Assembler MIPS R2000/3000

Alberto Montresor

Programma delle lezioni

- **Introduzione ai concetti di**
  - assembler, compilatore, linker, programma eseguibile
- **Introduzione all'assembly**
  - Sintassi del linguaggio assembly
  - Istruzioni del linguaggio assembly dei processori MIPS
  - Metodi di indirizzamento
- **Uso del tool SPIM**
  - Un semplice simulatore del processore MIPS R2000/3000
  - Come utilizzare SPIM

Programma delle lezioni

- **Esercitazioni sull'Assembly**
  - Scrittura di semplici programmi Assembly: ad es.
    - Programmi di ordinamento
    - Ricerche in array
- **Analisi dettagliata dei concetti di**
  - Assembler, Compilatore, Linker, Loader
  - Convenzioni riguardanti l'uso dei registri e delle chiamate di procedura
- **Uso del tool MPS**
  - Un simulatore MIPS più sofisticato
  - Come utilizzare MPS
  - Utilizzazione di MPS per l'interfacciamento fra programmi C e Assembly

Ricevimento

- **Dove?**
  - Ufficio dottorandi, piano terra, Dipartimento di Informatica
- **Quando?**
  - Prima / dopo / durante le lezioni
  - Nelle settimane in cui abbiamo lezione di martedì, il ricevimento è dalle 10 alle 16 del giovedì
  - Nelle settimane in cui abbiamo lezione di giovedì e nelle settimane in cui non abbiamo lezione, il ricevimento è dalle 10 alle 16 di martedì.
  - Tutti gli altri giorni: passate pure, ma non assicuro nulla…
  - Su appuntamento? Forse è meglio…

Alcuni concetti fondamentali

- **Linguaggio macchina**
  - Linguaggio basato su numeri binari utilizzato dai computer per memorizzare ed eseguire programmi
- **Linguaggio assembly**
  - Rappresentazione simbolica del linguaggio macchina, più comprensibile agli essere umani perché utilizza simboli invece di bit per rappresentare istruzioni, registri e dati.
- **Linguaggi ad alto livello**
  - Con i linguaggi ad alto livello, è possibile scrivere programmi con un linguaggio il più simile possibile a quello naturale, astraendosi dai dettagli della macchina.

Esempio: Linguaggio Macchina e Assembly

00100111101010010001111011001011	addiu $29, $29, -32
00100110010011001001100100110010	sw $31, 20($29)
01100100110010011001001100100110	sw $4, 32($29)
00100110010011001001100100110010	sw $5, 36($29)
11111000101001000111001010010100	sw $5, 36($29)
01010101011111111111000001101001	sw $0, 24($29)
10100101010101001111100011000l	sw $0, 28($29)
11010010100100010101001010101011	lw $14, 28($29)
10101010100000111111000011110001	lw $24, 24($29)
10001000111110001010001111010101	multu $14, $14
10010011111011100000111100001111	addiu $8, $14, 1
…	...

Linguggio macchina — Assembly

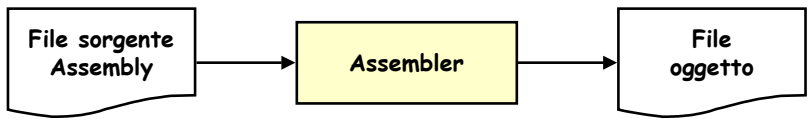
Esempio: Linguaggio C

```
int main(int argc, char *argv[])
{
   int i;
   int sum=0;
   for (int i=0; i <= 100; i++)
       sum += i * i;
   printf("The sum from 0 .. 100 is %d\n", sum);
}
```

Linguaggio C

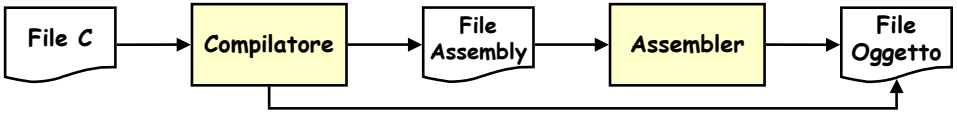
Assembler

- **Assembler**
  - Uno strumento che traduce programmi scritti nel linguaggio assembly in linguaggio macchina. L'assembler
    - legge un *file sorgente*
    - produce un *file oggetto* contenente linguaggio macchina ed altre informazioni necessarie per trasformare uno o più file oggetto in un programma eseguibile



Compilatore

- **Compilatore**
  - Uno strumento che traduce un programma scritto in un linguaggio ad alto livello in un:
    - programma equivalente scritto in linguaggio assembly, che può essere traslato in un file oggetto da un assembler
    - oppure, direttamente in un modulo oggetto



Linguaggio Assembly: Discussione

- **Quando utilizzare il linguaggio assembly:**
  - Quando dimensioni e velocità del programma sono fattori critici
    - Es: processore che gestisce l'ABS in un'auto
  - Per ottimizzare sezione critiche dal punto di vista della performance di un programma
  - Per utilizzare istruzioni particolare del processore altrimenti non utilizzate dai compilatori (ad es., istruzioni MMX)
  - Per sviluppare il nucleo di un sistema operativo, che necessita di utilizzare istruzioni particolari per gestire la protezione della memoria

Linguaggio Assembly: Discussione

- **Problemi nello sviluppo in linguaggio Assembly:**
  - E' complesso
  - E' error-prone
  - E' meno produttivo
  - Il codice prodotto non è portabile
  - Il codice prodotto non è leggibile

Assembler

- **Compito principale di un assembler è quello di semplificare il più possibile la vita del programmatore rispetto all'uso diretto del linguaggio macchina:**
  - Utilizzo di parole mnemoniche per identificare le istruzioni del linguaggio macchina

```
addiu $29, $29, -32
vs
00100111101111011111111111100000
```

  - Aumento del numero di istruzioni disponibili per il programmatore attraverso l'uso di *pseudoistruzioni*
  - Possibilità di definire macro
  - Possibilità di definire etichette
  - Possibilità di aggiungere commenti

File oggetto (o modulo)

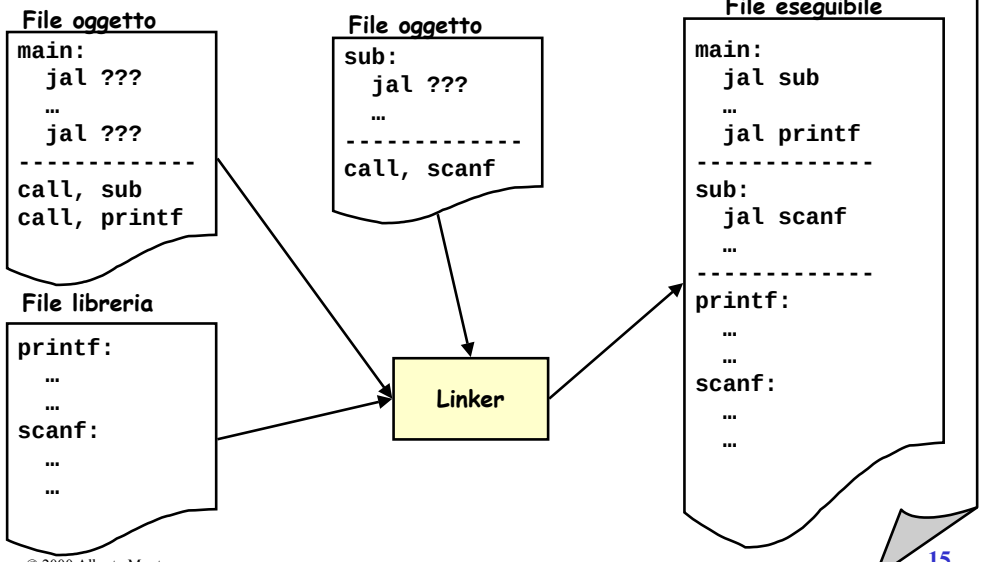
- **Un modulo può contenere**
  - Subroutine (funzioni, procedure, etc.)
  - Dati
  - Riferimenti a subroutine e dati di altri moduli (unresolved references)
- **Esempio:**

```
int main(int argc, char *argv[])
{
   int i;
   int sum=0;
   for (int i=0; i <= 100; i++)
       sum += i * i;
   printf("The sum from 0 .. 100 is %d\n", sum);
}
```

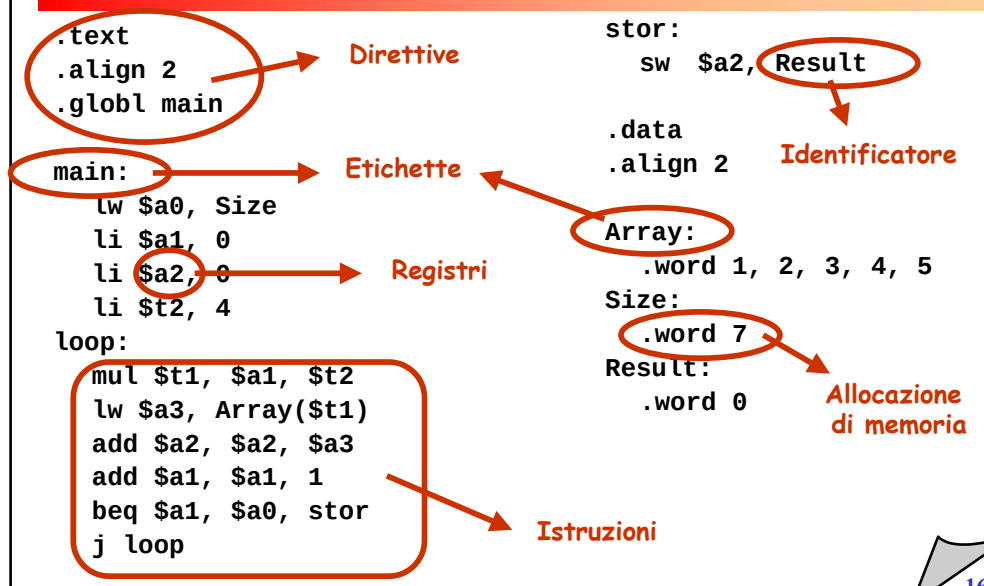

Linker

- **Linker**
  - Uno strumento che combina un insieme di file oggetto e *file libreria* in un *programma eseguibile*
  - Il linker ha tre compiti:
    - Ricercare nei file libreria le routine di libreria utilizzate dal programma (es. printf)
    - Determinare le locazioni di memoria che il codice di ogni modulo andrà ad utilizzare e aggiornare i riferimenti assoluti in modo opportuno
    - Risolvere i riferimenti tra i file diversi
  - Il programma eseguibile non deve contenere unresolved reference

Linker



Un esempio di programma in assembly



Elementi di un programma assembly

- **Direttive**
  - Forniscono informazioni addizionali utili all'assembler per gestire l'organizzazione del codice
- **Esempi:**
  - `.text`
    indica che le linee successive contengono istruzioni
  - `.data`
    indica che le linee successive contengono dati
  - .align n
    indica che gli elementi successivi devono essereallineati rispetto a suddivisioni di 2^n byte

Elementi di un programma assembly (cont.)

- **Identificatori**
  - Un identificatore consiste di una sequenza case-sensitive di caratteri alfanumerici
  - Ad esempio: *`main`*, *`loop`*,. *`stor`*, *`Size`*, *`Array`*, *`Result`*
- **Etichette**
  - Un'etichetta consiste di un identificatore seguito dal simbolo *`:`* (due punti)
  - Le etichette sono utili per identificare particolari sezioni del programma assembly (come punti del programma in cui saltare, dati, etc.)

Elementi di un programma assembly (cont.)

- **Etichette globali**
  - Etichette che possono essere referenziate da file diversi da quello in cui sono definite
- **Etichette locali**
  - Etichette che possono essere referenziate solo all'interno del file in cui sono definite
  - Le etichette sono locali per default
  - L'uso della direttiva *.globl* rende un'etichetta globale
- **Identificatori esterni:**
  - Identificatori che non sono definiti nel file (unresolved reference)

Elementi di un programma assembly (cont.)

- **Istruzioni**
  - Un'istruzione inizia con una parola riservata (keyword) e continua a seconda della sua sintassi
- **Quali sono le istruzioni:**
  - Ad ogni istruzione del linguaggio macchina MIPS corrisponde un'istruzione del linguaggio assembly

    Es: `bne reg1, reg2, address` (branch if not equal)
  - Pseudoistruzioni: istruzioni fornite dall'assembler ma non implementate in hardware
  - Es: `blt reg1, reg2, address` (branch if less than) diventa:

    `slt $at, reg1, reg2` (set less than)
    `bne $at, $zero, address` (branch if not equal)

Elementi di un programma assembly (cont.)

- **Allocazione di memoria nel segmento data:**
  - E' possibile allocare memoria nel segmento data utilizzando alcune direttive che permettono di specificare come la memoria verrà utilizzata e il valore iniziale della memoria stessa
  - Il valore iniziale può essere specificato tramite espressioni, costanti, stringhe. Esempi possibili possono essere:
    - "Hello World"
    - 0xAA+12
    - 10*10

Elementi di un programma assembly (cont.)

- **Direttive di allocazione di memoria:**
  - `.byte` $b_1, \ldots, b_n$
    Memorizza *n* byte consecutivi in byte successivi in memoria
  - `.half h1, …, hn`
    Memorizza *n* quantità a 16 bit in halfword successive in memoria
  - `.word w1, …, wn`
    Memorizza *n* quantità a 32 bit in word successive in memoria
  - `.float f1, …, fn`
    Memorizza *n* valori floating point a singola precisione in locazioni successive in memoria
  - `.double d1, …, dn`
    Memorizza *n* valori floating point a doppia precisione in locazioni successive in memoria
  - `.asciiz str`
    Memorizza la stringa *str* in memoria e terminala con il valore 0
  - `.space n`
    Alloca *n* byte

Elementi di un programma assembly (cont.)

- **Registri generali**

0	$zero	Valore fisso a 0
1	$at	Riservato
2-3	$v0-$v1	Risultati di una funzione
4-7	$a1-$a4	Argomenti di una funzione
8-15	$t0-$t7	Temporanei (non preservati fra chiamate)
16-23	$s0-$s7	Temporanei (preservati fra le chiamate)
24-25	$t8-$t9	Temporanei (non preservati fra chiamate)
26-27	$k0-$k1	Riservate per OS kernel
28	$gp	Pointer to global area
29	$sp	Stack pointer
30	$fp	Frame pointer
31	$ra	Return address

Elementi di un programma assembly (cont.)

- **Registri speciali**
  - PC Program counter
  - HI Risultato di una moltiplicazione, parte più significativa
  - LO Risultato di una moltiplicazione, parte meno significativa
- **Nota**
  - Questi non sono registri generali, quindi non possono essere utilizzati dalle istruzioni normali (come ad esempio le istruzioni di somma)
  - Esistono istruzioni speciali per gestirli
    - Istruzioni "Branch" e "Jump" per il PC
    - Istruzioni `mthi`, `mtlo`, `mfhi`, `mflo` per LO ed HI

SPIM

- **Cos'è SPIM e cosa fa?**
  - SPIM è un simulatore che esegue programmi per le architetture R2000/R3000
  - SPIM può leggere ed assemblare programmi scritti in linguaggio assembly MIPS
  - SPIM contiene inoltre un debugger per poter analizzare il funzionamento dei programmi prodotti
- **Utilizzeremo SPIM per provare i primi semplici esercizi in linguaggio assembly che vedremo**
- **Per esercizi più complessi, utilizzeremo un altro tool chiamato MPS**

Uso di un simulatore

- **Perché utilizzare un simulatore MIPS ?**
  - Ambiente controllato:
    - Controllo degli errori
    - Meccanismi di debug più sofisticati
  - Possibilità di utilizzare un ambiente MIPS in qualunque tipo di ambiente (uniformità):
    - A casa con Windows o Linux su macchine Intel
    - In laboratorio con Linux su macchine Intel e PowerPc
  - L'alternativa più comune (assembly 80X86) è molto complessa e non adatta ad un corso universitario

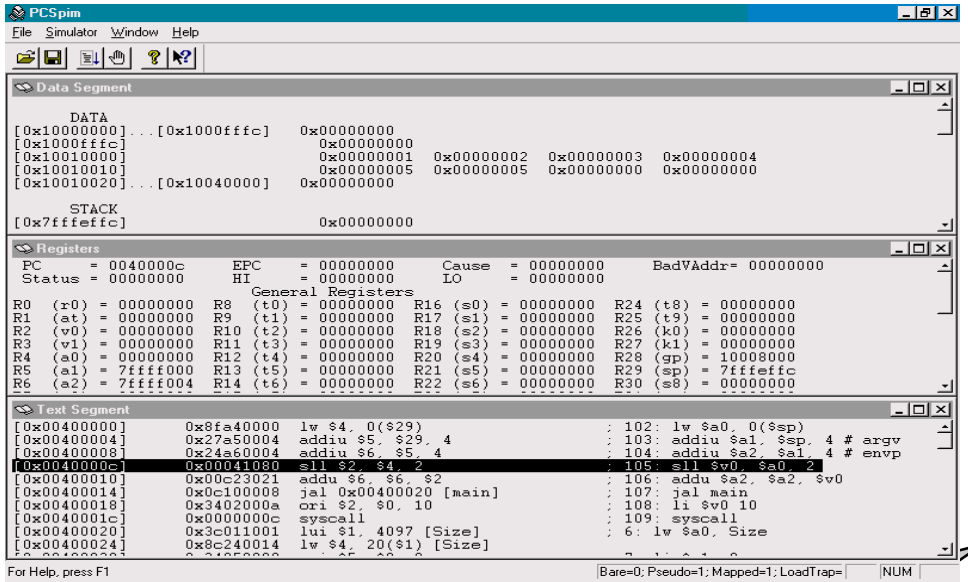
SPIM

- **Dove trovare SPIM?**
  - http://www.cs.wisc.edu/~larus/spim.html
- **Esistono due versioni di SPIM**
  - Linux/Unix (xspim)
  - Windows (winspim)
- **Per installare SPIM nella versione Windows**
  - Scaricate winspim.exe (file di installazione)
  - Fate doppio clic su winspim.exe
  - Seguite le indicazioni
- **Per eseguire SPIM nella versione Windows**
  - Trovate la voce PCSPIM per Windows nel menù programmi

SPIM

- **Per installare SPIM nella versione Unix/Linux**
  - Scaricate **spim.tar.gz**
  - Scompattatelo (**tar zxvf spim.tar.gz**)
  - Spostatevi nella directory **spim-6.3**
  - Digitate **make install** per installare spim e xspim
  - Installazioni più complesse possono essere realizzate seguendo le indicazioni nel file README
- **Per eseguire SPIM nella versione Unix/Linux**
  - Digitate **xspim** (avendo cura di fare in modo che il file xspim sia nel vostro path)

Interfaccia di WINSPIM



Interfaccia di XSPIM

Interfaccia utente: sezioni

- **Sezione registers:**
  - Mostra il valore di tutti i registri della CPU e della FPU MIPS
  - Notare che i registri generali vengono identificati sia dal numero progressivo (**R29**) che dall'identificatore mnemonico (**$sp**)
  - Il valore dei registri viene aggiornato ogni qualvolta il vostro programma viene interrotto
- **Pannello di controllo (in xspim) o menù (in WinSpim)**
  - Contiene i comandi che possono essere utilizzati su SPIM, come ad esempio run, load, etc.

Interfaccia utente: sezioni

- **Segmento text**
  - Mostra le istruzioni dei vostri programmi e del codice di sistema che viene caricato automaticamente alla partenza di SPIM
  - La prossima istruzione da eseguire viene evidenziata invertendo il colore della linea contenente l'istruzione
- **Le istruzioni vengono visualizzate nel seguente modo:**

```
[0x00400020]  0x3c011001  lui $1, 4097 [Size]   ;  6: lw $a0, Size
[0x00400024]  0x8c240014  lw $4, 20($1) [Size]
[0x00400028]  0x34050000  ori $5, $0, 0         ;  7: li $a1, 0
```

① ② ③ ④

Visualizzazione istruzioni

- **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 utente: sezioni

- **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
- **Sezione messaggi SPIM**
  - Utilizzata da SPIM per mostrare messaggi, come ad esempio messaggi di errore
- **Console**
  - Shell in cui vengono visualizzati i messaggi stampati dai vostri programmi

Interfaccia utente: 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 (more details later)
- **Comando STEP (Unix) e Simulator – Step (Windows)**
  - Esegue il programma passo-passo, ovvero una istruzione alla volta
  - Questa modalità permette di studiare al meglio il funzionamento del programma

Istruzioni

- **Le istruzioni del linguaggio assembly MIPS possono essere divise nelle seguenti categorie:**
  - *Istruzioni "Load and Store"*
    Queste istruzioni spostano dati fra la memoria e i registri generali del processore
  - *Istruzioni "Data Movement"*
    Queste istruzioni spostano dati fra i registri del processore
  - *Istruzioni "Load Immediate"*
    Queste istruzioni caricano nei registri valori immediati ("costanti")
  - *Istruzioni aritmetico/logiche*
    Queste istruzioni effettuano operazioni aritmetico-logiche sui registri del processore

Istruzioni

- *Istruzioni di salto*
  Queste istruzioni permettono di spostare l'esecuzione da un punto ad un altro di un programma
- *Istruzioni di confronto*
  Queste istruzioni effettuano il confronto fra i valori contenuti nei registri
- *Istruzioni "Branch"*
  Queste istruzioni permettono di spostare l'esecuzione da un punto ad un altro di un programma in presenza di certe condizioni

- **NOTA:**

  Nel seguito, le pseudoistruzioni sono evidenziate in blu e sottolineate

Istruzioni load and store

`lb rdest, address`	Load byte at address in register rdest
`lbu rdest, address`	Load unsigned byte at address in register rdest
`lh rdest, address`	Load halfword at address in register rdest
`lhu rdest, address`	Load unsigned halfword at address in register rdest
`lw rdest, address`	Load word from address in register rdest
`la rdest, address`	Load computed address in register rdest
`sb rsource, address`	Store lower byte at address from register rsource
`sb rsource, address`	Store lower halfword at address from register rsource
`sb rsource, address`	Store word at address from register rsource

Istruzioni Data Movement and Load Immediate

`move rsource, rdest`	Move register *rsource* into register *rdest*
`mfhi rdest`	Move register *hi* into register *rdest*
`mflo rdest`	Move register *lo* into register *rdest*
`mthi rsource`	Move register *rsource* into register *hi*
`mtlo rsource`	Move register *rsource* into register *lo*
`li rdest, imm`	Move the immediate *imm* into register *rdest*
`lui rdest, imm`	Move the lower halfword of immediate *imm* into the upper halfword of register *rdest*
`la rdest, imm`	Move the lower halfword of immediate *imm* into the upper halfword of register *rdest*

Istruzioni Aritmetico/Logiche

`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 (without overflow)
`negu rd, rs`	rd = - rs (with overflow)
`abs rd, rs`	rd = |rs|

Istruzioni Aritmetico/Logiche

`mult rs, rt`	hi,lo = rs * rt (signed)
`multu rs, rt`	hi,lo = rs * rt (unsigned)
`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)

Istruzioni Aritmetico/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

Istruzioni di confronto

`slt rd, rs, rt`	Set register *rd* to 1 if *rs* < *rt*, 0 otherwise
`sltu rd, rs, rt`	Set register *rd* to 1 if *rs* < *rt*, 0 otherwise
`slti rd, rs, imm`	Set register *rd* to 1 if *rs* < *imm*, 0 otherwise
`sltiu rd, rs, imm`	Set register *rd* to 1 if *rs* < *imm*, 0 otherwise

Istruzioni di Branch

`beq rs, rt, target`	Branch to instruction at *target* if *rs* = *rt*
`bne rs, rt, target`	Branch to instruction at *target* if *rs* != *rt*
`bgez rs, target`	Branch to instruction at *target* if *rs* >= 0
`bgtz rs, target`	Branch to instruction at *target* if *rs* > 0
`blez rs, target`	Branch to instruction at *target* if *rs* <= 0
`bltz rs, target`	Branch to instruction at *target* if *rs* < 0
`beqz rs, target`	Branch to instruction at *target* if *rs* = 0
`bnez rs, target`	Branch to instruction at *target* if *rs* != 0

Istruzioni di Branch

`bge rs, rt, target`	Branch to instruction at *target* if *rs* >= rt
`bgeu rs, rt, target`	Branch to instruction at *target* if *rs* >= rt (unsigned)
`bgt rs, rt, target`	Branch to instruction at *target* if *rs* > rt
`bgtu rs, rt, target`	Branch to instruction at *target* if *rs* > rt (unsigned)
`ble rs, rt, target`	Branch to instruction at *target* if *rs* <= rt
`bleu rs, rt, target`	Branch to instruction at *target* if *rs* <= rt (unsigned)
`blt rs, rt, target`	Branch to instruction at *target* if *rs* < rt
`bltu rs, rt, target`	Branch to instruction at *target* if *rs* < rt (unsigned)

Branch: come funziona esattamente?

- **Nel linguaggio macchina, l'indirizzo dell'istruzione target in un branch viene espresso come *instruction offset* rispetto all'istruzione corrente**
  - E' possibile saltare 2^{15}-1 istruzioni in avanti o 2^{15} istruzioni indietro
- **In linguaggio assembly, l'indirizzo dell'istruzione target è dato dall'etichetta associata**
  - Il calcolo viene fatto automaticamente dall'assembler
  - Ulteriore semplificazione dell'assembler rispetto al linguaggio macchina
- **Esempio:**

```
0x00400030 beq $5, $4, 8 [stor-0x00400030]   beq $a1, $a0, stor
```


Istruzioni di Jump

`j target`	Unconditionally jump to instruction at *target*
`jal target`	Unconditionally jump to instruction at *target*, and save the address of the next instruction in register *$ra*
`jr rsource`	Unconditionally jump to the instruction whose address is in register *rsource*
`jalr rsource, rdest`	Unconditionally jump to the instruction whose address is in register *rsource* and save the address of the next instruction in register *rdest*

Modi di indirizzamento

- **MIPS ha un'architettura load/store:**
  - Solo le istruzioni load e store accedono alla memoria
  - Le istruzioni di computazione lavorano solo con i registri
- **Nel linguaggio macchina, esiste una sola modalità di indirizzamento:**
  - *imm(register)* dove l'indirizzo è dato dalla somma del valore immediato *imm* più il contenuto del registro *register*
- **L'assembler fornisce i seguenti metodi di indirizzamento per le istruzioni load and store**
  - (register) contenuto di *register*
  - imm valore *imm*
  - imm(register) valore *imm* + contenuto di *register*
  - label + imm indirizzo di label + valore *imm*
  - label + imm(register) indirizzo di label + valore di *imm* + contenuto di *register*

Primo esempio di programma

```
.text
.align 2
.globl main
main:
   lw $a0, Size          # read the size of array
   li $a1, 0             # index i
   li $a2, 0             # a2 contains the sum
   li $t2, 4             # t2 contains the constant 4
loop:
   mul $t1, $a1, $t2     # t1 gets i * 4
   lw $a3, Array($t1)    # a3 = N[i]
   add $a2, $a2, $a3     # sum = sum + N[i]
   add $a1, $a1, 1       # i = i + 1
   beq $a1, $a0, stor    # go to stor if finished
   j loop
```


Primo esempio di programma

```
stor:
   sw $a2, Result        # store the sum at Result

.data
.align 2
Array:
   .word 8, 25, -5, 55, 33, 12, -78
Size:
   .word 7
Result:
   .word 0
```


Traduzione da assembly a codice macchina

```
lui $1, 4097 [Size]          11: lw $a0, Size
lw $4, 28($1) [Size]
ori $5, $0, 0                12: li $a1, 0
ori $6, $0, 0                13: li $a2, 0
ori $10, $0, 4               14: li $t2, 4
mult $5, $10                 16: mul $t1, $a1, $t2
mflo $9
lui $1, 4097 [Array]         17: lw $a3, Array($t1)
addu $1, $1, $9
lw $7, 0($1) [Array]
```


Traduzione da assembly a codice macchina

```
add $6, $6, $7                  18: add $a2, $a2, $a3
addi $5, $5, 1                  19: add $a1, $a1, 1
beq $5, $4, 8 [stor-0x00400030] 20: beq $a1, $a0, stor
j 0x00400014 [loop]             21: j loop
lui $1, 4097 [Result]           24: sw a2, Result
sw $6, 32($1) [Result]
```

Esercizi

- **Alcuni esercizi:**
  - Scrivere una routine che calcoli la lunghezza di una stringa che termina con 0 (null-terminated)
  - Scrivere una routine che calcoli il numero di occorrenze di un carattere in una stringa null-terminated
  - Scrivere una routine che sostituisca tutte le occorrenze di un carattere con un altro in una stringa null-terminated
- **Specificare una stringa tramite la direttiva .asciiz**
- **Memorizzare il risultato in una variabile di memoria**