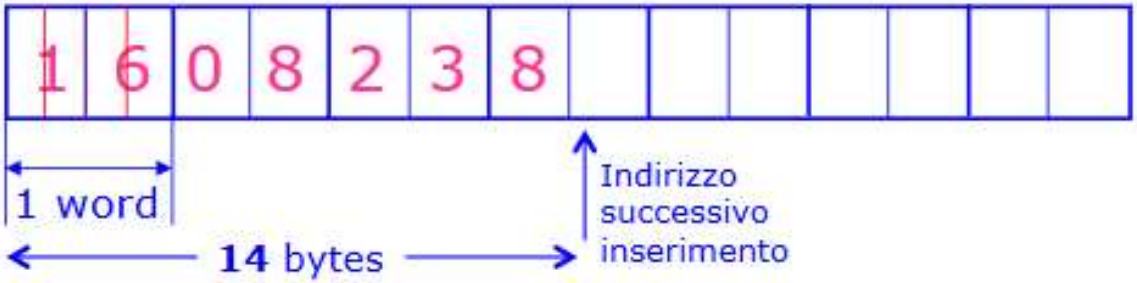
```
.word  1, 6, 0, 8, 2, 3, 8
```

Scrive 32 bit alla volta




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

Scrive 16 bit alla volta

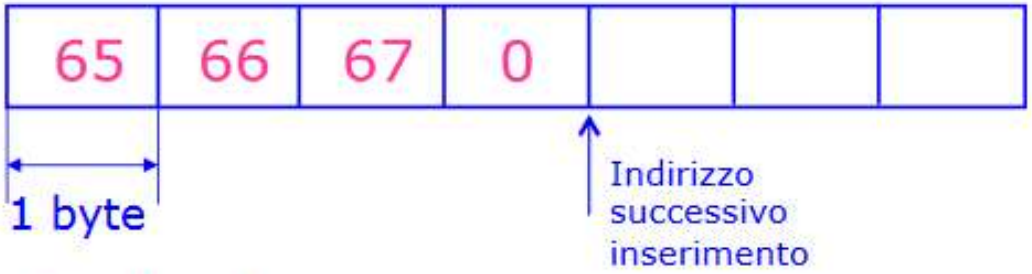



```
.byte  1, 6, 0, 8, 2, 3, 8
```

Scrive 8 bit alla volta

Esempi

```
.asciiz "ABC"
```

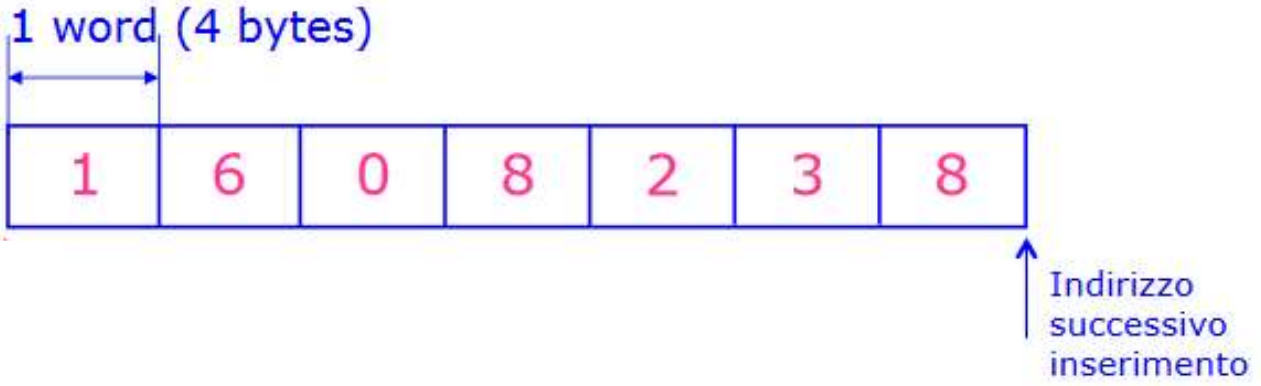


È equivalente a:

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

Esempi

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



Si tratta della allocazione statica di un array di interi.
Ma come accedere agli elementi dell'array?

Ricordandosi l'esatto indirizzo di memoria di ogni elemento?

Utilizzo di Identificatori

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

Identificatore

- **E' un nome** associato ad una particolare posizione del programma Assembler come l'indirizzo di una istruzione o di un dato
  - Es. «main» oppure «forloop» oppure «exitcode» …
  - Es. «A» associato ad una variabile 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

Etichetta o Label

- Una etichetta **introduce** un identificatore e lo associa al punto del programma in cui si trova.
- Un'etichetta consiste in un identificatore seguito dal simbolo «:»
  - Esempio: «main:», «forloop:», «exitcode:»,..
  - Esempio: «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
- 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.

Riferimenti

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

Forloop:
……(istruzioni)…….
……(istruzioni)…….
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
- Es.

Array .word 10,2,33,42,51 #istanzia un array di 5 interi inizializzati

Il secondo elemento dell'array si può referenziare con «Array+4»

Esercitiamoci con MARS..

```
.data
a: .word 8
b: .word 9
c: .word 10,11,12,13
```

Dopo il comando «Assemble»

Layout di memoria?

Esercitiamoci con MARS..

```
.data
a: .word 8
b: .word 9
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

- Endianess nascosta dal debugger (visualizzatore dell'immagine della memoria)
- Memorizzazione di un intero ogni 4 byte, in ordine di dichiarazione

Scopriamo l'Endianess

```
.data
a: .word 8
b: .word 9
c: .word 10,11,12,13
```

Come potremmo scoprire l'endianess?

Potrei vedere se all'indirizzo iniziale della memoria statica (0x10010000) trovo memorizzato «0x00» oppure «0x08»!

```
.text
…….# metti in $s0 l'indirizzo 0x10010000
lb $t0, 0($s0) # NUOVA ISTRUZIONE: LOAD BYTE!
```

Cosa leggete in 0x10010000?

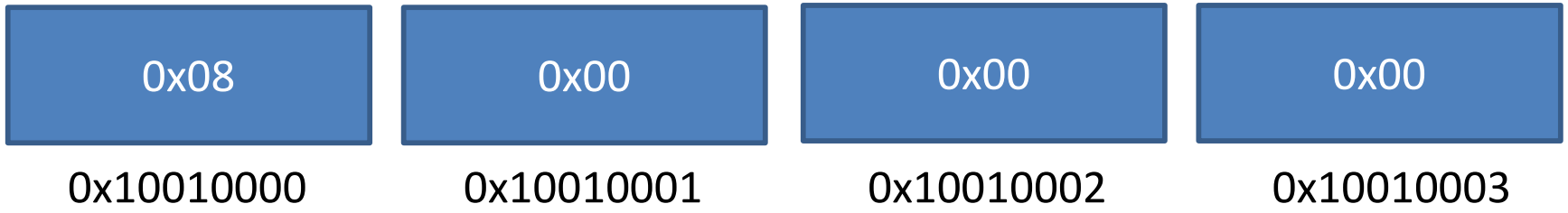
Qual è l'endianess della macchina?

Risposta

```
.text
addi $s0, $zero, 0x10010000
lb $t0, 0($s0)
```

In 0x10010000 leggo «8»

Dunque:



Si tratta di una architettura LITTLE ENDIAN

Cosa leggete all'indirizzo 0x10010004?

Esercitiamoci con MARS..

```
.data
a: .half 8
b: .half 9
c: .half 10,11,12,13
```

Dopo il comando «Assemble»

Layout di memoria?

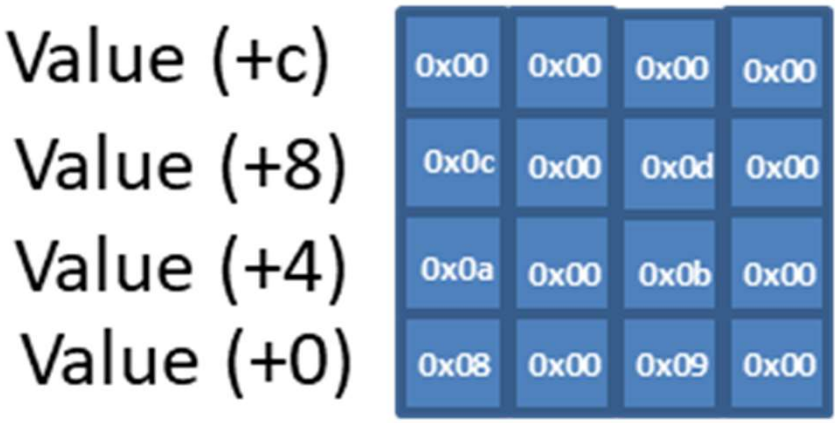
Esercitiamoci con MARS..

```
.data
a: .half 8
b: .half 9
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*
- Dunque, 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..

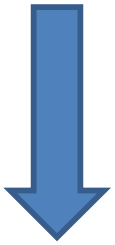
```
.data
a: .byte 8
b: .byte 9
c: .byte 10,11,12,13
```

Dopo il comando «Assemble»

Layout di memoria?

Esercitiamoci con MARS..

```
.data
a: .byte 8
b: .byte 9
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
- Dunque nella prima parola ad indirizzi crescenti troviamo 0x8, 0x9, 0x0a, 0x0b, che il debugger interpreta come «0x0b0a0908»

Esercitiamoci con MARS..

```
.data
a: .byte 8
Stringa: .asciiz “AB”
b: .byte 9
c: .byte 10,11,12,13
```

Dopo il comando «Assemble»

Layout di memoria?

Esercitiamoci con MARS..

```
.data
a: .byte 8
Stringa: .asciiz “AB”
b: .byte 9
c: .byte 10,11,12,13
```

Value (+c)	0x00	0x00	0x00	0x00
Value (+8)	0x0d	0x00	0x00	0x00
Value (+4)	0x09	0x0a	0x0b	0x0c
Value (+0)	0x08	0x41	0x42	0x00

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*
- I caratteri vengono memorizzati secondo la codifica ASCII
- Dunque nella prima parola ad indirizzi crescenti troviamo 0x08, 0x41 (‘A’), 0x42 (‘B’), 0x00 (terminatore), che il debugger interpreta come «0x00424108»

Esercitiamoci con MARS..

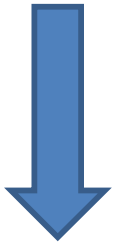
```
.data
a: .byte 8
Stringa: .ascii “AB”
b: .byte 9
c: .byte 10,11,12,13
```

Dopo il comando «Assemble»

Layout di memoria?

Esercitiamoci con MARS..

```
.data
a: .byte 8
Stringa: .ascii “AB”
b: .byte 9
c: .byte 10,11,12,13
```



Dopo il comando «Assemble»

- Con «.asciiz»:
  - ✓ Dunque nella prima parola ad indirizzi crescenti troviamo 0x08, 0x41, 0x42, 0x00, che il debugger interpreta come «0x00424108»
- Senza «.asciiz»:
  - ✓ Scompaiono i due «00» dalla posizione più significativa, ed abbiamo subito 0x09

Data Segment

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

Laboratorio di Architettura degli Elaboratori

- Tutorial sulle funzionalità di base del simulatore MARS
  - ✓ syscalls

Prof. **Davide Bertozzi** davide.bertozzi@unife.it

Le Syscall

- ***Le Syscall:*** Sono letteralmente ***chiamate al sistema operativo***, che servono principalmente per operazioni di input e output.
- MARS emula queste chiamate di sistema.
- Esistono diversi tipi di syscall, identificate da un numero, e funzionano in questo modo:
  1- Carichiamo in un apposito registro il codice della Syscall;
  2- Carichiamo gli eventuali argomenti in appositi registri;
  3- Chiamiamo la syscall;
  4- Recuperiamo gli eventuali valori di ritorno dagli appositi registri di risultato.

Trovate l'elenco completo di tutte le syscall al sito:
https://courses.missouristate.edu/KenVollmar/mars/Help/SyscallHelp.html

Syscall «termina programma»

Service	Code in $v0	Arguments	Result
exit (terminate execution)	10		

- Emula la chiamata al sistema operativo che causa la terminazione di un programma

Provate a farlo!

Implementazione

Service	Code in $v0	Arguments	Result
exit (terminate execution)	10		

- Emula la chiamata al sistema operativo che causa la terminazione di un programma

```
.text
addi $v0, $zero, 10
syscall
```

Syscall «lettura di un intero»

- Utilizziamo la Syscall «stampa intero»

Service	Code in $v0	Arguments	Result
print integer	1	$a0 = integer to print	

- Il codice della Syscall va nel registro $v0, mentre il numero intero da stampare va in $a0.

Provate a farlo!

Implementazione v1

- Utilizziamo la Syscall «stampa intero»

Service	Code in $v0	Arguments	Result
print integer	1	$a0 = integer to print	

- Il codice della Syscall va nel registro $v0, mentre il numero intero da stampare va in $a0.

```
.text
addi $a0, $zero, 42    # Carichiamo il valore da stampare in $a0
addi $v0, $zero, 1     # Carichiamo il codice della syscall in $v0
syscall                # Invochiamo la syscall con codice 1
# risultato: stampa 42
```

Più Semplice con le Pseudo-Istruzioni

- Utilizziamo la Syscall «stampa intero»

Service	Code in $v0	Arguments	Result
print integer	1	$a0 = integer to print	

- Il codice della Syscall va nel registro $v0, mentre il numero intero da stampare va in $a0.

```
.text
li $a0, 42 # Carichiamo il valore da stampare in $a0
li $v0, 1 # Carichiamo il codice della syscall in $v0
syscall                    # Invochiamo la syscall con codice 1
# risultato: stampa 42
```

- Grazie a «load immediate (li)», **posso caricare una costante in un registro**

L’Assemblatore all’Opera

```
.text
li $a0, 42 # Carichiamo il valore da stampare in $a0
li $v0, 1 # Carichiamo il codice della syscall in $v0
syscall                    # Invochiamo la syscall con codice 1
# risultato: stampa 42
```

```
.text
addi $4, $0, 0x0000002A
addi $2, $0, 0x00000001
syscall                    # Invochiamo la syscall con codice 1
# risultato: stampa 42
```

- **L’assemblatore in realtà usa «addiu». Vedremo a suo tempo la differenza con «addi».**

Più Semplice con le Pseudo-Istruzioni

- Variante:

```
.data
A: .word 42 # Allocazione di un intero inizializzato a 42
.text
lw $a0, A # Carichiamo il valore da stampare in $a0
li $v0, 1 # Carichiamo il codice della syscall in $v0
Syscall                    # Invochiamo la syscall con codice 1
# risultato: stampa 42
```

- Grazie alla estensione della semantica di «load word (lw)», **posso caricare direttamente un dato dalla memoria in un registro**

L'Assemblatore all'Opera

```
.text
lw $a0, A # Carichiamo il valore da stampare in $a0
li $v0, 1 # Carichiamo il codice della syscall in $v0
Syscall                    # Invochiamo la syscall con codice 1
# risultato: stampa 42
```

```
.text
lui $1, 0x00001001
lw $1, 0x00000000($1)
addi $2, $0, 0x00000001
syscall
```

Syscall «stampa stringa»

Service	Code in $v0	Arguments	Result
print string	4	$a0 = address of null-terminated string to print	

- Utilizziamo la pseudo-istruzione «load address (la)», che carica l'indirizzo di una locazione di memoria in un registro

Provate a farlo!

Syscall «stampa stringa»

Service	Code in $v0	Arguments	Result
print string	4	$a0 = address of null-terminated string to print	

- Utilizziamo la pseudo-istruzione «load address (la)», che carica l'indirizzo di una locazione di memoria in un registro

```
.data
stringa: .asciiz "Ciao\n" # allocazione di una stringa in memoria
.text
la $a0, stringa # Carichiamo l'indirizzo di «stringa» in $a0
li $v0, 4 # Carichiamo il codice della syscall in $v0
syscall                    # Invochiamo la syscall con codice 4
# risultato: stampa la stringa
```

L’Assemblatore all’Opera

```
.text
la $a0, stringa # Carichiamo l’indirizzo di «stringa» in $a0
li $v0, 4 # Carichiamo il codice della syscall in $v0
syscall                    # Invochiamo la syscall con codice 4
# risultato: stampa la stringa
```

```
.text
lui $1, 0x00001001
ori $4, $1, 0x00000000
addi $2, $0, 0x00000004
syscall
```

Syscall «leggi intero»

Service	Code in $v0	Arguments	Result
read integer	5		$v0 contains integer read

- L'intero letto da std input viene reso disponibile sul registro $v0

Provate a farlo!

Syscall «leggi intero»

Service	Code in $v0	Arguments	Result
read integer	5		$v0 contains integer read

- L'intero letto da std input viene reso disponibile sul registro $v0

```
.text
li $v0, 5 # Carichiamo il codice della syscall in $v0
syscall                    # Invochiamo la syscall con codice 5
                           # Valore letto in $v0
# stampo il valore letto

add $a0, $v0, $zero # travaso del valore letto in $a0
li $v0, 1 # syscall per la scrittura di un intero
syscall # stampa il valore letto
```

Syscall «leggi stringa»

Service	Code in $v0	Arguments	Result
read string	8	$a0 = address of input buffer $a1 = maximum number of characters to read	*See note below table*

- Occorre riservare un buffer in zona dati
- Specificare l'argomento «n» per leggere «n-1» caratteri

Provate a farlo!

Syscall «leggi stringa»

Service	Code in $v0	Arguments	Result
read string	8	$a0 = address of input buffer $a1 = maximum number of characters to read	*See note below table*

- Occorre riservare un buffer in zona dati
- Specificare l'argomento «n» per leggere «n-1» caratteri

```
.data
stringa: .space 8
.text
li $v0, 8 # Carichiamo il codice della syscall in $v0
la $a0, stringa # Indirizzo del buffer
li $a1, 8 # numero di caratteri da leggere (più uno)
Syscall                    # Invochiamo la syscall con codice 8
li $v0, 4 # stampo la stringa letta
Syscall
```

E’ possibile inserire uno spazio tra la stringa letta e la stringa scritta?

Soluzione

```
.data
stringa: .space 5
separatore: .asciiz ”\n”
.text
li $v0, 8 # Carichiamo il codice della syscall in $v0
la $a0, stringa # Indirizzo del buffer
li $a1, 5 # numero di caratteri da leggere (più uno)
syscall                    # Invochiamo la syscall con codice 5
li $v0, 4 # stampo il separatore
move $t0, $a0 # salvo l’indirizzo della stringa acquisita
la $a0, separatore # carico l’indirizzo del separatore
syscall # stampo il separatore
move $a0, $t0 # ripristino l’indirizzo della stringa acquisitia
syscall # stampo la stringa acquisita
```

Laboratorio di Architettura degli Elaboratori

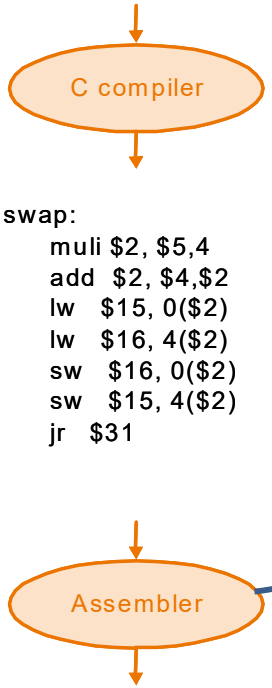
- Tutorial sulle funzionalità di base del simulatore MARS
  - ✓ Pseudo-istruzioni

Prof. **Davide Bertozzi** davide.bertozzi@unife.it

ASSEMBLATORE

High-level language program (in C)

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```



Assembly language program (for MIPS)

```
swap:
     muli $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
```

Binary machine language program (for MIPS)

```
00000000101000010000000000011000
00000000100011110000110000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

Assemblatore:

- Generazione del linguaggio macchina
- Generazione degli indirizzi assoluti di memoria
- Gestione della endianess e degli allineamenti in memoria
- Trasformazione delle pseudo-istruzioni in istruzioni dell'ISA

Pseudo-Istruzioni

- Ci sono istruzioni assembler che non sono di facile implementazione in hardware…..
- …..difatti non fanno parte del set di istruzioni (es., dell'ISA del MIPS), ma sono istruzioni «astratte» che l'Assembler mette a disposizione:
  - Esse vengono poi «tradotte» dall'Assemblatore nelle istruzioni che l'architettura MIPS «sa» eseguire.
  - Esse rendono più agevole la vita al programmatore, perché il loro significato è intuitivo, e corrispondono ad operazioni che il programmatore si trova ad usare frequentemente.
  - Per ogni istruzione, Il «text editor» di MARS suggerisce sia la disponibilità sia la sintassi dei vari comandi. Basta scrivere il nome del comando nell'editor e compare immediatamente in sovraimpressione un mini-tutorial del comando stesso, se disponibile.
- A queste istruzioni diamo il nome di «Pseudo-istruzioni».

Pseudo-Istruzioni

- Sono istruzioni assembler «virtuali», che l'Assemblatore mappa con facilità nelle istruzioni-macchina dell'Assembler reale.
- Sono un primo banale livello di astrazione (come le *label*). Ecco le principali:

SALTO CONDIZIONATO – pseudo-istruzioni

blt	$1, $2, spi	**if** $1 < $2 salta	salta se strettamente minore
bgt	$1, $2, spi	**if** $1 > $2 salta	salta se strettamente maggiore
ble	$1, $2, spi	**if** $1 ≤ $2 salta	salta se minore o uguale
bge	$1, $2, spi	**if** $1 ≥ $2 salta	salta se maggiore o uguale

TRASFERIMENTO TRA PROCESSORE E MEMORIA – pseudo-istruzioni

lw	$1, etichetta	$1 := mem ($gp + spi di etichetta)	carica parola (a 32 bit)
sw	$1, etichetta	mem ($gp + spi di etichetta) := $1	memorizza parola (a 32 bit)

CARICAMENTO DI COSTANTE / INDIRIZZO IN REGISTRO – pseudo-istruzioni

li	$1, cost	$1 := cost (32 bit)	carica costante a 32 bit
la	$1, indir	$1 := indir (32 bit)	carica indirizzo a 32 bit

TRASFERIMENTO TRA REGISTRI – pseudo-istruzione

move	$1, $2	$1 := $2	copia registro