

# ARCHITETTURE DEGLI ELABORATORI

A.A. 2020-2021

## Progetto: Liste Concatenate Doppie in RISC-V



08 Settembre 2021

**Jacob Angeles**

7024541

*jacob.angeles@stud.unifi.it*

# Table of Contents

1	Obiettivi .....	2
2	Algoritmi del <i>main</i> .....	2
2.1	Phase 0 – Verify number of commands .....	2
2.2	Phase 1 – Identify First Valid Letter .....	3
2.3	Phase 2 – Verify comand format .....	3
2.4	Phase 3 – Find execution point(tilda) .....	5
2.5	Phase 4 – Execute command/error message .....	5
3	Algoritmi dei comandi.....	5
3.1	ADD – Inserimento di un Elemento .....	5
3.2	DEL – Rimozione di un Elemento .....	7
3.3	PRINT – Stampa della Lista.....	9
3.4	REV – Inversione degli Elementi della Lista.....	10
3.5	SORT – Ordinamento della Lista.....	10
4	Registri & Memoria.....	12
5	Test .....	13

# 1 Obiettivi

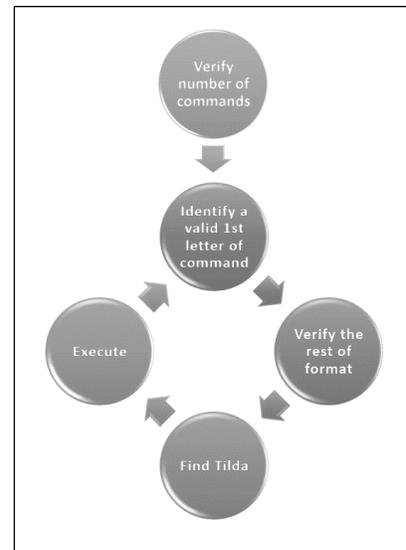
L'obiettivo del progetto è di realizzare una *liste concatenate doppie* con alcune delle sue operazioni fondamentali (*add, delete, print, reverse, sort*) in RISC-V.

## 2 Algoritmi del *main*

L'idea generale della **main** è divisa in 5 fasi:

- ✓ **Phase 0** - verifica del numero di comandi dentro la variabile *listInput*
- ✓ **Phase 1** - identificazione del primo carattere valido per un comando
- ✓ **Phase 2** - confronto del resto di caratteri del comando
- ✓ **Phase 3** - ricerca della tilda "~"
- ✓ **Phase 4** - esecuzione del comando o eventuali messaggio di errori

In questo modo è facile gestire e seguire il flusso degli istruzioni del programma. Le specifiche di tutte le fasi sono dettagliate di seguito.



### 2.1 Phase 0 – Verify number of commands

L'obiettivo di questa fase è individuare se il numero di comandi (*validi e non validi*) dentro la variabile "**listInput**" non supera il limite massimo (*max 30 comandi per ogni esecuzione del programma*).

Ogni comando è suddiviso dal carattere speciale tilda "~", e il resto invece, sono da considerare comando valido o non valido (i comandi validi verranno presentati successivamente).

```
function func_counter(listInput[])  
    int n, i ← 0  
    while listInput[i] is not null  
        if listInput[i] = '~' then  
            n++  
        i++  
    return n
```

Come è descritto nella *pseudo-codice*, si verifica ogni carattere all'interno del *listInput* e si conta quante sono le tilde (il totale è salvato nella variabile *n*). La funzione restituisce *n*.

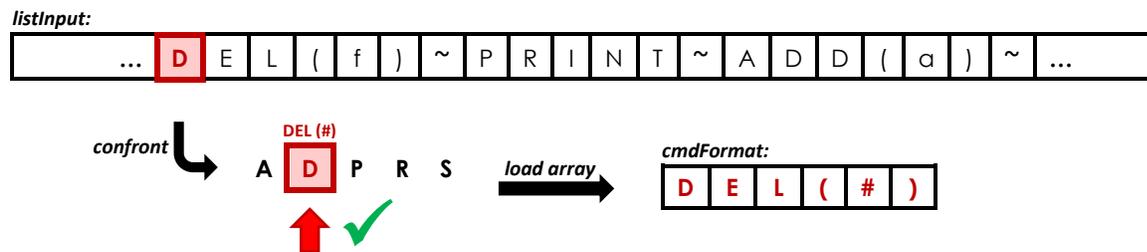
Nella *main*, se **n > 30** il programma esegue un messaggio di errore e termina l'esecuzione. Altrimenti se **n ≤ 30** prosegue alla prossima fase.

Dopo aver superato il primo requisito, bisogna verificare che ogni comando all'interno della variabile "listInput" sia corretto prima di eseguirlo.

Di seguito i comandi (con formato valido richiesto) che si effettuano sulla nostra lista concatenata: **ADD(#)**, **DEL(#)**, **PRINT**, **REV**, **SORT**. Il simbolo # rappresenta gli input del comando. Altri sequenze di caratteri che non rispettano questi formati sono da considerare non validi.

## 2.2 Phase 1 – Identify First Valid Letter

Lo scopo della fase 1 è di identificare la prima lettera valida che è potenzialmente da eseguire. Per essere valida, essa deve corrispondere alle seguenti lettere maiuscole: **A, D, P, R, S**. Successivamente, si carica un array che rappresenta il corretto formato da utilizzare per confrontare il resto dei caratteri nella fase successiva.



In questa fase, lo spazio non viene considerato una violazione del formato.

Se si è verificato un errore di 1° tipo, ovvero trovare un carattere non corrispondente a A, D, P, R, S o allo spazio e non è una tilda, verrà attivato lo stato di errore e si continua a cercare una tilda.

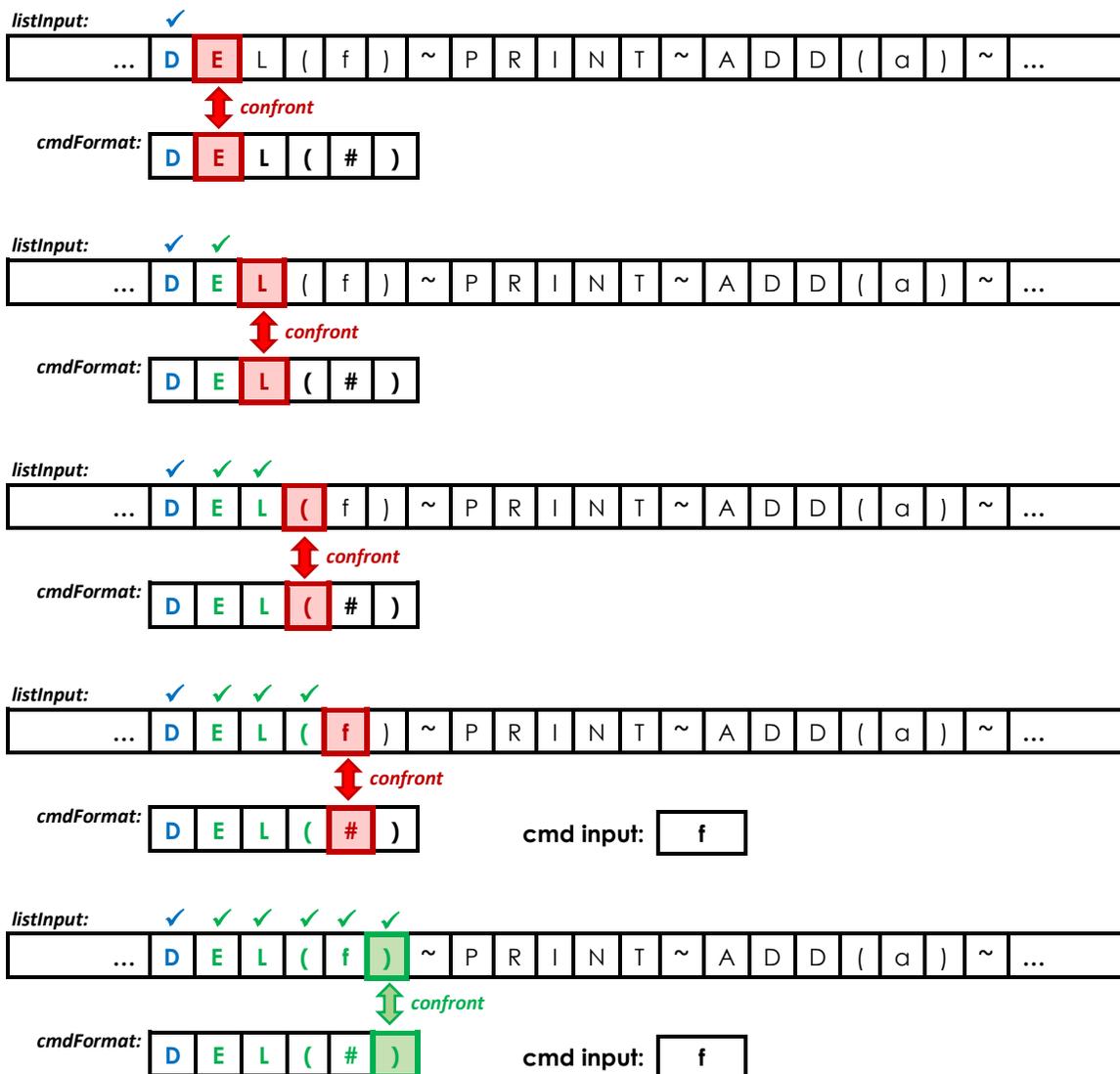
Se si è verificato un errore di 2° tipo (vale a dire trovare subito una tilda), verrà visualizzato un messaggio di errore.

## 2.3 Phase 2 – Verify comand format

Durante questa fase si convalida il formato del resto del comando utilizzando l'array *cmdFormat* come riferimento.

Nel caso di **ADD()** e **DEL()**, si usa il carattere # per capire se si deve salvare il valore nel puntatore del listInput come argomento del comando.

Una volta confermato si passa alla fase successiva.



In questa fase, lo spazio viene considerato una violazione del formato.

Se si è verificato un errore di 1° tipo (cioè trovare un carattere diverso da quello indicato al cmdFormat), e non è presente nè una tilda né uno spazio, verrà attivato lo stato di errore e si continua a cercare una tilda.

Se si è verificato un errore di 2° tipo (si trova quindi subito una tilda), verrà visualizzato un messaggio di errore e si passa alla fase 1.

## 2.4 Phase 3 – Find execution point(tilda)

L'obbiettivo di questa fase è verificare se ci sono ulteriori caratteri che violino il formato del comando. Un'alternativa è trovare la tilda (~) che sta ad indicare il punto di esecuzione del comando.

In questa fase, lo spazio non viene considerato una violazione del formato. Una volta trovata la tilda, si passa alla fase finale(fase 4).



## 2.5 Phase 4 – Execute command/error message

A questo punto, si esegue il comando. Nel caso in cui però si è verificato un errore nelle fasi precedenti, si effettua un messaggio di errore.

Terminata questa fase, si torna alla fase 1 finché ci saranno caratteri da controllare nella variabile *listInput*.

# 3 Algoritmi dei comandi

Di seguito gli algoritmi delle operazioni fondamentali(*add*, *delete*, *print*, *reverse*, *sort*) delle liste concatenate doppie.

## 3.1 ADD – Inserimento di un Elemento

Questa operazione aggiunge uno specifico elemento nella coda della lista concatenata.

Il primo passo da fare è identificare il posto nella memoria, tramite una funzione a parte ***get\_new\_address(lfsr)***. Essa restituisce l'indirizzo dove si colloca il nuovo elemento della lista concatenata.

Nel caso in cui esistessero già altri elementi nella lista concatenata, si collega il nuovo elemento all'ultimo elemento della lista cambiandolo il suo **PAHEAD** con l'indirizzo della nuova elemento. Esso salva come **PBACK** l'indirizzo dell'ultimo elemento della lista prima dell'inserimento.

Per concludere, si aggiornano i puntatori per tracciare l'ultimo elemento della lista e il riferimento lfsr. Vengono restituiti gli eventuali nuovi puntatori per l'ultimo elemento, il primo elemento e il nuovo lfsr.

```
function add(data, lfsr, tail_pointer, head_pointer)
    new_head_pointer ← head_pointer
    new_address, new_lfsr ← get_new_address(lfsr)
    new_address[pback] ← tail_pointer
    new_address[data] ← data
    new_address[pahead] ← head_pointer
    if tail_pointer is not null then
        pback[pahead] ← new_address
    else
        new_head_pointer ← new_address
    new_tail_pointer ← new_address
    return new_tail_pointer, new_lfsr, new_head_pointer
```

```

253 # -----
254 # - @name      Func_ADD(char DATA, byte lfsr, byte PTAIL, byte PHEAD)
255 # - @descript. Inserts the DATA char to the linked list.
256 # - @args     char  a2 - DATA
257 # - @args     byte  a3 - lfsr
258 # - @args     byte  a4 - PTAIL
259 # - @args     byte  a5 - PHEAD
260 # - @return   byte  a0 - new_PTAIL
261 # - @return   byte  a1 - new_lfsr
262 # - @return   byte  sp - new_PHEAD
263
264 Func_ADD:
265 addi sp, sp, -8
266 sw a2, 4(sp)      # Save DATA to the stack memory.
267 sw ra, 0(sp)     # Save return address to the stack memory.
268
269 add a2, a3, zero  # Setup lfsr argument.
270 jal Get_new_address # Call Get_new_address(lfsr) function.
271 add t0, a0, zero  # New address where DATA must be stored.
272 add t3, a1, zero  # New lfsr
273 lw t2, 4(sp)     # Retrieve DATA from the stack memory.
274
275 insertData:
276 li t1, 0xFFFFFFFF # Store PBACK
277 sw a4, 0(t0)      # Store DATA
278 sb t2, 4(t0)     # Store PAHEAD
279 sw t1, 5(t0)
280
281 add a0, t0, zero
282 jal msg_add1
283
284 beq a4, t1, setHead # IF PTAIL == 0xFFFFFFFF, set PHEAD.
285 sw t0, 5(s1)      # Update the PAHEAD of the previous element with the new address of the new element.
286 sw a5, 4(sp)     # return value - PHEAD
287 j end_add
288 setHead:
289 sw t0, 4(sp)     # return value - PHEAD
290
291 end_add:
292 add a0, t0, zero # return value - PTAIL
293 add a1, t3, zero # return value - lfsr
294 lw ra, 0(sp)
295 addi sp, sp, 4
296 jr ra
297

```

```

301 # =====
302 # - @name      Get_new_address(byte lfsr)
303 # - @descript. Generates a new memory address.
304 # - @args      byte  a2 - lfsr
305 # - @return    byte  a0 - new_address
306 # - @return    byte  a1 - new_LFSR
307
308 Get_new_address:
309 add t0, a2, zero
310
311 calculate_LFSR:
312 srli t1, t0, 2      # lfsr >> 2
313 xor t2, t0, t1     # newBit = lfsr >> 0 XOR lfsr >> 2
314 srli t1, t0, 3     # lfsr >> 3
315 xor t2, t2, t1     # newBit = newBit XOR lfsr >> 3
316 srli t1, t0, 5     # lfsr >> 5
317 xor t2, t2, t1     # newBit = newBit XOR lfsr >> 5
318 slli t2, t2, 15    # newBit = newBit << 15
319 srli t1, t0, 1     # lfsr >> 1
320 or t0, t2, t1      # newBit = newBit OR lfsr >> 2
321
322 trim_16bit:
323 li t1, 0xFFFF0000
324 or t0, t0, t1
325 xor t0, t0, t1
326
327 new_address:
328 li t1, 0x00010000
329 or t2, t1, t0      # newAddress = 0x001000 OR 16-bit-LFSR
330
331 check_address:
332 lb t4, 9(t2)
333 bne t4, zero, calculate_LFSR
334 lw t4, 0(t2)
335 beq t4, zero, end_get_new_address
336 j calculate_LFSR
337
338 end_get_new_address:
339 add a0, t2, zero   # return value - new address
340 add a1, t0, zero   # return value - new LFSR
341 jr ra

```

### 3.2 DEL – Rimozione di un Elemento

Questo comando elimina un dato elemento nella coda della lista concatenata.

Finché ci sono elementi nella lista, si identifica l'elemento che corrisponde all'elemento da eliminare. Se non ci sono, termina il comando.

In caso contrario, si salva il riferimento all'elemento precedente (PBACK) e

```

// assuming the LinkedList has atleast 1 element
function delete(data, head_pointer, tail_pointer, element_number)
    current_elem ← head_pointer
    new_tail_pointer ← tail_pointer
    new_head_pointer ← head_pointer
    new_element_number ← element_number
    while current_elem is not null
        if current_elem != data then
            current_elem ← current_elem[pahead]
        else
            this_pback ← current_elem[pback]
            this_pahead ← current_elem[pahead]

```

del successivo (PAHEAD) rispetto a quello da rimuovere, e si procede all'eliminazione dell'elemento nella lista.

Dopodiché si sistemano i riferimenti con l'indirizzo corretto per mantenere la lista connessa. Si restituisce gli eventuali nuovi puntatori per l'ultimo elemento, il primo elemento e il numero totale di elementi nella lista.

```

delete current_elem
new_element_number ← element_number - 1
if this_pahead is null then
    this_pback[pahead] ← null
    new_tail_pointer ← this_pback
else if this_pback is null then
    this_pback[pback] ← null
    new_head_pointer ← this_pahead
else
    temp_pback ← this_pback
    temp_pahead ← this_pahead
    this_pback[pahead] ← temp_pahead
    this_pahead[pback] ← temp_pback
return new_tail_pointer, new_head_pointer, new_element_number
return new_tail_pointer, new_head_pointer, new_element_number

```

```

Source code
Input type: Assembly

346 # =====
347 # - @name      Func_DEL(char DATA, byte PHEAD, byte PTAIL, int element_number)
348 # - @descript. Find the first element which corrisponds to the DATA within the linked list & eliminates it.
349 # - @args      char  a2 - DATA
350 # - @args      byte  a3 - PHEAD
351 # - @args      byte  a4 - PTAIL
352 # - @args      int   a5 - data_counter
353 # - @return    byte  a0 - new_PTAIL
354 # - @return    byte  a1 - new_PHEAD
355 # - @return    int   sp - new_value_data_counter
356
357 Func_DEL:
358 addi sp, sp, -4
359 sw ra, 0(sp)
360 add t0, a3, zero      # PHEAD - the address of the first element.
361
362 fetchData:
363 lb t1, 4(t0)
364 bne t1, a2, checkNext
365
366 savePointers:
367 lw t2, 0(t0)
368 lw t3, 5(t0)
369
370 deleteChar:
371 sw zero, 0(t0)
372 sb zero, 4(t0)
373 sw zero, 5(t0)
374
375 li t4, 0xFFFFFFFF
376
377 fixPHead:
378 bne t3, t4, fixPTail
379 sw t4, 5(t2)      # beq t3 FFFF
380 add t0, t2, zero
381 add t1, a3, zero
382 j _deleted
383
384 fixPTail:
385 bne t2, t4, fixPHeadTail
386 sw t4, 0(t3)      # beq t2 FFFF
387 add t0, a4, zero
388 add t1, t3, zero
389 j _deleted

```

```

390
391 fixPHeadTail:
392 add t5, t2, zero
393 add t6, t3, zero
394 sw t6, 5(t2)          # bne t2 FFFF
395 sw t5, 0(t3)         # bne t3 FFFF
396 add t0, a4, zero
397 add t1, a3, zero
398 j _deleted
399
400 checkNext:
401 li t4, 0xFFFFFFFF
402 lw t1, 5(t0)
403 beq t1, t4, char_not_found
404 add t0, t1, zero
405 j fetchData
406
407 char_not_found:
408 jal msg_notFound
409 add a0, a4, zero      # return value - PTAIL
410 add a1, a3, zero     # return value - PHEAD
411 j _end_del
412
413 _deleted:
414 jal msg_deleted
415 addi a5, a5, -1
416 add a0, t0, zero     # return value - PTAIL
417 add a1, t1, zero     # return value - PHEAD
418
419 _end_del:
420 lw ra, 0(sp)
421 sw a5, 0(sp)        # return value - data_counter
422 jr ra

```

### 3.3 PRINT – Stampa della Lista

A partire dal primo elemento, questa funzionalità stampa il contenuto della lista in ordine di apparizione. Essa fa riferimento al successivo elemento tramite il puntatore PAHEAD di ogni elemento.

```

// assuming the LinkedList has atleast 1 element
function print(head_pointer)
    current_elem ← head_pointer
    while current_elem is not null
        print current_elem[data]
        current_elem ← current_elem[pahead]
    return null

```

```

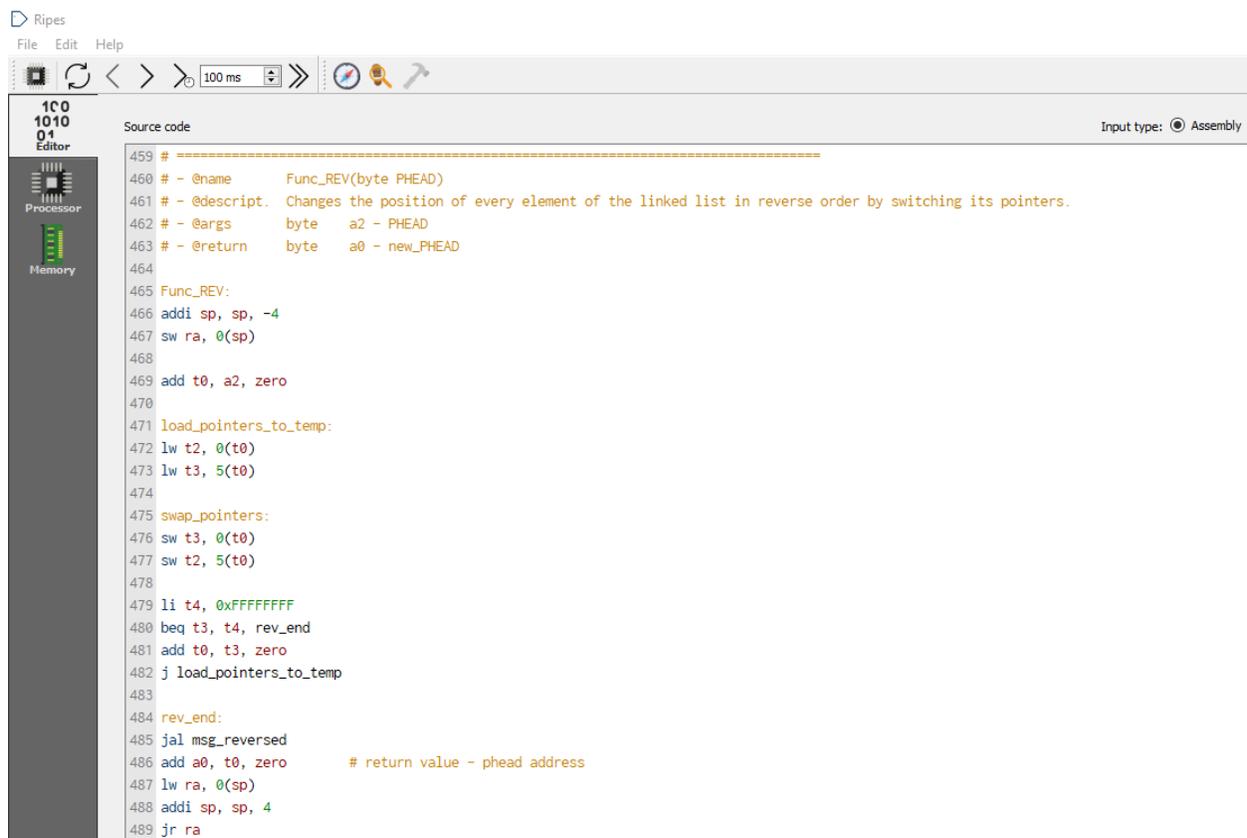
Editor
Processor
Memory
427 # =====
428 # - @name      Func_PRINT(byte PHEAD)
429 # - @descript. Prints every element of the linked list to the console.
430 # - @args     byte  a2 - PHEAD
431 # - @return    null
432
433 Func_PRINT:
434 addi sp, sp, -4
435 sw ra, 0(sp)
436
437 add t0, a2, zero
438 li t2, 0xFFFFFFFF
439
440 print_data:
441 lb a0, 4(t0)
442 li a7, 11
443 ecall
444
445 lw t1, 5(t0)
446 beq t1, t2, end_print
447 add t0, t1, zero
448 j print_data
449
450 end_print:
451 jal msg_newline
452 lw ra, 0(sp)
453 addi sp, sp, 4
454 jr ra

```

## 3.4 REV – Inversione degli Elementi della Lista

Questa operazione inverte gli elementi presenti nella lista concatenata, scambiando i puntatori. Quest'ultimi sono il PBACK, che avrà il valore di PAHEAD e vice versa, per ogni elemento della lista.

```
// assuming the LinkedList has atleast 1 element
function reverse(head_pointer)
    current_elem ← head_pointer
    while current_elem is not null
        temp_pback ← current_elem[pback]
        temp_pahead ← current_elem[pahead]
        current_elem[pback] ← temp_pahead
        current_elem[pahead] ← temp_pback
        current_elem ← temp_pahead
    return null
```



The screenshot shows the Ripes IDE interface. At the top, there's a menu bar with 'File', 'Edit', and 'Help'. Below it is a toolbar with various icons for navigation and execution. The main window is titled 'Source code' and displays assembly code for a function named 'Func\_REV'. The code includes comments in Italian and assembly instructions. On the left side, there's a sidebar with icons for 'Processor' and 'Memory'. The bottom right corner of the IDE shows 'Input type: Assembly'.

```
459 # =====
460 # - @name      Func_REV(byte PHEAD)
461 # - @descript. Changes the position of every element of the linked list in reverse order by switching its pointers.
462 # - @args      byte  a2 - PHEAD
463 # - @return    byte  a0 - new_PHEAD
464
465 Func_REV:
466 addi sp, sp, -4
467 sw ra, 0(sp)
468
469 add t0, a2, zero
470
471 load_pointers_to_temp:
472 lw t2, 0(t0)
473 lw t3, 5(t0)
474
475 swap_pointers:
476 sw t3, 0(t0)
477 sw t2, 5(t0)
478
479 li t4, 0xFFFFFFFF
480 beq t3, t4, rev_end
481 add t0, t3, zero
482 j load_pointers_to_temp
483
484 rev_end:
485 jal msg_reversed
486 add a0, t0, zero      # return value - phead address
487 lw ra, 0(sp)
488 addi sp, sp, 4
489 jr ra
```

## 3.5 SORT – Ordinamento della Lista

Questa funzionalità ordina gli elementi della lista in base all'algoritmo di ordinamento, bubble sort ricorsivo. Ogni elemento viene categorizzato tramite una funzione che restituisce: 1 per i caratteri speciali, 2 per i numeri, 3 per le lettere minuscole e 4 per le lettere maiuscole.

Prima di confrontare gli elementi per ogni valore ascii, si confronta per categoria. Se la categoria dell' primo elemento è maggiore rispetto al secondo, si scambia posizione. Se hanno lo stesso categoria, si confrontano i valori ascii per capire se si deve fare lo scambio.

```

494 # =====
495 # - @name      Bubblesort(byte PHEAD, int data_counter)
496 # - @descript. Sorts the elements of the linked list.
497 # - @args     byte  a2 - PHEAD
498 # - @args     int   a3 - data_counter
499 # - @return   null
500
501 Bubblesort:      # bubblesort(arr[], n)
502 li t5, 1
503
504 base_case:      # n==1
505 bne a3, t5, non_base_case
506 jr ra
507
508 non_base_case:  # n>1
509 li t6, 0        # i=0
510 addi t0, a3, -1 # n-1
511
512 while:
513 beq t0, t6, recursive_call
514
515 lb t2, 4(a2)    # Get arr[i]
516 lw t3, 5(a2)   # Get arr[i+1]
517 lb t3, 4(t3)
518
519 addi sp, sp, -12
520 sw t0, 0(sp)
521 sw a2, 4(sp)
522 sw ra, 0(sp)
523
524 add a2, t2, zero
525 jal Categorize # Perform categorize(arr[i]) function.
526 add t4, a0, zero # category(arr[i]) value.
527
528 add a2, t3, zero
529 jal Categorize # Perform categorize(arr[i+1]) function.
530 add t5, a0, zero # category(arr[i+1]) value.
531
532 lw ra, 0(sp)
533 lw a2, 4(sp)
534 lw t0, 8(sp)
535 addi sp, sp, 12
536
537 beq t4, t5 check_ascii # If category(arr[i]) == category(arr[i+1]), compare their ascii values.
538 bgt t4, t5 swap        # If category(arr[i]) > category(arr[i+1]), then swap values.
539 j sort_next           # Otherwise category(arr[i]) < category(arr[i+1]), no need to swap. Proceed to next pair.
540
541 check_ascii:
542 bgt t2, t3 swap        # If arr[i] > arr[i+1], then swap values.
543 j sort_next           # Otherwise arr[i] < arr[i+1], no need to swap. Proceed to next pair.
544
545 swap:
546 sb t3, 4(a2)
547 lw t4, 5(a2)
548 sb t2, 4(t4)
549
550 sort_next:
551 addi t6, t6, 1
552 lw a2, 5(a2)
553 j while
554
555 recursive_call:
556 add a2, s2, zero      # head pointer
557 addi a3, a3, -1      # n-1
558
559 addi sp, sp, -4
560 sw ra, 0(sp)
561
562 jal Bubblesort      # Bubblesort(arr[], n)
563
564 lw ra, 0(sp)
565 addi sp, sp, 4
566 jr ra

```

```

function bubblesort(head_pointer, n)
  if n=1 then
    return
  else
    arr ← head_pointer
    for (int i=0; i<n-1; i++)
      if category(arr[data]) > category(arr[pahead][data])
        swap(arr[data], arr[pahead][data])
      if category(arr[data]) = category(arr[pahead][data])
        if arr[data] > arr[pahead][data]
          swap(arr[data], arr[pahead][data])
    arr ← arr[pahead]
  bubblesort(head_pointer, n-1)

```

## 4 Registri & Memoria

Di seguito la tabella dei registri e come vengono utilizzate.

register	descrizione dell'utilizzo
s0	<b>listInput</b> – array of commands to be executed
s1	<b>PTAIL</b> – last element pointer
s2	<b>PHEAD</b> – first element pointer
s4	linked list <b>element counter</b>
s5	default <b>LFSR</b>
s6	<b>cmd_status</b> [ 0 - NULL, 65 - ADD, 68 - DEL, 80 - PRINT, 82 - REV, 83 - SORT ]
s7	<b>cmd_format</b> address
s8	<b>DATA</b> – function argument for ADD & DEL
s9	<b>no_error</b> variable – used to trace errors, [ 1 - true, 0 - false ]
s10	<b>listInput pointer value</b>
s11	<b>listInput pointer</b>

register	descrizione dell'utilizzo
a0	used as function's first return value
a1	used as function's second return value

\*\*Si usa la pila(stack), nel caso in cui serve più di due valori di ritorno.

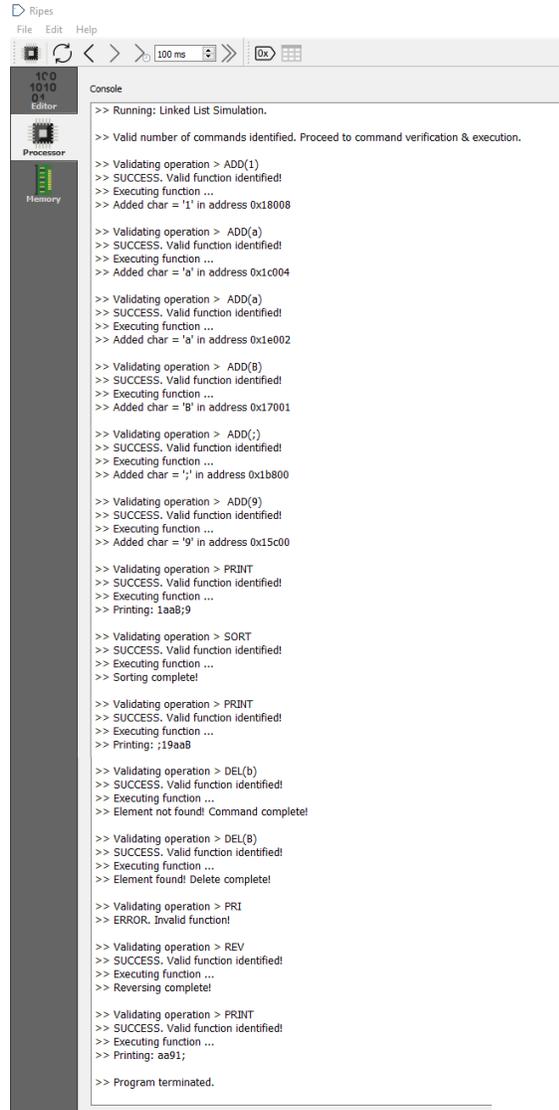
register	descrizione dell'utilizzo
a2 - a6	used as function arguments

# 5 Test

Questi sono i test effettuati per verificare il corretto funzionamento del programma.

## Test #1

**listInput** = "ADD(1) ~ ADD(a) ~ ADD(a) ~ ADD(B) ~ ADD(; ) ~ ADD(9) ~PRINT~SORT~PRINT~DEL(b) ~DEL(B) ~PRI~REV~PRINT"



Address	Word	Byte 0	Byte 1	Byte 2	Byte 3
0x0001b808					
0x0001b804	0x0000000a	a		p	
0x0001b800	0xffffffff	y	y	y	y

Address	Word	Byte 0	Byte 1	Byte 2	Byte 3
0x00017008	0				
0x00017004	0x0000000a		a		a
0x00017000	x	x		.	

Address	Word	Byte 0	Byte 1	Byte 2	Byte 3
0x0001e008	0x0000000a	A			
0x0001e004	0x00000009			9	
0x0001e000	x	x	x		p

Address	Word	Byte 0	Byte 1	Byte 2	Byte 3
0x0001c00c					
0x0001c008	0x00000001	1			
0x0001c004	0x0000000a		a		

Address	Word	Byte 0	Byte 1	Byte 2	Byte 3
0x00018010	y	y			
0x0001800c	0xffffffff	;	y	y	y
0x00018008	0x0000000a		A		

## Test #2

`listInput = " ADD(1) ~ ADD(a) ~ ADD() ~ ADD(B) ~ ADD ~ ADD(9) ~PRINT~SORT(a)~PRINT~DEL(bb) ~DEL(B) ~PRINT~REV~PRINT"`

```

>> Running: Linked List Simulation.
>> Valid number of commands identified. Proceed to command verification & execution.
>> Validating operation > ADD(1)
>> SUCCESS. Valid function identified!
>> Executing function ...
>> Added char = '1' in address 0x18008
>> Validating operation > ADD(a)
>> SUCCESS. Valid function identified!
>> Executing function ...
>> Added char = 'a' in address 0x1c004
>> Validating operation > ADD()
>> ERROR. Invalid function!
>> Validating operation > ADD(B)
>> SUCCESS. Valid function identified!
>> Executing function ...
>> Added char = 'B' in address 0x1e002
>> Validating operation > ADD
>> ERROR. Invalid function!
>> Validating operation > ADD(9)
>> SUCCESS. Valid function identified!
>> Executing function ...
>> Added char = '9' in address 0x17001
>> Validating operation > PRINT
>> SUCCESS. Valid function identified!
>> Executing function ...
>> Printing: 1aB9
>> Validating operation > SORT(a)
>> ERROR. Invalid function!
>> Validating operation > PRINT
>> SUCCESS. Valid function identified!
>> Executing function ...
>> Printing: 1aB9
>> Validating operation > DEL(bb)
>> ERROR. Invalid function!
>> Validating operation > DEL(B)
>> SUCCESS. Valid function identified!
>> Executing function ...
>> Element found! Delete complete!
>> Validating operation > PRINT
>> SUCCESS. Valid function identified!
>> Executing function ...
>> Printing: 1a9
>> Validating operation > REV
>> SUCCESS. Valid function identified!
>> Executing function ...
>> Reversing complete!
>> Validating operation > PRINT
>> SUCCESS. Valid function identified!
>> Executing function ...
>> Printing: 9a1
>> Program terminated.
  
```

Address	Word	Byte 0	Byte 1	Byte 2	Byte 3
0x00017008	0	0			
0x00017004	A9y	y	9	◆	Ä
0x00017000	x	x	y	y	y

Address	Word	Byte 0	Byte 1	Byte 2	Byte 3
0x0001c00c					
0x0001c008	0 0	a	0		0
0x0001c004	0p0	0	p	0	

Address	Word	Byte 0	Byte 1	Byte 2	Byte 3
0x00018010	y	y			
0x0001800c	yyy1	1	y	y	y
0x00018008	0◆	◆	Ä	0	

## Test #3

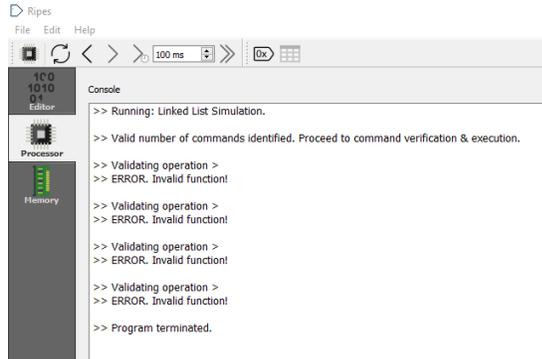
`listInput = ""`

```

>> Running: Linked List Simulation.
>> Valid number of commands identified. Proceed to command verification & execution.
>> Validating operation >
>> ERROR. Invalid function!
>> Program terminated.
  
```

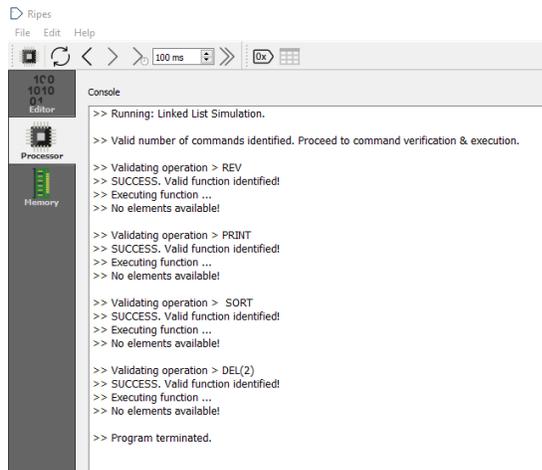
## Test #4

**listInput** = "~::~"



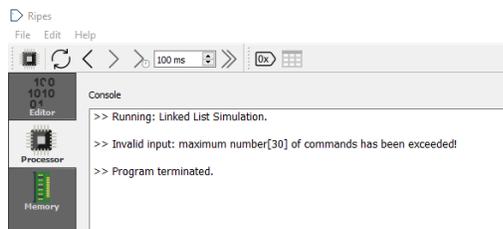
## Test #5

**listInput** = "REV ~PRINT ~ SORT~DEL(2)"



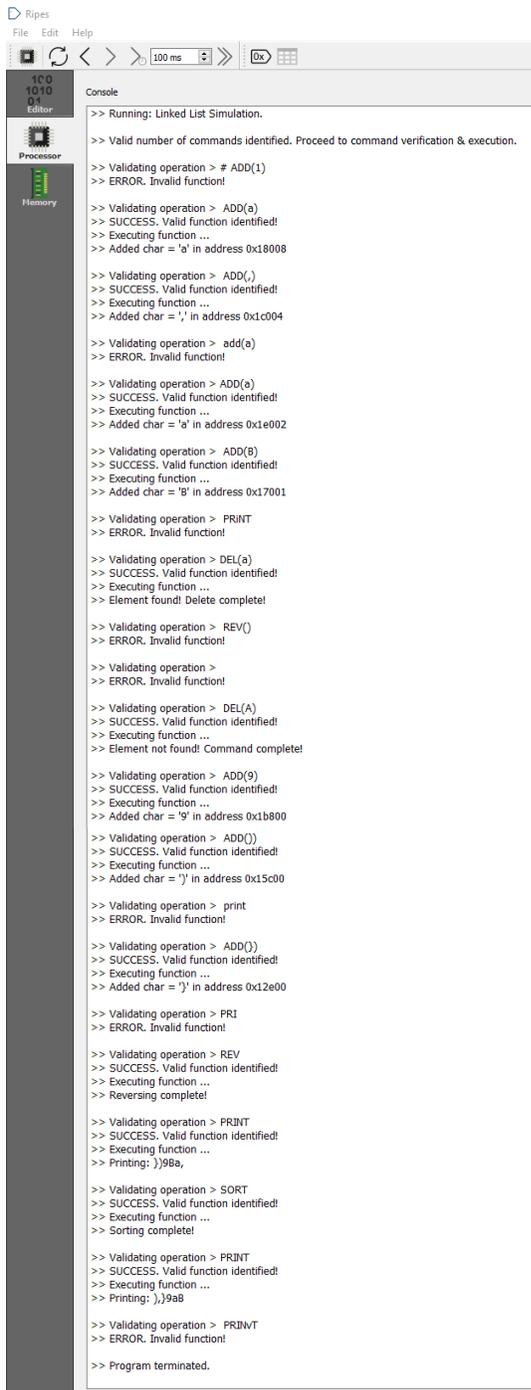
## Test #6

**listInput** = "  
ADD(1)~DEL(2)~add(3)~del(4)~SORT(5)~ADD(6)~DEL(7)~add(8)~del(9)~SORT(10)~DD(11)~DEL(12)~add(13)~del(14)~SORT(15)~ADD(16)~  
DEL(17)~add(18)~del(19)~SORT(20)~aDD(21)~DEL(22)~add(23)~de1(24)~SORT(25)~ADD(26)~DEL(27)~add(28)~de1(29)~SORT(30)~add(31  
)"



## Test #7

```
listInput = "# ADD(1)~ ADD(a)~ ADD(,) ~ add(a) ~ADD(a) ~ ADD(B) ~ PRiNT ~DEL(a) ~ REV() ~ ~ DEL(A) ~ ADD(9)~ ADD() ~  
print ~ ADD()} ~PRI~REV~PRINT ~SORT ~PRINT~ PRiNvT "
```



```
Console  
>> Running: Linked List Simulation.  
>> Valid number of commands identified. Proceed to command verification & execution.  
>> Validating operation > # ADD(1)  
>> ERROR. Invalid function!  
>> Validating operation > ADD(a)  
>> SUCCESS. Valid function identified!  
>> Executing function ...  
>> Added char = 'a' in address 0x18008  
>> Validating operation > ADD(,)   
>> SUCCESS. Valid function identified!  
>> Executing function ...  
>> Added char = ',' in address 0x1c004  
>> Validating operation > add(a)  
>> ERROR. Invalid function!  
>> Validating operation > ADD(a)  
>> SUCCESS. Valid function identified!  
>> Executing function ...  
>> Added char = 'a' in address 0x1e002  
>> Validating operation > ADD(B)  
>> SUCCESS. Valid function identified!  
>> Executing function ...  
>> Added char = 'B' in address 0x17001  
>> Validating operation > PRINT  
>> ERROR. Invalid function!  
>> Validating operation > DEL(a)  
>> SUCCESS. Valid function identified!  
>> Executing function ...  
>> Element found! Delete complete!  
>> Validating operation > REV()  
>> ERROR. Invalid function!  
>> Validating operation > DEL(A)  
>> SUCCESS. Valid function identified!  
>> Executing function ...  
>> Element not found! Command complete!  
>> Validating operation > ADD(9)  
>> SUCCESS. Valid function identified!  
>> Executing function ...  
>> Added char = '9' in address 0x1b800  
>> Validating operation > ADD()  
>> SUCCESS. Valid function identified!  
>> Executing function ...  
>> Added char = ')' in address 0x15c00  
>> Validating operation > print  
>> ERROR. Invalid function!  
>> Validating operation > ADD()  
>> SUCCESS. Valid function identified!  
>> Executing function ...  
>> Added char = ')' in address 0x12e00  
>> Validating operation > PRI  
>> ERROR. Invalid function!  
>> Validating operation > REV  
>> SUCCESS. Valid function identified!  
>> Executing function ...  
>> Reversing complete!  
>> Validating operation > PRINT  
>> SUCCESS. Valid function identified!  
>> Executing function ...  
>> Printing: )9Ba,  
>> Validating operation > SORT  
>> SUCCESS. Valid function identified!  
>> Executing function ...  
>> Sorting complete!  
>> Validating operation > PRINT  
>> SUCCESS. Valid function identified!  
>> Executing function ...  
>> Printing: ),9aB  
>> Validating operation > PRiNvT  
>> ERROR. Invalid function!  
>> Program terminated.
```