Optimization Report: Newlib RISC-V `memset` Function

Atal Gupta

February 9, 2025

1 Introduction

This report describes the process and techniques used to optimize the `memset` function for the RISC-V port of Newlib. Newlib is a widely used C library in embedded systems, and optimizing its functions for RISC-V can result in significant improvements in both static code size and execution performance. The main focus of this project was to optimize the `memset` routine, given that profiling indicated approximately 90% of calls occur with the byte count being a multiple of 4. The objective was to reduce the number of instructions (and thereby cycles) required without greatly increasing the overall code size.

2 Implementation Overview

Two versions of the `memset` function have been implemented:

- **memset_original.s**: A baseline implementation that fills memory in a byte-by-byte manner.
- **memset_optimized.s**: An enhanced version that leverages word-level (32-bit) stores when the fill size is a multiple of 4, while also handling misaligned memory addresses.

The optimized version first checks the alignment of the destination pointer. If the pointer is not 4-byte aligned, it processes a misaligned prefix with byte stores until alignment is achieved. Then, it builds a 32-bit word with the fill byte replicated across all four bytes and uses word stores to fill the bulk of the memory. Finally, any remaining bytes (if the total size is not an exact multiple of 4) are filled with additional byte stores.

3 Optimization Details

The following key optimizations were applied:

1. **Alignment Check:** The code uses the `andi` instruction to determine if the destination pointer is aligned to a 4-byte boundary. This ensures that subsequent word stores are performed on properly aligned addresses.

2. **Misaligned Prefix Handling:** If the destination pointer is misaligned, a loop is executed to store individual bytes until the pointer becomes aligned. This avoids potential penalties or issues with misaligned 32-bit accesses.

3. **Word-Store Loop:** Once aligned, the code constructs a 32-bit word by shifting the fill byte into the correct positions and combining them with `or` instructions. This allows the function to write four bytes at a time using the `sw` instruction, greatly reducing the number of instructions executed for large memory areas.

4. **Tail Processing:** For any remaining bytes that do not fit into a 32-bit word, a final loop performs byte-wise stores to complete the operation.

These optimizations reduce the overall instruction count, especially in cases where the fill size is large and a multiple of 4. This directly translates into lower cycle counts on RISC-V processors, while still maintaining a compact code footprint.

4 Performance Considerations

The optimized implementation offers several performance benefits:

- **Instruction Reduction:** By processing data in 32-bit chunks instead of byte-by-byte, the number of store instructions is significantly reduced. In ideal cases, the number of stores can drop by up to 75% compared to the baseline.
- **Cycle Count Improvement:** With fewer instructions executed, the cycle count is also reduced. This is particularly beneficial in embedded systems where every cycle matters.
- **Minimal Code Size Increase:** The additional instructions required to handle alignment and word-level operations are kept to a minimum to ensure that the overall code size remains small.

5 Repository and Source Code

The complete source code for this project is hosted on GitHub. It includes:

- `memset_original.s` — Baseline byte-by-byte implementation.
- `memset_optimized.s` — Optimized version with alignment handling and 32-bit word stores.
- `test_memset.c` — A comprehensive test suite that verifies correctness across various scenarios.

The GitHub repository is available at:
`https://github.com/AtalGupta/RISCV_Newlib_Coding_Challenge`

6 Conclusion

The optimizations implemented in the `memset` function demonstrate a significant reduction in instruction count and cycle usage, particularly for cases where the number of bytes to set is a multiple of 4. By incorporating alignment checks, misaligned prefix handling, and word-level operations, the optimized function is well-suited for resource-constrained RISC-V embedded systems. Future work may explore further enhancements using additional RISC-V extensions to improve performance and reduce code size even further.