# Tutorato Architettura degli Elaboratori 05

Alberto Paparella[1]

8 Maggio 2025

[1]Dipartimento di Matematica e Informatica, Università degli studi di Ferrara

# Tutorial sulle funzionalità di base del simulatore MARS - Parte 2

# Syscall

- Le **syscall** sono letteralmente **chiamate al sistema operativo**, che servono principalmente per operazioni di input e output
- MARS emula queste chiamate di sistema
- Esistono diversi tipi di syscall, identificate da un numero, e funzionano in questo modo:
  1. carichiamo in un apposito registro il codice della syscall
  2. carichiamo gli eventuali argomenti in appositi registri
  3. chiamiamo la syscall
  4. recuperiamo gli eventuali valori di ritorno dai registri di risultato
- Trovate l'elenco completo di tutte le syscall al sito: `https://dpetersanderson.github.io/Help/SyscallHelp.html`

# Syscall “termina programma”

| Servizio | Codice in $v0 | Argomenti | Risultato |
|---|---|---|---|
| exit (termina l'esecuzione) | 10 | | |

- Emula la chiamata al sistema operativo che causa la terminazione del programma

```
1 .text
2     addi    $v0, $zero, 10
3     syscall
```

# Syscall “stampa di un intero”

| Servizio | Codice in $v0 | Argomenti | Risultato |
|---|---|---|---|
| stampa intero | 1 | $a0: intero da stampare | |

- Il codice della syscall va nel registro $v0, mentre il numero intero da stampare va in $a0

```
1 .text
2     addi    $a0, $zero, 42    # Carichiamo il valore da
      stampare in $a0
3     addi    $v0, $zero, 1   # Carichiamo il codice della
      syscall in $v0
4     syscall                 # Invochiamo la syscall con
      codice 1
5 # Risultato: stampa 42
```

# Più semplice con le Pseudo-Istruzioni!

- Con "load immediate (li)" posso caricare una costante in un registro

```
.text
    li   $a0, 42 # Carichiamo il valore da stampare in $a0
    li   $v0, 1  # Carichiamo il codice della syscall in $v0
    syscall      # Invochiamo la syscall con codice 1
# Risultato: stampa 42
```

- Con l'estensione della semantica di "load word (lw)" posso caricare direttamente un dato dalla memoria in un registro

```
.data
    A:   .word    42  # Allocazione di un intero inizializzato
     a 42

.text
    li   $a0, 42 # Carichiamo il valore da stampare in $a0
    li   $v0, 1  # Carichiamo il codice della syscall in $v0
    syscall      # Invochiamo la syscall con codice 1
# Risultato: stampa 42
```

# Syscall “stampa stringa”

| Servizio | Codice in $v0 | Argomenti | Risultato |
|---|---|---|---|
| stampa stringa | 4 | $a0: indirizzo della stringa null-terminated da stampare | |

- Utilizziamo la pseudo-istruzione “load address (la)”, che carica l’indirizzo di una locazione di memoria in un registro

```
1 .data
2     stringa: .asciiz "Ciao\n" # Allocazione di una stringa
      in memoria
3
4 .text
5     la   $a0, stringa # Carichiamo l’indirizzo di ‘‘stringa’’
       in $a0
6     li   $v0, 4 # Carichiamo il codice della syscall in $v0
7     syscall     # Invochiamo la syscall con codice 4
8 # Risultato: stampa la stringa
```

# Syscall “leggi intero”

| Servizio | Codice in $v0 | Argomenti | Risultato |
|---|---|---|---|
| leggi intero | 5 | | $v0 contiene l’intero da leggere |

- L’intero letto da standard input viene reso disponibile sul registro $v0

```
1 .text
2     li   $v0, 5   #Carichiamo il codice della syscall in $v0
3     syscall       # Invochiamo la syscall con codice 5
4                   # Valore letto in $v0
5
6     # Stampo il valore letto
7     add $a0, $v0, $zero # Travaso del valore letto in $a0
8     li   $v0, 1   # Syscall per la scrittura di un intero
9     syscall       # Stampa il valore letto
```

# Syscall “leggi stringa”

| Servizio | Codice in $v0 | Argomenti | Risultato |
|---|---|---|---|
| leggi stringa | 8 | $a0: indirizzo del buffer di input - $a1: massimo numero di caratteri da leggere | |

- Occorre riservare un buffer in zona dati
- Specificare l’argomento “n” per leggere “n-1” caratteri

```
1  .data
2      stringa: .asciiz
3
4  .text
5      li   $v0, 8   # Carichiamo il codice della syscall in $v0
6      la   $a0, stringa     # Indirizzo del buffer
7      li   $a1, 5   # Numero di caratteri da leggere (piu' uno)
8      syscall      # Invochiamo la syscall con codice 5
9
10     li $v0, 4    # Stampo la stringa letta
11     syscall
```

# Esercizio

- Inserire uno spazio tra la stringa letta e la stringa scritta
- Attenzione alla gestione della memoria!

# Soluzione

```
.data
    stringa: .asciiz
    .space  5
    separatore: .asciiz "\n"

.text
    li  $v0, 8  # Carichiamo il codice della syscall in $v0
    la  $a0, stringa    # Inidirizzo del buffer
    li  $a1, 5  # Numero di caratteri da leggere (piu' uno)
    syscall # Invochiamo la syscall con codice 5

    li      $v0, 4      # Stampo il separatore
    move    $t0, $a0    # Salvo l'indirizzo della stringa
    acquisita
    la  $a0, separatore # Carico l'indirizzo del separatore
    syscall # Stampo il separatore

    move    $a0 ,$t0    # Ripristino l'indirizzo della
    stringa acquisita
    syscall # Stampo la stringa acquisita
```

# Assemblatore

1. Programma scritto in linguaggio di alto livello (e.g., C)
2. **Compilatore C**
3. Programma scritto in linaugaggio Assembly (per il MIPS)
4. **Assemblatore**
5. Programma scritto in linguaggio macchina binario (per il MIPS)

L'assemblatore si occupa di:

- Generazione del linguaggio macchina
- Generazione degli indirizzi assoluti di memoria
- Gestione della endianess e degli allineamenti in memoria
- **Trasformazione delle pseudo-istruzioni in istruzioni dell'ISA**

# Pseudo-Istruzioni

- Ci sono sitruzioni assembler che non sono di facile implementazione in hardware
- Difatti, non fanno parte del set di istruzioni (e.g., dell'ISA del MIPS), ma sono istruzioni **astratte** che l'Assembler mette a disposizione:
  - esse vengono poi **tradotte** dall'Assemblatore nelle istruzioni che l'architettura MIPS "sa" eseguire
  - esse rendono più agevole la vita al programmatore, perchè il loro significato è intuitivo, e corrispondono ad operazioni che il programmatore si trova ad usare frequentemente
  - per ogni istruzione, il **text editore** di MARS suggerisce sia la disponibilità, sia la sintassi dei vari comandi, basta scrivere il nome del comando nell'editor e compare in sovraimpressione un mini-tutorial del comando stesso, se disponibile
- A queste istruzioni siamo il nome di **pseudo-istruzioni**

# Pseudo-Istruzioni

- Sono istruzioni assembler **virtuali**, che l'assemblatore mappa con facilità nelle istruzioni-macchina dell'Assembler reale
- Sono un primo banale livello di astrazione (come le **label**).

| | | | |
|---|---|---|---|
| `blt` | $1, $2, spi | se $1 $<$ $s2 salta | salta se strettamente minore |
| `bgt` | $1, $2, spi | se $1 $>$ $s2 salta | salta se strettamente maggiore |
| `ble` | $1, $2, spi | se $1 $\leq$ $s2 salta | salta se minore o uguale |
| `bge` | $1, $2, spi | se $1 $\geq$ $s2 salta | salta se maggiore o uguale |

**Table 1:** Pseudo-istruzioni per il **salto condizionato**.

# Pseudo-Istruzioni

| | | | |
|---|---|---|---|
| `lw` | $1, etichetta | $1 := mem ($gp + spi di etichetta) | carica parola (32 bit) |
| `sw` | $1, etichetta | mem($gp + spi di etichetta) := $1 | memorizza parola (32 bit) |

**Table 2:** Pseudo-istruzioni per il **trasferimento tra processore e memoria**.

| | | | |
|---|---|---|---|
| `li` | $1, const | $1 := const (32 bit) | carica costante a 32 bit |
| `la` | $1, indir | $1 := indir (32 bit) | carica indirizzo a 32 bit |

**Table 3:** Pseudo-istruzioni per il **caricamento di const/indirizzo in registro**.

| | | | |
|---|---|---|---|
| `move` | $1, $2 | $1 := $s2 | copia registro |

**Table 4:** Pseudo-istruzioni per il **trasferimento tra registri**.

# Cicli

# Esercizio 1 (for loop)

Si traduca in Assembler il seguente codice, scritto in linguaggio C, corrispondente a uno statement di controllo di tipo **for loop**:

```c
#include <stdio.h>

int main() {
    for (int i=0; i<10; i++) {
        printf("%d", i);
    }
    return 0;
}
```

# Esercizio 2 (while loop)

Si traduca in Assembler il seguente codice, scritto in linguaggio C, corrispondente a uno statement di controllo di tipo **while loop**:

```c
#include <stdio.h>

int main() {
    int i=0;
    while (i<10) {
        printf("%d", i++);
    }
    return 0;
}
```

# Soluzioni

# Esercizio 1 (for loop)

```
1  .data
2
3  .text
4      Main:
5          li $t0, 0    # i in $t0
6          li $v0, 1
7      ForLoop:
8          beq $t0, 10, ExitForLoop
9          move $a0, $t0
10         syscall
11         addi $t0, $t0, 1
12         j ForLoop
13     ExitForLoop:
14         li $v0, 10
15         syscall
```

# Esercizio 2 (while loop)

```
1  .data
2
3  .text
4          li $t0, 0    # i in $t0
5      Main:
6          li $v0, 1
7      WhileLoop:
8          bge $t0, 10, ExitWhileLoop
9          move $a0, $t0
10         syscall
11         addi $t0, $t0, 1
12         j WhileLoop
13     ExitWhileLoop:
14         li $v0, 10
15         syscall
```

# Nota bene

La soluzione dei due esercizi è praticamente **la stessa**.

Infatti, ogni ciclo **for** in C può essere convertito in un ciclo **while**.