

# Relazione RISC-V LinkedList

Claudio Raimondi  
Matricola: 7158162  
claudio.raimondi@edu.unifi.it  
Corso: Architettura degli Elaboratori  
Docente: Prof. Zoppi

Data: 3 Giugno 2025

## Contents

<b>1</b>	<b>Overview generale</b>	<b>1</b>
1.1	Obiettivi del progetto . . . . .	1
1.2	Approccio ed Ambiente di sviluppo . . . . .	1
1.3	Uso della memoria e dei registri . . . . .	1
1.4	Design . . . . .	1
<b>2</b>	<b>Funzioni chiave</b>	<b>2</b>
2.1	Parsing . . . . .	2
2.2	Add . . . . .	2
2.3	Malloc . . . . .	2
2.4	Sort . . . . .	2
<b>3</b>	<b>Analisi quantitativa e ottimizzazione</b>	<b>3</b>
3.1	Cache . . . . .	3
3.2	Branches . . . . .	4
<b>4</b>	<b>Riferimenti</b>	<b>4</b>

# 1 Overview generale

## 1.1 Obiettivi del progetto

Il progetto consiste nell'implementazione di una **linked list** in assembly **RISC-V** 32-bit. La lista supporta operazioni di **inserimento**, **cancellazione**, **ordinamento**, **stampa** e **inversione** su elementi di tipo intero, ed è progettata per essere il più efficiente possibile mantenendo allo stesso tempo una buona modularità.

## 1.2 Approccio ed Ambiente di sviluppo

Per semplificare lo sviluppo ed il testing ho svolto il ruolo del compilatore: sviluppando prima un progetto funzionante in **C**, e successivamente traducendolo meticolosamente in assembly RISC-V. In questo modo ho ridotto al minimo la possibilità di verificarsi di errori umani, e ho potuto apprendere il punto di vista del compilatore.

## 1.3 Uso della memoria e dei registri

Per evitare conflitti tra registri, ho seguito regole di convenzione sull'uso dei registri, in particolare:

- **a0-a7** sono utilizzati per i parametri delle funzioni e per il valore di ritorno. (**a0** è il valore di ritorno e **a7** è utilizzato per le chiamate di sistema)
- **s0-s11** sono riservati per il *chiamante*.
- **t0-t6** sono utilizzati per variabili temporanee e si assume che siano disponibili per l'uso dal *chiamato*.
- **sp** è il puntatore allo stack.
- **ra** è il registro dell'indirizzo di ritorno.
- **zero** è il registro che contiene sempre il valore zero.

Inoltre dove necessario ho salvato i registri utilizzati nello stack, e li ho ripristinati al termine della funzione.

## 1.4 Design

Per rendere il codice il più modulare possibile, e per semplificare il **parsing**, ho adottato un approccio con array di funzioni, che in assembly si traduce come un array di indirizzi ad etichette:

```
1 list_functions:
2     .word list_add
3     .word list_del
4     .word list_print
5     .word list_sort
6     .word list_rev
```

## 2 Funzioni chiave

### 2.1 Parsing

Il parsing dell'input è stato implementato in 3 fasi:

- **Tokenizzazione:** l'input viene diviso in-place in token, sostituendo i delimitatori con il carattere '\0'.
- **Sanificazione:** per ogni token, viene verificato che corrisponda ad un comando valido, altrimenti viene ignorato.
- **Esecuzione:** per ogni token valido, viene chiamata la funzione corrispondente nell'array.

### 2.2 Add

La funzione `list_add` alloca e aggiunge un nuovo nodo in coda alla lista. Grazie al puntatore alla coda, l'inserimento è scalabile e sarà sempre costante con complessità  $O(1)$ . La chiamata a `malloc` non è gestita dal kernel come in C, ma deve essere implementata manualmente.

### 2.3 Malloc

Per implementare la funzione di allocazione dinamica della memoria, ho simulato un heap tramite una regione di memoria predefinita, che viene gestita tramite una free-list.

```
1 .data
2
3 node_size: .byte 5
4 max_nodes: .byte 30
5
6 mempool: .space 150
7 free_list: .space 30
```

Sapendo a priori la dimensione massima della lista, e la dimensione di ogni nodo, ho allocato un'area di memoria di 150 byte, che contiene 30 nodi da 5 byte ciascuno. La `free_list` invece è un array che contiene gli indici dei nodi liberi.

Per la ricerca vera e propria del primo nodo libero, per mantenere una certa semplicità ho optato per una ricerca lineare con complessità  $O(n)$ .

### 2.4 Sort

Per la funzione di ordinamento ho scelto l'algoritmo **Merge Sort** per 2 motivazioni principali:

- complessità sempre  $O(n \log n)$ , senza degenerare.
- mancanza di alternative per una lista concatenata, che non permette l'accesso casuale agli elementi.

Per la ricerca del punto medio della lista, ho utilizzato la tecnica dei **puntatori lento e veloce**, che dimezza la complessità rispetto ad una ricerca lineare:

```
1 t_node *slow = *head;
2 t_node *fast = *head;
3 while (fast->next && fast->next->next)
4 {
5     slow = slow->next;
6     fast = fast->next->next;
7 }
8 t_node *mid = slow->next;
```

La funzione merge è stata implementata in modo ricorsivo e totalmente *branchless* grazie all'utilizzo di variabili booleane come indici di array:

```
1 while ((a != NULL) & (b != NULL))
2 {
3     idx = (b->data < a->data);
4
5     chosen = nodes[idx];
6     current->next = chosen;
7     current = chosen;
8     nodes[idx] = chosen->next;
9     a = nodes[0];
10    b = nodes[1];
11 }
```

Questa, anche se sembra una scelta facile, in realtà potrebbe essere subottimale rispetto ad una soluzione branchy, a causa della latenza delle operazioni di load e store. In conclusione però, la predittibilità della versione branchless rimuove il rischio di branch mispredictions che aggiungevano ben più stalli alla pipeline.

## 3 Analisi quantitativa e ottimizzazione

### 3.1 Cache

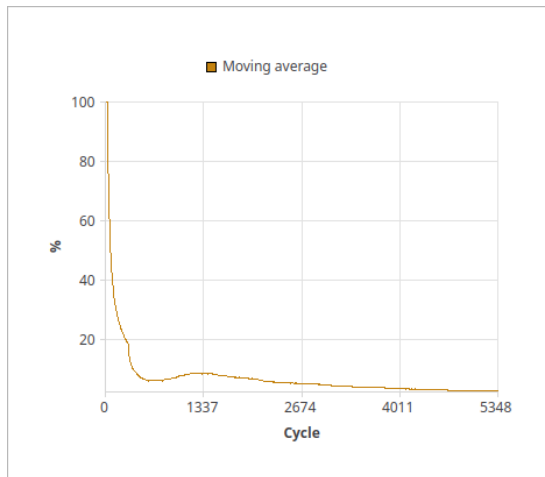
Per ottimizzare la località spaziale delle istruzioni, così da ridurre il numero di cache miss, le ho riordinate tenendo in mente diversi fattori:

- le funzioni chiamate più frequentemente hanno priorità più alta
- funzioni che si chiamano a vicenda sono vicine tra loro
- l'utente è più propenso a chiamare le funzioni **add** e **print**
- il branch predictor può favorire salti all'indietro

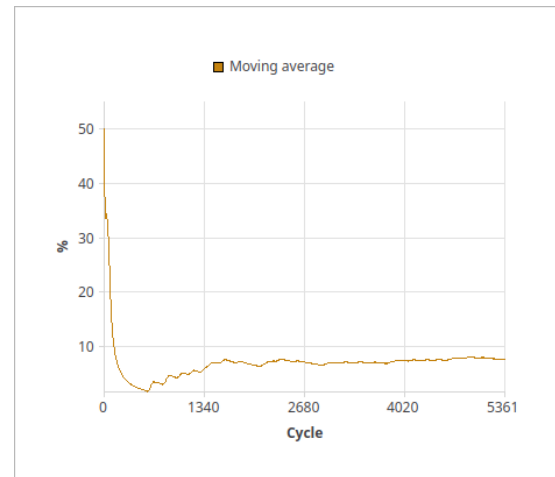
Eseguendo un semplice benchmark con il comando:

$$\text{ADD}(1) \sim \text{SORT} \sim \text{REV} \sim \text{DEL}(1) \sim \text{ADD}(1) \sim \text{PRINT}$$

otteniamo un *cache miss* rate di circa 8% per le **istruzioni**, e 2% per i **dati**, indipendentemente dal tipo di cache:



(a) Data Cache Miss



(b) Instruction Cache Miss

## 3.2 Branches

Per ridurre al minimo il numero di branch mispredictions, e quindi di stalli, ho utilizzato tecniche branchless:

- sacrificare lo short-circuiting utilizzando **operatori bitwise** invece di operatori logici (& e | invece di && e ||)
- utilizzare variabili booleane come indici di array, come visto nella funzione `merge`
- moltiplicazione per 0 e 1 (condizione e condizione opposta)

In questo modo, l'intero codice ha solo 24 salti condizionati.

## 4 Riferimenti

- [Codice sorgente su GitHub](#)
- [Simulatore RIPES](#)
- [RISC-V Foundation](#)