Architetture dei Calcolatori

Il Set di Istruzioni MIPS

Prof. Francesco Lo Presti

La programmazione dei calcolatori

- Linguaggio macchina (codifica con numeri binari)
  - Linguaggio direttamente eseguibile dal calcolatore
  - Attività di programmazione lunga e noiosa
  - Facile commettere errori
- Linguaggio assembler (o assembly)
  - Rappresentazione simbolica del linguaggio macchina
    - ✓ a cui aggiungere chiamate di sistema e macro
  - Più comprensibile del linguaggio macchina (simboli anziché sequenze di bit)
  - Tradotto dall'assemblatore in linguaggio macchina
    - ✓ Dalla forma simbolica dell'istruzione macchina al corrispondente formato binario
- Linguaggi ad alto livello
  - Tradotti dal compilatore in assembler

Benefici dei linguaggi ad alto livello

- Notazione vicina al linguaggio corrente (maggiore *espressività* e *leggibilità*)
- Indipendenza dalle caratteristiche peculiari dell'architettura (processore) su cui il programma va eseguito (*portabilità*)
  - Ideati non per essere compresi direttamente da macchine reali, ma da macchine astratte, in grado di effettuare operazioni più di alto livello rispetto alle operazioni elementari dei processori reali
- Permettono l'uso di librerie di funzionalità già scritte (riusabilità del codice)
- Incremento di *produttività*
  - Facilitano la programmazione, svincolandola dalla conoscenza dei dettagli architetturali della macchina utilizzata

Vantaggi e svantaggi dell'assembler

- Vantaggio: la dipendenza dall'architettura del calcolatore permette
  - Ottimizzazione delle prestazioni (maggiore *efficienza*)
  - Programmi (potenzialmente) più compatti
  - Massimo sfruttamento delle potenzialità dell'hardware sottostante
  - Importante per
    - ✓ Programmare controller di processi e macchinari (anche real-time)
    - ✓ Programmazione di apparati limitati (embedded computer, dispositivi portatili, telefonini cellulari, …)
- Principali svantaggi
  - Strutture di controllo in forme limitate (minore espressività)
  - Necessario conoscere i dettagli dell'architettura
  - Mancanza di portabilità su architetture diverse
  - Difficoltà di comprensione, possibile lunghezza maggiore, facilità di errore rispetto a programmi scritti in linguaggio ad alto livello (minore produttività del programmatore)

Architettura MIPS (RISC)

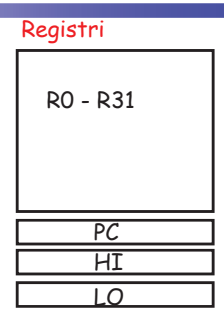
- Sviluppata e progettata a Stanford (USA)
  - Progettata nei primi anni '80
  - Prodotta e sviluppata da MIPS Technologies negli anni '90 (http://www.mips.com)
- Tecnologia attualmente utilizzata da
  - Sony (Playstation, Playstation 2, AIBO)
  - Nintendo 64
  - Router CISCO
  - Stampanti, macchine fotografiche digitali, palmtop
  - Set-top box, DVD
  - TV al plasma
- Rappresenta un buon modello architetturale per la didattica, perché è un'architettura semplice da comprendere

Principi di Progettazione ISA MIPS

- Simplicity favors regularity
  - Istruzioni di dimensione fissa – 32-bits
  - Numero limitato di formati istruzioni
- Smaller is faster
  - Set limitato di istruzioni
  - Numero limitato di registri
  - Numero limitato di modalita' di indirizzamento
- Good design demands good compromises
  - Tre formati istruzione
- Make the common case fast
  - Le istruzioni logico/aritmetiche operano sui registri (architettura load-store)
  - Uso di indirizzamento immediato

MIPS R3000 ISA

- Categorie di Istruzioni
  - Load/Store
  - Computational
  - Jump & Branch
  - Floating Point
    - ✓ coprocessor
  - Gestione della Memoria
  - Speciali
- 3 formati istruzione da 32 bit



I registri

- Il livello ISA dell'architettura MIPS richiede che
  - Gli operandi delle istruzioni provengano dai registri
    - ✓ Architettura di tipo *load-store* (o *registro-registro*)
- Il processore possiede un numero limitato di registri
  - Il processore MIPS possiede 32 registri di tipo general-purpose (GPR), ciascuno dei quali è composto da 32 bit (parola)
  - Il processore MIPS possiede ulteriori 32 registri da 32 bit per le operazioni in virgola mobile (floating point), detti FPR
- Per denotare i registri si usano nomi simbolici (convenzionali) preceduti da $
  - $s0, $s1, … per i registri che contengono variabili C
  - $t0, $t1, … per i registri di uso temporaneo
  - I registri possono anche essere indicati direttamente mediante il loro numero (0, …, 31) preceduto da $ (quindi $0, …, $31)

Convenzione dei nomi per i registri

0	$zero constant 0 (Hdware)
1	$at reserved for assembler
2	$v0 expression evaluation &
3	$v1 function results
4	$a0 arguments
5	$a1
6	$a2
7	$a3
8	$t0 temporary: caller saves
. . .	(callee can clobber)
15	$t7
16	$s0 callee saves
. . .	(caller can clobber)
23	$s7
24	$t8 temporary (cont'd)
25	$t9
26	$k0 reserved for OS kernel
27	$k1
28	$gp pointer to global area
29	$sp stack pointer
30	$fp frame pointer
31	$ra return address (Hdware)

Le costanti in MIPS

- Programmi fanno anche uso di costanti (spesso piccole)
  - ~50% operandi sono costanti
  - e.g., A = A + 5;
  - B = B + 1;
  - C = C - 18;
- Soluzioni
  - Hard-wired (come $zero)
  - …
- Permettere l' uso di costanti nelle istruzioni MIPS

```
addi $sp, $sp, 4
slti $t0, $t1, 10
andi $t0, $t0, 6
ori  $t0, $t0, 4
```


Istruzioni aritmetiche

- Istruzione aritmetiche assembler MIPS

```
add  $t0, $s1, $s2
sub  $t0, $s1, $s2
```

- Una singola operazione
- Esattamente tre operandi

destination ← source1 op source2

- Tutti gli operandi sono registri del processore (register file) ($t0,$s1,$s2)
- Ordine degli operandi e' fisso
  - Il primo operando e' il registro destinazione

Compilazione di istruzioni "complesse"

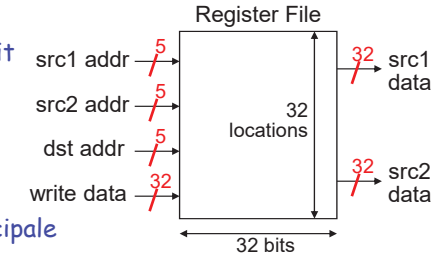
```
…
h = (b - c) + d
…
```

- Assumendo che b sia memorizzato nel registro $s1, c in $s2, e d in $s3 e che il risultato sia insererito in $s0, un modo di compilare l' istruzione C su indicata e' :

```
sub  $t0, $s1, $s2
add  $s0, $t0, $s3
```

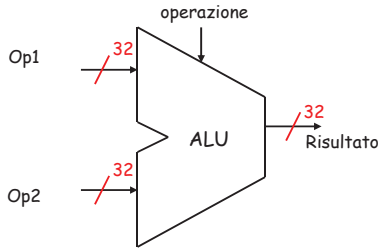
MIPS Register File

- Gli operandi delle istruzioni aritmetiche sono limitati al register file del processore (contenuto nel datapath)
  - trentadue registri da 32-bit
    - ✓ Due porte in lettura
    - ✓ Una porta in scrittura
- Registri
  - + veloci della memoria principale



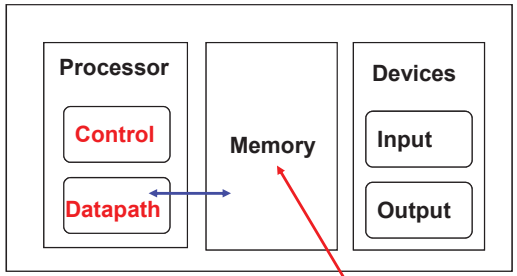
Arithmetic Logic Unit (ALU)

- L'unità logico-aritmetica è la rete combinatoria che svolge le operazioni logico-aritmetiche
  - somma, sottrazione
  - AND, OR
  - shift
  - ..
- due operandi



Registri e Memoria

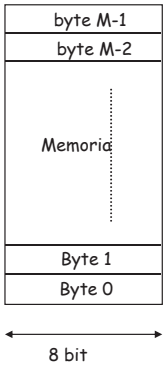
- Istruzioni aritmetiche operano sui registri
  - Vedremo poi il caso delle costanti
- Per eseguire operazioni, il compilatore DEVE associare registri alle variabili di un programma

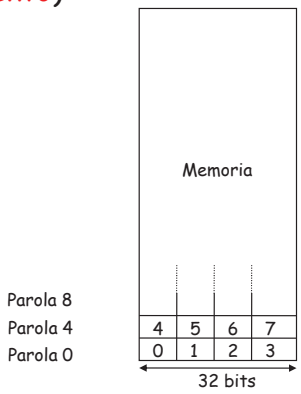


- Ma in genere i programmi hanno piu' variabili che registri (32)

Indirizzamento a byte

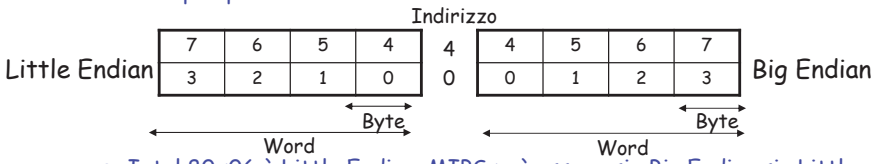
- MIPS (come la maggior parte delle architetture) indirizza i singoli byte
- Gli indirizzi delle parole sono invece multipli di 4 (vincolo di allineamento)





Ordinamento dei byte

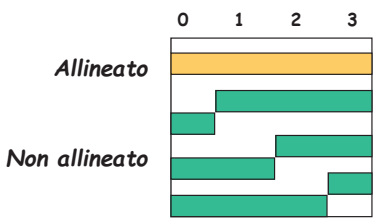
- Esistono due convenzioni per ordinare i byte all'interno di una parola (*endianess*)
  - Big Endian: il byte il cui indirizzo è x…00 è nella posizione più significativa della parola (big end)
  - Little Endian: il byte il cui indirizzo è x…00 è nella posizione meno significativa della parola (little end)
  - Esempio: parola di 32 bit



  - Intel 80x06 è Little Endian, MIPS può essere sia Big Endian sia Little Endian in dipendenza del valore logico su di un pin

Allineamento dei byte e terminologia

- L'allineamento richiede che la parola inizi ad un indirizzo multiplo della sua dimensione



- In MIPS, allineamento della parola ad un indirizzo multiplo di 4 (equivale a indirizzo mod 4 = 0)
  - Half word (16 bit) allineata ai multipli di 2
- La mancanza di allineamento causa un incremento della complessità dell'hardware ed un numero maggiore di accessi in memoria
- Terminologia delle sequenze di bit di particolare lunghezza

  4 bit: nibble
  8 bit: byte
  16 bit: half-word
  32 bit: word
  64 bit: double-word

Istruzione di trasferimento dati

- MIPS fornisce due istruzioni di base per il trasferimento di dati tra registri del processore e memoria
  - lw (load word): per trasferire una parola di memoria in un registro del processore
  - sw (store word): per trasferire il contenuto di un registro del processore in una parola di memoria
- Le istruzioni lw e sw richiedono come argomento l'indirizzo della locazione di memoria sulla quale devono operare

Istruzioni load e store

- L'istruzione di load trasferisce una copia dei dati contenuti in una specifica locazione di memoria ai registri del processore, lasciando inalterata la parola di memoria
  - Il processore invia l'indirizzo della locazione desiderata alla memoria e richiede un'operazione di *lettura* del suo contenuto
  - La memoria effettua la lettura dei dati memorizzati all'indirizzo specificato e li invia al processore
- L'istruzione di store trasferisce il contenuto di un registro del processore in una specifica locazione di memoria, sovrascrivendo il contenuto precedente di quella locazione
  - Il processore invia l'indirizzo della locazione desiderata alla memoria insieme ai dati che vi devono essere scritti, e richiede un'operazione di *scrittura*
  - La memoria effettua la scrittura dei dati all'indirizzo specificato

Istruzione lw

- In MIPS, l'istruzione lw ha tre argomenti
  - Il registro *destinazione* in cui caricare la parola letta dalla memoria
  - Una costante o spiazzamento (*offset*)
  - Un registro base (*base register*) che contiene il valore dell'indirizzo base (*base address*) da sommare all'offset
- Indirizzamento *registro base* (o *con spiazzamento*)
  - L'indirizzo della parola di memoria da caricare nel registro è ottenuto sommando il contenuto del registro base alla costante
- Esempio

```
lw $s1, 100($s2)      # $s1 = M[$s2+100]
```

  - Al registro destinazione $s1 è assegnato il valore contenuto all'indirizzo di memoria ($s2+100)

Istruzione sw

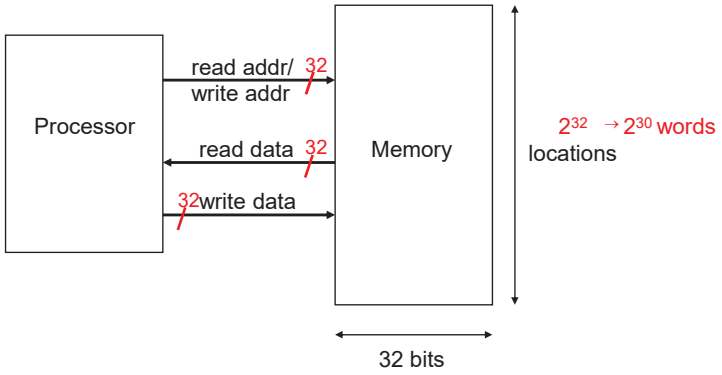
- In MIPS, l'istruzione sw ha tre argomenti, analogamente all'istruzione lw
  - Il registro *sorgente*, il cui contenuto deve essere scritto in memoria
  - Una costante o spiazzamento (*offset*)
  - Un registro base (*base register*) che contiene il valore dell'indirizzo base (*base address*) da sommare all'offset
- Indirizzamento *registro base* (o *con spiazzamento*)
  - L'indirizzo della parola di memoria da sovrascrivere è ottenuto sommando il contenuto del registro base alla costante
- Esempio

```
sw $s1, 100($s2)      # M[$s2+100] = $s1
```

  - Alla locazione di memoria di indirizzo ($s2+100) è assegnato il valore contenuto nel registro sorgente $s1

Interconnessione Processore-Memoria

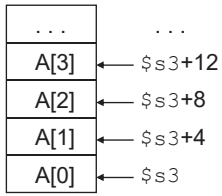
- La memoria e' vista come un array mono-dimensionale
  - Le cui locazioni sono indicizzate dagli indirizzi di memoria



Compilare con Load e Store

- Assumendo che b sia in `$s2` e che l' indirizzo base dell' array A sia in `$s3`, quali sono le istruzioni assembler MIPS corrispondenti all' istruzione C

```
A[8] = A[2] - b
```




```
lw   $t0, 8($s3)
sub  $t0, $t0, $s2
sw   $t0, 32($s3)
```

Compilare con un indice come variabile

- Assumendo che A sia un array di 50 elementi con indirizzo base in `$s4`, e le variabili b, c, ed i siano in `$s1`, `$s2`, e `$s3`, rispettivamente, quali sono le istruzioni assembler MIPS corrispondenti all' istruzione C

```
c = A[i] - b
```

```
add   $t1, $s3, $s3     #array index i is in $s3
add   $t1, $t1, $t1     #temp reg $t1 holds 4*i
add   $t1, $t1, $s4     #addr of A[i]
lw    $t0, 0($t1)
sub   $s2, $t0, $s1
```


Register spilling

- In genere i programmi usano più variabili di quanti sono i registri del processore
- Il compilatore cerca di mantenere le variabili usate più frequentemente nei registri e le altre variabili in memoria, usando istruzioni di load/store per trasferire le variabili tra registri e memoria
  - Il tempo di accesso ai registri è minore del tempo di accesso alla memoria
- La tecnica di mettere le variabili meno usate (o usate successivamente) in memoria viene detta *register spilling*

Load e Store di byte

- MIPS fornisce istruzioni per load e store di byte
  - utili per la manipolazioni di caratteri

```
lb    $t0, 1($s3)   #load byte from memory
sb    $t0, 6($s3)   #store byte to  memory
```

- Quale byte viene trasferito?
  - load byte legge un byte dalla memoria e lo copia negli 8 bit meno significativi del registro destinazione
    - ✓ gli altri bit rimangono inalterata
  - store byte copia il contenuto degli 8 bit meno significativi di un registro in un byte in memoria

Esempio

- Come si modifica il contenuto della memoria dopo l' esecuzione del seguente frammento di codice?

```
add   $s3, $zero, $zero
lb    $t0, 1($s3)
sb    $t0, 6($s3)
```

Memoria	
0x0 0 0 0 0 0 0 0	24
0x0 0 0 0 0 0 0 0	20
0x0 0 0 0 0 0 0 0	16
0x1 0 0 0 0 0 1 0	12
0x0 1 0 0 0 4 0 2	8
0xF F F F F F F F	4
0x0 0 9 0 1 2 A 0	0
Data	Indirizzi delle Parole (in decimale)

M[4] = 0xFFFF90FF

- ed il contenuto di $t0?

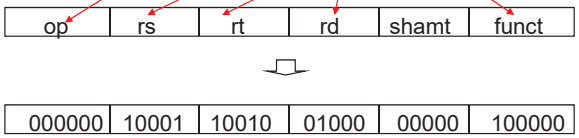
$t0 = 0x00000090

- e nel caso di architettura Little Endian?

M[4] = 0xFF12FFFF
$t0 = 0x00000012

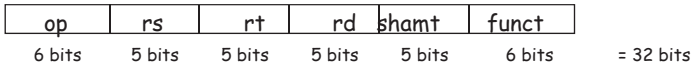
Linguaggio Macchina – Istruzioni Aritmetiche

- Le istruzioni, come i registri e le parole di memoria sono di 32 bit
  - Esempio: `add $t0, $s1, $s2`
  - Corrispondenza registri `$t0=$8, $s1=$17, $s2=$18`
- Formato Istruzioni:



- Istruzioni di tipo R
  - Operazioni logico aritmetiche fra registri

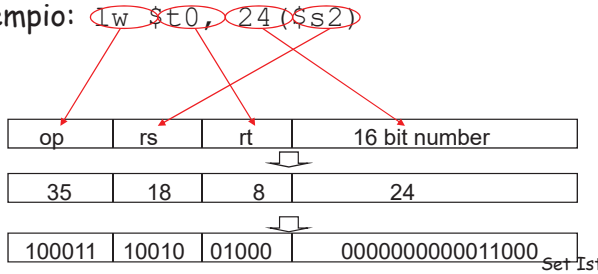
Campi Istruzione (tipo R)



- *op* opcode – operazione da eseguire
- *rs* indirizzo primo registro operando
- *rt* indirizzo del secondo registro operando
- *rd* indirizzo del registro destinazione
- *shamt* scorrimento (per le istruzioni di shift)
- *funct* Codice funzione che specifica la variante dell'operazione da eseguire

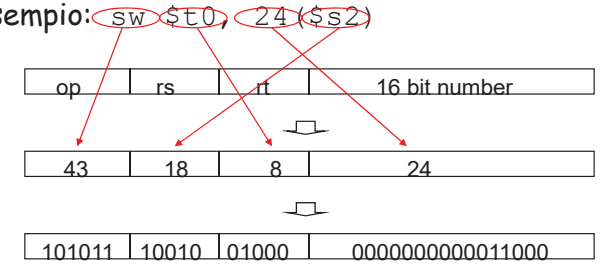
Linguaggio Macchina – Formato I

- Il formato di tipo R non è adatto a rappresentare istruzioni di tipo load/store
  - All'offset sarebbe riservato un campo di 5 bit (al massimo costanti di dimensione pari a 32)
- Si introduce un nuovo tipo di formato istruzione
  - Tipo I (immediatate) per istruzioni di trasferimento dati
- Esempio: `lw $t0, 24($s2)`



Linguaggio Macchina – Istruzione Store

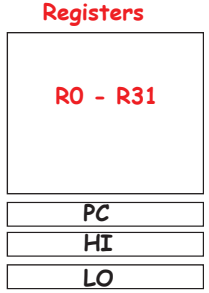
- Esempio: `sw $t0, 24($s2)`

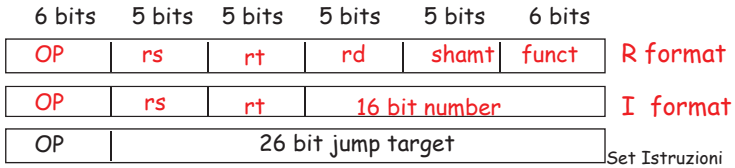


- Un indirizzo a 16-bit indica che l' accesso e' limitato a locazioni di memoria entro l' intervallo di $\pm 2^{13}$ o 8,192 parole ($\pm 2^{15}$ o 32,768 bytes) dell' indirizzo di base `$s2`

Review: MIPS R3000 ISA

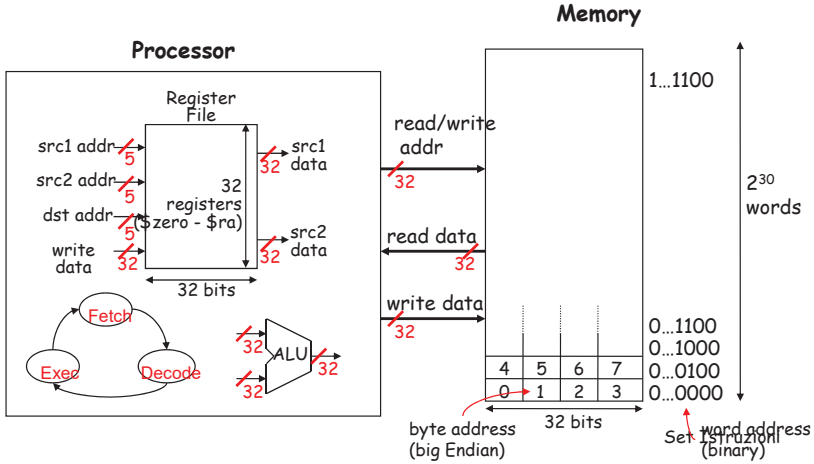
- Categorie di Istruzioni
  - Load/Store
  - Computational
  - Jump & Branch
  - Floating Point
    - ✓ coprocessor
  - Gestione della Memoria
  - Speciali
- 3 formati istruzione da 32 bit





Organizzazione MIPS (fino ad ora)

- Istruzioni aritmetiche – da/verso il register file
- Istruzioni Load/store di word – da/verso la memoria



Istruzioni per operazioni logiche: shift

- Shift (traslazione) dei bit di una parola a destra o sinistra
- sll (shift left logical)
- srl (shift right logical)
- Esempio: assumiamo che il registro $s0 contenga il valore 13

```
sll $t2, $s0, 8          # $t2 = 13 x 2^8 = 3328
```

- Formato di tipo R: si utilizza il campo shamt (shift amount)

op	rs	rt	rd	shamt	funct
6 bit	5 bit	5 bit	5 bit	5 bit	6 bit

```
sll $t2, $s0, 8
```

0	0	16	10	8	0

- Traslare a sinistra di i bit corrisponde a moltiplicare per 2^i
  - ✓ Utile per moltiplicare per 4 l'indice i di un array

Istruzioni per operazioni logiche

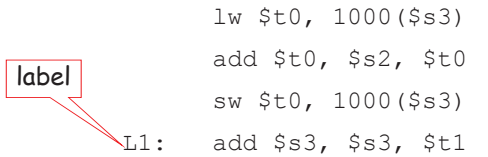
- Istruzione and per effettuare l'AND bit a bit tra due registri
  - Esempio `and $t0, $t1, $t2 # $t0 = $t1 & $t2`
    - ✓ Se $t2 = 0000 0000 0000 0000 0000 1101 0000 0000
    - ✓ e $t1 = 0000 0000 0000 0000 0011 1100 0000 0000
    - ✓ Allora, dopo l'esecuzione dell'istruzione and
    - ✓ $t0 = 0000 0000 0000 0000 0000 1100 0000 0000
- Istruzione or per effettuare l'OR bit a bit tra due registri
  - Esempio `or $t0, $t1, $t2 # $t0 = $t1 | $t2`
- Istruzione nor per effettuare il NOR bit a bit tra due registri
  - Esempio `nor $t0, $t1, $t2 # $t0 = ~($t1 | $t2)`
- Le istruzioni and, or e nor seguono il formato di tipo R

Istruzioni di scelta e controllo del flusso

- Queste istruzioni
  - Alterano l'ordine di esecuzione delle istruzioni
    - ✓ La prossima istruzione da eseguire non è quella successiva all'istruzione corrente
  - Permettono di eseguire cicli e condizioni
    - ✓ Ripetizione di sequenze di istruzioni in dipendenza dai risultati precedenti
- Due classi di istruzioni
  - Istruzioni di *conditional branch* (salto condizionato) del tipo "if condizione then esegui X"
    - ✓ Permettono di selezionare la prossima istruzione da eseguire in dipendenza di un valore calcolato
  - Istruzioni di *jump* (salto incondizionato) del tipo "esegui X come prossima istruzione"
    - ✓ Il salto viene sempre eseguito

Le etichette nei programmi

- Le istruzioni di un programma assembler possono avere delle etichette (*label*)



- Le istruzioni di branch permettono di specificare l'etichetta dell'istruzione successiva a seconda del risultato di un test (di uguaglianza o disuguaglianza)
- Anche le istruzioni di jump permettono di specificare l'etichetta dell'istruzione a cui saltare

Salto Condizionato

- Due Istruzioni per il salto condizionato:

```
bne $s0, $s1, Label     #go to Label if $s0≠$s1
beq $s0, $s1, Label     #go to Label if $s0=$s1
```

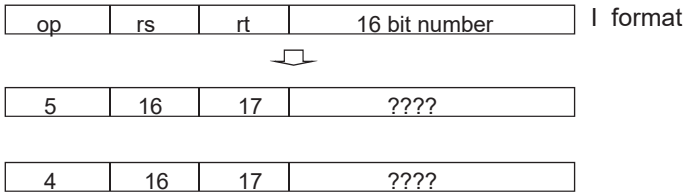
- Esempio: `if (i==j) h = i + j;`

```
        bne $s0, $s1, Lab1
        add $s3, $s0, $s1
Lab1:   ...
```


Salto Condizionato: Codice Macchina

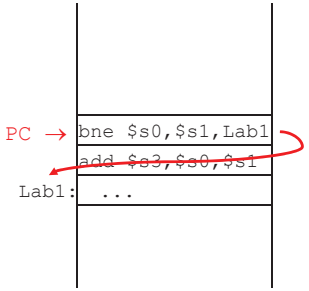
- Istruzioni:

```
bne $s0, $s1, Label   #go to Label if $s0≠$s1
beq $s0, $s1, Label   #go to Label if $s0=$s1
```



- Come si specifica l' indirizzo di salto?

Calcolo dell' indirizzo di salto



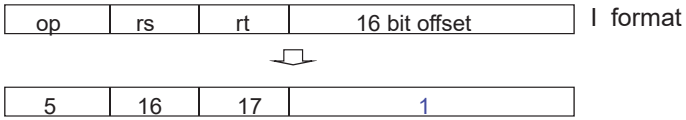
- Si usa il valore di un registro (come in lw e sw) al quale si aggiunge l' offset a 16-bit
  - Quale registro?
    - ✓ Registro dell' indirizzo istruzione
      - PC=Program Counter
    - ✓ PC viene aggiornato (PC<-PC+4) durante il ciclo di prelievo
    - ✓ Durante la fase di esecuzione is suo valore e' pari all' indirizzo della prossima istruzione
  - Limita la distanza del salto a -2^{15} to $+2^{15}-1$ istruzioni dall' istruzione che segue il salto
    - ✓ Problema? No
  - Principio di localita'

Salto Condizionato: Esempio Codice Macchina

- Codice Assembler

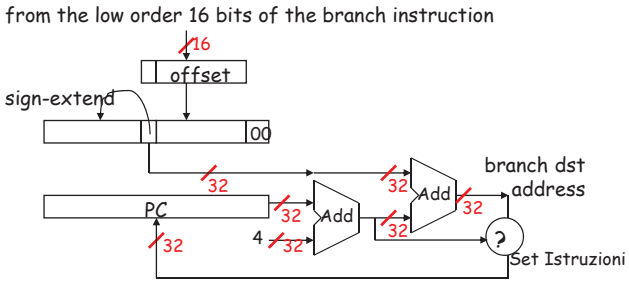
```
          bne $s0, $s1, Lab1
          add $s3, $s0, $s1
Lab1:     ...
```

- Codice macchina `bne`:

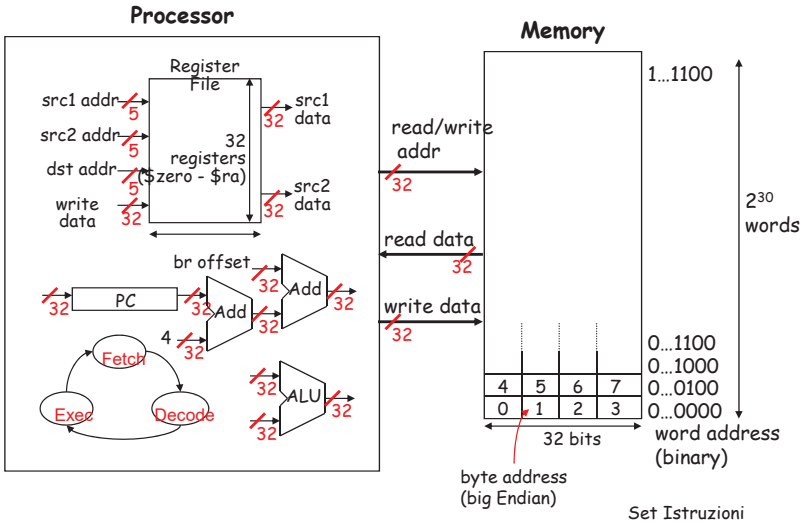


...e la sua implementazione

- Il contenuto aggiornato di PC (PC+4) viene sommato ai
- 16 bit meno significativi dell' istruzione di salto
  - Convertito in un valore a 32 bit
    - ✓ Concatenando 2 zeri a destra (x4)
    - ✓ Estendendo il segno dei risultanti 18 bits
- Il risultato viene usato per aggiornare PC in caso di salto



Organizzazione MIPS (fino ad ora)



Salto incondizionato

- Istruzione jump

```
j   label       #go to label
```

- Esempio:

```
if (i!=j)
  h=i+j;
else
  h=i-j;
```

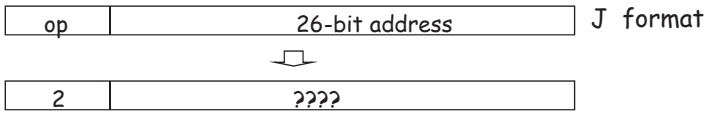
```
        beq  $s0, $s1, Else
        add  $s3, $s0, $s1
        j    Exit
Else:   sub  $s3, $s0, $s1
Exit:   ...
```


Salto incondizionato: Codice Macchina

- Istruzione:

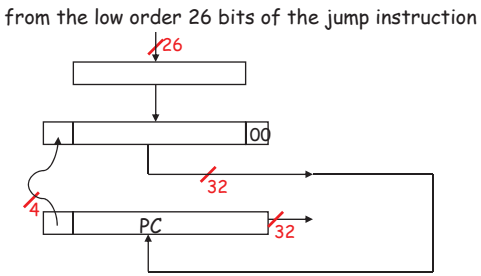
```
j   label       #go to label
```

- Codice Macchina:



Indirizzo di Salto ed implementazione

- L' indirizzo di destinazione (a 32 bit) e' specificato come indirizzo assoluto ottenuto
  - Concatenando i 4 bit piu' significativi del valore corrente di PC (PC+4) ai 26 bit
  - Concatenando 00 come i due bit meno significativi



Esercizi

- Assumendo che l'indirizzo base di A (B) sia in $s3 ($s4), e che la dimensione dell'array A sia memorizzata in $s5, scrivere in assembler le due seguenti porzioni di codice C:

1. Sommare tutti gli elementi di un array A di 100 interi che siano in posizione pari
   - Ovvero sum = A[0] + A[2] + … + [A98]
   - Codice C

```
sum = 0;
for (i=0; i<100; i+=2)
    sum = sum + A[i];
```

2. Sommare tutti gli elementi di un array A di 10 interi e memorizzare il risultato nell'elemento i-esimo dell'array B
   - Codice C

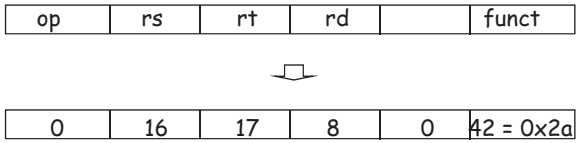
```
B[i] = 0;
for (j=0; j<10; j++)
    B[i] = B[i] + A[j];
```

Istruzione slt

- slt (set if less than):

```
slt $t0, $s0, $s1     # if $s0 < $s1
                      #     then
                      # $t0 = 1
                      #     else
                      # $t0 = 0
```

- Codice Macchina (formato R)



Istruzione slt

- Con slt, beq e bne si possono implementare tutti i test sui valori di due variabili (=,!=,<,<=,>,>=)

- Esempio
  - Codice C

    if (i < j) k = i + j;
    else k = i – j;

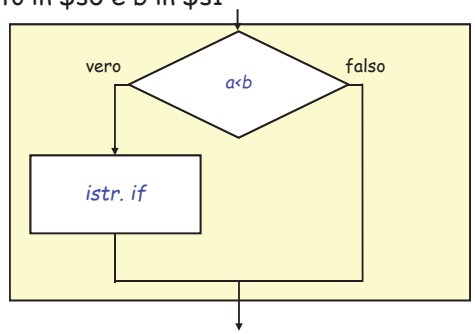
  - Codice MIPS
    - ✓ i,j,k associate a $s0, $s1 e $s2

```
slt $t0, $s0, $s1
beq $t0, $zero, Else
add $s2, $s0, $s1
j Exit
Else: sub $s2, $s0, $s1
Exit:
```


Compilazione Costrutto if

- si assume che a sia memorizzato in $s0 e b in $s1

```
if ( a<b ) {
    istr. if
}
```

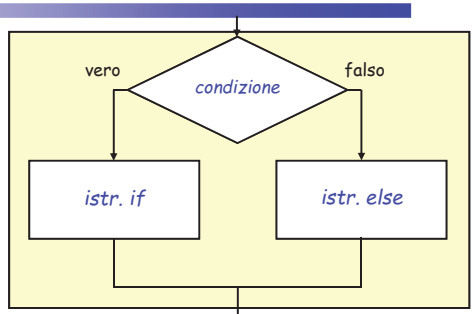



```
    if ( a≥b ) goto Exit;
    istr. if
Exit: ...
```

```
    slt $t0, $s0, $s1     # t0=0 iff $s0≥$s1
    beq $t0,$zero, Exit
    # codice MIPS per istr. if
Exit:...
```


Compilazione Costrutto if-else

```
if ( a<b ) {
    istr. if
}
else {
    istr. else
}
```

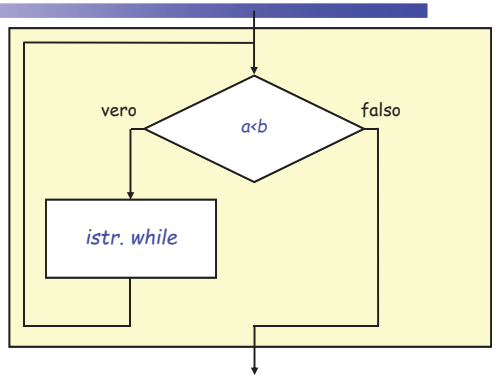



```
    if ( a≥b ) goto Else;
    istr. if
    goto Exit
Else:istr. else
Exit: ...
```

```
slt $t0, $s0, $s1     # t0=0 iff $s0≥$s1
beq $t0,$zero, Else
# codice MIPS per istr. if
j Exit
Else:# codice MIPS per istr. else
Exit:...
```

Compilazione Costrutto While

```
while ( a<b ) {
   istr. while
}
```

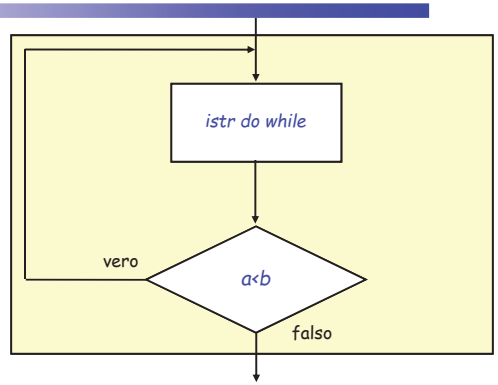



```
W_start: if(a≥b) goto Exit;
         istr. while
         goto W_start;
    Exit:...
```

```
W_start: slt $t0, $s0, $s1 # t0=0 iff $s0≥$s1
         beq $t0, $zero, Exit
         # codice MIPS per istr. while
         j W_start
   Exit:...
```


Compilazione Costrutto While Do-While

```
do {
   istr. do while
} while ( a<b);
```

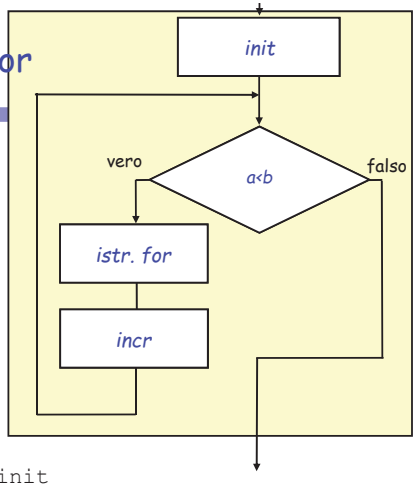



```
Do_start: istr. do while
          if(a≥b) goto Exit;
          goto Do_start;
    Exit: ...
```

```
Do_start: # codice MIPS per istr. do while
          slt $t0, $s0, $s1  # t0=0 iff $s0≥$s1
          beq $t0, $zero, Exit
          j Do_start
    Exit: ...
```


Compilazione Costrutto for

```
for ( init ; a<b ; incr ) {
  istr. for
}
```




```
          init
For_start: if(a≥b) goto Exit
          istr. for
          incr.
          goto For_start
    Exit: ...
```

```
          # codice MIPS per init
for_start: slt $t0, $s0, $s1 # t0=0 iff $s0≥$s1
          beq $t0, $zero, Exit
          # codice MIPS per istr. for
          # codice MIPS per incr
          j for_start
    Exit: ...
```


Array: Calcolo Indirizzi

Calcolo indirizzi in assembler:
Indirizzo di A [n] = indirizzo di A [0] + (n* sizeof (elementi di A))

- array di interi -> sizeof(elementi di A)=4

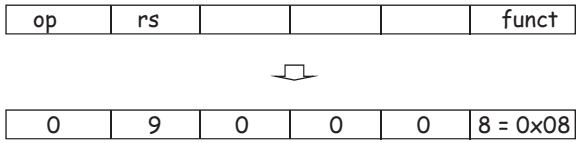
```
# $s0 = A.
# $t1 = n.

sll $t2, $t1, 2      # $t2=$t1*4
                     # offset di A[n] rispetto a A[0]
add $t2, $s0, $t2    # $t2=&A[n]=A+offset
sw  $t3, 0($t2)      # A[n] = $t3.
lw  $t3, 0($t2)      # $t3 = A[n].
```

Istruzione jr

- Istruzione jr (jump register)

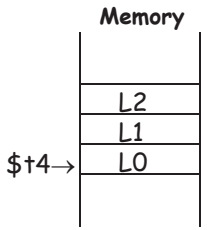
```
jr   $t1          #go to address in $t1
```

- Codice Macchina (formato R):



Compilazione Costrutto Switch

```
switch (k) {
    case 0:  h=i+j;  break; /*k=0*/
    case 1:  h=i+h;  break; /*k=1*/
    case 2:  h=i-j;  break; /*k=2*/
}
```

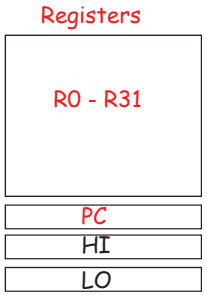


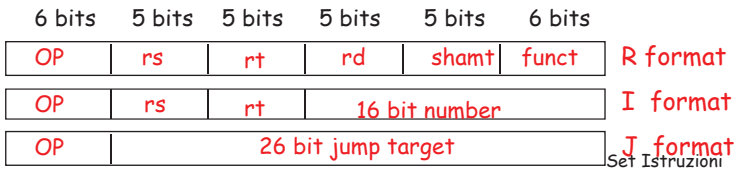
- Jump Table contiene gli indirizzi delle etichette L0, L1 e L2
- `$t4` punta alla JT, `k` in `$s2`

```
        sll   $t1, $s2, 2          #$t1 = 4*k
        add   $t1, $t1, $t4        #$t1 = addr of JT[k]
        lw    $t0, 0($t1)          #$t0 = JT[k]
        jr    $t0                  #jump based on $t0
L0:     add   $s3, $s0, $s1        #k=0 so h=i+j
        j     Exit
L1:     add   $s3, $s0, $s3        #k=1 so h=i+h
        j     Exit
L2:     sub   $s3, $s0, $s1        #k=2 so h=i-j
Exit:   . . .
```


Review: MIPS R3000 ISA

- Categorie di Istruzioni
  - Load/Store
  - Computational
  - Jump & Branch
  - Floating Point
    - ✓coprocessor
  - Gestione della Memoria
  - Speciali
- 3 formati istruzione da 32 bit



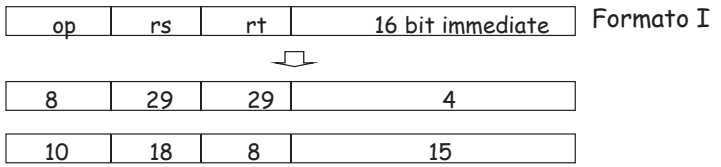


Istruzioni con indirizzamento immediato

- Esempi istr. con indirizzamento immediato MIPS:

```
addi $sp, $sp, 4     #$sp = $sp + 4
slti $t0, $s2, 15    #$t0 = 1 if $s2<15
```

- Codice Macchina:



- La costante e' all' interno della stessa istruzioni!
  - Formato I - formato Immediato
  - Limita i valori all' intervallo $+2^{15}-1$ to -2^{15}

Esercizi

- Scrivere un programma in assembler MIPS che, dato l'array di interi A di 100 elementi, effettua la somma solamente degli elementi di A minori di 10

```
sum = 0;
for (i=0; i<100; i++)
      if (A[i] < 10) sum = sum + A[i];
```

- Scrivere un programma in assembler MIPS che, dato l'array di interi A di 100 elementi, effettua la somma solamente degli elementi di A minori di 10 e maggiori di 5

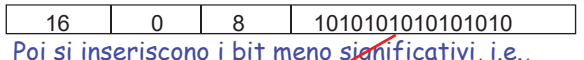
```
sum = 0;
for (i=0; i<100; i++)
      if ((A[i] > 5) && (A[i] < 10)) sum = sum + A[i];
```


Costanti maggiori di 16 bit? Istruzione lui/ori

- Si usa una soluzione in due passi
  1. Prima si caricano i bit piu' significativi con l' istruzione lui "load upper immediate"

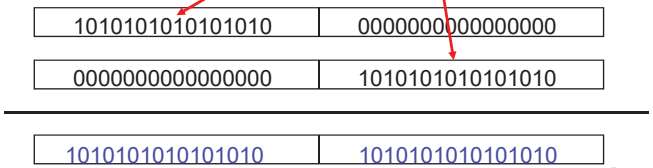
     `lui $t0, 1010101010101010`

     ✓ i 16 bit meno significativi vengono messi a 0



  2. Poi si inseriscono i bit meno significativi, i.e.,

     `ori $t0, $t0, 1010101010101010`



Caricamento di costanti a 32 bit

- Supponiamo di voler caricare il valore 4000000_{10} nel registro $s0
  - In binario
    0000 0000 0011 1101 0000 1001 0000 0000
- Separiamo il valore in binario in 2 valori a 16 bit
  - Il valore 61_{10} per i 16 bit più significativi
    In binario 0000 0000 0011 1101
  - Il valore 2304_{10} per i 16 bit meno significativi
    Pari a 0000 1001 0000 0000

```
lui $s0, 61        # 0000 0000 0011 1101 nei bit 31-16 di $s0
ori $s0, $s0, 2304 # 0000 1001 0000 0000 nei bit 15-0 di $s0
```


Salti ad Indirizzi "Distanti"

- bne e beq sono limitati ad una distanza compresa tra -2^{15} to $+2^{15}-1$ istruzioni dall' istruzione che segue il salto
- Il limite è superabile mediante l' uso combinato di branch e del jump
- La sostituzione viene fatta direttamente dall' assemblatore

```
beq $t0, $t1, L1
```

Indirizzo "Distante"

```
bne $t0, $t1, L2
j L1
L2:...
```

Salti ad Indirizzi "Distanti"

- I 4 bit più significativi del jump sono quelli del PC
  - l'indirizzo di salto è limitato all'interno della "pagina" di 256MB in cui si trova il PC
- Il limite è superabile mediante l' uso combinato di lui e del jr
- La sostituzione viene fatta direttamente dal linker

```
j   L1        assumiamo indirizzo L1= (L1H, L1L)
```

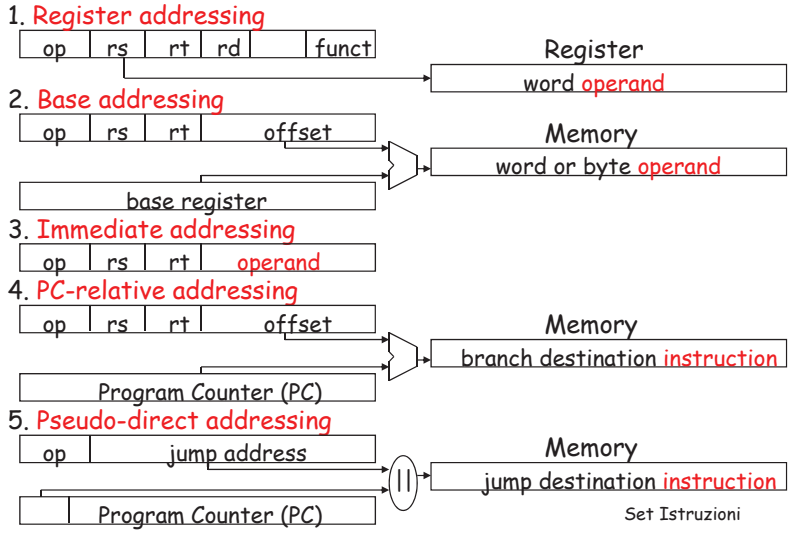
Indirizzo "Distante"

```
lui $t0, L1H
ori $t0, $t0, L1L
jr  $t0
```


Riepilogo Modalita' di Indirizzamento MIPS

1. A registro – l' operando e' il contenuto di un registro
2. Con registro base (a spiazzamento) – l' operando e' in una locazione di memoria data sommando al contenuto di un registro (base) una costante di 16-bit contenuta nell' istruzione
3. Immediato – l' operando e' una costante a 16 bit contenuta nell' istruzione
4. Relativo al PC –L' operando (istruzione) e' dato dalla sommma di PC con una costante a 16-bit contenuta nell' istruzione
5. Pseudo-diretto – L' operando (istruzione) e' una costante a 26-bit contenuta nell' istruzione a cui vengono prefissi i 4 bit piu' significativi di PC

Riepilogo Modalita' di Indirizzamento MIPS



Pseudoistruzioni

- Il linguaggio assembler rappresenta una "interfaccia" verso il software di livello più alto
- Per semplificare la programmazione, è possibile che siano aggiunte un insieme di pseudoistruzioni
- Le pseudoistruzioni sono un modo compatto ed intuitivo per specificare un insieme di istruzioni (come delle macro)
  - Non hanno una corrispondenza uno-a-uno con le istruzioni presenti nel linguaggio macchina
  - Non sono implementate in hardware
  - Ma sono abbastanza semplici da tradurre in linguaggio macchina e rappresentano un aiuto per il programmatore, in quanto arricchiscono il set di istruzioni assembler
- La traduzione della pseudoistruzione nelle istruzioni equivalenti è attuata automaticamente dall'assemblatore

Le pseudoistruzioni MIPS

- La pseudoistruzione move permette di copiare il valore di un registro in un altro registro
  - `move $t0, $t1`
    - ✓ Il registro $t0 prende il valore del registro $t1
- Come viene tradotta move in istruzioni MIPS?
  - `move $t0, $t1` ⇒ `add $t0, $zero, $t1`
- Si possono caricare costanti a 32 bit in un registro tramite la pseudoistruzione `li` (load immediate)
  - Vengono tradotte usando le istruzioni lui e ori
- E' possibile specificare anche costanti in base esadecimale

Le pseudoistruzioni MIPS (2)

- Le istruzioni di branch con slt possono essere complesse
- L'assembler MIPS mette a disposizione alcune pseudoistruzioni di branch che vengono aggiunte a beq e bne
- Serve un registro riservato per l'assemblatore ($at)
- Esempio: la pseudoistruzione blt (branch-on-less-than) viene implementata con slt e bne

```
blt $t0, $t1, 1000  ⇒   slt $at, $t0, $t1
                        bne $at, $zero, 1000
```

- Esistono anche
  - bgt (branch-on-greater-than),
  - bge (branch-on-greater-than-or-equal)
  - ble (branch-on-less-than-or-equal)
  - Per esercizio, determinare come sono implementate in MIPS

Valori con segno e senza

- Istruzioni di caricamento del byte
  - lb e lbu (load byte unsigned)
  - lb: tratta il byte come un numero con segno
    - ✓ Effettua l'estensione del segno sulla parola
  - lbu: non effettua l'estensione del segno (numero senza segno)
    - ✓ Per il caricamento dei byte si utilizza lbu
- Istruzioni di confronto
  - slt e sltu (set on less than unsigned)
  - Esempio
    - ✓ $s0 contiene 1111 1111 1111 1111 1111 1111 1111 1111
    - ✓ $s1 contiene 0000 0000 0000 0000 0000 0000 0000 0001
    - ✓ `slt $t0, $s0, $s1` `# $t0 = 1`
    - ✓ `sltu $t1, $s0, $s1` `# $t1 = 0`

Overflow nel MIPS

- Condizioni di overflow per somma e sottrazione

Operazione	A	B	Risultato
A+B	≥0	≥0	<0
A+B	<0	<0	≥0
A-B	≥0	<0	<0
A-B	<0	≥0	≥0

- Nel processore MIPS le istruzioni add, addi, sub causano eccezione nel caso di overflow
  - Eccezione: chiamata non prevista di una procedura appropriata per la gestione dell'eccezione
  - Registro EPC (Exception Program Counter) contenente l'indirizzo dell'istruzione che ha causato l'overflow
  - Istruzione mfc0 (move from system control) per copiare EPC in un registro generale
- Le istruzioni addu, addiu e subu (u per numeri unsigned) non causano eccezione in caso di overflow
  - Utili per manipolare indirizzi di memoria
  - Gli indirizzi di memoria sono solo positivi (sfruttano tutti e 32 i bit)

Istruzioni per moltiplicazione e divisione

- Problema per la moltiplicazione
  - Il risultato della moltiplicazione tra due interi a 32 bit richiede 64 bit
  - Non ci sono registri a 64 bit nel processore MIPS
- Problema per la divisione
  - Come risultato della divisione interessa conoscere sia il quoziente che il resto
  - Per il risultato occorrono due registri
- Soluzione nel processore MIPS
  - Due registri a 32 bit dedicati per le operazioni di moltiplicazione e divisione: registri Hi e Lo

Istruzione di moltiplicazione

- Per prodotto tra interi con segno
  - `mult rs, rt`: moltiplica il registro rs (moltiplicando) per il registro rt (moltiplicatore)
  - Mette nel registro Hi i 32 bit più significativi del prodotto e nel registro Lo i 32 bit meno significativi
  - Segue il formato R (registro destinazione e shamt nulli)
  - Esempio: `mult $t0, $t1`
    - ✓ Se $t0 = 10_{10} e $t1 = 6_{10} allora

Hi	0	Lo	60

- Per prodotto tra interi senza segno (unsigned)
  - `multu rs, rt`
- Entrambe le istruzioni MIPS di moltiplicazione ignorano l'overflow: è un compito lasciato al software (programmatore)

Istruzione di divisione

- Per divisione tra interi con segno
  - `div rs, rt`: divide il registro rs (dividendo) per il registro rt (divisore)
  - Mette nel registro Hi i 32 bit del resto e nel registro Lo i 32 bit del quoziente
  - Segue il formato R (registro destinazione e shamt nulli)
  - Esempio: `div $t0, $t1`
    - ✓ Se $t0 = 10_{10} e $t1 = 6_{10} allora

Hi	4	Lo	1

- Per divisione tra interi senza segno (unsigned)
  - `divu rs, rt`
- Entrambe le istruzioni MIPS di divisione ignorano l'overflow; ulteriore problema dato dalla divisione per 0

Segno del quoziente e resto

- Dividendo = quoziente · divisore + resto, ovvero
  $x = y \cdot (x \text{ div } y) + (x \bmod y)$
- Esempio: x=5 e y=3
  x div y = 1, x mod y = 2
- Esempio: x=-5 e y=3
  x div y = -1, x mod y = -5 - 3(-1) = -2
  In alternativa, anche
  x div y = -2, x mod y = 1
  ma sarebbe $-(x \div y) \neq (-x) \div y$!
  Quindi: segno del resto uguale al segno del dividendo
- Non c'è accordo tra i linguaggi di programmazione ad alto livello (es. x=-5 e y=3)
  - C: divisione e modulo indefiniti
  - Fortran: come esempio
  - MIPS: come esempio

Utilizzo dei registro Hi e Lo

- Istruzioni mflo/mfhi: copiano il contenuto dei registri Lo/Hi in un registro indicato
  - `mflo $s1`: il contenuto del registro Lo è copiato in $s1
  - `mfhi $s0`: il contenuto del registro Hi è copiato in $s0
- Esempio:

```
div  $s2, $s3
mflo $s1    # quoziente in $s1
mfhi $s0    # resto in $s0
```

- Entrambe le istruzioni mflo e mfhi seguono il formato R
  - primo operando nullo
  - secondo operando nullo
  - shamt nullo