Progetto Assembly RISC-V per il Corso di Architetture degli Elaboratori –A.A. 2020/2021 –

Gestione di Liste Concatenate

Autore: Nabiollah Tavakkoli

E-mail: nabiollah.tavakkoli@stud.unifi.it

Numero di matricola: 5953800

Data di consegna: 18/01/2022

- **I registri importanti:**
  - S0: contiene l'indirizzo alla testa della stringa dopo la chiamata del metodo TRIM
  - S2: contiene l'indirizzo alla testa della linkedList
  - S3: contiene il valore (0xFFFFFFFF) memorizzato nei primi o negli ultimi 4 byte dei nodi alla testa e alla coda della lista concatenata doppia
  - S4: indica l'indirizzo alla testa dell'array, dove gli elementi vengono aggiunti per poter essere ordinati (nel SORT)
  - S10: contiene la lunghezza della LISTA_CONCATENATA_DOPPIA, e tale lunghezza
    - aumenta con il comando ADD
    - diminuisce con il comando DEL
  - S11: contiene la lunghezza della ListInput dopo TRIM, e
    - il blocco "listinput_is_empty" sarà eseguito se la stringa dei comandi (LISTINPUT) è vuota
  - T0: indica l'indice della stringa a partire da cui caricare una parola / un byte
  - T1: indica la posizione della prossima TILDE

  - A1: contiene il valore da eliminare o aggiungere (IMPORTANTE in DEL, ADD)
  - A2: contiene l'indirizzo alla testa dell'ultimo nodo (l'indirizzo alla coda della lista)
  - A5: Indica, se il metodo ADD è stato chiamato più di una volta oppure no

- Nota:
  - La stringa "END" indica la fine del programma
  - Tutti i simboli "?" nei commenti sono in realtà "è"

1. metodo TRIM:

- Poiché gli spazi vicini alle ~ sono ammessi e devono essere tollerati dal programma, questo metodo è stato implementato per eliminare (ignorare) tali spazi (come ad esempio: ~ADD(a)~~spazio~~ ~, ~~~spazio~~DEL(b)~), ma NON deve eliminare gli spazi vicini alle parentesi (come ad esempio: ADD (~~spazio~~), D ~~spazio~~ EL (c))
- È una combinazione di CONFRONTO e CONCAT, viene eseguito nel seguente modo:
  - Carica ciascun carattere della stringa "ListInput"
    - se il codice ASCII del carattere è diverso dal codice ASCII dello "spazio" (32), allora tale carattere viene concatenato con il precedente in una stringa
    - altrimenti, se si trova uno spazio, controlla la sua posizione
      - Se il carattere successivo oppure precedente dallo spazio è una Tilde (cioè; ha un codice ASCII uguale a 126), allora lo spazio viene scartato (con incrementare il contatore, in realtà viene ignorato)
      - Altrimenti, viene concatenato
  - Dopo TRIM l'indirizzo alla testa della stringa risultante viene memorizzato in **s0**
- Questo metodo viene anche usato per calcolare la lunghezza della ListInput dopo il TRIM

2. metodo split_call:

- questo metodo verifica l'esistenza di un eventuale comando (una funzione) da chiamare, se tale comando è valido esso viene chiamato, altrimenti viene ignorato. Nel caso dei comandi ADD e DEL, il carattere da aggiungere o rimuovere viene memorizzato nel registro a1
- viene eseguito nel seguente modo:

I. Sono dichiarati i due registri t0, t1, dove:
- t0 contiene l'indice del prossimo byte oppure parola della stringa
- t1 contiene l'indirizzo della prossima Tilde (~) della stringa

II. Il blocco **funct_exists**:
- verifica la presenza di un eventuale comando (una funzione da chiamare), scorre la stringa byte-per-byte fino a trovare la Tilde oppure il valore nullo;
  - Se trova il valore nullo, vuol dire: siamo arrivati fino alla fine della stringa e ci sarà al max un comando da chiamare
  - Se trova la Tilde, vuol dire: eventualmente esiste una funzione da chiamare, allora va a verificare la sua presenza nel blocco "split_loop"

III. Il blocco **split loop**:
- carica una parola della stringa a partire da:
  - L'indirizzo alla testa della stringa (s0) + 0, inizialmente
  - L'indirizzo alla testa della stringa (s0) + l'indice a partire da cui caricare una parola oppure un byte dalla stringa (t0), nei passi successivi
- Dati i codici ASCII delle stringhe ["ADD(","DEL(", …], Se la parola contenente nel registro ha un codice ASCII uguale ad uno di essi, allora va a verificare che sia un comando valido (controllando altri byte e/o la lunghezza del comando) e poi chiama tale funzione:
  - Codice ASCII di "ADD("= 675562561
  - Codice ASCII di "DEL(" = 676087108
  - Codice ASCII di "PRIN"= 1313428048
  - Codice ASCII di "SORT"= 1414680403

- nel caso del comando “REV” la correttezza si verificata
  - ♦ controllando il primo byte e poi
  - ♦ controllando i successivi 2 byte (caricando un half-word)
- nel caso dei comandi “ADD” e” DEL” il codice ASCII del carattere da aggiunger o eliminare viene memorizzato nel registro a1, e poi la funzione viene chiamata

IV. nei due casi in cui il comando non sia valido oppure dopo la chiamata della funzione, viene chiamato il blocco “incremento_split” per controllare ed eseguire i successivi comandi oppure per uscire dal ciclo

V. nei due blocchi “funct_exists”,” split_loop” viene calcolato la lunghezza di ciascun comando, poi singolarmente verificato (nei rispettivi blocchi: SORT_controll, …)

comand_length + 1 = (posizione_della_prossima_Tilde +1) - posizione_della_vecchia_Tilde +1

3. LFSR:

❖ Dato un SEED su 16 bit, il metodo genera un altro valore su 16 bit che viene aggiunto all’indirizzo di partenza 0x00002000 per generare un nuovo indirizzo di memoria

❖ Il metodo viene suddiviso in 3 blocchi:

I. Nel primo blocco:

- L’indirizzo alla testa della lista viene memorizzato nel registro a3

II. Nel secondo blocco:

- I 16 bit meno significativi dell’ultimo indirizzo precedentemente utilizzato, viene memorizzato nel registro a4, usando:

  - ♦ li t3, 0x0000FFFF
  - ♦ and a4, a3, t3
  - ♦ Oppure
  - ♦ slli a4, s2, 16
  - ♦ srli a4, s4, 16

- Viene generato il nuovo 16-bit-LFSR
- Tale valore viene sommato all’indirizzo alla testa della lista-concatenata-doppia

III. Nel terzo blocco:

- Controlliamo se i prossimi 9 byte a partire dall’indirizzo appena generato, sono occupati oppure no:
  - ♦ Se sì, va a generare un nuovo indirizzo
  - ♦ Se no, tale valore viene salvato nel registro a0, e viene restituito al chiamante

4. ADD (char):

- Va usato
  - per aggiungere un nuovo valore (char) nel campo data di un nodo della lista
  - incrementa il contenuto del registro s10 di un'unità (che indica la lunghezza de lista)
- Viene eseguito nel seguente modo:
  - Controlla il numero di volte in cui "la funzione ADD" è stata chiamata (controllando il contenuto del registro a5)
    - Se la funzione non è stata già chiamata, viene eseguito il blocco "senza_LFSR"
      - Il valore del registro "a5" viene incrementato di un'unità, quindi tale blocco non può essere più chiamato
      - 0xFFFFFFFF viene memorizzato nei PRIMI 4 byte del nodo ESISTENTE
    - Altrimenti, viene chiamato il blocco "con_LFSR" che al suo interno chiama il metodo LFSR per generare il nuovo indirizzo di memoria, a partire dall'ultimo, e che sarà presente nel registro a0, e poi:
      - NUOVO indirizzo viene memorizzato negli ULTIMI 4 byte del nodo ESISTENTE
      - l'indirizzo alla testa del nodo ESISTENTE viene memorizzato nei PRIMI 4 byte del NUOVO nodo
      - il NUOVO INDIRIZZO viene memorizzato in a2
        - per poter generare il prossimo indirizzo a partire da esso (in LFSR)
  - E poi:
    - Il nuovo "char" presente in "a1" viene memorizzato nel campo DATA del nuovo nodo
    - 0xFFFFFFFF viene memorizzato negli ULTIMI 4 byte del nuovo nodo
  - Alla fine, il controllo passa al blocco "incremento_split"

5. DEL (char):

❖ Va usato per eliminare il primo “char”, se è presente nella lista, si esegue nel seguente modo:

I. Nel blocco DEL:
- l’indirizzo alla testa della lista viene memorizzato nel registro “a3”

II. Nel blocco find_element:
- cerchiamo il “char” da eliminare nella lista
  - Se tale valore esiste nella lista, il controllo passa al blocco DEL_loop
  - Se tale valore non esiste nella lista (se arriviamo fino alla fine della lista senza trovare il valore “char”), viene stampato: “tale valore non esiste nella lista”

III. Nel blocco DEL_loop:
- Controlliamo se il valore “char” si trova in CODA, TESTA oppure in nessuna di esse,
  - se non si trova in nessuna di esse allora:
    - **Modifichiamo il PAHEAD** dell’elemento PRECEDENTE, sovrascrivendo con l’indirizzo alla testa dell’elemento SUCCESSIVO rispetto a quello da rimuovere
    - **Modifichiamo il PBACK** dell’elemento SUCCESSIVO, sovrascrivendo con l’indirizzo alla testa dell’elemento PRECEDENTE rispetto a quello da rimuovere
  - Se si trova in testa alla lista, il controlla passa al blocco DEL_testa
  - Se si trova in coda alla lista, il controlla passa al blocco DEL_coda

IV. Il blocco DEL_testa:
- viene eseguito se l’elemento si trova in TESTA alla lista
- Controlla se tale nodo è l’unico della lista:
  - Se sì, diminuisce il valore del registro a5 in modo che il prossimo inserimento può essere eseguito, come il primo
  - Se no,
    - **Modifica il PBACK** dell’elemento SUCCESSIVO rispetto a quello da rimuovere, sovrascrivendo con 0xFFFFFFFF
    - **In questo caso,** l’indirizzo del nodo successivo diventa l’indirizzo alla testa della lista concatenata doppia, che viene assegnato al registro s2; poiché tale registro è di tipo S (Saved), allora prima di terminare l’applicazione, dobbiamo assegnare il valore dell’inizializzazione a s2 (li s2, 0x00002000) nel blocco end_func
      - ➢ In ogni caso, va al blocco “elimina” per eliminare tale nodo completamente

V. Il blocco DEL_coda:
- viene eseguito se l’elemento si trova in CODA alla lista, in tale caso:
  - **Modifica il PAHEAD** dell’elemento PRECEDENTE rispetto a quello da rimuovere, sovrascrivendo con 0xFFFFFFFF
  - **Aggiorna l'ultimo indirizzo memorizzato in a2**, a partire da cui viene calcolato il nuovo16-bit- LFSR (per generare un nuovo indirizzo)
  - va al blocco “elimina” per eliminare tale nodo completamento

VI. Il blocco elimina:
- viene eseguito per eliminare completamente il nodo contenente il valore “char”

VII. Il blocco el_notExists:
- viene eseguito, se tale valore (char) non si trova nella lista

6. PRINT:

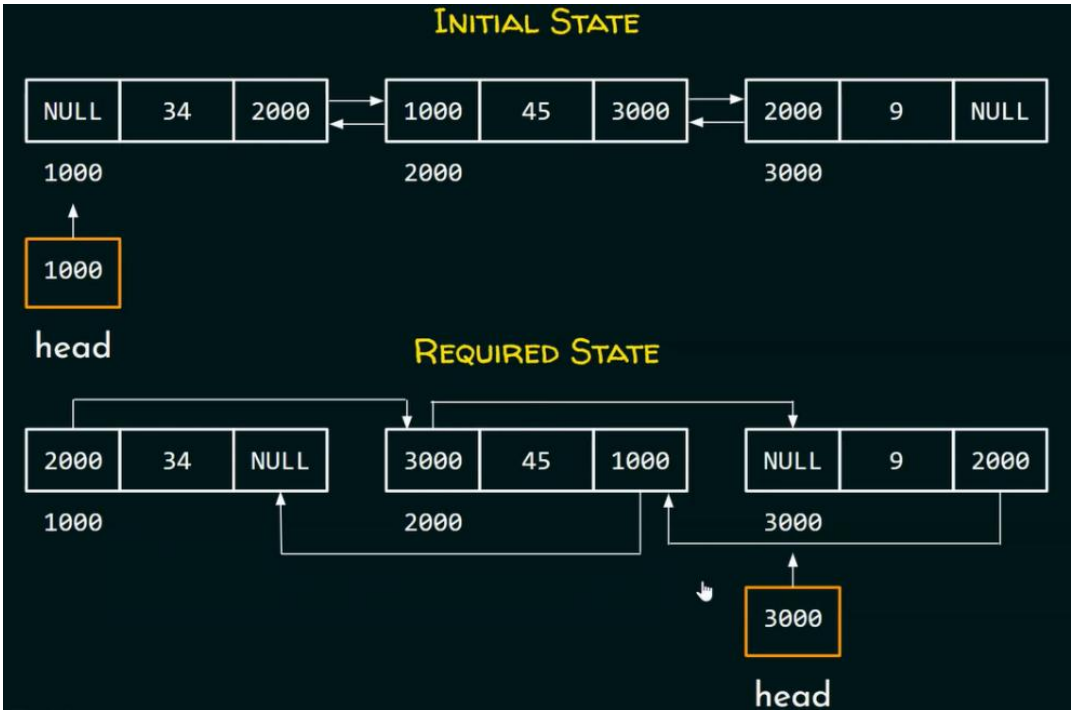
- Va utilizzato per stampare la lista, partendo dall'indirizzo alla testa della lista, stampa

PBACK | DATA | PAHEAD

- Viene eseguito nel seguente modo:
  - Se la lista è vuota, viene stampato "the list is currently empty"
  - Altrimenti, Stampa la lista finché PAHEAD assume valore 0xFFFFFFFF (indica la fine della lista)

7. REV:

- Questo metodo inverte la lista
- viene eseguito, modificando i PUNTATORI degli elementi della lista
  - L'indirizzo alla testa della LISTA viene memorizzato nel registro a2, a partire da cui viene calcolato il nuovo16-bit- LFSR (per generare un nuovo indirizzo)
  - L'indirizzo del nodo corrente viene memorizzato in a3
  - L'indirizzo del nodo precedentemente invertita viene memorizzato in t3
  - Finché "a3" non contiene il valore 0xFFFFFFFF
    - Gli indirizzi memorizzati nei puntatori PAHEAD e PBACK vengono scambiati (swap)
    - L'indirizzo alla testa del nodo corrente viene memorizzato nel registro "t3"
    - Dopo lo SWAP, l'indirizzo presente nel campo PBACK del nodo corrente, indica l'indirizzo del nodo successivo da invertire; tale indirizzo viene considerato il nuovo indirizzo del nodo corrente
  - Alla fine, se il registro "a3" contiene il valore 0xFFFFFFFF il metodo termina e l'indirizzo dell'ultimo nodo letto (memorizzato in "t3") indica, l'indirizzo alla testa della lista invertita e viene memorizzata nel registro "s2"



8. SORT:

- IDEA generale:
  - Inserire in un array tutti i valori presenti nel campo data dei nodi della lista
  - La dimensione di tale array = alla dimensione della lista-concatenata-doppia
  - Ordinare tale array usando l'algoritmo Bubble Sort, in ordine DECRESCENTE
  - Inserire gli elementi dell'array nella LISTA CONCATENATA DOPPIA, in questo ordine: **lettere MAIUSCOLE > lettere minuscole > numeri > carattere extra**
  - Chiamare la funzione REV, per ottenere una lista concatenata doppia ordinata in ordine CRECRENTE
- Il metodo viene suddiviso in 6 blocchi principali:

I. Nel blocco SORT:
   - L'indirizzo alla testa dell'array viene memorizzato nel registro "a0"
   - L'indirizzo alla testa della lista viene memorizzato nel registro "a4"

II. Nel blocco inserting_loop:
   - Inseriamo nell'array tutti i dati della lista

III. Nel blocco sorting_loop:
   - Chiamiamo il metodo "ordina" per ordinare l'array (in ordine DEC), e poi
   - Inseriamo gli elementi dell'array ordinato nella lista, secondo l'ordine richiesto; dopo ogni inserimento tale elemento viene sostituito con " ~ ", così non ci sarà bisogno eseguire 4 confronti per l'inserimento dei caratteri extra (basterà un solo confronto che permette di inserire tutti i valori diverso di ~) nella lista

IV. Nel blocco end_sorting:
   - Chiamiamo la funzione REV, per poter ottenere una lista ordinata in ordine CRESCENTE

V. I blocchi ordina e swap vanno utilizzati per implementare
   - un bubble-sort semplice (in ordine decrescente), dove a partire dall'indirizzo alla testa dell'array gli elementi vengono confrontati in due, e se il successivo risulta maggiore dell'elemento corrente, viene eseguito uno swap

TEST

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

Console

```
   |   0xffffffff   |   1   |   0x16803000   |
   |   0x2000   |   a   |   0x1dc03800   |
   |   0x16803000   |   a   |   0x18603c00   |
   |   0x1dc03800   |   B   |   0x1ab03e00   |
   |   0x18603c00   |   ;   |   0x1bd83f00   |
   |   0x1ab03e00   |   9   |   0xffffffff   |
   |   0xffffffff   |   ;   |   0x1ab03e00   |
   |   0x1bd83f00   |   1   |   0x18603c00   |
   |   0x1ab03e00   |   9   |   0x1dc03800   |
   |   0x18603c00   |   a   |   0x16803000   |
   |   0x1dc03800   |   a   |   0x2000   |
   |   0x16803000   |   B   |   0xffffffff   |
tale elemento da eliminare non esiste nella lista

   |   0xffffffff   |   a   |   0x1dc03800   |
   |   0x16803000   |   a   |   0x18603c00   |
   |   0x1dc03800   |   9   |   0x1ab03e00   |
   |   0x18603c00   |   1   |   0x1bd83f00   |
   |   0x1ab03e00   |   ;   |   0xffffffff   |
END
```

Execution info

Cycles:	4685
Instrs. retired:	3226
CPI:	1.45
IPC:	0.689
Clock rate:	7.64 KHz

```
listinput: .string "ADD(1)~ADD(a)~ADD()~ ADD(B)~ADD~ADD(9)~PRINT~SORT(a)~PRINT~DEL(bb)~DEL(B)~PRINT~REV~PRINT"
```

Console

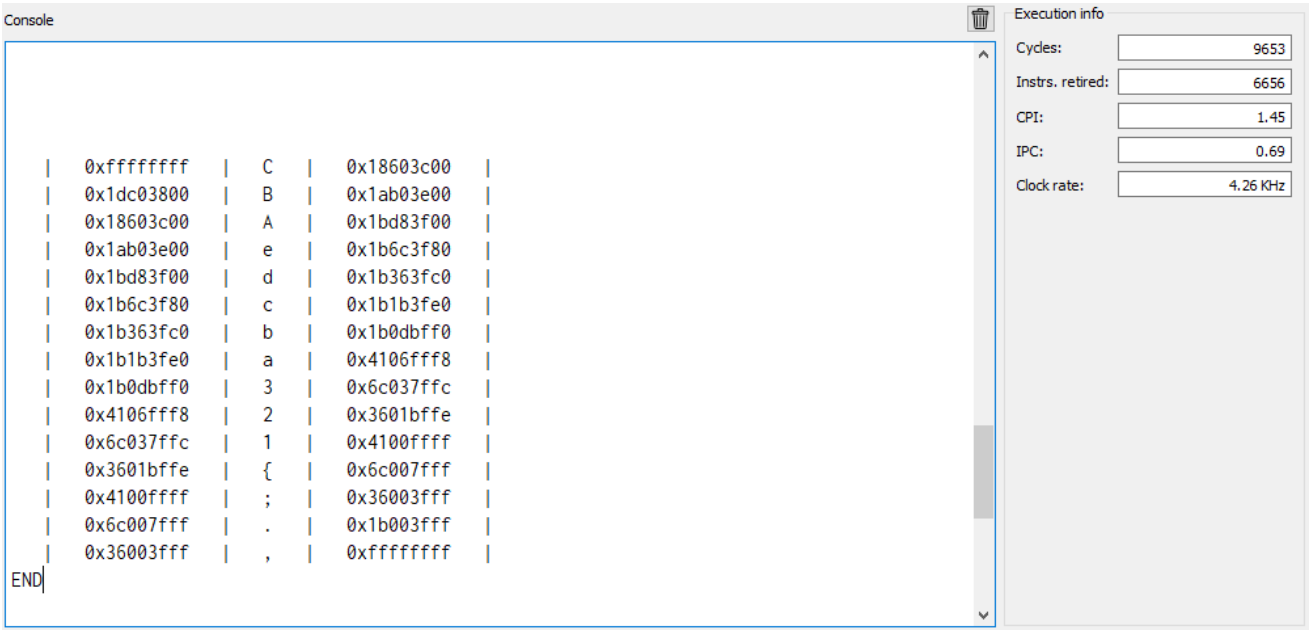
```
   |   0xffffffff   |   1   |   0x16803000   |
   |   0x2000   |   a   |   0x1dc03800   |
   |   0x16803000   |   B   |   0x18603c00   |
   |   0x1dc03800   |   9   |   0xffffffff   |
   |   0xffffffff   |   1   |   0x16803000   |
   |   0x2000   |   a   |   0x1dc03800   |
   |   0x16803000   |   B   |   0x18603c00   |
   |   0x1dc03800   |   9   |   0xffffffff   |
   |   0xffffffff   |   1   |   0x16803000   |
   |   0x2000   |   a   |   0x18603c00   |
   |   0x16803000   |   9   |   0xffffffff   |
   |   0xffffffff   |   9   |   0x16803000   |
   |   0x18603c00   |   a   |   0x2000   |
   |   0x16803000   |   1   |   0xffffffff   |
END
```

Execution info

Cycles:	3608
Instrs. retired:	2529
CPI:	1.43
IPC:	0.701
Clock rate:	5.14 KHz

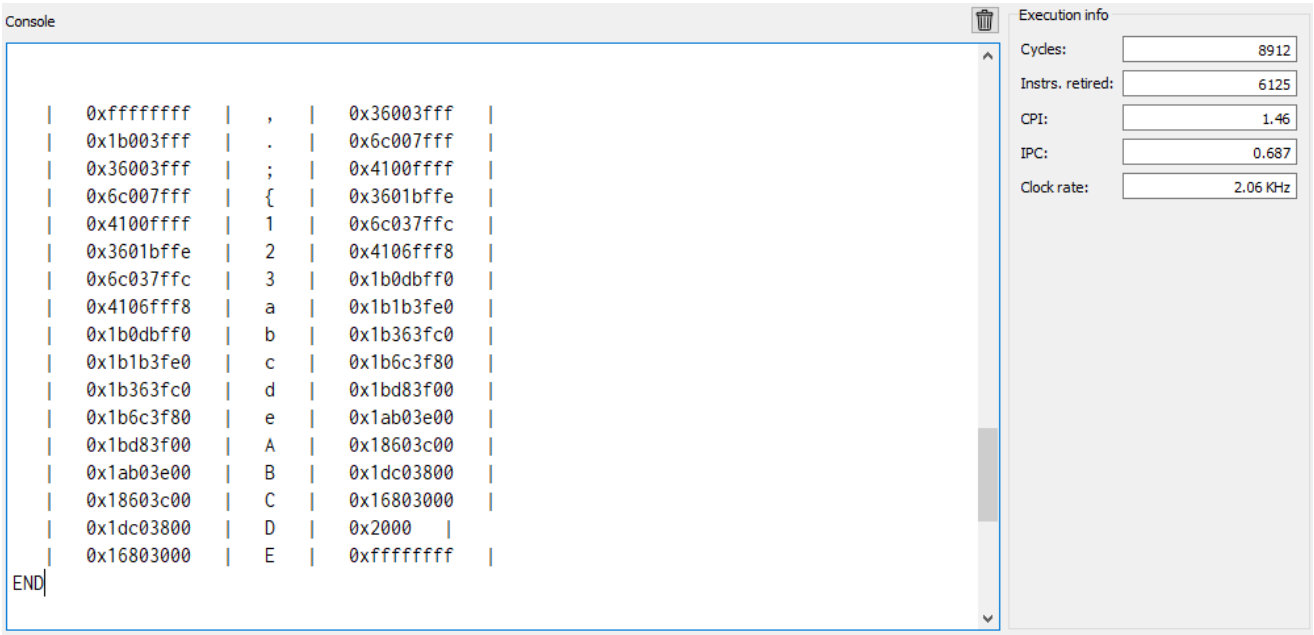
listinput: .string "ADD(A)~ADD(b)~ADD(C)~ ADD(a)~ADD(B) ~ADD(c)~ADD(,)~ADD(.)~ADD(;)~ADD(1)~ADD(2)~ADD(3)~ADD({)~ADD(d)~ADD(e)~ADD(D)~ADD(E)~ SORT~REV~DEL(D)~DEL(E) ~PRINT"

```
listinput: .string "ADD(A)~ADD(b)~ADD(C)~ ADD(a)~ADD(B) ~ADD(c)~ADD(,)~ADD(.)~ADD(;)~ADD(1)~ADD(2)~ADD(3)~ADD({)~ADD(d)~ADD(e)~ADD(D)~ADD(E)~SORT~REV~DEL(D)~DEL(E) ~PRINT"
```

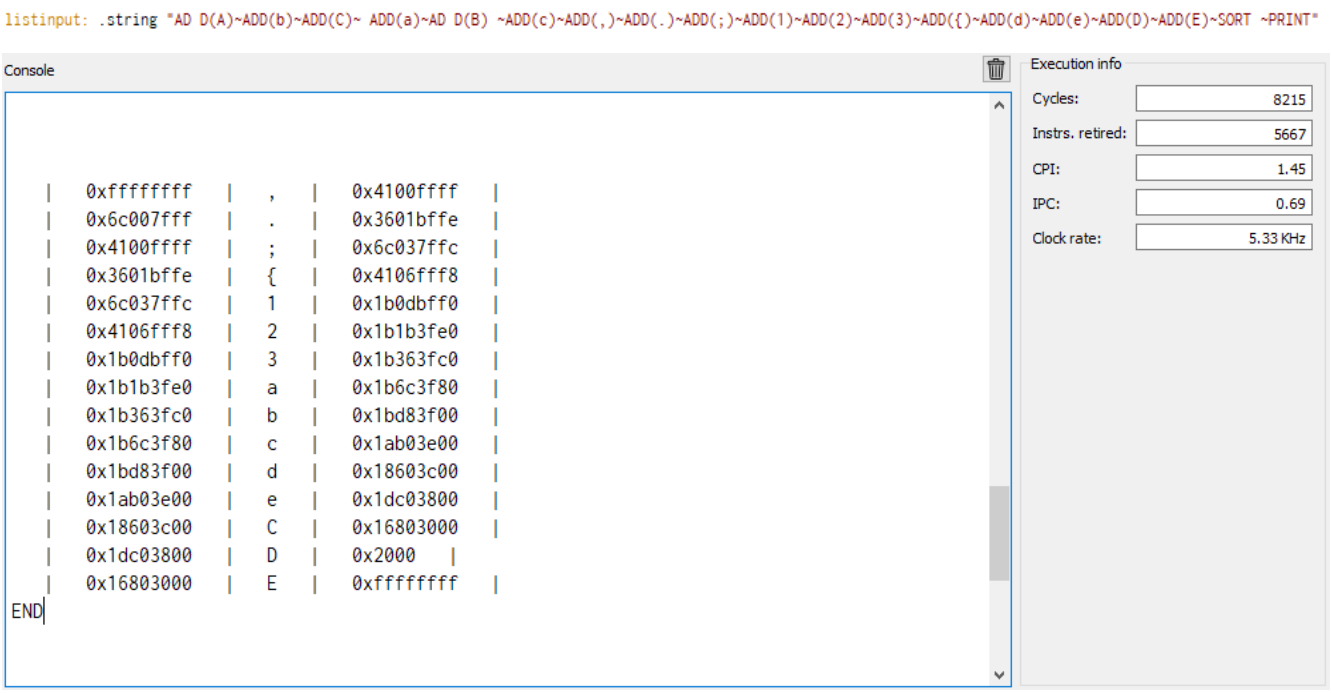


listinput: .string "ADD(A)~ADD(b)~ADD(C)~ ADD(a)~ADD(B) ~ADD(c)~ADD(,)~ADD(.)~ADD(;)~ADD(1)~ADD(2)~ADD(3)~ADD({)~ADD(d)~ADD(e)~ADD(D)~ADD(E)~ SORT ~PRINT"

```
listinput: .string "ADD(A)~ADD(b)~ADD(C)~ ADD(a)~ADD(B) ~ADD(c)~ADD(,)~ADD(.)~ADD(;)~ADD(1)~ADD(2)~ADD(3)~ADD({)~ADD(d)~ADD(e)~ADD(D)~ADD(E)~SORT ~PRINT"
```



listinput: .string "AD D(A)~ADD(b)~ADD(C)~ ADD(a)~AD D(B) ~ADD(c)~ADD(,)~ADD(.)~ADD(;)~ADD(1)~ADD(2)~ADD(3)~ADD({)~ADD(d)~ADD(e)~ADD(D)~ADD(E)~SORT ~PRINT"



listinput: .string ""