Relazione progetto finale Architetture degli Elaboratori Maggio 2025 - A.A. 2024/2025

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Silvio Santoriello

silvio.santoriello@edu.unifi.it

Mat. 7158636

Indice

Parsing della stringa di input

1.1 Intro

Secondo le specifiche del progetto, l'input viene passato tramite stringa *listInput* all'interno della sezione *.data*. e ogni comando ben formattato è separato da ∼ (carattere ASCII 126).

Il main trasferisce immediatamente il controllo alla funzione PARSING, che analizzerà carattere per carattere il *listInput*.

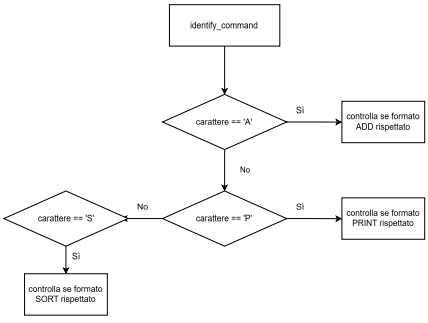
1.2 PARSING, identify_command, process_x_command

Il parsing ha come loop principale *parsing_loop* che controlla innanzitutto la presenza iniziale di spazio o tilde e nel caso li ignora, proseguendo la scansione.

Ho preferito dividere la fase di parsing in più parti:

- ◇ identificazione del comando (tramite *identify_command*)
- ◇ validazione del formato del comando (tramite *process_**x**_command* e *verify_**x**_format)*
- ◇ esecuzione/non esecuzione sulla base del risultato della validazione

Il comando, tramite l'iniziale *identify_command*, viene identificato attraverso una serie di controlli annidati, riassunti in parte dalla 1.1(a).



(a) Identificzione comando

```
identify_command:
        addi sp,sp,-4
        sw ra,0(sp)

        lb t0,0(s1)

        li t1,65    #inizia per A?
        beq t0,t1,process_add_command

        li t1, 80   #inizia per P?
        beq t0, t1, process_print_command

        li t1, 83   #inizia per S?
        beq t0, t1, process_sort_command

        li t1, 82   #inizia per R?
        beq t0, t1, process_rev_command

        li t1, 68   #inizia per D?
        beq t0, t1, process_del_command

        j invalid_command
```

(b) Codice

Figura 1.1:

Se il controllo non viene trasferito, scorro la stringa (tramite *find_next_command*), finché non trovo una tilde o un fine stringa, per poi ritornare nel loop principale e ricominciare la fase di identificazione comando.
Altrimenti,la procedura *process_**x**_command* trasferisce il controllo a *verify_**x**_format* (dove **x** è il comando in questione). Se il comando è correttamente formattato e finisce con un terminatore valido (spazio o ∼) allora la verifica termina correttamente (cioé il metodo ritorna **1** in *a1*)* e lo eseguo tramite *jump-and-link* alla rispettiva procedura.
Terminata l'operazione avanzo di un carattere.

1.3 verify_x_format

`parametri:` nessuno
`ritorno:` *a1* (1 - successo, 0 - fallimento)
`cosa fa:` ritorna se il formato del comando **x** è valido o meno

La funzione di verfica controlla sequenzialmente

1. se i caratteri successivi sono quelli propri del comando in questione
2. se il comando ha un parametro ammissibile (cioé un carattere ASCII fra 32 e 125
3. se il comando ha un terminatore valido (tramite *verify_command_terminator* che skippa eventuali spazi e controlla la presenza di un terminatore tilde/fine stringa valido

In caso di successo carica in *a1* il valore 1 e ritorna con successo.
In caso contrario vuol dire che è stata richiamata la procedura *invalid_format* che carica in *a1* il valore 0 e ritorna al chiamante.

* *Il valore di ritorno della funzione di validazione sarà in a1 per evitare interferenze con comandi che accettano parametri nel registro a0 come ADD e DEL*

Funzioni

2.1 Premessa

Le varie funzioni sono state implementate modularmente e sono state realizzate in modo da mantenere il più possibile valido il principio di regolarità, cruciale nell'architettura RISC-V. Tutte le funzioni che necessitano di parametri (cioé *ADD* e *DEL* li accettano nel registro *a0*, mentre tutte le funzioni che devono restituire dei valori li restituiscono in *a1*. Inoltre tutte le procedure salvano all'inizio

2.2 Funzioni principali

2.2.1 ADD

`parametri:` in *a0* carattere ASCII (sarà il DATA del nodo)
`ritorno:` nessuno
`cosa fa:` aggiunge un carattere ASCII alla lista concatenata

Per effettuare la ADD di un nodo, devo per prima cosa individuare un indirizzo di memoria di 5 byte libero, pertanto richiamo la procedura *find_next_free_addr* descritta di seguito, tra le procedure di supporto. Individuato l'indirizzo della prossima locazione di memoria libera, salva nel primo byte (deputato al DATA) il carattere passato come parametro e successivamente mi allineo ai successivi 4 byte, nei quali salverò l'indirizzo del prossimo nodo, che sarà nullo in quanto il nodo successivo non esiste.
A questo punto distiguiamo il caso della prima ADD della catena dal caso di ADD successivo.

- ◇ nel caso della prima ADD: aggiorno le variabili globali $HEAD\_PTR$ e $LAST\_PTR$ con l'indirizzo della testa del nodo appena inserito
- ◇ nel caso di nodo successivo: aggiorno il $PAHEAD$ del nodo precedente con l'indirizzo del nodo attuale e $LAST\_PTR$ con con l'indirizzo del nodo corrente.

2.2.2 DEL

`parametri:` in *a0* carattere ASCII (occorrenze da eliminare)
`ritorno:` nessuno
`cosa fa:` elimina **tutti** i nodi contenenti quel carattere

Se la lista non è vuota, inizia controllando la testa e se va eliminata, aggiorno HEAD_PTR con il *PAHEAD* di quel nodo (rendo quindi il nodo successivo la nuova testa). Dopo di che diminuisco il counter dei nodi e controllo nuovamente se il nodo diventato testa va anch'esso eliminato. Se la lista diventa vuota, aggiorno anche il *LAST_PTR* e termino.
Altrimenti, se la testa non va eliminata, scorro tutti gli altri nodi e per ognuno, controlla il carattere del nodo a lui seguente e se coincide con il carattere da eliminare aggiorna di conseguenza i *PAHEAD* del nodo precedente e del nodo successivo; quando arrivo all'ultimo nodo della catena, aggiorna il *LAST_PTR*.

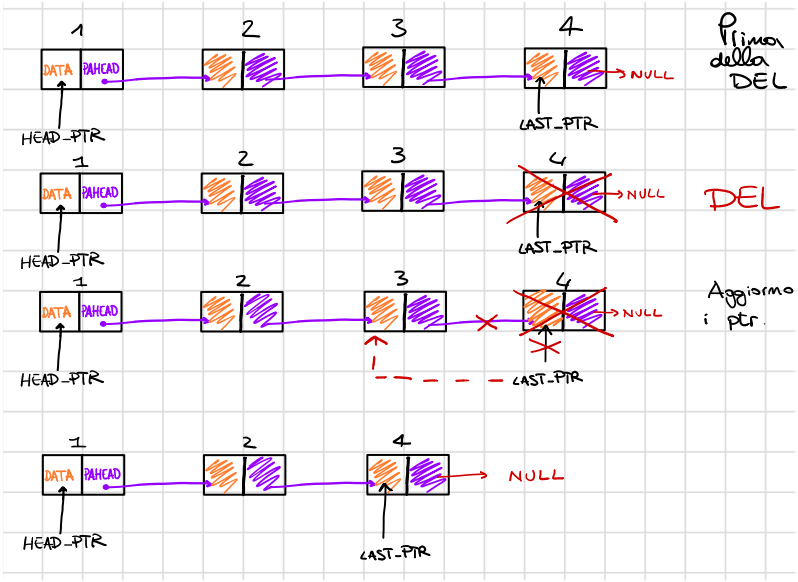


Figura 2.1: Esempio di funzionamento di DEL

2.2.3 PRINT

`parametri:` nessuno
`ritorno:` nessuno
`cosa fa:` stampa ordinatamente tutti i nodi della catena

La stampa è gestita ricorsivamente, a partire dalla testa tramite la procedura *print_recursive* (che accetta in *a0* un indirizzo, la quale stampa, se la HEAD non è zero (a causa di lista vuota o raggiungimento dell'ultimo elemento) il carattere di un nodo. Dopo di che, richiama ricorsivamente sé stessa, ma dopo aver aver scorso al nodo successivo e passando in *a0* l'indirizzo del nodo successivo.
Per quanto concerne la gestione della memoria, ad ogni chiamata di *print_recursive*

salvo nella stack sia il *ra* come avviene usualmente e anche l'*a0* che contiene l'indirizzo del nodo attuale (in totale salvo quindi 2 parole, cioé 8 bytes). Il salavataggio di *a0* mi serve per poi riprenderlo dopo aver stampato uno spazio separatore (RIPES richiede per la stampa di un carattere di passare in *a0* il carattere e in *a7* il codice 1 per poi effettuare la *ecall*).

2.2.4 SORT

`parametri:` nessuno
`ritorno:` nessuno
`cosa fa:` ordina tramite algoritmo bubble sort ricorsivo i vari nodi, sulla base delle regole di ordinamento prefissate

La funzione di ordinamento implementa bubbleSort ricorsivo con flag di controllo di avvenuti scambi. Il problema principale nel fare ciò è quello di rispettare le regole custom di ordinamento, brevemente riassunte di seguito:

- una lettera maiuscola (ASCII da 65 a 90 compresi) viene sempre ritenuta maggiore di una minuscola
- una lettera minuscola (ASCII da 97 a 122 compresi) viene sempre ritenuta maggiore di un numero
- un numero (ASCII da 48 a 57 compresi) viene sempre ritenuto maggiore di un carattere extra che implementiamo non sia una lettera funzione o numero
- non si considerano accettabili caratteri extra con codice ASCII minore di 32 o maggiore di 125.

L'approccio adottato è quello di *"categorizzare"* i vari caratteri ed effettuare per caratteri di tipo differente, confronti basati sulla categoria.
Nel mio caso:

- ◇ maiuscola → categoria 3
- ◇ maiuscola → categoria 2
- ◇ numero → categoria 1
- ◇ caratteri speciali → categoria 0
- ◇ carattere non valido → categoria -1

Tale categorizzazione viene effettuata dalla procedura *get_category* che prende in *a0* il carattere da valutare e restituisce in *a1* il numero della categoria.

Il resto del bubble sort viene gestito in maniera abbastanza semplice, tramite due cicli annidati. Il più esterno si occupa di gestire il caso in cui abbia finito la ricorsione e di resettare il flag di scambio avvenuto per ogni iterazione.
Il più interno, dopo aver salvato nella stack* controllato se è arrivato all'ultimo nodo, ottiene

*Ho decrementato lo stack pointer di 8 sebbene i byte che utilizzi siano effettivamente 7 (4 per contenere un indirizzo, e poi 3 per contenere il DATA del nodo corrente, del successivo e il contatore dei nodi ancora da processare) per evitare problemi di allineamento

le categorie del carattere corrente e quella del successivo e sulla base dei valori restituiti valuta se effettuare o meno lo swap fra i DATA dei due nodi.[†] Lo swap avviene appoggiandomi alla stack piuttosto che a registri temporanei e infine mettendo a *true* il flag di scambio avvenuto. [‡]

2.2.5 REV

```
parametri: nessuno
ritorno: nessuno
cosa fa: inverte la lista concatenata
```

2.3 Funzioni di supporto

Le funzioni principali *ADD* e *SORT* all'interno del loro corpo fanno riferimento a delle procedure di supporto: in particolare *ADD* usa *find_next_free_addr* per individuare 5 byte liberi in memoria per l'inserimento di un nodo, mentre *SORT* utilizza *get_category* per determinare la categoria del carattere.

2.3.1 *find_next_free_addr*

```
parametri: nessuno
ritorno: indirizzo di memoria (word) in a1
cosa fa: restituisce un indirizzo all'inizio di una porzione di 5 byte liberi
```

La funzione in sé non è particolarmente complicata: scansiona sequenzialmente ogni byte a partire da un indirizzo base fisso (*0x20000000*)* finché non ne individua 5 consecutivi liberi, ritornando poi l'indirizzo della testa e aggiornando la variabile globale *s5*che contiene la il byte immediatamente successivo ai 5 appena individuati, funzionale per scansioni successive.

2.3.2 *get_category*

```
parametri: carattere (1 byte) in a0
ritorno: categoria (intero) in a1
cosa fa: restituisce la categoria di un carattere (si veda 2.2.4)
```

La procedura è in realtà abbastanza semplice: preso un carattere, controlla innanzitutto la sua ammissibilità (cioé se è un ASCII tra 32 e 125), poi se è una lettera maiuscola (ASCII fra 65 e 90), maiuscola ...

[†] Non è necessario agire sui puntatori, ma semplicemente sulle informazioni contenute

[‡] L'uso di un flag mi permette di evitare, nel caso di catena già ordinato, di effettuare scambi superflui

* Ho scelto questo indirizzo di partenza per evitare del tutto conflitti con *listInput* molto lunghi

Registri, memoria e conclusioni

3.1 Convenzioni sui registri

Ho utilizzato 5 registri (*s1-s5*) per memorizzare variabili globali utili nel corso dell'esecuzione:

`s1:` contiene la testa alla stringa *listInput*
`s2:` contiene *HEAD_PTR*, puntatore alla testa del primo nodo
`s3:` contiene *LAST_PTR*, puntatore alla testa dell'ultimo nodo
`s4:` contiene un contatore al numero di nodi nella lisat
`s5:` contiene il *NEXT_FREE_ADDR*, successivo indirizzo libero per eventuale memorizzazione di nuovi nodi

3.2 Gestione della stack

Oltre al classico salvataggio del *ra* per il salvataggio dell'indirizzo di ritorno nel caso di procedure annidate o ricorsive, la stack viene usata nell'ambito della *SORT* vista la necessità di più registri temporanei per il salvataggio dei caratteri coinvolti e delle loro categorie.
Altro caso in cui l'uso della stack è necessario è durante l'operazione di *REV*, dove salvo i vari *DATA* dei vari caratteri, utilizzando 4 byte e non 1 solo per evitare problematiche di allineamento.

3.3 Note a margine

Il codice è stato testato sia su Linux che su Windows (ho notato problemi con la codifica del file e in particolare con le lettere accentuate) tramite Ripes v2.2.5 . Lo sviluppo è avvenuto utilizzando congiuntamente Visual Studio Code per un editing migliore e testando i risultati con Ripes. Si segnala inoltre l'utilizzo del modello IA Claude 3.7 Sonnet, di supporto alla stesura del codice e di particolare aiuto nella progettazione della della SORT e della logica di PARSING.

TEST

Di seguito alcune stringhe di test aggiuntivi (nel caso di lista vuota, stringa mal formattata, ordinamento ...), corredate di relativi output*:

```
    ADD(a) ~ ADD(b) ~ ADD(c) ~ PRINT ~ SORT ~ PRINT ~ DEL(b) ~ PRINT ~ REV ~
PRINT

    a b c
    a b c
    a c
    c a
    Program exited with code: 0


    ADD(a) ~ add(b) ~ ADD(cc) ~ ADD (d) ~ ADD~ ADD() ~ SORTA ~ PRINT ~ DEL()
~ DEL(a,b) ~ PRINT

   a
   a
   Program exited with code: 0


    ADD(a) ~ ADD(.) ~ ADD(2) ~ ADD(E) ~ ADD(r) ~ ADD(4) ~ ADD(,) ~ ADD(w) ~ PRINT
~ SORT ~ PRINT


    a . 2 E r 4 , w
    , . 2 4 a r w E

    Program exited with code: 0
```

*Le stringhe di test fornite nelle specifiche sono già state testate e i risultati sono quelli attesi