Calcolatori elettronici

Utilizzo del linguaggio Assembly MIPS

Giuseppe Martini

Indice

Capitolo 1 - Introduzione

Le **istruzioni** sono dei comandi nella "lingua del computer"
- **linguaggio Assembly**: formato di istruzioni comprensibili all'uomo
- **linguaggio macchina**: formato di istruzioni comprensibili dal computer (0 e 1)

1.1 Principi base

Principi base della progettazione dell'architettura MIPS:
1. la semplicità favorisce la regolarità
2. bisogna cercare di rendere il caso più comune il caso più veloce
3. tutto ciò che è piccolo è più veloce
4. un buon progetto richiede un buon compromesso

Principio base - Primo

"*La semplicità favorisce la regolarità.*"
- formato delle istruzioni consistente
- stesso numero di operandi (due sorgenti ed uno destinazione)
- più facile da decodificare e gestire in hardware

Principio base - Secondo

"*Bisogna cercare di rendere il caso più comune il caso più veloce.*"

Lo scopo del MIPS è di avere una CPU di tipo RISC[1] (in contrapposizione alle altre architetture, di tipo CISC[2]) con un numero ridotto di istruzioni (solo quelle più comuni) permettendo di eseguire più velocemente i casi più frequenti.

Principio base - Terzo

"*Tutto ciò che è piccolo è più veloce.*"

I RISC, poiché risparmiano spazio e unità di controllo, tendono a dedicare lo spazio disponibile per aumentare il numero dei registri (rispetto ai processori di tipo CISC).
MIPS ha un numero limitato di registri.

Nota bene: Se avesse molti registri sarebbe più complicato decodificare il numero dei registri.

Principio base - Quarto

"*Più facile da decodificare e gestire in hardware.*"

Esistono certe istruzioni che utilizzano 2 oppure 3 registri (es. `lw,sw` ne usano 2, `add,sub` ne usano 3).
Purtroppo non posso avere una regolarità totale; quindi, cerco di trovare il miglior compromesso.

Per garantire i principi 1 e 3, cerco di avere il minor numero di formati possibili.

[1] RISC: Reduced Instruction Set Computer
[2] CISC: Complex Instruction Set Computers

1.2 Istruzioni

Qui di seguito sono presenti alcuni esempi di utilizzo delle istruzioni MIPS.

1.2.1 Addizione e sottrazione algebrica

La `add` (*somma*) e la `sub` (*differenza*) sono mnemonici [3] che mi specificano il tipo di operazione da eseguire.

Addizione		**Sottrazione**	
Codice C	**Codice MIPS assembly**	**Codice C**	**Codice MIPS assembly**
`a = b + c`	`add a, b, c`	`a = b - c`	`sub a, b, c`

Operandi:
- `a`: operando destinazione (dove scrive il risultato della somma)
- `b,c`: operandi sorgenti (gli addendi da sommare)

1.2.2 Istruzioni multiple

Se ho da gestire più variabili devo utilizzare una variabile temporanea.

Codice C	**Codice MIPS assembly**
`a = b + c - d;`	`add t, b, c  # t = b + c` `sub a, t, d  # a = t - d`

Nota bene: il carattere '#' serve per iniziare un commento.

1.2.3 Istruzioni con registri

I registri tornano molto utili nelle istruzioni.

Codice C	**Codice MIPS assembly**
`a = b + c`	`# $s0 = a, $s1 = b, $s2 = c` `add $s0, $s1, $s2`

1.2.4 Istruzioni con costanti

La `addi` (add immediate) è un mnemonico che mi permette di sommare una costante.
Si dedicano **16 bit** per la rappresentazione delle costanti e si sommano in **complemento a due**.

Codice C	**Codice MIPS assembly**

`a = a + 4;`
`b = a - 12;`
	`# $s0 = a, $s1 = b`
	`addi $s0, $s0, 4`
	`addi $s1, $s0, -12`

Nota bene: non è necessario definire nell'insieme di istruzioni la `subi` in quanto basta aggiungere un segno negativo nella `addi` (risparmio un'istruzione).

[3] Mnemonico: il nome di un istruzione.

1.3 Operandi

Gli operandi possono essere:
- registri
- memorie
- costanti (alternativamente chiamati "**valori immediati**")

1.3.1 Registri

In MIPS si hanno 32 registri organizzati su 32 bit (quindi è un'architettura da 32 bit).
Quindi, si definisce il parallelismo di una CPU come il numero di bit dei **registri interni**.
Questa è la dimensione massima dei dati (in termini di bit/byte) che possono essere trasferiti da memoria a CPU e viceversa.

Non e' dunque legato al numero di registri, ma al numero di bit dei registri interni (nel MIPS questi due numeri sono sempre 32).

Nella tabella seguente (Tabella 1.1) sono elencati i registri utilizzati da MIPS.

Nome	Numero di registro	Utilizzo
`$0`	0	valore costante 0
`$at`	1	assembler temporary
`$v0-$v1`	2 - 3	valori di ritorno
`$a0-$a3`	4 - 7	argomenti
`$t0-$t7`	8 - 15	temporanei
`$s0-$s7`	16 - 23	variabili salvate
`$t8-$t9`	24 - 25	"molto" temporanei
`$k0-$k1`	26 - 27	temporanei per il SO
`$gp`	28	puntatore globale
`$sp`	29	puntatore dello stack
`$fp`	30	puntatore della finestra
`$ra`	31	indirizzo di ritorno

Tabella 1.1: Insieme dei registri MIPS

Nota bene: i registri sono più veloci della memoria.

1.3.2 Memorie

Le memorie sono necessarie in quanto non sono sufficienti 32 registri per memorizzare i dati.

La memoria ha una dimensione maggiore rispetto ai registri ma lavora con velocità ridotta.

Una buona prassi è quella di mantenere nei registri le variabili utilizzate più frequentemente e limitarsi ad usare la memoria il meno possibile.

Organizzazione della memoria

La memoria ha degli indirizzi che fanno riferimento ad un singolo byte. Tuttavia, in MIPS, i dati hanno un'organizzazione su 32 bit, quindi sono organizzati in word (indirizzi distanti di 4 byte).
Si può indirizzare la memoria associata a MIPS come singoli byte.

Indirizzi	**Dati**	
⋮	⋮	⋮
0000000C	4 0 F 3 0 7 8 8	Word 3
00000008	0 1 E E 2 8 4 2	Word 2
00000004	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

Lettura della memoria

Word

Nome	Valore
Istruzione (mnemonico)	`lw (load word)`
Formato	`lw <valore salvato>, <offset>(<indirizzo base>)`

Byte

Nome	Valore
Istruzione (mnemonico)	`lb (load byte)`

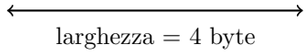
Per calcolare l'indirizzo in memoria devo fare la somma tra l'indirizzo base, che è contenuto in un registro, e un offset che è una costante.

Esempio
Caricare una word di dati nell'indirizzo di memoria 4 dentro `$s3` (`$s3` contiene il valore `0xF2F1AC07` dopo il caricamento).

Codice MIPS assembly

```
lw $s3, 4($0)   # legge la word all'indirizzo 4 dentro $s3
```

Indirizzi	**Dati**				
⋮	⋮				⋮
0000000C	4 0	F 3	0 7	8 8	Word 3
00000008	0 1	E E	2 8	4 2	Word 2
00000004	F 2	F 1	A C	0 7	Word 1
00000000	A B	C D	E F	7 8	Word 0



larghezza = 4 byte

Scrittura della memoria

Word

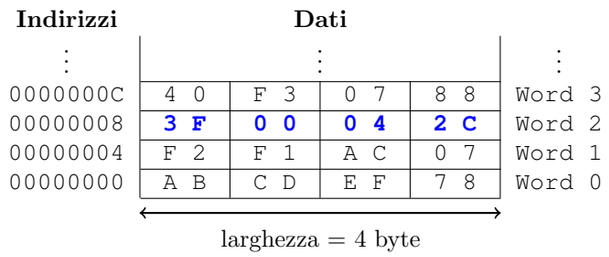
Nome	Valore
Istruzione (mnemonico)	`sw (store word)`
Formato	`sw <valore caricato>, <offset>(<indirizzo base>)`

Byte

Nome	Valore
Istruzione (mnemonico)	`sb (store byte)`

Esempio
Scrivere il valore di `$t4` nell'indirizzo di memoria 8.

Codice MIPS assembly
```
sw $t4, 0x8($0)    # scrive il valore in $t4 all'indirizzo di memoria 8
```



Nota bene: l'offset può essere scritto in decimale o in esadecimale.

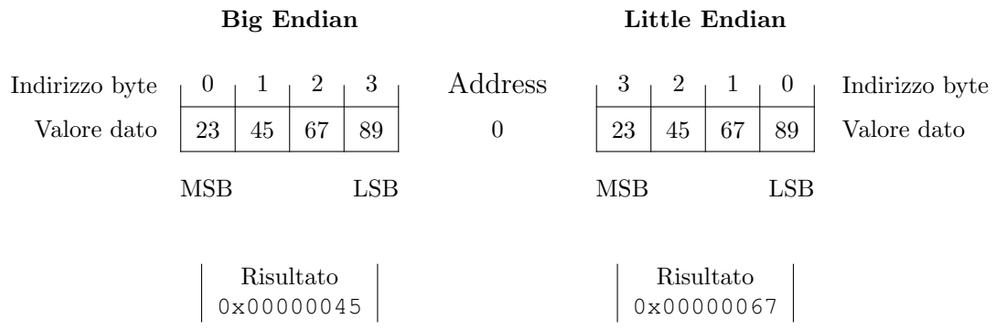
Little e Big Endian

Ci sono due tecniche per salvare una `word` in memoria:
- little endian: inizia dal byte meno significativo
- big endian: inizia dal byte più significativo

Esempio
Si supponga che il registro `$t0` inizialmente contenga il valore `0x23456789`.

Se il codice viene eseguito su un sistema big endian, qual'è il valore di `$s0`?
In un sistema little endian?

```
sw $t0, 0($0)
lb $s0, 1($0)
```



Allineamento dei dati

Nella lettura e scrittura di una word, gli indirizzi devono essere multipli di 4.

Per esempio, questa istruzione

```
lw $s0, 7($0)
```

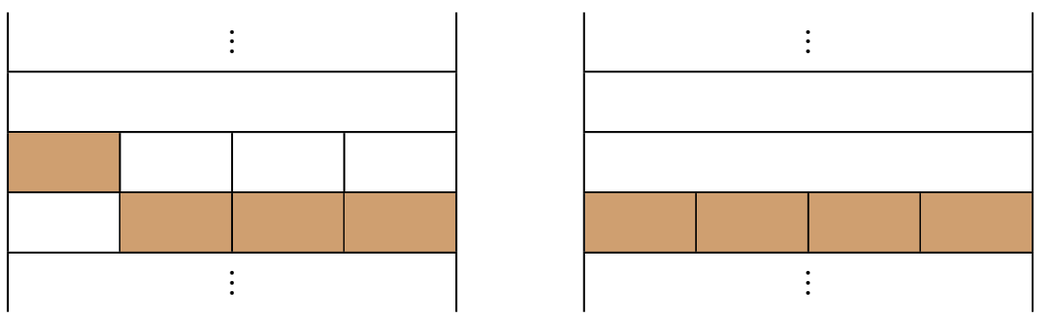
non ha senso, in quanto andrebbe a leggere una word all'indirizzo 7 (non è multiplo di 4).

Si può scegliere la posizione del mio codice/dati utilizzando la direttiva (pseudoistruzione) `.align` altrimenti il codice è allineato in automatico dal compilatore.

Nota bene: la dimensione di un'istruzione è di 32 bit.

Dato	Dimensione (byte)	Da salvare nell'indirizzo
byte	$1 = 2^0$	Qualunque
1/2 word	$2 = 2^1$	Multiplo di 2
word	$4 = 2^2$	Multiplo di 4
double	$8 = 2^3$	Multiplo di 8

Esempio
Esempio di allineamento di un tipo di dato `word` in memoria.



Allineamento errato **Allineamento corretto**

1.3.3 Costanti

I valori immediati (o costanti) hanno l'esigenza di essere rappresentati in questi due casi:
- definizione di un eventuale offset
- inserimento di una costante all'interno di un'espressione numerica

Nota bene: le costanti sono rappresentate da una codifica a 16 bit (in complemento a due, per numeri positivi e negativi).

Capitolo 2 - Formato delle istruzioni

Rappresentazione binaria delle istruzioni (i computer comprendono solo sequenze di 0 e 1).
Nota bene: tutte le istruzioni hanno una lunghezza pari a 32 bit.

Esistono 3 possibili formati per le istruzioni:
- tipo `R`: istruzioni che lavorano con i **registri**
- tipo `I`: istruzioni che lavorano con le **costanti**
- tipo `J`: istruzioni che lavorano con i **salti** (verrà discusso più avanti)

2.1 Tipo R

"*Istruzioni che lavorano con i registri*" (3 registri, `rs`, `rt`, `rd`).

Tipo R

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

`op` (`opcode`)	Codice operativo dell'istruzione (**0 per le istruzioni di tipo R**)	`shamt`	Operazioni di shift che indica di quanti bit shiftare la `word` (0 se non si vuole shiftare)
`rs`	Registro sorgente		
`rt`	——— " ———	`funct`	Discrimina il tipo di istruzione di tipo R (al massimo 64 (2^6) possibili istruzioni)
`rd`	Registro destinazione		

Esempio

```
add $s0, $s1, $s2        add rd, rs, rt
sub $t0, $t3, $t5        sub rd, rs, rt
```

Valori dei campi

op	rs	rt	rd	shamt	funct
0	17	18	16	0	32
0	11	13	8	0	34
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Codice macchina

op	rs	rt	rd	shamt	funct	
000000	10001	10010	10000	00000	100000	`(0x02328020)`
000000	01011	01101	01000	00000	100010	`(0x016D4022)`
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

2.2 Tipo I

"*Istruzioni che lavorano con le costanti*" (2 registri, `rs`, `rt` e un "immediato" `imm`).

`op` (`opcode`)	Codice operativo dell'istruzione (**0 per le istruzioni di tipo R**)	`rt`	Registro sorgente
`rs`	Registro sorgente	`imm`	Valore immediato da 16 bit con complemento a due

Tipo I

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Esempio

```
addi $s0, $s1,5          addi rt, rs, imm
addi $t0, $s3,-12        addi rt, rs, imm

lw $t2,32($0)            lw rt, imm(rs)
sw $s1,4($t1)            sw rt, imm(rs)
```

Valori dei campi

op	rs	rt	imm
8	17	16	5
8	19	8	-12
35	0	10	32
43	9	17	4
6 bits	5 bits	5 bits	16 bits

⇓

Codice macchina

op	rs	rt	imm	
001000	10001	10000	0000 0000 0000 0101	`(0x22300005)`
001000	10011	01000	1111 1111 1111 0100	`(0x2268FFF4)`
100011	00000	01010	0000 0000 0010 0000	`(0x8C0A0020)`
101011	01001	10001	0000 0000 0000 0100	`(0xAD310004)`
6 bits	5 bits	5 bits	16 bits	

Estensione del segno

Le operazioni sono fatti su dati di tipo omogeneo cioè con lo stesso formato. Quindi, essendo che i registri sono a 32 bit e gli immediati sono a 16 bit, bisogna effettuare una conversione a 32 bit. Per fare questo si utilizza una determinata operazione chiamata estensione del segno.

L'estensione del segno consiste nel vedere il segno corrispondente al bit più significativo e replicarlo per i 16 bit di peso maggiore (32 bit = bit estensione del segno (16) + bit valore immediato (16)).

Nota bene: quest'operazione è utilizzata in quanto non si altera il valore del numero.

Esempi

1. Il numero
   `1100 0000 0000 0000` (16 bit)
   diventa
   `1111 1111 1111 1111` `1100 0000 0000 0000` (32 bit)
2. I numeri
   `111,11111,1111111`
   valgono sempre
   `-1`
3. I numeri
   `10,110,1110,1111111110`
   valgono sempre
   `-2`

2.3 Tipo J

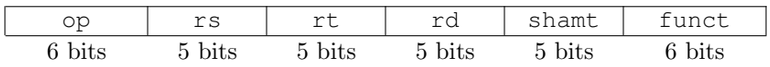
"*Istruzioni che lavorano con i salti*" (un indirizzo `addr`).

`op` (`opcode`)	Codice operativo dell'istruzione (**0 per le istruzioni di tipo R**)	`addr`	Indica l'indirizzo a cui saltare

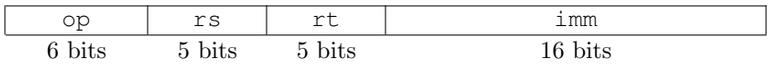
Tipo J

Confronto tipologie di formati

Tipo R



Tipo I



Tipo J

Esercizi

1. Qual'è il codice macchina corrispondente alla seguente istruzione assembly? (Tipo R)
   `add $t0, $t1, $t2`
2. Qual'è il codice macchina corrispondente alla seguente istruzione assembly? (Tipo I)
   `addi $t0, $t1, 0x1234`
3. Qual'è l'istruzione assembly corrispondente al seguente codice macchina?
   `1000 1100 0000 1000 0000 0000 0001 0100`

Capitolo 3 - Caricamento dei programmi

La CPU va a leggere le istruzioni dalla memoria facendo **l'operazione di fetch**. Conseguentemente, il processore andrà ad eseguire l'operazione specificata dall'istruzione che viene letta (questo a partire da un certo indirizzo di memoria).
L'indirizzo di memoria è specificato da un registro interno chiamato **Program Counter** (PC); questo contiene l'indirizzo della prima istruzione del programma.

Codice assembly	**Codice macchina**
`lw $t2, 32($0)`	`0x8C0A0020`
`add $s0, $s1, $s2`	`0x02328020`
`addi $t0, $s3, -12`	`0x2268FFF4`
`sub $t0, $t3, $t5`	`0x016D4022`

Programma caricato

Indirizzi	**Istruzioni**	
⋮	⋮	
0040000C	0 1 6 D 4 0 2 2	
00400008	2 2 6 8 F F F 4	
00400004	0 2 3 2 8 0 2 0	
00400000	8 C 0 A 0 0 2 0	← PC
⋮	⋮	

Nei programmi MIPS, le istruzioni, normalmente, vengono memorizzate partendo dall'indirizzo `0x00400000`.

Un'istruzione che viene caricata attraverso la fetch viene scritta nell'**Istruction Register** (IR).
Successivamente, la CPU decodifica l'istruzione:

1. legge l'opcode (il campo "`OP`" dell'istruzione)
   - se corrisponde a tutti 0
     - *allora* è un'istruzione di tipo R e bisogna andare a vedere l'operazione da effettuare in function (il campo "`funct`" dell'istruzione)
     - *altrimenti* l'opcode determina l'operazione da effettuare

Esempio 1

0010	0010	0011	0111	1111	1111	1111	0001
2	2	3	7	F	F	F	1

(**0x2237FFF1**)

⇓

op	rs	rt	imm	
001000	10001	10111	1111 1111 1111 0001	`addi $s7, $s1, -15`
8	17	23	-15	

Esempio 2

0000	0010	1111	0011	0100	0000	0010	0010
0	2	F	3	4	0	2	2

(**0x02F34022**)

⇓

op	rs	rt	rd	shamt	funct	
000000	10111	10011	01000	00000	100010	`sub $t0, $s7, $s3`
0	23	19	8	0	34	

Capitolo 4 - Costrutti standard

I comuni linguaggi di alto livello hanno istruzioni che mi permettono di fare determinate operazioni standard, tra cui:

`if / else` `cicli` `vettori / matrici` `funzioni`

4.1 Operazioni di tipo logico

Queste istruzioni lavorano sul singolo bit della `word`.

Tipologia	Operazione	C	Java	MIPS
Istruzioni di shift	Shift sinistro	<<	<<	`sll`
	Shift sinistro	>>	>>>	`srl`
Istruzioni logiche	Bitwise AND	`&`	`&`	`and,andi`
	Bitwise OR	|	|	`or,ori`
	Bitwise NOT	~	~	`nor`

4.1.1 Istruzioni logiche

- **`and`, `or`, `xor`, `nor`** (istruzioni di tipo R)
  - non esiste un'istruzione per effettuare l'operazione di `NOT`, in quanto ha un solo operando (operatore unario).
    Per risolvere questo problema si è deciso di non implementare il `NOT` ma di sfruttare un altro operatore, il `NOR`.
    ```
    NOR $t1, $t0, $0  # equivale a fare NOT $t0
    ```
- **`andi`, `ori`, `xori`** (istruzioni di tipo I)
  - non è implementata la `nori`

Estensione con zeri

Le istruzioni logiche che utilizzano un immediato devono effettuare una conversione a 32 bit.
Con l'estensione di zeri, si rappresenta l'immediato con i 16 bit meno significativi e i 16 bit più significativi sono posti a 0.

Esempio 1

Registri sorgente

`$s1`	1111	1111	1111	1111	0000	0000	0000	0000
`$s2`	0100	0110	1010	0001	1111	0000	1011	0111

Codice assembly

```
and $s3, $s1, $s2
or $s4, $s1, $s2
xor $s5, $s1, $s2
nor $s6, $s1, $s2
```

Risultato

`$s3`	0100	0110	1010	0001	0000	0000	0000	0000
`$s4`	1111	1111	1111	1111	1111	0000	1011	0111
`$s5`	1011	1001	0101	1110	1111	0000	1011	0111
`$s6`	0000	0000	0000	0000	0000	1111	0100	1000

Esempio 2

Registri sorgente

`$s1`	0000	0000	0000	0000	0000	0000	1111	1111
`imm`	0000	0000	0000	0000	1111	1010	0011	0100

estensione con zeri

Codice assembly

```
andi $s2, $s1, 0xFA34
ori $s3, $s1, 0xFA34
xori $s4, $s1, 0xFA34
```

Risultato

`$s2`	0000	0000	0000	0000	0000	0000	0011	0100
`$s3`	0000	0000	0000	0000	1111	1010	1111	1111
`$s4`	0000	0000	0000	0000	1111	1010	1100	1011

Esercizio

L'istruzione `nori` non fa parte del set di istruzioni MIPS perché la stessa funzionalità può essere implementata usando delle istruzioni già esistenti.
Scrivere un breve comando assembly che abbia la seguente funzionalità:

```
$t0 = $t1 NOR 0xF234
```

Nota: usare meno istruzioni possibili.

Soluzione

```
ORI $t0, $t1, 0xF234
NOR $t0, $t0, $0
```

4.1.2 Istruzioni di shift

Il campo che determina il numero di posizioni da shiftare è lo `shamt`.

op	rs	rt	rd	**shamt**	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Le operazioni di shift possibili sono 3:

`sll`	`srl`	`sra`
"shift logico a sinistra"	*"shift logico a destra"*	*"shift aritmetico a destra"*

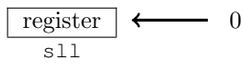
Esempi:

- `sll $t0, $t1, 5  # $t0 <= $t1 << 5`
- `srl $t0, $t1, 5  # $t0 <= $t1 >> 5`
- `sra $t0, $t1, 5  # $t0 <= $t1 >>> 5`

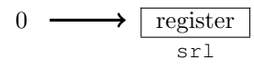
Nota bene: queste istruzioni non sono di tipo I nonostante utilizzino una costante (sono di tipo R).

Shift logico

a sinistra
"va a spostare la `word` dal registro sorgente al registro destinazione caricando il registro di destinazione con bit a 0 nelle cifre meno significative"
Con i numeri senza segno questo equivale ad una moltiplicazione per una potenza di $2^{\text{<numero di shift>}}$.



a destra
"va a spostare la `word` dal registro sorgente al registro destinazione caricando il registro di destinazione con bit a 0 nelle cifre più significative"
Con i numeri senza segno questo equivale ad una divisione per una potenza di $2^{\text{<numero di shift>}}$.

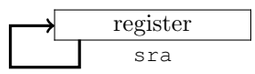


Esempio
Dato il numero `00 00 10 10`, con lo shift logico:

- a sinistra ottengo `00 10 10 00`
- a destra ottengo `00 00 00 10` (i bit meno significativi vengono persi)

Shift aritmetico

Lo shift aritmetico mi permette di fare una divisione per numeri in complemento a 2 (con segno).



Esempio
Dato il numero `1100` (-4), con lo shift aritmetico a destra ottengo `1110` ($-2 = -4/2$).

Shift variabile

Esistono anche le varianti delle istruzioni precedenti, con il numero di shift definito con una variabile anziché con un valore immediato.
Piuttosto di andare a leggere una costante, vado a leggere il valore di un registro.
Le operazioni possibili sono sempre 3:

`sllv`	`srlv`	`srav`
"shift logico variabile a sinistra"	*"shift logico variabile a destra"*	*"shift aritmetico variabile a destra"*

Esempi:

- `sllv $t0, $t1, $t2  # $t0 <= $t1 << $t2`
- `srlv $t0, $t1, $t2  # $t0 <= $t1 >> $t2`
- `srav $t0, $t1, $t2  # $t0 <= $t1 >>> $t2`

Nota bene: si dice variabile in quanto all'interno del codice posso variare il numero di bit per cui faccio lo shift.

Esempio

```
sll $t0, $s1, 2
srl $s2, $s1, 2
sra $s3, $s1, 2
```

Valori dei campi

op	rs	rt	rd	shamt	funct
0	0	17	8	2	0
0	0	17	18	2	2
0	0	17	19	2	3
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Codice macchina

op	rs	rt	rd	shamt	funct	
000000	00000	10001	01000	00010	000000	`(0x00114080)`
000000	00000	10001	10010	00010	000010	`(0x00119082)`
000000	00000	10001	10011	00010	000011	`(0x00119883)`
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

4.2 Codifica delle costanti

Poiché l'istruzione che contiene un immediato lo codifica su 16 bit, come si fa a lavorare con 32 bit?

Esempio
Voglio caricare nel registro `$s0` il contenuto di una costante (è un numero). È su 16 o 32 bit?

- se è su 16 bit questa operazione si fa con una normale istruzione di `addi`:

  Codice C
  ```
  // int è una word a 32 bit (con segno)
  a = 0x4f3c;
  ```
  Codice MIPS assembly
  ```
  # $s0 = a
  addi $s0, $0, 0x4f3c
  ```
  `Estensione del segno, 16 → 32 bit`

- se è su 32 bit si usano la `lui` e la `ori`:
  1. si "spezza" la `word` da 32 bit in 2 parti (alta e bassa) da 16 bit
  2. si prende la parte alta e si esegue la `lui` in un registro
  3. si prende la parte bassa e si esegue la `ori` tra il registro dove ho salvato la parte alta ed il valore della parte bassa salvando il risultato nello stesso registro di partenza

  Codice C
  ```
  // int è una word a 32 bit (con segno)
  a = 0xFEDC8765;
  ```
  Codice MIPS assembly
  ```
  # $s0 = a
  lui $s0, 0xFEDC
  ori $s0, $s0, 0x8765
  ```

Nota bene: queste istruzioni sono di tipo I in quanto sono presenti dei valori immediati (la `lui` ha un registro sorgente non utilizzato, posto a 0).

4.3 Moltiplicazione e divisione

Le istruzioni di moltiplicazione e divisione utilizzano 2 registri particolari (aggiuntivi, fuori dalla categoria dei 32 registri): `lo`, `hi`.
Sono dei registri interni su cui noi non abbiamo modo di agire a meno che si utilizzino determinate istruzioni.

Moltiplicazione

La moltiplicazione di 2 numeri a 32 bit può creare un problema di overflow ovvero potrebbe non essere possibile rappresentare il dato su 32 bit.
Per tutelarsi, MIPS restituisce un risultato a 64 bit.

```
mult $s0, $s1  # risultato contenuto in {hi, lo}, registri a 32 bit
```

La parte alta del risultato viene salvata in `hi` e la parte bassa in `lo`.

Divisione

La divisione di due numeri a 32 bit memorizza:
- il quoziente nel registro `lo`
- il resto nel registro `hi`

Per ottenere i valori contenuti nei registri esistono 2 istruzioni specifiche:
- la "*move from lo*" (`mflo`) che copia il valore del registro `lo` nel registro passato tramite parametro
- la "*move from hi*" (`mfhi`) che copia il valore del registro `hi` nel registro passato tramite parametro

Nota bene: queste istruzioni sono di tipo R in quanto non sono presenti dei valori immediati.

Esempio

```
mult $s0, $s1 # risultato -> parte alta in 'hi', parte bassa in 'lo'
div  $s0, $s1 # risultato -> quoziente in 'lo', resto in 'hi'

mflo $s2 # copia il contenuto del registro 'lo' nel registro 's2'
mfhi $s3 # copia il contenuto del registro 'hi' nel registro 's3'
```

Capitolo 5 - Pseudoistruzioni

Le pseudoistruzioni (anche chiamate macroistruzioni) sono istruzioni che vengono codificate in una o più istruzioni macchina (a differenza delle normali istruzioni MIPS).
Sono state create per rendere i programmi più leggibili.

5.1 Load immediate

La "*load immediate*" (`li`) è una pseudoistruzione (non è un'istruzione MIPS e quindi non si può identificare in una tipologia di istruzioni).

Esempio
`li $t0, 4` viene tradotta in `ori $t0, $0, 4`

Esempio
Se ho dei numeri da 64 bit, per esempio:

```
li $t0, 90000     |     li $t0, -5
```

vengono tradotti in un altro modo.
`li $t0, 90000`, per esempio, viene tradotta in

```
lui $at, 1 # carica la parte superiore (uguale a 65536 (2^16))
ori $t0, $at, 24464 # carica la parte inferiore (per arrivare a 90000)
```

Nota bene: il registro **`$at`** viene usato solamente per le pseudoistruzioni.

5.2 Load address

La "*load address*" (`la`) è una pseudoistruzione (non è un'istruzione MIPS e quindi non si può identificare in una tipologia di istruzioni).

Esempio
`la $t0, label` viene tradotta in

```
lui $at, n # load upper 16 bits of label
ori $t0, $at, m # lower 16 bits of label
```

5.3 Altre pseudoistruzioni

La moltiplicazione (`mul`) e la divisione sono delle macroistruzioni in quanto si basano su più istruzioni.

Moltiplicazione

```
mul rd, rs, rt
```

```
mult rs, rt
mflo rd
```

Divisione

```
div rd, rs, rt
```

```
    bne rt, $0, ok
    break $0
ok: div rs, rt
    mflo rd
```

Nota bene: la `mul` si utilizza solamente se siamo sicuri che il risultato sia su 32 bit.

Esempio

Il seguente esercizio serve per calcolare il volume e l'area di un parallelepipedo. Le formule sono le seguenti:
`volume = aSide · bSide · cSide`
`surfaceArea = 2 · (aSide · bSide + aSide · cSide + bSide · cSide)`

Codice

```
# Example to compute the volume and surface area
# of a rectangular parallelepiped.

# ------------------------------------------------------
# Data Declarations
.data
aside :          . word 73
bSide :          . word 14
cSide:           . word 16

volume :         . word 0
surfaceArea:     . word 0

# ------------------------------------------------------
# Text/code section

.text
.globl      main
main:
# -----
# Load variables into registers.
    lw $t0, aSide
    lw $t1, bSide
    lw $t2, cSide

# ----
# Find volume of a rectangular paralllelpiped.
#   volume aside * bSide * cSide
    mul $t3, $t0, $t1
    mul $t4, $t3, $t2
    sw $t4, volume

# -----
# Find surface area of a rectangular parallelepiped.
# surfaceArea = 2*(aSide*bSide+aSide*cSide+bSide*cSide)
    mul $t3, $t0, $t1       # aSide * bSide
    mul $t4, $t0, $t2       # aSide * cSide
    mul $t5, $t1, $t2       # bSide * cSide
    add $t6, $t3, $t4
    add $t7, $t6, $t5
    mul $t7, $t7, 2
    sw $t7, surfaceArea

# -----
# Done, terminate program.
    li $vo, 10              # call code for terminate
    syscall                 # system call (terminate)
.end main
```

Capitolo 6 - Controllo del flusso

Le seguenti istruzioni servono per fare i salti e permettono di definire un controllo del flusso all'interno del codice. I salti possono essere di diverso tipo:

- condizionali
- incondizionali

TODO: rivedere **Codice MIPS assembly**

```
addi $s0, $0, 4 # $s0 = 0 + 4 = 4
addi $s1, $0, 1 # $s1 = 0 + 1 = 1
sll $s1, $s1, 2 # $s1 = 1 << 2 = 4
beq $s0, $s1, target # branch is taken
addi $s1, $s1, 1 # not executed
sub $s1, $s1, $s0 # not executed

target:  # label
add $s1, $s1, $s0 # $s1 = 4 + 4 = 8
```

Codice MIPS assembly

```
addi $s0, $0, 4 # $s0 = 0 + 4 = 4
addi $s1, $0, 1 # $s1 = 0 + 1 = 1
sll $s1, $s1, 2 # $s1 = 1 << 2 = 4
bne $s0, $s1, target # branch not taken
addi $s1, $s1, 1 # not executed
sub $s1, $s1, $s0 # not executed

target:  # label
add $s1, $s1, $s0 # $s1 = 1 + 4 = 5
```

Codice MIPS assembly

```
addi $s0, $0, 4 # $s0 = 4
addi $s1, $0, 1 # $s1 = 1
j target # jump to target
sra $s1, $s1, 2 # not executed
addi $s1, $s1, 1 # not executed
sub $s1, $s1, $s0 # not executed

target:
add $s1, $s1, $s0 # $s1 = 1 + 4 = 5
```

Codice MIPS assembly

```
0x00002000      addi $s0, $0, 0x2010
0x00002004      jr $s0
0x00002008      addi $s1, $0, 1
0x0000200C      sra $s1, $s1, 2
0x00002010      lw $s3, 44($s1)
```

6.1 Costrutti di alto livello

TODO

6.1.1 if statement (TODO)

Codice C

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

f = f - i;
```

Codice MIPS assembly

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
  bne $s3, $s4, L1
  add $s0, $s1, $s2

L1:  sub $s0, $s0, $s3
```

Codice C

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

Codice MIPS assembly

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
  bne $s3, $s4, L1
  add $s0, $s1, $s2
  j done
L1:  sub $s0, $s0, $s3
done:
```

TODO

```
beq $s3, $s4, L2
J L1
L2:  add $s0, $s1, $s2
L1:  sub $s0, $s0, $s3
```

Nota bene: assembly testa i casi negativi (i != j) al contrario dell'alto livello (i == j).

6.1.2 Ciclo while

TODO

6.1.3 Ciclo for

TODO

6.1.4 Magnitude comparison (TODO)

slt - Set on Less Than TODO TODO

Codice C

```
// add the powers of 2 from 1
// to 100
int sum = 0;
int i;

for (i = 1; i < 101; i = i * 2) {
  sum = sum + i;
}
```

TODO

Codice MIPS assembly

```
# $s0 = i, $s1 = sum
       addi $s1, $0, 0
       addi $s0, $0, 1
       addi $t0, $0, 101
loop:  slt $t1, $s0, $t0
       beq $t1, $0, done
       add $s1, $s1, $s0
       sll $s0, $s0, 1
       j loop
done:
```