UNIVERSITÀ DEGLI STUDI DI VERONA

CORSO DI LAUREA IN INFORMATICA

Elaborato Assembly

Laboratorio Architettura degli Elaboratori

Notazione polacca inversa (RPN)

Federico Segala (VR457144)
Davide Zamboni (VR455880)

A.A 2020/2021

Indice

Descrizione del progetto

Si scriva un programma in assembly che legga in input una stringa rappresentante un'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.

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.
- 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.

Variabili utilizzate

Di seguito le variabili utilizzate nei rispettivi file:

File **postfix.s:**

- **risultato_tmp:** variabile di tipo long dove verrà salvato temporaneamente il numero convertito dalla stringa che poi verrà aggiunto allo stack.
- **negativo_flag:** flag che si alza qualora venga letto il simbolo ‘-‘ e come carattere successivo un numero. Verrà utilizzato per rendere negativo un numero letto dalla stringa.
- **stack_counter:** variabile di tipo long che mi permette di contare il numero di elementi aggiunti allo stack in modo tale da non avere errori durante il passaggio alle funzioni.

File **save_output.s:**

- **numtmp:** stringa contenente il risultato finale di lunghezza 12 poiché il numero più grande a 32 bit ha 10 cifre al quale si aggiungono il carattere di fine stringa ‘\0’ e, in caso, anche il segno ‘-‘.
- **negativo_flag:** flag che si alza quando viene passato alla funzione un numero negativo. Permette quindi di salvare il numero in stringa preceduto dal carattere ‘-‘.

File **invalid.s:**

- **stringa:** stringa contenente i caratteri “Invalid” che dovrà poi essere passata al main.

Etichette rilevanti del file *postfix.s*

Etichetta **loop:**

Tale etichetta permette l'acquisizione della stringa e verifica se il carattere letto è un numero oppure un operatore. Nel caso il carattere letto sia uno spazio passa alla funzione *increment*, mentre nel caso in cui sia il carattere di fine stringa passa all' etichetta *salvataggio_output*, entrambe successivamente descritte. In tutti gli altri casi salva la stringa "invalid".

Etichetta **increment:**

Una volta letto uno spazio, se il carattere precedente era un numero, controlla se il negativo_flag era alzato. In quel caso converte il numero in negativo. Successivamente salva il contenuto di %eax nello stack.

Etichetta **gestione_meno:**

Attraverso la lettura del carattere successivo tale etichetta permette di capire se il segno '-' è riferito ad un numero negativo oppure all'operazione di sottrazione. Nel primo caso passa all'etichetta che alza il negativo flag, nel secondo invece esegue la sottrazione tra gli ultimi due elementi presenti sullo stack.

Passaggio di valori alle funzioni create

Funzione **somma_funz.s**

La funzione chiamante alloca spazio sullo stack per il salvataggio del risultato. Una volta entrati nella funzione questa acquisisce i due operandi presenti sullo stack e ne esegue la somma. Il risultato viene caricato nella locazione già predisposta dal chiamante. Una volta usciti dalla funzione, il chiamante si occuperà di scaricare i due operandi dallo stack.

Funzione **sottrazione_funz.s**

Si comporta analogamente alla funzione somma_funz, eseguendo questa volta la sottrazione.

Funzione **moltiplicazione_funz.s**

Si comporta analogamente alla funzione somma_funz, eseguendo la moltiplicazione.

Funzione **save_output.s**

Tale funzione permette di convertire il risultato contenuto in %eax in stringa e di restituirlo al main, salvando il primo carattere della stringa nel registro %edi. Per fare ciò si avvale della stringa *numtmp* nella quale andrà a salvare la stringa che poi sarà invertita. Tale stringa è ottenuta attraverso ripetute divisioni per dieci del numero. In caso il numero da convertire sia negativo viene aggiunto il carattere '-' e il numero viene convertito a positivo.

Funzione **invalid.s**

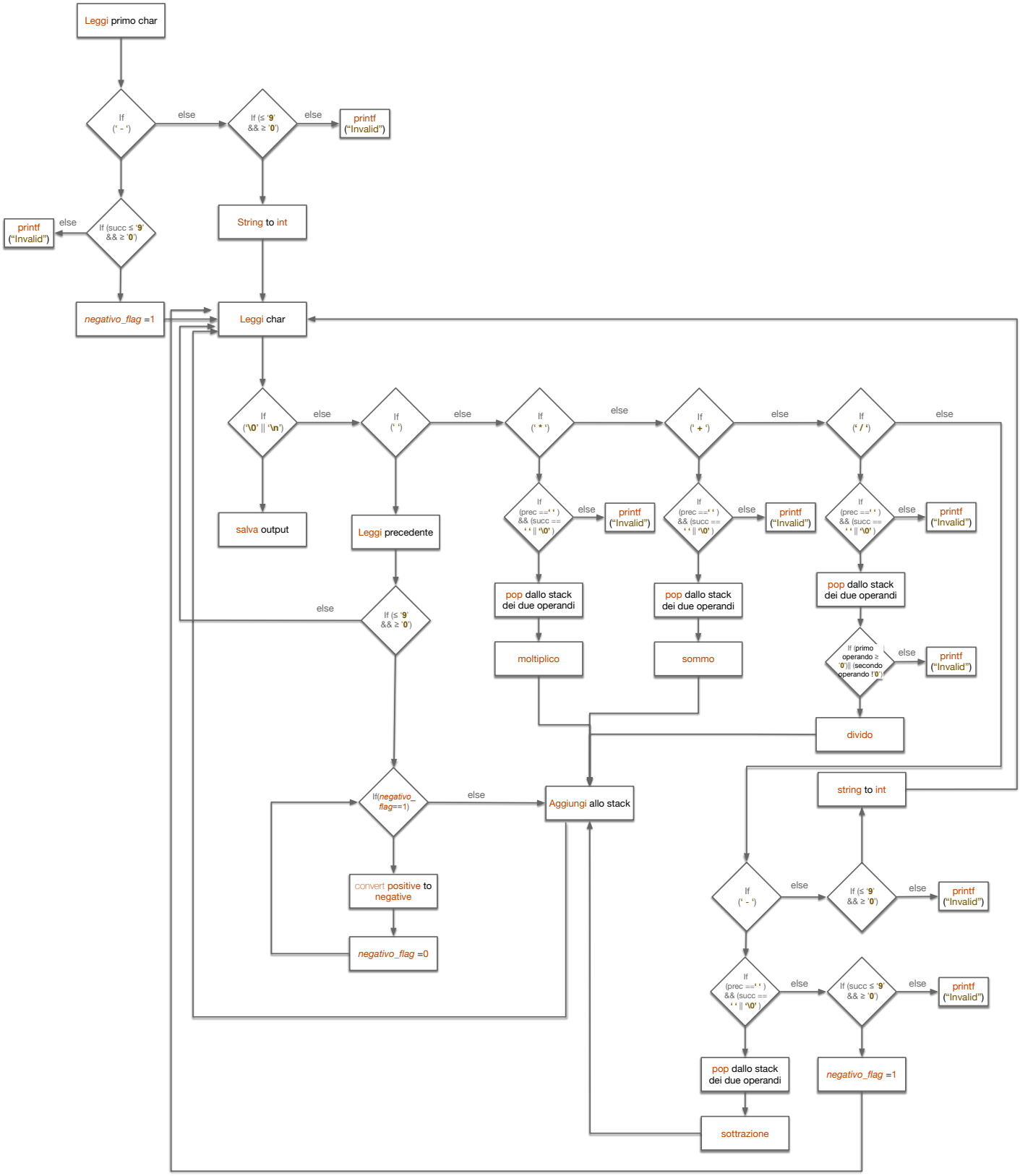
Questa funzione permette di copiare la stringa "invalid" nel registro %edi (definita nella sezione dati del file *invalid.s*) in modo tale che venga salvata nel valore di output del main.

Postfix.s

Anche lo stesso postifx.s è una funzione che a sua volta viene chiamata nel file main.c. Tale funzione ricava la posizione dei due parametri di input e output passati dal main, salvando rispettivamente il loro indirizzo iniziale in %esi ed %edi. Se la stringa è corretta, dopo aver calcolato il risultato, lo salva come stringa nel secondo parametro, altrimenti salva la stringa "Invalid" nel file di output.

Diagramma di flusso del codice

Di seguito il diagramma di flusso che riassume il comportamento del codice assembly:



Scelte Progettuali

Durante la scrittura del codice sono state svolte diverse **ottimizzazioni**, come il salvataggio di valori con una lunghezza limitata solo in una parte del registro.

Oltre al controllo che non ci siano simboli o caratteri non validi è stato implementato anche il **controllo** della correttezza generale **della scrittura della stringa**, attraverso una verifica del carattere precedente (eccetto il primo) e del successivo degli operandi e degli operatori. È stato coperto anche il caso particolare in cui la stringa di input sia vuota.

A livello progettuale è stato deciso inoltre di rendere valida anche una **stringa avente più spazi** consecutivi tra un valore e l'altro in modo tale da essere più flessibile (se avessimo voluto renderla non valida sarebbe bastato aggiungere un' istruzione di *compare* con il carattere spazio nell'etichetta *increment_number*). Sempre per rendere il programma ulteriormente flessibile è stato deciso inoltre di rendere valida anche una **stringa terminante col carattere '***n'*****. Il programma quindi in entrambi i casi procederà con lo svolgimento delle operazioni normalmente, producendo il rispettivo risultato.

Altro elemento fondamentale nel progetto è stata la **gestione dello stack**. Per evitare errori di segmentazione infatti abbiamo ritenuto opportuno introdurre un contatore che si incrementa o decrementa ogni volta che qualcosa viene aggiunto o tolto dalla pila. Questo fa sì che in fase di creazione della stringa di output l'ultimo valore sulla pila sia quello precedentemente presente in fase di chiamata della funzione postfix.s.

Per chiarezza del codice infine è stato deciso di **scomporre su più file la funzione postfix.s**. Sono state separate rispettivamente la funzione che stampa la stringa invalid, la somma, sottrazione, moltiplicazione e infine il salvataggio del risultato nella stringa di output. È stato deciso di mantenere altre funzioni all'interno del codice di postfix.s (es. divisione) in quanto la loro scomposizione sarebbe risultata macchinosa e poco intuitiva, data la struttura del codice principale.