Università degli Studi di Verona

Architettura degli Elaboratori

# Elaborato Assembly

*Mirko Morati, Noè Murr*

10 luglio 2016

# Indice

# 1 Premessa

Per la corretta lettura del pdf si informa che sono stati inseriti dei riferimenti per ogni etichetta e variabile trattata in modo da poter vedere e approfondire il corrispettivo codice nel programma.

# 2 Descrizione del progetto

Si vuole realizzare un programma `Assembly` per il monitoraggio di un motore a combustione interna il quale, ricevuto come ingresso il numero di giri/minuto del motore, fornisca in uscita la modalità di funzionamento corrente del motore: *Sotto Giri, Ottimale, Fuori Giri*. Il programma deve contare e visualizzare in uscita il numero dei secondi trascorsi nella modalità di funzionamento attuale ed inoltre attivare il segnale di allarme nel caso in cui il motore si trovi in modalità *Fuori Giri* da più di 15 secondi.

Di seguito verranno descritte le funzioni presenti in ogni file del programma, etichette, eventuali variabili e loro scopo.

# 3 syscall.inc

Header file contenente la definizione di alcune costanti, tramite la pseudo-operazione `.equ`, relative alle chiamate di sistema e ad alcuni standard utilizzati in tutti i file e riportati di seguito:

| | |
|---|---|
| SYS_EXIT | 1 |
| SYS_READ | 3 |
| SYS_WRITE | 4 |
| SYS_OPEN | 5 |
| SYS_CLOSE | 6 |
| STDIN | 0 |
| STDOUT | 1 |
| STDERR | 2 |
| SYSCALL | 0x80 |

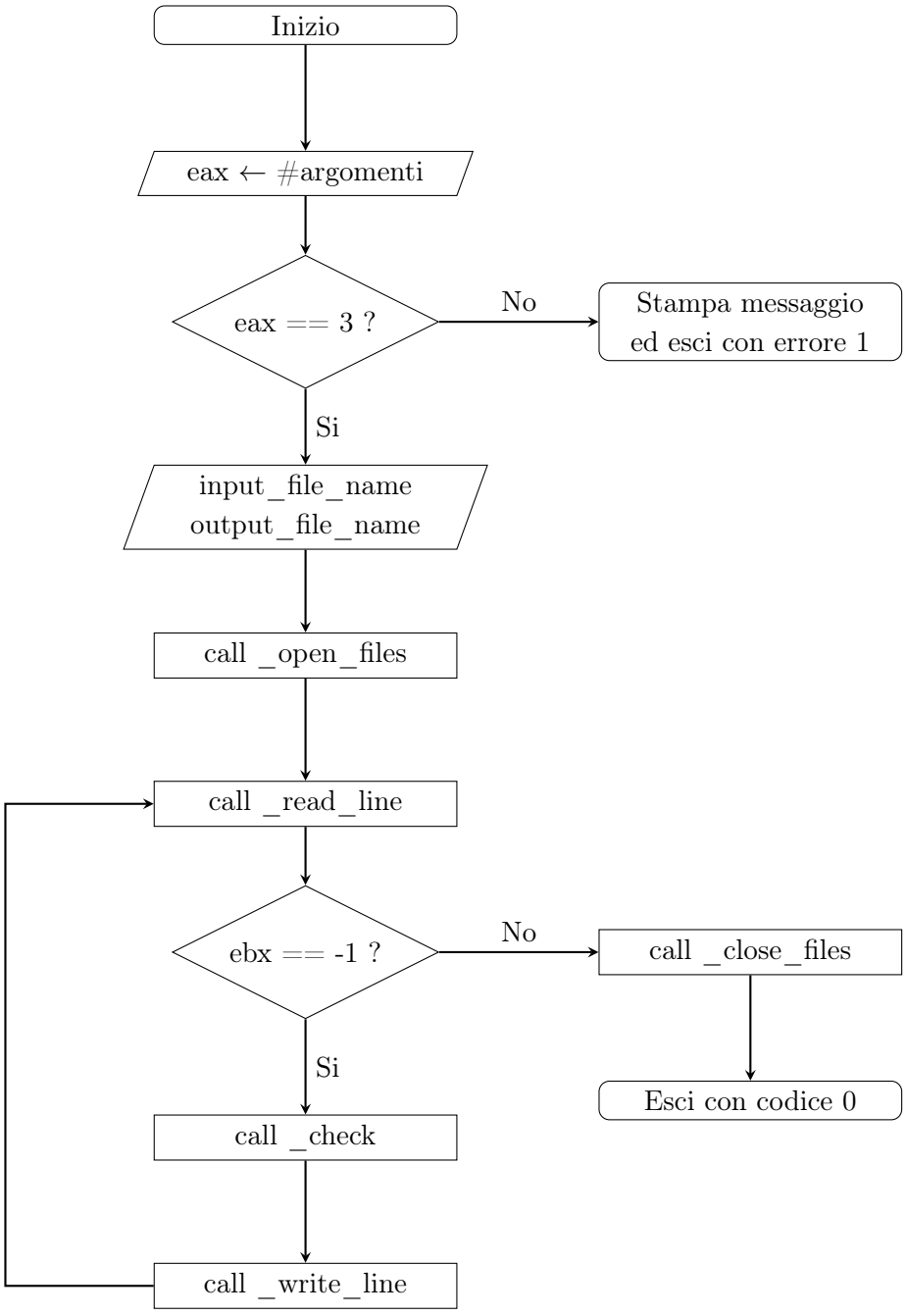
**Scelte Progettuali**

Nonostante il codice `Assembly` sia fortemente legato alla macchina sottostante, si è deciso di utilizzare un "trucchetto" per permetterne, per quanto possibile, la portabilità agevole su un'altra piattaforma. Infatti sarà sufficiente cambiare i codici delle `SYS_CALL` nel file `syscall.inc` per utilizzare le `SYS_CALL` di un altro sistema operativo. Questo vale solo su sistemi unix-like poiché i parametri delle funzioni essenziali (read, write, ...) sono comuni grazie allo standard `POSIX`.

# 4 main.s

File principale del programma.

## 4.1 Flowchart



## 4.2 Variabili Globali

- `input_fd`: Contiene il descrittore del file di input;
- `output_fd`: Contiene il descrittore del file di output;
- `init`: Contiene il valore del segnale INIT corrente;
- `reset`: Contiene il valore del segnale RESET corrente;
- `rpm`: Contiene il valore del segnale RPM corrente;
- `alm`: Contiene il valore del segnale ALM corrente;
- `mod`: Contiene il valore del segnale MOD corrente;
- `numb`: Contiene il valore del segnale NUMB corrente.

**Scelte Progettuali**

Si è scelto di utilizzare in modo estensivo le variabili globali in quanto in un progetto di così ridotte dimensioni si aumenta notevolmente la leggibilità del codice e nel contempo se ne riduce la complessità di scrittura. Se il progetto avesse richiesto computazioni più complesse questo approccio non sarebbe stato l'ideale, poiché sarebbero state compromesse le prestazioni totali dell'applicazione.

## 4.3 Variabili Locali

- `usage`: Stringa per la descrizione del corretto utilizzo del programma;
- `USAGE_LENGTH`: Contiene la lunghezza della stringa, necessaria per la stampa.

## 4.4 Funzioni ed Etichette

- `_start`: Punto di entrata del programma. Si occupa di controllare che il numero di parametri sia corretto, in caso contrario stampa la stringa `usage` e termina. Dopo il controllo chiama la funzione `_open_files` definita nel file `open_files.s`.
- `_main_loop`: Loop principale. Viene chiamata la funzione `_read_line` definita nel file `read_line.s`, nel caso in cui il contenuto del registro `ebx` sia equivalente a -1 significa che il file di input è terminato (**EOF** → End Of File) quindi salta a `_end`, altrimenti chiama la funzione `_check` definita nel file `check.s` e la funzione `_write_line` definita nel file `write_line.s`, dopodiché riesegue il ciclo.
- `_end`: Si occupa di chiudere tutti i file aperti e della corretta uscita dal programma tramite la chiamata di sistema `exit`.
- `_show_usage`: Nel caso in cui i parametri non siano corretti stampa a video la stringa `usage` e termina il programma segnalando errore con il codice 1.

# 5 open_files.s

Contiene la funzione che si occupa di aprire i file in modo corretto.

## 5.1 Variabili Locali

- `error_opening_files`: Stringa di errore in caso di errata apertura dei file (e.g. file mancante, file corrotto, ...).
- `ERROR_OPENING_LENGTH`: Costante che contiene la lunghezza della stringa di errore.

## 5.2 Funzioni ed Etichette

- `_open_files`: Si occupa di aprire i file e gestisce eventuali errori. In caso il file di output non esistesse, questo viene creato in automatico con permessi di lettura e scrittura. I descrittori ottenuti vengono salvati nelle corrispondenti variabili globali.
- `_error_opening_files`: In caso di errore viene stampato su `STDERR` la stringa opportuna, dopodiché il programma viene terminato con codice di errore 2.

# 6 read_line.s

Contiene la funzione che si occupa di leggere ed interpretare una riga per volta del file di input.

## 6.1 Variabili Locali

- `input_buff`: Buffer di dimensione `INPUT_BUFF_LEN` che conterrà i caratteri della riga letta dal file di input.
- `INPUT_BUFF_LEN`: Dimensione del buffer.

**Scelte Progettuali**

Si è scelto di utilizzare un buffer di dimensione 9 byte corrispondente alla lunghezza di una singola riga. Questa scelta è molto importante perché permette di non realizzare soluzioni hardcoded. La consegna specifica che non verranno utilizzati file di input con più di 100 righe: sfruttando tale informazione si sarebbe potuto scrivere un codice semplicistico che leggesse in un buffer l'intero file. Soluzione poco elegante e mal ottimizzata. Un'altra strada prendibile sarebbe stata quella di leggere un byte alla volta dal file di input, ma questa avrebbe richiesto un numero spropositato e inutile di chiamate al kernel. Con la lettura di una riga per volta il buffer rimane snello e comodo da manipolare. Questa soluzione permette all'applicazione di lavorare con file di dimensione arbitraria, inoltre lo riteniamo un metodo ottimale in quanto stabilisce un compromesso tra l'uso di `SYS_CALL` e flessibilità del codice. Oltre a questo si sarebbe potuto utilizzare un ulteriore metodo: quello di mappare in memoria il file ed accedervi come in un array, ma questo sforava dalle nozioni fornite dal corso.

### 6.2 Funzioni ed Etichette

- `_read_line`: Legge dal file una riga tramite la chiamata di sistema `read` e si occupa di richiedere la traduzione dei caratteri letti in interi mediante la funzione `_atoi` definita nel file `atoi.s` salvandoli successivamente nelle rispettive variabili globali. In caso i caratteri letti siano pari a 0 salta all'etichetta `_eof`;
- `_eof`: In caso di `EOF` mette -1 in `ebx` e ritorna.

## 7 atoi.s

Contiene la funzione che si occupa di convertire una serie di caratteri `ASCII` in un numero intero.

### 7.1 Funzioni ed Etichette

- `_atoi`: Vengono inizializzati i registri necessari alla conversione;
- `_atoi_loop`: Loop principale, converte la stringa puntata dal registro `edi` in un intero salvato e ritornato in `eax`.

## 8 check.s

Contiene la funzione che si occupa di settare sulla base dei valori di input e dei valori del ciclo precedente i corretti parametri delle variabili `alm, mod, numb`.

### 8.1 Funzioni ed Etichette

- `_check`: In base ai valori di `init` e `rpm` si occupa di saltare all'etichetta corretta.
- `_fg / _sg / _opt`: Etichette corrispondenti alle modalità di funzionamento previste dalle specifiche. Si occupano di settare i corretti valori di `alm, mod, numb`. L'unica modalità che necessita di una gestione particolare è `fg` in cui bisogna settare l'eventuale allarme.
- `_reset_numb`: Nel caso in cui sia stata cambiata la modalità di funzionamento oppure il valore della variabile `reset` sia pari a 1, viene settata la modalità corretta, resettato il conteggio `numb` e "spento" `alm`.
- `_set_alm`: Se il motore è nella modalità `fg` da più di 15 secondi, viene "acceso" l'allarme portando il valore di `alm` a 1.
- `_init_0`: Se il valore di `init` è pari a 0 tutte le variabili di output vengono poste a 0 dal momento che il motore è spento.
- `_end_check`: Abbiamo ritenuto opportuno (per evitare di preoccupare troppo il conducente) considerare un numero di secondi di massimo due cifre. Prima di terminare la funzione si controlla che il valore di `numb` non sia superiore a 99, in caso si salta a `_numb_overflow` che azzera `numb`.

**Scelte Progettuali**

Si è scelto di dividere il progetto in poche funzioni e di tenere tutti i controlli relativi a input/output in un'unica funzione chiamata `_check`. Sarebbe stato tranquillamente possibile fare una divisione in funzioni minori per aumentare la leggibilità (e.g. `check_rpm`, `check_init`, ...), ma è stato ritenuto più pratico, per la semplicità dei controlli, mantenere il codice in un unico file.

# 9 write_line.s

Contiene la funzione che si occupa di creare la stringa di output e scriverla sul corrispondente file. Ricordiamo qui la codifica utilizzata delle modalità di funzionamento e la struttura di una singola riga di output:

| | | |
|---|---|---|
| Spento | 0 | 00 |
| SG | 1 | 01 |
| OPT | 2 | 10 |
| FG | 3 | 11 |

| | | | | | |
|---|---|---|---|---|---|
| `alm` (1 byte) | , | `mod` (2 byte) | , | `numb` (2 byte) | `\n` |

## 9.1 Variabili Locali

- `output_buff`: Buffer di dimensione `OUTPUT_BUFF_LEN` che conterrà i caratteri della stringa da scrivere sul file di output.
- `OUTPUT_BUFF_LEN`: Dimensione del buffer.
- `MOD_XX`: Stringhe costanti che identificano la modalità di funzionamento corrispondente in binario.
- `MOD_LEN`: Dimensione delle stringhe di modalità.

## 9.2 Funzioni ed Etichette

- `_write_line`: Inizializza i registri necessari alla scrittura sulla variabile buffer.
- `_alm_X`: Aggiunge al buffer il corretto valore di `alm`.
- `_print_mod`: Aggiunge al buffer una virgola come separatore e in base al valore di `mod` salta alla corrispondente etichetta.
- `_mod_X`: La stringa corrispondente alla modalità X codificata in binario viene messa in `eax`, in seguito viene aggiunta al buffer nell'etichetta `_end_print_mod`.
- `_print_numb`: Si occupa di aggiungere al buffer il valore di `numb` opportunamente convertito in `ASCII` e di terminare la stringa con il carattere `\n`.

- `_numb_one_digit`: Se il valore di `numb` è minore di 10 si occupa di aggiungere uno 0 come prima cifra.

# 10 itoa.s

Contiene la funzione per convertire un valore da intero a una corrispondente stringa `ASCII`.

## 10.1 Funzioni ed Etichette

- `_itoa`: Inizializza i registri necessari alla conversione.
- `_itoa_dividi`: Si occupa di contare i caratteri necessari per la stringa e di posizionare il carattere `\0` alla fine della stringa.
- `_itoa_converti`: Scrive ogni cifra nella posizione corretta della stringa.

# 11 close_files.s

Contiene la funzione per chiudere correttamente un file.

# 12 Codice

## 12.1 syscall.inc

```
1   # file di definizione delle chiamate di sistema linux
2
3   .equ    SYS_EXIT, 1
4   .equ    SYS_READ, 3
5   .equ    SYS_WRITE, 4
6   .equ    SYS_OPEN, 5
7   .equ    SYS_CLOSE, 6
8
9   .equ    STDIN, 0
10  .equ    STDOUT, 1
11  .equ    STDERR, 2
12
13  .equ    SYSCALL, 0x80
14
```

## 12.2 main.s

```
1   # Progetto Assembly 2016
2   # File: main.s
3   # Autori: Noé Murr, Mirko Morati
4   #
5   # Descrizione: File principale, punto di inizio del programma.
6   .include    "syscall.inc"
7
8   .section    .data
9   input_fd:   .long 0         # variabile globale che conterrá il file
10  # descriptor del file di input
11
12  output_fd:  .long 0         # variabile globale che conterrá il file
13  # descriptor del file di output
14
15  # Variabili globali per i segnali di input
16  init:   .long 0
17  reset:  .long 0
18  rpm:    .long 0
19
20  # Variabili globali per i segnali di output
21  alm:    .long 0
22  numb:   .long 0
23  mod:    .long 0
24
25  # Codice del programma
26
27  .section    .text
28  .globl  input_fd
29  .globl  output_fd
30  .globl  init
31  .globl  reset
```

```
32          .globl  rpm
33          .globl  alm
34          .globl  numb
35          .globl  mod
36          .globl  _start
37
38          # Stringa per mostrare l'utilizzo del programma in caso di parametri errati
39          usage:  .asciz "usage: programName inputFilePath outputFilePath\n"
40          .equ    USAGE_LENGTH, .-usage
41
42          _start:
43          # Recupero i parametri del main
44          popl    %eax                # Numero parametri
45
46          # Controllo argomenti, se sbagliati mostro l'utilizzo corretto
47          cmpl    $3, %eax
48          jne     _show_usage
49
50          popl    %eax                # Nome programma
51          popl    %eax                # Primo parametro (nome file di input)
52          popl    %ebx                # Secondo parametro (nome file di output)
53
54          # NB: non salvo ebp in quanto non ha alcuna utilitá
55          # nella funzione start che comunque non ritorna
56
57          movl    %esp, %ebp
58
59          call    _open_files         # Apertura dei file
60
61          _main_loop:
62
63          call    _read_line          # Leggiamo la riga
64
65          cmpl    $-1, %ebx           # EOF se ebx == -1
66          je      _end
67
68          call    _check              # Controllo delle variabili
69
70          call    _write_line         # Scrittura delle variabili di output su file
71
72          jmp     _main_loop          # Leggi un altra riga finché non é EOF
73
74          _end:
75
76          call    _close_files        # Chiudi correttamente i file
77
78          # sys_exit(0);
79          movl    $SYS_EXIT, %eax
80          movl    $0, %ebx
81          int     $SYSCALL
82
83          _show_usage:
84          # esce in caso di errore con codice 1
```

```
85          # sys_write(stdout, usage, USAGE_LENGTH);
86          movl    $SYS_WRITE, %eax
87          movl    $STDOUT, %ebx
88          movl    $usage, %ecx
89          movl    $USAGE_LENGTH, %edx
90          int     $SYSCALL
91
92          # sys_exit(1);
93          movl    $SYS_EXIT, %eax
94          movl    $1, %ebx
95          int     $SYSCALL
96
97
```

## 12.3 open_files.s

```
1           # Progetto Assembly 2016
2           # File: open_files.s
3           # Autori: Noé Murr, Mirko Morati
4           #
5           # Descrizione: File contenente la funzione che si occupa di aprire i file
    di input e di
6           # output, i file descriptor vengono inseriti in variabili globali.
7           # Si suppone che il nome dei due file siano salvati negli indirizzi
    contenuti
8           # rispettivamente in %eax (input) ed in %ebx (output).
9
10          .include "syscall.inc"
11
12          .section .text
13
14          error_opening_files: .asciz "errore nell' apertura dei file\n"
15          .equ    ERROR_OPENING_LENGTH, .-error_opening_files
16
17          .globl  _open_files             # Dichiaro la funzione globale
18          .type   _open_files, @function  # Dichiaro l'etichetta come una funzione
19
20          _open_files:
21
22          pushl   %ebp
23          movl    %esp, %ebp
24
25          pushl   %ebx                    # Pusho l' indirizzo del file di output
26          # sullo stack
27
28          movl    %eax, %ebx              # Sposto l' indirizzo del file che vado
29          # ad aprire in %ebx
30
31          movl    $SYS_OPEN, %eax         # Chiamata di sistema open
32          movl    $0, %ecx                # read-only mode
33          int     $SYSCALL                # Apro il file
34
```

```
35          cmpl    $0, %eax
36          jl      _error_opening_files
37
38          movl    %eax, input_fd          # Metto il file descriptor nella
39          # sua variabile
40
41          popl    %ebx                    # Riprendo l' indirizzo del nome del file
42          # di output che avevo messo sullo stack
43
44          movl    $SYS_OPEN, %eax         # Chiamata di sistema open
45          movl    $01101, %ecx            # read and write mode
46          movl    $0666, %edx             # flags
47          int     $SYSCALL                # Apro il file
48
49          cmpl    $0, %eax
50          jl      _error_opening_files
51
52          movl    %eax, output_fd         # Metto il file descriptor nella
53          # sua variabile
54
55          movl    %ebp, %esp
56          popl    %ebp
57          ret                             # Ritorna al chiamante
58
59          _error_opening_files:
60          # Esce con codice di errore 2
61          # sys_write(stdout, usage, USAGE_LENGTH);
62          movl    $SYS_WRITE, %eax
63          movl    $STDERR, %ebx
64          movl    $error_opening_files, %ecx
65          movl    $ERROR_OPENING_LENGTH, %edx
66          int     $SYSCALL
67
68          # sys_exit(2);
69          movl    $SYS_EXIT, %eax
70          movl    $2, %ebx
71          int     $SYSCALL
72
```

## 12.4 read_line.s

```
1           # Progetto Assembly 2016
2           # File: read_line.s
3           # Autori: Noé Murr, Mirko Morati
4           #
5           # Descrizione: Funzione che legge una riga alla volta del file di input.
6
7           .include "syscall.inc"
8
9           .section .bss
10          .equ    INPUT_BUFF_LEN, 9
11          input_buff: .space INPUT_BUFF_LEN   # Input buffer di 9 byte
```

```
12
13      .section .text
14      .globl  _read_line
15      .type   _read_line, @function
16
17      _read_line:
18      pushl   %ebp
19      movl    %esp, %ebp
20
21      # Lettura riga
22      # sys_read(input_fd, input_buff, INPUT_BUFF_LEN);
23      movl    input_fd, %ebx
24      movl    $SYS_READ, %eax
25      leal    input_buff, %ecx
26      movl    $INPUT_BUFF_LEN, %edx
27      int     $SYSCALL
28
29      cmpl    $0, %eax                    # Se eax == 0 EOF
30      je      _eof
31
32      # Estrazione dei valori di init, reset, rpm dal buffer
33      leal    input_buff, %edi
34      call    _atoi
35      movl    %eax, init
36
37      incl    %edi                        # Salto il carattere ','
38
39      call    _atoi
40      movl    %eax, reset
41
42      incl    %edi                        # Salto il carattere ','
43
44      call    _atoi
45      movl    %eax, rpm
46
47      movl    %ebp, %esp
48      popl    %ebp
49
50      xorl    %ebx, %ebx                  # ebx = 0 permette di proseguire
51      ret
52
53      _eof:
54      # in caso di EOF %ebx = -1
55      movl    %ebp, %esp
56      popl    %ebp
57
58      movl    $-1, %ebx
59      ret
60
```

## 12.5 atoi.s

```
1         # Progetto Assembly 2016
2         # File: atoi.s
3         # Autori: Alessandro Righi, Noé Murr, Mirko Morati
4         #
5         # Descrizione: Funzione che converte una stringa in intero.
6
7         .section .text
8         .globl  _atoi
9         .type   _atoi, @function
10
11        # Funzione che converte una stringa di input in numero
12        # Prototipo C-style:
13        #   uint32_t atoi(const char *string);
14        # Parametri di input:
15        #   EDI - Stringa da convertire
16        # Parametri di output:
17        #   EAX - Valore convertito
18
19        _atoi:
20        xorl    %eax, %eax   # azzero il registro EAX per contenere il risultato
21        xorl    %ebx, %ebx   # azzero EBX
22        movl    $10, %ecx    # sposto 10 in ECX che conterrá il valore
    moltiplicativo
23
24        _atoi_loop:
25        xorl    %ebx, %ebx
26        movb    (%edi), %bl  # sposto un byte dalla stringa in BL
27        subb    $48, %bl     # sottraggo il valore ASCII dello 0 a BL
28        # per avere un valore intero
29
30        cmpb    $0, %bl      # Se il numero é minore di 0
31        jl      _atoi_end    # allora esco dal ciclo
32        cmpb    $10, %bl     # Se il numero é maggiore o uguale a 10
33        jge     _atoi_end    # esco dal ciclo
34
35        mull    %ecx         # altrimenti moltiplico EAX per 10
36        # (10 messo precedentemente in ECX)
37        addl    %ebx, %eax   # aggiungo a EAX il valore attuale
38        incl    %edi         # incremento EDI
39
40        jmp     _atoi_loop   # rieseguo il ciclo
41
42        _atoi_end:
43        ret
44
```

## 12.6 check.s

```
1         # Progetto Assembly 2016
2         # File: check.s
3         # Autori: Noé Murr, Mirko Morati
4         #
5         # Descrizione: Funzione che controlla le variabili
6         # init, reset, rpm e setta le variabili alm, mod e numb
7
8         .section .data
9
10        .section .text
11        .globl  _check
12        .type   _check, @function
13
14        _check:
15        pushl   %ebp
16        movl    %esp, %ebp
17
18        # Caso init == 0: alm = 0; mod = 0; numb = 0;
19        cmpl    $0, init
20        je      _init_0
21
22        # Caso SG: alm = 0; mod = 1; numb = reset == 1 ? 0 : numb + 1;
23        cmpl    $2000, rpm
24        jl      _sg
25
26        # Caso OPT: alm = 0; mod = 2; numb = reset == 1 ? 0 : numb + 1;
27        cmpl    $4000, rpm
28        jle     _opt
29
30        # Caso FG: alm = numb >= 15? 1 : 0; mod = 3; numb = reset == 1 ? 0 : numb +
      1;
31        _fg:
32        # Salviamo la nuova modalita' in %eax e controlliamo reset
33        movl    $3, %eax
34        cmpl    $1, reset
35        je      _reset_numb
36
37        # Se la nuova modalitá non é la stessa si resetta il numero di secondi
38        cmpl    $3, mod
39        jne     _reset_numb
40
41        incl    numb
42        movl    %eax, mod
43
44        # Se il numero di secondi é maggiore o uguale a 15 viene alzata l'allarme
45        cmpl    $15, numb
46        jge      _set_alm
47
48        jmp     _end_check
49
50        _opt:
```

```
51          movl    $2, %eax
52          cmpl    $1, reset
53          je      _reset_numb
54
55          cmpl    $2, mod
56          jne     _reset_numb
57
58          incl    numb
59          movl    %eax, mod
60
61          jmp     _end_check
62
63          _sg:
64          movl    $1, %eax
65          cmpl    $1, reset
66          je      _reset_numb
67
68          cmpl    $1, mod
69          jne      _reset_numb
70
71          incl    numb
72          movl    %eax, mod
73
74          jmp     _end_check
75
76          _reset_numb:
77          movl    %eax, mod
78          movl    $0, numb
79          movl    $0, alm
80
81          jmp     _end_check
82
83          _set_alm:
84          movl    $1, alm
85
86
87          jmp     _end_check
88
89          _init_0:
90          movl    $0, alm
91          movl    $0, numb
92          movl    $0, mod
93
94          _end_check:
95
96          # Se il numero di secondi supera i 99 allora dobbiamo ricominciare il
        conteggio
97          cmpl    $99, numb
98          jg      _numb_overflow
99          movl    %ebp, %esp
100         popl    %ebp
101
102         ret
```

```
103
104         _numb_overflow:
105         movl    $0, numb
106         jmp     _end_check
107
```

## 12.7 write_line.s

```
1         # Progetto Assembly 2016
2         # File: check.s
3         # Autori: Noé Murr, Mirko Morati
4         #
5         # Descrizione: Funzione che scrive una riga alla volta nel file di output
6
7         .include "syscall.inc"
8
9         .section .bss
10        .equ    OUTPUT_BUFF_LEN, 8
11        output_buff: .space OUTPUT_BUFF_LEN
12
13        .section .text
14        .globl  _write_line
15        .type   _write_line, @function
16        MOD_00: .ascii "00"           # motore spento
17        MOD_01: .ascii "01"           # motore sotto giri
18        MOD_10: .ascii "10"           # motore in stato ottimale
19        MOD_11: .ascii "11"           # motore fuori giri
20        .equ    MOD_LEN, 2
21
22        _write_line:
23        pushl   %ebp
24        movl    %esp, %ebp
25
26        leal    output_buff, %edi     # spostiamo il puntatore
27        # del buffer di output in EDI
28
29        cmpl    $1, alm               # se l'allarme é 1 stampiamo 1
30        # altrimenti 0 senza chiamare funzioni
31        je      _alm_1
32
33        _alm_0:
34        movl    $48, (%edi)
35        jmp     _print_mod
36
37        _alm_1:
38        movl    $49, (%edi)
39
40        _print_mod:
41        movl    $44, 1(%edi)          # aggiungiamo la virgola dopo
42        # il segnale di allarme
43        addl    $2, %edi              # spostiamo un immaginario cursore
44        # nella posizione dove stampare la mod
```

```
45
46          cmpl    $1, mod             # controlliamo il valore di mod
47          # e stampiamo la stringa corretta in base
48          # alla giusta modalita' di funzionamento
49          je      _mod_1
50          cmpl    $2, mod
51          je      _mod_2
52          cmpl    $3, mod
53          je      _mod_3
54
55          _mod_0:
56          movl    MOD_00, %eax
57          jmp _end_print_mod
58
59          _mod_1:
60          movl    MOD_01, %eax
61          jmp _end_print_mod
62
63          _mod_2:
64          movl    MOD_10, %eax
65          jmp _end_print_mod
66
67          _mod_3:
68          movl    MOD_11, %eax
69
70          _end_print_mod:
71          movl    %eax, (%edi)        # mettiamo la stringa nell' output_buff
72          addl    $MOD_LEN, %edi      # spostato il cursore (la posizione di edi)
73          # nel punto esatto dove scrivere
74          movl    $44, (%edi)         # aggiungiamo la virgola
75          incl    %edi                # spostiamo il cursore
76
77          cmpl    $10, numb           # controlliamo se il numero di secondi
78          # é ad una sola cifra, in tal caso
79          # aggiungiamola cifra 0
80          jl      _numb_one_digit
81
82          _print_numb:
83          movl    numb, %eax          # prepariamo la chiamata per itoa
84
85          call    _itoa               # chiamiamo itoa
86
87
88          leal    output_buff, %edi   # mettiamo il puntatore di output_buff in edi
89          addl    $7, %edi            # ci aggiungiamo 7 per arrivare
90          # alla fine della stringa,
91          movl    $10, (%edi)         # punto nel quale aggiungiamo un \n
92
93          movl    $SYS_WRITE, %eax
94          movl    output_fd, %ebx
95          leal    output_buff, %ecx
96          movl    $OUTPUT_BUFF_LEN, %edx
97          int     $SYSCALL
```

```
98
99
100         movl    %ebp, %esp
101         popl    %ebp
102
103         ret
104
105         _numb_one_digit:
106         movl    $48, (%edi)
107         incl    %edi
108         jmp     _print_numb
109
```

## 12.8 itoa.s

```
1           # Progetto Assembly 2016
2           # File: itoa.s
3           # Autori: Alessandro Righi, Noé Murr, Mirko Morati
4           #
5           # Descrizione: Funzione che converte un intero in stringa
6           # Prototipo C-style:
7           #   u_int32_t itoa(uint32_t val, char *string);
8           # Parametri di input:
9           #   EAX - Valore intero unsigned a 64bit da convertire
10          #   EDI - Puntatore alla stringa su cui salvare il risultato
11          # Parametri di output:
12          #   EAX - Lunghezza della stringa convertita (compresiva di \0 finale)
13
14          .section .text
15          .global _itoa
16          .type   _itoa, @function
17
18          _itoa:
19          movl    $10, %ecx   # porto il fattore moltiplicativo in ECX
20          movl    %eax, %ebx  # salvo temporaneamente il valore di EAX in EBX
21          xorl    %esi, %esi  # azzero il registro ESI
22
23          _itoa_dividi:
24          xorl    %edx, %edx  # azzero EDX per fare la divisione
25          divl    %ecx        # divide EAX per ECX, salva il resto in EDX
26          incl    %esi        # incrementa il contatore
27          testl   %eax, %eax  # se il valore di EAX non é zero ripeti il ciclo
28          jnz     _itoa_dividi
29
30          addl    %esi, %edi  # somma all'indirizzo del buffer
31          # il numero di caratteri del numero
32          movl    %ebx, %eax  # rimette il valore da convertire in EAX
33          movl    %esi, %ebx  # salvo il valore della lunghezza della stringa in EBX
34
35          movl    $0, (%edi)  # aggiungo un null terminator alla fine della stringa
36          decl    %edi        # decremento il contatore della stringa di 1
37
```

```
38          _itoa_converti:
39          xorl    %edx, %edx  # azzero EDX per fare la divisione
40          divl    %ecx        # divido EAX per ECX, salvo il valore del resto in EDX
41          addl    $48, %edx   # sommo 48 a EDX
42          movb    %dl, (%edi) # sposto il byte inferiore di EDX (DL)
43          # nella locazione di memoria puntata da EDI
44          decl    %edi        # decremento il puntatore della stringa
45          decl    %esi        # decremento il contatore
46          testl   %esi, %esi  # se il contatore non é 0 continua ad eseguire il loop
47          jnz     _itoa_converti
48
49          movl    %ebx, %eax  # porto il valore della lunghezza
50          # della stringa in EAX per ritornarlo
51          incl    %eax        # incremento di 1 EAX (in modo da includere il \0)
52          ret
53
```

## 12.9 close_files.s

```
1           # Progetto Assembly 2016
2           # File: check.s
3           # Autori: Noé Murr, Mirko Morati
4           #
5           # Descrizione: Funzione che chiude i file aperti precedentemente
6
7           .include "syscall.inc"
8
9           .section .text
10          .globl  _close_files            # Dichiaro la funzione globale
11          .type   _close_files, @function  # Dichiaro l' etichetta come una funzione
12
13          _close_files:
14
15          pushl   %ebp
16          movl    %esp, %ebp
17
18          # sys_close(input_fd);
19          movl    $SYS_CLOSE, %eax
20          movl    input_fd, %ebx
21          int     $SYSCALL
22
23          # sys_close(output_fd);
24          movl    $SYS_CLOSE, %eax
25          movl    output_fd, %ebx
26          int     $SYSCALL
27
28
29          movl    %ebp, %esp
30          popl    %ebp
31
32          ret
33
```