



Progetto Assembly RISC-V

Relazione Gestione di Liste Circolari

Andrei Jonathan Ghergut

Mat. 7029448

Università degli Studi di Firenze (andrei.ghergut@stud.unifi.it)

29.05.2023

Il progetto di Assembly RISC-V è stato realizzato e testato in Windows 10, utilizzando la versione di Ripes v2.2.6.

Funzionamento generale della soluzione adottata e registri utilizzati

La struttura dell'intero progetto ha le sue fondamenta nelle funzioni di *controlInput* e *checkAnotherOperation*, che analizza la stringa *listInput*, contenente le operazioni fondamentali per la manipolazione della lista concatenata circolare.

Nella sezione *.data* vengono inizializzati tre dati necessari:

- la string *listInput* dove tra gli apici verranno inserite le varie operazioni;
- la word *pHeadLinkedList* contenente l'indirizzo di memoria di partenza per la memorizzazione dei dati inseriti in lista;
- la word *newLine*, per a stampare a capo la stringa che dimostrerà la correttezza del programma.

L'indirizzo di memoria di partenza di dove verranno inseriti i dati nella lista corrisponde a 0x100001F4, distante esattamente 500 byte da 0x10000000, indirizzo di partenza dal quale Ripes inizia a memorizzare i dati; perciò è stato supposto che un'operazione possa occupare al più 16 byte di memoria dati che sono consentiti gli spazi, e dato un massimo di 30 operazioni accettabili dal programma.

Tutto ciò è stato idealizzato per essere sicuri che le operazioni di aggiunta di un elemento nella lista non si vada a sovrapporre con dati esistenti.

Nella sezione *.text* vengono caricati nei registri *s0* e *a3* l'indirizzo di partenza di dove viene memorizzata la *listInput* e l'indirizzo di dove vengono iniziati a memorizzare i dati in lista.

Infine vengono caricati i registri *a4*, *a7* e *s7* rispettivamente con i valori 0, 0 e 30, che rappresentano:

- il numero di elementi presenti in lista nel momento corrente (*a4*);
- il numero di operazioni eseguite (*a7*);
- il numero massimo di operazioni che si possono eseguire (*s7*).

1.1 Control Input

Il funzionamento di *controlInput* si basa su un indice di scorrimento, che controlla il primo carattere di un'operazione. Ogni volta che viene chiamata in causa, esegue un incremento delle operazioni controllate da inizio programma (valore contenuto in *a7*).

Sono cinque i diversi caratteri iniziali permessi alle operazioni:

- ASCII 65, che rappresenta la lettera maiuscola A;
- ASCII 68, che rappresenta la lettera maiuscola D;
- ASCII 80, che rappresenta la lettera maiuscola P;

```
21 controlInput:
22     addi a7, a7, 1
23     lb t0, 0(a0)
24
25     li t1, 65      # ASCII of 'A'
26     beq t0, t1, inputA
27
28     li t1, 68      # ASCII of 'D'
29     beq t0, t1, inputD
30
31     li t1, 80      # ASCII of 'P'
32     beq t0, t1, inputP
33
34     li t1, 82      # ASCII of 'R'
35     beq t0, t1, inputR
36
37     li t1, 83      # ASCII of 'S'
38     beq t0, t1, inputS
39
```

- ASCII 82, che rappresenta la lettera maiuscola R;
- ASCII 83, che rappresenta la lettera maiuscola S.

Se il carattere in esame corrisponde ad uno di questi cinque codici ASCII, verrà chiamata la funzione specifica che controllerà i caratteri successivi a quello appena analizzato (inputA, inputD, inputP, inputR, inputS).

Se invece il carattere è diverso da quelle elencate allora non è valido e si controllerà, se esiste, un'altra operazione con la funzione *checkAnotherOperation*.

1.2 Check Another Operation

Il funzionamento di *checkAnotherOperation* si basa sul controllo dell'esistenza di un nuovo comando da esaminare nella *listInput*.

La prima operazione che viene eseguita è quella di controllare se è possibile ancora effettuare un comando, perché come richiesto dalla consegna, non dovranno essere eseguiti più di 30 comandi.

Dopodiché si effettua il controllo fino a quando non si va incontro ad un codice ASCII pari a 126 e a quel punto viene effettuato un salto a *controlInput* al primo carattere che non sia uno spazio, dato che sono ammessi dal programma.

Se invece nel controllo andiamo incontro ad un codice ASCII pari a zero, vuol dire aver raggiunto il fine stringa di *listInput*, e perciò il programma finirà.

Figura 1.0 Descrizione ad alto livello della procedura *checkAnotherOperation*.

```

129 private static void checkAnotherOperation(short indexListInput) {
130
131     if (operationChecked == 30) {
132         endAll();
133     }
134
135     short codASCII = listInput[indexListInput];
136
137     if(codASCII == 126) {
138
139         indexListInput ++;
140         codASCII = listInput[indexListInput];
141
142         while(codASCII == 32) {
143             indexListInput ++;
144             codASCII = listInput[indexListInput];
145         }
146
147         controlInput(indexListInput);
148     }
149
150     if(codASCII == 0) {
151         endAll();
152     }
153
154     indexListInput ++;
155     checkAnotherOperation(indexListInput);
156 }

```

1.3 Control Operation ADD

Nel caso in cui il primo carattere corrispondesse al codice ASCII 65 (lettera maiuscola 'A') viene effettuato il salto alla funzione inputA per controllare la corretta formattazione in modo da eseguire una ADD.

Nel caso alla 'A' seguissero in ordine i codici ASCII 68, 68 e 40 (corrispondenti a 'D' e '('), verrebbe chiamata la funzione *checkIfValidChar* (spiegata in dettaglio in seguito) per verificare che il carattere da inserire sia un codice ASCII accettabile (da 32 a 125 compresi).

Se il carattere fosse valido, viene controllato che subito dopo ci sia il codice ASCII 41 (corrispondente a ')').

Arrivati a questo punto il comando risulta ben formattato ma non basta per chiamare la add; è obbligatorio controllare che il carattere successivo al comando sia una tilde (codice ASCII 126), oppure che la tilde sia preceduti esclusivamente dagli spazi (codice ASCII 32), ammessi dal programma.

Tale controllo viene effettuato con la chiamata a *verifyCorrectness*, anche questa spiegata in seguito.

Se vengono superati tutti i controlli, allora si passa alla chiamata dell'operazione add tramite un jump-and-link, che quando ritornerà effettuerà un salto alla funzione *checkAnotherOperation* .

Al contrario, se il programma non supera uno dei controlli sopra elencati, non verrà chiamata la add, ma subito si eseguirà un salto a *checkAnotherOperation*.

1.4 Control Operation DEL

Nel caso in cui il primo carattere corrispondesse al codice ASCII 68 (lettera maiuscola 'D') viene effettuato il salto alla funzione inputD per controllare la corretta formattazione in modo da eseguire una DEL.

I passaggi di controllo da eseguire per poter effettuare una DEL sono molto simili al precedente; infatti nel caso alla 'D' seguissero in ordine i codici ASCII 69, 76 e 40 (corrispondenti a 'E', 'L' e '('), verrebbe chiamata la funzione *checkIfValidChar* per verificare che il carattere da eliminare sia un codice ASCII accettabile (da 32 a 125 compresi).

Anche in questo caso, se il carattere fosse valido, viene controllato che subito dopo ci sia il codice ASCII 41 (corrispondente a ')').

Arrivati a questo punto si effettua la verifica dei caratteri successivi al comando ben formattato con la chiamata a *verifyCorrectness*.

Come nel controllo precedente, se non viene superato uno dei controlli si eseguirà un salto a *checkAnotherOperation*.

Se invece vengono superati tutti i controlli, si passa alla chiamata dell'operazione tramite un jump-and-link, che quando ritornerà effettuerà un salto alla funzione *checkAnotherOperation* per verificare l'esistenza di un altro comando da eseguire.

Il ragionamento alla base degli altri quattro comandi di controllo segue il diagramma di flusso della Figura 1.1.

1.5 Control Operation PRINT

Nel caso in cui il primo carattere corrispondesse al codice ASCII 80 (lettera maiuscola 'P') viene effettuato il salto alla funzione inputP per controllare la corretta formattazione in modo da eseguire PRINT.

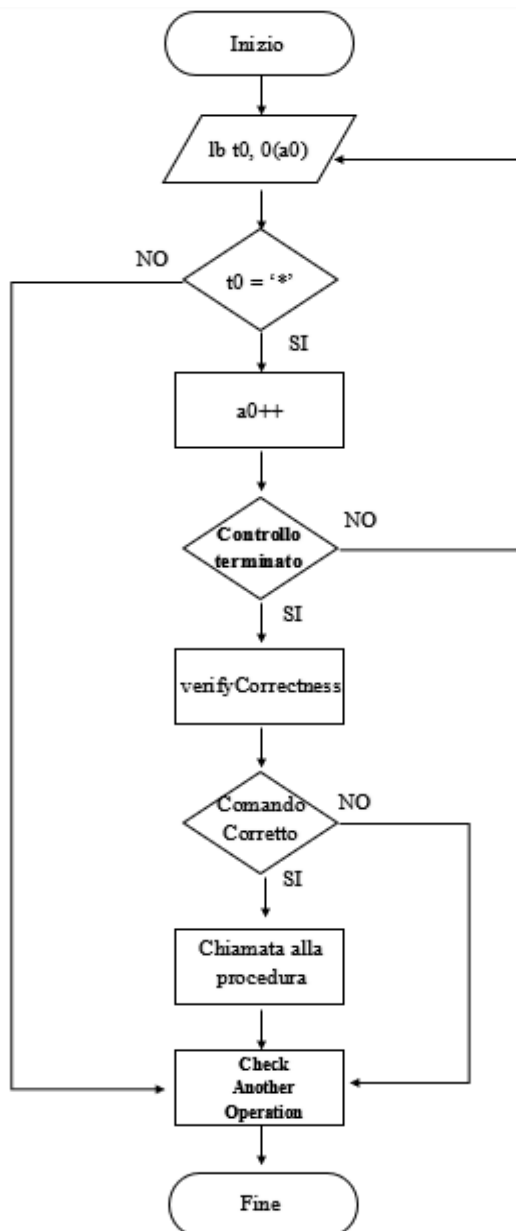
Si controllano i codici ASCII successivi; se sono rispettivamente 82, 73, 78, 84, allora ci sarà la chiamata alla funzione *verifyCorrectness* per la effettiva correttezza nella formattazione.

In caso positivo, viene salvato nello stack pointer i valori dei registri a0 e a7, contenenti rispettivamente l'indice di scorrimento di *listInput* e il contatore dei comandi effettuati dal programma fino a quel momento, in quanto nell'eseguire la funzione di stampa, si fa uso di questi due registri, dove quindi viene alterata l'informazione contenuta in essi al momento della chiamata.

Dopo aver salvato nello stack, viene chiamata la funzione *print* che stampa tutti i caratteri presenti nella lista in ordine di inserimento, e una volta terminato, viene ripristinato il valore dei registri ed effettuato il salto per il controllo al prossimo comando.

Invece, in caso negativo ad uno dei controlli, si effettua il salto direttamente al prossimo comando.

Figura 1.1 Diagramma di flusso di controllo di PRINT, REV, SORT, SDX, SSX.



1.6 Control Operation REV

Nel caso in cui il primo carattere corrispondesse al codice ASCII 82 (lettera maiuscola 'R') viene effettuato il salto alla funzione *inputR* per controllare la corretta formattazione in modo da eseguire una REV.

Si controllano i codici ASCII successivi; se sono rispettivamente 69 e 86, allora ci sarà la chiamata alla funzione *verifyCorrectness* per la effettiva correttezza nella formattazione.

In caso positivo, allora viene chiamata la funzione *rev* che invertirà tutti i puntatori, e una volta finito il suo compito, verrà effettuato il salto al prossimo comando.

In caso negativo invece si effettua il salto direttamente al prossimo comando.

1.7 Control Operation SORT, SDX, SSX

Nel caso in cui il primo carattere corrispondesse al codice ASCII 80 (lettera maiuscola 'S') viene effettuato il salto alla funzione *inputS* per controllare quali delle operazioni tra SORT, SDX e SSX dovrà essere eseguita.

Come prima cosa viene controllato il secondo carattere del comando; se corrisponde all'ASCII 79 (cioè la lettera maiuscola 'O') allora continua verificando che i successivi corrispondano in ordine al codice ASCII 82 (lettera maiuscola 'R') e 84 (lettera maiuscola 'T').

In caso di questa successione, come nei precedenti casi, viene verificata la corretta formattazione del comando prima di effettuare la chiamata alla procedura *sort*; infine una volta

ritornata sarà effettuato un salto a *checkAnotherOperation*.

Se invece il secondo carattere non corrisponde all' ASCII 79, viene effettuato un salto a *inputS_*, per verificare se si tratta dell' ASCII 68 (lettera maiuscola 'D') o 83 (lettera maiuscola 'S').

Se così fosse, in entrambi i casi si effettua il salto alla funzione *inputS_X* che controlla il valore del terzo carattere; se corrisponde all'ASCII 88 (lettera maiuscola 'X'), allora verrà effettuata prima la chiamata a *verifyCorrectness* e poi alla rispettiva funzione, *sdx* o *ssx*, per finire con il salto a *checkAnotherOperation*.

Ogni caso di errore nei passaggi di controllo corrisponderà al salto a *checkAnotherOperation*.

Spiegazione dettagliata delle operazioni fondamentali per una lista concatenata circolare

Ogni operazione fondamentale per una lista concatenata circolare è stata eseguita come una procedura separata chiamata con una *jal*, come da richiesta nella consegna.

La chiamata con *jump-and.link* è stata eseguita, come precedentemente spiegato, nella funzione di input corrispondente; perciò quando avviene la chiamata, è garantita la corretta formattazione del comando.

2.1 ADD

La procedura ADD ha il compito di aggiungere alla lista concatenata un nuovo elemento. Questa operazione avviene salvando per prima il return address nello stack, in quanto durante la sua esecuzione è possibile che venga fatta una chiamata ad un'altra procedura, cioè *findPHeadLinkedList*, spiegata in dettaglio più avanti.

Come prima cosa si controlla il numero di elementi presenti nella lista:

- se è uguale a 0, vuol dire che si può inserire l'elemento nell'indirizzo di memoria a cui punta il puntatore della *LinkedList*, facendo puntare il proprio PAHEAD a se stesso;
- se è diverso da 0, vuol dire che dovrà essere trovata un'area di memoria vuota da poter dedicare al nuovo elemento.

A livello di implementazione, è stato deciso di separare tutti gli elementi della lista di 10 byte, per evitare che le informazioni si sovrappongono.

Per trovare un'area di memoria vuota, viene chiamata la funzione *findPHeadLinkedList*.

Come si può intuire dal nome, *findPHeadLinkedList* cerca l'elemento col puntatore alla testa della lista, cioè l'ultimo elemento; una volta trovato, si scorre di 10 byte in avanti fintanto non venga trovato uno spazio di memoria libero.

Questo viene reso necessario perché se si effettuasse una ADD subito dopo una SDX (spiegata dettagliatamente in seguito), questo si andrebbe a sovrascrivere su dati già esistenti.

A questo punto viene cambiato il puntatore del precedente facendolo puntare all'indirizzo di memoria di quello da inserire; si inserisce l'elemento, e fatto puntatore alla testa della lista.

2.2 DEL

La procedura DEL ha il compito di eliminare dalla lista concatenata l'elemento desiderato, se esso esiste in lista. Durante l'implementazione del codice, è risultato utile l'utilizzo dello stack per salvare i seguenti dati:

1. return address, perché è possibile che venga eseguita un'altra *jal*;
2. l'indirizzo di memoria dell'elemento precedente a quello da eliminare, così da poter modificare i puntatori.

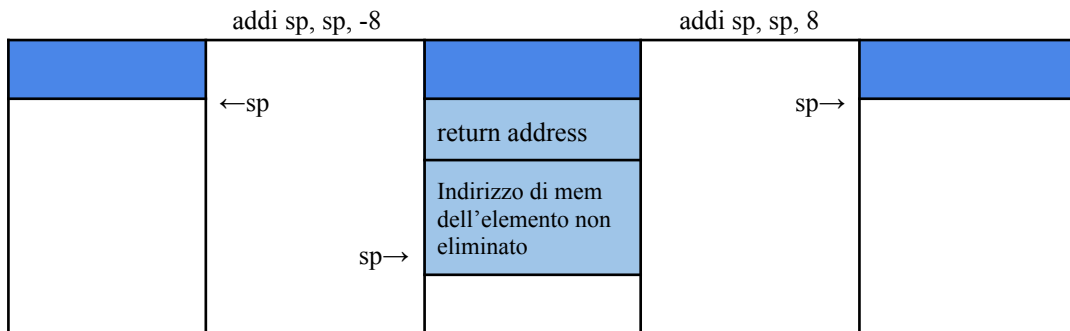


Figura 2.0 Utilizzo stack pointer nella DEL.

La prima operazione che viene eseguita è il controllo di quanti elementi sono presenti in lista:

- se la lista fosse vuota, si salta subito all'end ripristinando lo stack;
- se nella lista fosse presente un solo elemento, si controlla se questo sia quello da eliminare o meno.

Se così non fosse invece viene usato il registro t4 per scorrere la lista a partire dal pHeadLinkedList, cioè il puntatore alla testa della lista.

In t1 viene caricato il byte contenente l'elemento da confrontare, e in t2 invece viene caricata la word contenente il puntatore all'elemento successivo.

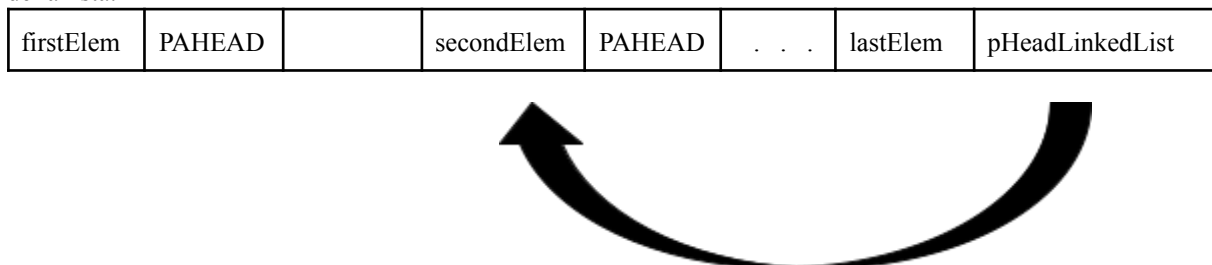
A questo punto vengono confrontati i valori ASCII di t1 e a2, rispettivamente l'elemento che si sta esaminando in lista e l'elemento da eliminare. Se sono uguali si salta a *delete* e si procede con l'eliminazione, altrimenti viene salvato nello stack l'indirizzo di memoria dell'elemento appena esaminato, e si continua a scorrere nella lista.

Il salvataggio della posizione in memoria di un elemento serve a poter cambiare il puntatore al prossimo nel caso il successivo fosse da eliminare.

In *delete* avviene la vera e propria eliminazione dell'elemento dalla lista. In questo caso però andremmo in contro a due situazioni diverse: il caso in cui l'elemento da eliminare fosse il primo della lista e il caso in cui non lo fosse.

Nel caso l'elemento da eliminare fosse il primo della lista, allora avviene la chiamata a *findPHeadLinkedList*, per trovare l'ultimo elemento e il suo puntatore alla testa, e cambiarlo, facendolo puntare all'indirizzo di memoria a cui punta l'elemento da eliminare.

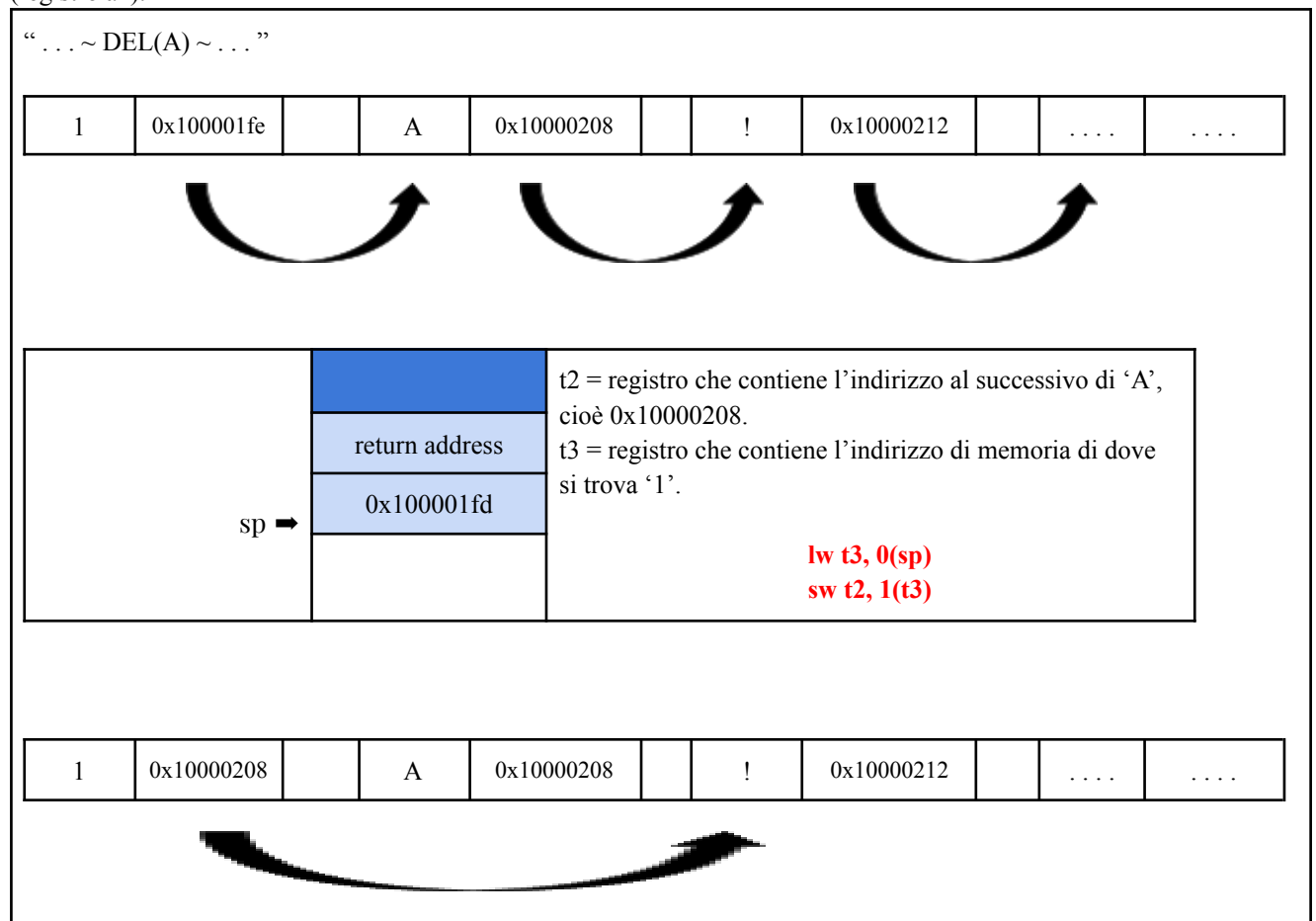
Figura 2.1 Rappresentazione cambio del puntatore al prossimo elemento nel caso in cui si eliminasse il primo della lista.



Se invece si tratta di un altro elemento qualsiasi che non sia il primo, si effettua un pop dallo stack caricando il dato nel registro t3 (vedi Figura 2.0). Il registro t3 conterrà l'indirizzo in memoria del precedente a quello da eliminare.

A questo punto verrà cambiato il puntatore precedente, facendolo puntare all'indirizzo di memoria a cui punta quello da eliminare, come spiegato in figura:

Figura 2.2 Funzionamento di una DEL nel caso l'elemento da eliminare non fosse il primo della lista
Ad ogni delete naturalmente viene decrementato di uno il registro contenente il numero elementi presenti in lista (registro a4).



2.3 PRINT

La procedura PRINT risulta molto semplice. Come prima cosa viene fatto un controllo sul numero di elementi presenti in lista, se è pari a zero viene eseguito un salto ad endPrint, stampando solo una newline, cioè “\n” e si ritorna al chiamante.

In caso contrario vengono usati i registri:

- t0 per scorrere la lista;
- a0 che viene caricato con il byte contenente l'elemento da stampare;
- a7 che viene caricato con il valore immediato corrispondente alla stampa dei caratteri, cioè 11.

Viene quindi eseguito un loop stampando tutti gli elementi della lista finché non si trova l'ultimo elemento; a quel punto viene eseguito un salto a endPrint.

2.4 SORT

La procedura SORT è stata effettuata usando un'implementazione che può essere suddivisa in quattro passi:

1. dedicare un'area di memoria apposita dove svolgere le operazioni di ordinamento della lista;
2. scorrere la lista e settare delle priorità ad ogni elemento per poi inserirli (il dato e la sua priorità) nell'area di memoria dedicata;
3. ordinare gli elementi in base alla priorità assegnata;

4. inserire nella lista gli elementi ordinati.

Passo 1

Il primo passo da effettuare è quello di dedicare uno spazio di memoria da dove partire per inserire gli elementi e le loro priorità; dato che deve essere distante dalla lista, è stato scelto l'indirizzo 0x10000bb8.

Passo 2

Il secondo passo consiste nell'assegnare una priorità a tutti gli elementi della lista, ed inserirli nell'area di memoria dedicata. Per fare questo si scorre tutta la lista a partire dal puntatore alla testa (0x100001F4) e si inserisce l'elemento a partire da 0x10000bb8, e il byte successivo contiene la sua priorità. In questo modo gli elementi, a partire da 0x10000bb8, sono distanti due byte.

Figura 2.3 Legenda ordinamento SORT e la loro suddivisione nella tabella ASCII.

Priorità	Categoria
1	Simboli
2	Numeri
3	Minuscole
4	Maiuscole

Binary	Oct	Dec	Hex	Glyph	Binary	Oct	Dec	Hex	Glyph	Binary	Oct	Dec	Hex	Glyph
010 0000	040	32	20		100 0000	100	64	40	@	110 0000	140	96	60	
010 0001	041	33	21	!	100 0001	101	65	41	A	110 0001	141	97	61	a
010 0010	042	34	22	"	100 0010	102	66	42	B	110 0010	142	98	62	b
010 0011	043	35	23	#	100 0011	103	67	43	C	110 0011	143	99	63	c
010 0100	044	36	24	\$	100 0100	104	68	44	D	110 0100	144	100	64	d
010 0101	045	37	25	%	100 0101	105	69	45	E	110 0101	145	101	65	e
010 0110	046	38	26	&	100 0110	106	70	46	F	110 0110	146	102	66	f
010 0111	047	39	27	'	100 0111	107	71	47	G	110 0111	147	103	67	g
010 1000	050	40	28	(100 1000	110	72	48	H	110 1000	150	104	68	h
010 1001	051	41	29)	100 1001	111	73	49	I	110 1001	151	105	69	i
010 1010	052	42	2A	*	100 1010	112	74	4A	J	110 1010	152	106	6A	j
010 1011	053	43	2B	+	100 1011	113	75	4B	K	110 1011	153	107	6B	k
010 1100	054	44	2C	,	100 1100	114	76	4C	L	110 1100	154	108	6C	l
010 1101	055	45	2D	-	100 1101	115	77	4D	M	110 1101	155	109	6D	m
010 1110	056	46	2E	.	100 1110	116	78	4E	N	110 1110	156	110	6E	n
010 1111	057	47	2F	/	100 1111	117	79	4F	O	110 1111	157	111	6F	o
011 0000	060	48	30	0	101 0000	120	80	50	P	111 0000	160	112	70	p
011 0001	061	49	31	1	101 0001	121	81	51	Q	111 0001	161	113	71	q
011 0010	062	50	32	2	101 0010	122	82	52	R	111 0010	162	114	72	r
011 0011	063	51	33	3	101 0011	123	83	53	S	111 0011	163	115	73	s
011 0100	064	52	34	4	101 0100	124	84	54	T	111 0100	164	116	74	t
011 0101	065	53	35	5	101 0101	125	85	55	U	111 0101	165	117	75	u
011 0110	066	54	36	6	101 0110	126	86	56	V	111 0110	166	118	76	v
011 0111	067	55	37	7	101 0111	127	87	57	W	111 0111	167	119	77	w
011 1000	070	56	38	8	101 1000	130	88	58	X	111 1000	170	120	78	x
011 1001	071	57	39	9	101 1001	131	89	59	Y	111 1001	171	121	79	y
011 1010	072	58	3A	:	101 1010	132	90	5A	Z	111 1010	172	122	7A	z
011 1011	073	59	3B	;	101 1011	133	91	5B	[111 1011	173	123	7B	{
011 1100	074	60	3C	<	101 1100	134	92	5C	\	111 1100	174	124	7C	
011 1101	075	61	3D	=	101 1101	135	93	5D]	111 1101	175	125	7D	}
011 1110	076	62	3E	>	101 1110	136	94	5E	^					
011 1111	077	63	3F	?	101 1111	137	95	5F	_	111 1110	176	126	7E	~

Passo 3

Il terzo passo consiste in un ciclo per confrontare gli elementi partendo dal primo. Così facendo ad ogni iterazione verrà trovato l'elemento con priorità inferiore. Nell'operazione di confronto ci potranno essere tre casi (viene indicato con 'a' l'elemento in causa, con 'b' l'elemento con il quale farò il confronto, pA e pB le priorità dei due elementi) :

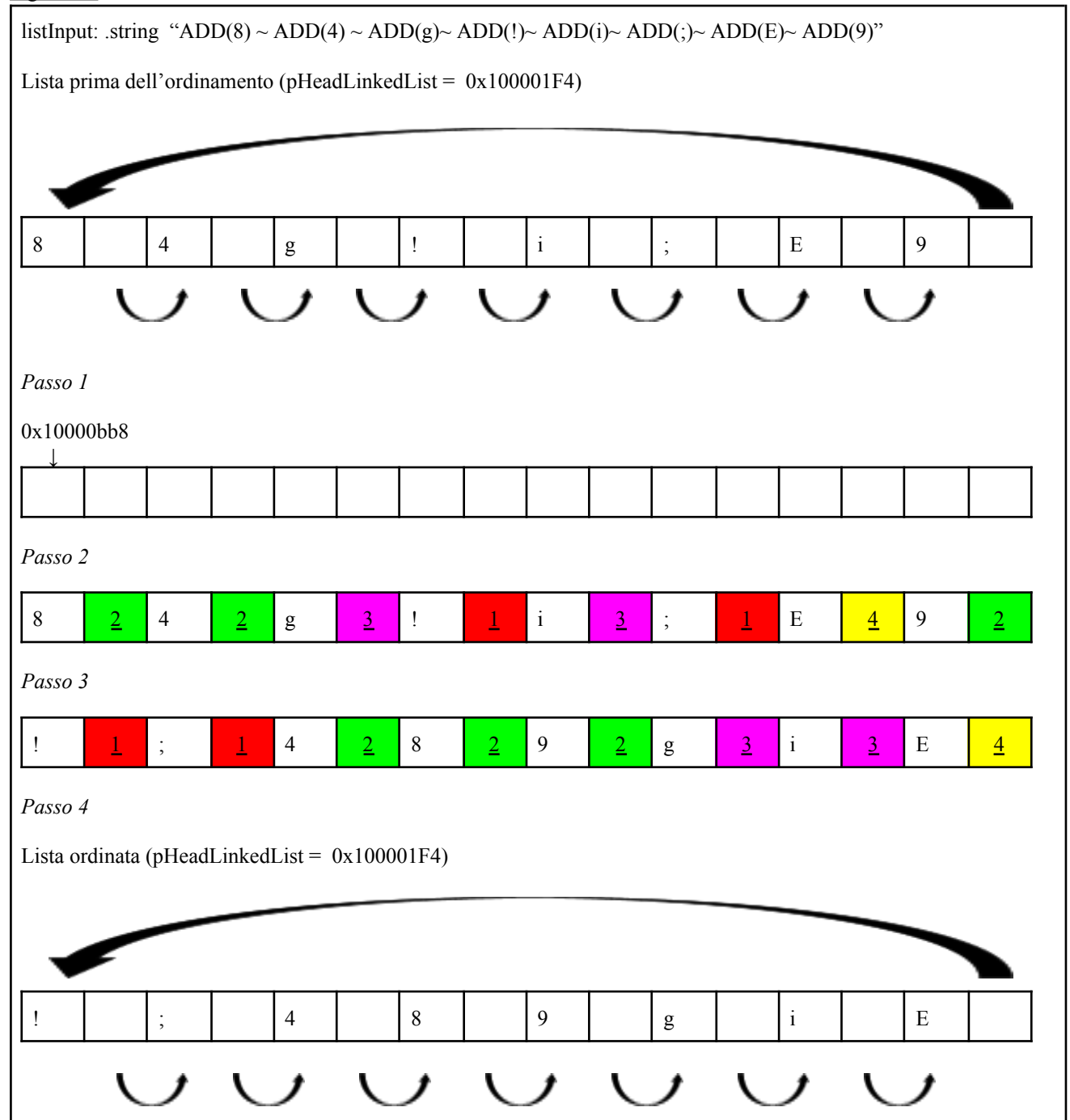
- se $pA < pB$, controllo l'elemento successivo;
- se $pA = pB$, si controlla quale dei due ha il codice ASCII inferiore;
- se $pA > pB$, si scambia a con b, e pA con pB.

A fine ciclo tutti gli elementi saranno in ordine crescente.

Passo 4

Il quarto passo consiste nell'inserire tutti gli elementi ordinati nella lista. Nella Figura 2.4 è raffigurato un esempio del funzionamento del SORT.

Figura 2.4 Funzionamento SORT



2.5 SDX

La procedura di SDX viene eseguita con l'aiuto dello stack, utile per salvare il return address, perché viene fatta una chiamata alla funzione findPHeadLinkedList.

Il funzionamento dello shift a destra si basa sul cambio del puntatore alla testa della lista; infatti pHeadLinkedList dovrà puntare all'elemento che si trova in ultima posizione della lista, che risulterà poi essere il primo una volta cambiato il puntatore.

2.6 SSX

La procedura di SSX viene eseguita semplicemente cambiando il puntatore alla testa. Per eseguire lo shift a sinistra basta quindi caricare in un registro t0 il PAHEAD del primo elemento della lista, e caricare in a3, cioè il registro contenente il pHeadLinkedList il valore di t0.

2.7 REV

La procedura di REV viene eseguita usufruendo dello stack. E' nello stack infatti che verranno salvati gli elementi della lista per poi investirli nella lista.

Per capire al meglio la procedura suddivido la soluzione in tre passi:

- salvare il valore iniziale dello stack, e poi decrementare quest'ultimo tanto quanto il numero di elementi nella lista;
- inserire gli elementi nello stack in ordine di apparizione nella lista;
- riprendere gli elementi dallo stack a partire dal fondo e inserirli nella lista

Passo 1

Il primo passo consiste nel prendere il valore iniziale dello stack pointer e salvarlo nel registro t0.

Il registro sp viene poi decrementato tanto quanto il numero di elementi presenti in lista (nel programma il dato è salvato in a4, perciò verrà eseguita la seguente operazione : sub sp, sp, a4).

Passo 2

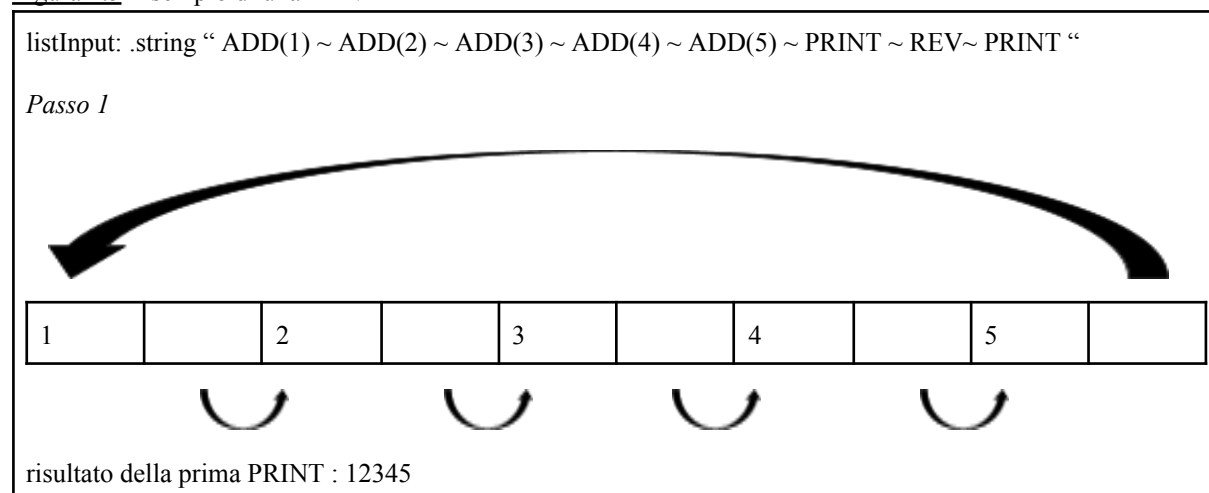
Il secondo passo da eseguire è quello di inserire nello stack gli elementi della lista, partendo dal primo fino all'ultimo elemento. Proprio per questo motivo viene salvato il valore di sp nel registro t0 prima del decremento di sp, perché nello stack il primo dato immerso deve essere anche l'ultimo da prelevare.

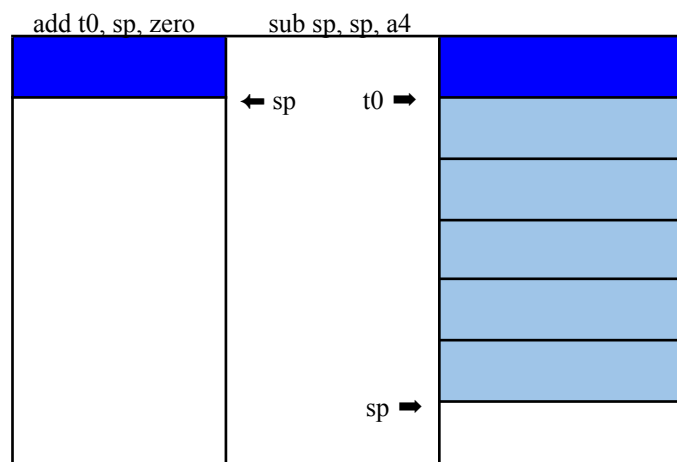
Passo 3

Il terzo passo è quello di riprendere i dati inseriti nello stack e toglierli uno ad uno, inserendoli di volta in volta nella lista, in modo da invertire l'informazione contenuta dagli elementi prima della chiamata a REV.

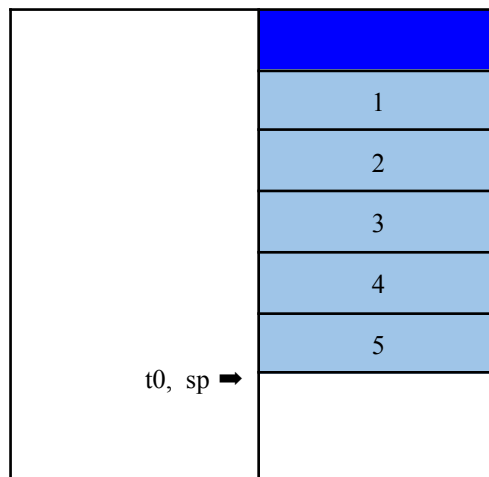
La [Figura 2.5](#) rappresenta un semplice esempio dei passi eseguiti dal programma per avere un'inversione dell'informazione contenuta dagli elementi della lista .

Figura 2.5 Esempio di una REV

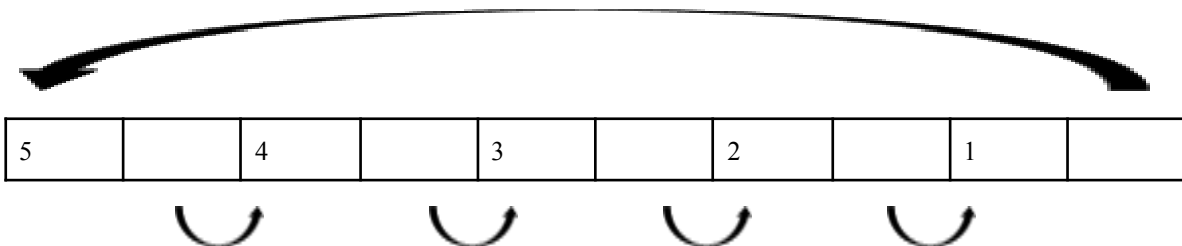




Passo 2



Passo 3



risultato della seconda PRINT : 54321

Viene poi ripristinato lo stack pointer con l'istruzione : add sp, sp, a4

Spiegazione dettagliata di operazioni di supporto per una lista concatenata circolare

Le operazioni di supporto sono fondamentali per il corretto funzionamento del programma.

Ogni operazione di supporto per una lista concatenata circolare è stata eseguita come una procedura separata chiamata con una jal.

3.1 Check If Valid Char

La funzione *checkIfValidChar* è una procedura che viene chiamata nei *controlInput* delle operazioni di ADD e DEL.

Il suo compito è quello di controllare se il codiceASCII contenuto tra le parentesi è valido o meno. Un codice ASCII è ritenuto accettabile se è tra 32 a 125 compresi.

Nell'esecuzione della procedura, per il passaggio di parametri di input e output, vengono utilizzati i registri 'a', in particolare registro a2 per il passaggio input e il registro a1 per l'output.

Se l'output sarà 1, vuol dire che il carattere non è valido, se invece sarà 2 il carattere è valido.

La [Figura 3.0](#) descrive la funzione *checkIfValidChar* in alto livello.

[Figura 3.0](#) Descrizione ad alto livello della procedura *checkIfValidChar*

```
84 private static byte checkIfValidChar(short x) {  
85  
86     if (x < 32 || x > 125) {  
87         return 1; // INVALID  
88     }  
89  
90     else return 2; // VALID  
91  
92 }
```

Il metodo *checkIfValidChar* restituisce in output il valore 1 o 2.

Dato che il tipo di dato *byte* può rappresentare valori da -128 a 127, è stato scelto come valore di ritorno.

Come parametro, il metodo prende uno *short*, visto che esistono codici ASCII che superano 127, ma che non sono accettabili dal programma.

3.2 Verify Correctness

La procedura *verifyCorrectness* viene utilizzata nei *controlInput* di tutte le operazioni con l'obiettivo di controllare che il comando sia ben formattato così come richiesto dalla consegna.

La funzione ammette solamente tre codici ASCII dopo l'ultimo carattere del comando correttamente formattato:

- cod ASCII 32, perché sono ammessi spazi vicino alle ~ ;
- cod ASCII 126, perché vuol dire che è ben formattato, e perciò si passerà al controllo del prossimo comando;
- cod ASCII zero, perché vuol significare la fine della stringa nel caso dell'ultimo comando.

Infatti se sono presenti due comandi anche ben formattati ma non separati da tilde, questo è considerato errato.

La [Figura 3.1](#) descrive la funzione *verifyCorrectness* in alto livello.

Figura 3.1 Descrizione ad alto livello della procedura *verifyCorrectness*

```
103 public static void verifyCorrectness(short indexListInput) {
104
105     short codASCII = listInput[indexListInput] ;
106
107     if(codASCII == 126) {
108         return;
109     }
110
111     else if(codASCII == 32) {
112         indexListInput ++;
113         verifyCorrectness(indexListInput);
114     }
115
116     else if(codASCII == 0) {
117         return;
118     }
119
120     else {
121         indexListInput ++;
122         checkAnotherOperation(indexListInput);
123     }
124 }
125
126 }
```

Il metodo *verifyCorrectness* non restituisce niente, però verifica che il comando sia corretto per poi poter effettivamente chiamare la procedura che esegue l'operazione.

Viene quindi effettuato il controllo del carattere di dove ci troviamo nella *listInput* al momento opportuno.

Nel caso in cui il codice ASCII fosse o zero o 126, allora si ritorna all'istruzione seguente alla chiamata di *verifyCorrectness* e si esegue l'operazione.

Nel caso in cui il codice ASCII fosse 32, allora esso è permesso e si va avanti a controllare il prossimo.

In tutti gli altri casi abbiamo a che fare con un codice ASCII non ammissibile dal nostro programma, perciò si passa al prossimo comando da esaminare.

3.4 Find pHead of the Linked List

La procedura *findPHeadLinkedList* viene utilizzata per trovare l'ultimo elemento della lista controllando il puntatore al prossimo di ognuno; infatti se un elemento punta al primo della lista, cioè il suo PAHEAD = *pHeadLinkedList*, allora si tratta dell'ultimo elemento.

Questa procedura viene utilizzata nelle operazioni di ADD, DEL e SDX.

Nell'operazione di ADD serve a trovare la posizione da dove iniziare a cercare il primo spazio in memoria libero per poter inserire un nuovo elemento.

Nell'operazione di DEL serve a trovare l'ultimo elemento e cambiare il proprio PAHEAD, nel caso in cui l'elemento da eliminare fosse il primo della lista.

Nell'operazione di SDX serve a trovare la posizione in memoria dell'ultimo elemento così da far puntare il *pHeadLinkedList* a quell'indirizzo.

Uso della memoria

Nell'implementazione del progetto si è reso indispensabile sia l'utilizzo dello stack, sia l'utilizzo di strategie per la memoria statica.

Per quanto riguarda la memoria statica, come già descritto, sono stati inseriti i dati nella lista a partire da 0x100001F4, distante esattamente 500 byte da 0x10000000, indirizzo di partenza dal quale Ripes inizia a memorizzare i dati.

Questa decisione è stata presa perché è stato supposto che un'operazione possa occupare al più 16 byte, in quanto sono consentiti anche gli spazi tra le tilde.

Questo non avrebbe dato noi nel caso estremo in cui tutte e 30 le operazioni accettabili dal programma avessero occupato uno spazio di 16 byte.

Nell'esecuzione dell'operazione di SORT è stato scelto di partire dall'indirizzo 0x10000bb8 perché risultava molto lontano dal pHeadLinkedList, il che non avrebbe dato problemi per eventuali sovrapposizioni di dati.

Lo stack ha avuto un ruolo fondamentale nel progetto, perché ha permesso di salvare dati importanti come il return address, nel caso in cui ci fossero chiamate annidate

Inoltre è risultato indispensabile nelle operazioni di DEL e REV spiegate in dettaglio ai punti [2.2](#) e [2.7](#).

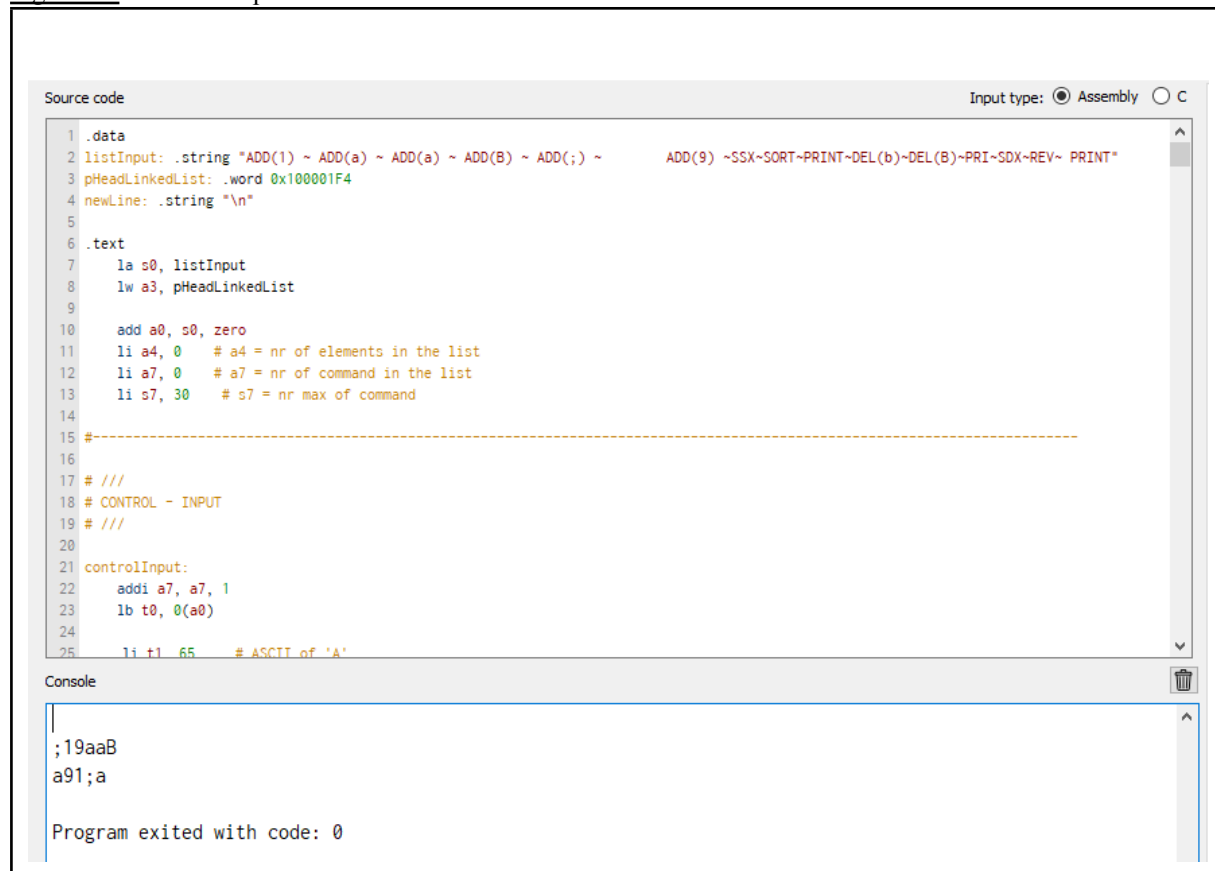
Test

Quest'ultima parte della relazione è dedicata alla dimostrazione del programma attraverso alcuni test. Innanzitutto verranno mostrati i test descritti nella parte "Esempio" della relazione.

Il primo esempio della consegna è raffigurato nella [Figura 5.0](#) (l'input invece in [Figura 5.1](#)) dove è possibile notare che:

- il programma ammette gli spazi tra le tilde;
- i comandi mal formattati come 'PRI' non vengono eseguiti;
- il programma termina senza errori grazie alla stampa finale, dove al registro a7 viene caricato il valore 10, rappresentante *Exit* per le system calls.

Figura 5.0 Primo esempio



The screenshot displays a code editor window with the title "Source code" and a tab labeled "Input type: Assembly". The code is written in MIPS assembly and includes comments in Italian. It defines data for a linked list and control input, then performs operations to calculate the number of elements and commands. The console output shows the memory address 19aaB, the value a91, and a final message indicating the program exited with code 0.

```
1 .data
2 listInput: .string "ADD(1) ~ ADD(a) ~ ADD(a) ~ ADD(B) ~ ADD(;) ~      ADD(9) ~SSX~SORT~PRINT~DEL(b)~DEL(B)~PRI~SDX~REV~ PRINT"
3 pHeadLinkedList: .word 0x100001F4
4 newLine: .string "\n"
5
6 .text
7     la s0, listInput
8     lw a3, pHeadLinkedList
9
10    add a0, s0, zero
11    li a4, 0    # a4 = nr of elements in the list
12    li a7, 0    # a7 = nr of command in the list
13    li s7, 30   # s7 = nr max of command
14
15    #-----
16
17    # ///
18    # CONTROL - INPUT
19    # ///
20
21    controlInput:
22        addi a7, a7, 1
23        lb t0, 0(a0)
24
25        li t1, 65    # ASCII of 'A'
```

Console

```
;19aaB
a91;a

Program exited with code: 0
```

Figura 5.1 Input del primo esempio

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

Il secondo esempio della consegna è raffigurato nella [Figura 5.2](#) (l'input invece in [Figura 5.3](#)) dove è possibile notare che:

- il programma ammette gli spazi tra le tilde;
- i comandi mal formattati come 'add(B)', 'ADD', 'SORT(a)' e 'DEL(bb)' non vengono eseguiti;
- il programma termina senza errori grazie alla stampa finale, dove al registro a7 viene caricato il valore 10, rappresentante *Exit* per le system calls.

Figura 5.2 Secondo esempio

```
1 .data
2 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"
3 pHeadLinkedList: .word 0x100001F4
4 newLine: .string "\n"
5
6 .text
7 la s0, listInput
8 lw a3, pHeadLinkedList
9
10 add a0, s0, zero
11 li a4, 0 # a4 = nr of elements in the list
12 li a7, 0 # a7 = nr of command in the list
13 li s7, 30 # s7 = nr max of command
14
15 #-----
16
17 # ///
18 # CONTROL - INPUT
19 # ///
20
21 controlInput:
22 addi a7, a7, 1
23 lb t0, 0(a0)
24
25 li t1, 65 # ASCII of 'A'
```

Console

```
1aB9
1aB9
1a9
19a

Program exited with code: 0
```

Figura 5.3 Input del secondo esempio

```
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"
```


Il quarto esempio, raffigurato in [Figura 5.7](#), punta a far vedere che due comandi ben formattati, ma non separati da tilde, non vengono eseguiti.

Figura 5.7 Quarto esempio

```
1 .data
2 listInput: .string "ADD(0)~ADD(1)~ADD(2)~ADD(3)~ADD(4)~ADD(5)~PRINT~ SDXSORT~PRINT"
3 pHeadLinkedList: .word 0x100001F4
4 newLine: .string "\n"
5
6 .text
7     la s0, listInput
8     lw a3, pHeadLinkedList
9
10    add a0, s0, zero
11    li a4, 0    # a4 = nr of elements in the list
12    li a7, 0    # a7 = nr of command in the list
13    li s7, 30   # s7 = nr max of command
14
15    #-----
16
17    # ///
18    # CONTROL - INPUT
19    # ///
20
21    controlInput:
22        addi a7, a7, 1
23        lb t0, 0(a0)
```

Console

```
012345
012345

Program exited with code: 0
```

Come è possibile vedere dall'output, nonostante che siano ben formattati, SDX e SORT non vengono eseguiti perché non sono separati dalla tilde.

Il quinto esempio, in Figura 5.8, punta a far vedere la correttezza di REV.

Figura 5.8 Quinto esempio

```
1 .data
2 listInput: .string "ADD(0)~ADD(m)~ADD(k)~ADD(9)~ADD(7)~ADD(!)~ADD(1)~ADD(:)~ADD(E)~ADD(Z)~PRINT~REV~PRINT"
3 pHeadLinkedList: .word 0x100001F4
4 newline: .string "\n"
5
6 .text
7     la s0, listInput
8     lw a3, pHeadLinkedList
9
10    add a0, s0, zero
11    li a4, 0    # a4 = nr of elements in the list
12    li a7, 0    # a7 = nr of command in the list
13    li s7, 30   # s7 = nr max of command
14
15    #-----
16
17    # ///
18    # CONTROL - INPUT
19    # ///
20
21    controlInput:
22        addi a7, a7, 1
23        lb t0, 0(a0)
```

Console

0mk97!1:EZ

ZE:1!79km0

Program exited with code: 0

Il sesto esempio, in [Figura 5.9](#), punta a dimostrare la correttezza di SORT.

Figura 5.9 Sesto esempio

```
1 .data
2 listInput: .string "ADD(0)~ADD(m)~ADD(k)~ADD(9)~ADD(7)~ADD(!)~ADD(1)~ADD(:)~ADD(E)~ADD(Z)~PRINT~SORT~PRINT"
3 pHeadLinkedList: .word 0x100001F4
4 newLine: .string "\n"
5
6 .text
7     la s0, listInput
8     lw a3, pHeadLinkedList
9
10    add a0, s0, zero
11    li a4, 0    # a4 = nr of elements in the list
12    li a7, 0    # a7 = nr of command in the list
13    li s7, 30   # s7 = nr max of command
14
15    #-----
16
17    # ///
18    # CONTROL - INPUT
19    # ///
20
21    controlInput:
22        addi a7, a7, 1
23        lb t0, 0(a0)
```

Console

0mk97!1:EZ

!:079klmEZ

Program exited with code: 0

Il settimo e ultimo esempio, in [Figura 6.0](#), punta a dimostrare la correttezza di SDX e SSX.

[Figura 6.0](#) Settimo esempio

```
1 .data
2 listInput: .string "ADD(0)~ADD(m)~ADD(k)~ADD(9)~ADD(7)~ADD(!)~ADD(1)~ADD(:)~ADD(E)~ADD(Z)~PRINT~SDX~PRINT~SSX~SSX~PRINT"
3 pHeadLinkedList: .word 0x100001F4
4 newline: .string "\n"
5
6 .text
7     la s0, listInput
8     lw a3, pHeadLinkedList
9
10    add a0, s0, zero
11    li a4, 0      # a4 = nr of elements in the list
12    li a7, 0      # a7 = nr of command in the list
13    li s7, 30     # s7 = nr max of command
14
15    -----
16
17    # ///
18    # CONTROL - INPUT
19    # ///
20
21    controlInput:
22        addi a7, a7, 1
23        lb t0, 0(a0)
```

Console

```
0mk97!1:EZ
Z0mk97!1:E
mk97!1:EZ0
```

```
Program exited with code: 0
```