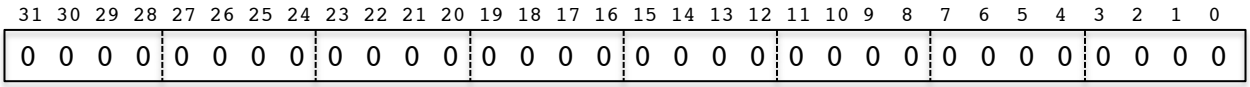
# MIPS

Sono descritte solamente le istruzioni di MIPS32, le pseudo-istruzioni, System Calls e direttive del linguaggio assembly che sono maggiormente usate.
MIPS è ***big-endian***, cioè, l'indirizzo di una word (4 bytes) è quello del byte più a sinistra.
I bits in una word sono numerati da destra verso sinistra:




Il bit più a destra è il meno significativo (bit 0) e quello più a sinistra è il più significativo (bit 31).

| Leggenda | |
|---|---|
| `d, s, t` | Registro destinazione e registri sorgente |
| $x_{n:k}$ | Dall'`n`-esimo al `k`-esimo bit di `x` |
| `x::y` | Concatenazione dei bits di `x` e `y` |
| `Lo, Hi` | Parte bassa (31:0) e parte alta (63:32) dell'Accumulatore da 64 bits |
| `PC` | Program Counter (indirizzo dell'istruzione corrente) |
| `imm`*N* | Costante (*immediate*) intera con segno da *N* bits ($-2^{N-1} – 2^{N-1} – 1$) |
| `imm`*N*`u` | Costante (*immediate*) intera senza segno da *N* bits ($0 – 2^{N} – 1$) |
| `L`*N*`(x)` | Low-order *N* bits di `x` (gli *N* bits più a destra) |
| `H`*N*`(x)` | High-order *N* bits di `x` (gli *N* bits più a sinistra) |
| `ExtS(x), ExtZ(x)` | Estensione a 32 bits di `x` con segno e con zero, rispettivamente |
| `MemW[addr]` | Word che inizia all'indirizzo `addr` |
| `label` | Etichetta dell'assembly; `label_a` denota l'indirizzo |
| | In grigio le estensioni e le pseudo-istruzioni dell'assembler |

| Registri | | |
|---|---|---|
| `$zero` | `0` | Sempre uguale a zero |
| `$v0-$v1` | `2,3` | Valori ritornati da una chiamata a funzione |
| `$a0-$a3` | `4-7` | I primi 4 parametri di una chiamata a funzione |
| `$t0-$t9` | `8-15,24,25` | Variabili temporanee; *non necessariamente da preservare* |
| `$s0-$s7` | `16-23` | Variabili; *da preservare nelle chiamate a funzione* |
| `$gp` | `28` | Global pointer; *da preservare* |
| `$sp` | `29` | Stack Pointer; *da preservare* |
| `$fp` | `30` | Frame Pointer; *da preservare* |
| `$ra` | `31` | Indirizzo di ritorno dell'ultima chiamata; *da preservare* |
| `$at` | `1` | Variabile temporanea riservata all'assembler |
| `$k0-$k1` | `26,27` | Riservati per OS Kernel |

| Operazioni Aritmetiche | | |
|---|---|---|
| **add** | d, s, t | d = s + t  (overflow) |
| **addi** | d, s, imm16 | d = s + imm16  (overflow) |
| | d, s, imm32 | d = s + imm32  (overflow) |
| **addu** | Come add ma senza overflow | |
| **clo** | d, s | d = #leadingOnes(s)  (s = 0xf0000011, d = 4) |
| **clz** | d, s | d = #leadingZeros(s)  (s = 0x00ffffff, d = 8) |
| **div** | s, t | Lo = s / t,  Hi = s mod t  (no exceptions) |
| | d, s, t | d = s / t  (break on division by zero) |
| | d, s, imm32 | d = s / imm32  (no exceptions) |
| **mul** | s, t | Lo = L32(s * t), Hi = H32(s * t) |
| | d, s, t | d, Lo = L32(s * t), Hi = H32(s * t) |
| | d, s, imm32 | d, Lo = L32(s * imm32), Hi = H32(s * imm32) |
| **rem** | d, s, t | d = s mod t  (resto di s diviso t) |
| | d, s, imm32 | d = s mod imm32 |
| **slt** | d, s, t | d = (s < t ? 1 : 0) |
| **slti** | d, s, imm16 | d = (s < imm16 ? 1 : 0) |
| **sub** | d, s, t | d = s – t  (overflow) |
| | d, s, imm32 | d = s – imm32  (overflow) |
| **subi** | d, s, imm32 | d = s – imm32  (overflow) |
| **subu** | Come sub ma senza overflow | |

Il linguaggio assembly permette di specificare gli indirizzi combinando una label, una costante e un registro. Con `pseudoAddr` intendiamo che l'indirizzo effettivo `effAddr` può essere specificato in tutte le forme seguenti:

| pseudoAddr | effAddr |
|---|---|
| label | label_a |
| label(s) | label_a + s |
| label + imm32 | label_a + imm32 |
| label + imm32(s) | label_a + imm32 + s |
| imm32 | imm32 |
| imm32(s) | imm32 + s |
| (s) | s |

## Operazioni di Trasferimento Dati

| | | |
|---|---|---|
| **la** | d, pseudoAddr | d = effAddr |
| **lb** | d, imm16(s) | d = ExtS(L8(MemW[s + imm16])) |
| | d, pseudoAddr | d = ExtS(L8(MemW[effAddr])) |
| **lbu** | Come lb con ExtZ al posto di ExtS | |
| **li** | d, imm32 | d = ExtS(imm32) |
| **lui** | d, imm16u | d = imm16u::$0^{16}$ (load upper immediate) |
| **lw** | d, imm16(s) | d = MemW[s + imm16] |
| | d, pseudoAddr | d = MemW[effAddr] |
| **move** | d, s | d = s |
| **sb** | s, imm16(t) | L8(MemW[t + imm16]) = L8(s) |
| | s, pseudoAddr | L8(MemW[effAddr]) = L8(s) |
| **sw** | s, imm16(t) | MemW[t + imm16] = s |
| | s, pseudoAddr | MemW[effAddr] = s |

## Operazioni Logiche e Shift

| | | |
|---|---|---|
| **and** | d, s, t | d = s & t (AND bit a bit) |
| **andi** | d, s, imm16 | d = s & ExtZ(imm16) (AND bit a bit) |
| | d, s, imm32 | d = s & ExtZ(imm32) (AND bit a bit) |
| | d, imm32 | d = d & ExtZ(imm32) (AND bit a bit) |
| **nor** | d, s, t | d = ~(s \| t) (OR e poi NOT bit a bit) |
| **not** | d, s | d = ~s (NOT bit a bit) |
| **or** | d, s, t | d = s \| t (OR bit a bit) |
| **ori** | d, s, imm16 | d = s \| ExtZ(imm16) (OR bit a bit) |
| | d, s, imm32 | d = s \| ExtZ(imm32) (OR bit a bit) |
| | d, imm32 | d = d \| ExtZ(imm32) (OR bit a bit) |
| **xor, xori** | Come or e ori con XOR al posto di OR | |
| **rol** | d, s, imm5u | d = $s_{31-k:0}$::$s_{31:31-k+1}$, k = imm5u (rotazione a sinistra dei bits di s di imm5u posizioni) |
| | d, s, t | d = d = $s_{31-k:0}$::$s_{31:31-k+1}$, k = L5(t) |
| **ror** | Come rol per la rotazione a destra dei bits | |
| **sll** | d, s, imm5u | d = $s_{31-k:0}$::$0^k$, k = imm5u (shift left logical) |
| **sllv** | d, s, t | d = $s_{31-k:0}$::$0^k$, k = L5(t) |
| **srl** | d, s, imm5u | d = $0^k$::$s_{31:k}$, k = imm5u (shift right logical) |
| **srlv** | d, s, t | d = $0^k$::$s_{31:k}$, k = L5(t) |

| Operazioni di Branch e Jump | | |
|---|---|---|
| **b** | label | PC = label_a |
| **beq** | s, t, imm16 | if s = t then PC = PC + 4 + 4*imm16 |
| | s, t, label | if s = t then PC = label_a |
| | s, imm32, label | if s = imm32 then PC = label_a |
| **beqz** | s, label | if s = 0 then PC = label_a |
| **bge** | s, t, label | if s >= t then PC = label_a |
| | s, imm32, label | if s >= imm32 then PC = label_a |
| **bgez** | s, imm16 | if s >= 0 then PC = PC + 4 + 4*imm16 |
| | s, label | if s >= 0 then PC = label_a |
| **bgt** | Come bge con “s >” al posto di “s >=“ | |
| **bgtz** | Come bgez con “s > 0” al posto di “s >= 0” | |
| **ble** | Come bge con “s <=“ al posto di “s >=“ | |
| **blez** | Come bgez con “s <= 0“ al posto di “s >= 0” | |
| **blt** | Come bge con “s <“ al posto di “s >=“ | |
| **bltz** | Come bgez con “s < 0” al posto di “s >= 0” | |
| **bne** | Come beq con “s !=“ al posto di “s =“ | |
| **j** | imm26u | $PC = (PC+4)_{31:28}::imm26u::00$ |
| | label | PC = label_a |
| **jal** | imm26u | $\$ra = PC + 4, PC = (PC+4)_{31:28}::imm26u::00$ |
| | label | $\$ra = PC + 4, PC = label_a$ |
| **jr** | s | PC = s |

| Operazioni di Accesso all’Accumulatore da 64 bits | | |
|---|---|---|
| **mfhi** | d | d = Hi |
| **mflo** | d | d = Lo |
| **mthi** | s | Hi = s |
| **mtlo** | s | Lo = s |

Il linguaggio assembly rende disponibili dei **servizi di sistema** (**System Calls**), principalmente per l'input e l'output. Per usare un servizio procedere come segue:

1. caricare il codice del servizio nel registro `$v0` (vedi tabella sotto);
2. caricare gli argomenti, se ci sono, nei registri `$a0`, `$a1`, `$a2` come specificato dal servizio;
3. eseguire l'istruzione `syscall`;
4. l'eventuale valore ritornato si trova nel registro `$v0`.

Esempio: stampa sulla console il valore del registro `$t0`

```
li $v0, 1        #codice 1 corrisponde al servizio print_integer
move $a0, $t0    #carica $t0 nel registro per gli argomenti $a0
syscall          #esegue il servizio richiesto
```

Esempio: legge in input un intero e lo carica in `$t0`:

```
li $v0, 5        #codice 5 corrisponde al servizio read_integer
syscall          #esegue il servizio richiesto
move $t0, $v0    #carica l'intero letto dall'input in $t0
```

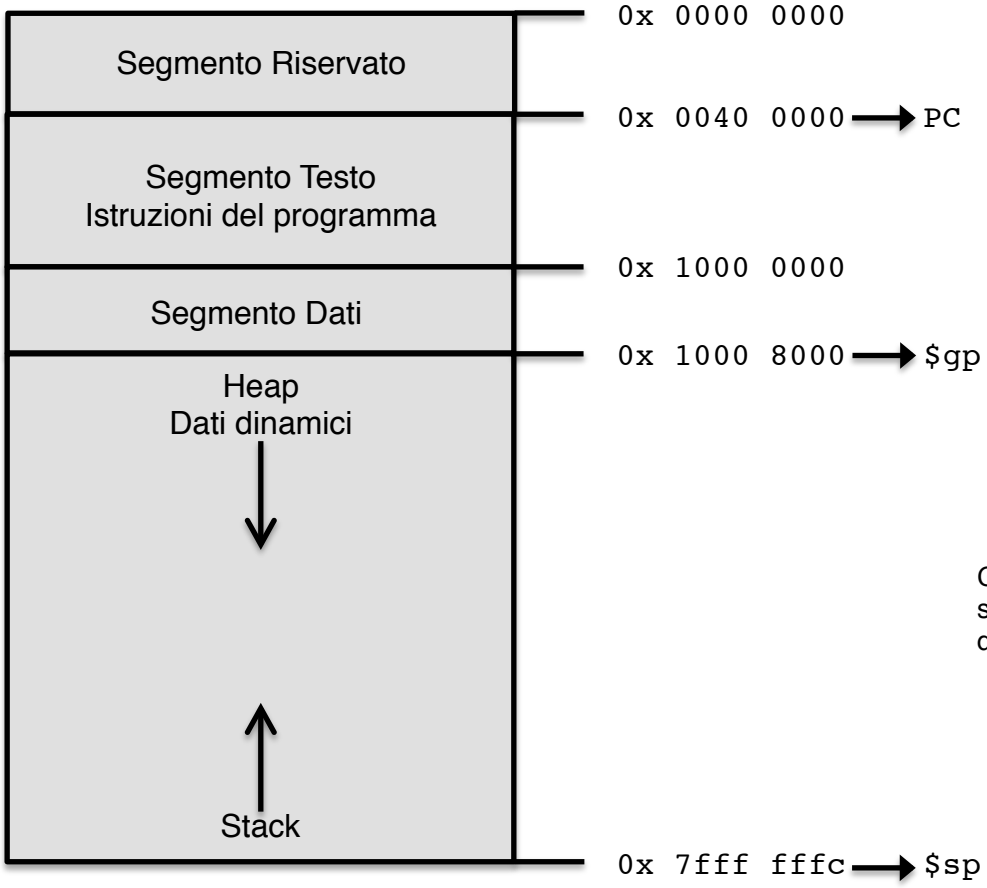
**Servizi di Sistema (System Calls)**

| **Servizio** | **Codice** | **Argomenti** | **Risultato** |
|---|---|---|---|
| **print_integer** | 1 | $a0 = intero da stampare | |
| **print_string** | 4 | $a0 = indirizzo di una stringa terminata con '\0' | |
| **read_integer** | 5 | | $v0 contiene l'intero letto |
| **read_string** | 8 | $a0 = indirizzo del buffer<br>$a1 = massimo numero di caratteri da leggere | Segue la semantica di fgets |
| **exit** | 10 | | Termina l'esecuzione |

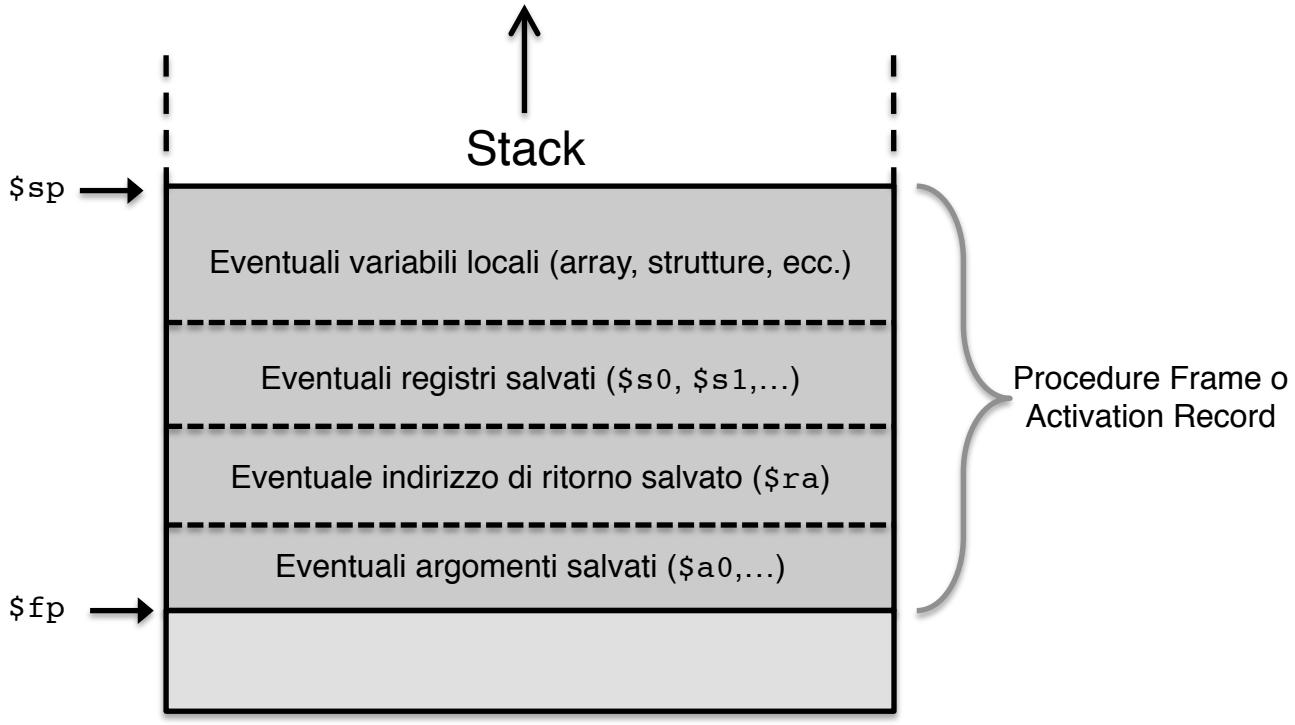
**Direttive dell'Assembler**

| | |
|---|---|
| **.align n** | Allinea il prossimo dato a $2^n$ bytes (0=byte, 1=half word, 2=word, 3=double word) |
| **.asciiz** | Memorizza la stringa nel Segmento Dati e la termina con il carattere '\0'. Esempio:<br>.asciiz "stringa"  #occupa 8 bytes |
| **.byte** | Memorizza i prossimi valori in bytes. Esempio:<br>.byte 1,2,3  #memorizza 1,2,3 in tre bytes consecutivi |
| **.data** | I prossimi dati sono memorizzati nel Segmento Dati. Esempio:<br>.data<br>.asciiz "Stringa" #memorizza la stringa nei primi 8 bytes<br>#del Segmento Dati<br>.byte 1,2,3  #li memorizza nei successi 3 bytes |
| **.space n** | Riserva i prossimi n bytes del Segmento Dati |
| **.text** | I prossimi elementi (istruzioni) sono memorizzati nel Segmento di Testo. |
| **.word** | Memorizza i prossimi valori in words. Esempio:<br>.word 10, 1000  #memorizza 10, 1000 in 2 words consecutive |

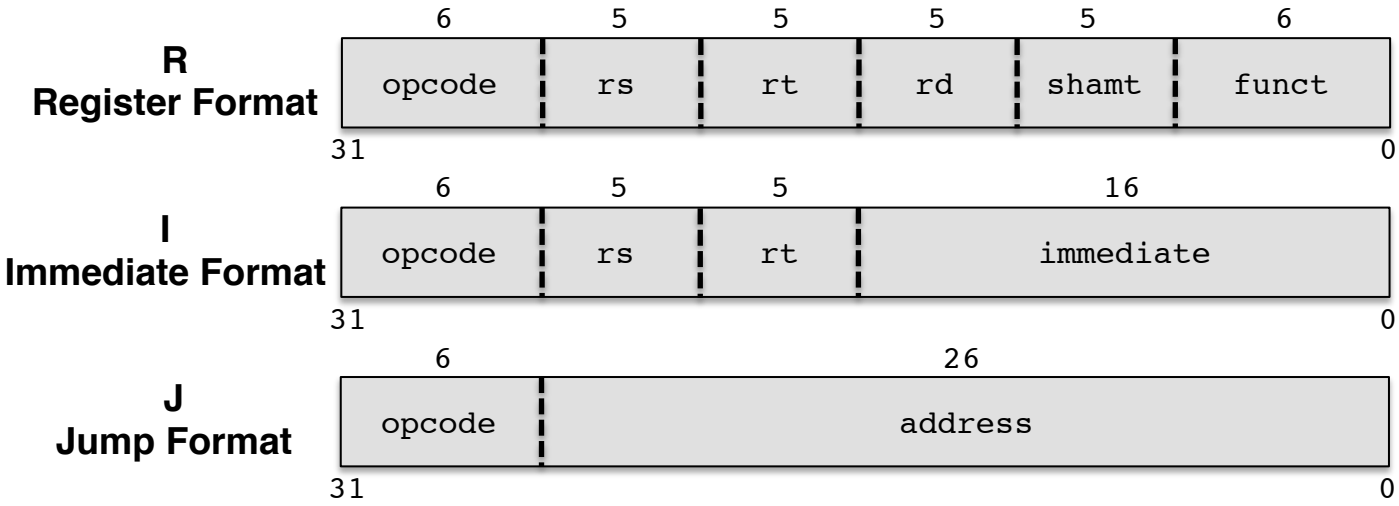
# Convezioni MIPS per l'uso della memoria e lo Stack




Gli indirizzi e le dimensioni dei segmenti della memoria dipendono dall'implementazione.



# Formati delle istruzioni MIPS




- `opcode`: campo che specifica l'operazione e il formato.
- `rs`, `rt`: numeri dei registri sorgente.
- `rd`: numero del registro destinazione.
- `shamt`: shift amount.
- `funct`: function code, seleziona la variante dell'operazione.

Esempi: