RELAZIONE PROGETTO – ARCHITETTURE DEGLI ELABORATORI (2022 – 2023)

AUTORE

Matricola: 7072913 Nome: Matteo Pascuzzo
Email: matteo.pascuzzo@stud.unifi.it Data di consegna: 29/05/2023

La seguente relazione verte sulla consegna del progetto assegnato per l'esame di Architetture degli Elaboratori. Lo scopo del progetto era quello di **implementare un codice RISC-V che gestisce 7 possibili operazioni fondamentali per una lista concatenata circolare**, le quali sono:

- ADD – Inserimento di un elemento
- DEL – Rimozione di un elemento
- PRINT – Stampa della lista
- SORT – Ordinamento della lista
- SDX – Shift a destra (rotazione in senso orario) degli elementi della lista
- SSX – Shift a sinistra (rotazione in senso antiorario) degli elementi della lista
- REV – Inversione degli elementi della lista

Ogni elemento della lista ha una dimensione di 5 byte, cosr suddivisi:

- DATA(Byte 0): contiene l'informazione
- PAHEAD (Byte 1-4): puntatore all'elemento successivo, o a sø stesso se unico elemento della lista

Il byte di informazione contenuto in ciascun nodo della lista rappresenta un carattere in codice ASCII i quali, tuttavia, non sono tutti accettabili come dato all'interno della lista. **Il range di dati accettabili dal programma è rappresentato dai codici ASCII da 32 a 125 compresi**.

MAIN PROCEDURE

Il programma è costituito da una procedura principale che elabora l'unico input inseribile dall'utente, dichiarato come una variabile string nominata *listInput* nel campo .data del codice. Lo scopo di tale stringa è quello di contenere i comandi da passare al programma per la gestione della stringa concatenata circolare.
Tali comandi devono essere separati dal carattere '~' (ASCII 126), **inoltre la stringa di input non dovrà contenere più di 30 comandi.**

Di seguito una porzione significativa di codice e la seguente spiegazione:

```
.data
listInput: .string "ADD(])-DEL(]) -PRINT - ADD(g) -REV - ADD($) -ADD(i) - ADD(()- ADD(6) - SDX - PRINT -DEL(g) - ADD(N)-ADD(D) - SSX -
newline: .string "\n"

.text
la s0 listInput
li s1 0                                 #contatore per scorrere la stringa in input
li s2 0                                 #contatore numero comandi
li s3 30                                #numero max comandi
li s4 0x00500000                        #indirizzo di memoria del primo elemento - PAHEAD
li s5 0x00500000                        #contatore di ciclo posizionale degli elementi
li s6 0                                 #contatore del numero di elementi nella lista concatenata
li s7 7                                 #flag


############################
############ Main ##########
############################

verifica_input_comando:
    add t1 s0 s1
    lb t2 0(t1)                         #punto alla prima lettera della stringa

    beq t2 zero end_main                #abbiamo raggiunto la fine della stringa

    beq s2 s3 end_main                  #abbiamo raggiunto il numero max di comandi ammessi

    li t3 32                            #carico il char Space in un registro
    beq t2 t3 aumenta_contatore_stringa #se il char a cui punto è uno spazio allora passo al char sucessivo

    li t3 65                            #carico A in un registro
    beq t2 t3 verifica_ADD

    li t3 68                            #carico D in un registro
    beq t2 t3 verifica_DEL

    li t3 80                            #carico P in un registro
    beq t2 t3 verifica_PRINT

    li t3 83                            #carico S in un registro --> rimandare ad un unico metodo che checka il char successivo
    beq t2 t3 verifica_S

    li t3 82                            #carico R in un registro
    beq t2 t3 verifica_REV

    j scorri_prossima_tilde
```

La procedura principale è volta quindi alla lettura di ogni singolo byte della stringa in input fino a quando non troviamo uno 0 (Carattere ASCII di fine stringa). Grazie all'ausilio di un contatore salvato all'interno del registro **s1** riesco a puntare al carattere della stringa corrente salvandone nel registro temporaneo **t2** il relativo contenuto, tale contatore sarà incrementato nelle porzioni del codice volte al controllo della corretta battitura dei vari comandi.
Una volta verificato che tale carattere non sia 0 (stringa terminata), o che non abbiamo raggiunto il limite massimo di comandi ammessi, bisogna controllare che il carattere in questione non sia uno **Space** (ASCII 32) poiché in tale caso dovremmo scorrere la stringa fin quando non troviamo qualcosa di diverso dallo spazio ma che non sia tilde. Ciò viene fatto tramite la seguente funzione di supporto:

```
aumenta_contatore_stringa:
    beq t2 zero end_main
    addi s1 s1 1
    j verifica_input_comando
```

la quale incrementa il contatore e salta allo switch iniziale in modo da poter leggere il carattere seguente della stringa.

Una volta operati questi controlli, la procedura continua con l'implementazione di uno **switch** che permette di eseguire il corretto algoritmo confrontando il carattere corrente con i caratteri ASCII ammissibili i quali sono: A(65), D(68), P(80), S(83), R(82). Se tale carattere non corrisponde a nessuno dei comandi per la gestione della lista, allora vuol dire che il comando passato nella stringa non è valido e quindi si deve passare al prossimo, per fare ciò usiamo la seguente funzione di supporto:

```
338 scorri_prossima_tilde:
339     beq t2 zero end_main
340     li t5 126                               #carico tilde in un registro
341     beq t2 t5 aumenta_contatore_stringa     #se il carattere a cui punto è tilde
342     addi s1 s1 1
343     add t1 s0 s1
344     lb t2 0(t1)
345     j scorri_prossima_tilde
346
```

la quale incrementa il contatore a ogni ciclo e quindi scorre la stringa fino a quando non trova un carattere '~', una volta trovato richiama la precedente funzione di supporto la quale, incrementando anch'essa il contatore di 1 permette, ritornando allo switch, di leggere il carattere successivo alla tilde alla quale volevamo andare. Tutto questo viene fatto usando sempre i soliti registri e salvando nel registro **t5** il codice ASCII del carattere '~'.
Il controllo del carattere permette quindi di eseguire il corretto algoritmo di controllo della correttezza del comando passato in input implementato in una procedura separata, diversa per ogni comando. **Ogni algoritmo** si divide in **3 procedure differenti** le quali servono a verificare che i comandi siano **corretti, completi ed unici** (ovvero non siano presenti 2 comandi non separati da tilde), vi è infatti una funzione che scorre il comando fino alla fine, una che si assicura che sia l'unico comando fino alla prossima tilde ed una che salta all'implementazione di quel comando.
In seguito sono riportati i vari algoritmi per ogni tipo di comando:

```
56 verifica_ADD:
57     beq s2 s3 scorri_prossima_tilde          #se sono al 31esimo comando non lo eseguo
58     addi s1 s1 1                             #aumento il contatore
59     add t1 s0 s1
60     lb t2 0(t1)                              #punto alla seconda lettera
61     li t3 68                                 #carico la lettera D in un registro
62     bne t2 t3 scorri_prossima_tilde
63
64     addi s1 s1 1
65     add t1 s0 s1
66     lb t2 0(t1)                              #punto alla terza lettera
67     bne t2 t3 scorri_prossima_tilde
68
69     addi s1 s1 1
70     add t1 s0 s1
71     lb t2 0(t1)                              #punto a quella che dovrebbe essere la tonda aperta
72     li t3 40                                 #metto la tonda aperta in un registro
73     bne t2 t3 scorri_prossima_tilde
74
75     addi s1 s1 1
76     add t1 s0 s1
77     lb t2 0(t1)                              #punto a quello che dovrebbe essere il carattere
78     li t4 32                                 #carico ASCII 32 (Space) in un registro
79     li t5 125                                #carico ASCII 125 (}) in un registro
80     blt t2 t4 scorri_prossima_tilde          #se il char è minore di 32 skippo
81     bgt t2 t5 scorri_prossima_tilde          #se il char è maggiore di } skippo
82     addi a0 t2 0                             #SALVO IN a0 IL CHAR CORRETTO
83
84     addi s1 s1 1
85     add t1 s0 s1
86     lb t2 0(t1)                              #punto a quello che dovrebbe essere la tonda chiusa
87     li t3 41                                 #metto la tonda chiusa in un registro
88     bne t2 t3 scorri_prossima_tilde
89
90 comando_unico_ADD:
91     addi s1 s1 1
92     add t1 s0 s1
93     lb t2 0(t1)
94     li t5 126
95     li t4 32
96     beq t2 t5 fine_verifica_ADD              #se trova un char tilde allora il comando è unico
97     beq t2 zero ADD                          #sono arrivato fin qui, quindi il comando è corretto ma trovo uno zero quindi sono a fine stringa: eseguo il comando
98     bne t2 t4 scorri_prossima_tilde          #se trova un char che non è Space, il comando non è unico e si va al prossimo comando, se trova space va avanti
99     j comando_unico_ADD
00
01 fine_verifica_ADD:
02     jal ADD
03     li a0 0                                  #resetto l'argomento da passare alle funzioni
04     addi s2 s2 1                             #aumento il contatore dei comandi effettuati
05     j aumenta_contatore_stringa
```

questa funzione, così come tutte le altre, si assicura di essere ancora nel range del numero dei caratteri ammissibili (salvato nel registro **s3**) confrontandolo col numero dei comandi corrente (salvato nel registro **s2**) dopodiché, tramite delle operazioni **bne** inizia a confrontare il carattere corrente (salvato nel registro temporaneo **t2**) con il carattere aspettatosi dalla definizione del comando stesso: es. dopo la 'A' mi aspetto che se il comando è corretto, il carattere esattamente successivo sia una 'D' etc.
Non appena ci si accorge che uno dei carattere non è quello aspettato, o che sono presenti degli spazi tra una lettera del comando ed un'altra, si richiama la funzione **scorri_prossima_tilde,** la quale, come precedentemente spiegato, grazie all'ausilio della funzione **aumenta_contatore_stringa** ci conduce al primo byte immediatamente successivo alla successiva tilde. Il comando ADD, così come il comando DEL, a differenza degli altri comandi si aspetta anche il **passaggio di un parametro** (codice ASCII da 32 a 125 compresi) **tra parentesi tonde.** Il controllo della validità del carattere viene eseguito in questa procedura e, se esso è valido, viene caricato nel registro **a0** in modo da essere pronto nel momento in cui si va a chiamare la relativa funzione. Se il carattere da passare alla funzione non è valido allora il comando è da considerarsi non valido e dunque si richiama la funzione scorri_prossima_tilde.
Successivamente ci si deve assicurare che tale comando sia **unico** ossia che a seguire tale comando ci siano solo spazi ed infine una tilde oppure 0, ciò avviene tramite la funzione comando_unico_ADD , la quale continua a scorrere la stringa e appena trova una tilde allora chiama la relativa funzione, se trova 0 significa che il comando è corretto ed è l'ultimo comando perchè la stringa è finita, altrimenti se trova un carattere che non è né space né tilde vuol dire che il comando non è valido e si scorre fino al prossimo tramite le funzioni di supporto. L'ultima funzione, nominata fine_verifica_DEL esegue una **jump and link** al relativo comando, una volta finita l'esecuzione di tale comando resetta il registro **a0** impostandolo a zero ed incrementa il numero di comandi eseguiti.
Si riporta anche l'algoritmo per il controllo della correttezza del comando DEL, la quale implementazione è identica a quella di ADD:

```
verifica_DEL:
    beq s2 s3 scorri_prossima_tilde         #se sono al 31esimo comando non lo eseguo
    addi s1 s1 1                            #aumento il contatore
    add t1 s0 s1
    lb t2 0(t1)                             #punto alla seconda lettera
    li t3 69                                #carico la lettera E in un registro
    bne t2 t3 scorri_prossima_tilde

    addi s1 s1 1
    add t1 s0 s1
    lb t2 0(t1)                             #punto alla terza lettera
    li t3 76                                #carico la lettera L in un registro
    bne t2 t3 scorri_prossima_tilde

    addi s1 s1 1
    add t1 s0 s1
    lb t2 0(t1)                             #punto a quella che dovrebbe essere la tonda aperta
    li t3 40                                #metto la tonda aperta in un registro
    bne t2 t3 scorri_prossima_tilde

    addi s1 s1 1
    add t1 s0 s1
    lb t2 0(t1)                             #punto a quello che dovrebbe essere il carattere
    li t4 32                                #carico ASCII 32 (Space) in un registro
    li t5 125                               #carico ASCII } in un registro
    blt t2 t4 scorri_prossima_tilde         #se il char è minore di 32 skippo
    bgt t2 t5 scorri_prossima_tilde         #se il char è maggiore di } skippo
    addi a0 t2 0                            #SALVO IN a0 IL CHAR CORRETTO

    addi s1 s1 1
    add t1 s0 s1
    lb t2 0(t1)                             #punto a quello che dovrebbe essere la tonda chiusa
    li t3 41                                #metto la tonda chiusa in un registro
    bne t2 t3 scorri_prossima_tilde

comando_unico_DEL:
    addi s1 s1 1
    add t1 s0 s1
    lb t2 0(t1)
    li t5 126
    li t4 32
    beq t2 t5 fine_verifica_DEL             #se trova un char tilde allora il comando è unico
    beq t2 zero DEL
    bne t2 t4 scorri_prossima_tilde         #se trova un char che non è Space, il comando non è unico e si va al prossimo comando, se trova space va avanti
    j comando_unico_DEL

fine_verifica_DEL:
    jal DEL
    li a0 0                                 #resetto l'argomento da passare alle funzioni
    addi s2 s2 1                            #aumento il contatore dei comandi effettuati
    j aumenta_contatore_stringa
```

L'implementazione dell'algoritmo per la correttezza del comando PRINT è identica ai comandi ADD e DEL con l'unica differenza che, ovviamente non è previsto il passaggio di nessun parametro al rispettivo comando. Vengono usati sempre gli stessi registri per lo scorrimento della stringa, per la memorizzazione del carattere corrente e per il salvataggio di numeri usati per poter fare i confronti. In seguito il codice:

```
verifica_PRINT:
    beq s2 s3 scorri_prossima_tilde          #se sono al 31esimo comando non lo eseguo
    addi s1 s1 1
    add t1 s0 s1
    lb t2 0(t1)
    li t3 82                                 #metto il char R in un registro
    bne t2 t3 scorri_prossima_tilde

    addi s1 s1 1
    add t1 s0 s1
    lb t2 0(t1)
    li t3 73                                 #metto il char I in un registro
    bne t2 t3 scorri_prossima_tilde

    addi s1 s1 1
    add t1 s0 s1
    lb t2 0(t1)
    li t3 78                                 #metto il char N in un registro
    bne t2 t3 scorri_prossima_tilde

    addi s1 s1 1
    add t1 s0 s1
    lb t2 0(t1)
    li t3 84                                 #metto il char T in un registro
    bne t2 t3 scorri_prossima_tilde

comando_unico_PRINT:
    addi s1 s1 1
    add t1 s0 s1
    lb t2 0(t1)
    li t5 126
    li t4 32
    beq t2 t5 fine_verifica_PRINT            #se trova un char tilde allora il comando è unico
    beq t2 zero PRINT
    bne t2 t4 scorri_prossima_tilde          #se trova un char che non è Space, il comando non è unico e si va al prossimo comando; se trova space va avanti
    j comando_unico_PRINT

fine_verifica_PRINT:
    jal PRINT
    li a0 0                                  #resetto l'argomento da passare alle funzioni
    addi s2 s2 1                             #aumento il contatore dei comandi effettuati
    j aumenta_contatore_stringa
```

Si riporta anche l'algoritmo per il controllo della correttezza del comando REV, la quale implementazione è identica a quella di PRINT:

```
verifica_REV:
    beq s2 s3 scorri_prossima_tilde          #se sono al 31esimo comando non lo eseguo
    addi s1 s1 1
    add t1 s0 s1
    lb t2 0(t1)
    li t3 69
    bne t2 t3 scorri_prossima_tilde

    addi s1 s1 1
    add t1 s0 s1
    lb t2 0(t1)
    li t3 86
    bne t2 t3 scorri_prossima_tilde

comando_unico_REV:
    addi s1 s1 1
    add t1 s0 s1
    lb t2 0(t1)
    li t5 126
    li t4 32
    beq t2 t5 fine_verifica_REV              #se trova un char tilde allora il comando è unico
    beq t2 zero REV
    bne t2 t4 scorri_prossima_tilde          #se trova un char che non è Space, il comando non è unico e si va al prossimo comando; se trova space va avanti
    j comando_unico_REV

fine_verifica_REV:
    jal REV
    li a0 0                                  #resetto l'argomento da passare alle funzioni
    addi s2 s2 1                             #aumento il contatore dei comandi effettuati
    j aumenta_contatore_stringa
```

Per quanto riguarda i 3 comandi la cui lettera iniziale è 'S', per essi è stato deciso di implementare un altro **switch** il quale scorre la stringa di una posizione, andando a leggere la seconda lettera del comando e indirizzando il flusso di conseguenza. In seguito il codice:

```
verifica_S:
    addi s1 s1 1
    add t1 s0 s1
    lb t2 0(t1)
    li t3 79                                #metto il char O in un registro
    li t4 68                                #metto il char D in un registro
    li t5 83                                #metto il char S in un registro

    beq t2 t3 verifica_SORT
    beq t2 t4 verifica_SDX
    beq t2 t5 verifica_SSX

    j scorri_prossima_tilde
```

L' implementazione degli algoritmi per il controllo della correttezza dei comandi SORT, SDX ed SSX segue quella dei comandi PRINT e REV, si procede quindi a riportarne il codice:

```
verifica_SORT:
    beq s2 s3 scorri_prossima_tilde         #se sono al 31esimo comando non lo eseguo
    addi s1 s1 1
    add t1 s0 s1
    lb t2 0(t1)
    li t3 82
    bne t2 t3 scorri_prossima_tilde

    addi s1 s1 1
    add t1 s0 s1
    lb t2 0(t1)
    li t3 84
    bne t2 t3 scorri_prossima_tilde

comando_unico_SORT:
    addi s1 s1 1
    add t1 s0 s1
    lb t2 0(t1)
    li t5 126
    li t4 32
    beq t2 t5 fine_verifica_SORT            #se trova un char tilde allora il comando è unico
    beq t2 zero SORT
    bne t2 t4 scorri_prossima_tilde         #se trova un char che non è Space, il comando non è unico e si va al prossimo comando; se trova space va avanti
    j comando_unico_SORT

fine_verifica_SORT:
    jal SORT
    li a0 0                                 #resetto l'argomento da passare alle funzioni
    addi s2 s2 1                            #aumento il contatore dei comandi effettuati
    j aumenta_contatore_stringa

verifica_SDX:
    beq s2 s3 scorri_prossima_tilde         #se sono al 31esimo comando non lo eseguo
    addi s1 s1 1
    add t1 s0 s1
    lb t2 0(t1)
    li t3 88
    bne t2 t3 scorri_prossima_tilde

comando_unico_SDX:
    addi s1 s1 1
    add t1 s0 s1
    lb t2 0(t1)
    li t5 126
    li t4 32
    beq t2 t5 fine_verifica_SDX             #se trova un char tilde allora il comando è unico
    beq t2 zero SDX
    bne t2 t4 scorri_prossima_tilde         #se trova un char che non è Space, il comando non è unico e si va al prossimo comando; se trova space va avanti

fine_verifica_SDX:
    jal SDX
    li a0 0
    addi s2 s2 1
    j aumenta_contatore_stringa


verifica_SSX:
    beq s2 s3 scorri_prossima_tilde         #se sono al 31esimo comando non lo eseguo
    addi s1 s1 1
    add t1 s0 s1
    lb t2 0(t1)
    li t3 88
    bne t2 t3 scorri_prossima_tilde

comando_unico_SSX:
    addi s1 s1 1
    add t1 s0 s1
    lb t2 0(t1)
    li t5 126
    li t4 32
    beq t2 t5 fine_verifica_SSX             #se trova un char tilde allora il comando è unico
    beq t2 zero SSX
    bne t2 t4 scorri_prossima_tilde         #se trova un char che non è Space, il comando non è unico e si va al prossimo comando; se trova space va avanti
    j comando_unico_SSX

fine_verifica_SSX:
    jal SSX
    li a0 0
    addi s2 s2 1
    j aumenta_contatore_stringa
```

OPERAZIONE ADD

La prima operazione implementata è l'operazione di ADD. Essa consiste nella creazione di un nuovo elemento della lista che contiene come informazione DATA = *char*, e viene aggiunto in coda alla lista esistente. In seguito il codice con relativa spiegazione:

```
352 ######################################
353 ################ ADD ##################
354 ######################################
355
356 ADD:
357     beq s6 zero ADD_primo_elemento              #controllo che sia il primo elemento della catena
358     sb a0 0(s5)                                 #Salvo il byte passato come parametro
359     sw s4 1(s5)                                 #salvo il puntatore
360     addi s6 s6 1
361     addi t6 s5 0
362     addi t6 t6 -5
363     j cerca_precedente
364
365 ADD_primo_elemento:
366     sb a0 0(s4)                                 #Salvo il byte passato come parametro
367     sw s4 1(s5)                                 #salvo il puntatore
368     addi s5 s5 5                                #punto alla cella in cui andrò il prossimo elemento
369     addi s6 s6 1                                #numero elementi ++
370     jr ra
371
372 cerca_precedente:
373     lb t3 0(t6)
374     bne t3 s7 collega_successivo
375     addi t6 t6 -5
376     j cerca_precedente
377
378 collega_successivo:
379     sw s5 1(t6)
380     addi s5 s5 5
381     jr ra
382
```

La funzione comincia con un primo controllo sul numero degli elementi: se esso è zero allora sto inserendo il primo elemento. Tramite una operazione **store byte** inserisco il carattere salvato in **a0** dall'algoritmo di verifica della correttezza nella locazione di memoria puntata da **PAHEAD**, ovvero l'indirizzo di memoria dal quale faccio partire la mia catena, il quale è memorizzato nel registro **s4.**
La funzione prevede anche l'utilizzo di un contatore di ciclo posizionale degli elementi (salvato nel registro **s5**), grazie al quale so sempre dove inserire il successivo elemento. Successivamente inserisco l'indirizzo di memoria del primo elemento come puntatore a se stesso, in modo tale da rispettare la circolarità della catena ed aumentiamo il numero di elementi presenti nella catena incrementando di 1 il valore contenuto nel registro **s6.**
Nel caso un cui non stiamo inserendo il primo elemento, dopo aver inserito l'elemento in memoria ed averlo fatto puntare al primo elemento tramite l'operazione di **store word,** dobbiamo collegare l'elemento appena inserito al precedente, allora scorriamo la lista verificando di non star puntando ad un elemento cancellato logicamente ed il primo che troviamo lo colleghiamo all'elemento appena inserito tramite una store word della locazione di memoria dell'ultimo elemento inserito al posto del puntatore dell'elemento che puntava al primo di modo che ora l'indirizzo di memoria che leggo è quello dell'elemento appena inserito. Una volta collegati i puntatori ritorno alla funzione fine_verifica_ADD la quale ritorna a sua volta alla porzione di codice che scorre la stringa in Input.

OPERAZIONE DEL

La seconda operazione implementata è l'operazione di DEL. Essa consiste nella ricerca dell'elemento con DATA = char esistente nella lista e, se esistente, lo elimina. Nel caso in cui pitf elementi con DATA = char siano presenti nella lista, li rimuove tutti. E' stato deciso di adottare la **cancellazione logica** (come avviene di solito nelle liste concatenate) .In seguito il codice con relativa spiegazione:

```
384 ######################################
385 ################ DEL ##################
386 ######################################
387
388 DEL:
389     addi t0 s4 0
390     addi t6 t0 5
391     lb t3 0(t0)
392     beq t3 a0 rimuovi_primo                    #funzione che rimuove SOLO il primo elemento
393
394 trova_elemento:
395     lb t3 0(t6)
396     beq t3 a0 elimina
397     lw t4 1(t0)
398     beq t4 s4 fine_DEL
399     beq t3 s7 prosegui
400     lw t4 1(t6)
401     beq t4 s4 fine_DEL
402     addi t0 t6 0
403     addi t6 t6 5
404     j trova_elemento
405
406 elimina:
407     sb s7 0(t6)
408     lw t4 1(t6)
409     beq t4 s4 cambia_PAHEAD
410     addi t6 t6 5
411     addi s6 s6 -1
412     j cerca_successivo
413
414 cambia_PAHEAD:
415     sw s4 1(t0)
416     jr ra
417
418 cerca_successivo:
419     lb t3 0(t6)
420     beq t3 a0 elimina
421     bne t3 s7 linka
422     addi t6 t6 5
423     j cerca_successivo
424
425 linka:
426     sw t6 1(t0)
427     lw t4 1(t6)
428     beq t4 s4 fine_DEL
429     addi t6 t6 5
430     j trova_elemento
431
432 prosegui:
433     addi t6 t6 5
434     j cerca_successivo
435
436 rimuovi_primo:
437     lb t4 1
438     beq s6 t4 reset
439     lb t3 0(t0)
440     beq t3 a0 rimuovi
441     bne t3 s7 aggiorna_secondo
442     addi t0 t0 5
443     j rimuovi_primo
444
445 rimuovi:
446     sb s7 0(t0)
447     addi s6 s6 -1
448     addi t0 t0 5
449     j rimuovi_primo
```

```
451 aggiorna_secondo:
452     addi t6 t0 0
453     j trova_ultimo
454
455 trova_ultimo:
456     lb t3 0(t6)
457     bne t3 s7 check_puntatore
458     addi t6 t6 5
459     j trova_ultimo
460
461 check_puntatore:
462     lw t4 1(t6)
463     beq t4 s4 collega_alla_testa
464     addi t6 t6 5
465     j trova_ultimo
466
467 collega_alla_testa:
468     addi s4 t0 0
469     sw s4 1(t6)
470     j DEL
471
472 reset:
473     sb s7 0(t0)
474     addi s4 s5 0
475     jr ra
476
477 fine_DEL:
478     jr ra
479
```

L'algoritmo scorre la lista caricando in un registro temporaneo il primo elemento della lista, il quale viene comparato all'elemento da eliminare. Se il primo elemento è proprio quello da eliminare allora si richiama la procedura **rimuovi_primo** la quale in primis controlla il numero degli elementi. Se il numero degli elementi al momento della cancellazione è maggiore di uno allora viene chiamata la procedura **rimuovi**, la quale procede alla **cancellazione logica** facendo una store byte di un carattere arbitrario scelto come flag (ASCII 7). In seguito si aggiorna il PAHEAD della lista, si decrementa il numero degli elementi nella lista e si scorre all'elemento successivo richiamando di nuovo la procedura **rimuovi_primo.** Questo avviene per coprire quella casistica nella quale **non è solo il primo elemento quello da eliminare**, ma anche il secondo etc.
Nel caso in cui l'elemento da eliminare fosse solo il primo allora si scorre la lista fino a trovare l'ultimo elemento, una volta trovato lo si collega alla nuova testa della lista (funzione **collega_alla_testa**), inserendo come puntatore dell'ultimo elemento l'indirizzo di memoria del nuovo primo elemento.
Se ,invece, mi accorgo di dover **cancellare l'unico elemento presente nella lista** (vd. funzione **rimuovi_primo**) allora la soluzione adottata è quella di far “ricominciare” una nuova lista a partire dall'indirizzo del contatore di ciclo posizionale (**s5**), il quale si trova sempre su una locazione di memoria libera.

L'algoritmo ovvia alla casistica in cui gli elementi da cancellare siano più di uno ma dei quali almeno uno sia l'elemento alla testa della catena, richiamando, una volta effettuata la cancellazione del primo elemento, tutto l'algoritmo da capo in modo da cancellare anche elementi che non si trovano alla testa della lista. Ciò avviene col metodo **trova_elemento**, il quale scorre la catena, saltando gli elementi cancellati logicamente, fino a quando non trova l'elemento da cancellare oppure vede che il puntatore dell'elemento corrente punta al primo elemento fermandosi di conseguenza. Una volta trovato procede alla cancellazione logica tramite il metodo **elimina,** controllando anche che non sia l'ultimo. Nel caso effettivamente quello fosse l'ultimo elemento allora tramite la funzione **cambia_PAHEAD** procede a inserire l'indirizzo del primo elemento come puntatore dell'elemento precedente all'ultimo. Altrimenti vuol dire che l'elemento

dal cancellare non è l'ultimo quindi la funzione **elimina** richiama la funzione **cerca_successivo** la quale scorre fino a trovare il primo elemento non cancellato logicamente e procede a collegarli tramite la funzione **linka,** la quale, una volta collegati richiama **trova_elemento** per cercare vedere se ci sono altri possibili elementi da cancellare. Il modo in cui ci si accorge di essere arrivati alla fine della lista è controllando sempre il puntatore all'elemento successivo, se esso coincide con l'indirizzo del primo elemento allora quello su cui sono attualmente è l'ultimo.

OPERAZIONE PRINT

La terza operazione implementata è l'operazione PRINT la quale stampa tutti i DATA degli elementi della lista, in ordine di apparizione.
L'algoritmo si assicura che ci siano elementi nella lista facendo un controllo, se non vi sono elementi nella lista allora ritorna senza fare niente, altrimenti scorre la lista facendo attenzione a saltare gli elementi cancellati logicamente dopodiché mette nel registro **a0** il carattere da stampare e chiama la system call per stampare caratteri, inoltre ad ogni iterazione del ciclo verifica di non essere all'ultimo elemento verificando il puntatore di ciascun elemento.
Infine, prima di terminare si assicura di andare a capo tramite una system call usando la stringa *newline* dichiarata nel campo .data del programma. In seguito il codice:

```
480 ########################################
481 ############### PRINT ##################
482 ########################################
483
484 PRINT:
485     beq s6 zero fine_print
486     addi t6 s4 0
487
488 check:
489     lb t3 0(t6)
490     bne t3 s7 stampa
491     addi t6 t6 5
492     j check
493
494 stampa:
495     lb a0 0(t6)
496     li a7 11
497     ecall
498     lw t4 1(t6)
499     beq t4 s4 fine_print
500     addi t6 t6 5
501     j check
502
503 fine_print:
504     la a0 newline
505     li a7 4
506     ecall
507     jr ra
508
```

OPERAZIONE REV

La quarta operazione implementata è l'operazione REV la quale inverte gli elementi della lista. L'idea dietro a questo algoritmo è quella che inserendo gli elementi nella stack e successivamente prelevandoli, essi risulteranno già invertiti.
L'algoritmo inizialmente controlla il numero degli elementi nella lista: se nullo ritorna senza fare niente, altrimenti alloca spazio nella stack pari a 30 byte (caso in cui abbia come listinput 30 comandi ADD validi). Le funzioni **da_catena_a_stack** e **metti_in_catena** si occupano di scorrere la catena e inserire gli elementi nella stack, una volta raggiunta la fine della catena si esegue la funzione **da_stack_a_catena** la quale fa dei **pop** dalla catena e ad ogni iterazione inserisce l'elemento nella catena. Dal momento che l'operazione REV non modifica il numero degli elementi, il modo in cui ci si ferma è sempre tramite la catena, ovvero sempre controllando i puntatori dei vari elementi. Al termine dell'algoritmo dealloco lo spazio usato per la stack e ritorno. In seguito il codice:

```
510 ######################################
511 ################ REV ##################
512 ######################################
513
514 REV:
515     beq s6 zero nessun_elemento
516     addi sp sp -30                      #massimo numero di elementi
517     addi t6 s4 0
518
519 da_catena_a_stack:
520     lb t3 0(t6)
521     bne t3 s7 metti_in_stack
522     addi t6 t6 5
523     j da_catena_a_stack
524
525 metti_in_stack:
526     sb t3 0(sp)
527     addi sp sp 1
528     lw t4 1(t6)
529     beq t4 s4 end_chain
530     addi t6 t6 5
531     j da_catena_a_stack
532
533 end_chain:
534     addi sp sp -1
535     addi t6 s4 0
536     j da_stack_a_catena
537
538 da_stack_a_catena:
539     lb t3 0(t6)
540     bne t3 s7 metti_in_catena
541     addi t6 t6 5
542     j da_stack_a_catena
543
544
545 metti_in_catena:
546     lb t3 0(sp)
547     sb t3 0(t6)
548     addi sp sp -1
549     lw t4 1(t6)
550     beq t4 s4 fine_stack
551     addi t6 t6 5
552     j da_stack_a_catena
553
554 nessun_elemento:
555     jr ra
556
557 fine_stack:
558     addi sp sp 30
559     jr ra
560
```

OPERAZIONE SORT

La quinta operazione implementata è l'operazione di SORT la quale ordina gli elementi della lista secondo il seguente ordinamento:

- 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 non sia lettera o numero

All'interno di ogni categoria vige l'ordinamento dato dal codice ASCII. Per esempio, date due lettere maiuscole x e x', x < x' se e solo se ASCII(x) < ASCII(x'). Lo stesso vale per le lettere minuscole, per i numeri e per i caratteri extra.

L'dea dietro all'implementazione del SORT è quella di dover ovviare al fatto che, essendo i caratteri extra **non tutti consecutivi** nella tabella ASCII e dovendo essi comparire per primi nell'ordinamento, urge la necessità di classificarli in base ad un'altra metrica. Quello che è stato deciso, infatti, è di considerare i caratteri come appartenenti a 3 diverse categorie:

- Lettere maiuscole → indicate con 0
- Lettere minuscole → indicate con 1
- Numeri → indicati con 2
- Extra → indicati con 3

L'algoritmo compara gli elementi a due a due (salvati nei registri temporanei **t0** e **t2**) determinando brutalmente la categoria dei due elementi (0 ,1, 2, o 3), comparandoli con gli estremi delle varie categorie di caratteri (salvate nei registri **a2** e **a3**). Una volta determinate, si procede alla funzione **compara** la quale controlla a quale categoria appartengono i due caratteri.

Se il primo carattere appartiene ad una categoria "maggiore" del secondo carattere allora il loro ordine relativo è corretto, se i caratteri appartengono alla stessa categoria allora si chiama il metodo compara_ASCII il quale ordina gli elementi semplicemente in base al loro codice ASCII mentre se il primo carattere appartiene ad una categoria "minore" del secondo carattere allora li scambio. Infine, il metodo **increment_loop** mi farà scorrere la lista per poter comparare tra loro altri due caratteri. Se i caratteri risultano ordinati allora l'algoritmo ritorna alla riga successiva alla chiamata effettuata con jal nel metodo fine_verifica_sort.

In seguito il codice:

```
#####################################
################ SORT #################
#####################################

SORT:
    lw s8, 1(s4)
    addi a6, s4, 0
    lw t3, 1(s4)
    beq t3, zero, fine_sort                         #la lista è vuota

    add t3, s4, zero
    lw t1, 1(t3)

    beq t1, s4, fine_sort                           #la lista contiene un solo elemento che quindi è ordinato

sorting_loop:
    lb t0, 0(t1)
    lb t2, 0(t3)
    li a2, 65                                       #carattere 'A' in codice ASCII
    li a3, 90                                       #carattere 'Z' in codice ASCII
    blt t2, a2, primo_minuscola
    bgt t2, a3, primo_minuscola
    li a4, 0                                        #se arrivo qui vuol dire che è un carattere maiuscolo --> indicato con 0
    j secondo_maiuscola

primo_minuscola:
    li a2, 97                                       #carattere 'a' in codice ASCII
    li a3, 122                                      #carattere 'z' in codice ASCII
    blt t2, a2, primo_numero
    bgt t2, a3, primo_numero
    li a4, 1                                        #se arrivo fin qui vuol dire che è un carattere minuscolo --> indicato con 1
    j secondo_maiuscola

primo_numero:
    li a2, 48                                       #carattere '0' in codice ASCII
    li a3, 57                                       #carattere '9' in codice ASCII
    blt t2, a2, primo_extra
    bgt t2, a3, primo_extra
    li a4, 2                                        #se arrivo fin qui vuol dire che è un carattere numerico --> indicato con 2
    j secondo_maiuscola

primo_extra:
    li a4, 3                                        #se arrivo fin qui vuol dire che è un carattere speciale --> indicato con 3

secondo_maiuscola:
    li a2, 65
    li a3, 90
    blt t0, a2, secondo_minuscola
    bgt t0, a3, secondo_minuscola
    li a5, 0                                        #se arrivo fin qui vuol dire che è un carattere maiuscolo --> indicato con 0
    j compara

secondo_minuscola:
    li a2, 97                                       #carattere 'a' in codice ASCII
    li a3, 122                                      #carattere 'z' in codice ASCII
    blt t0, a2, secondo_numero
    bgt t0, a3, secondo_numero
    li a5, 1                                        #se arrivo fin qui vuol dire che è un carattere minuscolo --> indicato con 1
    j compara

secondo_numero:
    li a2, 48                                       #carattere '0' in codice ASCII
    li a3, 57                                       #carattere '9' in codice ASCII
    blt t0, a2, secondo_extra
    bgt t0, a3, secondo_extra
    li a5, 2                                        #se arrivo fin qui vuol dire che è un carattere numerico --> indicato con 2
    j compara

secondo_extra:
    li a5, 3                                        #se arrivo fin qui vuol dire che è un carattere speciale --> indicato con 3

compara:
    bgt a4, a5, increment_loop                      #i due elementi appartengono a due categorie diverse
    beq a4, a5, compara_ASCII                       #i due elementi appartengono alla stessa categoria
    sb t2, 0(t1)
    sb t0, 0(t3)
    j increment_loop

compara_ASCII:
    blt t2, t0, increment_loop
    sb t2, 0(t1)
    sb t0, 0(t3)

increment_loop:
    addi t3, t1, 0
    lw t1, 1(t1)
    bne t1, a6, sorting_loop
    addi a6, t3, 0
    add t3, s4, zero
    lw t1, 1(t3)
    bne a6, s8, sorting_loop

fine_sort:
    jr ra
```

OPERAZIONI SDX & SSX

La sesta e settima implementazione sono lo shift a destra e lo shift a sinistra degli elementi della lista. Per implementare questi due algoritmi è stato deciso di copiare gli elementi della lista a partire da una locazione di memoria il cui indirizzo è contenuto nel registro **s9** ottenendo l'array di caratteri sul quale si andrà a lavorare. Se il numero di caratteri nella lista è zero oppure uno allora non si fa niente. Successivamente si salverà un altro indirizzo di memoria arbitrario nel registro **s10,** a partire dal quale, dopo alcuni passaggi avrò il mio array di caratteri shiftato di una posizione a destra oppure a sinistra a seconda dell'algoritmo. Infine con l'aiuto di un paio di funzioni si copiano gli elementi dalla stringa “shiftata” nella catena.
Il metodo con cui si shiftano le stringhe è leggermente diverso nei due metodi ma **concettualmente uguale:** per lo shift a destra, dopo aver copiato gli elementi dalla catena nella prima stringa ci salviamo il byte nell'ultima posizione dell'array di caratteri (sostituendolo con il carattere flag) e, nel passaggio in cui si copiano gli elementi nella stringa “shiftata”, mettiamo come primo elemento il carattere che ci siamo appena salvati ed in seguito tutti gli altri.
Per lo shift a sinistra agiamo nello stesso modo, solo che il carattere da salvare in un registro sarà il primo il quale andrà posizionato nella prima posizione (usiamo sempre il carattere flag per capire quando fermarci nello scorrimento dell'array di caratteri). I registri usati, oltre ad s9 ed s10 sono tutti registri temporanei che vanno da **t0** a **t6.**

```
660 ######################################
661 ############### SDX ##################
662 ######################################
663
664 SDX:
665     beq s6 zero fine_SDX
666     li t0 1
667     beq s6 t0 fine_SDX
668     li s9 0x00600000
669     addi t0 s9 0
670     addi t6 s4 0
671
672 da_catena_a_stringaSDX:
673     lb t3 0(t6)
674     bne t3 s7 metti_in_stringaSDX
675     addi t6 t6 5
676     j da_catena_a_stringaSDX
677
678 metti_in_stringaSDX:
679     sb t3 0(t0)
680     lw t4 1(t6)
681     beq t4 s4 salva_ultimo_SDX
682     addi t0 t0 1
683     addi t6 t6 5
684     j da_catena_a_stringaSDX
685
686 salva_ultimo_SDX:
687     lb t1 0(t0)                         #salvo l'ultimo char
688     sb s7 0(t0)                         #metto il flag al posto dell'ultimo elemento
689     li s10 0x00625000                   #stringa shiftata
690     addi t0 s10 0
691     sb t1 0(t0)                         #setto l'ultimo char come primo nella stringa shiftata
692     addi t1 s9 0
693     addi t0 t0 1
694     j metti_il_resto_SDX
695
```

```
metti_il_resto_SDX:
    lb t2 0(t1)
    beq t2 s7 metti_flag_SDX
    sb t2 0(t0)
    addi t0 t0 1
    addi t1 t1 1
    j metti_il_resto_SDX

metti_flag_SDX:
    sb s7 0(t0)
    j put_back_in_chain_SDX

put_back_in_chain_SDX:
    addi t0 s10 0
    addi t6 s4 0
    j stringa_to_catena_SDX

stringa_to_catena_SDX:
    lb t1 0(t0)                         #carica in un registro i valori della stringa
    beq t1 s7 fine_SDX
    lb t2 0(t6)                         #controlla se è il nodo cancellato logicamente
    beq t2 s7 prossimo_SDX
    sb t1 0(t6)
    addi t0 t0 1
    addi t6 t6 5
    j stringa_to_catena_SDX

prossimo_SDX:
    addi t6 t6 5
    j stringa_to_catena_SDX

fine_SDX:
    jr ra
```

```
#######################################
################ SSX ##################
#######################################

SSX:
    beq s6 zero fine_SSX
    li t0 1
    beq s6 t0 fine_SSX
    li s9 0x00600000
    addi t0 s9 0
    addi t6 s4 0
    
da_catena_a_stringa_SSX:
    lb t3 0(t6)
    bne t3 s7 metti_in_stringa_SSX
    addi t6 t6 5
    j da_catena_a_stringa_SSX

metti_in_stringa_SSX:
    sb t3 0(t0)
    lw t4 1(t6)
    beq t4 s4 salva_primo_SSX
    addi t0 t0 1
    addi t6 t6 5
    j da_catena_a_stringa_SSX

salva_primo_SSX:
    addi t0 t0 1
    sb s7 0(t0)                         #metto il flag alla fine di s9
    addi t0 s9 0
    lb t1 0(t0)                         #salvo il primo carattere che poi sarà l'ultimo
    li s10 0x00625000
    addi t2 s10 0
    addi t0 t0 1
    j metti_il_resto_SSX
```

```
767 metti_il_resto_SSX:
768     lb t3 0(t0)
769     beq t3 s7 metti_flag_e_ultimo_SSX
770     sb t3 0(t2)
771     addi t2 t2 1
772     addi t0 t0 1
773     j metti_il_resto_SSX
774
775 metti_flag_e_ultimo_SSX:
776     sb t1 0(t2)
777     addi t2 t2 1
778     sb s7 0(t2)
779     j put_back_in_chain_SSX
780
781 put_back_in_chain_SSX:
782     addi t0 s10 0
783     addi t6 s4 0
784     j stringa_to_catena_SSX
785
786 stringa_to_catena_SSX:
787     lb t1 0(t0)                         #carica in un registro i valori della stringa
788     beq t1 s7 fine_SSX
789     lb t2 0(t6)                         #controlla se è un nodo cancellato logicamente
790     beq t2 s7 prossimo_SSX
791     sb t1 0(t6)
792     addi t0 t0 1
793     addi t6 t6 5
794     j stringa_to_catena_SSX
795
796 prossimo_SSX:
797     addi t6 t6 5
798     j stringa_to_catena_SSX
799
800 fine_SSX:
801     jr ra
802
```

END MAIN

Il programma termina con la system call che fa uscire il programma con codice 0.

```
805 ########################################
806 ########################################
807 ########################################
808
809 end_main:
810     li a7 10
811     ecall
812
```

ULTERIORI TEST

"ADD(])~DEL(]) ~PRINT ~ ADD(g) ~REV ~ ADD($) ~ADD(i) ~ ADD(()~ ADD(6) ~ SDX ~ PRINT ~DEL(g) ~ ADD(N)~ADD(D) ~ SSX ~ PRINT ~ADD(>) ~ ADD(;) ~ ADD(!) ~REV PRINT ~ ADD(M) ~ PRINT~DEL(M) ~ ADD(L)~SORT ~PRINT ~SDX ~ PRINT ~ DEL(6) ~ PRINT~PRINT~PRINT ~ PRINT"

ADD(z) ~ADD(:) ~ADD(B) ~ADD(z) ~ ADD(;) ~ ADD(c) ~ ADD(0) ~ADD(a) ~ADD(b) ~ PRINT ~ DEL(z) ~ PRINT ~ REV ~ PRINT ~ SDX ~ PRINT ~ SSX ~ PRINT ~ SORT ~ PRINT