



Lezione 12

Assembly II

Set di Istruzioni MIPS

Strutture di controllo in Assembly

Proff. A. Borghese, F. Pedersini

Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano

Classificazione delle istruzioni



❖ **Duplica classificazione** delle istruzioni macchina

➤ in base a:

1. Categoria funzionale dell'istruzione

- logico-aritmetica
- trasferimento dati (memoria, I/O)
- controllo di flusso (salto)
- ...

2. Formato (tipo) e codifica dell'istruzione

- Tipo e dimensione istruzione
- Posizione operandi e risultato
- Tipo e dimensione dei dati
- Operazioni consentite

Classificazione "ortogonale"
del Set Istruzioni:

		Categoria/funzione		
Formato (tipo)	add ₁	load ₁	...	jmp ₁
	add ₁	load ₁	...	--

	add _N	--	...	jmp _N



Instruction Set Architecture MIPS:

- ❖ Tutte le istruzioni MIPS hanno la stessa dimensione: **32 bit**
 - I 32 bit hanno un significato diverso a seconda del formato (o tipo) di istruzione
- ❖ La categoria di istruzione è riconosciuto in base al valore dei **6 bit più significativi**:
(6 bit: codice operativo - "**OPCODE**")
- ❖ Le istruzioni MIPS sono di **3 tipi** (formati)
 - **Tipo R (register)** Istruzioni aritmetico-logiche
 - **Tipo I (immediate)** Istruzioni di accesso alla memoria o contenenti delle costanti
 - **Tipo J (jump)** Istruzioni di salto



- ❖ **Categorie di istruzioni MIPS**
 - Istruzioni aritmetico-logiche
 - Istruzioni di trasferimento dati
 - Istruzioni di salto



Istruzioni aritmetico-logiche

- ❖ MIPS: un'istruzione aritmetico-logica possiede **tre operandi**:
 - **due registri** contenenti i valori da elaborare (**2 registri sorgente**) oppure **1 registro** (1° operando) ed un **numero** (2° operando)
 - **un registro** che conterrà il risultato (**registro destinazione**)
- ❖ L'ordine degli operandi è fisso
 - Prima il registro destinazione, poi i due registri sorgente, in ordine
- ❖ **Struttura istruzione, in Assembly**:
 - codice operativo e tre campi relativi ai tre operandi:

OPCODE DEST, SORG1, SORG2



Istruzioni: *add, sub, tipo R*

❖ **add**: Addizione

add rd, rs, rt

- somma il contenuto di due registri sorgente **rs** e **rt**
- e mette la somma nel registro destinazione: **rd**

add rd, rs, rt # **rd** ← **rs** + **rt**

❖ **sub**: Sottrazione

sub rd rs rt

- sottrae il contenuto di due registri sorgente **rs** e **rt**
- e mette la differenza nel registro destinazione **rd**

sub rd, rs, rt # **rd** ← **rs** - **rt**



Varianti: *unsigned*, *tipo I*

`addi $s1, $s2, 100` `#add immediate`

`subi $s1, $s2, 100` `#sub immediate`

- Somma/sottrazione di una costante: il valore del secondo operando è presente nell'istruzione come costante (ultimi 16 bit dell'istruzione)

`addu $s0, $s1, $s2` `#add unsigned`

`subu $s0, $s1, $s2` `#sub unsigned`

- L'operazione viene eseguita tra **numeri senza segno**

`addiu $s0, $s1, 100` `#add immediate unsigned`

`subiu $s0, $s1, 100` `#sub immediate unsigned`

- Il secondo operando è una **costante, senza segno**

Moltiplicazione



❖ Assembly MIPS:

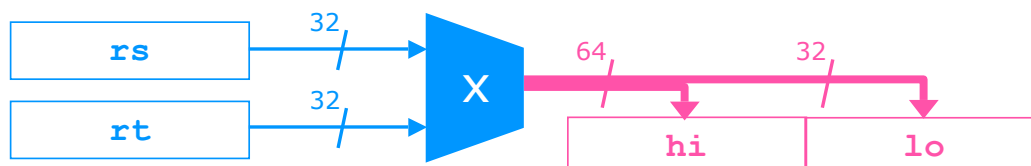
`mult rs rt` `# multiply`

`multu rs rt` `# unsigned multiply`

- La moltiplicazione di due numeri di **32 bit** dà come risultato un numero rappresentabile in **64 bit**
- Il registro destinazione è **implicito**: il risultato viene posto sempre in due registri dedicati: [**hi** , **lo**]

HIGH-order 32-bit word → **hi**

LOW-order 32-bit word → **lo**





- ❖ Il risultato di moltiplicazione / divisione si preleva dal registro **hi** e dal registro **lo** utilizzando le due istruzioni:

```
mfhi rd    # move from hi: rd ← hi
mflo rd    # move from lo: rd ← lo
```

- ❖ Il risultato viene trasferito nel registro destinazione specificato



- ❖ Divisione in MIPS:

```
div rs rt    # division: rs/rt
divu rs rt   # unsigned division
```

- ❖ Come nella moltiplicazione, anche nella divisione il registro destinazione è **implicito**, di **dimensione doppia**: [hi , lo]
 - Necessari 64 bit: quoziente (32 bit) + resto (32 bit)
 - Quoziente (intero) della divisione nel registro **lo**
 - Resto della divisione nel registro **hi**

QUOZIENTE (32-bit) → lo

RESTO (32-bit) → hi



Pseudoistruzioni

- ❖ Per semplificare la programmazione, in ogni Assembly vengono definite alcune **pseudoistruzioni**
 - Significato intuitivo
 - Non hanno un corrispondente 1 a 1 con le istruzioni dell'ISA
- ❖ **Vantaggi:**
 - **Parziale standardizzazione** del linguaggio Assembly
 - ✦ La stessa pseudoistruzione viene tradotta in modi differenti, per architetture (I.S.A.) differenti
 - **Rappresentazione più compatta ed intuitiva** di istruzioni Assembly comunemente utilizzate
 - ✦ La traduzione della pseudoistruzione nelle istruzioni equivalenti viene attuata automaticamente dall' Assembler



Esempi:

Pseudoistruzione:

```
move $t0, $t1  
# t0 ← t1
```

→

Codice MIPS:

```
add $t0, $zero, $t1
```

```
mul $s0, $t1, $t2  
# s0 ← t1*t2
```

→

```
mult $t1, $t2  
mflo $s0
```

```
div $s0, $t1, $t2  
# s0 ← t1/t2
```

→

```
div $t1, $t2  
mflo $s0
```



❖ Categorie di istruzione:

- Istruzioni aritmetico-logiche
- Istruzioni di trasferimento dati
- Istruzioni di salto

Istruzioni di trasferimento dati



❖ MIPS fornisce due operazioni base per il trasferimento dei dati:

- **lw (load word)**
per trasferire una parola di memoria in un registro
- **sw (store word)**
per trasferire il contenuto di un registro in una cella di memoria

lw e **sw** necessitano, come argomenti:

- dell'indirizzo della locazione di memoria su cui operare
- del registro in cui scrivere / da cui leggere il dato



- ❖ L'istruzione di **load** copia la parola contenuta in una specifica locazione di memoria a un registro della **CPU**:

load r1, LOC # r1 ← [LOC]

- La **CPU** invia l'indirizzo della locazione desiderata (**LOC**) alla memoria e richiede un'operazione di lettura del suo contenuto.
 - La memoria effettua la lettura dei dati memorizzati all'indirizzo **LOC** e li invia alla **CPU**
- ❖ L'istruzione di **store** copia la parola da un registro della **CPU** in una specifica locazione di memoria:

store r2, LOC # [LOC] ← r2

- La **CPU** invia l'indirizzo della locazione desiderata alla memoria, assieme con i dati che vi devono essere scritti e richiede un'operazione di scrittura.
- La memoria effettua la scrittura dei dati all'indirizzo specificato.



- ❖ Nel **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 alla costante.
- ❖ L'indirizzo della parola di memoria da caricare nel registro destinazione è ottenuto come **somma della costante e del contenuto del registro base.**
- ❖ **L'indirizzo è espresso in bytes !!!**



lw \$s1, 100 (\$s2) # \$s1 ← M[\$s2+100]

- ❖ Al registro destinazione \$s1 è assegnato il valore contenuto all'indirizzo di memoria:

$(\$s2 + 100)$ (indirizzo di byte)

sw \$s1, 100 (\$s2) # M[\$s2 + 100] ← \$s1

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

Memorizzazione di arrays



- ❖ Arrays di parole di 32 bit (4 byte):
- ❖ L'elemento **i-esimo** **A[i]** si trova all'indirizzo: $br + 4*i$
 - **br** è il registro base; **i** è l'indice ad alto livello
 - Il registro base punta all'inizio dell'array: primo byte del primo elemento
 - Spiazzamento $4i$ per selezionare l'elemento **i** dell'array

Elem. array	parola (word)				Indirizzo	Cella
A[0] :	0	1	2	3	\$s3 →	A[0]
A[1] :	4	5	6	7	\$s3 + 4 →	A[1]
A[2] :	8	9	10	11	\$s3 + 8 →	A[2]

A[k-2] :					\$s3 + 4i →	A[i]
A[k-1] :	2^{k-4}	2^{k-3}	2^{k-2}	2^{k-1}	



Codice C: $A[12] = h + A[8];$

Si suppone che:

- la variabile **h** sia associata al registro **\$s2**
- l'indirizzo del primo elemento dell'array (*base address*) sia contenuto nel registro **\$s3 (A[0])**

Codice MIPS:

```
lw  $t0, 32($s3)    # $t0 ← M[$s3 + 32]
add $t0, $s2, $t0   # $t0 ← $s2 + $t0
sw  $t0, 48($s3)    # M[$s3 + 48] ← $t0
```



❖ Istruzione C: $g = h + A[i]$

❖ Si suppone che:

- le variabili **g**, **h**, **i** siano associate rispettivamente ai registri **\$s1**, **\$s2**, ed **\$s4**
- l'indirizzo del primo elemento dell'array (*base address*) sia contenuto nel registro **\$s3**



- ❖ L'elemento **i-esimo** dell'array, $A[i]$ si trova nella locazione di memoria di indirizzo $(\$s3 + 4 * i)$
- ❖ Caricamento dell'indirizzo di $A[i]$ nel registro temporaneo $\$t1$:

```
mul $t1, $s4, 4      # $t1 ← 4 * i
add $t1, $t1, $s3    # $t1 ← add. of A[i]
                    # that is ($s3 + 4 * i)
```

- ❖ Per trasferire $A[i]$ nel registro temporaneo $\$t0$:

```
lw $t0, 0($t1)      # $t0 ← A[i]
```

- ❖ Per sommare h e $A[i]$ e mettere il risultato in g :

```
add $s1, $s2, $t0   # g = h + A[i]
```



- ❖ Categorie di istruzione:
 - Istruzioni aritmetico-logiche
 - Istruzioni di trasferimento dati
 - Istruzioni di salto



- ❖ Istruzioni di **salto condizionato (branch)**: il salto viene eseguito solo se una certa condizione risulta soddisfatta.

- **beq** (*branch on equal*)
- **bne** (*branch on not equal*)

```
beq r1, r2, L1      # go to L1 if (r1 == r2)
bne r1, r2, L1      # go to L1 if (r1 != r2)
```

- ❖ Istruzioni di **salto incondizionato (jump)**: il salto va sempre eseguito.

- **j** (*jump*)
- **jr** (*jump register*)
- **jal** (*jump and link*)

```
j    L1      # go to L1
jr   r31     # go to add. contained in r31
jal  L1      # go to L1, save address of next
                # instruction in ra
```



- ❖ **Alterano l'ordine di esecuzione delle istruzioni**
 - La prossima istruzione da eseguire non è l'istruzione successiva all'istruzione corrente
- ❖ **Permettono di eseguire cicli e valutare condizioni**
 - In assembly le strutture di controllo sono molto semplici e primitive

Struttura: *if ... then*



- ❖ Codice C:

```
if (i==j) f=g+h;
```

- ❖ Si suppone che le variabili *f*, *g*, *h*, *i* e *j* siano associate rispettivamente ai registri: *\$s0*, *\$s1*, *\$s2*, *\$s3* e *\$s4*

- ❖ La condizione viene trasformata in codice C in:

```
if (i != j) goto Label;
    f=g+h;
Label:
```

- ❖ Codice MIPS:

```
    bne $s3, $s4, Label      # go to Label if i≠j
    add $s0, $s1, $s2       # f=g+h (skipped if i ≠ j)
Label:
```

Struttura: *if... then ... else*



Codice C:

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

- Si suppone che le variabili *f*, *g*, *h*, *i* e *j* siano associate rispettivamente ai registri *\$s0*, *\$s1*, *\$s2*, *\$s3* e *\$s4*

- ❖ Codice MIPS:

```
    bne $s3, $s4, Else      # go to Else if i≠j
    add $s0, $s1, $s2       # f=g+h (skipped if i≠j)
    j   End                # go to End
Else:
    sub $s0, $s1, $s2       # f=g-h
End:
    ...
```

Struttura: **do ... while (repeat)**



❖ Codice C:

```
do
    g = g + A[i];
    i = i + j;
while (i != h)
```

❖ **g** e **h** siano associate a **\$s1** e **\$s2**,
i e **j** associate a **\$s3** e **\$s4**,
\$s5 contenga il *base address* di **A**

❖ **A[i]**: array con indice variabile

⇒ devo **moltiplicare i** per **4** ad ogni iterazione del ciclo per indirizzare il vettore **A**.

Struttura: **do ... while (repeat)**



Codice C modificato (uso **goto**):

```
Ciclo:    g = g + A[i];           // g e h → $s1 e $s2
          i = i + j;           // i e j → $s3 e $s4
          if (i != h) goto Ciclo; // A[0] → $s5
```

Codice MIPS:

```
Loop:    muli $t1, $s3, 4        # $t1 ← 4 * i
          add $t1, $t1, $s5      # $t1 ← add. of A[i]
          lw $t0, 0($t1)        # $t0 ← A[i]
          add $s1, $s1, $t0      # g ← g + A[i]
          add $s3, $s3, $s4      # i ← i + j
          bne $s3, $s2, Loop     # goto Loop if i≠h
```



❖ Codice C:

```
while (A[i] == k)
    i = i + j;
```

❖ Codice C modificato:

```
Ciclo:    if (A[i] != k) goto Fine;
          i = i + j;
          goto Ciclo;

Fine:
```

- *i*, *j* e *k* siano associate a: *\$s3*, *\$s4*, *\$s5*
- *\$s6* contenga il *base address* di *A[]*



❖ Codice C modificato:

```
Ciclo:    if (A[i] != k) goto Fine;
          i = i + j;
          goto Ciclo;

Fine:
```

Associazioni:

```
i, j, k → $s3, $s4, $s5
A[0] → $s6
```

❖ Codice MIPS:

```
Loop: muli $t1, $s3, 4           # $t1 ← 4 * i
      add  $t1, $t1, $s6        # $t1 ← addr. A[i]
      lw   $t0, 0($t1)         # $t0 ← A[i]
      bne $t0, $s5, Exit       # if A[i]≠k goto Exit
      add $s3, $s3, $s4        # i ← i + j
      j   Loop                 # go to Loop

Exit:
```



- ❖ MIPS mette a disposizione **branch** solo nel caso uguale o diverso, non maggiore o minore.
 - Spesso è utile condizionare l'esecuzione di una istruzione al fatto che una variabile sia minore di una altra
- ❖ Istruzione **slt**:
slt \$s1, \$s2, \$s3 **# set if less than**
 - Assegna il valore **\$s1 = 1** se **\$s2 < \$s3**;
altrimenti assegna il valore **\$s1=0**
- ❖ Con **slt**, **beq** e **bne** si possono implementare tutti i test sui valori di due variabili: (**=**, **≠**, **<**, **≤**, **>**, **≥**)

Esempio



Codice C:

```
if (i < j) then
    k = i + j;
else
    k = i - j;
```

Codice C modificato:

```
if (i < j)    t = 1;
else        t = 0;

If (t == 0)
    goto Else;
k = i + j;
goto Exit;
Else:
    k = i - j;
Exit:
```

Codice Assembly:

```
#$s0 ed $s1 contengono i e j
#$s2 contiene k
```

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




Codice C:

```
switch(k)
{
    case 0:    f = i + j; break;
    case 1:    f = g + h; break;
    case 2:    f = g - h; break;
    case 3:    f = i - j; break;
    default:   break;
}
```

Implementabile in due modi:

1. Mediante una serie di:

if-then-else

2. Mediante una *jump address table*

- Tabella che contiene una serie di indirizzi a istruzioni alternative



❖ Codice C alternativo:

```
if (k < 0)
    t = 1;
else
    t = 0;
if (t == 1) // k<0
    goto Exit;
if (k == 0) // k>=0
    goto L0;
k--; if (k == 0) // k=1
    goto L1;
k--; if (k == 0) // k=2
    goto L2;
k--; if (k == 0) // k=3
    goto L3;
goto Exit; // k>3
...
```

```
...
L0:    f = i + j;
        goto Exit;
L1:    f = g + h;
        goto Exit;
L2:    f = g - h;
        goto Exit;
L3:    f = i - j;
        goto Exit;

Exit:
```



```

# $s0, ..., $s5 contengono f,g,h,i,j,k
    slt $t3, $s5, $zero
    bne $t3, $zero, Exit      # if k<0
# case vero e proprio
    beq $s5, $zero, L0
    subi $s5, $s5, 1
    beq $s5, $zero, L1
    subi $s5, $s5, 1
    beq $s5, $zero, L2
    subi $s5, $s5, 1
    beq $s5, $zero, L3
    j Exit;                  # if k>3
L0:   add $s0, $s3, $s4
      j Exit
L1:   add $s0, $s1, $s2
      j Exit
L2:   sub $s0, $s1, $s2
      j Exit
L3:   sub $s0, $s3, $s4
Exit:
    
```

Jump address table



- ❖ **Jump address table:** utilizzo il valore della variabile-switch (k) per calcolare l'indirizzo di salto:

k	Byte address: $\$t4 + 4 \cdot k$	$A[k]$
0	$\$t4 \rightarrow$	indirizzo di L0 :
1	$\$t4 + 4 \rightarrow$	indirizzo di L1 :
2	$\$t4 + 8 \rightarrow$	indirizzo di L2 :
3	$\$t4 + 12 \rightarrow$	indirizzo di L3 :

Struttura switch/case



```
# $s0, ..., $s5 contengono f,g,h,i,j,k
# $t4 contiene lo start address della
# jump address table (che si suppone parta da k=0)

# verifica prima i limiti (default)
    slt $t3, $s5, $zero
    bne $t3, $zero, Exit
    slti $t3, $s5, 4
    beq $t3, $zero, Exit
#case vero e proprio
    muli $t1, $s5, 4
    add $t1, $t4, $t1
    lw $t0, 0($t1)
    jr $t0 # jump to A[k]
L0: add $s0, $s3, $s4
    j Exit
L1: add $s0, $s1, $s2
    j Exit
L2: sub $s0, $s1, $s2
    j Exit
L3: sub $s0, $s3, $s4
Exit:
```