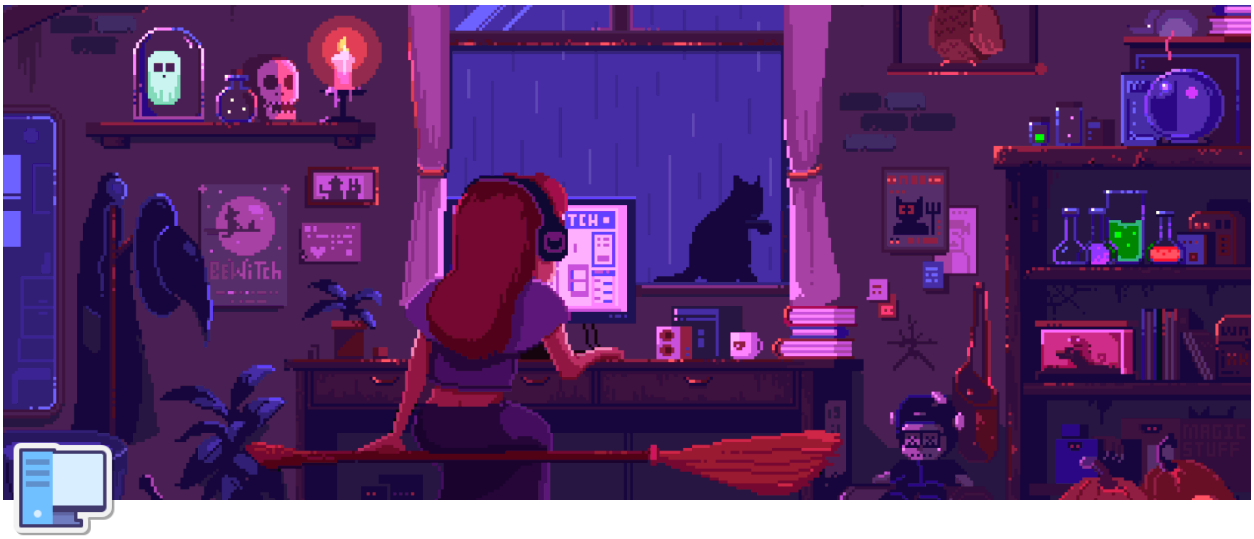


Il linguaggio Assembly MIPS

▼ ***INDICE***

2 Il linguaggio Assembly MIPS

Ora che sappiamo che per impartire comandi ad un calcolatore è necessario conoscere il linguaggio(in particolare le sue istruzioni) andiamo a vedere il “vocabolario” di un reale computer, sia in forma umanamente leggibile(linguaggio Assembly), sia in forma meccanicalmente leggibile(linguaggio macchina).

2.1 Formato delle istruzioni

Partiamo subito col dire che le istruzioni della CPU dell'architettura MIPS, seguono una struttura molto semplice:

$$< operazione >< destinazione >, < sorgente >< argomenti >$$

Per capire meglio di cosa parliamo vediamo subito un esempio:

```
add $s0, $t0, $t1
```

L'istruzione appena scritta corrisponde alle seguenti 3 operazioni:

- Leggi il registro $\$t0$ e il registro $\$t1$(sorgenti);
- Somma i loro valori;
- Scrivi il risultato sul registro $\$s0$(destinazione).

> ⚠️ ATTENZIONE:
>
> Questa struttura è solo una generalizzazione, visto che non è rispettata da tutte le istruzioni.

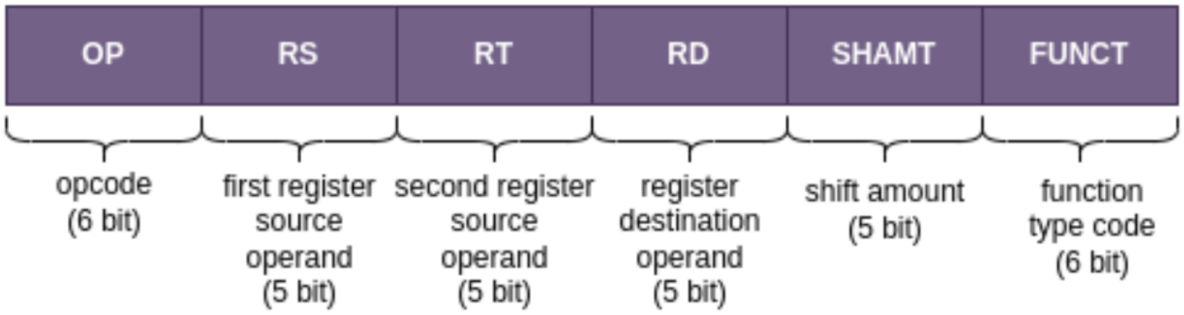
Detto questo però sorge un dubbio, se tutte le istruzioni corrispondono ad una word di 32 bit, come possono avere una struttura variabile?

La risposta è semplice perché ad avere un formato fisso non sono le istruzioni bensì le istruzioni interpretate dall'assemblatore in codice macchina.

Tali istruzioni vengono tradotte dall'assemblatore in un formato specifico determinato dalla tipologia stessa dell'istruzione:

- Istruzioni r-type(tipo di registro):
  - Senza accesso alla memoria;
  - Istruzioni di tipo aritmetico e di tipo logico;

  Il formato dei bit é:



Vediamone subito un esempio:

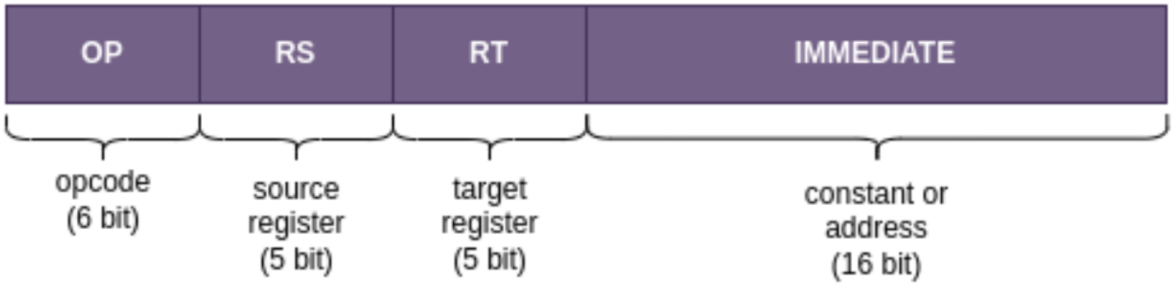
Istruzione	OP	RS	RT	RD	SHAMT	FUNCT
add $t0,$s1, $s2	000000	10001	10010	01000	00000	100000
sub $t0,$s1, $s2	000000	10001	10010	01000	00000	100010

Dove:

- Opcode(OP): indica la categoria di operazione da eseguire, in questo caso, 000000 essendo un'operazione aritmetica;
- First register(RS): indica il primo registro sorgente da cui leggere il valore. In questo caso 10001 corrisponde al registro $\$s1$;
- Second register(RT): indica il secondo registro da cui leggere il valore. In questo caso 10010 corrisponde al registro $\$s2$;
- Destination register(RD): indica il registro su cui scrivere il risultato. In questo caso, 01000 corrisponde al registro $\$t0$;
- Shift amount(SHAMT): indica la quantità di bit da shiftare. In questo caso è 0 perché non shiftiamo nulla.
- Function code(FUNCT): indica la specifica secondaria dell'operazione da eseguire, dunque corrisponde ad un'estensione dell'Opcode.

- Istruzioni i-type(tipo immediato):
  - Operazioni di Load e Store;
  - Utilizzate dai salti condizionati(ossia relativi al Program Counter).

  Il formato dei bit è:



Vediamone subito un esempio:

Istruzione	OP	RS	RT	IMMEDIATE
addi $t2,$s2,17	00100	11010	01010	0000000000010001

- Opcode(OP): viene specificata l'operazione di addizione immediata, ossia non tra due registri ma tra un registro e un valore costante(nel nostro caso 17);
- First register(RS): viene letto il registro $\$s2$;
- Second register(RT): viene specificato dove scrivere il risultato della somma, in questo su $\$t2$;
- Immediate: viene specificato il valore costante con cui fare la somma immediata, ossia $17(10001$ in binario).

- Istruzioni j-type(tipo jump):
  - Utilizzate dai salti non condizionati(ossia assoluti);

Il formato dei bit è:

Vediamone subito un esempio:

Istruzione	OP	ADDRESS
j 2500	000010	00000000000010011100010000

- Opcode(OP): viene specificata l'operazione di jump incondizionato;

- Address: viene specificato l'indirizzo su cui effettuare il jump. Il valore indicato, in realtà, rappresenta $2500 \cdot 4 (o\ 2500 << 2)$, perché l'Architettura MIPS è indicizzata al byte, dunque la 2500-esima parola corrisponde all'indirizzo 10000 della memoria.

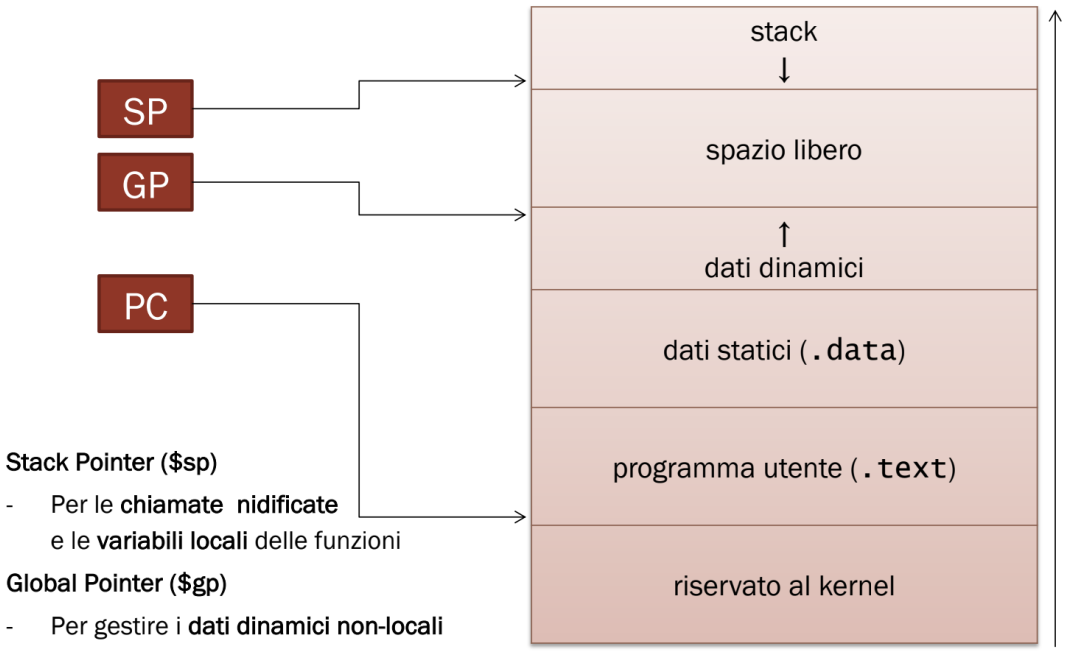
2.2 Lista delle istruzioni

Qui sotto verrà riportata una tabella con tutti i comandi più importanti dal poter utilizzare:

Tipo di istruzioni	Istruzioni	Esempio	Significato	Commenti
Aritmetiche	Somma	`add $s1,$s2,$s3`	`$s1 = $s2 + $s3`	Operandi in tre registri
	Sottrazione	`sub $s1,$s2,$s3`	`$s1 = $s2 - $s3`	Operandi in tre registri
	Somma immediata	`addi $s1,$s2,20`	`$s1 = $s2 + 20`	Utilizzata per sommare delle costanti
Trasferimento dati	Lettura parola	`lw $s1,20($s2)`	`$s1=Memoria[$s2+20]`	Trasferimento di una parola da memoria a registro
	Memorizzazione parola	`sw $s1,20($s2)`	`Memoria[$s2+20]=$s1`	Trasferimento di una parola da registro a memoria
	Lettura mezza parola	`lh $s1,20($s2)`	`$s1=Memoria[$s2+20]`	Trasferimento di una mezza parola da memoria a registro
	Lettura mezza parola, senza segno	`lhu $s1,20($s2)`	`$s1=Memoria[$s2+20]`	Trasferimento di una mezza parola da memoria a registro
	Memorizzazione mezza parola	`sh $s1,20($s2)`	`Memoria[$s2+20]=$s1`	Trasferimento di una mezza parola da registro a memoria
	Lettura byte	`lb $s1,20($s2)`	`$s1=Memoria[$s2+20]`	Trasferimento di un byte da memoria a registro
	Lettura byte senza segno	`lbu $s1,20($s2)`	`$s1=Memoria[$s2+20]`	Trasferimento di un byte da memoria a registro
	Memorizzazione byte	`sb $s1,20($s2)`	`Memoria[$s2+20]=$s1`	Trasferimento di un byte da registro a memoria
	Lettura di una parola e blocco	`ll $s1,20($s2)`	`$s1=Memoria[$s2+20]`	Caricamento di una parola come prima fase di un'operazione atomica
	Memorizzazione condizionata di una parola	`sc $s1,20($s2)`	`Memoria[$s2+20]=$s1; $s1=0 oppure 1`	Memorizzazione di una parola come seconda fase di un'operazione atomica
	Caricamento costante nella mezza parola superiore	`lui $s1,20`	`$s1 = 20 * 2^16`	Caricamento di una costante nei 16 bit più significativi
Logiche	And	`and $s1,$s2,$s3`	`$s1 = $s2 & $s3`	Operandi in tre registri; AND bit a bit
	Or	`or $s1,$s2,$s3`	`$s1 = $s2 | $s3`	Operandi in tre registri; OR bit a bit
	Nor	`nor $s1,$s2,$s3`	`$s1 = ~($s2 | $s3)`	Operandi in tre registri; NOR bit a bit

2.1 Parti della memoria

Adesso andiamo a vedere la struttura della memoria, definendo quali sono le parti principali:



Come possiamo vedere abbiamo:

- Stack: si occupa delle operazioni relative alle funzioni, salvandone le chiamate ricorsive. Non ha una dimensione fissa, infatti si può espandere nella sezione di memoria libera;
- Dati dinamici(Dynamic Data/Heap): contiene tutti i dati dinamici usati nel programma, anch'essa può espandersi nella sezione di memoria libera;
- Dati statici(Static Data): contiene tutti i dati statici che vengono definiti nel programma(etichettate sotto la direttiva $.data$). Il registro $\$gp$(Global Pointer), viene utilizzato per gestire gli indicizzamenti all'interno di questa zona;
- Programma utente(Program Istruction): contiene tutte le istruzioni del programma(etichettate sotto la direttiva $.text$). Al suo interno opera il Program Counter, ossia il registro che memorizza la posizione in memoria dell'istruzione successiva da eseguire;
- Riservato al Kernel(Kernel-reserved): ossia lo spazio di memoria inutilizzabile dal programmatore, poiché riservato al Kernel del Sistema Operativo.

2.2 Direttive principali ed esempi di codice

Le direttive principali del linguaggio Assembly, non corrispondono in modo diretto ad una particolare istruzione, ma vengono interpretate dall'assemblatore.

Le direttive principali del linguaggio Assembly MIPS sono:

- .data: utilizzata per definire dati statici;
- .text: utilizzata per definire le istruzioni del programma;
- .asciiz: utilizzata per definire una stringa di caratteri terminata da un byte *null*, ossia “0”, che va ad indicare la fine della stringa;
- .byte: utilizzata per definire una sequenza di byte;
- .double: utilizzata per definire una sequenza di valori double(ossia a doppia precisione);
- .float: utlizzata per definire una sequenza di valori float(ossia a singola precisione);
- .half: utilizzata per definire una sequenza di half words;
- .word: utilizzata per definire una sequenza di word;

Andiamo ora a vedere come scrivere un codice in Assembly:

Innanzitutto dobbiamo andare a scrivere *.gobl main,* dove grazie a questo possiamo dire che la “funzione” è globale ossia che può essere accessibile da tutti i file che si trovano al di fuori del file stesso.

Successivamente andremo a fare gli step fondamentali per poter programmare, uno di questi è il *.data*, dove andremo ad elencare tutte le variabili che utilizzeremo nel *.text*(nel quale andremo a scrivere invece il “comportamento” di tali variabili.

Proviamo ora a vedere un semplice esempio:

📎 ***ESEMPIO DI CODICE INIZIALE:***

Proviamo in .data a scrivere un numero:

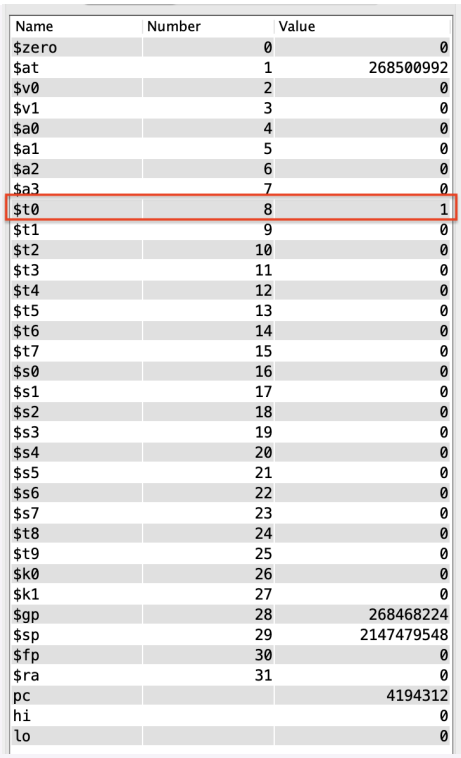
```
.globl main
.data
numero: word 1
.text
main:
```

Scrivendo questo vuol'dire che questo numero(ossia 1) viene rappresentata in 32 bit, quindi nel nostro caso ci saranno 31 bit uguali a 0 e l'ultimo uguale ad 1.

Questo però non si trova all'interno di nessun registro, quindi andiamo a vedere come spostare una variabile all'interno di un registro:

```
.globl main
.data
numero: word 1
.text
main:
lw $t0, numero
```

Per far questo come possiamo vedere all'interno del main andiamo a usare il comando lw(ossia load word) il quale andrà a spostare il nostro numero all'interno del registro $\$t0$, quindi se andiamo a vedere nella tabella a destra su MARS possiamo vedere che il valore di $\$t0$ è diventato 1:



Name	Number	Value
$zero	0	0
$at	1	268500992
$v0	2	0
$v1	3	0
$a0	4	0
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	1
$t1	9	0
$t2	10	0
$t3	11	0
$t4	12	0
$t5	13	0
$t6	14	0
$t7	15	0
$s0	16	0
$s1	17	0
$s2	18	0
$s3	19	0
$s4	20	0
$s5	21	0
$s6	22	0
$s7	23	0
$t8	24	0
$t9	25	0
$k0	26	0
$k1	27	0
$gp	28	268468224
$sp	29	2147479548
$fp	30	0
$ra	31	0
pc		4194312
hi		0
lo		0

2.2.1 Addizione tra interi

Riprendendo l'esempio precedente, dove abbiamo caricato numero(1) all'interno del registro $\$t0$, proviamo a spostare il valore all'interno di $\$t1$.

Per far questo basterà scrivere:

```
.globl main
.data
numero: word 1
.text
main:
lw $t0, numero
move $t1, $t0
```

Con il comando *move* andremo a copiare ed incollare il valore contenuto nel registro $\$t0$ all'interno del registro $\$t1$(quindi non priveremo $\$t0$ del suo valore).

Ci sono altri comandi con il quale possiamo effettuare questo spostamento, e sono:

- add: dove il primo registro è la destinazione e gli altri 2 registri sono gli addendi, quindi nel caso volessimo usare questo comando per spostare un valore dovremo scrivere:

```
.globl main
.data
numero: word 1
.text
main:
lw $t0, numero
move $t1, $t0
add $t2, $t0, $0
```

Scrivendo questo stiamo semplicemente caricando la somma tra $\$t0$ e $\$0$(che vale sempre 0) quindi $\$t2$ varrà 1.

> 📖 ***NOTA:***
>
> Ovviamente da come si può ben capire questo comando può essere utilizzato anche per fare le somme tra due interi.

- addi: il comando è molto simile al precedente. l'unica differenza è che addi lo uso quando come terzo operando abbiamo un numero, ossia:

```
.globl main
.data
numero: word 1
.text
main:
lw $t0, numero
move $t1, $t0
add $t2, $t0, $0
addi $t3, $t2, 5
```

In questo caso semplicemente con il comando andremo a dare come valore al registro $\$t3$ la somma tra $\$t2$(che vale 1) e il numero 5, di conseguenza il suo valore sarà 6.

2.2.2 Sottrazione tra interi

Per fare una sottrazione esistono 2 comandi molto simili ai due comandi appena visti, ossia add ed addi.

I comandi che servono ad effettuare una sottrazione tra interi sono *sub* e subi, intuitivamente possiamo notare che il funzionamento è simile ai comandi per la somma, ma andiamo comunque a vedere degli esempi:

📎 ***ESEMPI SUB E SUBI:***

Andiamo a vedere come prima cosa il funzionamento di sub:

```
.globl main
.data
numero: .word 10
numero2: .word 5
.text
main:
lw $t0, numero
lw $t1, numero2
sub $t3, $t0, $t1
```

Come si può notare per fare una sottrazione abbiamo dichiarato un'altra variabile in .data e successivamente è stata caricata sul registro $\$t1$.

Invece subi ha lo stesso funzionamento di addi, ossia che viene usato solo quando il terzo operando è un numero:

```
.globl main
.data
numero: .word 10
.text
main:
lw $t0, numero
subi $t1, $t0, 6
```

Quindi a $\$t0(10)$ sottraiamo 6, quindi il valore di $\$t1$ sarà 4.

📖 ***NOTA:***

Come possiamo vedere per “dichiarare” una variabile abbiamo sempre prima scritto dentro il .data e poi successivamente abbiamo caricato quel numero all’interno di un registro tramite *lw*.

Esiste però un modo più veloce, senza l’uso del .data e del comando *lw*, ossia:

```
.globl main
.data
numero: .word 10
.text
main:
lw $t0, numero
li $t1, 4
sub $t2, $t0, $t1
```

Come possiamo vedere tramite il comando *li* andiamo a caricare nel registro $\$t1$ un valore numerico immediato, che in questo caso è 4.

2.2.3 System calls

Con i termine System calls(oppure Syscall) si intende un set di servizi complessi messi a disposizione dal Kernel del Sistema Operativo. Un esempio tipico di una Syscall è il print usato in Python.

Generalmente, ogni architettura, MIPS inclusa, segue una struttura del seguente formato(scritto prendendo come base il precedente esempio):

```
li $v0, 1
move $a0, $t2
syscall
```

Andiamo ad analizzare le ultime 5 righe:

- Prima riga: in questa riga abbiamo intenzione di stampare un numero, questo lo si capisce osservando la seguente tabella:

Service	Code in $v0	Arguments	Result
print integer	1	$a0 = integer to print	
print float	2	$f12 = float to print	
print double	3	$f12 = double to print	
print string	4	$a0 = address of null-terminated string to print	
read integer	5		$v0 contains integer read
read float	6		$f0 contains float read
read double	7		$f0 contains double read
read string	8	$a0 = address of input buffer $a1 = maximum number of characters to read	*See note below table*
sbrk (allocate heap memory)	9	$a0 = number of bytes to allocate	$v0 contains address of allocated memory
exit (terminate execution)	10		

Dove per stampare un Integer dobbiamo scrivere:

```
li $v0, 1
```

- Seconda riga: dove molto semplicemente spostiamo il valore di $\$t2$ in $\$a0$, ossia il registro della syscall;
- Terza riga: con syscall intendiamo il classico print;

⚠️ ***ATTENZIONE:***

Alla fine di tutto bisogna terminare l'esecuzione del programma, in questo modo:

```
li $v0, 10
syscall
```

2.2.3 Operazioni con stringa

Una stringa è semplicemente una variabile, senza un valore numerico, ma solo lettere e numeri all'interno di 2 apici.

Per capire meglio di cosa parliamo, vediamo subito come si dichiara una stringa nel $.data$:

```
.globl main
.data
stringa: .asciiz "Hello world!"
.text
main:
```

In questo caso la direttiva cambia, essendo una stringa, e useremo $.asciiz$(utilizzata appositamente per definire una stringa).

Successivamente dobbiamo caricare la stringa su un registro usando il comando *la*, che a differenza del *lw* va a caricare l'indirizzo della nostra variabile.

```
la $t0, stringa
```

Ora proviamo a stampare la nostra stringa:

```
.globl main
.data
stringa: .asciiz "Hello world!"
.text
main:
la $t0, stringa
li $v0, 4 #4 per indicare che andiamo a stampare una stringa
move $a0, $t0
syscall
li $v0,10
syscall
```