Project Assembly RISC-V for the Course of Architetture degli Elaboratori -A.A. 2024/2025-

Gestione di Liste Concatenate

Information

- Author: CHEN Xiong
- Email: xiong.chen@edu.unifi.it
- Matricola:7144793

- Date of sumbit:17/5/2025
- Versione RIPES usata: 2.2.6

Description of Project:

The program is built around a main function Main, which receives the address of listInput, parses the commands, and delegates them to appropriate sub-functions (**ADD, DEL, SORT, PRINT, REV**) based on command type.
In addition to these core functions, there are auxiliary functions like **receive_Operator, clear_inputOperator, getWeight/cancelWeight, trim, next_operator, and receive_parameter**, which support the main logic.
The program follows register usage conventions: ‘a’ registers act as global variables accessible by all functions. ‘s’ registers store values local to a specific function, initialized after pushing the stack. ‘t’ registers are used for temporary values or immediate loading.
The program includes basic error detection. If listInput contains more than 30 commands, execution halts and an error message is displayed.

Definition of constant:

The following command names are predefined as strings:

- ADD: .string "ADD"
- DEL: .string "DEL"
- PRINT: .string "PRINT"
- SORT: .string "SORT"
- REV: .string "REV"

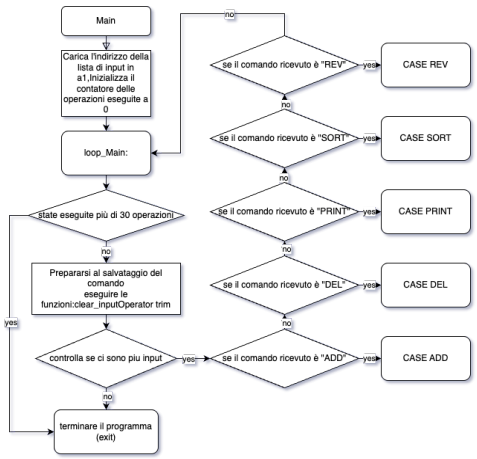
Error Message Definition
The error message (a string) is defined in RAM:

- MsgListInputError: .string "listInput non dovrà contenere più di 30 comandi!"

The following memory addresses are also defined:

- listInput: .string (Used to store the list input)
- inputOperator: .string "AAAAAAAA" (Each command input is limited to 8 bytes or fewer.)
- list:.string "" (Allocated space for the linked list)

Main:



The **Main** function initializes the address of listInput and sets up the registers, including a counter for executed commands. It then enters the **loop_Main**, where it reads each command, matches it to the corresponding sub-function (**ADD, DEL**, etc.), and executes it. If a command is invalid or has already been executed, Main calls next_operator to fetch the next command and repeats the matching process.

Input:

- address of listInput
- single comand

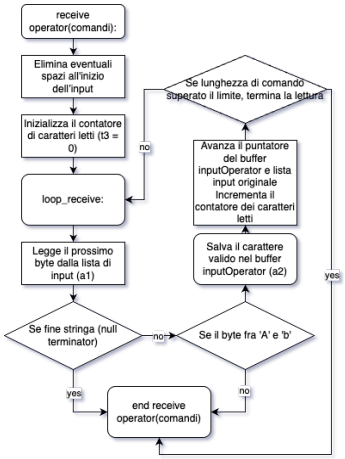
Output:

- Modified linked list

Register & Stack Usage:

- a1: stores the address of listInput
- a2: holds each individual command
- a6: counts the number of executed commands
- t registers: used for immediate values, comparisons, and branching

receive_Operator:



This function reads the listInput and extracts a command (operator) separated by '~'. It first calls trim to remove any leading spaces. Then, it reads bytes from listInput one by one. Reading stops when a null byte is encountered, or the character is outside the range 'A' to 'Z'. The function then returns to **Main** to analyze and execute the command.

Input:

- Address of listInput

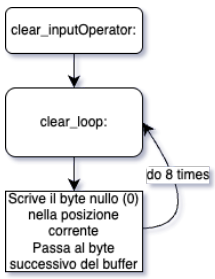
Output:

- Parsed command

Register & Stack Usage:

- a1: address of listInput
- a2: stores the parsed command
- t3: counts the number of characters read
- Other t registers: used for loading bytes from listInput and immediate values

clear_inputOperator:



This simple function clears the contents of **inputOperator** after a command has been executed, preparing it to store the next command.

Input:

- Address of inputOperator (used to store a single command)

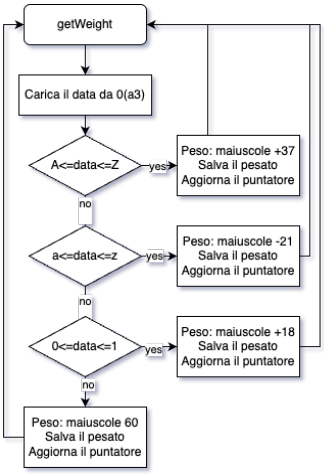
Output:

- Cleared space in inputOperator

Register & Stack Usage:

- t0: holds the address of inputOperator
- t1: pointer used to iterate through inputOperator
- t2: counter for clearing bytes

getWeight/cancelWeight:



These auxiliary functions support the recursive sort function by adjusting character weights to influence sorting order.
getWeight: increases the ASCII value of certain character types (e.g., adds 37 to uppercase letters), so they are sorted after lowercase letters.
cancelWeight: reverses the transformation, restoring original character values after sorting.
The functions classify characters by checking their ASCII ranges and apply different weights accordingly.
Input:

- Address of the linked list

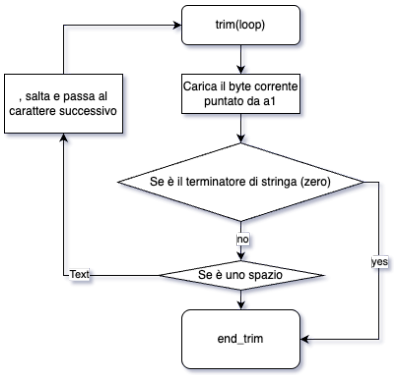
Output:

- Modified list after getWeight or restored list after cancelWeight

Register & Stack Usage:

- t1, t2: hold immediate values for ASCII range boundaries
- t3: pointer to traverse the linked list

trim:



This function skips leading spaces in listInput. If the current character is a space (' '), the pointer advances until a non-space character is found. It then returns the address of that character.
Input:

- Address of listInput

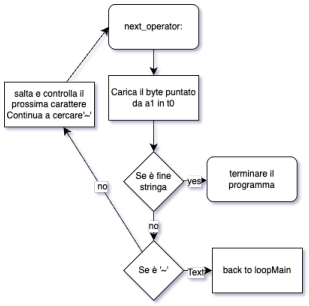
Output:

- Address of the first non-space character

Register & Stack Usage:

- a3: pointer to listInput
- t0: loads the currently pointed byte
- t1: holds the ASCII code for space (' ')

next_operator:



This function advances the pointer to the next command in listInput by scanning for the delimiter '~', which signals the start of a new command.
Input:

- Address of listInput

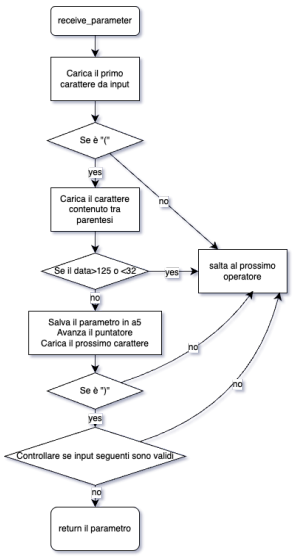
Output:

- Pointer to the next command

Register & Stack Usage:

- a1: pointer to listInput
- t0: loads the current byte
- t2: holds the ASCII code for '~'

receive_parameter:



This auxiliary function for **DEL** and **ADD** reads and validates parameters following a command. It checks if the parameter is enclosed in parentheses and within a valid range.

Input:

- Address of listInput

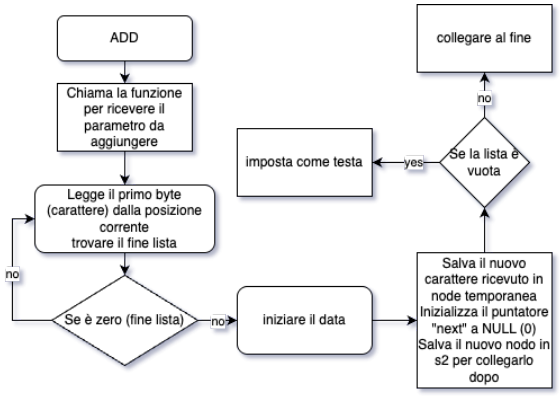
output:

- Parameter extracted from the command

Register & Stack Usage:

- s0, s1: ASCII codes for left and right parentheses
- t0: loads characters from listInput
- a5: stores the extracted parameter

ADD:



This function executes the **ADD** command by receiving a parameter (via **receive_parameter**) and adding it to the linked list. If the linked list is empty, it initializes a3 to point to the new node.

Input:

- Address of listInput
- Parameter to add
- Address of linked list

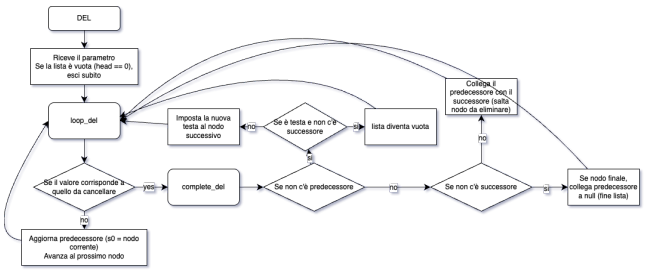
Output:

- Updated linked list with added data

Register & Stack Usagek:

- s0: address of linked list
- s1: address of linked list head
- t0: loads bytes from linked list

DEL:



This function deletes all nodes in the linked list matching the given parameter. It iterates through the list, tracking the predecessor and successor nodes. Upon finding a matching node, it removes it and reconnects the list based on the presence of predecessor and successor. The process repeats until the end of the list.

Input:

- Address of linked list
- Parameter to delete

Output:

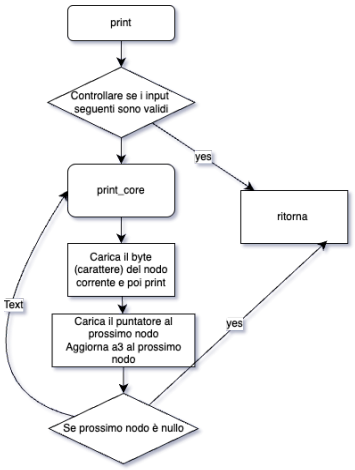
- Updated linked list after deletion

Register & Stack Usage:

- a5: parameter value

- s0: predecessor node address
- s1: current node address
- s2: successor node address
- t registers: load characters from linked list

PRINT:



This recursive function prints the data at the address pointed to by a3, then advances a3 to the next node and calls itself again. It stops when a3 is null.

Input:

- Address of linked list

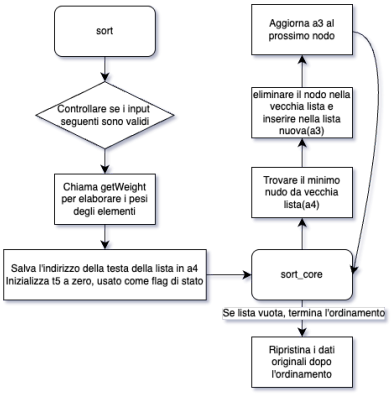
Output:

- Prints all data from the linked list

Register & Stack Usage:

- a3: pointer to the current node
- t0: loads data from the current node
- t1: points to the next node

SORT:



This recursive function implements insertion sort on a linked list. It uses a3 to build a new sorted list and a4 to scan the old list. The function repeatedly finds the minimum character in a4, deletes it from a4, and appends it to a3. The process repeats until a4 is empty, then the function returns.

Input:

- Address of linked list

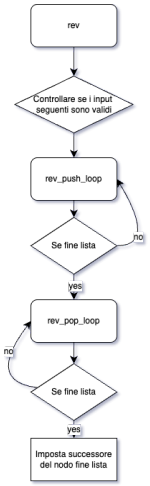
Output:

- Sorted linked list

Register & Stack Usage:

- a3: address of the new (sorted) linked list
- a4: address of the old (unsorted) linked list
- t5: flag indicating existence of head node
- t registers: load bytes from listInput or immediate values

REV:



This function reverses the linked list using a stack and its LIFO principle. It pushes each character onto the stack while traversing the list, then pops them off to rebuild the list in reverse order.

Input:

- Address of linked list

Output:

- Reversed linked list

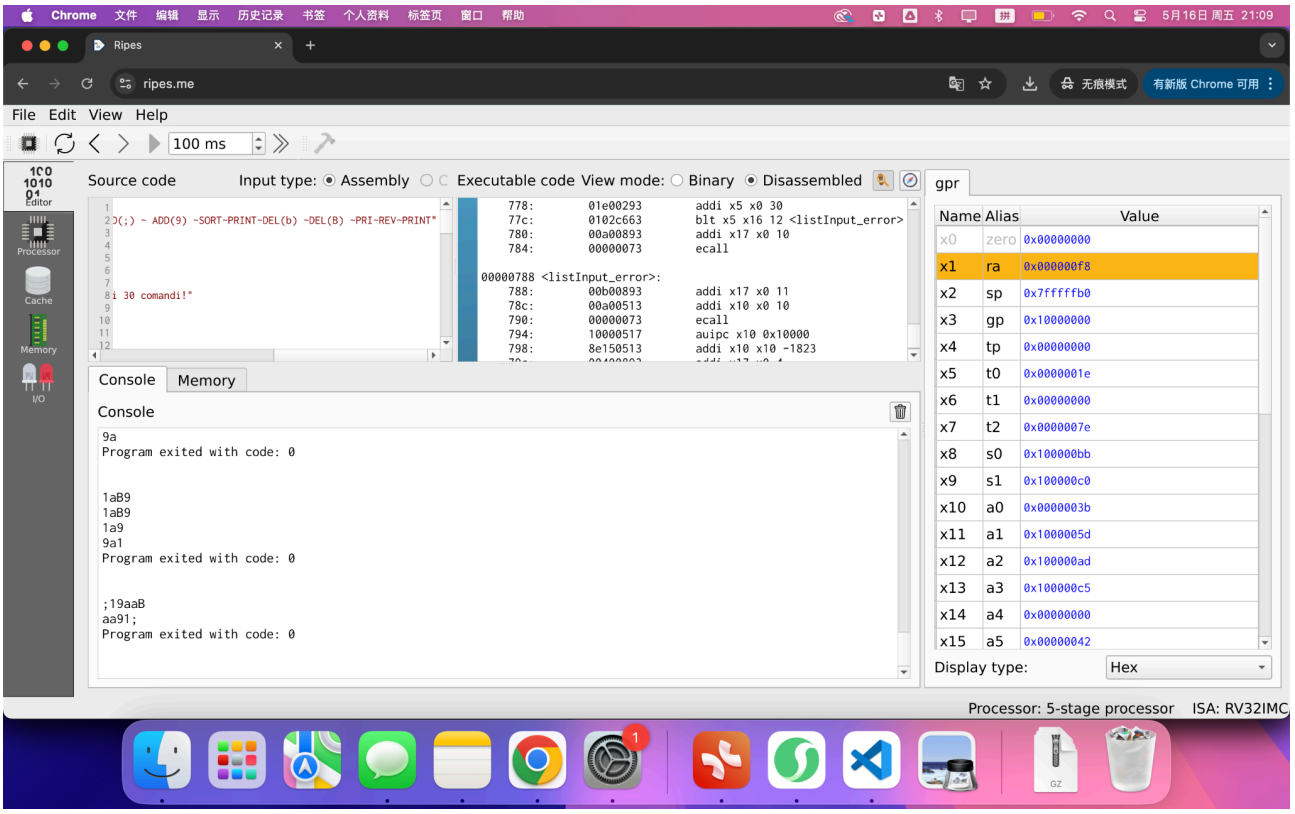
Register & Stack Usage:

- t registers: used as pointers to traverse the linked list

TEST:

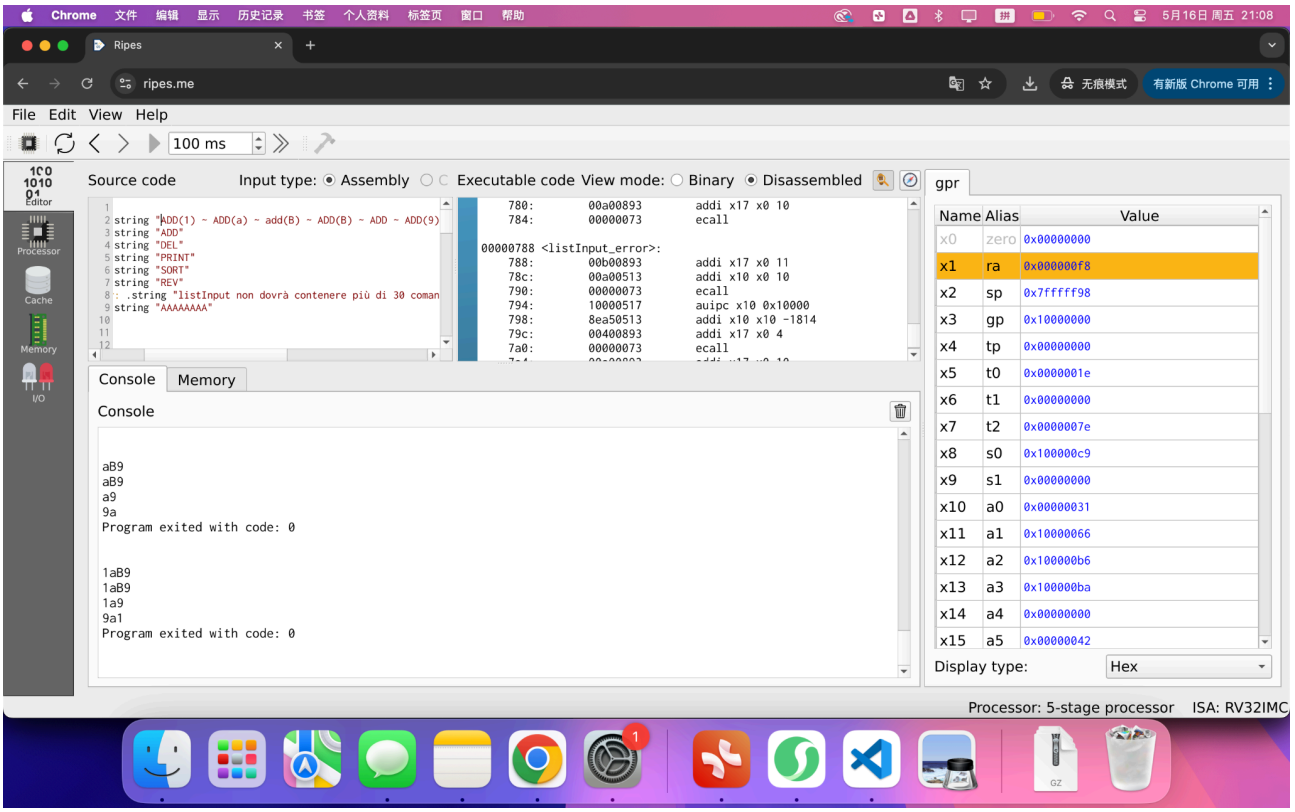
listInput = “ADD(1) ~ ADD(a) ~ ADD(a) ~ ADD(B) ~ ADD(;) ~ ADD(9) ~SORT~PRINT~DEL(b) ~DEL(B) ~PRI~REV~PRINT”

Comando Corrente	ADD(1)	ADD(a)	ADD(a)	ADD(B)	ADD(;)	ADD(9)	SORT	PRINT	DEL(b)	DEL(B)	PRI	REV	PRINT
Elementi in Lista	1	1a	1aa	1aaB	1aaB;	1aaB;9	;19aaB	;19aaB	;19aaB	;19aa	;19aa	aa91;	aa91;



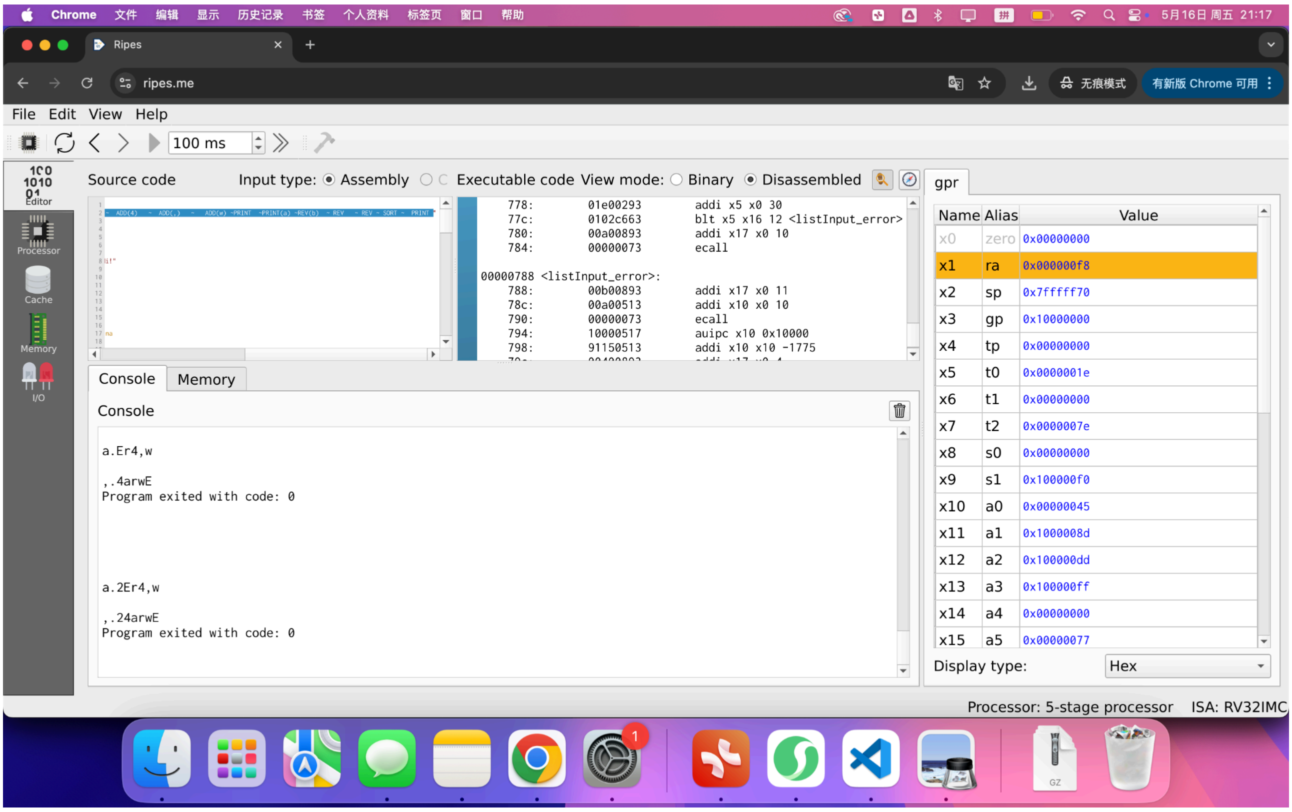
listInput = “ADD(1) ~ ADD(a) ~ add(B) ~ ADD(B) ~ ADD ~ ADD(9) ~PRINT~SORT(a)~PRINT~DEL(bb) ~DEL(B) ~PRINT~REV~PRINT”

Comando Corrente	ADD(1)	ADD(a)	add(B)	ADD(B)	ADD	ADD(9)	PRINT	SORT(a)	PRINT	DEL(bb)	DEL(B)	PRINT	REV	PRINT
Elementi in Lista	1	1a	1a	1aB	1aB	1aB9	1aB9	1aB9	1aB9	1aB9	1a9	1a9	9a1	9a1



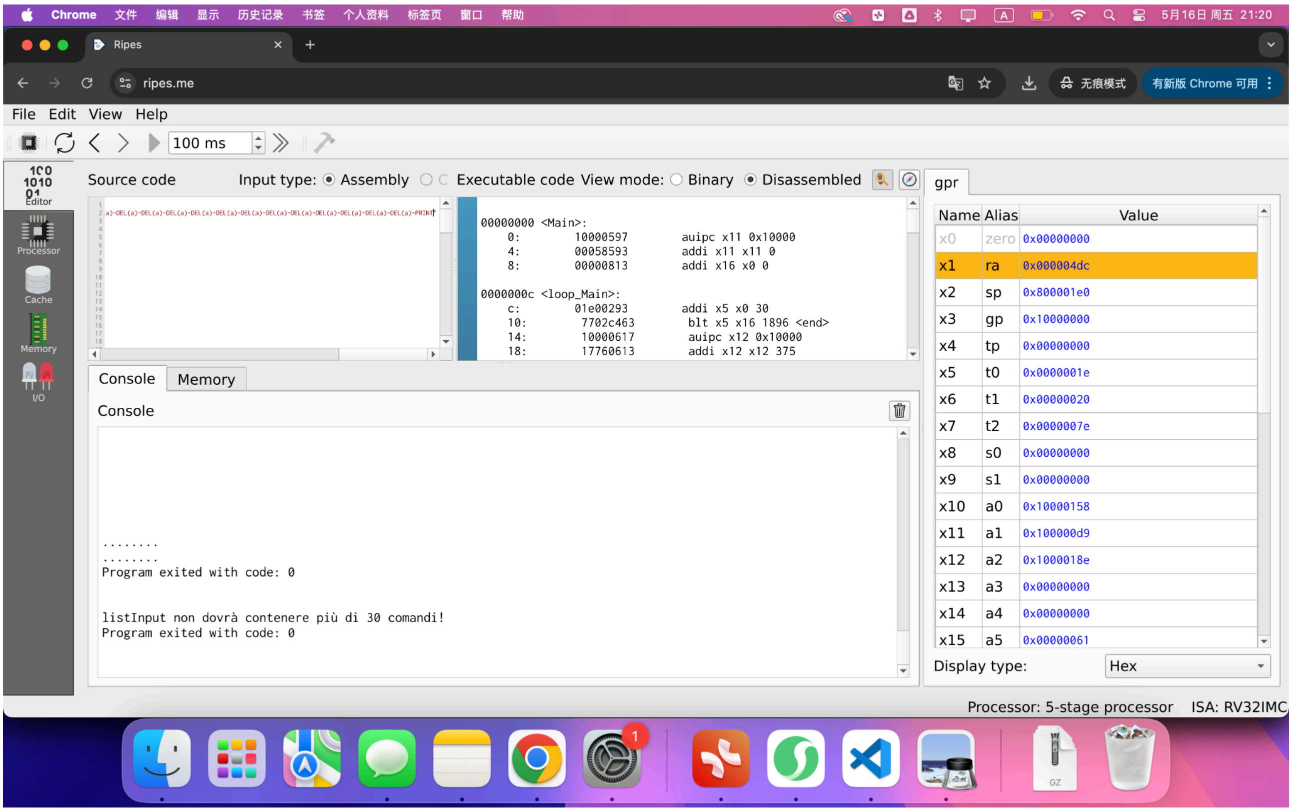
listInput = “ADD(a) ~ ADD(.) ~ ADD(2) ~ ADD(E) ~ ADD(r) ~ ADD(4) ~ ADD(,) ~ ADD(w) ~PRINT ~PRINT(a) ~REV(b) ~ REV ~ REV ~ SORT ~ PRINT”

ComandoCorrente	ADD(a)	ADD(.)	ADD(2)	ADD(E)	ADD(r)	ADD(4)	ADD(,)	ADD(w)	PRINT	PRINT(a)	REV(b)	REV	REV	SORT	PRINT
Elementi in Lista	a	a.	a.2	a.2E	a.2Er	a.2Er4	a.2Er4,	a.2Er4,w	a.2Er4,w	a.2Er4,w	a.2Er4,w	w,4rE2.a	w,4rE2.a	a.2Er4,w	,.24arwE



listInput = "DEL(a)~PRINT"

Expected to display ErrorMessage



listInput = “ADD(a) ~ add(a) ~ ~ ADD(c) ~ ADDS(v) ~ ADD({) ~DELA({) ~ REV PRINT ~ SORT~ PRINT”

Comando Corrente	ADD(a)	add(a)		ADD(c)	ADD(v)	ADD({)	DELA({)	REV PRINT	SORT	PRINT
Elementi in Lista	a	a	a	ac	acv	ac{	ac{	ac{	{ac	{ac