RELAZIONE PROGETTO - ARCHITETTURE DEGLI ELABORATORI A.S 2022/23

-Autore: Alessio Majid
-Matricola: 7073646
-Email: alessio.majid@stud.unifi.it

Questo progetto riguarda l'implementazione di un codice RISC-V per gestire operazioni su una lista concatenata circolare. La lista contiene elementi, ognuno dei quali ha una dimensione di 5 byte. Ogni elemento è composto da:

- DATA (Byte 0): Questo byte contiene l'informazione principale, che rappresenta un carattere in codice ASCII. Tuttavia, solo i caratteri con codici ASCII compresi tra 32 e 125 sono accettabili come dati all'interno della lista.
- PAHEAD (Byte 1-4): Questi quattro byte costituiscono un puntatore all'elemento successivo nella lista. Se l'elemento è l'unico nella lista, il puntatore può anche riferirsi a se stesso.

Le sette operazioni fondamentali che il codice RISC-V deve gestire sulla lista concatenata circolare sono:

1. ADD: Inserimento di un nuovo elemento nella lista.
2. DEL: Rimozione di un elemento dalla lista.
3. PRINT: Stampa di tutti gli elementi della lista.
4. SORT: Ordinamento degli elementi nella lista.
5. SDX: Shift a destra (rotazione in senso orario) degli elementi della lista.
6. SSX: Shift a sinistra (rotazione in senso antiorario) degli elementi della lista.
7. REV: Inversione dell'ordine degli elementi della lista.

È importante assicurarsi che il programma gestisca correttamente le operazioni richieste sulla lista e rispetti le specifiche fornite, come il range di caratteri ASCII accettabili. (32-125 compresi)

MAIN PROCEDURE

Il programma è strutturato attorno a una procedura principale che elabora l'unico input che può essere inserito dall'utente. Questo input è rappresentato da una variabile di tipo stringa denominata "listInput", dichiarata nel campo ".data" del codice. Lo scopo di questa stringa è quello di contenere i comandi da comunicare al programma per la gestione della lista concatenata circolare.

I comandi da passare al programma devono essere separati tra loro utilizzando il carattere tilde '~' (ASCII 126). Inoltre, è importante notare che la stringa di input non può contenere più di 30 comandi.

Di seguito viene presentato un estratto significativo del codice, accompagnato da una spiegazione dettagliata:

```
#Alessio Majid
#7073646

#prova di comandi extra
#ADD(1) ~ ADD(L) ~ ADD(a) ~ ADD(H) ~ ADD(:) ~ ADD(5) ~SDX~SORT~PRINT~DEL(a) ~DEL(J) ~PRI~SSX~REV~PRINT
#ADD(1) ~ SDX ~ ADD(F) ~ add(v) ~ ADD(v) ~ ADD ~ ADD(1) ~PRINT~SORT(a)~PRINT~DEL(vv) ~DEL(v) ~PRINT~REV~SSX~PRINT

.data
listInput: .string "ADD(A) ~ADD(:) ~ADD(h) ~ADD(z) ~ ADD(;) ~ ADD(R) ~ ADD(4) ~ADD(a) ~ADD(b) ~ PRINT ~ DEL(4) ~ PRINT ~SORT ~ PRINT ~ REV ~ PRINT ~ SDX ~ PRINT ~ SSX ~ PRINT ~ SORT ~ PRINT"
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 0x00400000                    #indirizzo di memoria del primo elemento - PAHEAD
li s5 0x00400000                    #contatore di ciclo posizionale degli elementi
li s6 0                             #numero di elementi nella lista concatenata
li s7 9                             #flag

check_input:
    add t1 s0 s1
    lb t2 0(t1)                      #carico il carattere corrente per le verifiche
    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 Space in un registro
    beq t2 t3 increment_counter      #se il char a cui punto ? uno spazio allora passo al char sucessivo
    li t3 65                         #carico A in un registro
    beq t2 t3 check_add
    li t3 68                         #carico D in un registro
    beq t2 t3 check_del
    li t3 80                         #carico P in un registro
    beq t2 t3 check_print
    li t3 83                         #carico S in un registro
    beq t2 t3 check_s
    li t3 82                         #carico R in un registro
    beq t2 t3 check_rev
    j skip_command
```

La procedura principale è focalizzata sulla lettura di ciascun byte individualmente dalla stringa in ingresso fino a quando non si incontra il valore 0 (che corrisponde al carattere ASCII di fine stringa). Utilizzando un contatore conservato nel registro s1, si riesce a individuare il carattere corrente all'interno della stringa e a memorizzarne il contenuto relativo nel registro temporaneo t2. Questo contatore viene incrementato nelle sezioni del codice dedicate alla verifica dell'accuratezza dei vari comandi.

Dopo aver verificato che il carattere non sia 0 (fine stringa) e che non si sia superato il numero massimo consentito di comandi, è necessario effettuare un controllo ulteriore per determinare se il carattere in questione sia uno spazio (ASCII 32). In caso affermativo, l'esecuzione deve avanzare nella stringa finché non si trova un carattere diverso dallo spazio ma che non sia il tilde '~'. Questa operazione è gestita da una funzione di supporto, la quale incrementa il contatore e passa allo switch iniziale, permettendo così la lettura del carattere successivo nella stringa.

Dopo aver effettuato tali controlli, la procedura procede con l'implementazione di una struttura di controllo basata su uno switch. Questa struttura consente l'esecuzione degli algoritmi appropriati, confrontando il carattere corrente con i caratteri ASCII accettabili per i comandi. I caratteri accettati sono: A (65), D (68), P (80), S (83), R (82).

Se il carattere corrente non corrisponde a nessuno di questi comandi di gestione della lista, ciò implica che il comando fornito nella stringa non è valido. Di conseguenza, si procede al comando successivo, adottando la seguente funzione di supporto per effettuare questa transizione:

```
294 skip_command:
295     beq t2 zero end_main
296     li t5 126                           #carico tilde in un registro
297     beq t2 t5 increment_counter         #se il carattere a cui punto e' tilde
298     addi s1 s1 1                        #contatore++
299     add t1 s0 s1
300     lb t2 0(t1)                         #carico il carattere corrente per le verifiche
301     j skip_command
```

La funzione di supporto, in ogni ciclo, incrementa il contatore, permettendo così di avanzare nella stringa fino a quando non si trova un carattere tilde '~'.
Il controllo del carattere consente quindi l'esecuzione dell'algoritmo corretto per verificare la correttezza del comando fornito in input. Questo algoritmo è implementato in una procedura

separata, diversa per ogni comando. Ciascun algoritmo è suddiviso in tre procedure distinte. Queste procedure sono utilizzate per verificare che i comandi siano corretti, completi ed unici (cioè non ci siano due comandi separati da tilde), e includono le seguenti funzioni:

1. Una funzione che attraversa il comando fino alla sua fine.
2. Una funzione che garantisce che il comando sia l'unico fino al tilde successivo.
3. Una funzione che passa all'implementazione specifica di quel comando.

In seguito, sono elencati gli algoritmi per ciascun tipo di comando:

```
check_add:
    beq s2 s3 skip_command                  #se sono al 31esimo comando non lo eseguo
    addi s1 s1 1                            #aumento il contatore
    add t1 s0 s1
    lb t2 0(t1)                             #carico il carattere corrente per le verifiche
    li t3 68                                #carico la lettera D in un registro
    bne t2 t3 skip_command
    addi s1 s1 1                            #aumento il contatore
    add t1 s0 s1
    lb t2 0(t1)                             #carico il carattere corrente per le verifiche
    bne t2 t3 skip_command
    addi s1 s1 1                            #aumento il contatore
    add t1 s0 s1
    lb t2 0(t1)                             #carico il carattere corrente per le verifiche
    li t3 40                                #metto la tonda aperta in un registro
    bne t2 t3 skip_command
    addi s1 s1 1                            #aumento il contatore
    add t1 s0 s1
    lb t2 0(t1)                             #carico il carattere corrente per le verifiche
    li t4 32                                #carico ASCII 32 (Space) in un registro
    li t5 125                               #carico ASCII 125 (}) in un registro
    blt t2 t4 skip_command                  #se il char e' minore di 32 salto
    bgt t2 t5 skip_command                  #se il char e' maggiore di } salto
    addi a0 t2 0                            #salvo in a0 il carattere
    addi s1 s1 1                            #contatore++
    add t1 s0 s1
    lb t2 0(t1)                             #carico il carattere corrente per le verifiche
    li t3 41                                #controllo la validita' del comando
    bne t2 t3 skip_command

unique_ADD:
    addi s1 s1 1                            #contatore++
    add t1 s0 s1
    lb t2 0(t1)                              #carico il carattere corrente per le verifiche
    li t5 126                                #carico ASCII 126 (tilde) in un registro
    li t4 32                                 #carico ASCII 32 (Space) in un registro
    beq t2 t5 prepare_execute_ADD            #se trova un char tilde allora il comando e' corretto
    beq t2 zero ADD                          #sono arrivato qui, il comando e' corretto ma sono a fine stringa
    bne t2 t4 skip_command                   #se salto qui il comando non e' corretto
    j unique_ADD

prepare_execute_ADD:
    jal ADD
    li a0 0                                 #resetto l'argomento da passare alle funzioni
    addi s2 s2 1                            #numero di comandi++
    j increment_counter
```

Questa funzione, insieme alle altre, si assicura di rimanere all'interno del range del numero consentito di caratteri (salvato nel registro s3), comparandolo con il numero di comandi corrente (salvato nel registro s2). Successivamente, tramite istruzioni di salto condizionato (bne), inizia a confrontare il carattere corrente (salvato nel registro temporaneo t2) con il carattere previsto secondo la definizione del comando stesso. Ad esempio, dopo la 'A', ci si aspetta che se il comando è corretto, il carattere immediatamente successivo sia una 'D', e così via per gli altri comandi.

Se si rileva che uno dei caratteri non corrisponde a quello previsto oppure ci sono spazi tra le lettere del comando, la funzione chiama la procedura "skip_command". Questa procedura, come precedentemente descritto, utilizza la funzione "increment_counter" per portare all'istante successivo al carattere tilde. I comandi ADD e DEL, a differenza degli altri, richiedono anche il passaggio di un parametro (codice ASCII compreso tra 32 e 125) racchiuso tra parentesi tonde. La validità di questo parametro viene controllata in questa stessa procedura. Se il carattere è valido, viene caricato nel registro a0, pronto per essere utilizzato quando verrà chiamata la relativa funzione. Se il carattere non è valido, il comando viene considerato non valido e la procedura richiama "skip_command".

Successivamente, è necessario garantire che il comando sia unico, ovvero che segua solo spazi e infine un tilde o 0. Questo controllo viene eseguito tramite la funzione "unique_ADD", che continua a scorrere la stringa. Appena trova un tilde, chiama la relativa funzione. Se trova 0, significa che il comando è corretto ed è l'ultimo comando poiché la stringa è finita. Se trova un carattere diverso da spazio e tilde, ciò indica che il comando non è valido e si passa al prossimo utilizzando le funzioni di supporto.

L'ultima funzione, chiamata "prepare_execute_DEL", esegue un salto con collegamento (jump and link) al comando corrispondente. Una volta terminata l'esecuzione di tale comando, resetta il registro a0 a zero e incrementa il conteggio dei comandi eseguiti.

Di seguito è riportato anche l'algoritmo per il controllo della correttezza del comando DEL, la cui implementazione è analoga a quella del comando ADD:

```
88 check_del:
89     beq s2 s3 skip_command                #se sono al 31esimo comando non lo eseguo
90     addi s1 s1 1                          #contatore++
91     add t1 s0 s1
92     lb t2 0(t1)                           #carico il carattere corrente per le verifiche
93     li t3 69                              #carico la lettera E in un registro
94     bne t2 t3 skip_command                #se la lettera non coincide il comando e' errato
95     addi s1 s1 1                          #contatore++
96     add t1 s0 s1
97     lb t2 0(t1)                           #carico il carattere corrente per le verifiche
98     li t3 76                              #carico la lettera L in un registro
99     bne t2 t3 skip_command                #se la lettera non coincide il comando e' errato
00     addi s1 s1 1                          #contatore++
01     add t1 s0 s1
02     lb t2 0(t1)                           #carico il carattere corrente per le verifiche
03     li t3 40                              #metto la tonda aperta in un registro
04     bne t2 t3 skip_command                #se la lettera non coincide il comando e' errato
05     addi s1 s1 1                          #contatore++
06     add t1 s0 s1
07     lb t2 0(t1)                           #carico il carattere corrente per le verifiche
08     li t4 32                              #carico ASCII 32 (Space) in un registro
09     li t5 125                             #carico ASCII } in un registro
10     blt t2 t4 skip_command                #se il char e' minore di 32 salto
11     bgt t2 t5 skip_command                #se il char e' maggiore di } salto
12     addi a0 t2 0                          #salvo in a0 l'argomento della funzione
13     addi s1 s1 1                          #contatore++
14     add t1 s0 s1
15     lb t2 0(t1)                           #carico il carattere corrente per le verifiche
16     li t3 41                              #metto la tonda chiusa in un registro
17     bne t2 t3 skip_command                #se la lettera non coincide il comando e' errato
18
19 unique_DEL:
20     addi s1 s1 1                          #contatore++
21     add t1 s0 s1
22     lb t2 0(t1)                           #carico il carattere corrente per le verifiche
23     li t5 126                             #carico ASCII 126 (tilde) in un registro
24     li t4 32                              #carico ASCII 32 (Space) in un registro
25     beq t2 t5 prepare_execute_DEL         #se trova un char tilde allora il comando e' unico
26     beq t2 zero DEL
27     bne t2 t4 skip_command                #se trova un char che non e' Space salta
28     j unique_DEL                          #se arriva qui e' un comando unico
29
30 prepare_execute_DEL:
31     jal DEL
32     li a0 0                               #resetto l'argomento da passare alle funzioni
33     addi s2 s2 1                          #numero di comandi++
34     j increment_counter
35
```

L'implementazione dell'algoritmo per verificare la correttezza del comando PRINT è identica a quella dei comandi ADD e DEL. La differenza principale risiede nel fatto che, naturalmente, il comando PRINT non richiede il passaggio di alcun parametro. Si utilizzano sempre gli

stessi registri per il percorso attraverso la stringa, la memorizzazione del carattere corrente e il salvataggio dei numeri necessari per i confronti. Di seguito è riportato il codice corrispondente:

```
check_print:
    beq s2 s3 skip_command              #se sono al 31esimo comando non lo eseguo
    addi s1 s1 1                        #contatore++
    add t1 s0 s1
    lb t2 0(t1)                         #carico il carattere corrente per le verifiche
    li t3 82                            #metto il char R in un registro
    bne t2 t3 skip_command              #se la lettera non coincide il comando e' errato
    addi s1 s1 1                        #contatore++
    add t1 s0 s1
    lb t2 0(t1)                         #carico il carattere corrente per le verifiche
    li t3 73                            #metto il char I in un registro
    bne t2 t3 skip_command              #se la lettera non coincide il comando e' errato
    addi s1 s1 1                        #contatore++
    add t1 s0 s1
    lb t2 0(t1)                         #carico il carattere corrente per le verifiche
    li t3 78                            #metto il char N in un registro
    bne t2 t3 skip_command              #se la lettera non coincide il comando e' errato
    addi s1 s1 1                        #contatore++
    add t1 s0 s1
    lb t2 0(t1)                         #carico il carattere corrente per le verifiche
    li t3 84                            #metto il char T in un registro
    bne t2 t3 skip_command              #se la lettera non coincide il comando e' errato

unique_PRINT:
    addi s1 s1 1                        #contatore++
    add t1 s0 s1
    lb t2 0(t1)                         #carico il carattere corrente per le verifiche
    li t5 126                           #carico ASCII 126 (tilde) in un registro
    li t4 32                            #carico ASCII 32 (Space) in un registro
    beq t2 t5 prepare_execute_PRINT     #se trova un char tilde allora il comando e' unico
    beq t2 zero PRINT
    bne t2 t4 skip_command              #se trova un char che non e' Space salta
    j unique_PRINT                      #se arriva qui e' un comando unico

prepare_execute_PRINT:
    jal PRINT
    li a0 0                             #resetto l'argomento da passare alle funzioni
    addi s2 s2 1                        #numero di comandi++
    j increment_counter
```

L'implementazione dell'algoritmo per verificare la correttezza del comando REV è analoga a quella dell'algoritmo PRINT. Di seguito una porzione significativa:

```
176 check_rev:
177     beq s2 s3 skip_command                  #se sono al 31esimo comando non lo eseguo
178     addi s1 s1 1                            #contatore++
179     add t1 s0 s1
180     lb t2 0(t1)                             #carico il carattere corrente per le verifiche
181     li t3 69
182     bne t2 t3 skip_command
183     addi s1 s1 1                            #contatore++
184     add t1 s0 s1
185     lb t2 0(t1)                             #carico il carattere corrente per le verifiche
186     li t3 86
187     bne t2 t3 skip_command
188
189 unique_REV:
190     addi s1 s1 1                            #contatore++
191     add t1 s0 s1
192     lb t2 0(t1)                             #carico il carattere corrente per le verifiche
193     li t5 126
194     li t4 32
195     beq t2 t5 prepare_execute_REV           #se trova un char tilde allora il comando e' unico
196     beq t2 zero REV
197     bne t2 t4 skip_command                  #se trova un char che non e' Space salta    j unique_REV
198
199 prepare_execute_REV:
200     jal REV
201     li a0 0                                 #resetto l'argomento da passare alle funzioni
202     addi s2 s2 1                            #numero di comandi++
203     j increment_counter
```

Per quanto concerne i tre comandi la cui lettera iniziale è 'S', è stato scelto di incorporare un ulteriore blocco di istruzioni "switch". Questo blocco scorre la stringa di input di una posizione, prelevando così la seconda lettera del comando e orientando il flusso del programma di conseguenza. Dopo di ciò, l'implementazione degli algoritmi che verificano la correttezza dei comandi SORT, SDX ed SSX segue il medesimo approccio adottato per i comandi PRINT e REV. In seguito, il codice corrispondente viene riportato:

```
205 check_s:
206     addi s1 s1 1                        #contatore++
207     add t1 s0 s1
208     lb t2 0(t1)                         #carico il carattere corrente per le verifiche
209     li t3 79                            #metto il char O in un registro
210     li t4 68                            #metto il char D in un registro
211     li t5 83                            #metto il char S in un registro
212     beq t2 t3 verify_SORT
213     beq t2 t4 verify_SDX
214     beq t2 t5 verify_SSX
215     j skip_command
```

```
verify_SDX:
    beq s2 s3 skip_command              #se sono al 31esimo comando non lo eseguo
    addi s1 s1 1                        #contatore++
    add t1 s0 s1
    lb t2 0(t1)                         #carico il carattere corrente per le verifiche
    li t3 88
    bne t2 t3 skip_command

unique_SDX:
    addi s1 s1 1                        #contatore++
    add t1 s0 s1
    lb t2 0(t1)                         #carico il carattere corrente per le verifiche
    li t5 126
    li t4 32
    beq t2 t5 prepare_execute_SDX       #se trova un char tilde allora il comando e' unico
    beq t2 zero SDX
    bne t2 t4 skip_command              #se trova un char che non e' Space salta

prepare_execute_SDX:
    jal SDX
    li a0 0
    addi s2 s2 1                        #numero di comandi++
    j increment_counter
```

```
verify_SORT:
    beq s2 s3 skip_command              #se sono al 31esimo comando non lo eseguo
    addi s1 s1 1                        #contatore++
    add t1 s0 s1
    lb t2 0(t1)                         #carico il carattere corrente per le verifiche
    li t3 82
    bne t2 t3 skip_command
    addi s1 s1 1                        #contatore++
    add t1 s0 s1
    lb t2 0(t1)                         #carico il carattere corrente per le verifiche
    li t3 84
    bne t2 t3 skip_command

unique_SORT:
    addi s1 s1 1                        #contatore++
    add t1 s0 s1
    lb t2 0(t1)                         #carico il carattere corrente per le verifiche
    li t5 126
    li t4 32
    beq t2 t5 prepare_execute_SORT      #se trova un char tilde allora il comando e' unico
    beq t2 zero SORT
    bne t2 t4 skip_command              #se trova un char che non e' Space salta

prepare_execute_SORT:
    jal SORT
    li a0 0                             #resetto l'argomento da passare alle funzioni
    addi s2 s2 1                        #numero di comandi++
    j increment_counter
```

```
verify_SSX:
    beq s2 s3 skip_command              #se sono al 31esimo comando non lo eseguo
    addi s1 s1 1                        #contatore++
    add t1 s0 s1
    lb t2 0(t1)                         #carico il carattere corrente per le verifiche
    li t3 88
    bne t2 t3 skip_command

unique_SSX:
    addi s1 s1 1                        #contatore++
    add t1 s0 s1
    lb t2 0(t1)                         #carico il carattere corrente per le verifiche
    li t5 126
    li t4 32
    beq t2 t5 prepare_execute_SSX       #se trova un char tilde allora il comando e' unico
    beq t2 zero SSX
    bne t2 t4 skip_command              #se trova un char che non e' Space salta

prepare_execute_SSX:
    jal SSX
    li a0 0
    addi s2 s2 1                        #numero di comandi++
    j increment_counter
```

ADD

La procedura di add consiste nell'inserimento in coda di un carattere (ammesso nel range dato) all'interno della struttura dati creata. La procedura inizia con una prima verifica del numero degli elementi: se il numero è zero, indica l'inserimento del primo elemento. Tramite l'istruzione sb il carattere precedentemente validato dall'algoritmo di verifica della correttezza e salvato in a0, viene inserito nella posizione di memoria indicata da PAHEAD. Questo indirizzo è l'indirizzo di partenza della catena, conservato nel registro s4.

Inoltre, la procedura fa uso di un contatore di ciclo posizionale degli elementi, memorizzato nel registro s5. Questo contatore è utile per tener traccia del punto in cui inserire il prossimo elemento. Successivamente, l'indirizzo di memoria del primo elemento è impostato come puntatore a se stesso, garantendo così la circolarità della catena. Contestualmente, il numero di elementi nella catena viene incrementato di 1, aumentando il valore nel registro s6.

Nel caso in cui non si stia inserendo il primo elemento, dopo aver inserito l'elemento in memoria e averlo collegato al primo elemento tramite un'istruzione sw, è necessario collegare il nuovo elemento all'elemento precedente. Pertanto, la lista viene esaminata per verificare che non si stia puntando a un elemento logicamente cancellato. Il primo elemento non cancellato che viene trovato viene collegato all'elemento appena inserito mediante

un'istruzione sw, sovrascrivendo il puntatore dell'elemento che puntava al primo. Ciò fa sì che l'indirizzo di memoria ora punti all'elemento appena inserito.

Una volta che i puntatori sono stati collegati correttamente, la procedura ritorna alla funzione "prepare_execute_ADD", che a sua volta ritorna alla porzione di codice che sta iterando attraverso la stringa in input. Di seguito una porzione significativa del codice:

```
308 ADD:
309     beq s6 zero first_add                   #controllo che sia il primo elemento della catena
310     sb a0 0(s5)                             #Salvo il byte passato come parametro
311     sw s4 1(s5)                             #salvo il puntatore
312     addi s6 s6 1
313     addi t6 s5 0
314     addi t6 t6 -5
315     j search_previous
316
317 first_add:
318     sb a0 0(s4)                              #Salvo il byte passato come parametro
319     sw s4 1(s5)                              #salvo il puntatore
320     addi s5 s5 5                             #punto alla cella in cui andra' il prossimo elemento
321     addi s6 s6 1                             #numero elementi ++
322     jr ra
323
324 search_previous:
325     lb t3 0(t6)
326     bne t3 s7 link_following
327     addi t6 t6 -5
328     j search_previous
329
330 link_following:
331     sw s5 1(t6)
332     addi s5 s5 5
333     jr ra
334
```

DELETE

La procedura di delete consiste nella cancellazione logica di un carattere o di tutte le occorrenze di un determinato carattere (ammesso nel range dato) all'interno della struttura dati creata. L'algoritmo attraversa la lista, inserendo il primo elemento in un registro temporaneo. Questo elemento viene confrontato con l'elemento da eliminare. Se l'elemento corrente coincide con quello da eliminare, la procedura richiama la funzione delete_first. All'interno di questa funzione, inizialmente viene verificato il numero degli elementi. Se il numero di elementi al momento dell'eliminazione è maggiore di uno, la funzione delete_nth viene chiamata per la cancellazione logica. Qui si esegue un'operazione sb utilizzando un carattere arbitrario come flag (ASCII 9). Successivamente, l'indirizzo di inizio della lista (PAHEAD) viene aggiornato, il numero di elementi diminuisce e si passa all'elemento successivo richiamando nuovamente la funzione delete_first. Questo copre il caso in cui non solo il primo elemento debba essere eliminato, ma anche successivi elementi.

Nel caso in cui l'elemento da eliminare sia solo il primo, la lista viene attraversata fino a trovare l'ultimo elemento. Quest'ultimo viene collegato alla nuova testa della lista tramite la funzione link_to_head. Questo collegamento coinvolge l'aggiornamento del puntatore dell'ultimo elemento con l'indirizzo di memoria del nuovo primo elemento.

Se l'unico elemento presente nella lista deve essere eliminato viene adottato l'approccio di creare una nuova lista a partire dall'indirizzo del contatore di ciclo posizionale (s5), che si trova sempre in una locazione di memoria libera.

L'algoritmo gestisce il caso in cui ci siano più elementi da eliminare, tra cui almeno uno è l'elemento in testa alla catena. Dopo la cancellazione del primo elemento, l'algoritmo viene richiamato nuovamente dall'inizio per eliminare gli altri elementi. Questo è realizzato tramite il metodo find, che scansiona la catena, saltando gli elementi logicamente cancellati, fino a trovare l'elemento da eliminare o fino a quando il puntatore dell'elemento corrente punta nuovamente al primo elemento, indicando il termine della lista. Una volta individuato l'elemento, la cancellazione logica avviene tramite il metodo delete_nth, tenendo conto se l'elemento è l'ultimo. Se l'elemento è effettivamente l'ultimo, la funzione change_PAHEAD aggiorna il puntatore dell'elemento precedente all'ultimo con l'indirizzo del primo elemento.

Altrimenti, se l'elemento da cancellare non è l'ultimo, la funzione find_following viene richiamata per individuare il primo elemento non cancellato logicamente e collegarlo tramite la funzione link. Questo collegamento avvia una nuova iterazione attraverso la funzione find per cercare ulteriori elementi da eliminare. Il raggiungimento della fine della lista è determinato controllando se il puntatore all'elemento successivo coincide con l'indirizzo del primo elemento, segnalando che l'elemento corrente è l'ultimo. Di seguito una porzione significativa del codice:

```
DEL:
    addi t0 s4 0
    addi t6 t0 5
    lb t3 0(t0)
    beq t3 a0 delete_first                  #funzione che rimuove SOLO il primo elemento

find:
    lb t3 0(t6)
    beq t3 a0 delete_found
    lw t4 1(t0)
    beq t4 s4 end_DEL
    beq t3 s7 skip_flag
    lw t4 1(t6)
    beq t4 s4 end_DEL
    addi t0 t6 0
    addi t6 t6 5
    j find

delete_found:
    sb s7 0(t6)
    lw t4 1(t6)
    beq t4 s4 change_pahead
    addi t6 t6 5
    addi s6 s6 -1
    j search_following

change_pahead:
    sw s4 1(t0)
    jr ra

search_following:
    lb t3 0(t6)
    beq t3 a0 delete_found
    bne t3 s7 link
    addi t6 t6 5
    j search_following

link:
    sw t6 1(t0)
    lw t4 1(t6)
    beq t4 s4 end_DEL
    addi t6 t6 5
    j find
skip_flag:
    addi t6 t6 5
    j search_following

delete_first:
    lb t4 1
    beq s6 t4 reset
    lb t3 0(t0)
    beq t3 a0 delete_nth
    bne t3 s7 update_second
    addi t0 t0 5
    j delete_first
```

```
392 delete_nth:
393     sb s7 0(t0)
394     addi s6 s6 -1
395     addi t0 t0 5
396     j delete_first
397 
398 update_second:
399     addi t6 t0 0
400     j find_last
401 
402 find_last:
403     lb t3 0(t6)
404     bne t3 s7 check_pointer
405     addi t6 t6 5
406     j find_last
407 
408 check_pointer:
409     lw t4 1(t6)
410     beq t4 s4 link_to_head
411     addi t6 t6 5
412     j find_last
413 
414 link_to_head:
415     addi s4 t0 0
416     sw s4 1(t6)
417     j DEL
418 
419 reset:
420     sb s7 0(t0)
421     addi s4 s5 0
422     jr ra
423 
424 end_DEL:
425     jr ra
```

PRINT

Viene implementata la terza operazione, denominata PRINT, che ha il compito di stampare tutti i dati degli elementi presenti nella lista, rispettando l'ordine di apparizione. L'algoritmo verifica preliminarmente la presenza di elementi nella lista; in caso contrario, termina senza compiere alcuna azione. In caso contrario, scorre l'intera lista con l'attenzione di evitare gli elementi cancellati logicamente. Per ciascun elemento, il carattere da stampare viene inserito nel registro a0 e viene effettuata una chiamata al sistema per la stampa dei caratteri.

Inoltre, in ogni ciclo, viene verificato se l'elemento corrente è l'ultimo, ciò è verificato attraverso il controllo del puntatore di ciascun elemento. Infine, prima di concludere, è garantito che venga aggiunto un carattere di nuova riga attraverso una chiamata al sistema che utilizza la stringa "newline", precedentemente dichiarata nella sezione .data del programma. Di seguito è riportato il codice corrispondente:

```
427 PRINT:
428     beq s6 zero end_print
429     addi t6 s4 0
430 
431 check:
432     lb t3 0(t6)
433     bne t3 s7 print_loop
434     addi t6 t6 5
435     j check
436 
437 print_loop:
438     lb a0 0(t6)
439     li a7 11
440     ecall
441     lw t4 1(t6)
442     beq t4 s4 end_print
443     addi t6 t6 5
444     j check
445 
446 end_print:
447     la a0 newline
448     li a7 4
449     ecall
450     jr ra
```

REV

La quarta operazione implementata è chiamata REV e ha il compito di invertire l'ordine degli elementi nella lista. L'approccio seguito da questo algoritmo si basa sull'idea di immagazzinare gli elementi nella stack e successivamente recuperarli, sfruttando così l'inversione naturale che avviene durante l'accesso in ordine inverso.

L'algoritmo inizia con una verifica del numero di elementi presenti nella lista. Se il numero è zero, l'algoritmo termina senza compiere alcuna azione. In caso contrario, riserva uno spazio in memoria della dimensione di 30 byte nella pila (questo caso copre l'ipotesi massima di avere 30 comandi ADD validi come input nella lista). Le funzioni add_to_stack e to_stack vengono utilizzate per attraversare la catena degli elementi e inserirli nella pila. Una volta raggiunta la fine della catena, viene attivata la funzione add_to_chain che estrae gli elementi singolarmente dalla pila e, ad ogni iterazione, li inserisce nuovamente nella catena. Dato che

l'operazione di inversione non altera il numero di elementi nella lista, il criterio di terminazione è basato sulla catena stessa, ovvero monitorando costantemente i puntatori dei vari elementi.

Al termine dell'algoritmo, lo spazio precedentemente allocato nella pila viene deallocato. Di seguito è riportato il codice corrispondente:

```
REV:
    beq s6 zero empty_chain
    addi sp sp -30                              #massimo numero di elementi
    addi t6 s4 0

to_stack:
    lb t3 0(t6)
    bne t3 s7 add_to_stack
    addi t6 t6 5
    j to_stack

add_to_stack:
    sb t3 0(sp)
    addi sp sp 1
    lw t4 1(t6)
    beq t4 s4 end_chain
    addi t6 t6 5
    j to_stack

end_chain:
    addi sp sp -1
    addi t6 s4 0
    j to_chain

to_chain:
    lb t3 0(t6)
    bne t3 s7 add_to_chain
    addi t6 t6 5
    j to_chain


add_to_chain:
    lb t3 0(sp)
    sb t3 0(t6)
    addi sp sp -1
    lw t4 1(t6)
    beq t4 s4 end_stack
    addi t6 t6 5
    j to_chain

empty_chain:
    jr ra

end_stack:
    addi sp sp 30
    jr ra
```

SSX / SDX

Le seguenti implementazioni riguardano l'operazione di shift a destra e shift a sinistra sugli elementi della lista. Per realizzare questi algoritmi, è stato scelto di copiare gli elementi della lista da una posizione di memoria definita dall'indirizzo nel registro s9, generando così un array di caratteri su cui operare. Nel caso in cui il numero di caratteri nella lista sia zero o uno, l'algoritmo non compie alcuna azione. Successivamente, un altro indirizzo di memoria arbitrario viene memorizzato nel registro s10. Da questo indirizzo, attraverso una serie di passaggi, viene ottenuto un array di caratteri shiftato verso destra o verso sinistra, a seconda dell'algoritmo specifico. Successivamente, mediante l'utilizzo di funzioni dedicate, gli elementi vengono copiati dalla “stringa” shiftata nella catena.

Le due procedure sono leggermente diverse ma concettualmente analoghe: nel caso dello SDX, dopo aver copiato gli elementi dalla catena nella prima stringa, si preserva il byte nell'ultima posizione dell'array di caratteri (sostituendolo con il carattere flag). Nella fase in cui gli elementi vengono copiati nella stringa shiftata, il primo elemento inserito è il carattere appena salvato, seguito dagli altri elementi.

Per quanto riguarda lo SSX, il procedimento è simile, con la differenza che il carattere da conservare in un registro è il primo dell'array. Questo carattere verrà posizionato nella prima posizione, utilizzando sempre il carattere flag come riferimento per determinare quando interrompere lo scorrimento dell'array di caratteri. Oltre ai registri s9 e s10, vengono utilizzati registri temporanei da t0 a t6.

L'approccio coinvolge quindi la creazione di un array di caratteri su cui eseguire l'operazione, il salvataggio di indirizzi di memoria nei registri s9 e s10, e l'utilizzo di funzioni specifiche per copiare gli elementi nella catena. Di seguito una porzione significativa del codice:

```
588 SDX:
589     beq s6 zero end_SDX
590     li t0 1
591     beq s6 t0 end_SDX
592     li s9 0x00600000
593     addi t0 s9 0
594     addi t6 s4 0
595 
596 stringify_SDX:
597     lb t3 0(t6)
598     bne t3 s7 to_string_SDX
599     addi t6 t6 5
600     j stringify_SDX
601 
602 to_string_SDX:
603     sb t3 0(t0)
604     lw t4 1(t6)
605     beq t4 s4 save_last_SDX
606     addi t0 t0 1
607     addi t6 t6 5
608     j stringify_SDX
```

```
save_last_SDX:
    lb t1 0(t0)                         #salvo l'ultimo char
    sb s7 0(t0)                         #metto il flag al posto dell'ultimo elemento
    li s10 0x00525000                   #stringa shiftata
    addi t0 s10 0
    sb t1 0(t0)                         #setto l'ultimo char come primo nella stringa shiftata
    addi t1 s9 0
    addi t0 t0 1
    j execute_SDX

execute_SDX:
    lb t2 0(t1)                         #carico il carattere corrente per le verifiche
    beq t2 s7 flag_SDX
    sb t2 0(t0)
    addi t0 t0 1
    addi t1 t1 1
    j execute_SDX

flag_SDX:
    sb s7 0(t0)
    j back_in_chain_SDX

back_in_chain_SDX:
    addi t0 s10 0
    addi t6 s4 0
    j string_to_chain_SDX

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

next_SDX:
    addi t6 t6 5
    j string_to_chain_SDX

end_SDX:
    jr ra
```

```
SSX:
    beq s6 zero end_SSX
    li t0 1
    beq s6 t0 end_SSX
    li s9 0x00600000
    addi t0 s9 0
    addi t6 s4 0
```

```
668 to_string__SSX:
669     sb t3 0(t0)
670     lw t4 1(t6)
671     beq t4 s4 save_first_SSX
672     addi t0 t0 1
673     addi t6 t6 5
674     j stringify__SSX
675 
676 save_first_SSX:
677     addi t0 t0 1
678     sb s7 0(t0)                         #metto il flag alla fine di s9
679     addi t0 s9 0
680     lb t1 0(t0)                         #salvo il primo carattere che poi sara' l'ultimo
681     li s10 0x00525000
682     addi t2 s10 0
683     addi t0 t0 1
684     j execute_SSX
685 
686 execute_SSX:
687     lb t3 0(t0)
688     beq t3 s7 last_SSX
689     sb t3 0(t2)
690     addi t2 t2 1
691     addi t0 t0 1
692     j execute_SSX
693 
694 last_SSX:
695     sb t1 0(t2)
696     addi t2 t2 1
697     sb s7 0(t2)
698     j back_in_chain_SSX
699 
700 back_in_chain_SSX:
701     addi t0 s10 0
702     addi t6 s4 0
703     j string_to_chain_SSX
704 
```

```
705 string_to_chain_SSX:
706     lb t1 0(t0)                         #carica in un registro i valori della stringa
707     beq t1 s7 end_SSX
708     lb t2 0(t6)                         #controlla se e' un nodo cancellato logicamente
709     beq t2 s7 next_SSX
710     sb t1 0(t6)
711     addi t0 t0 1
712     addi t6 t6 5
713     j string_to_chain_SSX
714 
715 next_SSX:
716     addi t6 t6 5
717     j string_to_chain_SSX
718 
719 end_SSX:
720     jr ra
721 
```

SORT

L’ operazione implementata è il SORT, che organizza gli elementi della lista seguendo un determinato ordine. L'ordinamento è definito come segue:

- Le lettere maiuscole (ASCII da 65 a 90 inclusi) sono sempre considerate superiori alle lettere minuscole.
- Le lettere minuscole (ASCII da 97 a 122 inclusi) sono sempre considerate superiori ai numeri.
- I numeri (ASCII da 48 a 57 inclusi) sono sempre considerati superiori ai caratteri extra che non sono né lettere né numeri.

All'interno di ciascuna categoria, l'ordine è dettato dal codice ASCII. Ad esempio, date due lettere maiuscole x e x', x < x' se e solo se ASCII(x) < ASCII(x'). Lo stesso principio vale per le lettere minuscole, i numeri e i caratteri extra.

L'idea alla base dell'implementazione è di affrontare la sfida posta dal fatto che i caratteri extra non sono consecutivi nella tabella ASCII. Poiché questi caratteri devono apparire per primi nell'ordinamento, è necessario assegnare loro una classificazione basata su un'altra metrica. È stato deciso di suddividere i caratteri in quattro categorie:

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

L'algoritmo confronta gli elementi a coppie (salvati nei registri temporanei t0 e t2), determinando brutalmente la categoria dei due elementi (0, 1, 2 o 3), confrontandoli con i limiti delle diverse categorie di caratteri (salvati nei registri a2 e a3). Una volta assegnate le categorie, viene eseguita la funzione compare, che verifica a quale categoria appartengono i due caratteri. Se il primo carattere appartiene a una categoria "superiore" al secondo carattere, l'ordine relativo è corretto. Se i caratteri appartengono alla stessa categoria, viene chiamato il metodo compare_ASCII, che li ordina in base al loro codice ASCII. Se il primo carattere appartiene a una categoria "inferiore" al secondo carattere, avviene uno scambio.

Infine, il metodo increment_loop scorre attraverso la lista per confrontare i successivi due caratteri tra loro. Di seguito una porzione significativa del codice:

```
499 SORT:
500     lw s8, 1(s4)
501     addi a6, s4, 0
502     lw t3, 1(s4)
503     beq t3, zero, end_sort                          #se non ci sono elementi termina
504     add t3, s4, zero
505     lw t1, 1(t3)
506     beq t1, s4, end_sort                            #la lista contiene un solo elemento (gia' ordinato)
507
508 sorting_loop:
```

```
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, check_lowercase_first
    bgt t2, a3, check_lowercase_first
    li a4, 0                                    #se arrivo qui vuol dire che e' un carattere maiuscolo --> indicato con 0
    j check_uppercase_second

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

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

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

check_uppercase_second:
    li a2, 65
    li a3, 90
    blt t0, a2, check_lowercase_second
    bgt t0, a3, check_lowercase_second
    li a5, 0                                    #se arrivo fin qui vuol dire che e' un carattere maiuscolo --> indicato con 0
    j compare

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

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

check_extra_second:
    li a5, 3                                    #se arrivo fin qui vuol dire che e' un carattere speciale --> indicato con 3
```

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

compare_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

end_sort:
    jr ra
```