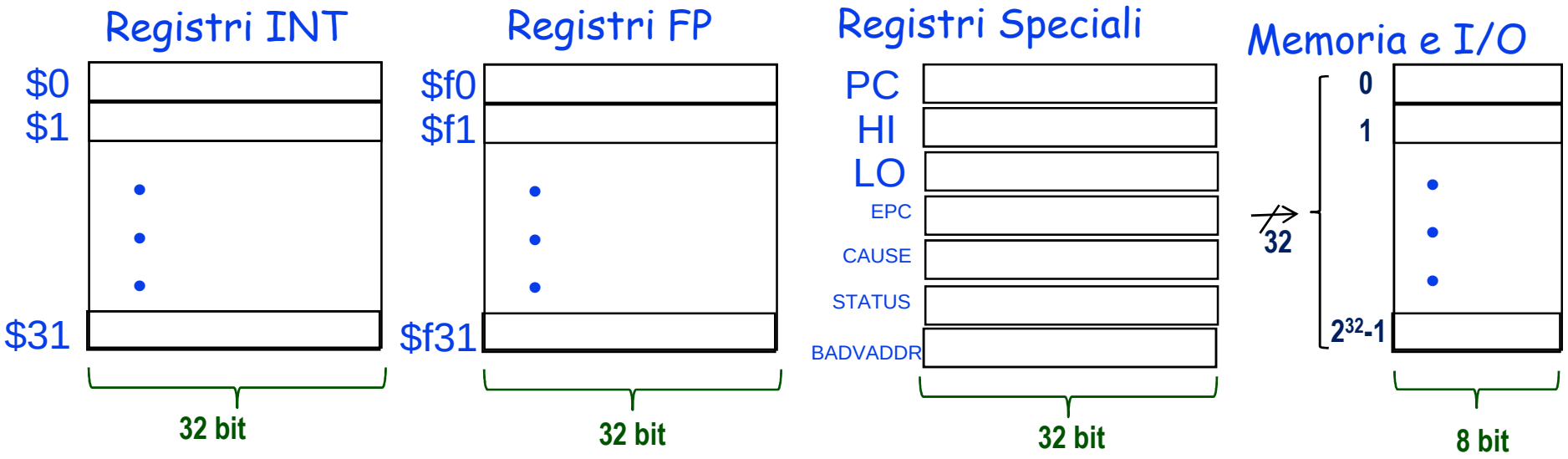
Lezione 2
Assembly MIPS (2)

http://www.dii.unisi.it/~giorgi/didattica/arcal1

Instruction Set Architecture (es. MIPS 3000)



- **Categorie di Istruzioni**
  - **Load/Store**
  - **Calcolo (per Interi e per Frazionari)**
  - **Jump/Branch**
  - **Istruzioni speciali**

· Formato delle Istruzioni:
· lunghezza FISSA a 32 bit → CPU a 32 bit

Ricapitolazione:

- **Istruzione** — **Significato**

Istruzione	Significato
`add $1,$2,$3`	`$1 = $2 + $3`
`sub $1,$2,$3`	`$1 = $2 – $3`
`slt $1,$2,$3`	`$1 = ($2 < $3) ? 1 : 0`
`lw $1,100($2)`	`$1 = Memory[$2+100]`
`sw $1,100($2)`	`Memory[$2+100] = $1`
`bne $4,$5,L`	`Next inst. is at Label if $4!=$5`
`beq $4,$5,L`	`Next inst. is at Label if $4==$5`
`j Label`	`Next inst. is at Label`

- **Formati:**

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

Aritmetica con segno e senza segno

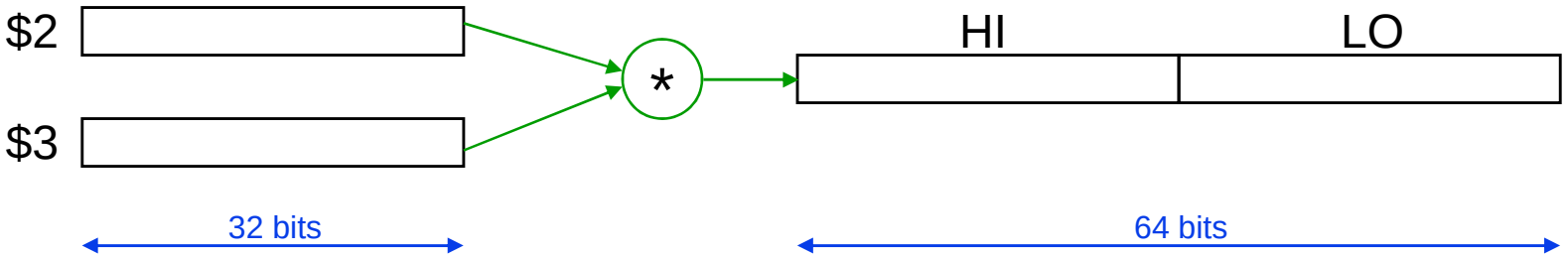
- Le istruzioni aritmetiche fin qui esaminate operano su interi con segno
  - Gli operandi (a 32 bit, nei registri) sono rappresentati in complemento a due: i valori vanno da -2^{31} a $2^{31}-1$
  - Anche i byte-offset di `lw` e `sw` e il numero di istruzioni di cui saltare nella bne/beq sono interi con segno da -2^{15} a $2^{15}-1$

- E' possibile utilizzare anche operandi senza segno
  - In tal caso i valori vanno da 0 a $2^{32}-1$
  - Invece di `add`, `sub`, `slt` si useranno `addu`, `subu`, `sltu`
    - `slt` e `sltu` forniranno un risultato diverso se un operando e' negativo (es. `-4 < 3` → 1 con `slt`, ma 0 con `sltu`)
    - `addu` e `subu` manipolano gli operandi esattamente nello stesso modo di `add` e `sub`, ma NON segnalano alcuna situazione di overflow al processore**

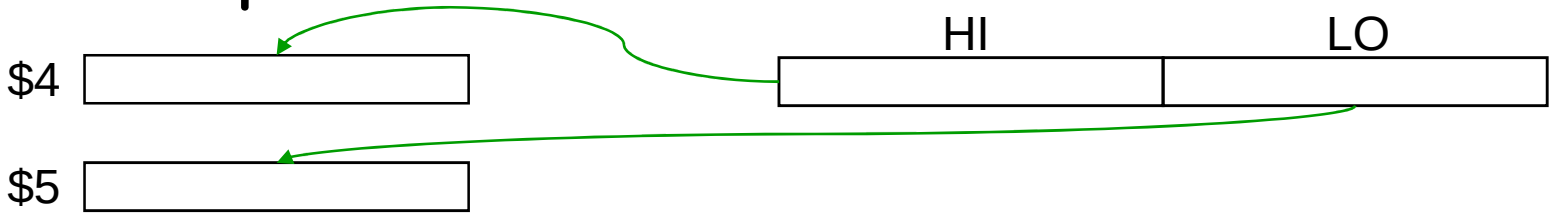
** il processore gestisce tali situazioni con il meccanismo della "eccezione", che sara' esaminato piu' avanti nel corso

Moltiplicazione

`mult $2, $3` **(HI,LO) = $2 * $3**



- **Per recuperare il risultato**



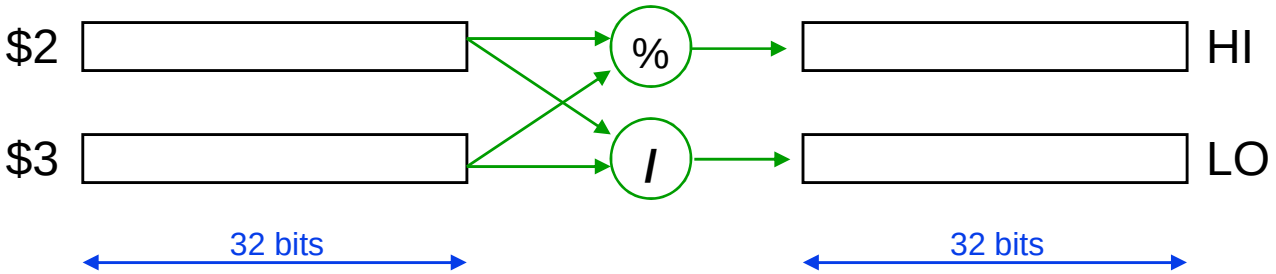

```
mfhi $4
mflo $5
```

Nota1: `mfhi`="move from HI", `mflo`="move from LO"
Nota2: si possono usare HI e LO anche come registri di appoggio: `mthi`="move to HI", `mtlo`="move to LO"

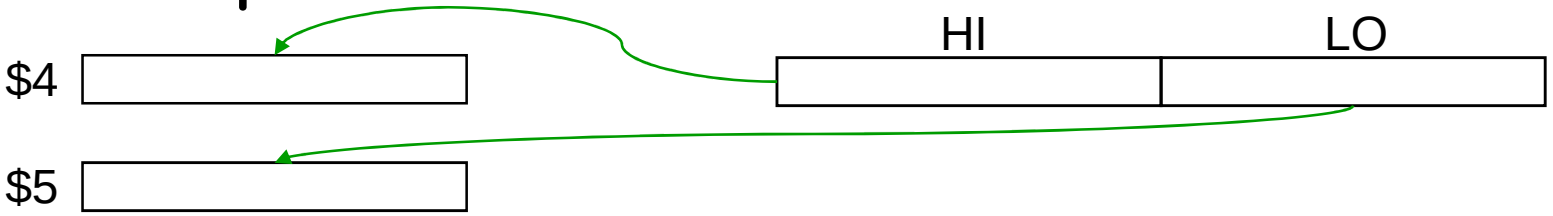
Divisione

```
div $2, $3
```

```
LO = $2 / $3
HI = $2 % $3
```



- **Per recuperare il risultato**




```
mfhi $4
mflo $5
```

Nota: esistono anche le versioni unsigned: `multu` e `divu`

Costanti

- Le costanti "piccole" sono le piu' usate (50% dei casi)

```
e.g.,  A = A + 5;
       B = B + 1;
       C = C - 18;
```

- Approccio RISC:
  - Mettere le costanti "piccole" nell'istruzione
  - Mettere le costanti meno frequenti in memoria
  - Creare un registro hard-wired (come $zero) per la costante 0
- Istruzioni MIPS :

```
addi $29, $29,  4
slti $8,  $18, 10
```

  - per codificare queste istruzioni si usa ancora il formato I
    (I sta per 'immediato': uno degli operandi e' dentro l'istruzione stessa)
    QUINDI la costante piccola e' un numero da -32768 a 32767

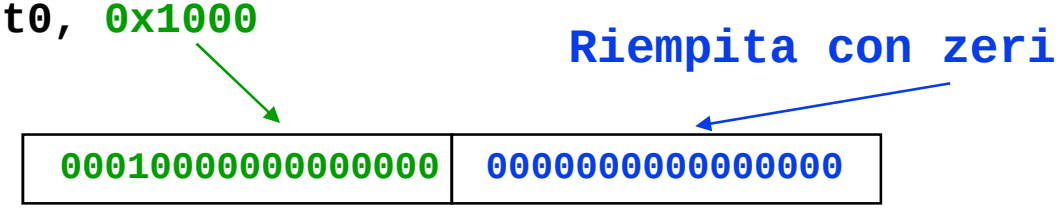
Nota1: le varianti unsigned sono `addiu` and `sltiu` (→`0...65535`)
Nota2: la "`subi`" non esiste... si fa con la `addi` coll'immediato negato

Che si fa con le costanti "grandi"?

- Vorremo caricare costanti a 32 bit nei registri
  - Es. 0x1000'2222
  - La parte bassa (0x2222) la potrei caricare con addiu $t0, $0, 0x2222
  - Come carico pero' la parte alta ?

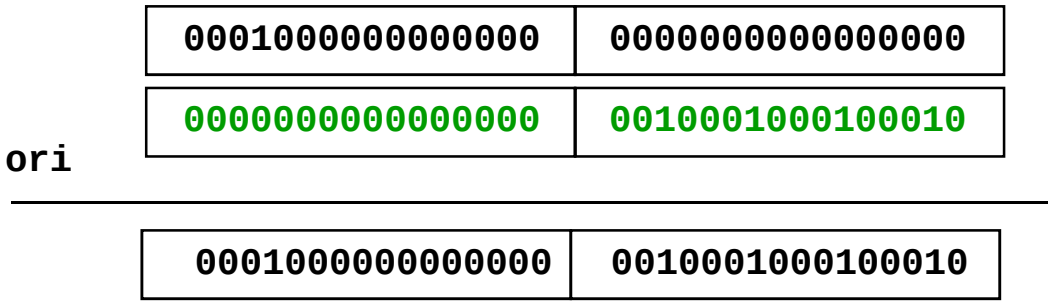
1) Si introduce una nuova istruzione ("load upper immediate" = lui)

```
lui $t0, 0x1000
```



2) Si usa successivamente una addiu (o una ori) per caricare la parte bassa

```
ori $t0, $t0, 0x2222
```



Pseudoistruzioni

- La coppia di istruzioni
  - `lui $1, 16bit_alti`
  - `ori $1, $1, 16bit_bassi`

E' talmente frequente che a livello di assembler si puo' creare una 'pseudoistruzione' la = "load address"

  - `la $1, costante_a_32_bit`
    - dove `costante_a_32_bit:=(16bit_alti|16bit_bassi)`
    - viene automaticamente tradotta nella coppia di istruzioni native lui+ori

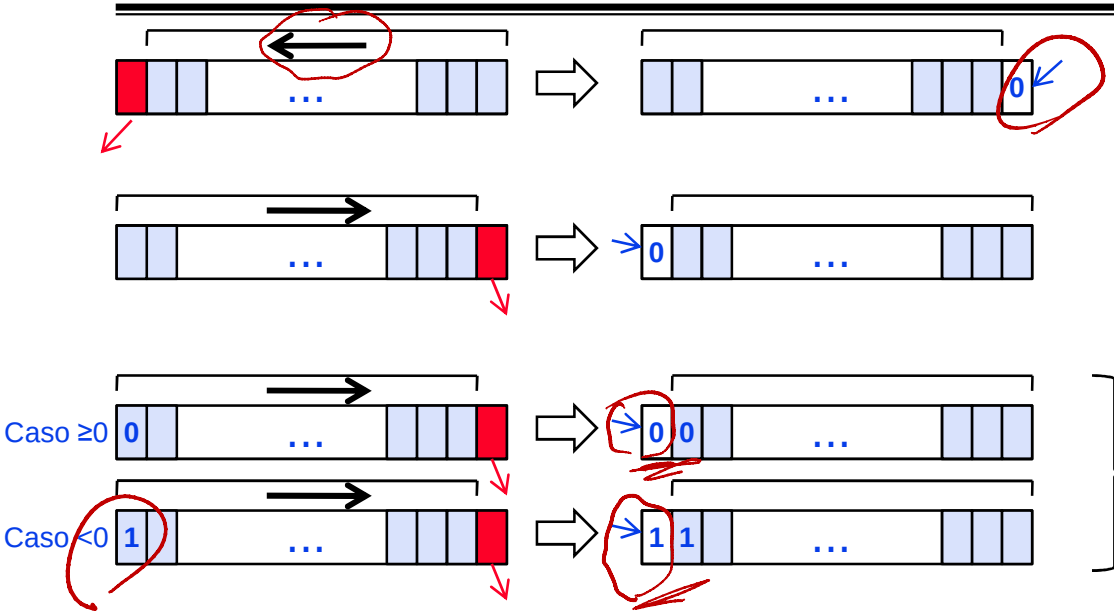
- Esistono altre pseudoistruzioni, ad es.
  - `muli $1, $2, costante_a_16_bit`
    - L'istruzione nativa e' pero' la mult
- Noi cercheremo di usare sempre istruzioni native
  - Unica pseudoistruzione ammessa: `la`

MIPS: istruzioni logiche

Instruzione	*Esempio*	*Significato*	*Commento*
and	and $1,$2,$3	$1 = $2 & $3	3 reg. operands; Logical AND
or	or $1,$2,$3	$1 = $2 | $3	3 reg. operands; Logical OR
xor	xor $1,$2,$3	$1 = $2 ⊕ $3	3 reg. operands; Logical XOR
nor	nor $1,$2,$3	$1 = ~($2 |$3)	3 reg. operands; Logical NOR
and immediate	andi $1,$2,10	$1 = $2 & 10	Logical AND reg, constant
or immediate	ori $1,$2,10	$1 = $2 | 10	Logical OR reg, constant
xor immediate	xori $1, $2,10	$1 = $2 ⊕ 10	Logical XOR reg, constant
shift left logical	sll $1,$2,10	$1 = $2 << 10	Shift left by constant
shift right logical	srl $1,$2,10	$1 = $2 >> 10	Shift right by constant
shift right arithm.	sra $1,$2,10	$1 = $2 >> 10	Shift right (sign extend)
shift left logical	sllv $1,$2,$3	$1 = $2 << $3	Shift left by variable
shift right logical	srlv $1,$2, $3	$1 = $2 >> $3	Shift right by variable
shift right arithm.	srav $1,$2, $3	$1 = $2 >> $3	Shift right arith. by variable

Nota1: "`not`" non esiste... si fa con `nor` e un operando `$0`
Nota2: "`nori`" non esiste... si fa con la `andi` e negando gli operandi

TRASLAZIONE (SHIFT)



Shift Left Logical (SLL)

Shift Right Logical (SRL)

Shift Right Arithmetic (SRA)

Nota: SLA non esiste ...=SLL

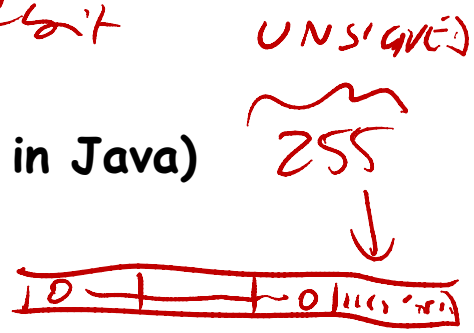
Istruzioni per la manipolazione di stringhe

- Fino a questo punto abbiamo visto istruzioni che operano su quantita' a 32 bit
- Per accedere al singolo byte sono a disposizione
  - Utile per le stringhe di caratteri ASCII

lb	$5, 0($4)	"load byte"
sb	$5, 0($4)	"store byte"

- Per accedere alla half-word (16 bit) ci sono
  - Utile per le stringhe di caratteri UNICODE (es. in Java)

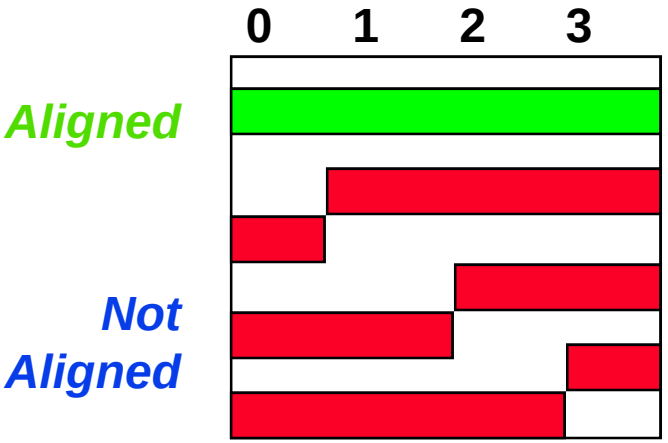
lh	$5, 0($4)	"load half-word"
sh	$5, 0($4)	"store half-word"



Nota: in fase di caricamento (load), dovendo porre la quantita' da 8 o 16 bit in 32 bit, viene automaticamente effettuata l'estensione del segno. Se cio' non si vuole si devono usare `lbu` (al posto di `lb`) e `lhu` al posto di `lh`→estensione con 0

Restrizioni sull'allineamento degli indirizzi

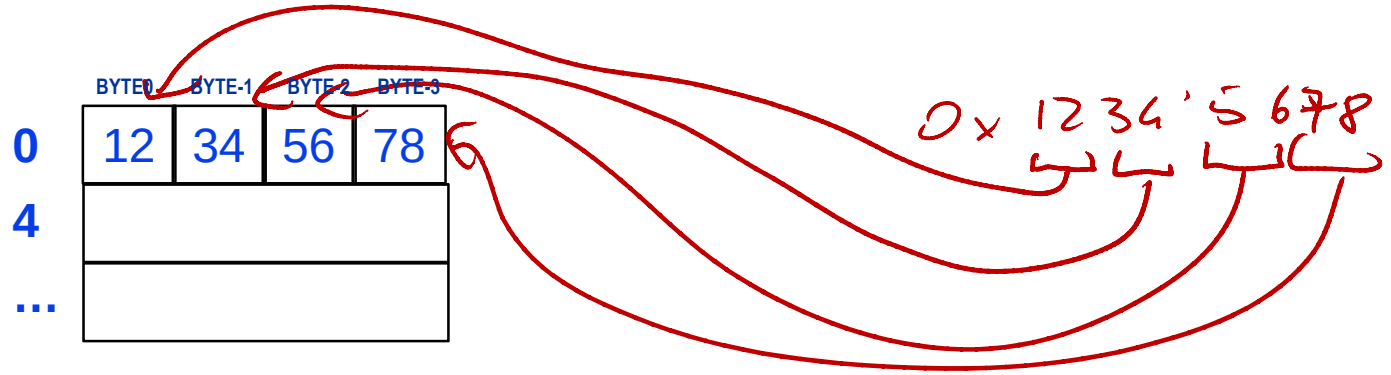
- La memoria e' classicamente indirizzata "al byte"
  - Le istruzioni di load e store usano indirizzi al byte, pero'
    - `lw` e `sw` trasferiscono 32 bit
    - `lh`, `lhu` e `sh` trasferiscono 16 bit
    - solo `lb`, `lbu`, `lb` trasferiscono 8 bit
- Nel MIPS, l'indirizzo non puo' essere qualunque…
  - per `lw` e `sw` deve essere allineato ad un multiplo di 4
  - Per `lh`, `lhu`, `sh` deve essere allineato ad un multiplo di 2
- Esempi di dati ALLINEATI e NON ALLINEATI "alla word"



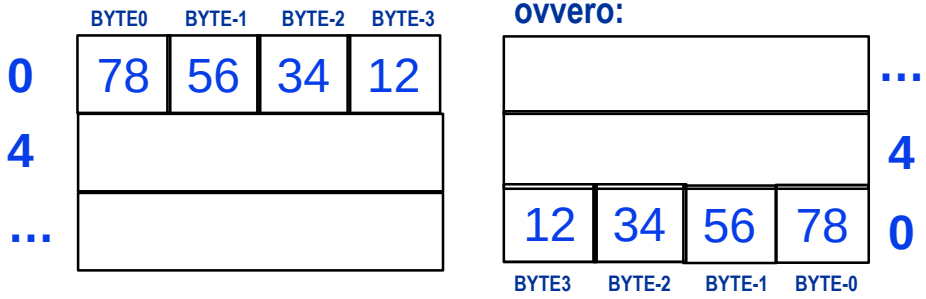
Nota: se si specifica un indirizzo non allineato rispetto a quanto l'istruzione desidera, viene generata una eccezione

Ordinamento dei byte (Endianess)

- **Big Endian: si inizia con la parte "piu' grossa" ovvero a indirizzi piu' bassi i byte piu' significativi: IBM 360/370, Motorola 68k, MIPS, Sparc(pre-v9)**
  - **Se ad esempio devo memorizzare 0x12345678 in B.E.**



- **Little Endian: si inizia con la parte "piu' piccola" ovvero a indirizzi piu' bassi i byte meno significativi: Intel 80x86, DEC Vax, DEC Alpha (Windows NT)**



Nota: e' importante sapere l'endianess di una macchina quando si scandiscono stringhe o dati byte-a-byte SPECIALMENTE dopo un TRASFERIMENTO DA RETE in cui la macchina remota puo' avere una endianess diversa

Nota2: MIPS piu' recenti - es. quello prodotto da LSI-Logic (LR-3000) - permettono di scegliere la endianess preferita ! Cosi' sono «bi-endian» anche SPARCv9, HP-PA, Alpha, ARM

Esempio: traduzione dei puntatori del C in assembly

```
typedef struct tabletag {
   int i;
   char *cp, c;
} table;
table z[20];
table *p;
char c;
```

DEF.

Ipotesi:

&z ≡ $7
&p ≡ $8
&c ≡ $9

```
p=&z[4];
```

⇒

```
addi  $1, $7, 36
sw    $1, 0($8)
```

```
c=p->c;
```

⇒

```
(addi  $1, $7, 36)
lb     $2, 8($1)
sb     $2, 0($9)
```