UNIVERSITÀ
di **VERONA**

Elaborato Assembly

Architettura degli Elaboratori

A.A. 2020/2021 - Corso di Laurea in Informatica

Enrico Bragastini
VR456374

Davide Bianchini
VR456697

Andrea Mafficini
VR462441

Indice

Descrizione dell'elaborato

La **notazione polacca inversa** (reverse polish notation, **RPN**) è una notazione per la scrittura di espressioni aritmetiche in cui gli operatori binari, anziché utilizzare la tradizionale notazione infissa, usano quella postfissa; ad esempio, l'espressione $5 + 2$ in RPN verrebbe scritta 5 2 +. La **RPN** è particolarmente utile perché *non necessita dell'utilizzo di parentesi.*

Si intende realizzare un programma in assembly che, letta in input una stringa rappresentante un'espressione ben formata in **RPN**, scriva in output il risultato ottenuto dalla valutazione dell'espressione. Per il calcolo di un'espressione in **RPN** si considerano solamente gli operatori fondamentali: somma, sottrazione, moltiplicazione e divisione. Nel caso in cui l'espressione contenga caratteri diversi da numeri o da operatori, verrà restituita come output la stringa `Invalid`.

Il codice sorgente *main.c* fa una chiamata a una funzione *extern* chiamata *postfix* scritta in assembly. Questa funzione riceve come parametri due puntatori ai relativi array che rappresentano la stringa di input e la stringa di output. La funzione si occuperà quindi di leggere la stringa di input, elaborare il risultato e scriverlo nell'array di output. La lettura e la scrittura su file vengono gestite dal *main.c*.

Esempio:

```
$ echo "100 10 - 10 * -4 /" > in_1.txt
$ ./postfix
$ cat out_1.txt
  -225
```

L'espressione postfissa "100 10 - 10 * -4 /" corrisponde all'espressione in notazione infissa $(100 - 10) * 10/(-4)$, che dà come risultato -225.

Metodo di lavoro

Abbiamo deciso di lavorare sempre in gruppo, senza dividere il progetto in parti a cui lavorare singolarmente. Per comodità, abbiamo usufruito della piattaforma Zoom per lavorare tutti e tre contemporaneamente al progetto.

Inoltre, per condividere il codice ed evitare spiacevoli inconvenienti, abbiamo fatto uso di Git e GitHub.

Inizialmente, la necessità era quella di avere una visione generale del codice, abbiamo quindi creato varie etichette, per suddividere il programma. In ogni sezione abbiamo scritto delle *pseudo-istruzioni* per descrivere cosa avrebbe fatto quella determinata parte di codice assembly.

Una volta terminata questa fase iniziale di *brainstorming*, abbiamo implementato e commentato uno ad uno il codice di ogni etichetta, testandone il corretto funzionamento prima di passare alla etichetta successiva.

Codice Assembly

3.1 Scelte progettuali e struttura del codice

Abbiamo scelto di non suddividere il codice in file multipli o di utilizzare ulteriori funzioni in quanto non ne abbiamo sentita la necessità. Il codice è stato ben suddiviso in sezioni utilizzando le **etichette**. Inoltre è ben commentato e non supera le 270 righe, per cui è facilmente leggibile e comprensibile.

Il sorgente contenuto nel file *main.c* fa una chiamata alla funzione *postfix* passandole come argomenti i puntatori agli array di input e output. Questi puntatori, che si trovano nello stack, vengono copiati nei registri `%esi` e `%edi`.

Ciclicamente vengono letti uno alla volta tutti i caratteri della stringa in input. In base al carattere trovato vengono svolte determinate azioni:

- **CIFRA**: Quando viene trovata una cifra, questa viene inserita nella variabile *buffer*. Verrà fatta la push del buffer una volta trovato lo spazio che segue l'ultima cifra di un numero.
- **SPAZIO**: Quando viene trovato uno spazio, è necessario valutare se questo si presenta dopo una cifra o dopo un operatore. Nel primo caso indica la fine di un numero, si procederà quindi al push del buffer. Nel secondo caso nessuna operazione è necessaria, si procederà alla lettura del carattere successivo.
- **SOMMA**: Quando viene trovato il carattere di somma, viene fatta la pop dei due operandi, che devono essere sommati e il risultato viene messo in pila.
- **TRATTINO**: Quando viene trovato il trattino, è necessario fare delle considerazioni perché potrebbe indicare l'inizio di un numero negativo oppure l'operatore di sottrazione. Viene quindi valutato il carattere successivo, se quest'ultimo è uno spazio viene eseguita la sottrazione in maniera analoga alla somma. Altrimenti si pone la variabile $FLAG$ a -1 utilizzata al momento della push del buffer, per negare il numero.
- **MOLTIPLICAZIONE**: Quando viene trovato l'asterisco, viene fatta la pop dei due operandi, che devono essere moltiplicati. Il risultato viene messo in pila.
- **DIVISIONE**: Quando viene trovato lo slash, ci si assicura inizialmente che il divisore sia diverso da zero e il dividendo sia maggiore di zero. Se queste condizioni sono verificate, viene fatta la pop dei due operandi, viene eseguita la divisione e il risultato viene messo in pila.
- **INVALID**: Il riscontro di un carattere invalido è conseguenza di tutti i controlli eseguiti precedentemente. Quando un carattere non risulta essere una cifra, un operatore o uno spazio, per forza sarà un carattere invalido. Verrà quindi stampata la stringa `Invalid`.

3.2 Variabili

Le variabili utilizzate sono quattro e tutte posizionate nella sezione *.data*

- **buffer**, di tipo *int*, rappresenta il buffer in cui vengono salvate le cifre del numero che si sta leggendo. Il suo valore di default è 0 e viene riportata a tale valore alla fine della lettura di ogni numero.
- **flag**, di tipo *int*, rappresenta il segno del numero che si sta leggendo. Il suo valore di default è 1, ovvero numero positivo, viene settata a -1 ogni qualvolta si andrà a leggere un numero negativo.
- **invalid_string**, di tipo *ascii*, rappresenta la stringa da scrivere nella stringa output nel caso vengano trovati dei caratteri invalidi nella stringa di input.
- **invalid_string_len**, di tipo *long*, rappresenta la lunghezza della stringa contenuta nella precedente variabile.

3.3 Diagramma di flusso