ELABORATO ASM 2019/2020

ARCHITETTURA DEGLI ELABORATORI

Studenti:

CASTELLI MARCO – VR443630

LLESHAJ LEDJO - VR450678

POIANA EMANUELE – VR446080

Descrizione del progetto

Il programma implementa la gestione automatizzata di un parcheggio mediante l'uso di Assembly e C.
Il parcheggio è suddiviso in 3 settori: i settori A e B hanno 31 posti macchina ciascuno, mentre il settore C ha 24 posti macchina, 2 sbarre una in accesso e una in uscita.
A seguito dell'inserimento delle macchine presenti inizialmente nei rispettivi settori A, B, C, il programma inizia il suo funzionamento autonomo, il sistema riceve in ingresso una stringa contenente il tipo di richiesta che può essere IN- oppure OUT-, seguito dal settore in cui l'utente vuole parcheggiare (A, B o C). Ad ogni richiesta il dispositivo risponde aprendo una sbarra e aggiornando il conteggio dei posti liberi nei vari settori.
Se un utente chiede di occupare un settore già completo oppure nel caso in cui la stringa sia errata, il sistema non apre alcuna sbarra.

Descrizione degli INPUT (file testin.txt)

SETTORE-NPOSTI (esempio: A-24) → ci sono 3 righe di questo tipo che indicano il settore e il numero di posti occupati inizialmente prima del funzionamento automatico del dispositivo.

TIPO-SETTORE (esempio: IN-A oppure OUT-C) → durante il funzionamento automatico si può entrare/IN o uscire/OUT solamente nei settori A, B, C. Combinazioni errate verranno scartate.

Descrizione degli OUTPUT (file testout.txt)

Sintassi stringa in uscita: SBARRE-NPOSTIA-NPOSTIB-NPOSTIC-LIGHTS

SBARRE: è una coppia di char che rappresentano l'apertura (O, open) o chiusura (C, closed) delle sbarre di ingresso e uscita rispettivamente (esempio: CO significa che la sbarra in ingresso è chiusa e quella in uscita aperta).

NPOSTIx (dove x può essere A, B o C): sono il numero di posti attualmente occupati nel rispettivo settore, espressi sempre con 2 cifre (es 02).

LIGHTS: è una terna di valori che rappresenta lo stato di accensione di una luce rossa in corrispondenza a ciascun settore per indicare che è pieno (esempio: 110 significa che i settori A e B sono pieni mentre ci sono ancora posti disponibili nel settore C).

Scelte progettuali

Abbiamo sviluppato un prototipo della parte ASM in C, alla quale abbiamo fatto riferimento per comprendere meglio la gestione delle stringhe sviluppata in seguito su ASM.

Metodo utilizzato

L'interconnessione tra il codice C e il codice ASM avviene mediante una chiamata di funzione nel codice C che richiama l'elaborazione del codice in ASM, il quale è stato sviluppato sul file elaborazione.s.

Tutta la parte ASM è caratterizzata dalla chiamata, a volte strutturata ciclicamente, di piccole funzioni tra di loro, questa scelta è stata fatta per rendere leggibile con facilità ogni sezione del programma evitando la creazione di macro funzioni complicate da correggere in caso di errore.

Parametri passati

Sono stati passati al blocco di assembly 2 indirizzi come parametri: bufferin e bufferout_asm.

1. "bufferin" allocato nello stack, viene caricato nel registro %esi e contiene la sequenza di ingresso da dare in input al programma ASM.
2. "bufferout_asm" allocato nello stack, viene caricato nel registro %edi e contiene l'output del programma ASM.

Accorgimenti

Dato l'inserimento iniziale di uno dei parcheggi nella forma "PARC-NAUTO", il programma inserisce in sequenza i valori nei parcheggi quindi il doppio inserimento di un parcheggio lo sovrappone a quello del successivo, quindi l'inserimento lo diamo per corretto fin da subito *(vedasi descrizione codice sottostante per capire meglio)*.

Funzionamento del codice

Logica di progettazione

Per coerenza con quanto fatto in C, creiamo la stringa bufferout gestendo sia numeri che lettere come caratteri presenti in determinate posizioni (come se fosse un array), gestiamo quindi in modo separato sia decina che unità per ciascun parcheggio, effettuando quindi opportuni salti per poter incrementare e decrementare correttamente la decina e l'unità. Ad ogni operazione di automatico andremo a creare una stringa del tipo sotto riportato che formerà una nuova riga di bufferout.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
C/O	C/O	-	decinaA	UnitaA	-	decinaB	unitaB	-	decinaC	unitaC	-	pienoA	pienoB	pienoC	\n

Funzionamento generale

Il codice è suddiviso concettualmente e sintatticamente in 4 macro-blocchi principali all'interno dei quali vengono effettuate differenti operazioni.

I quattro macro-blocchi sono i seguenti:

- "Operazioni preliminari" → all'interno di questo blocco vengono effettuate le operazioni di salvataggio dei registri generali sullo stack, di prelievo della stringa del bufferin e bufferout, di azzeramento dei registri generali per uso nel codice, di controllo se il file che viene passato è vuoto ed infine di dichiarazione di due registri 'dl' e 'ch' per operazioni successive.

- "Salvataggio"→ questo blocco consente di salvare i valori dei parcheggi A,B,C in fase iniziale, sono presenti dei controlli che non inizializzano il valore del parcheggio se viene inserita la combinazione PARCHEGGIO—(A--); PARCHEGGIO-VALORENONNUMERICO (A-D) ed inoltre se si inserisce come nome parcheggio un valore diverso da A, B, C. In tutti questi casi il programma va alla riga successiva e continua a rimanere in salvataggio finchè non trova e salva 3 combinazioni corrette. Naturalmente le combinazioni possono trovarsi in ordine diverso (es. A,B,C oppure C,B,A ecc..) tuttavia, il codice se analizza correttamente due o più inserimenti sullo stesso parcheggio (A,B,A tralasciando C o combinazioni simili) non funziona questo perché abbiamo ipotizzato che l'utente digiti in fase iniziale sempre i tre diversi parcheggi in ordine diverso ma, non con ripetizioni.

- "Automatico" → a seguito dei tre inserimenti nei parcheggi il codice arriva in questo blocco, in tale blocco vengono effettuate le operazioni di ingresso-uscita e apertura sbarre. L'esecuzione rimane in questo blocco finchè non legge il carattere '\0' che termina la stringa e a questo punto salta al blocco end. Se la combinazione è errata si lasciano le sbarre chiuse senza alterare i valori dei parcheggi, se la combinazione invece è corretta si passa al rispettivo ingresso o uscita per A, B, C che si collega a sua volta al modulo che apre la sbarra in uscita o in entrata.

- "Blocco finale"→ in cui sono presenti i moduli riganext, copiaInfo e end. Intuibilmente rigaNext consente di raggiungere la nuova riga presente in bufferin e salta differentemente se siamo ancora in fase di salvataggio o fase automatica, copiaInfo consente di copiare nella

nuova riga di bufferout le informazioni quali le macchine presenti nei parcheggi e i bit dei parcheggi pieni e che eventualmente verranno modificate nelle operazioni successive, inoltre fa puntare edi a una nuova riga incrementandolo di 16 . Il modulo end aggiunge il carattere '\0' alla stringa bufferout e scarica lo stack.

Funzionamento specifico

PARTE SALVATAGGIO

Sfrutto il registro 'dl' e 'ch', il primo lo utilizzo per mantenere il numero di caratteri di ogni riga di bufferin in modo che posso sommare a %esi per raggiungere la riga successiva di bufferin; il secondo invece lo utilizzo per mantenere il numero di salvataggi effettuati A,B,C =3 e lo inizializzo a 0.
Se sono stati effettuati già i 3 salvataggi salto al modulo automatico altrimenti controllo che non ci siano errori sintattici e se voglio salvare in A o in B o in C oppure vado in Error.
SalvainA, SalvainB e sSalvainC sono logicamente equivalenti, e inseriscono nelle opportune posizioni decine e unita del settore in cui voglio salvare i valori, se sono al di sotto di 31 per A e B e di 24 per C, in caso negativo il settore viene inizializzato automaticamente a pieno. Se viene specificato un numero con solo unità 'dl' diventa 4 mentre se ho un numero sia con decine che unità 'dl' diventa 5 questo perché sommando dl a esi vado al primo carattere della riga successiva di bufferin.
Ad ogni salvataggio CORRETTO 'ch' viene incrementato.

PARTE AUTOMATICA

Se la stringa bufferin è finita salto a end, altrimenti inizializzo con valori standard (trattinini, \n e sbarre chiuse) la stringa bufferout, alla prima iterazione i valori dei parcheggi e i bit Pieni saranno già presenti mentre, dalla seconda e così via sarà necessario il modulo copiaInfo per non perdere le informazioni.
Dopodichè ci sono numerosi controlli per arrivare a leggere una stringa corretta tra IN-A, IN-B, IN-C, OUT-A, OUT-B, OUT-C altrimenti si va al modulo Error. Se il comando digitato è corretto si va nel rispettivo modulo di ingresso o uscita per A, B, C. Se entriamo in ingressoX si controlla che il bit pieno relativo al settore non sia 1 solo in questo caso si può incrementare il valore (e poi eventualmente impostare a 1 il bit pieno) e aprire la sbarra, mentre se il settore è pieno si lasciano le sbarre chiuse, non si applicano modifiche e si passa ad analizzare il comando successivo tramite il modulo Error.
Per quanto riguardo uscitaX se il valore nel parcheggio X è positivo si può uscire e quindi decrementare il valore presente nel parcheggio e impostare a 0 i bit pieno del settore, altrimenti si va al modulo Error come per il caso di prima.
Modulo Error nel caso di combinazioni errate conta il numero di caratteri presenti nella riga di bufferin per andare alla successiva e produce su bufferout una stringa in cui le sbarre sono chiuse, di questo tipo: CC-A-B-C-XYZ. Le stringhe non consentite sono le seguenti: tutte le stringhe diverse da IN-A, IN-B, IN-C, OUT-A, OUT-B, OUT-C seguite naturalmente da \n, vengono infatti effettuati nel codice diversi controlli per trovare il possibile errore. Se la combinazione invece è corretta si passa al rispettivo ingresso o uscita per A, B, C che si collega a sua volta al modulo che apre la sbarra in uscita o in entrata.

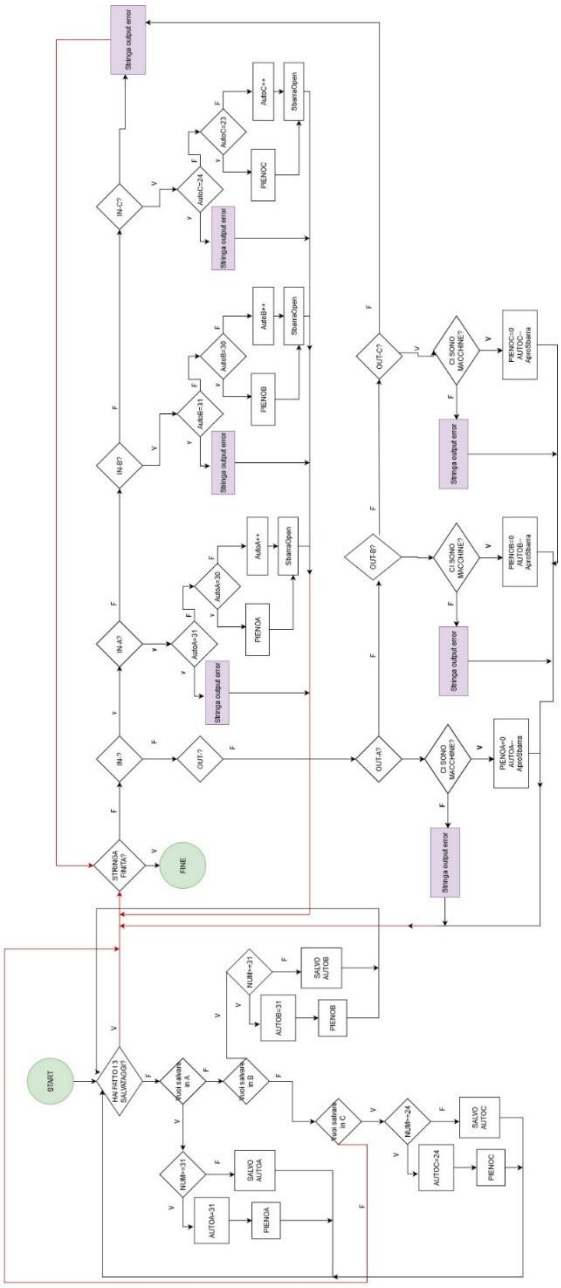
PARTE FINALE

rigaNext sfruttiamo il registro dl e uniamo dh che inizializziamo a 0 formando edx, in modo che è come se sommassi solo dl a esi, in questo modo %esi punta al primo carattere della nuova riga i bufferin. Dopodiche si controlla se sono stati effettuati i 3 salvataggi nei parcheggi, in caso ch valga meno di 3 si salta al modulo di salvataggio altrimenti si incrementa ch e se vale 4 significa che

abbiamo appena terminato i salvataggi e quindi si salta al modulo automatico senza crearsi una nuova riga di %edi, se invece ch vale >4 salto a copiaInfo che mi copia i valori relativi ai parcheggi (decina e unità) e i bit pieni che saranno quindi disponibili all'eventuale modifica con il comando successivo.

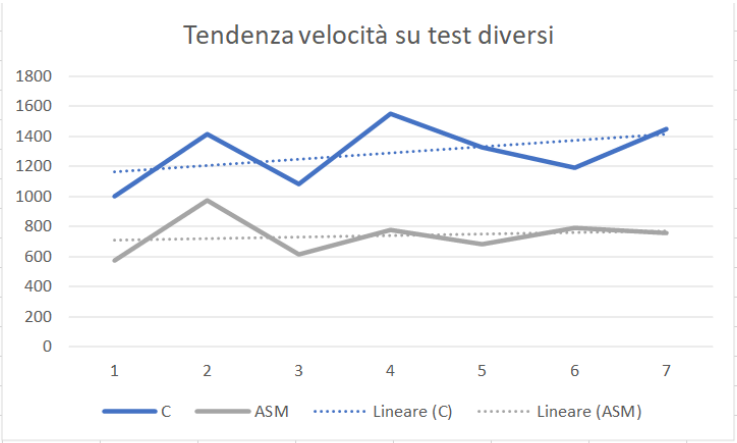
Se arrivo a end ho finito di effettuare i comandi metto il terminatore all'ultima posizione di bufferout e scarico lo stack portando allo stato iniziale tutti i registri, in particolare %edi alla fine punterà all'inizio della stringa bufferout.

Diagramma di flusso (per risultare più leggibile data la dimensione lo abbiamo ruotato in verticale)



Guadagni prestazionali

File diversi		
Prova	C	ASM
1	1003	576
2	1415	976
3	1086	617
4	1549	781
5	1330	684
6	1193	795
7	1450	760



Si nota che indipendentemente dalla dimensione delle diverse prove svolte il tempo di esecuzione del programma in C rimane sempre superiore a quello de programma in assembly.

Test Prof	C	ASM	Ottimizzazione
1	1273	722	1,763157895
2	1212	693	1,748917749
3	1230	770	1,597402597
4	1190	689	1,727140784
5	1361	692	1,966763006

Rapporto tra tempo medio di esecuzione in linguaggio C e tempo medio di esecuzione in linguaggio ASM: 1,754. *Il seguente test è stato fatto sul file fornitoci di esempio "testin.txt".*

Considerazioni finali

La sezione ASM è in media circa 1,75 volte più veloce di quella in C nonostante l'algoritmo risulti nettamente più complesso e più lungo.

La migliora dataci dal codice ASM è probabilmente dovuta al fatto che il codice ASM usa principalmente i registri della cpu che hanno un tempo di accesso di gran lunga inferiore alla memoria, accedendo ad essa solo nelle operazioni di lettura e scrittura delle stringhe di input e output.