

Assembly - parte II

Architettura
degli Elaboratori e delle Reti,
Turno I



Alberto Borghese
Università degli Studi di Milano
Dipartimento di Scienze dell'Informazione
borgnese@dsi.unimi.it

1

I tipi di istruzione

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

Istruzioni di trasferimento dati

- Gli operandi di una istruzione aritmetica devono risiedere nei registri
- I registri MIPS sono 32
- Cosa succede ai programmi i cui dati richiedono più di 32 registri (32 variabili)?
(Register Spilling)

Alcuni dati risiedono in memoria



Servono istruzioni apposite per trasferire dati da memoria a registri e viceversa

Istruzioni di trasferimento dati

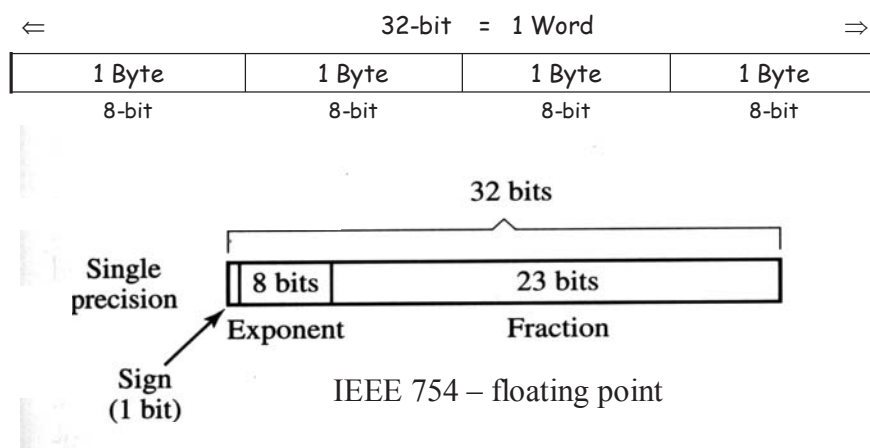
- MIPS fornisce due operazioni base per il trasferimento dei dati:
 - ◆ **lw** (load word) per trasferire una parola di memoria in un registro
 - ◆ **sw** (store word) per trasferire il contenuto di un registro in una parola di memoria

lw e sw richiedono come argomento l'indirizzo della locazione di memoria sulla quale devono operare

La memoria nel MIPS

- I contenuti delle locazioni di memoria possono rappresentare sia istruzioni che dati.
- La memoria è vista come un unico grande array uni-dimensionale
- Un **indirizzo di memoria** costituisce un indice all'interno dell'array
- MIPS utilizza un **indirizzamento al byte**, cioè l'indice punta ad un byte di memoria
 - ◆ byte consecutivi hanno indirizzi consecutivi
 - ◆ indirizzi di parole consecutive (adiacenti) differiscono di un fattore 4 (8-bit x 4 = 32-bit).

Indirizzamento dei byte all'interno della parola



Indirizzamento dei byte all'interno della parola

- In genere, la più piccola unità di memoria indirizzabile è il *byte*.
- La lunghezza di parola è 32-bit \Rightarrow in una singola parola possono essere disposti 4 byte. In genere, l'indirizzo delle parole viene specificato indicando l'indirizzo del loro *primo byte*.
- Il primo byte all'interno della parola può essere indirizzato secondo due modalità:
 - ◆ Disposizione *big-endian* (MIPS, PowerPC, 68000)
 - ◆ Disposizione *little-endian* (Intel)

Disposizione big-endian

	Indirizzo di byte			
Parola 0	0	1	2	3
Parola 4	4	5	6	7
Parola 8	8	9	10	11
...				
Parola 2^k-4	2^k-4	2^k-3	2^k-2	2^k-1

- Nella disposizione *big-endian* i byte sono numerati partendo dalla posizione **più** significativa; alla parola viene assegnato lo stesso indirizzo del suo byte più significativo.

Disposizione little-endian

	Indirizzo di byte			
Parola 0	3	2	1	0
Parola 4	7	6	5	4
Parola 8	11	10	9	8
...				
Parola 2^k-4	2^k-1	2^k-2	2^k-3	2^k-4

Sign bit
↙

- Nella disposizione *little-endian* i byte sono numerati partendo dalla posizione **meno** significativa; alla parola viene assegnato lo stesso indirizzo del suo byte meno significativo.

Indirizzamento della memoria

- Gli indirizzi di parole adiacenti in memoria differiscono per un fattore quattro (8-bit \times 4=32-bit)
- In MIPS ogni parola (word) deve iniziare ad un indirizzo multiplo di 4
 - ◆ Half word (16-bit) allineate ai multipli di 2
- Per convenzione l'indirizzo di una parola coincide con l'indirizzo del suo byte *più a sinistra* (*disposizione big-endian*)

Indirizzamento della memoria

Indirizzo di byte

0	32 bit	0	1	2	3
4	32 bit	4	5	6	7
8	32 bit	8	9	10	11
12	32 bit				
		2^{k-4}	2^{k-3}	2^{k-2}	2^{k-1}

© N.A. Borghese, C. Silvano and E. Rosti – Università di Milano 27/04/2002

11/48

Organizzazione logica della memoria

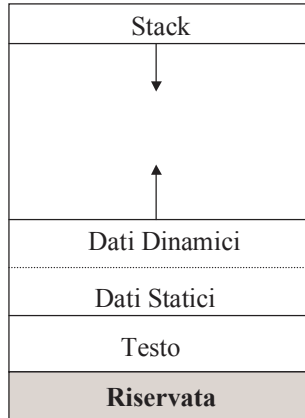
Nei sistemi basati su processore MIPS (e Intel) la memoria è solitamente divisa in **tre** parti:

- ◆ **Segmento testo:** contiene le **istruzioni** del programma
- ◆ **Segmento dati:** ulteriormente suddiviso in:
 - **dati statici:** contiene dati la cui dimensione è conosciuta al momento della compilazione e il cui intervallo di vita coincide con l'esecuzione del programma
 - **dati dinamici:** contiene dati ai quali lo spazio è allocato dinamicamente al momento dell'esecuzione del programma su richiesta del programma stesso.
- ◆ **Segmento stack:** contiene lo stack allocato automaticamente da un programma durante l'esecuzione.

© N.A. Borghese, C. Silvano and E. Rosti – Università di Milano 27/04/2002

12/48

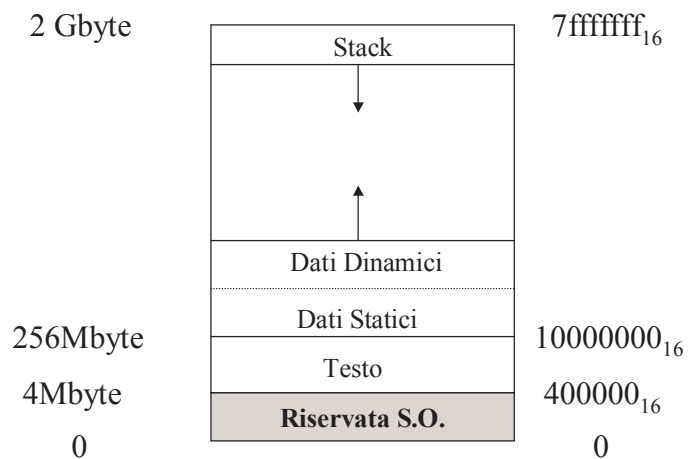
Organizzazione logica della memoria



© N.A. Borghese, C. Silvano and E. Rosti – Università di Milano 27/04/2002

13/48

Organizzazione logica della memoria



© N.A. Borghese, C. Silvano and E. Rosti – Università di Milano 27/04/2002

14/48

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 della CPU
 - ◆ **sw (store word)** per trasferire il contenuto di un registro della CPU in una parola di memoria

lw e sw richiedono come argomento l'indirizzo della locazione di memoria sulla quale devono operare

Istruzione *load*

- L'istruzione di *load* trasferisce una copia dei dati/istruzioni contenuti in una specifica locazione di memoria ai registri della CPU, lasciando inalterata la parola di memoria:
`load LOC, r1 # r1 ← [LOC]`
- La CPU 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 alla CPU.

Istruzione di *store*

- L'istruzione di *store* trasferisce una parola di informazione dai registri della *CPU* in una specifica locazione di memoria, sovrascrivendo il contenuto precedente di quella locazione:

```
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.

Istruzione *lw*

- 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 dalla somma della costante e del contenuto del registro base.

Memorizzazione di un vettore

Indirizzo di byte

A[0]	0	1	2	3
	4	5	6	7
Offset (A[2])	8	9	10	11
	2^{k-4}	2^{k-3}	2^{k-2}	2^{k-1}

© N.A. Borghese, C. Silvano and E. Rosti – Università di Milano 27/04/2002

19/48

Istruzione lw: trasferimento da memoria a registro

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



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

© N.A. Borghese, C. Silvano and E. Rosti – Università di Milano 27/04/2002

20/48

Istruzione sw: trasferimento da registro a memoria

- Possiede argomenti analoghi alla lw

Esempio:

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

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

lw & sw: esempio di compilazione

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
```

Array

s3	A[0]
s3 + 4	A[1]
s3 + 8	A[2]

Array

- L'elemento numero ***i*-esimo** di un array si troverà nella locazione $br + 4 * i$ dove:
 - ◆ br è il registro base;
 - ◆ i è l'indice ad alto livello;
 - ◆ il fattore 4 dipende dall'indirizzamento al byte della memoria nel MIPS

Array: esempio

- Sia A un array di N word
- 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 $\$s3$

Array: esempio indirizzamento

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

```
muli $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]
```

Register Spilling

- I programmi in genere possiedono più variabili dei registri della CPU.
- 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.
- La tecnica di mettere le variabili meno usate (o usate successivamente) in memoria viene chiamata ***Register Spilling***.

Istruzioni aritmetiche vs. load/store

- Le istruzioni aritmetiche leggono il contenuto di due registri (operandi), eseguono una computazione e scrivono il risultato in un terzo registro (destinazione o risultato)
- Le operazioni di trasferimento dati leggono e scrivono un solo operando senza effettuare nessuna computazione

I tipi di istruzione

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

Registro \$zero

- Spesso la verifica di uguaglianza richiede il confronto con il valore 0 \Rightarrow per rendere più veloce il confronto, in MIPS il registro \$zero contiene il valore 0 e non può mai essere utilizzato per contenere altri dati.

Le strutture di controllo

- Queste istruzioni:
 - ◆ 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 condizioni
- In assembly le strutture di controllo sono molto semplici e primitive

Istruzioni di salto condizionato e incondizionato

- Istruzioni di salto: viene caricato un nuovo indirizzo nel contatore di programma (PC) invece dell'indirizzo seguente l'indirizzo di salto secondo l'ordine sequenziale delle istruzioni.
- Istruzioni di **salto condizionato (conditional branch)**: il salto viene eseguito solo se una certa condizione risulta soddisfatta.
- Esempi: `beq` (*branch on equal*) e `bne` (*branch on not equal*)
- Istruzioni di **salto incondizionato (unconditional jump)**: il salto viene sempre eseguito.
Esempi: `j` (*jump*) e `jr` (*jump register*) e `jal` (*jump and link*)

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

```
j    L1             # go to L1
jr   r31            # go to add. contained in r31
jal  L1             # go to L1. Save add. of next
                        # instruction in reg. ra (ad
                        # esempio return address).
```


Esempio 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 Etichetta;`

`f=g+h;`

Etichetta:

Codice MIPS:

```
bne $s3, $s4, Etichetta      # go to Lab1 if i≠j
add $s0, $s1, $s2           # f=g+h (skipped if i ≠ j)
Etichetta:
```

Esempio 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 (skipped if i = j)
End:
```

Esempio: do ... while (repeate)

Codice C:

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

- Si suppone che **g** e **h** siano associate a **\$s1** e **\$s2**, **i** e **j** associate a **\$s3** e **\$s4** e che **\$s5** contenga il *base address* di **A**.
- Si noti che il corpo del ciclo modifica la variabile **i**
⇒ devo moltiplicare **i** per **4** ad ogni iterazione del ciclo per indirizzare il vettore **A**.

Esempio: do ... while

Codice C modificato:

```

i = 0;
Ciclo: g = g + A[i];
       i = i + j;
       if (i != h) goto Ciclo;
```

g e h → **\$s1** e **\$s2**
i e j → **\$s3** e **\$s4**
A[0] → **\$s5**

Codice MIPS:

```
add $s3, $zero, $zero
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  # go to Loop if i ≠ h
```

Esempio: while

Codice C:

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

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

▪ Si suppone che i , j e k siano associate a $\$s3$, $\$s4$, e $\$s5$ e che $\$s6$ contenga il *base address* di A

Esempio: while

Codice C:

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

i, j e $k \rightarrow \$s3, \$s4, \text{ e } \$s5$
 $A[0] \rightarrow \$s6$

Codice MIPS:

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

Strutture di controllo

- Cosa posso fare se il contenuto di un registro è minore o maggiore del contenuto di un altro?
- 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:
 - ♦ `slt $s1, $s2, $s3` # set on less than
 - ↳ Assegna il valore 1 a \$s1 se \$s2 < \$s3; altrimenti assegna il valore 0
- Con `slt`, `beq` e `bne` si possono implementare tutti i test sui valori di due variabili (=, !=, <, <=, >, >=)

Esempio

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

```
if (i < j)
  t = 1;
If (t == 0) goto Else;
k = i + j;
goto Exit;
Else: k = i - j;
Exit:
```



```
#$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:
```

Struttura switch/case

- Può essere implementata mediante una serie di *if-then-else*
- Alternativa: uso di una *jump address table* cioè di una tabella che contiene una serie di indirizzi di istruzioni alternative

Struttura switch/case

```
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;  
}
```

Struttura switch/case

```
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:

© N.A. Borghese, C. Silvano and E. Rosti – Università di Milano 27/04/2002

43/48

Struttura switch/case

```
#$s0, ..., $s5 contengono f,...,k
#$t2 contiene la costante 4

    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:

© N.A. Borghese, C. Silvano and E. Rosti – Università di Milano 27/04/2002

44/48

Jump address table

Byte address	
t4 + 12	L3
t4 + 8	L2
t4 + 4	L1
t4	L0

Struttura switch/case

```
#$s0, .., $s5 contengono f,..,k
#$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          # j 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:
```

Le procedure

Registri coinvolti:

\$a0, \$a1, \$a2, \$a3 - per il passaggio dei parametri per argomento.

\$v0, \$v1 - registri per il ritorno di valori calcolati.

\$ra - registro di ritorno. Punta a PC + 4.

Esempio di procedura

```
Swap (int v[ ], int k);  
{  
  int temp;  
  temp = v[k];  
  v[k] = v[k+1];  
  v[k+1] = temp;  
}
```



```
swap:  
muli $a2, $a0, 4  
add $t0, $a1, $a2  
lw $t1, 0($t0)  
lw $t2, 4($t0)  
sw $t2, 0($t0)  
sw $t1, 4($t0)  
jr $ra
```

```
$a0 -> k  
$a1 -> v[0]
```