RELAZIONE PROGETTO ARCHITETTURE DEGLI ELABORATORI 2022-2023

Richiesta 2
Main 3
Parsing dei comandi 4
ADD 6
DEL 7
SDX 9
SSX 10
REV 11
PRINT 13
SORT 13
Funzioni ausiliarie 13
list_to_array 13
array_to_list 14
adjust_char 14
should_swap 16
quicksort 16

Richiesta

Il progetto di AE 22-23 mira all'implementazione di un codice RISC-V che gestisce alcune delle operazioni fondamentali per una lista concatenata circolare, tra le quali:

- **ADD** - Inserimento di un elemento
- **DEL** - Rimozione di un elemento
- **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
- **SORT** - Ordinamento della lista
- **PRINT** - Stampa della lista

Le operazioni verranno eseguite in base ai comandi forniti all'interno di una stringa denominata listInput. I comandi verranno formattati, nel modo specificato in seguito.

Main

All’interno del main vengono inizializzati i registri da s0 a s11, viene effettuato il parsing dei comandi e viene gestita la fine del programma.

Di seguito descrivo i valori che ho salvato nei registri da s0 a s11 all’interno del main:

- **s0:** Registro che funge da puntatore alla testa della lista circolare, inizialmente è valorizzato a 0 indicando che non c'è alcun nodo nella lista.

- **s1**: Registro dove è salvato l'indirizzo della stringa listInput, questa stringa conterrà i comandi che dovranno essere eseguiti dal programma.

- **s2**: Registro che contiene il numero di comandi disponibili. All’inizio è valorizzato con 30 e viene decrementato ogni volta che viene eseguito un comando. Quando arriva a 0 il programma termina.

- **s3**: Registro che contiene l’indirizzo di memoria dove i nodi della lista circolare verranno salvati. All’inizio vale 0x00100000, indirizzo scelto arbitrariamente. Questo registro viene incrementato via via che vengono aggiunti nodi.

- **s4-s9**: Questi registri vengono inizializzati con i valori ASCII delle lettere maiuscole (A-Z), delle lettere minuscole (a-z) e dei numeri (0-9). Questi valori verranno utilizzati dalla funzione ausiliaria “adjust_char”.

- **s10**: Registro che contiene l’Indirizzo dell’array che sarà usato nelle operazioni di sorting. Come indirizzo ho scelto 0x00, cosicché i valori dell’array non verranno sovrascritti dalla funzione “ADD”.

Parsing dei comandi

Il progetto richiede che il main debba elaborare l'unico input del programma, dichiarato come una variabile string listInput nel campo .data del codice.
Le seguenti sono le richieste per il controllo dell'input:

A. Due comandi ben formati ma non separati da tilde sono considerati mal formattati.
B. Per le operazioni ADD e DEL, si deve avere un solo carattere ASCII tra le parentesi. Il valore del carattere deve essere compreso tra 32 e 125.
C. I caratteri dei comandi devono essere consecutivi.
D. Gli spazi vicino alle ~ sono permessi e il programma li deve tollerare.
E. I comandi devono essere espressi in lettere maiuscole.
F. Ogni comando mal formattato deve essere ignorato.
G. listInput non dovrà contenere più di 30 comandi.

Il parsing dei comandi avviene nella sezione di codice "parse_commands", che descrivo di seguito:

beqz s2, end_main: quando s2 arriva a zero, sono stati eseguiti 30 comandi e il programma passa a end_main.

skip_spaces: Ciclo che carica il carattere della stringa listInput puntato da s1, se esso è uno spazio ritorna nel ciclo, se è il carattere di fine stringa il programma passa a end_main, se è un carattere diverso dallo spazio e dal fine stringa, il codice continua in **process_command.** Così facendo vengono saltati gli eventuali spazi prima del primo comando nella stringa listInput, e gli eventuali spazi dopo ogni tilde.

process_command: Basandosi sul carattere puntato da s1, viene deciso quale funzione di controllo invocare per determinare la validità del comando. Ad esempio, se il primo carattere è "A", invoca check_add. Visto che SSX, SDX e SORT iniziano con la S, controllo anche la seconda lettera per capire quale check eseguire dopo.

check_add, check_print, check_del, ecc.: Ognuna di queste sezioni verifica la corretta formattazione del comando specifico. Per esempio, check_add verifica se il comando è scritto come "ADD(x)" dove x è un carattere valido.
Nel caso specifico di ADD (ma funziona allo stesso modo per gli altri comandi), carico in un registro il codice ASCII di A e controllo che corrisponda alla A del comando in listInput, e così via per ogni lettera e parentesi. Se il carattere non corrisponde vado a

check_next_command. Una volta arrivato al carattere DATA da aggiungere o da cancellare dalla lista, controllo prima che esso sia compreso tra 32 e 125, e che dopo DATA ci sia una parentesi tonda chiusa. Se tutti questi controlli vengono verificati, allora carico in t5 la funzione da eseguire. Non posso saltare subito con jal alla funzione da eseguire, perchè devo prima verificare che ci sia una tilde o il fine stringa dopo il comando. Faccio questo controllo in check_tilde_after_command.

check_tilde_after_command: Inizio ponendo il registro t4 a 0. Questo registro segnala o meno la presenza della tilde dopo il comando. Poi controllo se, una volta identificato un comando correttamente formattato, questo sia seguito da eventuali spazi e una tilde oppure da eventuali spazi e il carattere di fine stringa. Se ciò non si verifica, imposto il registro t4 a 1, considero che il comando non sia ben formattato e procedo a cercare la prossima tilde, ignorando tutto ciò che si trova in mezzo. Si procede poi alla sezione di codice **execute_command.**

execute_command:
In questa sezione di codice viene controllato il registro t4 se è 0 la tilde è stata trovata e viene eseguita la funzione associata al comando facendo, “jalr t5”. Inoltre decremento di 1 il registro s2, ovvero il contatore dei comandi. Dopodiché si passa a **check_next_command**.

check_next_command: Incremento il puntatore s1, e torno a parse_commands.

end_main: Termina l'esecuzione del programma, mettendo in a7 il valore 10.

ADD

La funzione ADD inserisce un elemento nella lista circolare, memorizzando il campo DATA nel nuovo nodo. Se la lista è vuota, il nodo viene inserito come primo nodo. Se la lista ha almeno un nodo, il nuovo nodo diventa la coda.

I nodi vengono memorizzati in questo modo (x è un registro uguale per entrambe le istruzioni e r è un registro qualsiasi):
sw a0, 0(x) —> Salvataggio DATA
sw r, 4(x) —> Salvataggio PAHEAD

Di seguito spiego la funzione in dettaglio:
La funzione ADD prende in input in a0, il campo DATA. Per generare un nuovo indirizzo utilizzo il registro s3, che all'inizio vale 0x00100000, e eseguo la seguente istruzione: "addi s3, s3, 20". In questo modo ogni volta che entro in ADD, viene generato un indirizzo diverso rispetto agli altri nodi presenti nella lista circolare.

Dopodichè memorizzo il valore DATA in a0 nell'indirizzo in s3.
Prima di memorizzare il campo PAHEAD controllo il valore del registro s0, che punta alla testa della lista.
Se s0 è zero, la lista è vuota, e devo eseguire questi comandi:

```
mv s0, s3
sw s0, 4(s3)
```

La testa della lista (s0) viene fatta puntare al nuovo nodo.
Il puntatore all'interno del nuovo nodo viene impostato per puntare a se stesso, indicando che è l'unico nodo nella lista circolare. Infine eseguo jr ra per uscire dalla funzione.

Se s0 è diverso da zero, c'è almeno un elemento nella lista circolare, quindi devo aggiungere il nuovo nodo alla fine della lista circolare.

Quindi con un ciclo scorro la lista circolare, iniziando dalla testa della lista che è puntata da s0. Quando trovo l'ultimo nodo, cioè il nodo che punta alla testa (s0), esco dal ciclo. Il registro t2 viene via via aggiornato nel ciclo, alla fine del ciclo avrà l'indirizzo dell'ultimo nodo.
Eseguo questi comandi:

```
sw s0, 4(s3)
sw s3, 4(t2)
```

Ovvero Imposto il campo PAHEAD del nuovo nodo, ovvero la nuova coda, per farlo puntare alla testa. Poi aggiorno il puntatore del penultimo nodo, salvato in t2, per farlo puntare alla nuova coda. Alla fine esco dalla funzione con jr ra.

DEL

La funzione DEL ha l'obiettivo di eliminare tutti i nodi da una lista circolare che contengono lo stesso valore del campo DATA indicato nel registro a0.

Se la lista è vuota, ovvero s0 è zero, la funzione esegue jr ra e termina.
Altrimenti, copio il puntatore della testa della lista s0 in t4 (spiego in seguito il motivo), ed eseguo **start_loop.**

In **start_loop** carico in un registro il campo DATA del nodo puntato da t4.
Se corrisponde al valore da eliminare, vado a **delete_head** per gestire la cancellazione. Altrimenti vado in **delete_loop**.

In **delete_head**
Se il valore da eliminare si trova nella testa della lista, la funzione sposta il puntatore della testa al nodo successivo. Non vado a cambiare subito s0, ma invece memorizzo la nuova testa in t4, questo perchè se cambiassi subito s0, non saprei quando terminare il ciclo **delete_loop** visto che esso termina quando trova il nodo che punta a s0. Prima di memorizzare la testa in t4, controllo se il nodo da eliminare è sia la testa che la coda, ovvero è l'unico elemento della lista. In questo caso metto s0 = 0 e eseguo jr ra per terminare la funzione. Altrimenti continuo a cercare altri nodi da eliminare ripartendo da **start_loop.** Riparto da **start_loop** e non da **delete_loop** perchè è possibile che il successivo nodo da eliminare sia la nuova testa**.**

In **delete_loop**, scorro la lista cercando il primo nodo con campo DATA uguale a quello da eliminare. Se il nodo da eliminare viene trovato, si passa a **delete_node,** altrimenti si continua in **delete_loop,** il ciclo continua fino a quando si trova la coda ovvero il nodo che punta a s0. Riparto da **delete_loop** perchè sono sicuro che il nodo da eliminare non sia più in testa**.** Una volta finito il ciclo si passa a **update_tail.**

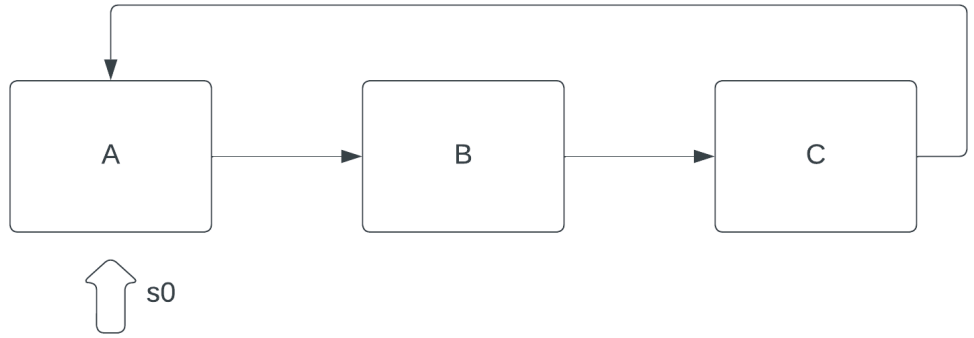
In **delete_node**, collego i nodi adiacenti al nodo da eliminare, cancellando effettivamente il nodo.

In **update_tail**, copio in s0 il valore della nuova testa eventualmente memorizzata in t4.

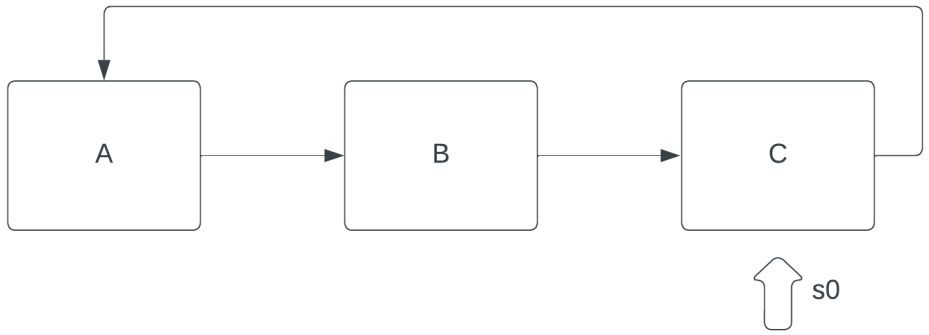
SDX

La funzione SDX ha il compito di "shiftare" di una posizione a destra ogni nodo della lista circolare.

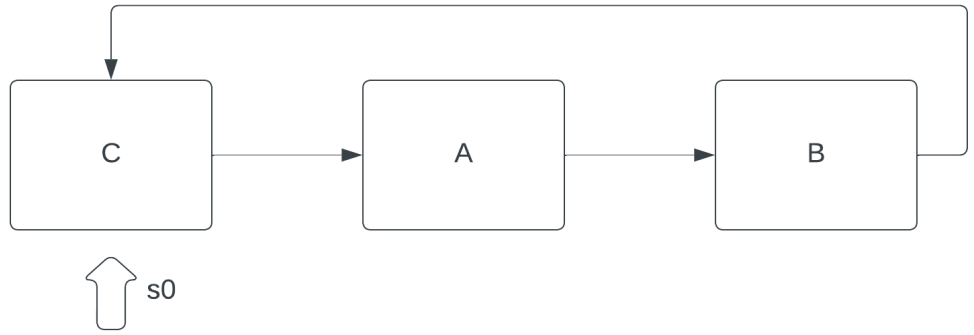
La seguente è la lista circolare prima della chiamata alla funzione SDX.



Una volta che sono nella funzione SDX, controllo che la lista non sia vuota, e scorro la lista fino a quando trovo l'ultimo nodo. Imposto quest'ultimo come nuova testa.



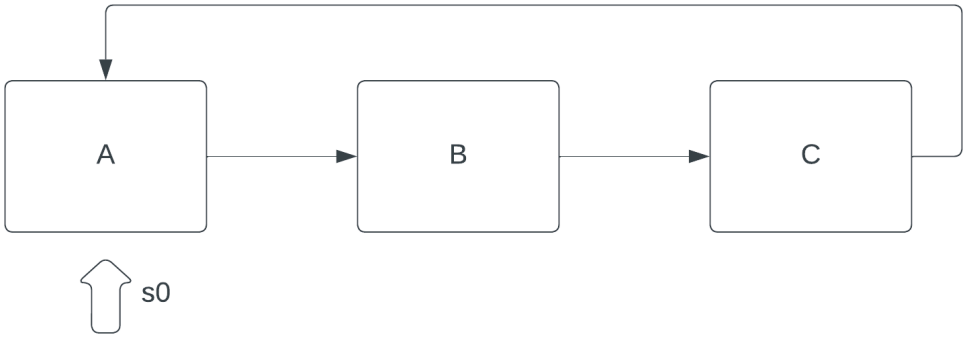
La lista circolare è stata effettivamente "shiftata" a destra.



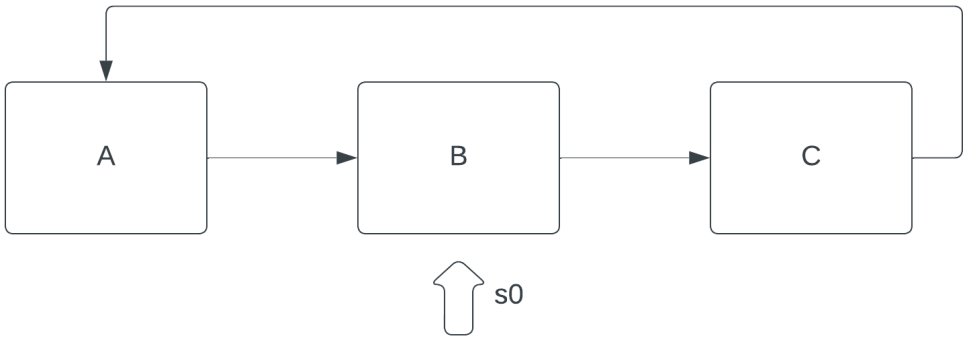
SSX

La funzione SSX ha il compito di "shiftare" di una posizione a sinistra ogni nodo della lista circolare.

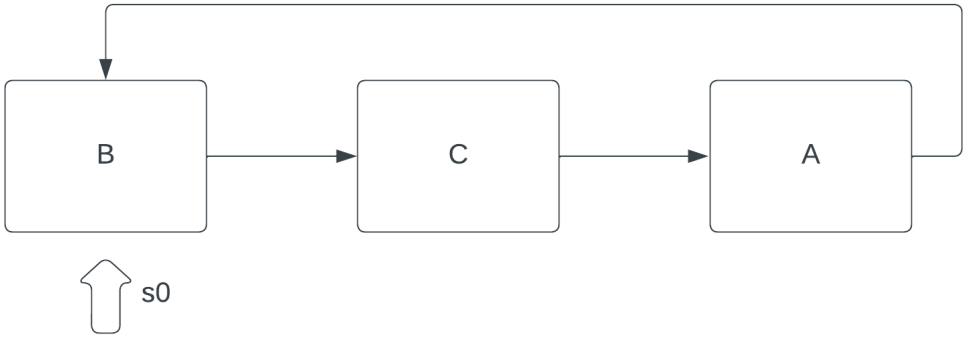
La seguente è la lista circolare prima della chiamata alla funzione SSX.



Una volta che sono nella funzione SSX, controllo che la lista non sia vuota e aggiorno la testa facendola puntare al secondo nodo. Questa è l'istruzione: "lw s0, 4(s0)".



La lista circolare è stata effettivamente "shiftata" a sinistra.

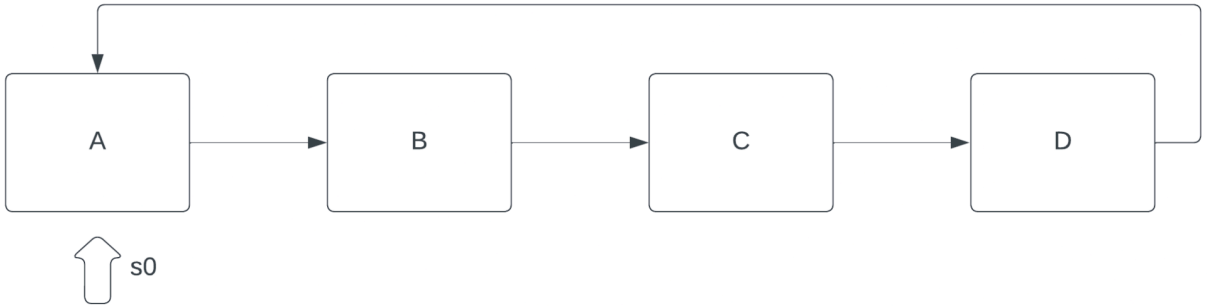


REV

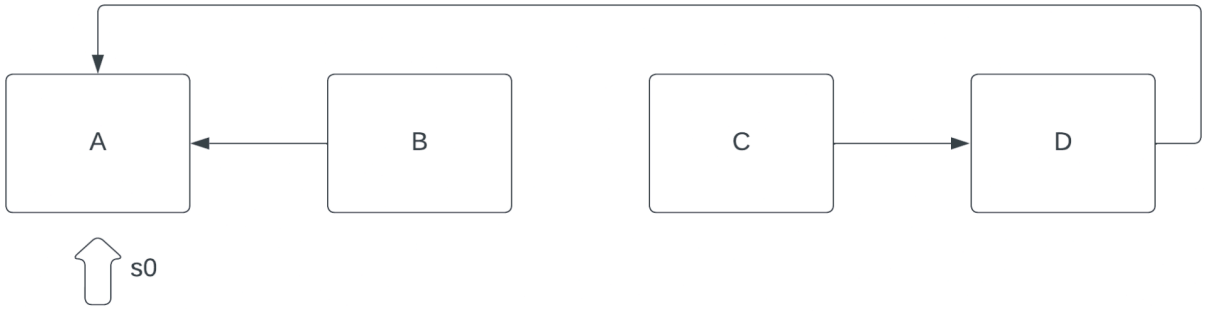
Questa funzionalità inverte la lista circolare. Dopo l'inversione, la testa della lista sarà ciò che era originariamente la coda e viceversa.

Se la lista è vuota o ha un solo elemento, la funzione esegue jr ra e termina.
Altrimenti, la funzione inizia dalla testa della lista ed entra in un ciclo che attraversa i nodi della lista. Mentre itera, la funzione cambia la direzione di ogni puntatore, facendo sì che ogni elemento punti al suo predecessore invece che al suo successivo.
Una volta raggiunta la fine della lista (riconosciuta dal fatto che il prossimo elemento sarebbe di nuovo la testa), la funzione esce dal ciclo, e l'ultimo elemento viene fatto puntare al primo elemento, preservando la natura circolare della lista.
Infine, la testa della lista viene aggiornata per puntare all'originale ultimo elemento, ovvero la nuova testa, completando così l'inversione.

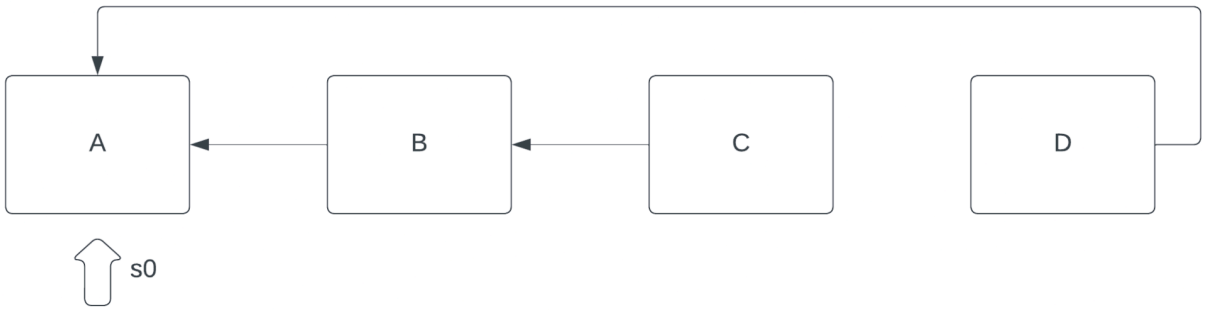
Ecco un esempio con degli schemi di una lista circolare con 4 elementi

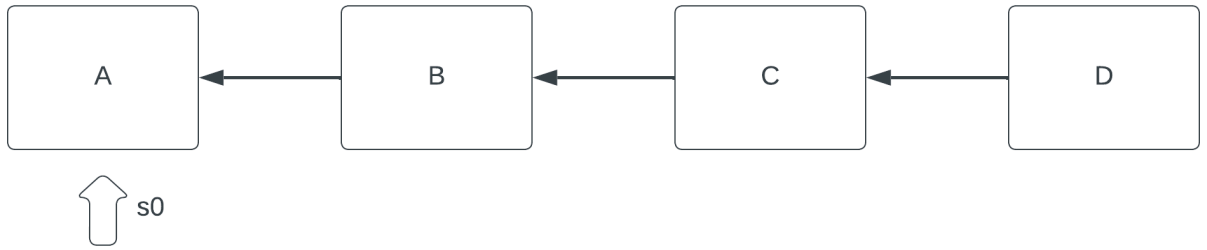


La funzione entra nel ciclo e comincia ad invertire i puntatori.

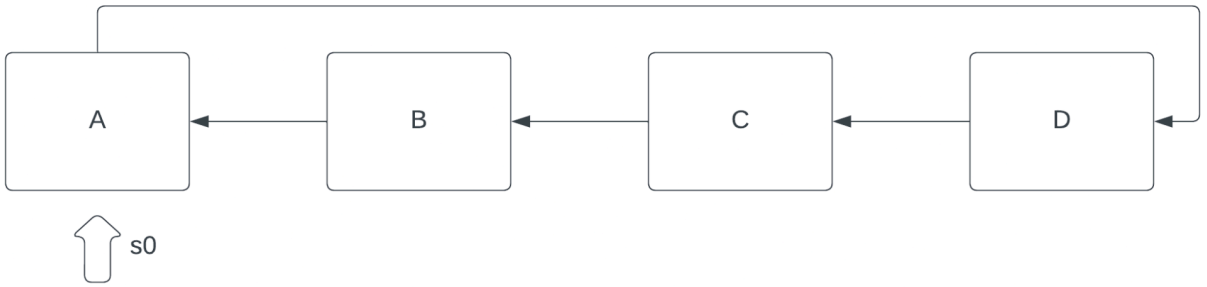


In questo punto il ciclo trova l'ultimo nodo. Prima di terminare, il ciclo invertirà anche il puntatore di questo nodo.

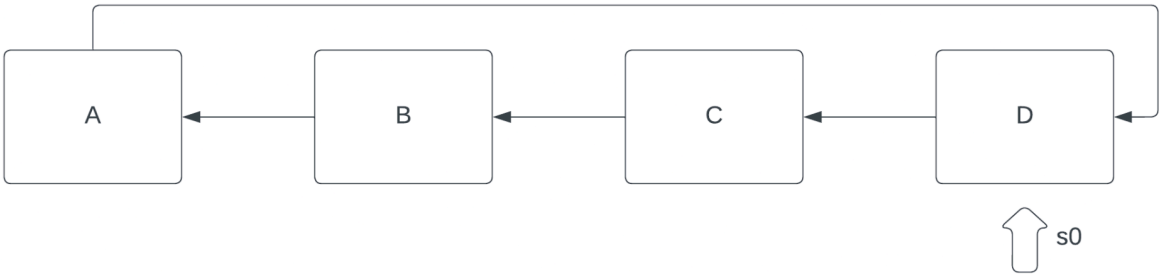




Il ciclo è terminato, e adesso il PAHEAD dell'ultimo nodo (che sarebbe A) viene impostato alla nuova testa.



Infine viene spostato il puntatore della testa.



PRINT

Questa funzionalità stampa il contenuto della lista circolare, in ordine di apparizione dalla testa fino alla coda.
Se la lista è vuota, ovvero s0 è zero, la funzione esegue jr ra e termina.
Se la lista non è vuota, la funzione inizia dalla testa della lista e procede attraversando ogni nodo stampando il campo DATA. Dopo aver stampato tutti i nodi, la funzione stampa “\n” per separare i vari output nella console.
Infine, la funzione esegue jr ra e termina.

SORT

Questa funzionalità ordina gli elementi della lista utilizzando l’algoritmo quicksort.
Se la lista non è vuota, creo un array con i valori DATA di ogni nodo della lista circolare chiamando la funzione list_to_array. Dopodiché, ordino l'array chiamando l’algoritmo quicksort. Infine chiamo array_to_list, che crea una nuova lista circolare, con i valori dell’array che sono stati ordinati. Ho preferito usare un array invece che una lista, perché gli array forniscono un accesso diretto e semplice agli elementi durante l'ordinamento. Prima di chiamare le funzioni ausiliarie salvo il registro ra nello stack, perché verrà sovrascritto dalle chiamate alle funzioni (jal). Salvandolo nello stack, mi assicuro di poterlo recuperare successivamente e tornare correttamente alla funzione chiamante una volta terminata l'esecuzione di SORT. Utilizzo lo stack per salvare ra e non dei registri, perché le funzioni chiamate potrebbero sovrascrivere i registri.

Funzioni ausiliarie

list_to_array

La funzione **list_to_array** copia i valori del campo DATA di tutti i nodi della lista circolare in un array. L’indirizzo dell’array è salvato in s10.
La funzione inizia impostando un puntatore per attraversare la lista e un secondo puntatore per l'indice dell'array.
La funzione entra in un ciclo in cui viene caricato il valore del nodo corrente dalla lista, e esso viene memorizzato nell'array. Si continua questo processo fino a quando si arriva alla coda della lista. Al termine del processo di copiatura, viene inserito un delimitatore all'ultima posizione dell'array per indicare la fine dei dati. La funzione manda in output in a0, il numero di elementi dell’array.

array_to_list

La funzione array_to_list copia i valori dell'array, che indirizzo in s10, in una nuova lista circolare.
La funzione inizia preparando lo spazio nello stack per memorizzare ra e il valore del puntatore all'array. Successivamente, imposto s0 a 0, in modo da "cancellare" la lista circolare precedente. Imposto un registro con il valore di s10.
Entro in un ciclo che estrae il valore DATA dall'array, controllo se il valore è lo zero, che funge da delimitatore per segnalare la fine dell'array. Se si tratta del delimitatore, la copia è completa e il ciclo termina.
Altrimenti, salvo temporaneamente il valore del puntatore all'array, nello stack. Questo perché chiamerò la funzione ADD, che potrebbe sovrascrivere il puntatore. Per aggiungere il valore corrente dell'array nella lista circolare, chiamo la funzione ADD. Una volta aggiunto l'elemento, ripristino il valore del puntatore all'array dallo stack. Il ciclo poi riprende, procedendo con il valore successivo nell'array. Una volta che tutti i valori sono stati aggiunti alla lista circolare, ripristino il valore del registro di ritorno (ra) dallo stack e eseguo jr ra.

adjust_char

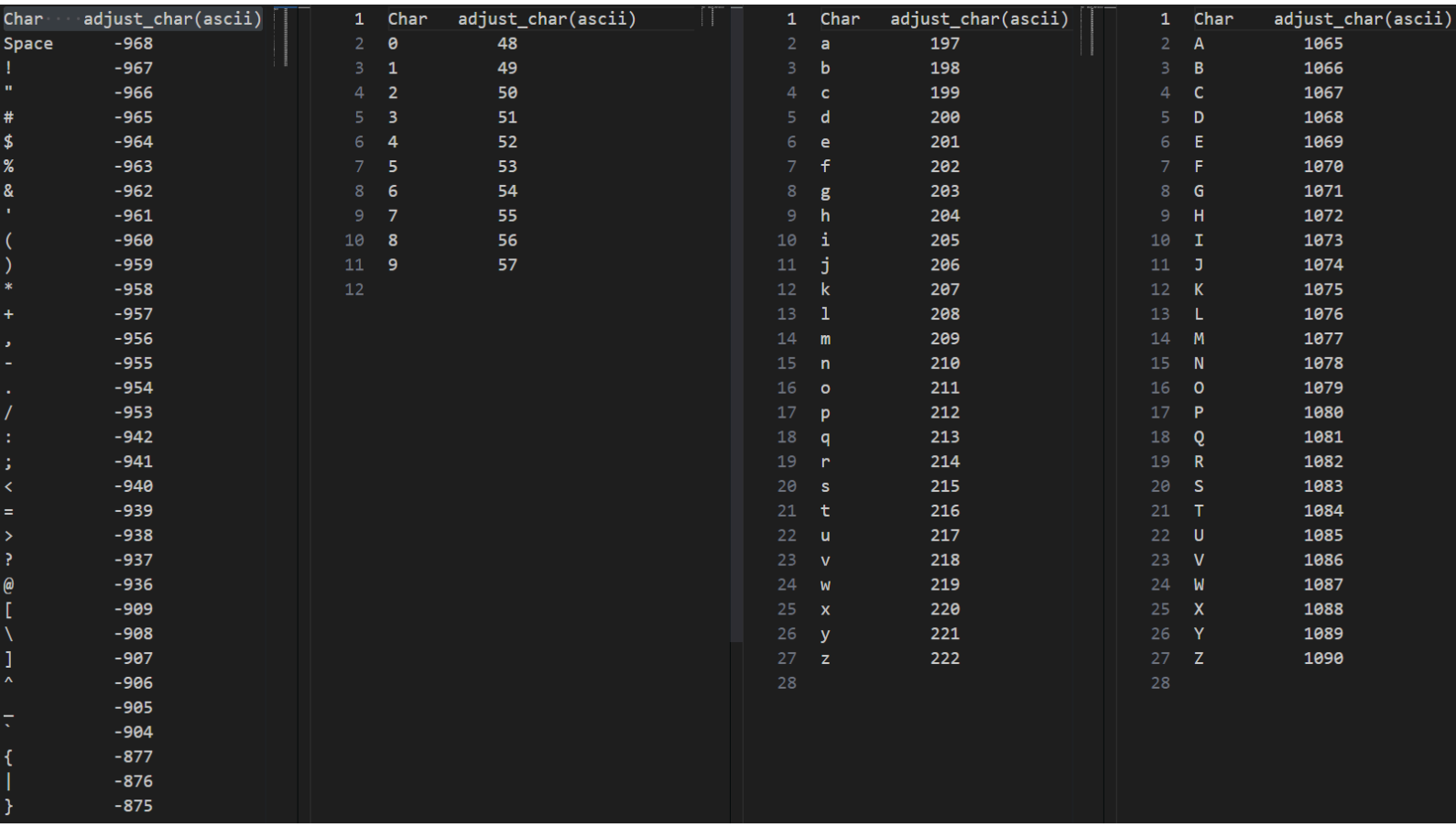
Funzione che prende il carattere in a0 e lo modifica in base all'ordinamento richiesto. L'ordinamento è specificato come maiuscole > minuscole > numeri > caratteri speciali. Inoltre, all'interno di ogni categoria vige l'ordinamento dato dal codice ASCII.
Per esempio, date due lettere maiuscole x e x', x < x' se e solo se ASCII(x) < ASCII(x').

La funzione adjust_char, corrisponde al seguente pseudocodice:

```
function adjusted_char(ascii):
    if 'A' <= ascii <= 'Z':        # lettere maiuscole
        return ascii + 1000
    else if 'a' <= ascii <= 'z':   # lettere minuscole
        return ascii + 100
    else if '0' <= ascii <= '9':   # numeri
        return ascii
    else:                          # caratteri extra
        return ascii - 1000
```

Quindi quel che vado a fare è incrementare, decrementare o lasciare invariata la “priorità” di ciascun carattere ASCII. In questo modo per capire se x < y, dove x e y possono appartenere a qualsiasi categoria di carattere ascii, posso fare adjust_char(x) < adjust_char(y). Nel codice risc v, ho utilizzato i registri da s4 a s9 per capire a quale categoria appartiene il carattere ascii. Così evito di caricare nei registri i valori per il confronto, ogni volta che chiamo la funzione.

Passando tutti i caratteri ammessi dal progetto, nella funzione adjust_char si ottiene:



Char	adjust_char(ascii)
Space	-968
!	-967
"	-966
#	-965
$	-964
%	-963
&	-962
'	-961
(	-960
)	-959
*	-958
+	-957
,	-956
-	-955
.	-954
/	-953
:	-942
;	-941
<	-940
=	-939
>	-938
?	-937
@	-936
[	-909
\	-908
]	-907
^	-906
_	-905
`	-904
{	-877
|	-876
}	-875

Char	adjust_char(ascii)
0	48
1	49
2	50
3	51
4	52
5	53
6	54
7	55
8	56
9	57

Char	adjust_char(ascii)
a	197
b	198
c	199
d	200
e	201
f	202
g	203
h	204
i	205
j	206
k	207
l	208
m	209
n	210
o	211
p	212
q	213
r	214
s	215
t	216
u	217
v	218
w	219
x	220
y	221
z	222

Char	adjust_char(ascii)
A	1065
B	1066
C	1067
D	1068
E	1069
F	1070
G	1071
H	1072
I	1073
J	1074
K	1075
L	1076
M	1077
N	1078
O	1079
P	1080
Q	1081
R	1082
S	1083
T	1084
U	1085
V	1086
W	1087
X	1088
Y	1089
Z	1090

Come si può vedere si ha maiuscole > minuscole > numeri > caratteri speciali.

should_swap

Questa funzione determina se i due caratteri in input in a0 e a1, devono essere scambiati tra loro, in base al criterio di ordinamento descritto precedentemente.

La funzione controlla prime se a0 e a1 sono uguali. In questo caso si imposta a0=0 per segnalare che i valori dei registri non devono essere scambiati e si esce dalla funzione. Se i caratteri non sono uguali viene chiamato adjust_char prima per il carattere a0 e poi per il carattere a1.
Se a0 è maggiore di a1, allora si imposta a0=1 per segnalare che i valori devono essere scambiati, altrimenti a0=0. Infine termino chiamando jr ra.

quicksort

Il quick sort funziona selezionando un "pivot" dall'array e ordinando gli altri elementi attorno ad esso. Gli elementi minori del pivot vengono spostati a sinistra del pivot e gli elementi maggiori vengono spostati a destra. Questo processo viene chiamato "partizione". Una volta che l'array è stato partizionato, il quick sort viene applicato ricorsivamente alle due sotto-sezioni dell'array (ovvero gli elementi a sinistra e a destra del pivot).
Il quick sort nel mio codice viene applicato all’array con indirizzo s10.

quicksort:
Il codice inizialmente controlla se l'indice di inizio è minore dell'indice di fine. Se non lo è, l'algoritmo ha terminato l'ordinamento per quella particolare sezione.
Se l'indice di inizio è minore dell'indice di fine, viene chiamata la funzione partition.
Dopo la partizione, viene fatto un ordinamento ricorsivo delle due metà dell'array.

partition:
Per la partizione come pivot scelgo l'elemento alla fine dell'array e ordino gli elementi attorno ad esso. Inizio dal primo elemento più a sinistra dell'array e mantengo traccia dell'indice "i" degli elementi più piccoli o uguali al pivot. Ogni volta che incontro un elemento più piccolo o uguale al pivot, scambio l'elemento corrente con l'elemento all'indice "i" e incremento l'indice "i". Se, invece, l'elemento corrente è maggiore del pivot, lo ignoro e continuo a scorrere.

Alla fine di questo processo, il pivot verrà posizionato alla sua giusta posizione nell'array, ossia tra gli elementi minori (a sinistra) e quelli maggiori (a destra) di esso. Per il confrontare gli elementi con il pivot utilizzo la funzione should_swap.

Di seguito lo pseudocodice:

```
function quicksort(array, low, high):
    if low < high:
        pi = partition(array, low, high)
        quicksort(array, low, pi - 1)
        quicksort(array, pi + 1, high)


function partition(array, low, high):
    pivot = array[high]
    i = low - 1

    while(low < high):
        if should_swap(array[low],pivot)
            i = i + 1
            (array[i], array[low]) = (array[low], array[i])    # swap
        low = low + 1

    (array[i + 1], array[high]) = (array[high], array[i + 1]) # swap

    return i + 1
```