# Relazione Progetto Architetture degli Elaboratori 2022-2023:


| | |
|---|---|
| Autore | Ede Boanini |
| Matricola | [redacted] |
| E-mail | [redacted] |
| Data di consegna | 09/09/2023 |
| Versione RIPES (IDE) | v2.2.5 |


## Obbiettivo e descrizione del progetto:

L'obbiettivo del progetto è la **creazione e gestione di una lista concatenata circolare** implementata utilizzando il codice RISC-V.
Questa struttura dati dinamica è composta da una sequenza di nodi, ciascuno contenente un campo di dati arbitrario e un riferimento che punta al nodo successivo.
Questo tipo di struttura permette l'accesso solo in maniera sequenziale.

**Ogni nodo della lista avrà la dimensione totale di 5 byte di cui:**

- I primi 4 byte (byte 0-3) saranno riservati al PAHEAD, cioè il puntatore all'elemento successivo oppure a sé stesso se è l'unico elemento presente nella lista
- Il byte 4 (byte 4) conterrà l'informazione del nodo, rappresentato da un codice ASCII compreso nell'intervallo [32, 125]

È importante aggiungere che i puntatori all'elemento successivo di ogni nodo, avranno dimensione in memoria di 32 bit, equivalente a una word RISC-V.

Implementare le seguenti operazioni che permetteranno la manipolazione della struttura dati:

- ADD (inserimento di un nuovo nodo)
- DEL (rimozione di un nodo)
- PRINT (stampa a video la lista concatenata circolare)
- SORT (ordinamento della lista concatenata circolare)
- SDX (rotazione in senso orario dei nodi della lista concatenata circolare)
- SSX (rotazione in senso antiorario dei nodi della lista concatenata circolare)
- REV (inversione dei nodi della lista concatenata circolare)

*Nota ordinamento caratteri ASCII:
Nel momento in cui l'utente eseguirà l'operazione SORT, i caratteri ASCII dovranno seguire il seguente ordine: simboli < numeri < minuscole < maiuscole.
Esempio:
la lista A2;b7 si ordinerà nella seguente maniera ;27bA

## Descrizione della sezione .data del programma:

Questa sezione inizializza la variabile string '`listInput`' che conterrà una serie di comandi separati dal simbolo ~ (codice ASCII: 126).
Alcuni comandi possono contenere dei parametri e se ben formattati, eseguono l'operazione.
I comandi contenuti nella variabile '`listInput`'' non dovranno essere più di 30.

Importante specificare che i comandi, per poter essere accettati dal programma ed eseguire le operazioni a loro assegnate dovranno essere formattati nel seguente modo:

- ADD(char), dovrà contenere un solo carattere ASCII tra le parentesi
- DEL(char), dovrà contenere un solo carattere ASCII tra le parentesi
- PRINT
- SORT
- SDX
- SSX
- REV

Esempio di tutti i comandi accettati dal programma:

```
10 .data
11 listInput: .string "ADD(1) ~ SSX ~ ADD(a) ~ ADD(B) ~ ADD(9) ~ PRINT~SORT~DEL(B) ~REV~SDX~ DEL(a)"
```

| Comandi accettati | Comandi NON accettati |
|---|---|
| ADD(1) | ADD(11), add(1), add(11), AD D(1), ADD (1) |
| DEL(1) | DEL(11), del(1), del(11), DE L(1), DEL (1) |
| PRINT | print, PRINT(), PRINT(char) |
| SORT | sort, SORT(), SORT(char) |
| SDX | sdx, SDX(), SDX(char) |
| SSX | ssx, SSX(), SSX(char) |
| REV | rev, REV(), REV(char) |

# Descrizione del main del programma:

La funzione '`main`' definita nella sezione `.text` contiene la dichiarazione e l'inizializzazione delle variabili e il codice che controlla i comandi inseriti dall'utente in una stringa (listInput), analizzando ogni comando carattere per carattere.

| | | |
|---|---|---|
| s0 | Puntatore alla testa della stringa | conterrà la stringa listInput (partenendo da indirizzo 0x10000000) |
| s1 | Indirizzo in memoria della testa della lista concatenata circolare | (da 0x00000900) |
| s2 | Loop counter | servirà per poterlo incrementare di volta in volta mentre scorriamo i caratteri dei comandi contenuti in s0 (listInput) |
| s3 | Numero di comandi inseriti | terrà il conto del numero di comandi inseriti (quelli ben formattati ed eseguiti nel programma) |
| s4 | Numero massimo di comandi inseribili | viene inizializzata e dichiarata a 30, cioè il numero massimo di comandi inseribili |
| s5 | Indirizzo in memoria della testa della lista concatenata circolare | rappresenta l'indirizzo in memoria da cui il programma inizierà ad inserire gli elementi (da 0x00000900); utile per la procedura add |

### Descrizione ad alto livello:

Il programma inizia leggendo una serie di comandi di una stringa contenuta nel registro s0 (listInput). Ogni comando è composto da una o più lettere seguite da un eventuale carattere tra

parentesi. Il programma può riconoscere comandi come "ADD(char)", "DEL(char)", "PRINT", "SORT", "SDX", "SSX", o "REV".

- Il ciclo `'loop_commands'` esamina la stringa carattere per carattere finché non raggiunge la fine o viene raggiunto un numero massimo di comandi (30).
- Per ogni carattere letto, il programma verifica se corrisponde a una delle lettere iniziali dei comandi (A, D, P, R, S). [Importante dire che SORT, SSX e SDX hanno la stessa prima lettera e per poter capire qual è tra i tre, controllo anche il secondo carattere in 'checkSecondLetter' prima di eseguire la procedura corretta].
- Se trova una corrispondenza, il programma verifica se il resto del comando coincide con uno dei comandi completi. Se sì, esegue la procedura corrispondente. Ad esempio, se trova "ADD(char)", esegue la procedura add (con 'jal add') e memorizza il carattere in a0. Per "DEL(char)", esegue la procedura del (con 'jal del') e memorizza il carattere in a0. Per "PRINT", esegue print la funzione print (con 'j print'). Per "SORT", esegue la procedura sort (con 'jal sort'). Per "SDX", esegue la procedura sdx (con 'jal sdx').. Per "SSX", esegue la procedura ssx (con 'jal ssx'). Per "REV", esegue la procedura rev (con 'jal rev').
- Il programma tiene traccia del numero di comandi inseriti (s3) solo se il comando è ben formato e viene eseguito correttamente
- Dopo aver eseguito il comando, passa al seguente con 'nextCommand', che controllerà se il carattere successivo è uno spazio o una parentesi chiusa. Continuerà a controllare fino a trovare la tilde (~) che è fondamentale perché è ciò che separa un comando da un altro.

| Uso dei registri | |
|---|---|
| s0, s1, s2, s3, s4, s5 | La descrizione dell'utilizzo di questi registri è fornita nella tabella precedente |
| a0 | Il carattere ASCII contenuto in a0 viene passato come argomento alla funzione. Utile per ADD(char) e DEL(char) |
| a1 | Utilizzato per il confronto del carattere del comando |
| t0, t1, t2, | Registri temporanei |
| s1 | Contiene l'indirizzo della testa della lista |
| Uso della memoria | |
| Stack | Conserva il registro di ritorno |

# Descrizione delle procedure:

## ADD

L'obbiettivo di questa procedura è aggiungere il carattere tra parentesi del comando ADD(char) come primo elemento se la lista è vuota oppure in coda se esiste più di un elemento alla lista concatenata circolare.

Dato che i caratteri ASCII accettabili per poter effettuare l'inserimento sono compresi nell'intervallo [32, 125], effettuiamo questo controllo. Se non è appartiene a questo intervallo, scarta il comando e passa al successivo, altrimenti è d'obbligo controllare se è un primo inserimento oppure un successivo inserimento:

- **Primo inserimento:**
  La lista sarà vuota quindi basta impostare il valore del puntatore all'elemento successivo uguale alla testa della lista e poi salvare il carattere ASCII.

- **Successivo inserimento:**
  Aggiungiamo il nuovo nodo in coda alla lista. Per prima cosa, effettua la ricerca dell'ultimo elemento (colui che ha il puntatore all'elemento successivo uguale all'indirizzo della testa della lista), dopo averlo identificato, aggiunge il nuovo elemento che punterà alla testa della lista e il precedente (che prima era l'ultimo) che punta a quello appena inserito, salvando ovviamente poi il carattere ASCII.

| Uso dei registri | |
|---|---|
| sp (registro dello stack pointer) | Utilizzato per salvare il registro di ritorno (ra). |
| ra | il registro ra conterrà l'indirizzo di ritorno precedentemente salvato nello stack e tornerà al punto di chiamata della procedura, cioè all'istruzione successiva di 'jal add' |
| t0, t1, t3, t4, t6 | Registri temporanei |
| a0 | Contiene il carattere ASCII da aggiungere alla lista |
| s1 | Contiene l'indirizzo della testa della lista |
| s5 | Utilizzato per tenere traccia dell'indirizzo in cui verrà creato un il nuovo elemento |
| **Uso della memoria** | |
| Stack | Conserva il registro di ritorno |

*Esempio:*

```
.data
listInput: .string "ADD(1) ~ ADD(a) ~ ADD(a) ~ ADD(B) ~ ADD(;) ~ ADD(9)"
```

*Visualizzazione del contenuto della memoria:*

| Address | Word | Byte 0 | Byte 1 | Byte 2 | Byte 3 |
|---|---|---|---|---|---|
| 0x00000934 | x | x | x | x | x |
| 0x00000930 | x | x | x | x | x |
| 0x0000092c | (9) 0x00000039 | 0x39 | 0x00 | 0x00 | 0x00 |
| 0x00000928 | 0x00000900 | 0x00 | 0x09 | 0x00 | 0x00 |
| 0x00000924 | (;) 0x0000003b | 0x3b | 0x00 | 0x00 | 0x00 |
| 0x00000920 | 0x00000928 | 0x28 | 0x09 | 0x00 | 0x00 |
| 0x0000091c | (B) 0x00000042 | 0x42 | 0x00 | 0x00 | 0x00 |
| 0x00000918 | 0x00000920 | 0x20 | 0x09 | 0x00 | 0x00 |
| 0x00000914 | (a) 0x00000061 | 0x61 | 0x00 | 0x00 | 0x00 |
| 0x00000910 | 0x00000918 | 0x18 | 0x09 | 0x00 | 0x00 |
| 0x0000090c | (a) 0x00000061 | 0x61 | 0x00 | 0x00 | 0x00 |
| 0x00000908 | 0x00000910 | 0x10 | 0x09 | 0x00 | 0x00 |
| 0x00000904 | (1) 0x00000031 | 0x31 | 0x00 | 0x00 | 0x00 |
| 0x00000900 | 0x00000908 | 0x08 | 0x09 | 0x00 | 0x00 |
| 0x000008fc | Primo elemento | (PAHEAD) puntatore all'elemento successivo | | Crattere ASCII | x |

# DEL

L'obbiettivo di questa procedura è eliminare il carattere o più caratteri dalla lista concatenata circolare.

Vengono gestiti i seguenti casi:

- **La lista è vuota**
  Il comando viene scartato

- **Il carattere da eliminare è l'unico elemento della lista**
  Se abbiamo solo un elemento nella lista ed è colui che dobbiamo eliminare, imposta a zero il valore del puntatore ed il carattere e stampa, altrimenti stampa l'elemento presente nella lista.

- **Il carattere da eliminare è l'elemento testa della lista**
  Se la lista è composta da più di un elemento, scorrimi la lista fino a trovare il carattere da eliminare (t3), caricandomi in t2 il suo puntatore all'elemento successivo e in t4 anche il puntatore all'elemento successivo dell'elemento da eliminare.
  Se il carattere corrente (t3) corrisponde a quello da eliminare (a0), ed è la testa, scorrimi la lista fino a trovare l'ultimo elemento e a questo punto, sapendo che il secondo elemento diventerà la nuova testa (abbiamo salvato l'indirizzo del secondo elemento in t2), dovrò modificare il puntatore al successivo dell'ultimo elemento e assegnarli il valore dell'indirizzo del secondo elemento (nuova testa della lista). Dopo aver rimosso la testa, continua a cercare se ci sono altri elementi da eliminare.

- **Il carattere da eliminare è in mezzo alla lista**
  Se la lista è composta da più di un elemento, scorrimi la lista fino a trovare il carattere da eliminare (t3), caricandomi in t2 il suo puntatore all'elemento successivo e in t4 anche il puntatore all'elemento successivo dell'elemento da eliminare.
  Se il carattere corrente (t3) corrisponde a quello da eliminare (a0), scorrimi la lista dalla testa per trovare l'elemento che ha come puntatore al successivo l'indirizzo dell'elemento da eliminare. Una volta trovato l'elemento precedente da quello che dobbiamo eliminare (basta vedere che il valore del puntatore sia uguale all'indirizzo di dove si trova l'elemento da eliminare), gli modifichiamo il valore del puntatore al successivo sostituendolo con l'indirizzo dell'elemento subito dopo quello da eliminare. Dopo aver rimosso l'elemento, continua a cercare se ci sono altri elementi da eliminare.

- **Il carattere da eliminare è in coda alla lista**
  Se la lista è composta da più di un elemento, scorrimi la lista fino a trovare il carattere da eliminare (t3), caricandomi in t2 il suo puntatore all'elemento successivo e in t4 anche il puntatore all'elemento successivo dell'elemento da eliminare.
  Se il carattere corrente (t3) corrisponde a quello da eliminare (a0), ed è l'ultimo elemento della lista, riscorrerò la lista dalla testa per trovare l'elemento precedente che ha come puntatore al successivo l'indirizzo dell'elemento da eliminare.
  Trovato il precedente, basterà modificare il suo puntatore al successivo e assegnarli l'indirizzo della testa della lista. Dopo aver rimosso l'ultimo elemento, continua a cercare se ci sono altri elementi da eliminare.

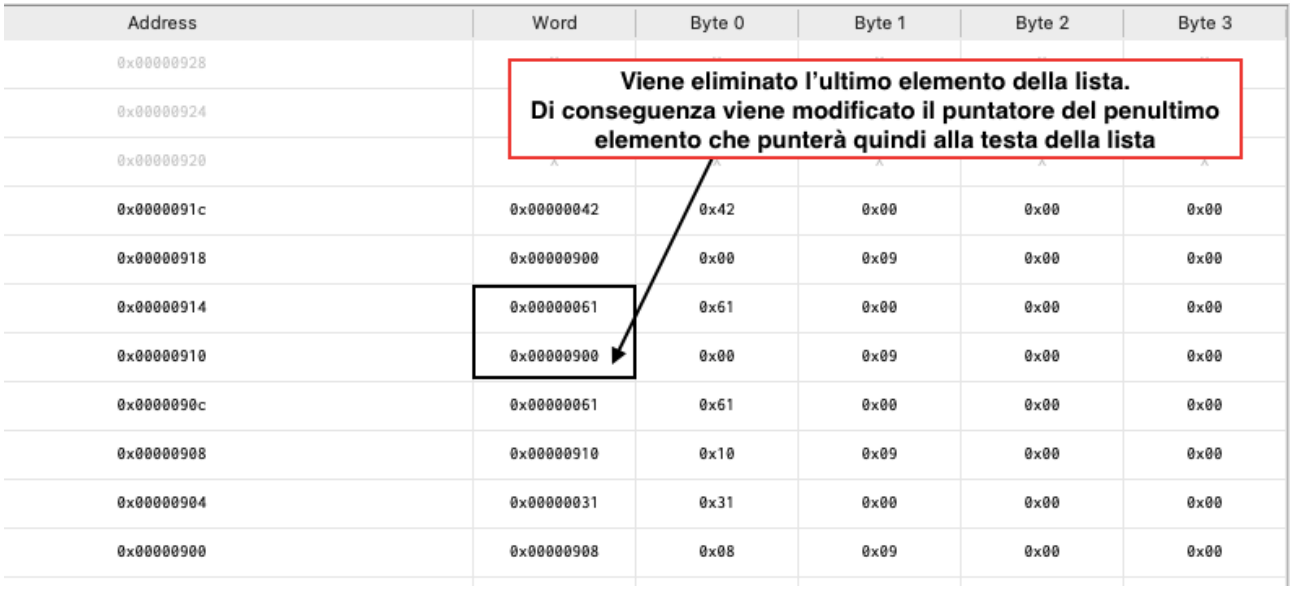
| Uso dei registri | |
|---|---|
| sp (registro dello stack pointer) | Utilizzato per salvare il registro di ritorno (ra). |

| | |
|---|---|
| ra | il registro ra conterrà l'indirizzo di ritorno precedentemente salvato nello stack e tornerà al punto di chiamata della procedura, cioè all'istruzione successiva di 'jal del |
| t0, t1, t2, t3, t4, t5, t6 | Registri temporanei |
| a0 | Contiene il carattere ASCII da eliminare dalla lista |
| s1 | Contiene l'indirizzo della testa della lista |
| s7, s9 | Utilizzati come registri di appoggio per completare correttamente la procedura |
| Uso della memoria | |
| Stack | Conserva il registro di ritorno |

*Esempio:*

```
.data
listInput: .string "ADD(1) ~ ADD(a) ~ ADD(a) ~ ADD(B) ~ DEL(B)"
```

*Visualizzazione del contenuto della memoria:*




| Address | Word | Byte 0 | Byte 1 | Byte 2 | Byte 3 |
|---|---|---|---|---|---|
| 0x00000928 | | | | | |
| 0x00000924 | | | | | |
| 0x00000920 | | | | | |
| 0x0000091c | 0x00000042 | 0x42 | 0x00 | 0x00 | 0x00 |
| 0x00000918 | 0x00000900 | 0x00 | 0x09 | 0x00 | 0x00 |
| 0x00000914 | 0x00000061 | 0x61 | 0x00 | 0x00 | 0x00 |
| 0x00000910 | 0x00000900 | 0x00 | 0x09 | 0x00 | 0x00 |
| 0x0000090c | 0x00000061 | 0x61 | 0x00 | 0x00 | 0x00 |
| 0x00000908 | 0x00000910 | 0x10 | 0x09 | 0x00 | 0x00 |
| 0x00000904 | 0x00000031 | 0x31 | 0x00 | 0x00 | 0x00 |
| 0x00000900 | 0x00000908 | 0x08 | 0x09 | 0x00 | 0x00 |

## PRINT

L'obbiettivo di questa procedura è stampare a video i caratteri ASCII contenuti nella lista concatenata circolare.

Controlla se la lista è vuota verificando se il puntatore all'elemento successivo è zero (nel caso la lista fosse vuota, stamperà a video uno spazio, codice ASCII 32), altrimenti stampa a video il primo carattere e controlla se è l'unico elemento della lista. Se non è l'unico esegue il ciclo '`print_loop`' che consentirà di stampare gli altri caratteri ASCII della lista fino a che non troverà l'ultimo elemento (colui che ha il valore del puntatore all'elemento successivo uguale all'indirizzo della testa della lista).

| Uso dei registri | |
|---|---|
| t0, t1 | Registri temporanei |
| a0 | Contiene inizialmente il simbolo ~ e successivamente i caratteri ASCII della lista da stampare a video |
| s1 | Contiene l'indirizzo della testa della lista |

# SORT

L'obbiettivo di questa procedura è ordinare i caratteri ASCII contenuti nella lista concatenata circolare secondo questo ordine: simboli < numeri < minuscole < maiuscole.

Per poter mantenere questo ordine, dato che i simboli sono sparsi nella tabella ASCII e le maiuscole sono minori delle minuscole, ho raggruppato i simboli in questo intervallo [0,31], mentre le maiuscole in questo [128, 153].

Spiegazione raggruppamento **simboli** nel nuovo intervallo [0,31]:
Nella tabella ASCII i simboli sono compresi nei seguenti intervalli: [32, 47], [58, 64], [91, 96], [123,125]. In totale abbiamo 32 simboli. Quindi andremo a sottrarre il valore del simbolo ASCII in modo da farlo rientrare in questo intervallo [0,31] dato che ci stanno tutti e 32 i simboli:

- [32, 47], a questi simboli gli sottraiamo 32:
  $\{32-32=0\},..,\{47-32=15\}$ quindi questi saranno compresi nel nuovo intervallo [0,15]
- [58, 64], a questi simboli gli sottraiamo 42:
  $\{58-42=16\},..,\{64-42=22\}$ quindi questi saranno compresi nel nuovo intervallo [16,22]
- [91,96], a questi simboli gli sottraiamo 68:
  $\{91-68=23\},..,\{96-68=28\}$ quindi questi saranno compresi nel nuovo intervallo [23,28]
- [123, 125], a questi simboli gli sottraiamo 94:
  $\{123-94=29\},..,\{125-94=31\}$ quindi questi saranno compresi nel nuovo intervallo [29,31]

Spiegazione raggruppamento **maiuscole** nel nuovo intervallo [128,153]:
Le maiuscole sono comprese nel seguente intervallo: [65,90].
Dato che nella tabella ASCII le minuscole sono nell'intervallo [97, 122] e l'ultimo carattere della tabella è il 127, possiamo mettere in atto la stessa tecnica usata per i simboli ma sommando alle maiuscole il valore di 63:

- [65, 90], a queste maiuscole gli sommiamo 63:
  $\{65+63=128\},..,\{90+63=153\}$ quindi questi saranno compresi nel nuovo intervallo [128,153]

Gli altri caratteri ASCII come numeri e minuscole non era necessario alterarli.
Di conseguenza ottengo l'ordine desiderato:

- Simboli [0, 31]
- Numeri [48, 57]
- Minuscole [97, 122]
- Maiuscole [128, 153]

Dopodiché ho implementato il Bubble Sort ricorsivo per ordinare la lista:
Questo algoritmo confronta le coppie di elementi partendo dalla testa della lista e avanzando verso la coda. Se il secondo elemento è minore del primo (nella coppia da confrontare), avviene lo scambio e viene incrementato il contatore t0 (registro che tiene conto del numero degli scambi). L'algoritmo continua ad eseguire questi passaggi fino a quando non raggiunge la fine della lista e il numero di scambi è uguale a zero (Vorrebbe dire che la lista è ordinata poiché non ha effettuato scambi).

È importante specificare che prima di controllare a quale categoria appartiene il carattere ASCII, (se simbolo, numero, minuscola o maiuscola) mi salvo temporaneamente i caratteri da confrontare in s10 e s11. Questo è utile perché nel caso i caratteri fossero simboli o maiuscole, vengono trasformati e quindi mi sarà utile per poterli stampare correttamente a video alla fine del processo di ordinamento.

I puntatori non vengono alterati, ma vengono scambiati i caratteri veri e propri in memoria.

| Uso dei registri | |
|---|---|
| sp (registro dello stack pointer) | Utilizzato per salvare il registro di ritorno (ra). |
| ra | il registro ra conterrà l'indirizzo di ritorno precedentemente salvato nello stack e tornerà al punto di chiamata della procedura, cioè all'istruzione successiva di 'jal sort |
| t0 | Tiene traccia del numero di scambi effettuati durante l'ordinamento |
| t1 | Usato per memorizzare il primo carattere della lista |
| t2, t4 | Puntatori |
| t3 | Il primo carattere della coppia da confrontare (il BubbleSort confronta a coppie di elementi) |
| t5 | Il secondo carattere della coppia da confrontare |
| t6 | Utilizzato per determinare che tipo di carattere (simbolo, numero, minuscola, maiuscola) è t3 e t5, la coppia di caratteri da confrontare durante l'ordinamento |
| a0 | Contiene il carattere ASCII da aggiungere alla lista |
| s1 | Contiene l'indirizzo della testa della lista |
| s10, s11 | Caratteri ASCII prima della trasformazione (utile se è un simbolo o maiuscola) |
| Uso della memoria | |
| Stack | Conserva il registro di ritorno, e in più lo stack serve per memorizzare temporaneamente il valore di t3, che sarà di fondamentale importanza per il confronto durante l'ordinamento |

*Esempio:*

```
.data
listInput: .string "ADD(B) ~ ADD(a) ~ ADD(1) ~ ADD(a) ~ SORT"
```

*Visualizzazione del contenuto della memoria:*

| Address | Word | Byte 0 | Byte 1 | Byte 2 | Byte 3 |
|---|---|---|---|---|---|
| 0x0000091c | a | a | | | |
| 0x00000918 | | | | | |
| 0x00000914 | 1 | 1 | | | |
| 0x00000910 | | | | | |
| 0x0000090c | a | a | | | |
| 0x00000908 | | | | | |
| 0x00000904 | B | B | | | |
| 0x00000900 | | | | | |

PRIMA DELL'ORDINAMENTO

| Address | Word | Byte 0 | Byte 1 | Byte 2 | Byte 3 |
|---|---|---|---|---|---|
| 0x0000091c | B | B | | | |
| 0x00000918 | | | | | |
| 0x00000914 | a | a | | | |
| 0x00000910 | | | | | |
| 0x0000090c | a | a | | | |
| 0x00000908 | | | | | |
| 0x00000904 | 1 | 1 | | | |
| 0x00000900 | | | | | |

DOPO L'ORDINAMENTO

## SDX

L'obbiettivo di questa procedura è spostare gli elementi verso destra, rendendo l'ultimo elemento il nuovo primo elemento della lista circolare.

Inizia controllando che la lista non sia vuota, e stampa il carattere nel caso fosse l'unico elemento della lista poiché non occorre eseguire uno spostamento verso destra dato che esiste solo un elemento. In caso contrario, cioè se la lista contiene più di un elemento, procede a spostare l'ultimo elemento in prima posizione cambiando solo il valore di s1, che contiene l'indirizzo della testa della lista. Questo vuol dire che trovato l'ultimo elemento, cioè con puntatore al successivo uguale all'indirizzo della testa, cambia il valore di s1, impostando quest'ultimo elemento come primo.
In questo caso non alteriamo né caratteri, né i puntatori in memoria, ma semplicemente impostiamo l'indirizzo in cui si trova l'ultimo elemento come testa.

| Uso dei registri | |
|---|---|
| sp (registro dello stack pointer) | Utilizzato per salvare il registro di ritorno (ra). |
| ra | il registro ra conterrà l'indirizzo di ritorno precedentemente salvato nello stack e tornerà al punto di chiamata della procedura, cioè all'istruzione successiva di 'jal sdx |
| t0, t3 | Registri temporanei |
| t1 | Importante perché conterrà l'indirizzo del nuovo primo elemento |
| s1 | Contiene l'indirizzo della testa della lista |
| Uso della memoria | |
| Stack | Conserva il registro di ritorno |

## SSX

L'obbiettivo di questa procedura è spostare gli elementi verso sinistra, rendendo il primo elemento il nuovo ultimo e il secondo elemento il nuovo primo elemento della lista circolare.

Come prima cosa controlla che la lista non sia vuota, e stampa il carattere nel caso fosse l'unico della lista poiché non occorre eseguire uno spostamento verso sinistra dato che esiste un solo elemento. In caso contrario, cioè se la lista contiene più di un elemento, procede a spostare il primo elemento in ultima posizione cambiando anche solo in questo caso il valore di s1 (indirizzo della testa della lista).

Questo vuol dire che il secondo elemento della lista diventerà il primo. Quindi basta identificare l'indirizzo del secondo elemento e impostarlo come nuova testa della lista.

| Uso dei registri | |
|---|---|
| sp (registro dello stack pointer) | Utilizzato per salvare il registro di ritorno (ra). |
| ra | il registro ra conterrà l'indirizzo di ritorno precedentemente salvato nello stack e tornerà al punto di chiamata della procedura, cioè all'istruzione successiva di 'jal sdx |
| t0 | Registro temporanei |
| s1 | Contiene l'indirizzo della testa della lista, e poi della nuova testa della lista |
| Uso della memoria | |
| Stack | Conserva il registro di ritorno |

## REV

L'obbiettivo di questa procedura è invertire l'ordine degli elementi della lista.

Come prima cosa controlla che la lista non sia vuota, e stampa il carattere nel caso fosse l'unico della lista poiché non è necessario invertire dato che ne esiste solo uno. In caso contrario, cioè se la lista contiene più di un elemento, mi procede a salvare tutti i caratteri nello stack fino all'ultimo elemento. Successivamente, estrae gli elementi dallo stack e li salva partendo dalla testa della lista. Questo processo inverte l'ordine degli elementi, mantenendo inalterati i puntatori in memoria.

Ho adottato la tecnica dello stack perché è una struttura dati LIFO, cioè Last In First Out, l'ultimo ad essere stato inserito nello stack è il primo ad essere estratto, ed è efficace per poter invertire la lista.

| Uso dei registri | |
|---|---|
| sp (registro dello stack pointer) | Utilizzato per salvare il registro di ritorno (ra). |
| ra | il registro ra conterrà l'indirizzo di ritorno precedentemente salvato nello stack e tornerà al punto di chiamata della procedura, cioè all'istruzione successiva di 'jal sdx |
| t0, t1, t2, t3, t4, t5, t6 | Registro temporanei |
| s1 | Contiene l'indirizzo della testa della lista |
| Uso della memoria | |
| Stack | Conserva il registro di ritorno.<br>In questa procedura lo stack è fondamentale per poter memorizzare temporaneamente i caratteri e poi salvarli definitivamente nella memoria a partire dalla testa della lista |

*Esempio:*

```
.data
listInput: .string "ADD(1) ~ ADD(2) ~ ADD(3) ~ ADD(4) ~ REV"
```

*Visualizzazione del contenuto della memoria:*

| Address | Word | Byte 0 | Byte 1 | Byte 2 | Byte 3 |
|---|---|---|---|---|---|
| 0x0000091c | 4 | 4 | | | |
| 0x00000918 | | | | | |
| 0x00000914 | 3 | 3 | | | |
| 0x00000910 | | | | | |
| 0x0000090c | 2 | 2 | | | |
| 0x00000908 | | | | | |
| 0x00000904 | 1 | 1 | | | |
| 0x00000900 | | | | | |

**PRIMA DI INVERTIRE LA LISTA**

| Address | Word | Byte 0 | Byte 1 | Byte 2 | Byte 3 |
|---|---|---|---|---|---|
| 0x0000091c | 1 | 1 | | | |
| 0x00000918 | | | | | |
| 0x00000914 | 2 | 2 | | | |
| 0x00000910 | | | | | |
| 0x0000090c | 3 | 3 | | | |
| 0x00000908 | | | | | |
| 0x00000904 | 4 | 4 | | | |
| 0x00000900 | | | | | |

**DOPO AVER INVERTITO LA LISTA**

# Test eseguiti nel programma:

Un comando mal formattato viene scartato e non aggiunto al numero di comandi eseguiti dal programma (s3 è il registro che contiene il numero di comandi eseguiti).
Per ogni test, viene mostrata la stringa di comandi eseguita e l'output della console.

**Primo test:**

Sono in tutto 15 comandi, e l'unico comando mal formattato è 'PRI', difatti verrà scartato e non contato come comando eseguito in s3.
Alla fine, `s3` avrà il valore di 14, che sono in comandi ben formattati ed eseguiti correttamente nel programma.

```
.data
listInput: .string "ADD(1) ~ ADD(a) ~ ADD(a) ~ ADD(B) ~ ADD(;) ~ ADD(9) ~SSX~SORT~PRINT~DEL(b) ~DEL(B) ~PRI~SDX~REV~PRINT"
```

```
~1~1a~1aa~1aaB~1aaB;~1aaB;9~aaB;91~;19aaB~;19aaB~;19aaB~;19aa~a;19a~a91;a~a91;a
```

**Secondo test:**

Sono in tutto 16 comandi, e quelli mal formattati sono ‘add(B), ADD, SORT(a), DEL(bb)’, difatti verranno scartati e non contati come comandi eseguiti in s3.
Alla fine, `s3` avrà il valore di 12, che sono in comandi ben formattati ed eseguiti correttamente nel programma.

```
.data
listInput: .string "ADD(1) ~ SSX ~ ADD(a) ~ add(B) ~ ADD(B) ~ ADD ~ ADD(9) ~PRINT~SORT(a)~PRINT~DEL(bb) ~DEL(B) ~PRINT~REV~SDX~PRINT"
```

```
~1~1~1a~1aB~1aB9~1aB9~1aB9~1a9~1a9~9a1~19a~19a
```

**Terzo test:**

Sono in tutto 16 comandi, e quelli mal formattati sono ‘add(r), add(5), del(r), SO RT’, difatti verranno scartati e non contati come comandi eseguiti in s3.
Alla fine, `s3` avrà il valore di 12, che sono in comandi ben formattati ed eseguiti correttamente nel programma.

```
.data
listInput: .string "REV~add(r)~ADD(A)~ADD(r)~add(5)~ ADD(1)~ADD(5)~ADD(e)~del(r)~ DEL(5)~REV~PRINT~SO RT~PRINT~ SORT ~ PRINT"
```

```
~A~Ar~Ar1~Ar15~Ar15e~Ar1e~e1rA~e1rA~e1rA~1erA~1erA
```

**Quarto test:**

Sono in tutto 13 comandi, e quelli mal formattati sono ‘ADD(42) e PRINT(A)’, difatti verranno scartati e non contati come comandi eseguiti in s3.
Alla fine, `s3` avrà il valore di 11, che sono in comandi ben formattati ed eseguiti correttamente nel programma.

```
.data
listInput: .string "ADD(o) ~ADD(G) ~ADD(42) ~ADD(7) ~ADD(@)~ADD(o)~PRINT~DEL(o)~SORT~PRINT~ REV~ PRINT(A) ~PRINT"
```

```
~o~oG~oG7~oG7@~oG7@o~oG7@o~G7@~@7G~@7G~G7@~G7@
```

**Quinto test:**

Sono in tutto 13 comandi, e quello mal formattato è ‘PRI NT’, difatti verrà scartato e non contato come comando eseguito in s3.
Alla fine, `s3` avrà il valore di 12, che sono in comandi ben formattati ed eseguiti correttamente nel programma.

```
.data
listInput: .string "ADD(1) ~ADD(2) ~ADD(1) ~ADD(4) ~ADD(;)~PRI NT~DEL(1)~SORT~SSX~ REV~ DEL(4)~DEL(2) ~PRINT"
```

```
~1~12~121~1214~1214;~24;~;24~24;~;42~;2~;~;
```

**Sesto test:**

Sono in tutto 12 comandi, e quello mal formattato è 'SSX(n), difatti verrà scartato e non contato come comando eseguito in s3.
Alla fine, `s3` avrà il valore di 11, che sono in comandi ben formattati ed eseguiti correttamente nel programma.

```
.data
listInput: .string "ADD(1) ~ADD(2) ~ADD(1) ~ADD(4) ~SSX(n)~DEL(1)~SORT~SSX~ REV~ DEL(4)~DEL(2) ~PRINT"
```

```
~1~12~121~1214~24~24~42~24~2~ ~
```

**Ulteriori test non presenti nell'IDE RIPES ma allegati sopra:

```
#REV~add(r)~ADD(A)~ADD(r)~add(5)~ ADD(1)~ADD(5)~ADD(e)~del(r)~ DEL(5)~REV~PRINT~SO
RT~PRINT~ SORT ~ PRINT

#ADD(1) ~ADD(2) ~ADD(1) ~ADD(4) ~ADD(;)~PRI NT~DEL(1)~SORT~SSX~ REV~ DEL(4)~DEL(2) ~PRINT
```