Relazione elaborato del corso di Architettura degli elaboratori

Progetto assembly

Anno accademico 2019/2020

Studenti:

Alessandro Marconcini (matricola VR421504)

Enrico Pachera (matricola VR422511)

SOMMARIO:

Introduzione:

Lo scopo del progetto consisteva nella realizzazione di un parcheggio gestito in modo autonomo. Il sistema è composto da tre settori di posti auto che sono stati chiamati A, B e C. I primi due possiedono un massimo di 31 posti occupabili e l'ultimo ne possiede solamente 24.

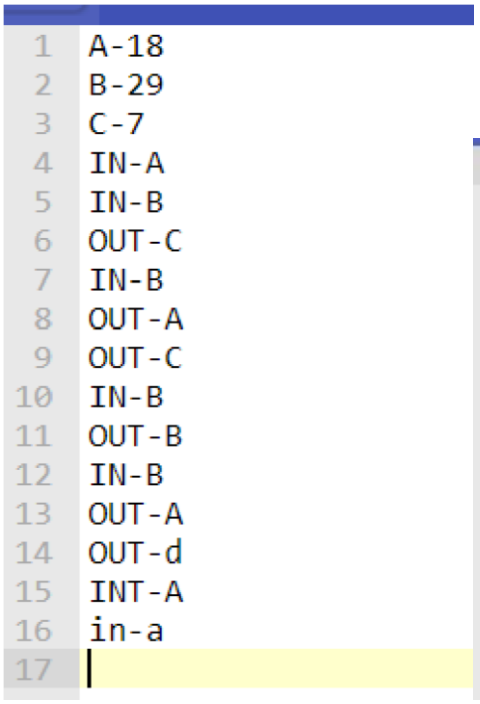
Oltre ai settori il sistema possiede due sbarre, che quando chiuse non lasciano passare le auto. Queste sbarre sono rispettivamente una in entrata e una in uscita e vengono alzate o tenute basse in base a determinate situazioni.

Durante la notte il sistema viene lasciato spento e le sbarre rimangono alzate tutto il tempo permettendo il libero circolo dei veicoli. Allo stesso modo non viene tenuto conto di quante auto siano in sosta nei determinati settori durante questa fase.

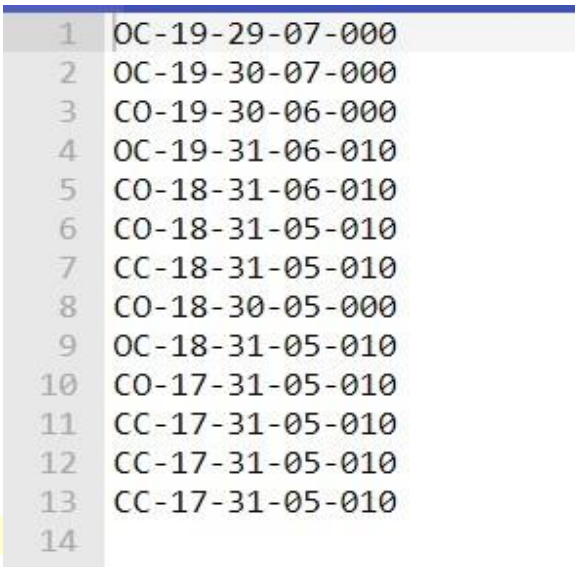
Nel momento in cui un operatore accende il dispositivo, egli inserisce manualmente il numero di auto che stanno utilizzando i posti nei vari settori rispettivamente. L'inserimento di uno o più dati viene simulato tramite la lettura di un file testin.txt, un file di testo che contiene il numero di auto presenti nei vari settori e l'ordine sequenziale in cui N utenti cercano di entrare o uscire dal parcheggio.

Ad ogni riga del file, dopo le tre righe relative agli inserimenti, corrisponde un'azione, entrare o uscire che sia. Nel momento in cui un utente tenta di eseguire una determinata azione un file testout.txt viene aggiornato con le seguenti informazioni scritte al suo interno: se le due sbarre sono state aperte, quante auto ci sono in quali settori,quali settori sono pieni.

Il sistema è stato realizzato gestendo anche alcune possibili casistiche di errore. Nel momento in cui un utente si sbaglia (viene letta una stringa corrotta) e genera un'anomalia, il sistema risponde in due modalità: se l'errore è sull'inserimento dei valori dei settori allora spegne il dispositivo notificando l'errore (l'operatore dovrà semplicemente riavviare il sistema da capo), se invece l'errore è una stringa corrotta prodotta da un utente in entrata o uscita il sistema non apre nessuna sbarra ma consente di poter svolgere ancora azioni per accedere al parcheggio.



Esempio file testin.txt



Esempio file testout.txt

Da C ad assembly:

Se analizziamo il codice dell'elaborato si può notare come esso sia composto da una parte di codice C e una parte di codice assembly inserita all'interno della precedente con lo scopo di ottimizzare il dispositivo.

Tramite l'esecuzione di parking è possibile infatti ottenere stampati a video due valori temporali misurati in nanosecondi, quello relativo all' esecuzione del codice C, che se implementato mostra un'enorme differenza col successivo e quello relativo all'esecuzione del codice asm. Per fare un esempio mandando in esecuzione il codice asm ci mette al massimo poco più di 6000 ns per eseguire la sua parte ,mentre la stampa [printf("%s", bufferout_asm);] della stringa di output a video impiega più di 150.000 ns quindi si può supporre che lo stesso programma scritto in C possa impiegarci maggiormente.

Per poter passare dalla parte di C alla parte di assembly è stata utilizzata la chiamata a una funzione denominata parking_asm alla quale sono stati passati due parametri, bufferin e bufferout. Bufferin è la stringa di input che viene letta dal file testin.txt, bufferout invece è la stringa dove vengono inseriti i caratteri che vengono stampati nel file testout.txt.

Inserimento della funzione

```
8
9
10  /* Inserite eventuali extern modules qui */
11
12  extern void parking_asm(char bufferin[],char bufferout_asm[]);
13
```

Richiamo della funzione con il passaggio dei parametri effettivi

```
83
84      tic_asm = current_timestamp();
85
86      /* Assembly inline:
87      Inserite qui il vostro blocco di codice assembly inline o richiamo a funzioni assembly.
88      Il blocco di codice prende come input 'bufferin' e deve restituire una variabile stringa
        'bufferout_asm' che verrà poi salvata su file. */
89
90      parking_asm(bufferin,bufferout_asm);
91
92      toc_asm = current_timestamp();
```

Descrizione delle variabili:

In questo elaborato non è stato utilizzata la sezione bss, destinata alle variabili non inizializzate ma bensì solamente la sezione data .

```
.section .data

### STRINGHE DI ERRORE ###

str_settore: .ascii "Errore nell'inizializzazione di uno dei settori! Rivedi il file di testo\n"  # Stringa di errore per i settori
str_settore_len: .long .-str_settore  #Lunghezza della stringa str_settore

### ALTRI VALORI ###

NPOSTIA: .long 0
NPOSTIB: .long 0
NPOSTIC: .long 0
valori_inseriti: .long 0 # Tiene conto se tutti e 3 i valori sono stati inseriti
SBARRE:  .long 0
LIGHTS: .long 0
ecx_tmp: .long 0
cont: .long 0
```

str_settore: contiene la stringa che viene stampata a video in caso si presenti un errore nell'inserimento dei valori riguardanti i settori del parcheggio;

str_settore_len: contiene il numero di caratteri presenti nella stringa str_settore, utile per la stampa a video;

NPOSTIA: rappresenta il quantitativo di posti occupati dalle auto nel settore A;

NPOSTIB: rappresenta il quantitativo di posti occupati dalle auto nel settore B;

NPOSTIC: rappresenta il quantitativo di posti occupati dalle auto nel settore C;

valori_inseriti: viene utilizzata per determinare quanti e quali settori sono stati inizializzati e quanti e quali no (100->solo A,10->solo B,1->solo C,110->AB,011->BC,101->AC e 111->tutti); SBARRE: contiene due cifre numeriche che identificano se le sbarre in ingresso e in uscita sono state aperte oppure no (nel file testout.txt C significa closed e O significa opened).

LIGHTS: utilizzata per determinare quali settori sono pieni o meno (esempio se A ha 31 posti occupati allora il valore diventa 100 ,se B diventa 010 e per C 001 e via così per tutte le combinazioni);

ecx_tmp: contiene il valore del registro ecx, che consente di puntare al registro edi per scrivere nelle varie celle di bufferout_asm

cont: contatore che serve per indicare se esso è zero oppure uno e viene utilizzato per far verificare l'ultimo ciclo di stampa al sistema

Descrizioni delle funzioni:

Il codice può essere suddiviso in una prima parte che si occupa di leggere i primi caratteri della stringa di bufferin, con lo scopo di immettere i valori dei vari settori del parcheggio e una seconda parte che invece legge riga per riga la stringa nelle parti corrispondenti alle azioni di entrata oppure uscita dal parcheggi. In quest' ultima vengono considerate delle richieste di entrata e uscita poichè non sempre esse sono soddisfabili.

Sezione text e preparazione dei registri

```
.section .text

    .global parking_asm

parking_asm:

    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %esi  #bufferin
    movl 12(%ebp) ,%edi #bufferout_asm

    movl $0,%ecx
    movl $0, %eax
    movl $0, %ebx
    movl $0, %edx
```

La sezione text è la vera e propria sezione dove si trova tutto il codice assembly. parking_asm è la prima etichetta che si può trovare ed essa denomina l'inizio del corpo della funzione, da essa partono le prime righe di codice.

Per prima cosa il contenuto d ebp viene messo sullo stack ed esp (stack pointer) viene posizionato sul frame che è stato

appena occupato, questo per evitare i poblemi derivanti da eventuali chiamate nidificate (Il valore verrà rimesso in ebp alla fine).

Lo stack è una pila di frame composti da 4 byte ciascuno ed essa cresce verso il basso (Caso architettura 32 bit). Nel momento in cui è stata chiamata la funzione in C, lo stack si è riempito posizionando in ordine inverso i parametri con cui è stata chiamata.

Per recuperare i due parametri bisogna andare alla ricerca di questi risalendo la pila,in particolare di due frame per bufferin (Risalgo di 8) e tre per bufferout_asm (Risalgo di 12). Visto che i due buffer contengono entrambi stringhe, per essere lette e scritte è necessario che i loro puntatori vengano salvati all'interno di due registri, utilizzati solitamente per queste operazioni, edi ed esi, che poi verranno utilizzati più avanti.

Infine in questa prima parte vengono azzerati tutti i registri che verranno utilizzati maggiormente, per evitare che per qualche valore che contengono ancora dal precedente codice C vengano a crearsi problematiche difficili da notare.

Prima_lettura

```
prima_lettura:

    call leggi_lettera # Questa funzione legge lettera+trattino della stringa e poi mette il risultato in eax

    inc %ecx #Salto il trattino

    pushl %eax # 2 PUSH <--- metto eax sullo stack per utilizzarla dopo, prima necessito di un valore numerico

    call leggi_valore # Questa funzione legge ogni cifra numerica fino ad arrivare al carattere "\n"
                      # Il valore di ritorno è in eax
```

Immediatamente successiva all'azzeramento vi è l'etichetta leggi, che effettua una lettura ciclica di alcuni caratteri presenti nella stringa di bufferin che non ci interessano (Anche se non sono presenti nel file di testo essi vengono comunque letti).

Il codice avanza nel momento in cui si giunge alla prima lettera tra A,B o C rigorosamente maiuscole.

Nel codice di prima_lettura vengono chiamate due sottofunzioni assembly, leggi_lettera e leggi_valore, fondamentali per comporre una lettura completa di una stringa di tipo X-NM dove X è il settore ed NM le due cifre numeriche da memorizzare nel settore (potrebbe essere una singola cifra, in quel caso è 0M).

Ognuna delle due funzioni salva il risultato in eax.

Leggi lettera

```
.type leggi_lettera, @function

leggi_lettera:
    movl $0, %ebx

lettura_settore:

    movb (%esi,%ecx,1), %bl
    # Faccio un doppio check #
    cmpb $65, %bl
    jge lettera

    # Altrimenti#
    jmp errore_inserimento_settori

errore_inserimento_settori:

    movl $4, %eax
    movl $1, %ebx
    leal str_settore, %ecx
    movl str_settore_len, %edx
    int $0x80

    movl $1, %eax
    movl $1, %ebx  # <--- EXIT_FAILURE
    int $0x80

lettera:

    movb %bl, %al

    cmpb $68,%al
    jl lettera_return

    jmp errore_inserimento_settori

lettera_return:

    incl %ecx
    Ret # Ritorno al main
```

Leggi valore

```
leggi_valore:

    xorl %eax, %eax #Azzero eax
    xorl %ebx, %ebx
    movb (%ecx,%esi,1), %bl

    # Check se è un capo riga #
    incl %ecx

    cmpb $10, %bl  # Controllo sullo \n
    je fine_lettura_valore

    subl $48,%ebx

    # Check che sia un numero #

    cmpb $0 ,%bl  # Se è una cifra decimale è compresa tra 0 e 10 con 10 non compreso

    jge forse_numero
    jmp errore_inserimento_settori

forse_numero:

    cmpb $10, %bl

    jl numero
    jmp errore_inserimento_settori

numero:

    pushl %ebx # Metto la cifra sullo stack
    jmp leggi_valore

fine_lettura_valore:

    popl %ebx
    cmp $65,%ebx

    jl cifra

    #Altrimenti
    pushl %ebx
    Ret
```

```
cifra:

    cmp $0,%al
    jne lascia_in_ebx

    movl %ebx ,%eax

    jmp fine_lettura_valore

    #Risultato in eax
    Ret

lascia_in_ebx: # Se sono qui significa che invece di una ho due cifre

    movl %ebx,%edx
    movl %eax,%ebx
    movl %edx, %eax

    movl $10, %edx
    mulb %dl

    addl %ebx, %eax

    jmp fine_lettura_valore
```

In lettura_settore e lettera vengono svolti due check condizionali per controllare che il valore ottenuto dalla lettura sia compreso tra 65 e 67, numeri decimali che tramite la tabella ascii codificano la A e la C.

Nel caso in cui il valore ottenuto non è quello cercato significa allora che non sapremo mai quale sia i valore di un determinato settore e per questo tutto il file testout.txt conterrà informazioni errate. Per far fronte a ciò è stato deciso di portare il programma verso una exit(1), una system call che porta il programma ad essere terminato. Il valore 1 identifica una terminazione di tipo EXIT_FAILURE, ovvero con errore. Poco prima però avviene la stampa di una stringa che informa l'operatore di aver inserito informazioni errate che non si possono utilizzare. Questa scelta progettuale al massimo può creare il problema di dover reinserire i valori da capo (Nel nostro caso bisogna modificare il file testin.txt).

In leggi_valore viene ripresa la strategia del doppio check per vericare se il valore assunto in bl sia un numero o meno, se lo è viene messo sullo stack, per poi essere recuperato successivamente. Ad ogni ciclo di lettura

del numero corrisponde la lettura di una singola cifra. Questa funzione è stata realizzata con lo scopo di leggere un numero di una o due cifre al massimo poichè i settori non arrivano mai alla terza cifra di conteggio.

Quando si arriva alla seconda cifra,essa viene momentaneamente messa sullo stack per poi essere ripresa e messa in ebx. L'altra si troverà già in al.

Inizialmente si era tentato di utilizzare le componenti al e ah dello stesso registro eax, tuttavia questo non è stato possibile poichè generavano un valore indesiderato attraverso la moltiplicazione. Utilizzare due registri differenti è risultata una scelta più adeguata.

La seconda cifra infatti viene moltiplicata per 10 e sommata alla prima per ottenere effettivamente il valore numerico di interesse da assegnare alla variabile relativa al settore.

Il risultato è ancora una volta messo in eax.

Check per l'assegnamento

```
# Eseguo il check di che lettera si tratta #

popl %ebx # Ritorno a 1 PUSH eseguiti (rimane solo ebp che abbiamo salvato sullo stack)

cmp $65, %ebx
je assegnamento_A
cmp $66, %ebx
je assegnamento_B
cmp $67, %ebx
je assegnamento_C

jmp errore_inserimento_settori
```

La lettera che si ottiene dalla funzione leggi_lettera viene messa sullo stack per poter operare sui valori numerici ma successivamente viene recuperata per assegnare il valore nel settore ad essa dedicato.

Assegnamento di A

```
assegnamento_A:

    cmp $31, %eax #Check se il settore può ospitare il numero di auto inserite
    jg assegnamento_max_A

    movl %eax,(NPOSTIA)
    movl (valori_inseriti), %ebx
    addl $100, %ebx
    movl %ebx, (valori_inseriti)
    jmp post_assegnamento

assegnamento_max_A:

    movl $31, (NPOSTIA)
    movl (valori_inseriti), %ebx
    addl $100, %ebx
    movl %ebx, (valori_inseriti)
    jmp post_assegnamento
```

Una volta deciso in quale settore va effettuato l'assegnamento, viene effettuato un controllo sulla quantità che sta per essere immessa nella variabile e, se essa è superiore al valore massimo, viene settata al valore massimo, perciò in caso di inserimento esagerato il settore viene considerato completamente occupato.Se questa situazione non avviene e l'inserimento è minore del massimo valore di capienza, viene semplicemente aggiornata la variabile.

```
post_assegnamento:

    movl (valori_inseriti), %ebx
    cmp $111, %ebx

    je fine_inserimenti #Ho inserito tutti e 3 i valori dei registri

    jmp prima_lettura # Ricomincio a leggere da capo il prossimo valore
```

Nel momento in cui l'assegnamento avviene, anche valori_inseriti viene aggiornata, con lo scopo di capire quale e quanti settori sono stati settati. Nel caso del settaggio doppio di un settore esso è consentito, ma valori_inseriti non raggiungerà mai il valore necessario per proseguire nel codice (111, 100 da A, 10 da B e 1 da C), percui tenterà di leggere nuovamente una lettera andando in errore.

Se invece gli inserimento avvengono in modo giusto il codice fa partire il funzionamento autonomo del parcheggio, dove l'operatore non dovrà più inserire nulla ,ma sarà compito degli utenti che usufruiscono del dispositivo.

Fine inserimenti

```
fine_inserimenti:

    # Arrivato qui mi devo iniziare a comportare in modo diverso

    movb (%esi,%ecx,1), %bl
    incl %ecx

    cmpb $73, %bl  # se è I
    je i

    cmpb $79, %bl # se è O
    je o

    jmp errore_entrata_uscita
```

Qui inizia il funzionamento autonomo del dispositivo, dove legge le righe contenenti le richieste di entrata e uscita dell'utente.

La riga viene trattata lettera per lettera per formare le formule IN oppure OUT.

```
i:
    movb (%esi,%ecx,1), %bl # Prelevo una ipotetica N
    incl %ecx

    cmpb $78, %bl # se è IN
    je in

    jmp errore_entrata_uscita

in:
    incl %ecx # Salto il trattino

    movb (%esi,%ecx,1), %bl # Prelevo la lettera del settore da incrementare
    incl %ecx

    cmpb $65, %bl
    je in_A

    cmpb $66, %bl
    je in_B

    cmpb $67, %bl
    je in_C

    jmp errore_entrata_uscita
```

```
o:

    movb (%esi,%ecx,1), %bl # ipotetico U
    incl %ecx
    cmpb $85, %bl
    je ou

    jmp errore_entrata_uscita

ou:
    movb (%esi,%ecx,1), %bl # ipotetico T
    incl %ecx
    cmpb $84, %bl
    je out

    jmp errore_entrata_uscita

out:

    incl %ecx # Salto il trattino

    movb (%esi,%ecx,1), %bl # mi aspetto A,B o C
    incl %ecx

    cmpb $65, %bl
    je out_A

    cmpb $66, %bl
    je out_B

    cmpb $67, %bl
    je out_C

    jmp errore_entrata_uscita
```

Se per caso nessuna delle due tipologie di formule non viene raggiunta, allora quella situazione viene trattata come errore_entrata_uscita.

Errore entrata/uscita

```
errore_entrata_uscita:

    movb $22, (SBARRE) #CC

    cmpb $10, %bl
    je fine_scorrimento_errore

    incl %ecx
    movb (%esi,%ecx,1), %bl
    jmp errore_entrata_uscita

fine_scorrimento_errore:

    incl %ecx
    jmp stampa
```

Se ciò avviene allora la riga viene letta fino al carattere di capo riga (\n) in quanto essa viene considerata una stringa corrotta, per poi passare alla seguente. Si può notare come il valore della variabile SBARRE venga settato come 22, valore che diviene utile nella stampa.

Incremento del settore A

```
in_A:

    movl (NPOSTIA), %eax
    incl %eax
    incl %ecx # voglio saltare il \n

    cmp $31, %eax
    jg A_pieno

    movl %eax, (NPOSTIA)
    movb $12, (SBARRE) #OC
    jmp stampa

A_pieno:

    movl $31, %eax
    movl %eax,(NPOSTIA)
    movb $22, (SBARRE) #CC
    jmp stampa
```

Qui sopra è riportato il codice riguardante le possibilità di modifica del settore per quanto riguarda A. Il dispositivo funziona in maniera affine per B e per C andando a cambiare per quest'ultima il valore di check per capire se il suo settore è pieno, da 31 a 24.

Infatti A potrebbe essere libero, pieno oppure divenirlo al momento, ciò che cambia è il comportamento delle sbarre poichè nel primo caso e nel terzo si apre quella in entrata,quella in uscita no,nel secondo rimangono chiuse entrambe.

Decremento del settore A

```
out_A:

    movl (NPOSTIA), %eax
    incl %ecx # Salto il carattere \n

    cmp $0, %eax # Se il settore è vuoto non avviene il decremento
    je not_out_A

    subl $1, %eax
    movl %eax ,(NPOSTIA)
    movb $21, (SBARRE) # CO
    jmp stampa

not_out_A:

    movb $22, (SBARRE) #CC
    movl %eax, (NPOSTIA)
    jmp stampa
```

Per ogni settore,come nel caso di entrata,l'uscita viene trattata in due modi: se il settore si dimostra vuoto allora nessuno potrà uscire (in una situazione reale è anche impossibile che accada) e quindi la sbarra di uscita non si aprirà, altrimenti avviene il decremento del contatore del numero di posti occupati.

Anche in questo caso vengono assegnati dei valori alla variabile SBARRE che vengono utilizzati nella stampa successivamente.

Stampa

```
stampa:

    cmpb $0 ,%bl

    je ultima_stampa

    jmp stampa_pt1

ultima_stampa:

    addl $1,(cont)
    jmp stampa_pt1

stampa_pt1:

    # Valori di LIGTHS #

    movl $0, (LIGTHS)
    movl (NPOSTIA), %eax

    cmp $31, %eax

    jge add_100
```

La funzione stampa è quella più articolata. Essa si compone di diverse parti che hanno lo scopo di comporre la stringa finale in bufferout_asm.

Per prima cosa si occupa di capire se è la stampa riguardante l'ultima riga, in particolare l'etichetta ultima_stampa aggiorna la variabile cont, necessaria per la stampa della riga finale(è stata aggiunta in seguito all'accorgimento del fatto che non venisse stampata nel file l'ultima stringa).

Una volta effettuato il controllo se cont è diverso da 1 allora si passa ad aggiornare il valore di LIGTHS, che contiene il dato dei settori pieni (ad esempio se vale 100 è pieno A, 010 B, 110 A e B, 001 C e così via). Inizialmente la variabile è posta a zero, ma poi viene aggiornata tramite delle somme di valori in base alle circostanze.

Inizia stampa

```
inizia_stampa:

    pushl %ecx # Metto momentaneamente il contatore di bufferin sullo stack

    movl (ecx_tmp), %ecx

    # Considero bufferout_asm che si trova in edi #

    # SBARRE #

    movl (SBARRE), %eax

    cmp $11, %eax
    je oo
    cmp $12 ,%eax
    je oc
    cmp $21, %eax
    je co
    cmp $22, %eax
    je cc
    jmp errore_entrata_uscita
```

In questa parte ecx, contatore che fino a questo momento si è spostato per andare a leggere casella per casella il contenuto della stringa bufferin, viene messo momentaneamente sullo stack per poter andare a lavorare su bufferout_asm e quindi utilizzare il suo contatore.

ecx_tmp contiene solo il contatore relativo a bufferout_asm, così è possibile recuperare tale valore ad ogni ciclo di stampa.

Entrambe sbarre aperte

```
oo:
    movl $79, %ebx
    movb %bl, (%ecx,%edi,1)
    inc %ecx
    movb %bl, (%ecx,%edi,1)
    inc %ecx

    jmp dopo_SBARRE
```

Nell'immagine qui sopra si può vedere un esempio di come il valore di SBARRE venga riportato a stringa in base ai valori della tabella ascii. Le etichette co,cc e oc si comportano in maniera molto simile.

Chiamate delle funzioni num2str

```
dopo_SBARRE:

    # Metto il trattino #

    movl $45, %ebx
    movb %bl, (%ecx,%edi,1)
    inc %ecx

NPOSTI_:

    movl (NPOSTIA) ,%eax
    call num2str_NPOSTI
    movl (NPOSTIB), %eax
    call num2str_NPOSTI
    movl (NPOSTIC), %eax
    call num2str_NPOSTI

LIGTHS_:

    movb $3, %dl
    call num2str_LIGTHS
```

I due caratteri che rappresentano la situazione delle sbarre sono solo i primi valori necessari per la stampa sul file. Infatti, dopo un trattino bisogna lavorare sulle cifre dei valori presenti nei settori e sulle cifre presenti in LIGHTS. Per questo compito sono state realizzate due funzioni separate denominate num2str_NPOSTI e num2str_LIGTHS.

Num2str_NPOSTI

```
.type num2str_NPOSTI, @function

num2str_NPOSTI:

    movl $10 ,%ebx
    divb %bl
    movb %ah,%dl
    addb $48, %al
    addb $48,%dl
    movb %al, (%ecx,%edi,1)
    inc %ecx
    movb %dl, (%ecx,%edi,1)
    inc %ecx

    # Metto il trattino #

    movl $45, %ebx
    movb %bl, (%ecx,%edi,1)
    inc %ecx

    Ret
```

Divide i numeri per le due cifre e somma ad ognuna separatamente 48 per arrivare al valore ascii del numero corrispondente e infine tutto viene messo in ordine sulla stringa con un trattino finale.

Num2str_LIGTHS

```
.type num2str_LIGHTS, @function

num2str_LIGHTS:

    movl (LIGHTS), %eax

    cmp $0, %dl
    je fine_trasferimento

    movl $10 ,%ebx
    divb %bl
    movb %al,(LIGHTS)
    movb %ah ,%bl
    addb $48 ,%bl
    movb %bl, (%ecx,%edi,1)
    inc %ecx
    dec %edx

    jmp num2str_LIGHTS # Il loop

fine_trasferimento:
    Ret
```

Ogni volta questa funzione esegue un loop pari a 4 cicli dove i primi 3 servono effettivamente per smontare le cifre di LIGTHS attraverso la stessa tecnica della divisione adottata nella funzione precedente, mentre il 4 ci porta a fine _trasferimento, dove viene invocato semplicemente il return.

Fine riga di bufferout_asm

```
fine_riga_bufferout_asm:

    movb $10, (%ecx,%edi,1) # <-- Carattere \n
    inc %ecx

    movl %ecx, (ecx_tmp)

    # Recupero i valori precedenti dallo stack per ricominciare

    popl %eax # Vecchio contatore

    # Controllo se esi punta a \n o a \0 #

    movb (%eax,%esi,1), %bl
    cmpb $0, %bl

    je fine_stringa

    movl %eax, %ecx
    jmp fine_inserimenti

fine_stringa:

    cmp $0,(cont)
    je stampa

    movl (ecx_tmp) ,%eax

    #Metto il carattere \0 a fine stringa di bufferout_asm #

    movb $0, (%eax,%edi,1)

    # Esco dalla funzione assembly per tornare al codice C #

    popl %ebp
    Ret
```

Come ultima operazione viene ripreso dallo stack il contenuto di ecx di bufferin (quello di bufferout_asm viene salvato nella variabile ecx_tmp) e viene effettuato il controllo sul carattere "\0", carattere di interesse poichè è il carattere che determina la casella finale della stringa da leggere e perciò che si conclude il ciclo di lettura delle righe del file testin.txt. Al contrario se non incontriamo quel carattere allora la ricerca ricomincia da fine_inserimenti, per poi tornare qui alla fine del prossimo ciclo.

Conclusa quindi la parte di codice assembly lo stack deve essere rimesso in modo da poter effettuare il return per la funzione globale (nel nostro caso bisogna solo eseguire una pop del vecchio contenuto di

ebp). Nel caso in cui cont sia pari a zero allora avviene solo un'ulteriore stampa e quindi il ciclo non riprende completamente.

Codice ad alto livello

Main:

Azzero i registri / sistemo ebp e preparo i buffer per R/W

while(char_prelevato != A,B o C)

    continua a leggere caratteri

chiama leggi_lettera

salto trattino

metto eax (lettera) sullo stack

chiama leggi_valore

riprendo lettera dallo stack

if(lettera = A)

    assegno valore numerico ad A

if(lettera = B)

    assegno valore numerico a B

if(lettera = C)

    assegno valore numerico a C

else

    errore_settori

Inizio a scorrere le righe

```
if(char_prelevato = I)
        if(char_prelevato+1 = N)
        salto trattino
        if(lettera = A)
                if(posti_A < 31)
                        posti_A++
        if(lettera = B)
                if(posti_B < 31)
                        posti_B++
        if(lettera = C)
                if(posti_C < 24)
                        posti_C++
if(char_prelevato = O)
        if(char_prelevato+1 = U)
                if(char_prelevato+2 = T)
                        salto trattino
                        if(lettera = A)
                                if(posti_A > 0)
                                        posti_A--
                        if(lettera = B)
                                if(posti_B >0)
                                        posti_B--
                        if(lettera = C)
```

```
                    if(posti_C > 0)
                posti_C--
else
    scorri la stringa corrotta per passare alla prossima
if(cont = 0)
    cont++
if(posti_A = 31)
    LIGTHS +100
if(posti_B = 31)
    LIGTHS + 10
if(posti_C = 24)
    LIGTHS++
Metto ecx_bufferin sullo stack
Considero ecx_bufferout_asm
if(SBARRE = 11)
    assegno "OO" a bufferout
if(SBARRE = 12)
    assegno "OC" a bufferout
if(SBARRE = 21)
    assegno "CO" a bufferout
if(SBARRE = 22)
    assegno "CC" a bufferout
else
```

errore_entrata_uscita

metto il trattino

converto posti dei settori in stringa e metto su bufferout con trattino

converto LIGTHS in stringa e metto su bufferout

if(altre righe)

salvo contatore di bufferout

riprendo contatore di bufferin

torno su e ricomincio a leggere da IN/OUT

else

Metto carattere di fine stringa in bufferout

metto a posto lo stack per il return

return al codice C

Leggi_lettera:

if(lettera >= A AND lettera < D)

conservo lettera

return

else

errore_settori

```
Leggi_valore:
if(char_prelevato != capo_riga)
        char_prelevato = char_prelevato - 48
        if(char_prelevato >= 0 AND char_prelevato < 10)
                Metto char_prelevato sullo stack
        else
                errore_settori
else
        tolgo dallo stack l'ultimo valore messo sulla cima
        if(valore < A)
                considero la cifra per la somma (eventualmente moltiplico per 10)
        else
                rimetto il valore sullo stack
                return
```

Scelte progettuali

1) Si è cercato di realizzare funzioni precise, con lo scopo di essere utilizzate propriamente per il progetto e non in modo generico. Possiamo osservare da ciò che nel caso si inserisse un settore D bisognerebbe modificare una parte un po' corposa di codice, la stessa cosa se i settori diventassero di più cifre rispetto alle due assegnate.
2) Si è adottato un sistema "propedeutico" per capire se si trattasse di utente in entrata o uscita per poter seguire meglio l'andamento della stringa, era possibile infatti in alternativa realizzare una serie di if che su ogni carattere prelevato andassero a verificare l'uguaglianza con I,N,O,U,T,- ... ma magari ci si perdeva poi nei salti
3) Si è adottato l'uso di "11", "12", "21" e "22" con lo scopo iniziale di non aver problemi con una possibile combinazione "00". Col tempo si è capito che era possibile utilizzare "0" "1" "2" "3" oppure "1" "2" "3" "4" oppure altri tanti valori numerici ma è stata tenuta l'idea di base

4) Le variabili di output sono state conservate rispetto alla specifica per garantire una maggior comprensione del codice

5) Le etichette relative alla sezione di "stampa" non servono propriamente per stampare,bensì per riempire il bufferout_asm, questo proviene dal fatto che inizialmente si era pensato di stampare il contenuto di ogni riga singolarmente per poi accorgerci del fatto che nella seconda parte del codice C vi fosse la funzione adeguata. (Nel caso avessimo voluto scrivere su file avremmo utilizzato le system call open,write e close in quest'ordine, per poter ottenere il file descriptor del file,scrivere e chiudere infine il file).

6) Alla fine della stringa di bufferout_asm è stato inserito il valore "\0",esso non era propriamente necessario ma per questioni di debugging è stato utilizzato per stampare la stringa con un printf poichè era sorto un problema con la stampa finale su file. Nella visualizzazione del file testout.txt infatti il carattere di fine stringa non appare.