Architettura degli Elaboratori

Relazione Laboratorio ASM

A cura di:
Alessio Zattoni (Matr. VR457153)

A.A. 2020/2021

"Realizzazione algoritmo RPN"

INDICE

1. ***TESTO ELABORATO*** 3

2. ***VARIABILI*** 6

3. ***FUNZIONI*** 6

4. ***MAKEFILE*** 7

5. ***RAPPRESENTAZIONE STACK*** 8

6. ***FLOW CHART*** 9

7. ***DESCRIZIONE CODICE*** 11

8. ***GDB*** 13

9. ***SCELTE PROGETTUALI*** 16

TESTO 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 considerino ad esempio le due espressioni:

- *2 * 5 + 1*
- *2 * (5 + 1)*

Nel secondo caso le parentesi sono necessarie per indicare che l'addizione va eseguita prima della moltiplicazione.
*In RPN questo non è necessario perché le due espressioni vengono scritte in maniera diversa: mentre la prima corrisponde a 2 5 * 1 +, la seconda viene scritta come 2 5 1 + *. Un altro vantaggio della RPN è quello di essere facilmente implementabile utilizzando uno stack.*
*Per calcolare il valore di un'espressione, è sufficiente scandirla da sinistra verso destra: quando viene letto un numero lo si salva nello stack, quando viene letta un'operazione binaria si prelevano due numeri dallo stack, si esegue l'operazione tra tali numeri e si salva nuovamente il risultato nello stack. Ad esempio, volendo valutare il valore dell'espressione 2 5 1 + *, si procede nel seguente modo:*

1.metto il valore 2 nello stack;

2.metto il valore 5 nello stack;

3.metto il valore 1 nello stack;

4.estraggo i primi due valori memorizzati in cima allo stack (5 e 1);

5.faccio la somma e salvo il risultato nello stack;

6.estraggo i primi due valori memorizzati in cima allo stack (2 e 6);

7. faccio la moltiplicazione e salvo il risultato;

8. A questo punto l'intera stringa è stata elaborata e nello stack è memorizzato il risultato finale.

Obiettivo:

*Si scriva un programma in assembly che legga in input una stringa rappresentanteun'espressione ben formata in numero di operandi e operazioni in RPN (si considerino solo gli operatori + - * /) e scriva in output il risultato ottenuto dalla valutazione dell'espressione.*

Requisiti & Vincoli:

Le espressioni che verranno usate per testare il progetto hanno i seguenti vincoli:

- Gli operatori considerati sono i 4 fondamentali e codificati con i seguenti simboli:

- *+ Addizione*
- ** Moltiplicazione (non x)*
- *- Sottrazione*
- */ Divisione (non \)*

- Un operando può essere composto da più cifre intere con segno (10, -327, 5670)

- Solo gli operandi negativi hanno il segno riportato esplicitamente in testa.
- Gli operandi hanno un valore massimo codificabile in 32-bit.
- Il risultato di una moltiplicazione o di una divisione può essere codificato al massimo in 32-bit.
- Il risultato di una divisione dà sempre risultati interi, quindi senza resto.
- Il dividendo di una divisione delle istanze utilizzate è sempre positivo, mentre il divisore può essere negativo. Esempi:

- *-6 / 2 che in RPN diventa -6 2 / non è valido.*
- *6 / -2 che in RPN diventa 6 -2 / è valido.*

- Tra ogni operatore e/o operando vi è uno spazio che li separa.
- L'ultimo operatore dell'espressione è seguito dal simbolo di fine stringa "\0".
- Le espressioni NON hanno limite di lunghezza.
- L'eseguibile generato si dovrà chiamare postfix- Non è consentito l'utilizzo di chiamate a funzioni descritte in altri linguaggi all'interno del codice Assembly.
Se le stringhe inserite non sono valide (contengono simboli che non sono operatori o numeri) il programma deve restituire la stringa scritta esattamente nel seguente modo: Invalid
NON verranno fatti test con stringhe il cui numero di operandi non coincide con il numero di operatori. Il programma deve essere lanciato da riga di comando con due stringhe come parametri, la prima stringa identifica il nome del file .txt da usare come input, la seconda quello da usare come output.

VARIABILI

- ***is_neg*** : variabile utilizzata come flag per sapere se il numero è negativo, se lo è il numero viene cambiato di segno e successivamente viene inserito nello stack. Questa variabile è di tipo byte, viene inizializzata a 0 con la dichiarazione.
- ***tmp***: utilizzata come stringa di appoggio per memorizzare i valori di output, in questa stringa i valori verranno inseriti al contrario (usando la divisione ed estraendo il resto), per poi essere inseriti in output partendo dall'ultima posizione di **tmp.** Questa variabile è di tipo ascii (stringa), in fase di dichiarazione viene inizializzata tutta a 0 per convenzione.

FUNZIONI

1) ***POSTFIX*** **:** è la funzione che svolge l' algoritmo Rpn. La funzione scrive il risultato nell'indirizzo di memoria puntato da output.

   Input:

   - **char * input**: contiene il puntatore al file di input (sarà un file contenente delle operazioni rpn).

   - **char* output:** utilizzato per memorizzare il risultato del file di input.

   Output:

   - **void**: nella funzione si scrive sul file di output per side effect. Dato che opera sui puntatori non serve restituire nessun valore.

2) ***PARSING*:** Questa funzione esegue l'analisi della stringa, se non ci sono simboli o caratteri errati questa funzione non fa niente. Altrimenti scrive sul puntatore ad output "Invalid".

Input:

- **void**: Questa funzione lavora sulla stringa di input recuperabile dallo stack, la funzione è annidata dentro a **POSTFIX.**

Output:

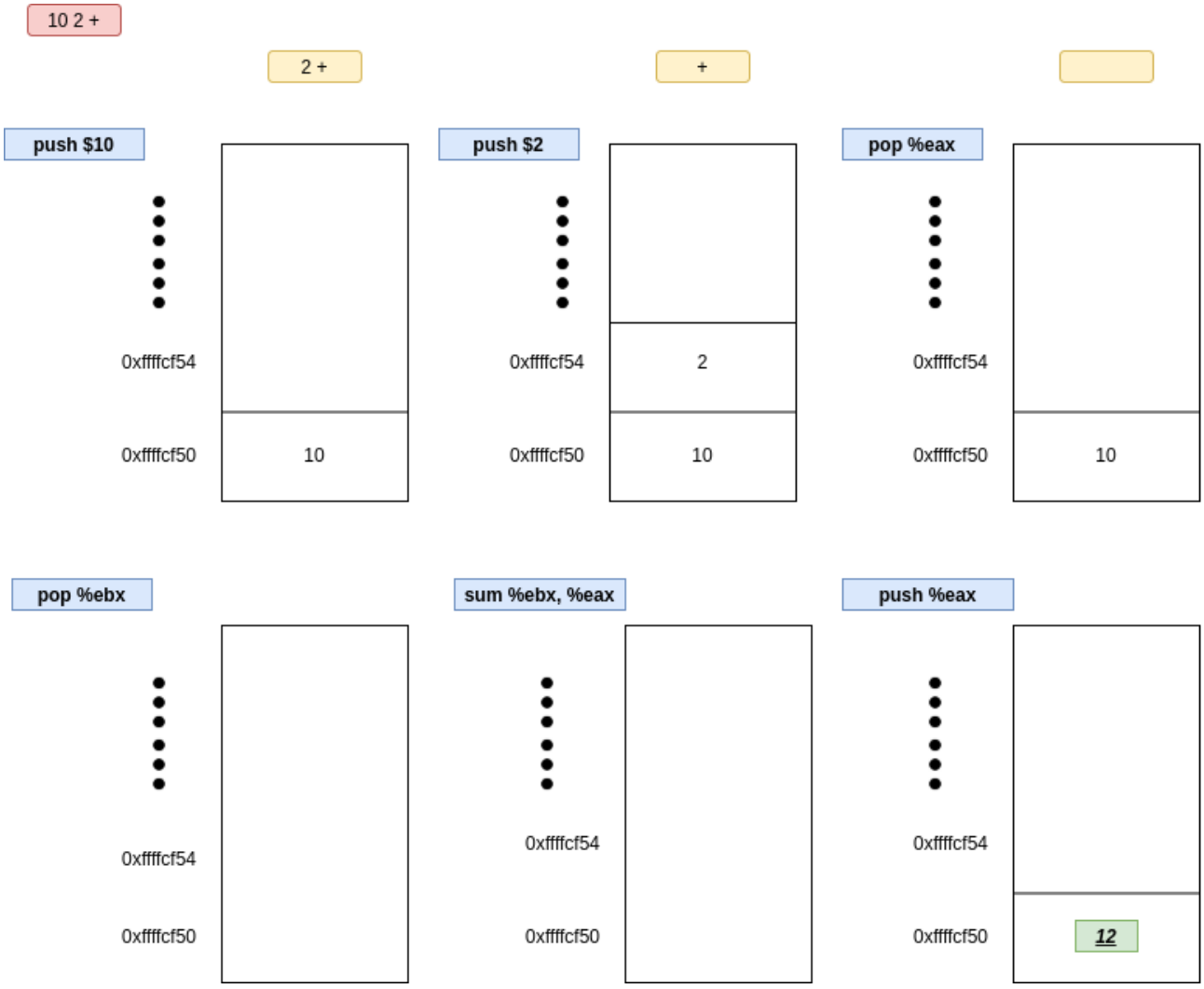
- **int flag**: Questa funzione controlla solo la validità della stringa. Se è valida non tocca eax, se non è valida invece alza un flag, inizializzando eax a 1. Una volta che finisce ci sarà un controllo su eax per decidere se interrompere il programma o proseguire.

MAKEFILE

```
AS = as --32
GCC = gcc -m32
FLAGS = -gstabs

bin/postfix: src/main.c obj/postfix.o obj/parsing.o
	$(GCC) src/main.c obj/postfix.o obj/parsing.o -o bin/postfix

obj/postfix.o: src/postfix.s
	$(AS) src/postfix.s -o obj/postfix.o

obj/parsing.o: src/parsing.s
	$(AS) src/parsing.s -o obj/parsing.o

clean:
	rm -rf bin/* && rm -rf obj/*
```

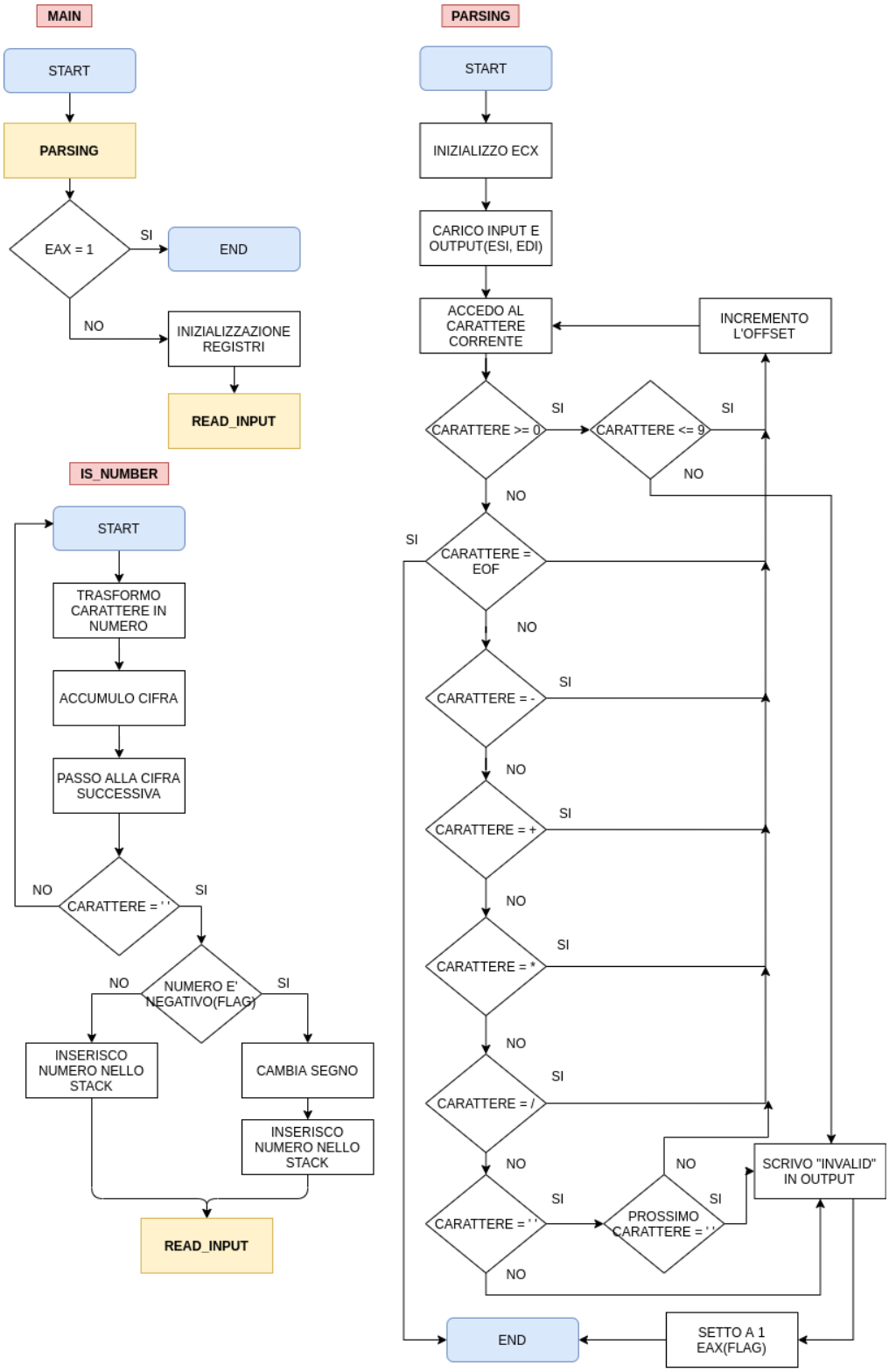
Il makefile prima assembla le due funzioni con l'assembler GAS, poi unisce il main.c con i due file oggetto creati in precedenza. Questa fase è realizzata attraverso GCC. Nel file è stata mantenuta la variabile FLAGS utilizzata in fase di debugging, successivamente è stata tolta dalla creazione dei due file oggetto. Questa decisione è dovuta al fatto che i simboli di debugging rallentano il programma. Inoltre è stata aggiunta una flag alla variabile GCC per creare un eseguibile a 32 bit.

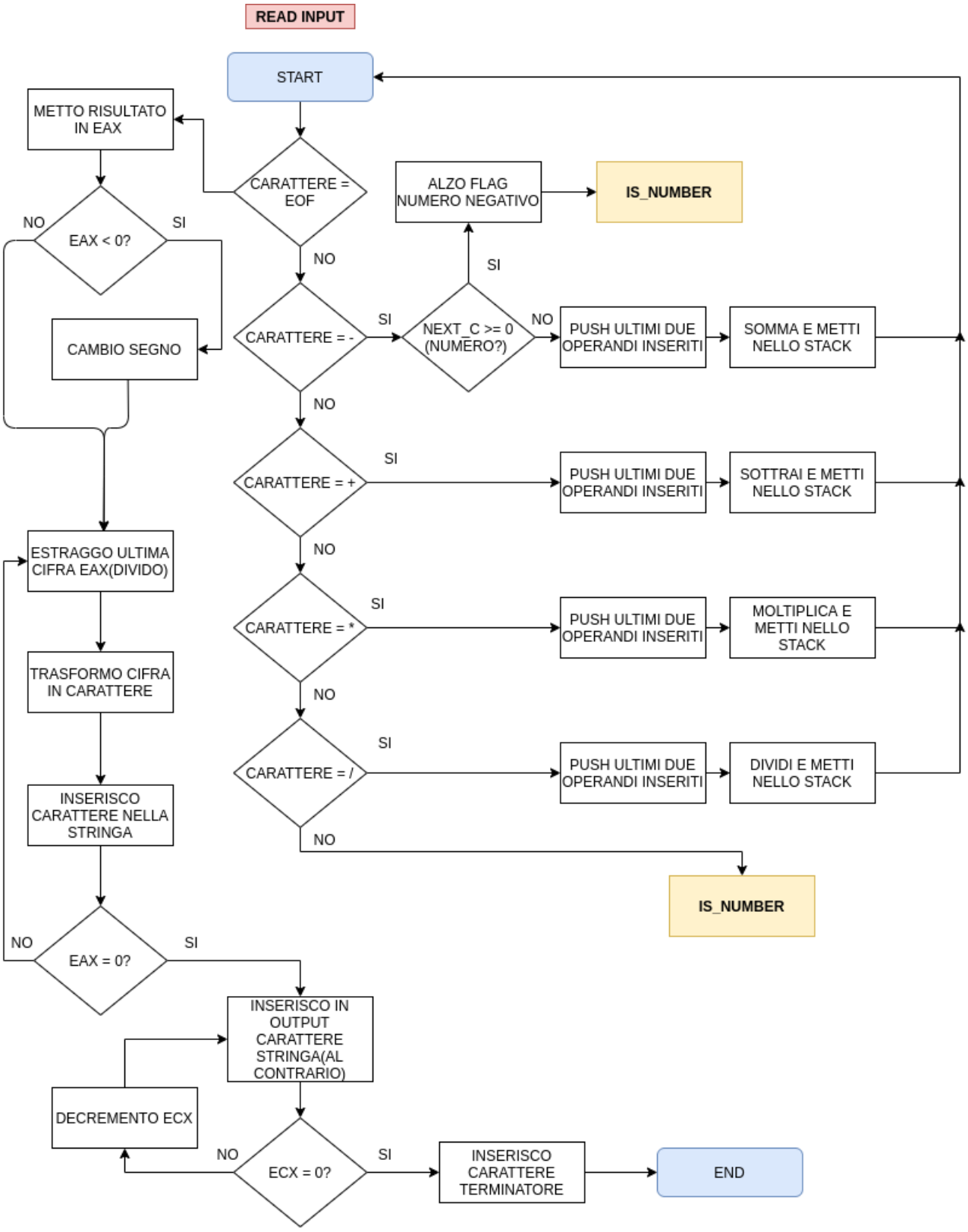
RAPPRESENTAZIONE STACK



Questo schema rappresenta lo stack quando si esegue l'algoritmo rpn. Si è fatto uso dei due registri(eax, ebx) a scopo esemplificativo. Nella prima fase vengono caricati gli operandi, successivamente quando si trova un operatore basta estrarre gli operandi. Dato che lo stack è strutturato come FIFO, il primo operando che esce è l' ultimo che si inserisce. La fase successiva consiste nell'eseguire l'operazione tra i due operandi. Infine il risultato va inserito nello stack. A questo punto se non ci sono altri numeri o espressioni (operatori) nella stringa allora quello che si ha nello stack è il risultato, in questo caso 12.

FLOW CHART


MAIN
START
PARSING
EAX = 1
SI
END
NO
INIZIALIZZAZIONE REGISTRI
READ_INPUT
IS_NUMBER
START
TRASFORMO CARATTERE IN NUMERO
ACCUMULO CIFRA
PASSO ALLA CIFRA SUCCESSIVA
CARATTERE = ' '
NO
SI
NUMERO E' NEGATIVO(FLAG)
NO
SI
INSERISCO NUMERO NELLO STACK
CAMBIA SEGNO
INSERISCO NUMERO NELLO STACK
READ_INPUT
PARSING
START
INIZIALIZZO ECX
CARICO INPUT E OUTPUT(ESI, EDI)
ACCEDO AL CARATTERE CORRENTE
INCREMENTO L'OFFSET
CARATTERE >= 0
SI
CARATTERE <= 9
SI
NO
NO
CARATTERE = EOF
SI
NO
CARATTERE = -
SI
NO
CARATTERE = +
SI
NO
CARATTERE = *
SI
NO
CARATTERE = /
SI
NO
CARATTERE = ' '
SI
PROSSIMO CARATTERE = ' '
SI
NO
SCRIVO "INVALID" IN OUTPUT
NO
END
SETTO A 1 EAX(FLAG)


READ INPUT
START
METTO RISULTATO IN EAX
CARATTERE = EOF
ALZO FLAG NUMERO NEGATIVO
IS_NUMBER
NO
EAX < 0?
SI
NO
SI
CARATTERE = -
NEXT_C >= 0 (NUMERO?)
NO
PUSH ULTIMI DUE OPERANDI INSERITI
SOMMA E METTI NELLO STACK
CAMBIO SEGNO
NO
CARATTERE = +
SI
PUSH ULTIMI DUE OPERANDI INSERITI
SOTTRAI E METTI NELLO STACK
NO
ESTRAGGO ULTIMA CIFRA EAX(DIVIDO)
CARATTERE = *
SI
PUSH ULTIMI DUE OPERANDI INSERITI
MOLTIPLICA E METTI NELLO STACK
NO
TRASFORMO CIFRA IN CARATTERE
CARATTERE = /
SI
PUSH ULTIMI DUE OPERANDI INSERITI
DIVIDI E METTI NELLO STACK
NO
INSERISCO CARATTERE NELLA STRINGA
IS_NUMBER
NO
EAX = 0?
SI
INSERISCO IN OUTPUT CARATTERE STRINGA(AL CONTRARIO)
DECREMENTO ECX
NO
ECX = 0?
SI
INSERISCO CARATTERE TERMINATORE
END

DESCRIZIONE CODICE

Il codice inizia con una chiamata alla funzione **parsing**, tale funzione inizializza il registro ecx per un successivo utilizzo come spiazzamento. Carico in edi l'indirizzo di output e in esi l'indirizzo di input, da qui parte un ciclo (parse) il quale scansiona di volta in volta un carattere di input; utilizzando l'indirizzamento indiretto a registro con spiazzamento.

In parse vengono controllati i caratteri, se il carattere corrente è +, -, * , / , ' ', EOF o un numero proseguo il ciclo, se incontro il carattere '\n' il programma termina eseguendo **error_msg,** altrimenti arrivo alla jmp. La jmp carica nell'indirizzo di output la stringa "Invalid", sfruttando l' indirizzamento indiretto con spiazzamento. Questa volta lo spiazzamento è sempre +1 dal valore precedente, inoltre viene aggiunto il carattere di fine stringa "\0". Se incontro uno spazio, tramite **double** controllo se il prossimo carattere è uno spazio, se lo è la funzione finisce eseguendo **error_msg,** altrimenti prosegue il flusso. Il confronto per verificare se il carattere è un numero (unità) viene fatto tramite dei cmp sfruttando il fatto che un numero valido è >= 0 e <= 9. Se la stringa di input è corretta allora la funzione esegue la ret, scaricando lo stack e saltando all'indirizzo indicato dallo esp (è il program counter della funzione chiamante aumentato di 1, quindi punta all'istruzione successiva). Altrimenti tramite **error_msg** carico il registro eax con il valore 1, in questo modo lo utilizzo come flag, per segnare che la stringa non è valida. Quando ritorna il programma restituisce eax (1 non valido, altrimenti valido). Facendo una compare nella funzione postfix è possibile gestire la fine della funzione o il suo seguito.

La funzione postfix inizializza i vari registri per un uso successivo, sposta in esi l'indirizzo puntato da input e in edi l'indirizzo puntato da output. Successivamente parte il ciclo **read_input**, il quale estrae volta per volta un carattere (a questo punto la stringa è valida, non devo fare controlli) e controlla se è un operando o un operatore. Se non è un operatore allora di sicuro è un numero. Da qui parte il ciclo **is_number**

che serve per estrarre il numero convertendolo da ascii ad intero, finché non trovo uno spazio moltiplico per 10 e accumulo.

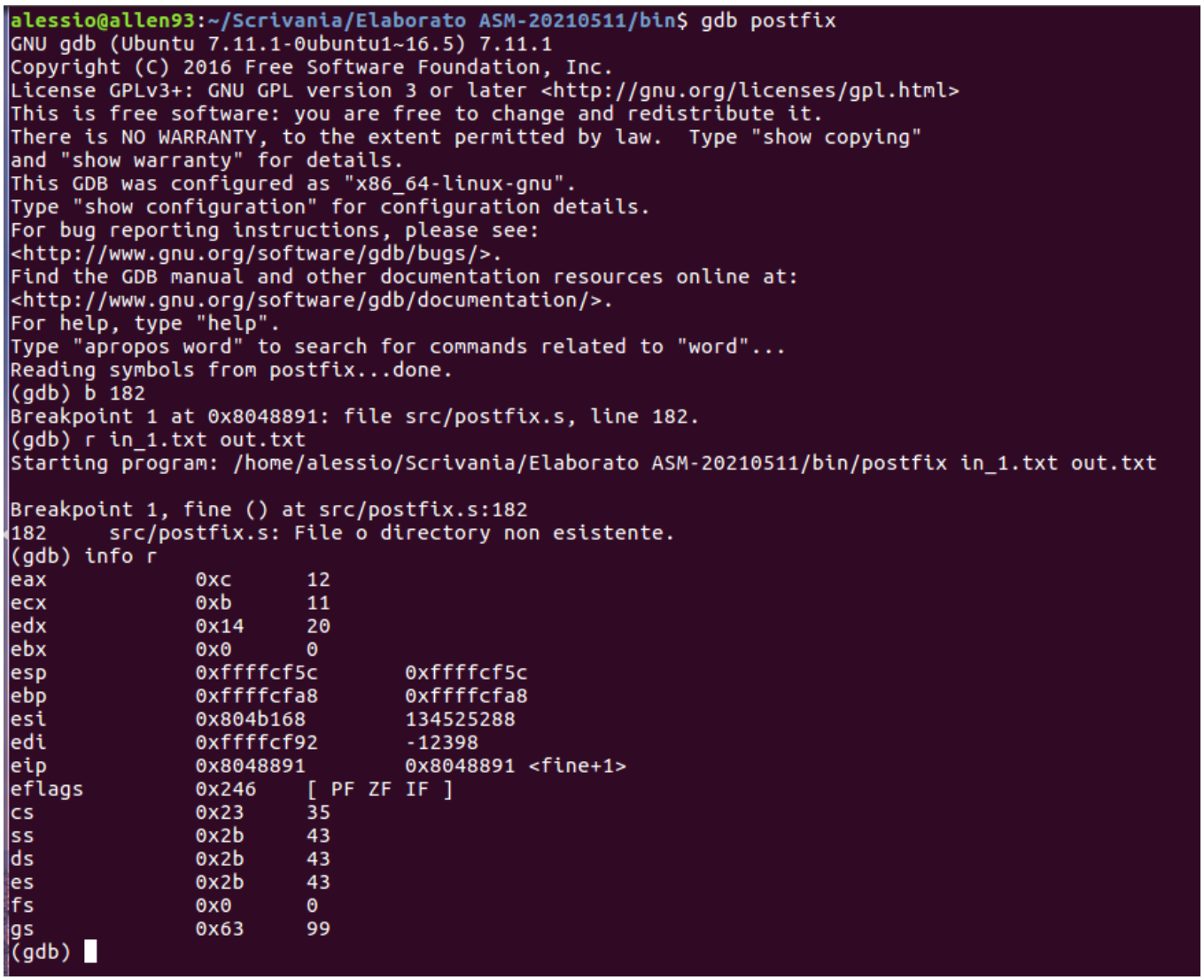
Quando trovo uno spazio, se il numero è negativo moltiplico per -1 e inserisco il numero nello stack, altrimenti lo inserisco direttamente senza moltiplicarlo; a questo punto riparte il ciclo. Se ora trovo un operatore allora estraggo dallo stack gli ultimi due valori, eseguo l' operazione, inserisco il risultato nello stack e inizio di nuovo il ciclo **read_input**.

Con la **subtraction** bisogna gestire il caso in cui si trova una sottrazione e subito dopo un numero senza uno spazio (-4 =! - 4). Nel primo caso vuol dire che la sottrazione si riferisce ad un numero, quindi alzo una flag con **negative_token** e ritorno a gestire il numero con **is_number.** Invece in caso contrario vuol dire che devo eseguire la sottrazione, con relativi push e pop. Una volta finito di analizzare la stringa, tramite una pop estraggo il risultato, il quale si trova in cima allo stack. Se negativo allora giro il segno e metto al primo posto nel puntatore di output il simbolo '-', altrimenti passo a **write.** Con questa sezione carico il registro ebx per un utilizzo successivo, carico l' indirizzo di memoria di **tmp** (stringa di appoggio temporanea) e lancio **continua_a_dividere**. Questa parte serve per estrarre l' ultimo valore del numero (modulo), cambiarlo da intero a carattere, caricarlo nella stringa temporanea. Questa operazione viene eseguita finché non si raggiunge lo zero; una volta raggiunto si azzera ebx. Nella stringa tmp troverò i valori al contrario (123 → 321).

Tramite **ribalta** carico in output il numero dall'ultima posizione alla prima, utilizzando l' indirizzamento con spiazzamento ed ecx, il quale puntava già inseme ad esi all'ultima cella di **tmp.** Attraverso il comando loop viene decrementato ecx ad ogni iterazione, raggiungendo la prima cella della stringa all'ultima iterazione. Infine aggiungo il carattere di terminazione e chiamo la ret, la quale ritorna al programma c. Il programma c scrive il contenuto puntato da output sul file specificato all'avvio del programma.

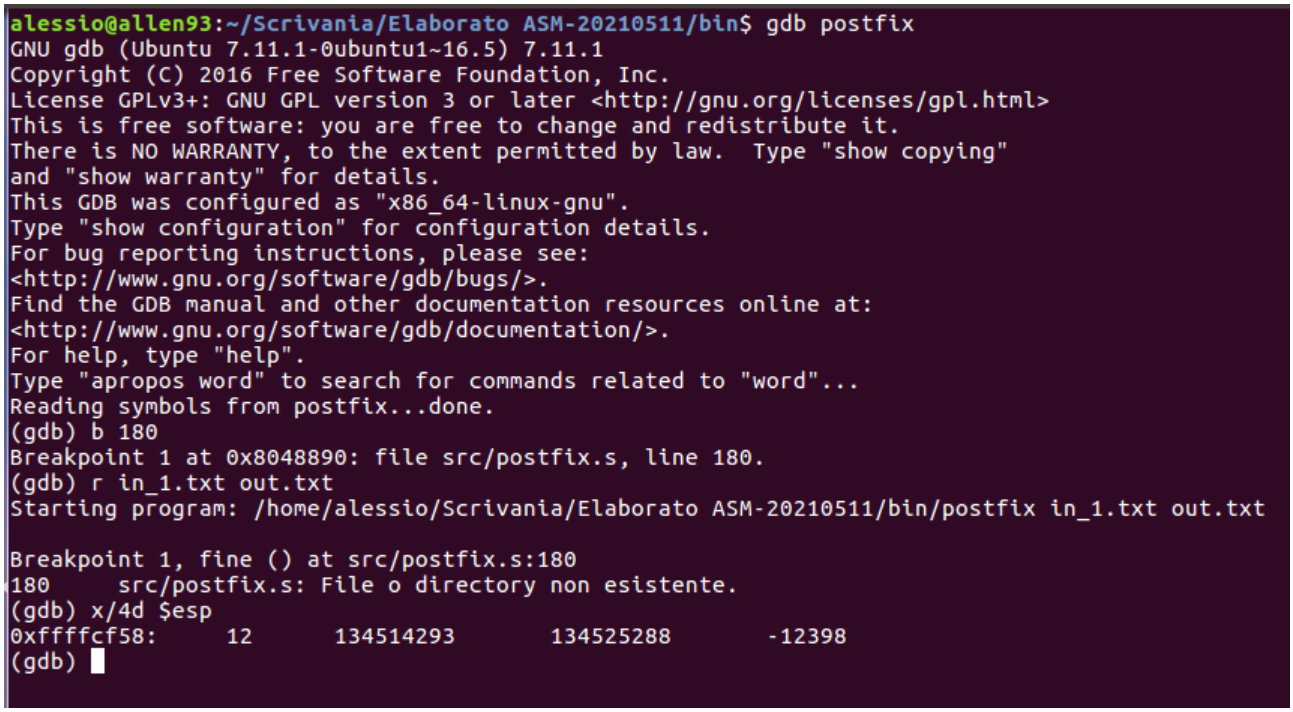
GDB

In questa sezione verranno mostrate delle istanze di debugging:



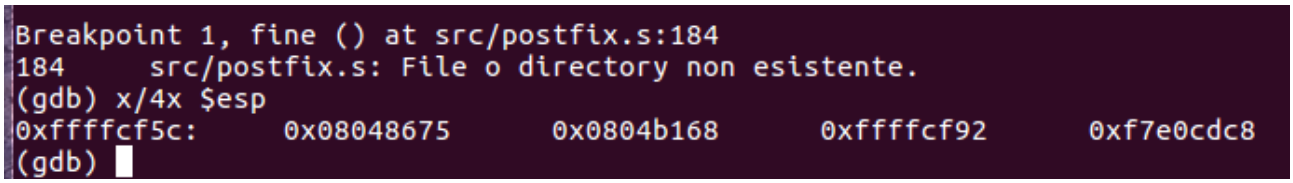
In questa immagine si può notare il risultato appena estratto dalla cima dello stack (12), messo in eax(popl %eax). La stringa di input è 30 2 + 20 - .

Inoltre si può notare che in esi ho un indirizzo di memoria (puntatore a input), anche in edi ho un indirizzo di memoria (puntatore a output).



In questa immagine si può notare il contenuto dello stack pointer quando la stringa di input è stata processata. Lo stack pointer punta a 12 (risultato rpn), in seconda posizione dall'alto si trova il program counter della funzione chiamante, in terza posizione dall'alto si trova l' indirizzo puntato da input ed infine nella quarta posizione dall'alto si trova l' indirizzo puntato da output.

Nell'immagine precedente si possono vedere dei riscontri con eax, esi , edi. Tutti i valori sono mostrati in formato decimale.



Questa immagine rappresenta sempre l'istanza mostrata in precedenza, però con valori in formato esadecimale. In questa istanza si possono notare i veri valori degli indirizzi di memoria.

```
alessio@allen93:~/Scrivania/Elaborato ASM-20210511/bin$ gdb postfix
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from postfix...done.
(gdb) b 222
Breakpoint 1 at 0x80488c9: file src/postfix.s, line 222.
(gdb) r in_1.txt out.txt
Starting program: /home/alessio/Scrivania/Elaborato ASM-20210511/bin/postfix in_1.txt out.txt

Breakpoint 1, continua_a_dividere () at src/postfix.s:222
222     src/postfix.s: File o directory non esistente.
(gdb) x/16c ($esi)
0x804a04f:      50 '2'  49 '1'  48 '0'  48 '0'  48 '0'  48 '0'  48 '0'  48 '0'
0x804a057:      48 '0'  48 '0'  48 '0'  48 '0'  48 '0'  48 '0'  0 '\000'        0 '\000'
(gdb)
```

In questa immagine si può notare come vengono inserite le varie cifre nella stringa tmp, dopo un processo di conversione da int a char. Per prima cosa il programma estrarrà il carattere ‘2’ e lo inserirà nella prima posizione puntata da output, dopo di che estrarrà il carattere ‘1’ e lo metterà nella posizione puntata da output + 1(successiva). In questo modo inserisco nel modo giusto il numero memorizzato al contrario in tmp.

```
0xffffcf92:     73 'I'  110 'n' 118 'v' 97 'a'  108 'l' 105 'i' 100 'd' 0 '\000'
0xffffcf9a:     -1 '\377'       -1 '\377'       0 '\000'        26 '\032'
(gdb)
```

In questa immagine si può notare cosa contiene la memoria puntata da output dopo l’analisi della stringa 1 2 +@ 3 * da parte del sotto programma parsing.

```
0xffffcf92:     49 '1'  50 '2'  0 '\000'
0xffffcf9a:     -1 '\377'       -1 '\377'
(gdb)
```

In questa immagine si può notare cosa contiene la memoria puntata da output, dopo aver processato la stringa 30 2 + 20 - .

SCELTE PROGETTUALI

- In questo progetto è stata creata una funzione per eseguire il parsing della stringa. La funzione è esterna al file per una questione di ordine e leggibilità. Oltre ad un fatto di riusabilità per un progetto futuro.
- Nella funzione parsing viene alzato un flag se la stringa non è accettata, solo successivamente quando tornerà al chiamante (postfix) verrà gestita questa situazione, interrompendo subito il programma e ritornando al chiamante (funzione c). La funzione c a questo punto non farà altro che scrivere ciò che trova nella parte di memoria puntata da output, nel file di output. In questo caso troverà la scritta “Invalid” come da consegna.
- Per gestire i valori negativi viene utilizzata una flag, di conseguenza verrà cambiato il segno solo se la flag è a 1.
- Se il risultato è negativo, il programma cambia il segno e inserirà in testa il segno ‘-’ già nella prima locazione puntata da output. Così facendo si andrà a gestire un numero positivo, evitando errori e blocchi dovuti alla gestione di numeri negativi.
- In parsing se si incontra uno spazio, viene controllato se il carattere che segue è anch’esso un spazio. Se lo è il programma termina eseguendo **error_msg**, sennò continua il ciclo per fare il parsing della stringa.
- In parsing quando si incontra il carattere ‘\n’ il programma termina. l’editor di testo utilizzato per il progetto è sublime text. Questo editor non aggiunge il carattere ‘\n’ a fine stringa, altri programmi come gedit invece si. In questo progetto è stata mantenuta la scelta di usare un editor che non aggiunge ‘\n’ a fine stringa, così da poter gestire il carattere ‘\n’ come non ammesso.