Esercizio:

Scrivi, assembla, e testa (come sopra) un programma in assembly MIPS che

- Inizializza il registro $15 a 10
- Inizializza il registro $19 a 5.
- Implementa la soluzione dell'esercizio dell'ultima lezione, cioè calcola l'espressione:

$$\$20 = ((\$15 + \$19) / (\$15 - \$19))^2$$

21

Esercizio (vedi lucidi precedenti): una soluzione

Usiamo (o riutilizziamo) un registro per ogni sotto-espressione:

```
li $15 10
li $19 5
add $8 $15 $19
sub $9 $15 $19
div $20 $8 $9
mul $20 $20 $20
```

(Testa questo programma su MIPSweb, verifica che in $20 venga scritto il risultato giusto)

22

Esercizio (variante):

Scrivi, assembla, e testa un programma in assembly MIPS che

- Inizializza il registro $15 al valore 10
- Implementa la soluzione dell'esercizio dell'ultima lezione, cioè calcola l'espressione:

$$\$20 = ((\$15 + 5) / (\$15 - 5))^2$$

23

Esercizio 1 (variante): una soluzione

```
li $15 10
addi $8 $15 5
addi $9 $15 -5
div $20 $8 $9
mul $20 $20 $20
```

24

Università degli Studi di Milano «La Statale»
Dipartimento di Informatica

Esempi di programmi in MIPS

Variabili e RAM
1/2

Marco Tarini

26

Direttive Assembler

- Sono «indicazioni» fornite all'Assembler su come trattare il codice Assembly che le segue
- Sintassi: le direttive sono parole precedute da un punto

- In www.mipsweb.di.unimi.it
  tutte le direttive sono visibili sotto
  help → *directives*
  (l'help è l'icona con ? in alto)

30

Alcune direttive Assembler

`.data` : «quello che segue sono dati (numerici, etc) da memorizzare nel segmento **data** della RAM (nell'ordine specificato)»

- In pratica: precede i dati del nostro software.

`.text` : «quello che segue sono istruzioni da memorizzare nel segmento **text** della RAM (nell'ordine specificato)»

- In pratica: precede il codice del nostro software.

(ricorda: software = codice eseguibile + dati)

31

Dati di un programma: primo esempio

- Scriviamo un programma MIPS che:
  - Inizializza questi tre valori word nel settore dati della memoria RAM
    0xABBAABBA
    0x10003456
    «il word che codifica l'intero -5000»
  - Copia il primo di questi tre word nel registro $8

33

Soluzione

```
.data
0xABBAABBA
0x10003456
-5000

.text
li $7 0x10010000
lw $8 ($7)
```

Chiedo all'assembler di effetture la conversione dalla base 10 (comodo!)

So che il segmento di RAM «data» comincia a questo inidrizzo

34

Etichette (label)

- Un'etichetta (label) è un "segnalibro" fisso posto in un indirizzo della memoria statica (data, o, come vedremo, text) di cui tiene traccia l'assembler

```
.data
qui:
5000

.text
la $5 qui
lw $6 ($5)
```

Definizione di una label (nota il due punti finale) Stiamo chiedendo all'assembler di "segnarsi" l'indirizzo di memoria al quale stiamo per mettere il dato successivo (che vale 5000).

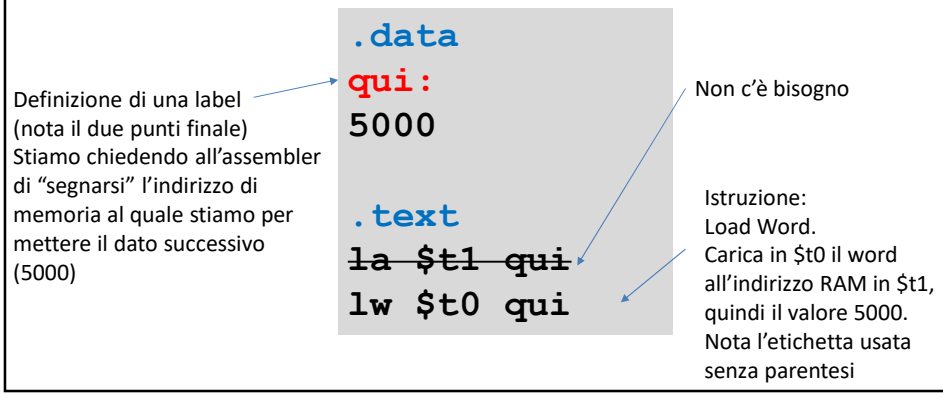
Pseudo istruzione: **load address** Viene tradotta con il comando MIPS che assegna a $5 l'indirizzo corrispondete alla label "qui" (nota: usata senza i due punti)

Istruzione: Load Word. Carica in $6 il word all'indirizzo RAM in $5, quindi il valore 5000

40

Scorciatoria sintattica

- L'assembler ci consente scorciatorie come:
  (le tradurrà in comandi come sopra)



41

Mettiamoci nei panni di un compilatore

- Nei prossimi esempi (questa lezione e altre), vediamo come un compilatore può tradurre in Assembly MIPS costrutti di linguaggi ad alto livello come Go, C (o pseudo codice)
- Esercizio sempre utile: riporta questi programmi sulla http://mipsweb.di.unimi.it
  e osserva come questi programmi di esempio
  - Vengono tradotti in linguaggio macchina (in RAM)
  - Vengono eseguiti, osservandone l'effetto passo passo su memoria RAM e registri

42

Variabili e operazioni fra variabili

in Go:

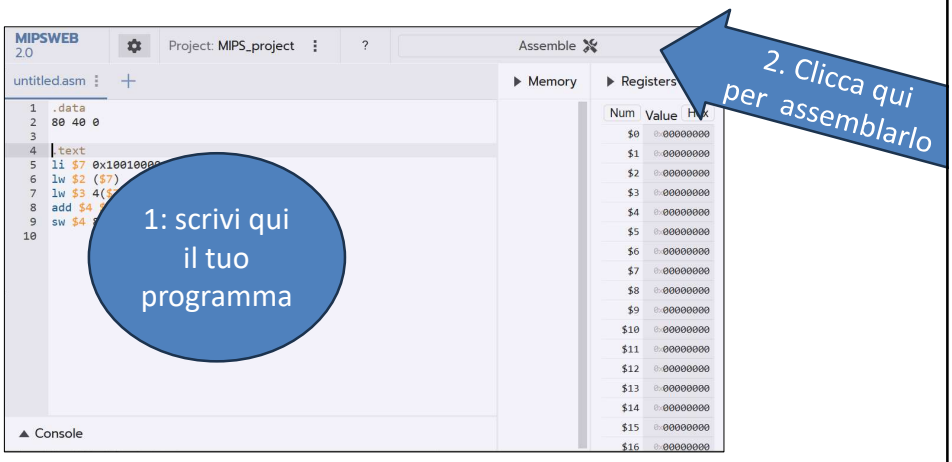
```
x := 80
y := 40
z : int

z = x + y
```

in assembly

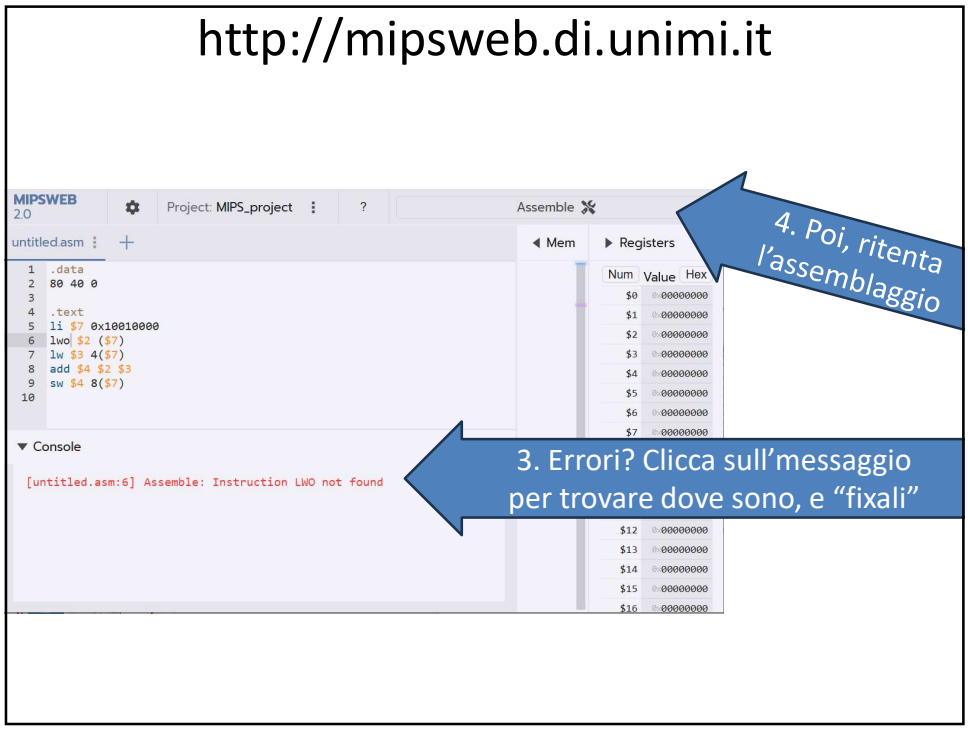
```
.data
80
40
0

.text
li $7 0x10010000
lw $2 ($7)
lw $3 4($7)
add $4 $2 $3
sw $4 8($7)
```

43

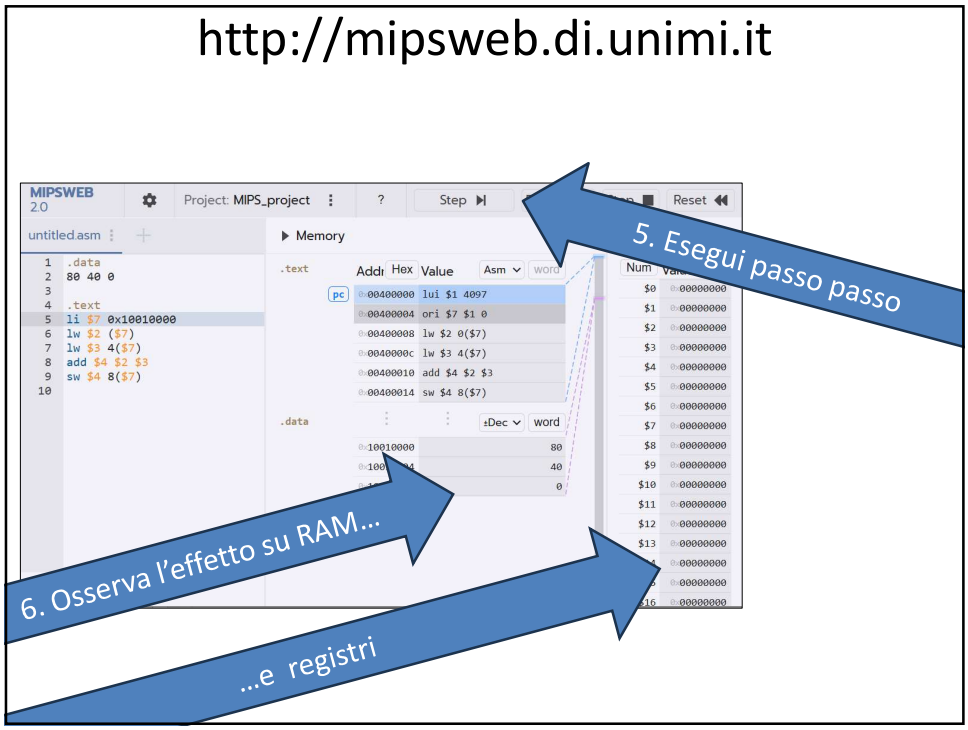
http://mipsweb.di.unimi.it



44


http://mipsweb.di.unimi.it
MIPSWEB 2.0
Project: MIPS_project
Assemble
untitled.asm
Mem
Registers
.data
80 40 0
.text
li $7 0x10010000
lwo $2 ($7)
lw $3 4($7)
add $4 $2 $3
sw $4 8($7)
Num Value Hex
Console
[untitled.asm:6] Assemble: Instruction LWO not found
4. Poi, ritenta l'assemblaggio
3. Errori? Clicca sull'messaggio per trovare dove sono, e "fixali"

45


http://mipsweb.di.unimi.it
MIPSWEB 2.0
Project: MIPS_project
Step
Reset
untitled.asm
Memory
.data
80 40 0
.text
li $7 0x10010000
lw $2 ($7)
lw $3 4($7)
add $4 $2 $3
sw $4 8($7)
Addr Hex Value
00400000 lui $1 4097
00400004 ori $7 $1 0
00400008 lw $2 0($7)
0040000c lw $3 4($7)
00400010 add $4 $2 $3
00400014 sw $4 8($7)
10010000 80
40
0
Num
5. Esegui passo passo
6. Osserva l'effetto su RAM...
...e registri

46

Variabili e operazioni fra variabili: note

- Le tre variabili sono posizionate in memoria RAM, nel segmento dati
  - Ciascuno di loro occupa un word, 4 byte.
- Per sommare x e y, devo prima travasare questi valori nei registri, con due comandi **load word** (lw)
  - E, per far questo, devo prima prendere la locazione della prima (chiamata, ad alto livello, con l'identificatore «x»), questa locazione in RAM (address, indirizzo) è 0x10010000 (la posiziono in $t7)
  - La seconda, «y», è 4 byte (1 word) dopo la «x» (nota l'offset della seconda load word)
- Il risultato della somma va poi memorizzato in RAM con un comando store word (sw)
  - Nuovamente, posso ricostruire correttamente l'indirizzo della variabile «z» osservando che sarà 8 byte dopo quello di «x» (il cui address è ancora memorizzato in $t7)

47

Variabili e operazioni fra variabili: usando le etichette

in Go:

```
x := 80
y := 40
z : int

z = x + y
```

in assembly

```
.data
x: 80
y: 40
z: 0

.text
li $7 x
lw $2 ($7)
lw $3 4($7)
add $4 $2 $3
sw $4 8($7)
```

Oppure anche (meno efficientemente!)

```
lw $2 x
lw $3 y
add $4 $2 $3
sw $4 z
```

48

Vettori (array)

- Un Array non è altro che una sequenza di *n* dati in RAM dello stesso formato (e dimensione) memorizzati in sequenza, (cioè ad indirizzi successivi)
- Per es, un vettore di interi a 32 bit = una successione di words (agli indirizzi n, n+4, n+8…)
- Ad alto livello, come in Go, i vettori si accedono con una sintassi del tipo: `v[3]` (il quarto elemento del vettore, cioè quello preceduto da 3 elementi)

- A basso livello, possiamo calcolare l'indirizzo di ogni elemento del vettore:
  - Se `b` è registro da cui parte un vettore `v`, cioè anche l'indirizzo del suo primo elemento `v[0];`
  - Allora l'elemento i-esimo `v[i]` ha indirizzo `b + 4*i`. (se 4 è la dimesione di un elemento dell'array)

49

Dichiariamo un vettore… (di voti agli esami)

ad alto livello (qui, in Go)

```
var voti := [7]int {23,21,18,16,27,30,24}
```

in assembly:

Dopo la # è un commento (è ignorato dall'assembler)

```
.data
voti:
23 # voti[0]
21 # voti[1]
18 # voti[2]
16 # voti[3]
27 # voti[4]
30 # voti[5]
24 # voti[6]
```

oppure, anche su una riga

```
.data
voti: 23 21 18 16 27 30 24
```

- A basso livello, il nostro vettore è semplicemente una sequenza di dati in RAM
  - contigua (tutti di fila)
  - tutti dello stesso formato
- nota: l'elemento di indice 2 è il 3° dei voti, (è preceduto da 2 voti)

50

…e usiamolo. Esempio

ad alto livello - continuazione

```
voti[6] += 2  // incremento di 2 il voto di indice 6 (il 7°)
```

in assembly - continuazione

```
.text
la $7 voti     # in $7 ho l'inizio del vettore voti
lw $9 24($7)   # in $9 ho messo voti[6]. 6x4 = 24
addi $9 $9 2   # $9 += 2
sw $9 24($7)
```

51

…e usiamolo. Esempio 2

ad alto livello (Go)

```
voti[5] = (voti[2] + voti[3]) /2
// il voto numero 5 (il 6°) assume la media (per difetto)
// dei voti 2 e 3 (il 3° e il 4° voto della lista)
```

in assembly:

```
.text
la $7 voti
lw $10 8($7)
lw $11 12($7)
add $10 $10 $11
srl $10 $10 1 # $10 = $10 / 2
sw $10 20($7)
```

- L'indirizzo del voto di indice *i* è ottenuto con un offset di 4 volte il numero *i* (sia in lettura che in scrittura)
- Nota la divisione per due (per difetto) effettuata con uno shift a destra (srl = Shift Right Logical) di un bit.
- Ottimizzazione rispetto a «div $10 $10 2» (che è una pseudo istruzione tradotta in due istruzioni)

52

…e usiamolo. Esempio 2

ad alto livello (Go)

```
i := 3
… // qui altro codice… (potrebbe cambiare il valore di i)
voto[i]=30 // il voto dell'esame i-esimo diventa 30
```

in assembly:

```
.text
la $7 voti

la $8 i
lw $10 ($8)        oppure  lw $10 i

sll $10 $10 2 # oppure mul $10 $10 4  (meno efficiente)
add $4 $7 $10 # ora $4 è indirizzo di voti[i], base + 4*i

li $3 30
sw $3 ($4)
```

53

…e usiamolo. Esempio 2

Note:

- In questo caso l'assembler non può calcolare l'offset dell'elemento «staticamente» e usarlo nella store (come immediate) perché "non sa" che valore assume la variabile i in un dato momento.
  - Terminologia: «staticamente» = prima dell'esecuzione.
  - Qui: durante l'assemblaggio da assembly a linguaggio macchina
- Il computo di (base del vettore + 4 i) va quindi effettuato dinamicamente, qui tramite operazioni **shift a sinistra (sll)** e **add**
  - Terminologia: «dinamicamente» = *a tempo di esecuzione*
  - Nota l'uso di shift a sinistra di due bit per moltiplicare per 4
  - La pseudoistruzione **mul** avrebbe invece comportato 2 istruzioni
- Per memorizzare 30 alla RAM all'indirizzo risultante, è necessario prima memorizzare temporaneamnete il valore 30 in un registro
  - «`sw 30 ($4)`» non sarebbe valido (come potrebbe un'istruzione MIPS, di 32 bit, contenere anche… i 32 bit da memorizzare in RAM?)

54

Specificare i dati in altri formati

- Posso cambiare il formato in cui specifico i dati con apposite «direttive»
- Ognuna di queste direttive cambia il modo in cui l'assembler interpreta i dati che scrivo di seguito
  - valgono cioè fino a contrordine
- Nota: questo è indipendente dalle etichette
  - ricordare la sitassi:
    l'etichetta termina con due-punti
    la direttiva inizia con punto

55

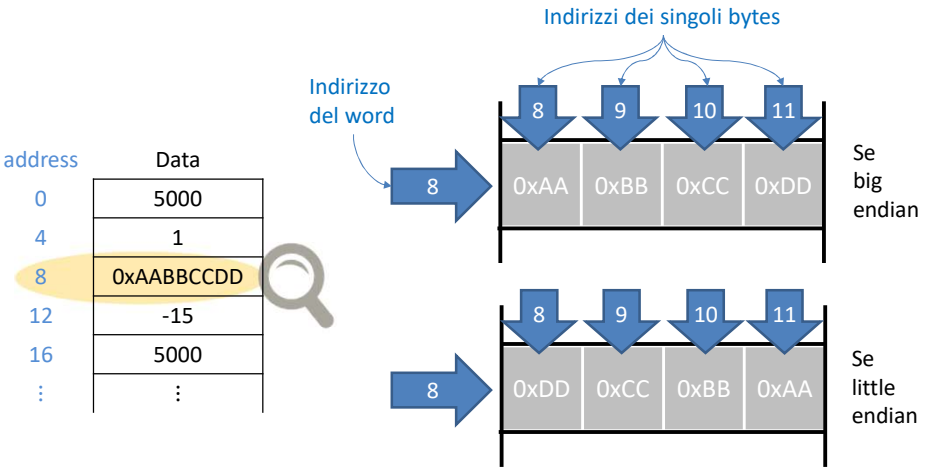
Direttive per la specifica dati

`.word`	Ogni numero che segue = un word (4 bytes) (è il default, se non si speficica alcuna direttiva)
`.half`	Ogni numero che segue = un half-word (2 bytes)
`.byte`	Ogni numero che segue = un byte
`.ascii`	Ogni *stringa* che segue, messa fra "virgolette" = 1 byte per lettera – il codice ASCII di quella lettera
`.asciiz`	Come sopra, più un ultimo ulteriore byte 0x00 per terminare la stringa
`.space n`	Lascia qui *n* byte di spazio (salta *n* byte prima di inserire il prossimo dato)
`.align n`	Lascia qui un certo numero di byte per rendere la prossima locazione di memoria divisible per 2^n

56

Endianness

- L'indirizzo in RAM di un word è l'indirizzo **del primo** dei 4 byte che compongono quel word

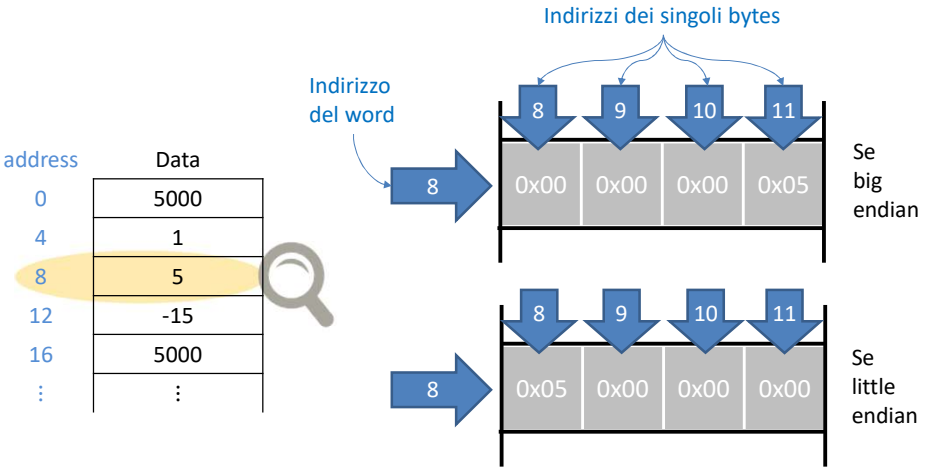


- Cosa si intende con **«del primo»?** Dipende dalla «**endianness**» dell'architettura

57

Endianness

- L'indirizzo in RAM di un word è l'indirizzo **del primo** dei 4 byte che compongono quel word



- Cosa si intende con **«del primo»?** Dipende dalla «**endianness**» dell'architettura

58

Direttiva .byte

- La direttiva byte specifica che i dati che seguono occupano un byte ciascuno

```
.byte 0xAA 0xBB 0xCC 0xDD
```

equivalente a:

```
.word 0xAABBCCDD
```

(se la macchina è big endian, come il MIPS)

La famiglia di architetture x86 è Little Endian (Intel Core i7, AMD Phenom II, FX, …).

59

Direttiva .half

- La direttiva half (half-word) specifica che i dati che seguono occupano due byte ciascuno

```
.half 0xAABB 0xCCDD
```

equivalente a:

```
.word 0xAABBCCDD
```

(se la macchina è big endian)

60

Dati numerici

- Posso esprimere i valori anche in esadecimale:

```
.word 0xABCD0000
.half 0xAB13  0xAC01
.byte 0xAF
```

- O in decimale
  (i negativi interpretati in complemento a 2)

```
.word -1
```
equivalente a
```
.word 0xFFFFFFFF
```
equivalente a
```
.word 4294967295
```

```
.byte -1
```
equivalente a
```
.byte 0xFF
```
equivalente a
```
.byte 255
```

61

Direttiva .ascii

```
.ascii "Derp"
```

Chiedo all'assembly di inserire il codice numerico ASCII di questi caratteri

equivalente a:

```
.byte 'D' 'e' 'r' 'p'
```

equivalente a:

```
.byte 0x44 0x65 0x62 0x70
```

Ogni lettera della striga che segue la direttiva .ascii
viene tradotta in un codice ASCII (7 bits) in 1 byte (estesi con uno 0)

62

Direttiva .asciiz

Il valore 0 (nota: non è il carattere '0'!) denota la fine di una stringa!

```
.asciiz "Derp"
```

equivalente a:

```
.byte 'D' 'e' 'r' 'p' 0
```

equivalente a:

```
.byte 0x44 0x65 0x62 0x70 0x00
```

63

Osservazione su direttive e etichette

- Combinando i meccanismi di direttive e etichette, ottengo una sintassi che somiglia superficialmente a quella di una *dichiarazione delle variabili* in un linguaggio ad alto livello (C, Java, Go…)
  - La direttiva somiglia al TIPO della variabile
  - L'etichetta somiglia all'IDENTIFICATORE della variabile
  - Il valore del dato somiglia all'INIZIALIZZAZIONE

```
peso: .word 40725
```
assembly MIPS

```
peso := int 40725 ;
```
Go

```
int peso = 40725 ;
```
C o Java

64

Osservazione su direttive e etichette

- La somiglianza è solo superficiale
- A differenza delle variabili in un linguaggio ad alto livello, i nostri dati in memoria RAM non sono associati ad alcun tipo
- Sta al programmatore (o al compilatore) MIPS usarli in modo consistente con loro semantica / tipo
- Per es
  - nulla distingue in indirizzo di memoria a cui memorizzo un array di numeri da quello in cui memorizzo in numero solo
  - posso definire 4 byte in successione, e poi usarli come un singolo word, oppure come una stringa di 4 lettere

65

Esercizi

- Modifica gli esempi con il vettore di voti
  in modo che ogni voto sia memorizzato su 16 bit invece che su 32 bit
  - Cosa cambia …
    1 nel segmento dati
    2 nel segmento text
    (sia nelle istruzioni usate, che il calcolo degli indirizzi)
- Scrivi un piccolo programma MIPS che dato un vettore di 4 elementi di tipo half, una variabile «somma» e una variabile «media», riempia queste due variabili con la somma e la media rispettivamente.
- Scrivi un piccolo programma che abbia come dati la stringa «ciaomamma» (terminata dal carattere 0), e come istruzioni un codice per volgere in maiuscolo la prima m. (conta le lettere per calcolare l'offset)

TESTA QUESTI TUOI PROGRAMMI su mipsweb.di.unimi.it !

66