Table of Contents

Introduction ***3***

Exercise 1 ..... ***4***

Exercise 2 ..... ***5***

Differences Between Original and Extended RISC-V Instructions 5

Cycle Measurement Results (Table 3) 7

Advantages of Extensions 7

Disadvantages of Extensions 7

Creative Aspects of Improvement 8

Is the Extension Worthwhile for a 24x24 Screen? 8

Conclusion ***9***

Introduction

Microprogramming is a powerful tool that extends traditional assembly language by allowing the creation of specialized instructions tailored to common tasks. These custom instructions help streamline operations, significantly boosting performance. In practical applications, this optimization saves both time and money, as microprogrammed functions are designed with specific tasks in mind, eliminating the inefficiencies of more generic instruction sets. By focusing on the unique needs of each application, microprogramming ensures critical tasks are completed faster and more efficiently, making it an essential technique in fields ranging from embedded systems to high-performance computing.

This project illustrates the value of microprogramming by using a traffic light simulation system as a real-world example. It involves creating custom microcode instructions to optimize tasks like filling and updating memory blocks, showing how microprogramming can reduce clock cycles for complex operations. By comparing processes with and without these custom instructions, the project highlights the performance benefits and design flexibility of microcode extension. The traffic light system provides an accessible context to explore these ideas and demonstrates how theoretical microprogramming concepts lead to practical optimized solutions.

The report is organized to offer a clear and thorough overview of the work; The first section explains the microcode implementation, including the design of custom instructions like `vfill` and `vadd` that improve memory manipulation, The second section covers the assembly implementation, showing how these microcode extensions enhance standard assembly code Finally, the report compares performance in terms of clock cycles with and without the microcode extensions, emphasizing the effectiveness of these custom instructions in optimizing operations. This approach not only showcases the depth of microprogramming but demonstrates its real-world impact through concrete, quantitative analysis.

Exercise 1

This exercise provides a detailed exploration of how the microcode is structured, including fetch, decode, and execute stages, and introduces specialized instructions such as ***vfill*** and ***vadd*** to streamline repetitive tasks. Through this, the groundwork is laid for achieving better clock cycle efficiency and demonstrating the practical benefits of microcode extensions.

Instruction Name	RT Language Design	Control Signals per Cycle	Design Decisions
lx reg1 u32	Cycle 1: MAR ← PC Cycle 2: MBR ← Mem[MAR], PC ← PC + 4 Cycle 3: RF[reg1] ← MBR	Cycle 1: (T2, C0) Cycle2: (TA, R, BW=11, M1, C1, M2, C2) Cycle3: (T1, LC, SELC=10101, MR=0)	Efficiently loads a 32-bit immediate value (u32) into the specified register by leveraging memory access and program counter incrementation.
call u32	Cycle 1: RT1 ← PC Cycle 2: SP ← SP + 4, MAR ← SP Cycle 3: MBR ← PC, Mem[MAR] ← MBR Cycle 4: PC ← RT1	Cycle 1: (T2, C0) Cycle2: (SELA=00010, SELC=00010, MR=1, MB=10, MC=1, SELCOP=1011, T6, LC, C0) Cycle3: (T2, M1=0, C1; T4, M2=0, C2, A0=1, B=1, C=0)	Implements a subroutine call by saving the current program counter to the stack and jumping to the target address. Designed to ensure stack integrity for nested calls.
return	Cycle 1: MAR ← SP Cycle 2: MBR ← Mem[SP] Cycle 3: PC ← MBR Cycle 4: SP ← SP + 4	Cycle1: (SELA=00010, MR=1, T9=1, C0=1) Cycle2: (TA=1, R, BW=11, M1=1, C1=1; SELA=00010, MR=1, MA=0, MB=10, SELCOP=1010, MC=1, T6=1, LC) Cycle3: (T1=1, M2=0, C2=1, A0=1, B=1, C=0)	Enables the program to return from a subroutine by restoring the program counter from the stack. Prioritizes stack cleanliness to avoid address corruption.
stop	Cycle 1: SP ← 0x00 Cycle 2: PC ← 0x00	Cycle1: (SELA=00000, SELC=00010, T9, LC=1, MR=1; SELA=00000, T9, M2=0, C2=1, A0=1, B=1, C=0)	Halts execution by resetting the stack pointer and program counter. Simplifies program termination.

vfill reg1 (reg2) val1	Cycle 1: RT1 ← RF[reg1], RT2 ← RF[reg2] Cycle 2: Mem[RT2] ← val1, RT1 ← RT1 - 1 Cycle 3: RT2 ← RT2 + 1 Cycle 4: Repeat if RT1 > 0, else exit	Cycle1: (SELA=10000, T9, C4; OFFSET=0, SIZE=01000, T3, C1; SELC=10101, MC=1) Cycle2: (Td, W, BW=0; SELA=10101, SELC=10101, MC=1, SELCOP=1011, T6, LC, SELP=11) Cyclej3: (A0=0, B=1, C=0, MADDR=loop)	Implements a memory fill operation with a specific value, looping through addresses starting from reg2. Highly efficient for block initialization.
vadd reg1 (reg2) reg3	Cycle 1: RT1 ← RF[reg1], RT2 ← RF[reg2], RT3 ← RF[reg3] Cycle 2: Temp ← Mem[RT2] + RT3, Mem[RT2] ← Temp Cycle 3: RT1 ← RT1 - 1, RT2 ← RT2 + 1 Cycle 4: Repeat if RT1 > 0, else exit	Cycle1: (SELA=10000, SELB=10101, SELC=10011, MR=0, T9, C4) Cycle2: (TA, R, BW=0, M1=1, SELCOP=1010, T6, M1=0, C1; Td, Ta, W, BW=0; SELA=10101, SELC=10101, MC=1, SELCOP=1011, T6, LC, SELP=11) Cycle3: (A0=0, B=1, C=0, MADDR=loop)	Adds a constant value to a range of memory addresses, iteratively updating them. Optimized for repetitive mathematical operations on memory.

Exercise 2

This section delves into a detailed comparison between the baseline RISC-V instruction set and the custom microprogramming extensions introduced in Exercise 1. Beyond performance gains, this analysis explores conceptual depth, creative design strategies, and their broader implications on system architecture and real-world applications.

Differences Between Original and Extended RISC-V Instructions

The original RISC-V instruction set relies on its generic architecture, which processes all instructions sequentially without specific optimization for repetitive tasks. While robust and versatile, this approach leads to inefficiencies in scenarios with high redundancy, such as iterative memory operations. The proposed extensions, including vfill and vadd, encapsulate these operations into compact, highly efficient instructions. These specialized instructions significantly reduce the instruction fetch, decode, and execution overhead, demonstrating a shift from general-purpose computing toward task-specific optimization.

A subtle yet transformative difference lies in the ability of extensions to abstract low-level loops and branching mechanics into a single cycle operation. This abstraction not only reduces hardware strain during execution but also minimizes potential pipeline stalls in more advanced processors, a design detail that remains underexplored in traditional architectures.

Cycle Measurement Results (Table 3)

	Clock cycles without extension	Clock cycles with extension	Improvement (%)
Traffic light 8*24	54059	9955	81,6%

Advantages of Extensions

1. **Execution Efficiency:** The extensions achieve an **81.6% cycle reduction**, lowering the execution cycles from **54,059** to **9,959**. This improvement rate is not merely numerical but reflects an optimized architecture that scales gracefully with problem complexity.

2. **Energy Efficiency:** Reducing the number of cycles has a compounding effect on power consumption, making these extensions ideal for embedded systems and battery-powered devices like IoT nodes.

3. **Pipeline Harmony:** By encapsulating iterative operations, the extensions reduce branching overhead and harmonize the execution pipeline, mitigating penalties caused by control hazards.

4. **Memory Bandwidth Optimization:** Extensions like vfill and vadd minimize memory access cycles by consolidating operations, reducing contention in shared memory architectures—a benefit often overlooked in simpler analyses when using assembly general functions.

Disadvantages of Extensions

1. **Hardware Complexity and Cost:** Integrating custom microcode logic necessitates redesigning control units and memory decoders. While advantageous in specific domains, it introduces non-recurring engineering costs and marginally increases chip size.

2. **Niche Suitability:** Extensions tailored for repetitive processes might underperform or become redundant in systems with diverse, unpredictable workloads. Designing for balance between specialization and generalization is critical.

3. **Debugging Complexity:** Abstracting operations into microcode hides intermediate states, potentially complicating debugging and testing workflows for developers unfamiliar with the underlying extensions.

Creative Aspects of Improvement

The proposed extensions take on a particularly valuable role when applied to our project, which involves managing a 24x24 pixel screen. With 576 bytes of pixel data stored consecutively in memory, the repetitive nature of tasks like filling the screen with a uniform color (vfill) or modifying pixel values with offsets (vadd) becomes evident. These tasks are fundamental to screen management operations, such as refreshing the display or generating transitions. By introducing extensions tailored to these operations, we align hardware capabilities with the specific demands of the application, resulting in significant optimizations.

For example, while the vfill instruction already simplifies filling the screen with a single color, a further creative enhancement could involve adding functionality to target specific rows or columns of pixels rather than the entire memory block. Similarly, vadd could be adapted to allow for conditional additions based on current pixel values, enabling effects like gradient transformations or selective color alterations directly within hardware operations.

These ideas, while intuitive, underscore the value of thinking critically about how frequently repeated tasks, like managing pixel data in a 24x24 grid, can be abstracted into hardware extensions. This approach not only improves efficiency but also reduces programming complexity, making it possible to execute screen manipulations in fewer cycles. The project demonstrates that creatively aligning extensions to specific real-world applications can yield substantial benefits, both in terms of computational performance and development efficiency.

Is the Extension Worthwhile for a 24x24 Screen?

For a 24x24 screen, implementing these extensions transcends mere efficiency gains—it embodies a paradigm shift in how computational workloads are addressed. The cycle reduction is critical for real-time applications such as traffic lights, where delays directly impact operational efficacy and user safety. Furthermore, the compact representation of instructions reduces system memory overhead, enabling simpler, cost-effective processors for mass deployment.

Conclusion

In this project, we explored the importance of microprogramming and custom extensions to optimize processes, specifically for managing a 24x24 screen. By comparing the standard RISC-V instruction set with the proposed extensions, we demonstrated significant improvements in clock cycles, achieving an 81.6% reduction in execution time. This highlights the potential of tailoring hardware instructions to specific applications, especially for repetitive and resource-intensive tasks like screen operations. The extensions not only improved efficiency but also simplified programming, showing how hardware-level optimizations can align closely with application needs.

This project helped us understand the practical benefits of extending instruction sets and how these extensions directly impact performance. It also reinforced the importance of measuring improvements and considering trade-offs when introducing custom instructions. While the extensions optimized tasks like filling and modifying screen data, it became evident that designing these instructions requires a clear understanding of both hardware and application requirements.

The project overall took us about 15 hours on the code and 6 hours for the supporting report. The most time consuming part was that after we finished the main conceptual design and functionality, several technical challenges emerged, particularly with the ALU unit and its associated operations. The complexity of the ALU's control signals, such as MA, MB, and SELCOP, often made the process prone to errors. Selecting the correct data source for the ALU was critical, as misconfigurations led to incorrect computations and data flow interruptions.

Additionally, managing temporary registers like RT1, RT2, and RT3 proved to be a recurring issue. These registers were frequently occupied with intermediate values, leaving limited flexibility for further operations. This constraint forced us to use the main register file to offload and reload data, which introduced overhead and increased instruction cycle counts.

Efficiently addressing memory for the 24x24 screen added another layer of complexity. The screen's pixel data, stored as 576 consecutive bytes, required precise memory offset calculations for operations like vfill and vadd. These operations had to dynamically iterate over memory locations while ensuring alignment and data integrity, which demanded extra ALU cycles and careful sequencing of microinstructions. Debugging also presented significant challenges, as missteps in control signal timing often caused unintended data overwrites or incorrect ALU outputs. For example, failing to reset SELCOP between successive operations sometimes led to carryover effects that disrupted computations. These interrelated challenges underscored the difficulty of balancing hardware constraints with performance optimization.