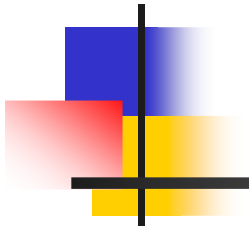
Università degli Studi di Roma "La Sapienza"

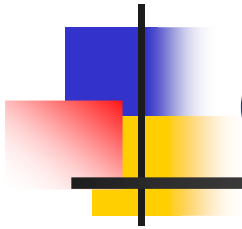
Architettura degli elaboratori II

Istruzioni, programmi e simulatore SPIM



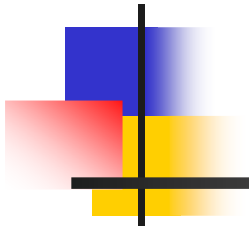
Indice degli argomenti

- **Le istruzioni assembly**
  - Categorie di istruzioni assembly
  - Istruzioni di caricamento e salvataggio
  - Gestione del segno
  - Ordine dei bytes (endianness)
  - Modi di indirizzamento
  - Istruzioni aritmetiche, logiche e di scorrimento
  - Istruzioni di confronto e salto condizionato
  - Istruzioni di salto non condizionato



Categorie istruzioni assembly

- Istruzioni "Load and Store"
  - Spostano dati tra la memoria e i registri generali del processore
- Istruzioni "Load Immediate"
  - Caricano nei registri valori costanti
- Istruzioni "Data Movement"
  - Spostano dati tra i registri del processore
- Istruzioni logico-aritmetiche
  - Effettuano operazioni aritmetiche, logiche o di scorrimento sui registri del processore
- Istruzioni di confronto
  - Effettuano il confronto tra i valori contenuti nei registri
- Istruzioni di salto condizionato
  - Spostano l'esecuzione da un punto ad un altro di un programma in presenza di certe Condizioni
- Istruzioni di salto non condizionato
  - Spostano l'esecuzione da un punto ad un altro di un programma



Architettura LOAD-STORE

L'architettura di MIPS è di tipo Load-and-Store

In pratica, la maggioranza delle istruzioni di MIPS operano tramite i registri interni al processore

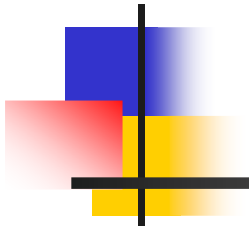
add $t0, $t1, $t2

Somma $t1+$t2 e mette il risultato in $t0

Per questo motivo

LOAD: i dati da elaborare devono essere prelevati dalla memoria

STORE: i risultati devono essere salvati in memoria

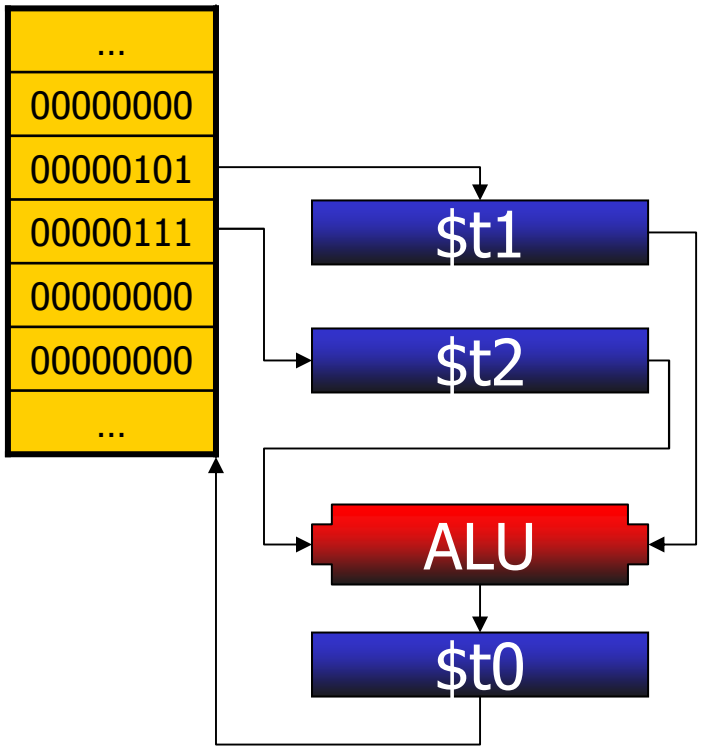


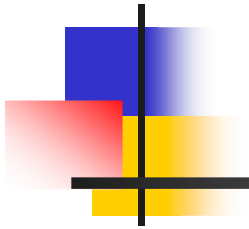
Istruzioni LOAD-STORE

lb rdest, address	*Carica un byte sito all'indirizzo address nel registo rdest*
lh rdest, address	*Carica un halfword sito all'indirizzo address nel registo rdest*
lw rdest, address	*Carica una word sita all'indirizzo address nel registo rdest*
la rdest, address	*Carica un indirizzo address nel registo rdest*
sb rsource, address	*Memorizza un byte all'indirizzo address prelevandolo dal registro rsource*
sh rsource, address	*Memorizza un halfword all'indirizzo address prelevandolo dal registro rsource*
sw rsource, address	*Memorizza una word all'indirizzo address prelevandola dal registro rsource*

Esempio istruzioni L-S

```
.text                   # Inizia il codice

lw $t1, x               # carica x

lw $t2, y               # carica y

add $t0, $t1, $t2       # li somma

sw $t0, z               # copia il risultato

.data                   # Iniziano i dati

x: .word 5

y: .word 7

z: .word 0
```





Endianness

Rappresentazione di una word (32 bit) in 4 byte di memoria

- **Little Endian**: memorizza prima la “little end” (ovvero i bit meno significativi) della word
- **Big Endian**: memorizza prima la “big end” (ovvero i bit più significativi) della word

Nota:

- I processori x86 sono little-endian
- Il processore MIPS può essere usato sia in modo little endian che big endian
- L’endianness di SPIM dipende dal sistema in cui viene eseguito; quindi, in generale è little endian

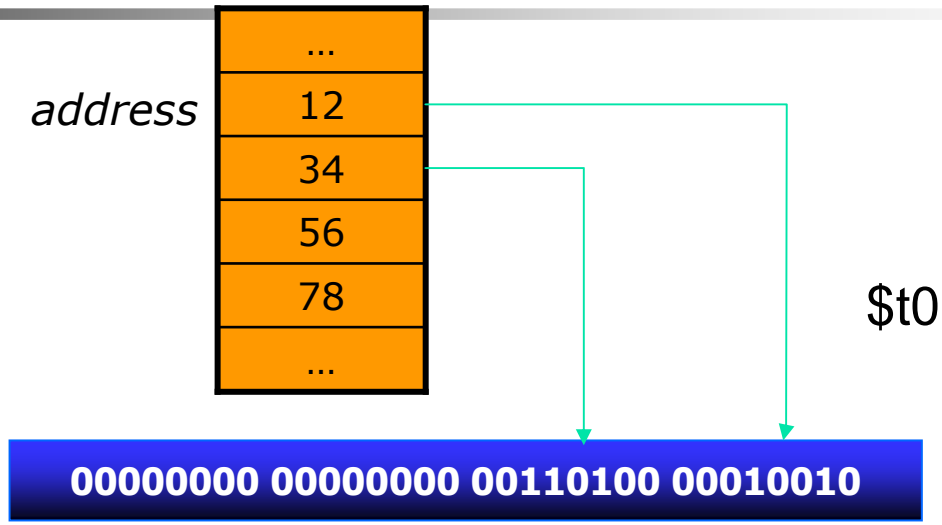
Quando bisogna affrontare i problemi di endianness?

- Quando si "mischiano" operazioni su 8, 16 e 32 bit.
- Quando invece si utilizzano operazioni uniformi, non vi sono problemi di sorta

Endianness: esempio

Little Endian:

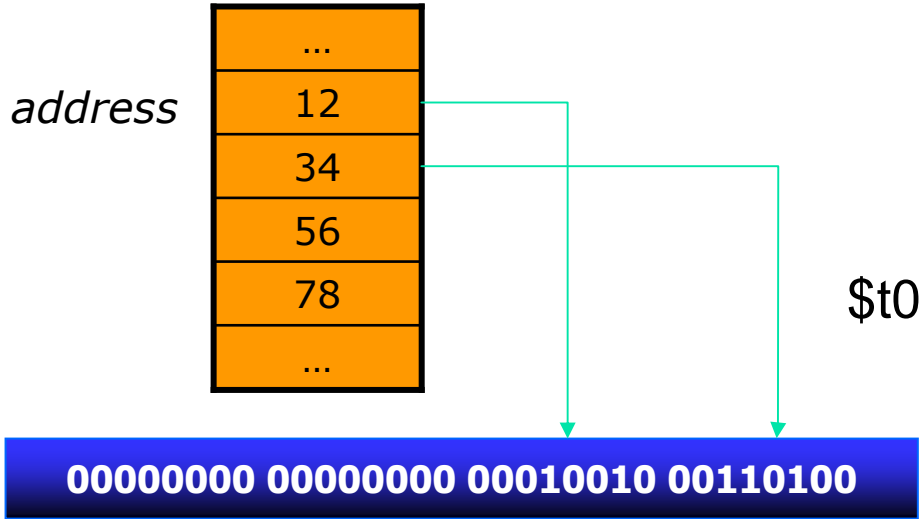
lh $t0,address



$t0=0x00003412

Big Endian:

lh $t0,address

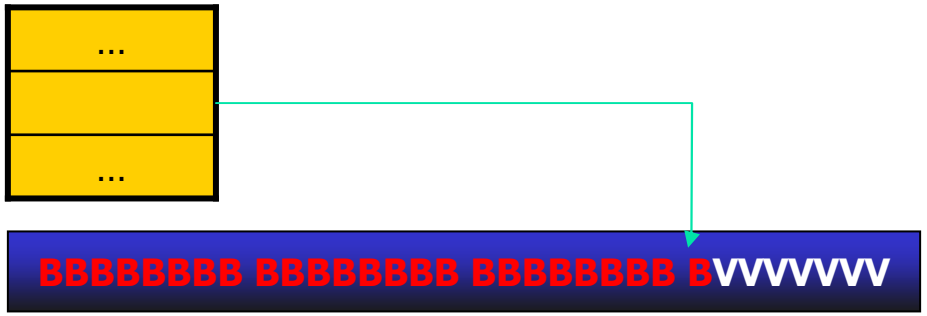


$t0=0x00001234

Load di bytes con e senza segno

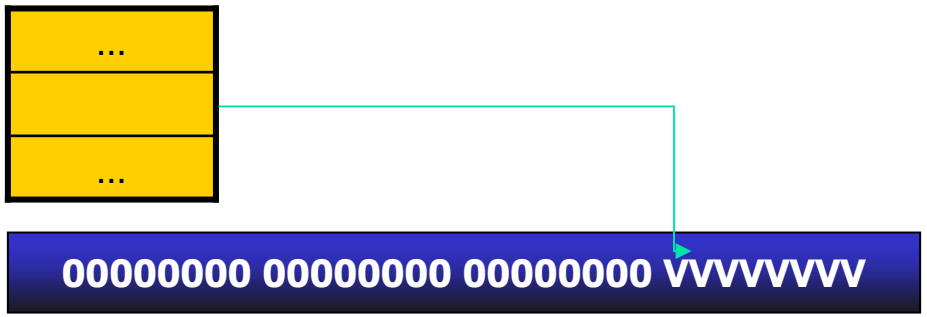
lb $t0, address

- Carica il byte indirizzato da *address* nel byte meno significativo del registro
- Il valore codifica un **numero intero**
- **Il bit di segno viene esteso**



lbu $t0, address

- Carica il byte indirizzato da address nel byte meno significativo del registro
- Il valore codifica un **numero naturale**
- **Gli altri tre byte vengono posti a zero**



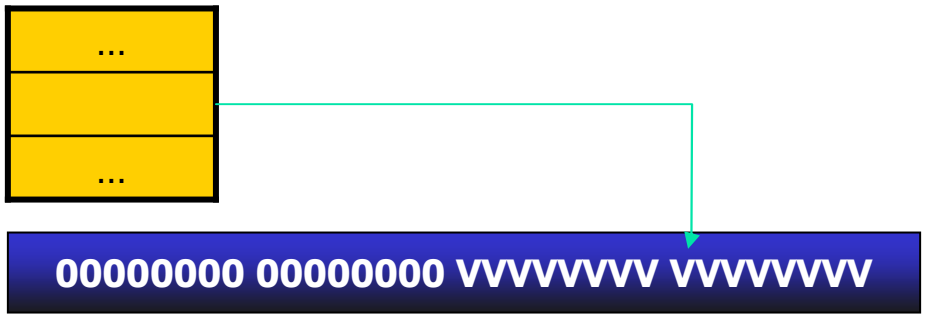
Load di halfwords con segno e segno

lh $t0, address

- Carica i 2 byte indirizzati da *address* nei 2 byte meno significativi del registro
- **Il bit di segno viene esteso**

lhu $t0, address

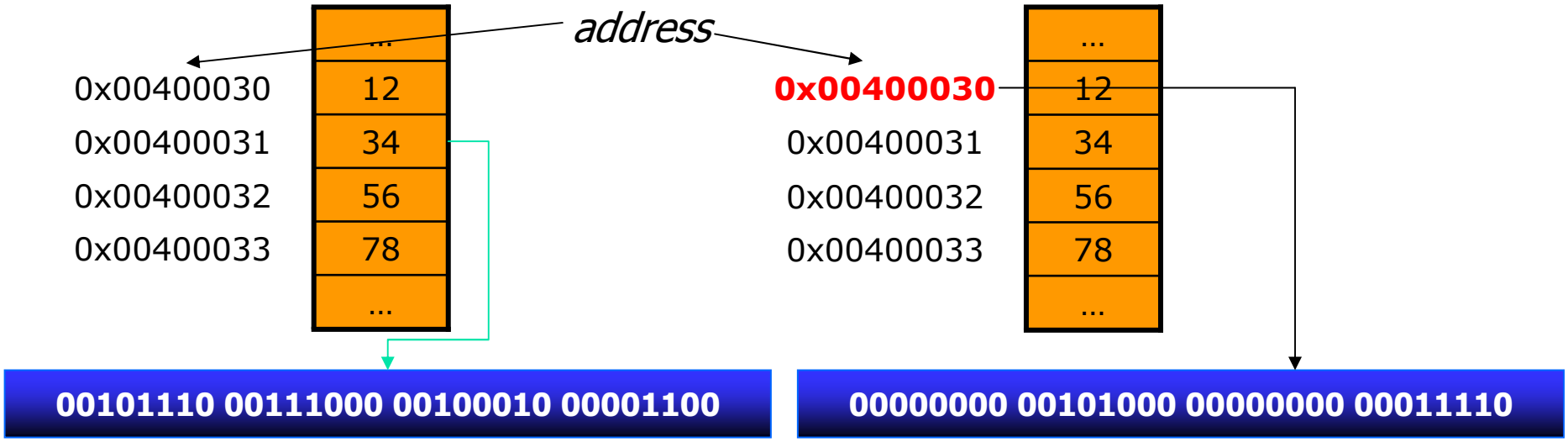
- Carica i 2 byte indirizzati da address nei 2 byte meno significativi del registro
- **Gli altri due byte vengono posti a zero**



Load Adress vs Load Word

- L'istruzione **lw $t0, address** carica ***il contenuto*** della word indirizzata dall'etichetta *address* in memoria
- L'istruzione **la $t0, address** carica ***l'indirizzo*** della word indirizzata dall'etichetta *address* in memoria

(utile quando si vuole caricare in un registro l'indirizzo di una particolare zona di memoria)



$t0=0x78563412

$t0=0x00400030

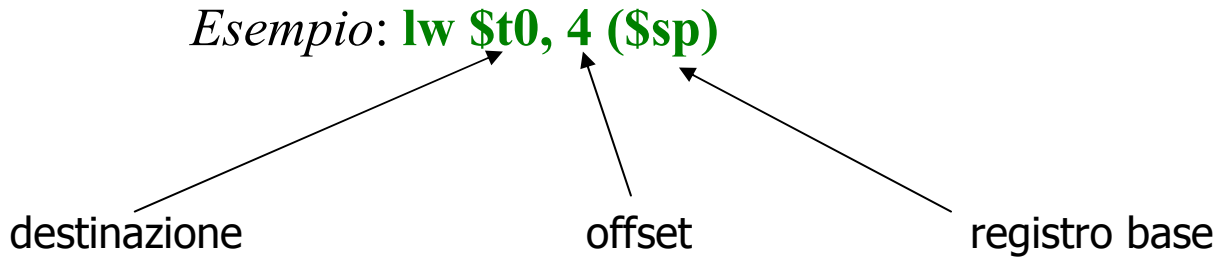
Modi di Indirizzamento

Un modo per esprimere un indirizzo di memoria

NB: alla fine il risultato è sempre un indirizzo di memoria dove si andrà a leggere (load) o scrivere (store) un dato.

Nel linguaggio macchina MIPS, esiste un solo modo di indirizzamento

- Specifica un indirizzo tramite registro base e un'offset.
- L'indirizzo è dato dal contenuto del registro base sommato all'offset



Modi di Indirizzamento

L'assembler fornisce per comodità più modi di indirizzamento

N.B.: sono pseudo-istruzioni !!

Base register:

- Specifica un indirizzo tramite registro base, assumendo un offset 0.
- L'indirizzo è dato dal contenuto del base register

Esempio: **lw $t0, ($a0)**

Expression:

- Specifica un indirizzo assoluto con una espressione fatta solo di somme e sottrazioni di costanti espresse in decimale o esadecimale.
- L'indirizzo è dato dal risultato dell'espressione (costante).

Esempio: **lw $t0, 0x10 01 00 0c + -4**

N.B.: I numeri negativi vanno comunque preceduti anche dal simbolo +

Modi di Indirizzamento

Relocable-symbol

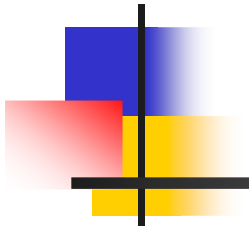
- base = specificato da un'etichetta
- offset:
  - può non esserci

    Esempio: **lw $t0, array**
  - specificato da un registro indice

    Esempio: **lw $t0, array ($t1)**
  - specificato tramite un'espressione

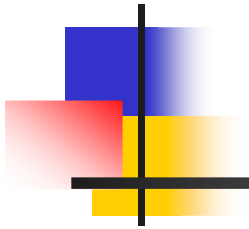
    Esempio: **lw $t0, array + 4**
  - specificato tramite un registro indice e un'espressione

    Esempio: **lw $t0, array + 4($t1)**



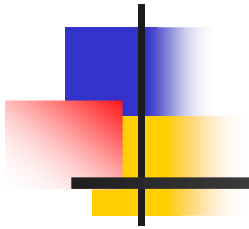
Istruzioni Data Movement and Load Immediate

move rdest, rsource	*Muove il registro rsource nel registro rdest*
mfhi rdest	*Muove il registro hi nel registro rdest*
mflo rdest	*Muove il registro lo nel registro rdest*
mthi rsource	*Muove il registro rsource nel registro hi*
mtlo rsource	*Muove il registro rsource nel registro lo*
li rdest, imm	*Muove il numero imm nel registro rdest*



Istruzioni Aritmetiche

add rd, rs, rt	*rd = rs + rt*	*(with overflow)*
addu rd, rs, rt	*rd = rs + rt*	*(without overflow)*
addi rd, rs, imm	*rd = rs + imm*	*(with overflow)*
addiu rd, rs, imm	*rd = rs + imm*	*(without overflow)*
sub rd, rs, rt	*rd = rs - rt*	*(with overflow)*
subu rd, rs, rt	*rd = rs – rt*	*(without overflow)*
neg rd, rs	*rd = - rs*	*(with overflow)*
negu rd, rs	*rd = - rs*	*(without overflow)*
abs rd, rs	*rd = |rs|*	.



Istruzioni Aritmetiche

mult rs, rt	*hi,lo = rs * rt (signed, overflow impossibile)*
multu rs, rt	*hi,lo = rs * rt (unsigned, overflow impossibile)*
mul rd, rs, rt	*rd = rs * rt (without overflow)*
mulo rd, rs, rt	*rd = rs * rt (with overflow)*
mulou rd, rs, rt	*rd = rs * rt (with overflow, unsigned))*
div rs, rt	*hi,lo = resto e quoz. di rs / rt (signed with overflow)*
divu rs, rt	*hi,lo = resto e quoz. di rs / rt (unsigned, no overflow)*
div rd, rs, rt	*rd = rs / rt (signed with overflow)*
divu rd, rs, rt	*rd = rs / rt (unsigned)*
rem rd, rs, rt	*rd = resto di rs / rt (signed)*
remu rd, rs, rt	*rd = resto di rs / rt (unsigned)*

Istruzioni Logiche

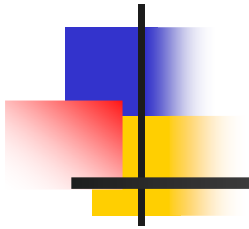
and rd, rs, rt	*rd = rs **AND** rt*
andi rd, rs, imm	*rd = rs **AND** imm*
or rd, rs, rt	*rd = rs **OR** rt*
ori rd, rs, imm	*rd = rs **OR** imm*
xor rd, rs, rt	*rd = rs **XOR** rt*
xori rd, rs, imm	*rd = rs **XOR** imm*
nor rd, rs, rt	*rd = **NOT** (rs **OR** rt)*
not rd, rs	*rd = **NOT** rs*

Istruzioni di Shift

sll rd, rs, rt	*rd = rs shifted left rt mod 32 bits*
srl rd, rs, rt	*rd = rs shifted right rt mod 32 bits*
sra rd, rs, rt	*rd = rs shifted right rt mod 32 bits (signed)*
rol rd, rs, rt	*rd = rs rotated left rt mod 32 bits*
ror rd, rs, rt	*rd = rs rotated right rt mod 32 bits*

Note:

- *rt* può anche essere una costante es. **sll $t1, $t2, 4**
- **Shift** perde bit da una parte, dall'altra entrano degli zero
- **Rotate** inserisce da una parte i bit che escono dall'altra
- Direzione: Left [right] verso quelli più [meno] significativi



Istruzioni Immediate ed Unsigned

Versioni immediate (i)

- Le versioni immediate (i) delle istruzioni precedenti utilizzano un **valore costante** (risultato di un'espressione) al posto di uno degli operandi
- Non sempre è necessario specificare la i; l'assemblatore è in grado di riconoscere le istruzioni immediate. Esempio: **add $t0, $t0, 1**

Versioni unsigned (u)

- Le versioni unsigned delle istruzioni non gestiscono le problematiche relative all'overflow

Istruzioni di confronto

slt rd, rs, rt — *rd* = ***1*** *se rs* **<** *rt*, ***0*** *altrimenti*

slti rd, rs, imm — *rd* = ***1*** *se rs* **<** *imm*, ***0*** *altrimenti*

sgt, sge, sle, seq, sne — *Analoghe per testare* > , ≥ , ≤ , = e ≠

NB: *Di solito queste istruzioni vengono usate solo dall'assembler.*
I programmatori usano direttamente le istruzioni di ***salto condizionato***

Istruzioni di salto condizionato

bgez rs, target	*Salta all'istruzione puntata da target se rs ≥ 0*
bgtz rs, target	*Salta all'istruzione puntata da target se rs > 0*
blez rs, target	*Salta all'istruzione puntata da target se rs ≤ 0*
bltz rs, target	*Salta all'istruzione puntata da target se rs < 0*
beqz rs, target	*Salta all'istruzione puntata da target se rs = 0*
bnez rs, target	*Salta all'istruzione puntata da target se rs ≠ 0*
beq rs, rt, target	*Salta all'istruzione puntata da target se rs = rt*
bne rs, rt, target	*Salta all'istruzione puntata da target se rs ≠ rt*
bge rs, rt, target	*Salta all'istruzione puntata da target se rs ≥ rt*
bgt rs, rt, target	*Salta all'istruzione puntata da target se rs > rt*
ble rs, rt, target	*Salta all'istruzione puntata da target se rs ≤ rt*
blt rs, rt, target	*Salta all'istruzione puntata da target se rs < rt*

Branch: osservazione

Nel linguaggio macchina, l'indirizzo dell'istruzione target in un branch viene espresso come *instruction offset* (una half-word con segno) rispetto all'istruzione corrente

È possibile saltare 2^{15} -1 istruzioni in avanti o 2^{15} istruzioni indietro

2^{15} = circa 32000

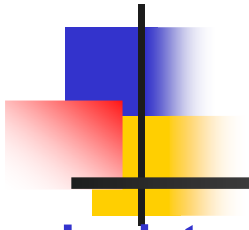
In linguaggio assembly, l'indirizzo dell'istruzione target è dato dall'etichetta associata

Il calcolo viene fatto automaticamente dall'assembler

Istruzioni di salto incondizionato

j target	*Salto incondizionato all'istruzione target*
jal target	*Salto incondizionato all'istruzione target* ***e salvataggio dell'indirizzo della prossima istruzione in $ra***
jr rsource	*Salto incondizionato all'istruzione che ha indirizzo memorizzato nel registro rsource*
jalr rsource, rdest	*Salto incondizionato all'istruzione che ha indirizzo memorizzato nel registro rsource* ***e salvataggio dell'indirizzo della prossima istruzione in rdest***

Sequenza di istruzioni



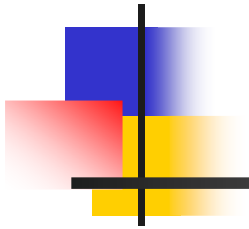
Le istruzioni formano sempre una sequenza che corrisponde all'ordine in cui vengono eseguite

- Il microprocessore non ha una visione di insieme del programma: "vede" solo l'istruzione che deve eseguire e il contenuto dei registri.
- Esegue una istruzione alla volta nell'ordine scritto in memoria

Fanno eccezione le istruzioni di salto che fanno proseguire l'esecuzione all'indirizzo specificato.

- L'istruzione che viene eseguita dopo un salto è quella che si trova all'indirizzo specificato come destinazione del salto.
- Se il salto è condizionato e la condizione è falsa si prosegue normalmente con l'istruzione successiva.

L'intero programma è una sequenza di istruzioni; con i salti si possono controllare l'ordine di esecuzione



Chiamata a funzioni di SO

SPIM fornisce un piccolo insieme di servizi "operating-system-like" attraverso l'istruzione syscall

Per richiedere un servizio:

- Si carica il numero che identifica il servizio nel registro **$v0**
- Eventuali argomenti vengono caricati nei registri **$a0-$a1**
- Eventuali risultati vengono collocati dalla chiamata di sistema nel registro **$v0**

Esempio

```
li $v0,1        # N.B.: '1' identifica print_int
li $a0,5        # 5 è l'argomento da stampare
syscall         # procedi alla stampa
```

Elenco System Call di SPIM

SERVIZI	CODICE	ARGOMENTI	RISULTATO
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = string	
read_int	5		integer in $v0
read_float	6		float in $f0
read_double	7		double in $f0
read_string	8	$a0=buffer, $a1=length	
sbrk	9	$a0=amount	address in $v0
exit	10		

Esempio di uso di System Call

```
.text
li $v0,4          # print_string
la $a0,prompt1
syscall           # stampa il prompt
li $v0,5          # read_int
syscall           # legge 1° intero
move $t1, $v0
li $v0,4          # print_string
la $a0,prompt2
syscall           # stampa il prompt
li $v0,5          # read_int
syscall           # legge 2° intero
move $t2, $v0
add $t0, $t1, $t2
li $v0,4          # print_string
la $a0,msg
syscall           # stampa "La somma è"
li $v0, 1         # print_int
move $a0, $t0
syscall           # stampa la somma
li $v0, 10        # exit
syscall           # termina

.data
prompt1: .asciiz "Primo numero: "
prompt2: .asciiz "Secondo numero: "
msg: .ascii "La somma è "
```

SPIM

- SPIM è un simulatore che esegue programmi per le architetture RISC R2000/R3000 (PowerPC = Mac)
- SPIM può leggere ed assemblare programmi scritti in linguaggio assembly MIPS
- SPIM contiene inoltre un debugger per poter analizzare il funzionamento dei programmi prodotti passo passo

Riferimento per il download di SPIM

http://www.cs.wisc.edu/~larus/spim.html

… e il solito http://twiki.di.uniroma1.it

Interfaccia

PCSpim

File Simulator Window Help

```
 PC       = 00000000   EPC      = 00000000   Cause    = 00000000   BadVAddr= 00000000
 Status   = 3000ff10   HI       = 00000000   LO       = 00000000
                                 General Registers
R0   (r0) = 00000000  R8   (t0) = 00000000  R16  (s0) = 00000000  R24  (t8) = 00000000
R1   (at) = 00000000  R9   (t1) = 00000000  R17  (s1) = 00000000  R25  (t9) = 00000000
R2   (v0) = 00000000  R10  (t2) = 00000000  R18  (s2) = 00000000  R26  (k0) = 00000000
R3   (v1) = 00000000  R11  (t3) = 00000000  R19  (s3) = 00000000  R27  (k1) = 00000000
R4   (a0) = 00000000  R12  (t4) = 00000000  R20  (s4) = 00000000  R28  (gp) = 10008000

        KERNEL



        DATA
[0x10000000]...[0x10040000]     0x00000000

        STACK
[0x7fffeffc]                    0x00000000

        KERNEL DATA
[0x90000000]...[0x90010000]     0x00000000

SPIM Version Version 7.1 of January 2, 2005
Copyright 1990-2004 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
DOS and Windows ports by David A. Carley (dac@cs.wisc.edu).
Copyright 1997 by Morgan Kaufmann Publishers, Inc.
See the file README for a full copyright notice.
```

For Help, press F1 PC=0x00000000 EPC=0x00000000 Cause=0x00000000

Comandi principali

- **Comando LOAD (Unix) e File – Open (Windows)** 

Carica un file scritto in assembly (estensione .s, .asm) e ne assembla il contenuto in memoria

- **Comando RUN (Unix) e Simulator – Go (Windows)** 

Esegue il programma, fino alla terminazione o fino all'incontro di un breakpoint

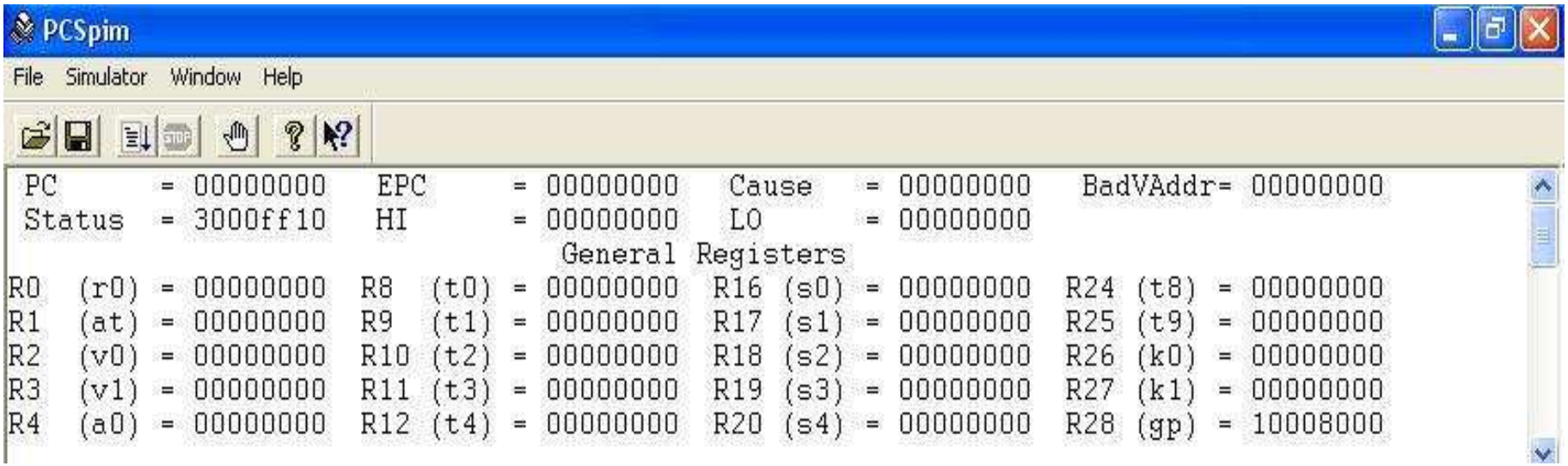
- **Comando STEP (Unix) e Simulator – Step (Windows)** 

Esegue il programma passo-passo, ovvero una istruzione alla volta

Questa modalità permette di studiare nel dettaglio il funzionamento del programma

Interfaccia: sezione registri

- Mostra il valore di tutti i registri della CPU e della FPU MIPS
- Notare che i registri generali vengono identificati sia dal numero progressivo (**R28**) che dall'identificatore mnemonico (**$gp**)
- Il valore dei registri viene aggiornato ogni qualvolta il vostro programma viene interrotto



Interfaccia: sezione codice

- Mostra le istruzioni dei programmi e del codice di sistema che viene caricato automaticamente alla partenza di SPIM

```
[0x00400000]    0x3c011001  lui $1, 4097 [x]              ; 2: lw $t1, x
[0x00400004]    0x8c290000  lw $9, 0($1) [x]
[0x00400008]    0x3c011001  lui $1, 4097 [y]              ; 3: lw $t2, y
[0x0040000c]    0x8c2a0004  lw $10, 4($1) [y]
[0x00400010]    0x012a4020  add $8, $9, $10               ; 4: add $t0,$t1,$t2
[0x00400014]    0x3c011001  lui $1, 4097 [z]              ; 5: sw $t0,z
[0x00400018]    0xac280008  sw $8, 8($1) [z]
```

Descrizione degli elementi

1. Indirizzo esadecimale dell'istruzione
2. Codifica numerica esadecimale dell'istruzione in linguaggio macchina
3. Descrizione mnemonica dell'istruzione in linguaggio macchina
4. Linea effettiva presente nel file assembly che si sta eseguendo

Nota: Ad alcune istruzioni assembly (pseudoistruzioni) corrispondono più istruzioni in linguaggio macchina

Interfaccia: sezione dati

Segmenti data e stack

- Mostra il contenuto del segmento dati e stack della memoria del programma
- I valori contenuti nella memoria vengono aggiornati ogni qualvolta il vostro programma viene interrotto

```
        DATA
[0x10000000]...[0x10010000]     0x00000000
[0x10010000]                    0x00000004  0x00000006  0x00000000  0x00000000
[0x10010010]...[0x10040000]     0x00000000

        STACK
[0x7fffeffc]                    0x00000000
```

Interfaccia: messaggi

- Utilizzata da SPIM per mostrare messaggi, come ad esempio messaggi di errore o di corretto caricamento del file ed esecuzione

```
All Rights Reserved.
DOS and Windows ports by David A. Carley (dac@cs.wisc.edu).
Copyright 1997 by Morgan Kaufmann Publishers, Inc.
See the file README for a full copyright notice.
Memory and registers cleared and the simulator reinitialized.
```

Interfaccia: console

- Shell in cui vengono visualizzati i messaggi stampati dal programma