RISC-V LinkedList Report

Claudio Raimondi
Student ID: 7158162
claudio.raimondi@edu.unifi.it
Course: Computer Architecture
Professor: Prof. Zoppi

Date: June 3, 2025

Contents

1 General Overview

1.1 Project Objectives

The project consists of implementing a **linked list** in 32-bit **RISC-V** assembly. The list supports operations for **insertion**, **deletion**, **sorting**, **printing**, and **reversal** on integer type elements, and is designed to be as efficient as possible while maintaining good modularity.

1.2 Approach and Development Environment

To simplify development and testing, I took on the role of the compiler: first developing a working project in **C**, and then meticulously translating it to RISC-V assembly. This way I minimized the possibility of human errors occurring and was able to learn the compiler's perspective.

1.3 Memory and Register Usage

To avoid register conflicts, I followed register usage convention rules, in particular:

- `a0-a7` are used for function parameters and return values. (`a0` is the return value and `a7` is used for system calls)
- `s0-s11` are reserved for the *caller*.
- `t0-t6` are used for temporary variables and are assumed to be available for use by the *callee*.
- `sp` is the stack pointer.
- `ra` is the return address register.
- `zero` is the register that always contains the value zero.

Additionally, where necessary, I saved the used registers on the stack and restored them at the end of the function.

1.4 Design

To make the code as modular as possible and to simplify **parsing**, I adopted an approach with function arrays, which in assembly translates to an array of addresses to labels:

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

2 Key Functions

2.1 Parsing

Input parsing was implemented in 3 phases:

- **Tokenization**: the input is divided in-place into tokens, replacing delimiters with the '\0' character.
- **Sanitization**: for each token, it is verified that it corresponds to a valid command, otherwise it is ignored.
- **Execution**: for each valid token, the corresponding function in the array is called.

2.2 Add

The `list_add` function allocates and adds a new node at the tail of the list. By maintaining a pointer to the tail of the list, insertion is scalable and will always be constant with $O(1)$ complexity. The `malloc` call is not handled by the kernel as in C, but must be implemented manually.

2.3 Malloc

To implement the dynamic memory allocation function, I simulated the heap through a predefined memory region, which is managed through a 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
```

Knowing in advance the maximum size of the list and the size of each node, I allocated a memory area of 150 bytes, which contains 30 nodes of 5 bytes each. The `free_list` instead is an array that contains the indices of free nodes.

For the actual search for the first free node, to maintain some simplicity I opted for a linear search with $O(n)$ complexity.

2.4 Sort

For the sorting function I chose to implement the **Merge Sort** algorithm for 2 main reasons:

- always $O(n \log n)$ complexity, without degrading.
- lack of alternatives for a linked list, which does not allow random access to elements.

To search for the midpoint of the list, I used the **slow and fast pointers** technique, which halves the complexity compared to a linear search:

```
t_node *slow = *head;
t_node *fast = *head;
while (fast->next && fast->next->next)
{
  slow = slow->next;
  fast = fast->next->next;
}
t_node *mid = slow->next;
```

The merge function was implemented recursively and completely *branchless* thanks to the use of boolean variables as array indices:

```
while ((a != NULL) & (b != NULL))
{
  idx = (b->data < a->data);

  chosen = nodes[idx];
  current->next = chosen;
  current = chosen;
  nodes[idx] = chosen->next;
  a = nodes[0];
  b = nodes[1];
}
```

This, although it seems like an easy choice, might actually be suboptimal compared to a branchy solution, due to the latency of load and store operations. In conclusion however, the predictability of the branchless version removes the risk of branch mispredictions that added far more pipeline stalls.

3 Quantitative Analysis and Optimization

3.1 Cache

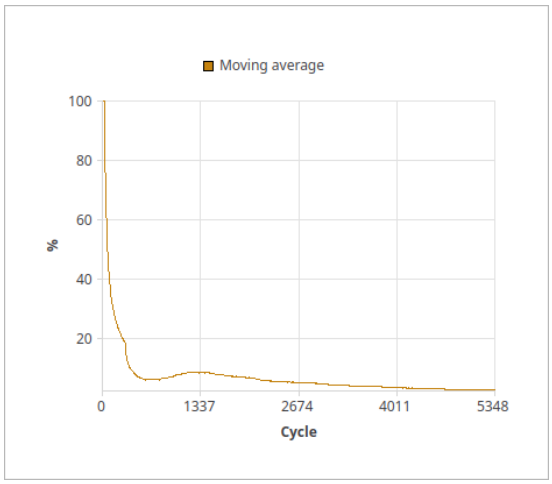
To optimize the spatial locality of instructions, so as to reduce the number of cache misses, I reordered them keeping several factors in mind:

- more frequently called functions have higher priority
- functions that call each other are close to each other
- the user is more likely to call the `add` and `print` functions
- the branch predictor can favor backward jumps

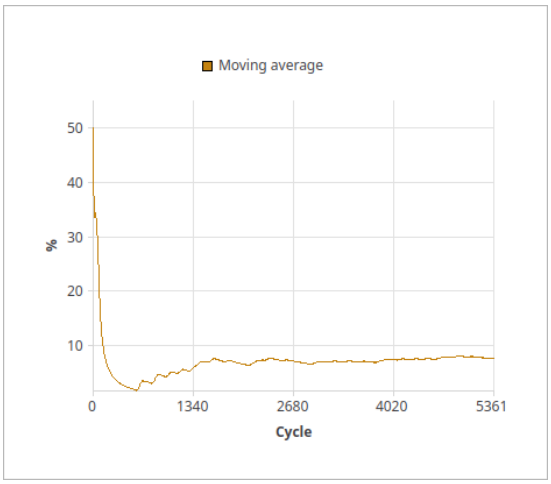
Running a simple benchmark with the command:

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

we get a *cache miss* rate of about 8% for **instructions**, and 2% for **data**, regardless of cache type:



(a) Data Cache Miss



(b) Instruction Cache Miss

3.2 Branches

To minimize the number of branch mispredictions, and therefore stalls, I used branchless techniques:

- sacrifice short-circuiting by using **bitwise operators** instead of logical operators (`&` and `|` instead of `&&` and `||`)
- use boolean variables as array indices, as seen in the `merge` function
- multiplication by 0 and 1 (condition and opposite condition)

This way, the entire code has only 24 conditional jumps.

4 References

- Source code on GitHub
- RIPES Simulator
- RISC-V Foundation