Università degli Studi di Milano
Corso di Laurea in Informatica, A.A. 2018-2019

Sinonimi dei registri e pseudoistruzioni

Turno A
Nicola Basilico
Dipartimento di Informatica
Via Celoria 18 - 20133 Milano (MI)
Ufficio 4008
nicola.basilico@unimi.it
+39 02.503.16289

Turno B
Marco Tarini
Dipartimento di Informatica
Via Celoria 18 - 20133 Milano (MI)
Ufficio 4010
marco.tarini@unimi.it
+39 02.503.16217

Sinomimi dei registri

- Per il IS MIPS, il registro 0 è speciale.
  - vale sempre 0
- tutti gli altri 31 registri sono equivalenti
  - il programmatore può usarli liberamente
- tuttavia, esistono *convenzioni* sul loro uso
  - non imposte dall IS (o dall'HW)
- l'assembler MIPS ci aiuta ribattezzando i registri in modi che riflettono il loro uso convenzionale
  - per es: `$t7` è un sinonimo del registro `$15`
  - per es: `$s3` è un sinonimo del registro `$19`
  - dove: `t` = valure temporaneo, `s` = valore da salvare
  - nota: `$zero` è un sinonimo del registro `$0`
- in assembly, possiamo riferirci ai registri usando i loro sinonimo
  - l'assembler traduce da sinonimi a registro
- vedremo più avanti nel corso il significato di vari altri sinonimi

Sinonimi dei registri MIPS

Registro:	$0	$1	$2	$3	$4	$5	$6	$7
Sinonimo:	$r0	$at	$v0	$v1	$a0	$a1	$a2	$a3
Registro:	$8	$9	$10	$11	$12	$13	$14	$15
Sinonimo:	$t0	$t1	$t2	$t3	$t4	$t5	$t6	$t7
Registro:	$16	$17	$18	$19	$20	$21	$22	$23
Sinonimo:	$s0	$s1	$s2	$s3	$s4	$s5	$s6	$s7
Registro:	$24	$25	$26	$27	$28	$29	$30	$31
Sinonimo:	$t8	$t9	$k0	$k1	$gp	$sp	$s8	$ra

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**.

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.

Pseudo-istruzioni

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

- mul $s0, $t1, $t2 # pseudo istruzione
  - mult $t1, $t2
  - mflo $s0
- div $s0, $t1, $t2 # pseudo istruzione
  - div $t1, $t2
  - mflo $s0

Pseudo-istruzioni: esempi 1/3

MIPS (molto RISC) non prevede istruzioni per copiare valori o registri in un registro, perché si possono ottenere come somme con (il registro) zero

Non sarebbe molto intuitivo usare una somma, quindi l'assembly introduce alcune pseudoistruzioni:

pseudo-istruzione MOVE:

```
move $t0 $t1 # copia il valore $t1 in $t0
```

diventa:
```
add $t0 $zero $t1
```
oppure:
```
addi $t0 $t1 0
```

pseudo-istruzione LOAD IMMEDIATE:

```
li $s2 350 # copia il valore 350 in $s2
```

diventa:
```
addi $s2, $zero, 350
```

Pseudo-istruzioni: esempi 2/3

Il risultato di una moltiplicazione fra word a 32 bit è (potenzialmente) un numero a 64 bit.

In MIPS il risultato della moltiplicazione appare quindi in due registri (HI e LO) facendo eccezione alla regola «il primo registro è il risultato dell'operazione»

Quando il risultato del prodotto è minore del massimo valore esprimibile, non ci sarebbe bisogno di questa complicazione, e basterebbe usare solo LO.

L'assembly mips ci mette a disposizione la…

pseudo-istruzione MUL: (per prodotti che non fanno troppi miliardi)

```
mul $t1 $t3 $t4   # t1 = t3 x t4
```

diventa:

```
mult $t3 $t4
mflo $t1
```

Pseudo-istruzioni: esempi 2/3

Il risultato di una div (divisione intera) consiste in due numeri, quoziente e resto, che appaiono negli stessi due registri speciali (HI e LO).

Quando ci serve solo il quoziente o solo il resto, possiamo usare le pseudoistruzioni…

pseudo-istruzione DIV: (divisione intera)

```
div $s1 $t3 $t4   # s1 = t3 / t4
```

diventa:

```
div $t3 $t4
mflo $s1
```

pseudo-istruzione REM: (reminder, «resto», detto anche modulo)

```
rem  $s2 $t3 t4   # s1 = t3 % t4
```

diventa:

```
div $t3 $t4
mfhi $s1
```

Esercizio 1.3

- Si implementi il codice Assembly che effettui la moltiplicazione e la divisione tra i numeri 100 e 45, utilizzando le istruzioni dell'ISA e le pseudoistruzioni.
- I risultati vanno messi rispettivametne in $s1 e $s2
- Osservare la traduzione di pseudoistruzioni in istruzioni!
  e di registri in sinonimi

Esercizio 1.3

```
main:

li $t1  100
li $t2  45
mul $s1 $t1 $t2
div $s2 $t1 $t2
```

Università degli Studi di Milano
Corso di Laurea in Informatica, A.A. 2018-2019

Indirizzamento, lettura e scrittura della memoria

Turno A
Nicola Basilico
Dipartimento di Informatica
Via Celoria 18 - 20133 Milano (MI)
Ufficio 4008
nicola.basilico@unimi.it
+39 02.503.16289

Turno B
Marco Tarini
Dipartimento di Informatica
Via Celoria 18 - 20133 Milano (MI)
Ufficio 4010
marco.tarini@unimi.it
+39 02.503.16217

Organizzazione della memoria

- Cosa contiene la memoria?
  - **Programmi:** le istruzioni da eseguire
  - **Dati:** le strutture su cui operare

Eng: *word*
l'unità base dei trasferimenti tra memoria e registri
(es: i comandi MIPS *load word* e *store word* operano su un word)

- Come è organizzata (a basso livello)?
  - Un array uni-dimensionale di elementi dette **parole** (della RAM)
  - Ogni parola è univocamente associata ad un **indirizzo** (come l'indice di un array)

address	Data
⋮	⋮
3	429
2	131
1	1
0	5000

Parola associata all'indirizzo 2, Mem[2]=131

Parola (word) in MIPS

e quindi anche nel emulatore MARS

dimensione di 1 parola di RAM = dimensione di 1 registro = dimensione di 1 istruzione = dimensione di 1 indirizzo RAM

1 parola = **32** bit (cioè **4** byte)

(nota: in *altri* Instruction Set, nessuna di queste ugualianze vale neccessariamente)

Parola (word) in MIPS

e quindi anche nel emulatore MARS

byte address	Data
⋮	⋮
12	429
8	131
4	1
0	5000

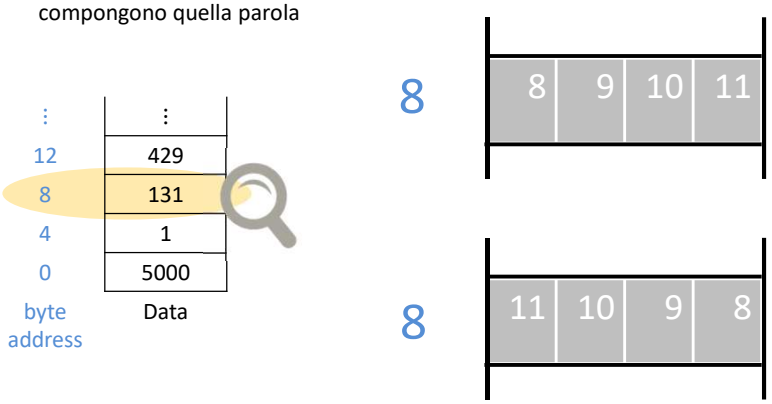
Singolo byte: un elemento di memoria spesso utile

Lo spazio degli indirizzi permette di indirizzare ognuno dei 4 bytes che compongono una parola.

Gli indirizzi di due parole consecutive differiscono di 4

Endianness

- L'indirizzo di una parola di memoria è anche l'indirizzo di uno dei 4 byte che compongono quella parola

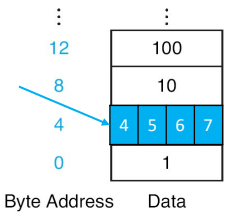


- Ma, tra i 4, **quale?** Dipende dall'ordine dei byte: la «**endianness**» dell'architettura

Endianness

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

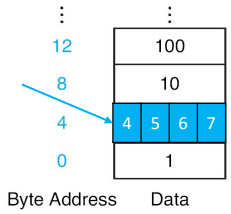
Big endian: il primo byte è quello **più** significativo (quello più a **sinistra**, **big** end)

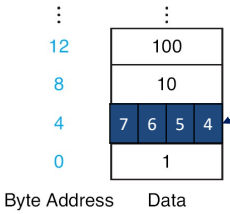


Endianness

- 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)



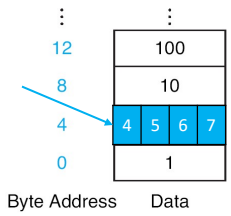


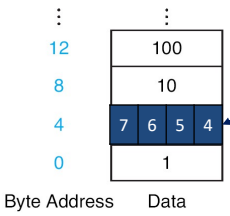
Little endian: il primo byte è quello **meno** significativo (quello più a **destra**, **little** end)

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)



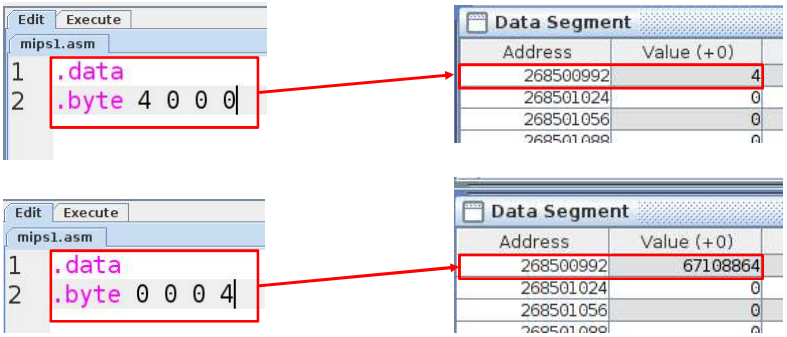


Little endian: il primo byte è quello **meno** significativo (quello più a **destra**, **little** end)

- MIPS è una architettura Big Endian, ma …
- … il nostro emulatore MARS (e anche SPIM) eredita la endianness della macchina su cui è eseguito

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:



MIPS: Letture e scritture RAM

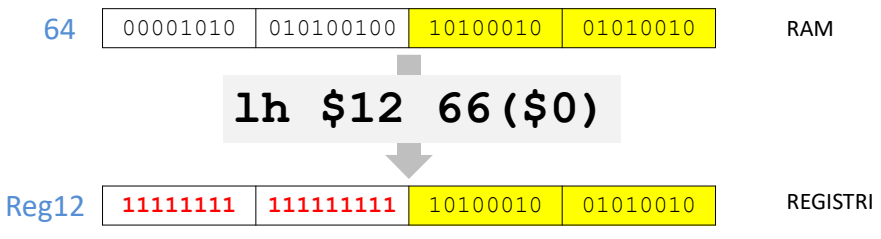
Istruzione	*Commento*	
sw $3, 500($4)	Store word	da registro a mem (store)
sh $3, 502($2)	Store half	
sb $2, 41($3)	Store byte	
lw $1, 30($2)	Load word	da mem a registro (load)
lh $1, 40($3)	Load halfword	
lhu $1, 40($3)	Load halfword unsigned	
lb $1, 40($3)	Load byte	
lbu $1, 40($3)	Load byte unsigned	
lui $1, 40($3)	Load Upper Immediate (16 bits shifted left by 16)	

MIPS: Letture e scritture RAM

- Store: da registri --> a RAM
- Load: da RAM --> a registri
- Primo operando:
  il registro da / a quale operare
- Secondo operando, scritto fra parentesi ($14):
  il registro che *contiene l'indirizzo RAM*
  - ricordare: in MIPS 1 registro = 1 indrizzo memoria
- Numero prima della parentesi:
  «offset» Viene aggiunto all'indirizzo
  - è opzionale: se non viene specificato, allora vale 0
  - a cosa serve? (vedi dopo)

Load and Store

- Quando si caricano sottopezzi di parola (**half words** o bytes) MIPS estende in segno



- Questo comportamento si può disabilitare usando le versioni UNSIGNED

Osservazione

- Il linguaggio MIPS non avrebbe potuto optare per istruzioni su memoria che specificano direttamente l'indirizzo RAM sul quale operare. Perché?
  - Un indirizzo di memoria = 32bit
  - Un'istruzione = 32 bit
    (e comprende anche altro! l'OPCODE,
    il registro da usare, etc)
  - Quindi: un'istruzione non può contenere
    un'indirizzo di memoria
- Soluzione del MIPS:
  - le op di memoria specificano solo quale registro contiene l'indirizzo RAM (e non l'indirizzo stesso)
  - e un offset (in 16 bit)
  - nota: altri IS affrontano il prob in modo diverso

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`.

Esercizio 2.1

- Nome del file sorgente: *arraysum.asm*
- Si assuma che:
  - una variabile `h` sia memorizzata all'indirizzo contenuto in `$s1`;
  - Il base address di un array `A` sia contenuto nel registro `$s2`.
- Si scriva, senza eseguirlo, il codice Assembly che effettui:

```
A[12] = h + A[8];
```

```
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
```

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     # $t0 = &h
addi $s2, $zero, 0x10000004     # $t1 = A
```

32 bit

Direttive Assembler

- Sono «meta-istruzioni» che forniscono all'Assembler informazioni operazionali su generare il codice
- In MIPS, le direttive cominciano con un punto
- Su MARS, la lista delle direttive è visibili da *help* → *directives*

Direttive Assembler: `.data` e `.text`

`.data` significa: quel che segue fa parte del segmento dati del programma

`.text` significa: quel che segue fa parte del segmento istruzioni del programma

Reminder: programma = dati + istruzioni.

tenuti in due segmenti separati di RAM

Direttive Assembler: `.space`

```
.space 40
```

significa: «qui lascia 40 bytes di memoria libera».

Cioè: quel che segue verrà scritto dall'assembler dopo aver lasciato uno spazio di 40 bytes (= 10 parole)

Lo spazio lasciato vuoto (tipicamente, nella sezione dati) lo potrà riempire il programma durante la sua esecuzione, per es attraverso delle «store word»

Esempio

- Allochiamo alloca 20 bytes (5 parole) nella parte dati
- Nella parte istruzioni, voglio un programma che scriva il numero 1234 nella 1ma di queste parole

```
.data
.space 20        # lascia 20 bytes vuoti qui

.text
   li $t2 1234   # t2 sarà valore da copiare in mem
   li $t6 ???    # t6 sarà indirizzo RAM dove copiarlo
   sw $t2 ($t6)  # copio t2 nell'indirizzo indicato da t6
```

- Problema: come scopro a quale indirizzo sono stati messi i 20 byte lasciati nella parte data?
- io (il programmatore) non lo so. Ma l'assembler lo sa!

Label

- Soluzione: le Label
- Una Label è un indirizzo di mem di cui l'assembler deve tener traccia.
- ha un nome arbitrario (scelti da noi) che termina con il due-punti

```
.data
foo:             # l'etichetta!
   .space 20

.text
   li $t2 1234   # t2 sarà valore da copiare in mem
   la $t6 foo    # t6 sarà indirizzo RAM dove copiarlo
   sw $t2 ($t6)  # copio t2 nell'indirizzo indicato da s6
```

- qui, l'etichetta "foo" sarà associata (dall'assembler) all'indirizzo RAM dei 20 byte di spazio allocato
- con la pseudoistruzione **`la`** , chiedo la copia di questo indirizzo (nel registro t6)

Offset: esempio

- Allochiamo alloca 20 bytes (5 parole) nella parte dati
- Nella parte istruzioni, voglio un programma che scriva il numero 1234 nella **3za** di queste parole

```
.data
foo: .space 20    # lasciamo 20 bytes bytes vuoti qui

.text
   li $t2 1234    # t2 sarà valore da copiare in mem
   la $t6 foo     # t6 sarà indirizzo RAM dove copiarlo
   sw $t2 8($t6)  # copio t2 nell'indirizzo indicato da t6
                  # aumentato di 8 byte (2 word)
```

Inizializzazione diretta degli indirizzi

- Come fare a caricare un indirizzo in un registro?
  Esempio: caricare in `$s1` l'indirizzo `0x1A0BC204` ricorda: registro = 32 bit
- Tentativo di risposta: `li $s1 0x1A0BC204`

  32 bit
- Questa pesudoistruzione verrebbe tradotta come: `addi $s1 $0 0x1A0BC204`

  32 bit
- ERRORE!
  Il valore «immediato» di `addi` deve essere un intero a max 16 bit!
  (l'intera istruzione occupa solo 32 bit)

Inizializzazione diretta degli indirizzi

- Come fare a caricare un indirizzo in un registro?
  Esempio: caricare in `$s1` l'indirizzo `0x1A0BC204` ricorda: registro = 32 bit
  (16bit 16bit)
- Risposta corretta: ci servono DUE istruzioni separate, ciascuna per caricare 16 bit.

```
lui   $t0 0x1A0B
```

"Load Upper Immediate":
$t0 diventa 0x1A0B0000
(upper 16 bits)

```
ori   $s1 $t0 0xC204
```

"OR Immediate":
$s1 diventa:

```
0x1A0B0000 or
0x0000C024 =
----------
0x1A0BC024
```

(variante: usare ADDI invece di ORI)

Inizializzazione diretta degli indirizzi

- Metodo più comodo: usare la pseudo-istruzione «**load address**»:
- L'assembler la traddurà nel modo più opportuno, in una o due istruzioni (a seconda della lunghezza indirizzo da copiare nel registro)
  Ad esempio:

```
la   $s1   0x0000C204  ---->   addi $s1 $0 0xC204

la   $s1   0xA010C204  ---->   lui  $at     0xA010
                               ori  $s1 $at 0xC204
```

- Nota: il registro $at viene potenzialmente sovrascritto!
  Infatti questo registro è per convenzione «riservato dall'assembler».
  I programmatori non devono usarlo!
  L'assembler lo usa per tradurre alcune pseudoistruzione (come questa).