ECS201A - ASSIGNMENT1

Hezhi Xie, Mo Lyu
Email: hezxie@ucdavis.edu, molyu@ucdavis.edu
Github UserName: helenxie-bit, MOLl02
Github TeamName: ArchTech

1 ANALYSIS AND SIMULATION

1. What metrics should you use to measure the performance of a computer system? Why?

When assessing the performance of a computer system, a range of metrics are used to capture different aspects of its functionality and efficiency. These metrics offer insights into various performance characteristics crucial for different operational requirements. Key metrics include the processing speed, assessed from various perspectives:

(a) **Latency:** The delay between a command and its execution. Low latency is vital in real-time applications like gaming or financial trading systems.
(b) **Response Time:** The time a system takes to respond to a request, crucial for user experience in interactive applications and efficiency in processing systems.
(c) **Service Time:** The actual time taken to perform a task, excluding any delays. Important for assessing the efficiency of individual system components.
(d) **Processor Speed (Clock Speed):** Measured in gigahertz (GHz), indicating how many cycles per second the CPU can execute. A higher clock speed generally means a faster processor, crucial for tasks requiring heavy computational work.
(e) **Throughput:** The amount of work or data processed by a system in a given time. A crucial metric for evaluating overall system performance, especially in data processing and network systems.

In addition, other metrics related to various aspects of the computer system should also be considered:

(a) **Network Performance:** Evaluates the efficiency of data transmission within a network. Key measurements include network latency (delay in data transmission) and network throughput (amount of data transmitted successfully over the network).
(b) **Cache and Memory Performance:** Examines the effectiveness of the system's cache and memory operations. Key measurements include cache hit rate, cache miss rate, memory read/ write speed, and memory bandwidth utilization.
(c) **Storage Performance:** Assesses the efficiency and capacity of storage devices. Key measurements include disk I/O speed (speed of data read from or written to storage devices) and storage capacity (total amount of data a storage device can hold).
(d) **Availability:** Measures the proportion of time a system is operational and accessible. This is critical for systems where downtime leads to significant impacts, like servers or critical computing systems.
(e) **Scalability:** Measure the capability of a system to handle increased workload or to be easily expanded. Crucial for systems expected to grow over time.
(f) **Reliability:** Measures the system's ability to resist failures and recover effectively. Key measurements include Mean Time Between Failures (MTBF) and Mean Time To Repair (MTTR).
(g) **Power Efficiency:** Measures energy consumption (total energy consumed by the system) as well as energy efficiency (efficiency of the system in terms of performance per unit of energy). Especially important for energy-efficient computing and in environments where power is a limiting factor.
(h) **Size and Weight:** Particularly relevant for mobile and portable computing devices where physical constraints are a concern.

2. Why is it not always possible to use the same metrics for performance to evaluate computer systems?

Performance metrics for computer systems differ significantly due to the diversity in system purposes, hardware and software configurations, and user requirements. Computer systems are designed for a wide range of purposes, each with unique performance requirements. Using the same metrics for these vastly different systems would not accurately reflect their performance in their intended roles. Variations in hardware and software necessitate different benchmarks, such as high-performance clusters being judged on parallel processing and throughputs. Some metrics focus on user experience, such as responsiveness and latency, while others measure technical performance, like processor speed and memory bandwidth. The importance of these metrics varies depending on whether the user experience or raw technical performance is more critical for the system's intended use. Additionally, mobile devices emphasize power efficiency and size due to resource constraints, contrasting with desktop systems. Specialized applications, ranging from scientific simulations to video editing, also demand specific performance metrics, underscoring the inappropriateness of a one-size-fits-all approach in performance evaluation.

3. Define the Iron law of processor performance.

The Iron Law is a formal method for evaluating processor performance, illustrating the trade-off between complexity and the count of primitive instructions employed by processors for computations. It dissects the overall computation time of a program into three factors:

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{ClockCycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{ClockCycle}}$$

Where:

- Instructions per Program is the total number of machine-level instructions the program contains.
- Cycles per Instruction (CPI) is the average number of clock cycles each instruction takes to execute, dependent on the CPU architecture and instruction complexity.
- Time per Cycle is the duration of a single clock cycle, which is the inverse of the CPU's clock frequency, dependent on the design of transistors.

2 ANALYSIS AND SIMULATION: STEP I

1. At the same clock frequency, between a single-cycle CPU (HW1TimingSimpleCPU) and an in-order pipelined CPU (HW1MinorCPU) which CPU will exhibit better performance? Why?

Before simulation:

Single-cycle and in-order pipelined are two different clock-period strategies leading to different micro-architectures.

In a single-cycle processor, each instruction is executed in a single clock cycle. Since each cycle requires some constant amount of time, we will spend the same amount of time to execute every instructions, regardless of how complex the instructions may be. To ensure that the processor operates correctly, the slowest instruction must be able to complete execution correctly in one clock tick. This is the big disadvantage of single-cycle CPU: the machine must operate at the speed of the slowest instruction and faster instruction cannot execute more quickly.

In-order pipelined processors exploit instruction level parallelism to allow multiple instructions to be in execution at the same time. This is accomplished by adding state registers between the instruction fetch, instruction decode, execute, memory access, and write-back

stages of the circuit. Therefore, multiple instructions can be in the pipeline simultaneously, each at a different stage.

Therefore, at the same clock frequency (which means the same clock time), an in-order pipelined CPU tends to exhibit better performance.

After simulation:

Based on the simulation results shown in Figure 1, Figure 2, and Figure 3, it is evident that employing HW1MinorCPU yields a speedup of approximately 2.6x compared to HW1TimingSimpleCPU across varying clock frequencies. For instance, at a clock frequency of 4GHz, the computation time is reduced from 100.24ms with HW1TimingSimpleCPU to 38.19ms with HW1MinorCPU.

Given that the number of instructions remains nearly constant between the two CPUs, the observed speedup in computation time can be primarily attributed to improvements in CPI. Specifically, at the 4GHz clock frequency, HW1TimingSimpleCPU exhibits a CPI of 3.88, whereas HW1MinorCPU achieves a significantly lower CPI of 1.48.

This alignment with the predictions made in the ”Before simulation” section strengthens the validity of our analysis.

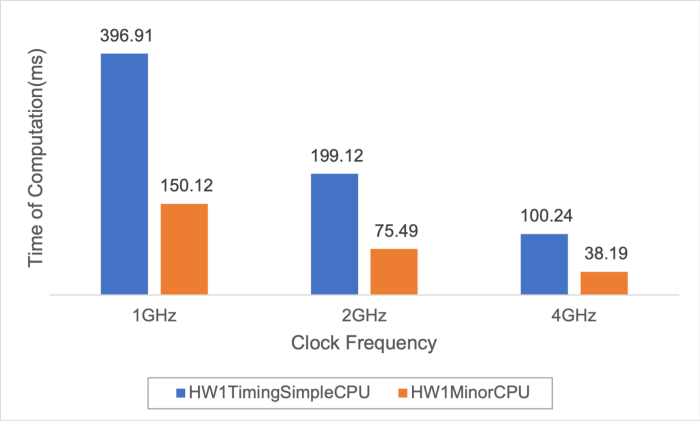

Figure 1: Comparison of Time Between Different CPUs

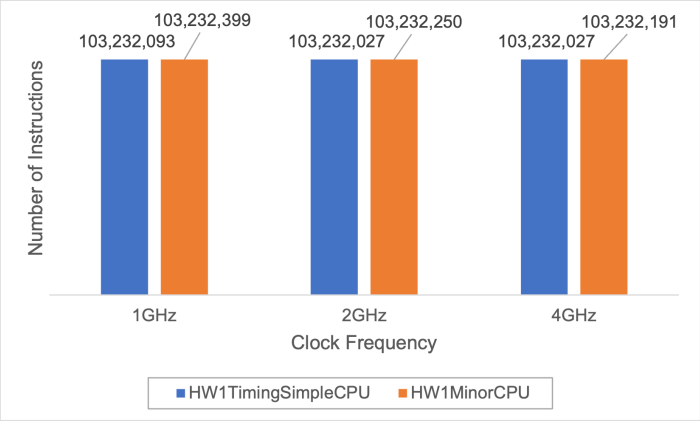

Figure 2: Comparison of Number of Instructions Between Different CPUs

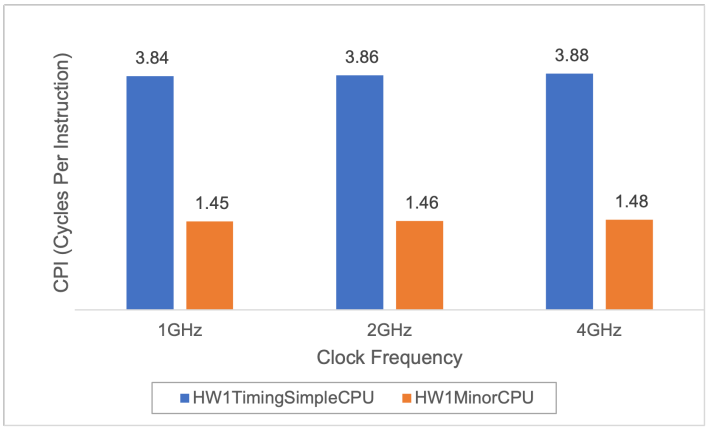

Figure 3: Comparison of CPI Between Different CPUs

2. Between a single-cycle CPU (HW1TimingSimpleCPU) and an in-order pipelined CPU (HW1MinorCPU) CPU which one is going to be more sensitive to changing the clock frequency? Why?

Before simulation:

According to the Iron Law equation:

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{ClockCycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{ClockCycle}}$$

When adjusting the clock frequency (which affects the Time per Cycle), the number of instructions remains constant, making the sensitivity of performance reliant on CPI (Cycles per Instruction).

For a single-cycle CPU, where each instruction executes within a single clock cycle, CPI remains almost unaffected by changes in clock frequency. Conversely, in an in-order pipelined CPU, the division of instruction execution into multiple stages means that an increase in clock frequency reduces the time available for each stage. This can result in pipeline stalls, hazards, or diminished performance, adversely affecting CPI and partially offsetting the advantages of a higher clock frequency in enhancing overall performance.

As a result, a single-cycle CPU experiences more pronounced performance improvements with an increase in clock frequency, making it more sensitive to changes in clock frequency compared to an in-order pipelined CPU.

After simulation:

Based on the simulation results shown in Figure 4, for HW1TimingSimpleCPU, the time of computation achieves a speedup of 1.993x when transitioning from a 1GHz to 2GHz clock frequency and 1.986x when transitioning from 2GHz to 4GHz. In comparison, for HW1MinorCPU, the time of computation experiences a speedup of 1.989x when transitioning from 1GHz to 2GHz and 1.977x when transitioning from 2GHz to 4GHz. While the speedup figures appear similar for both CPUs, it's noteworthy that HW1TimingSimpleCPU demonstrates a superior speedup compared to HW1MinorCPU, aligning with the analysis presented in the "Before Simulation" section.

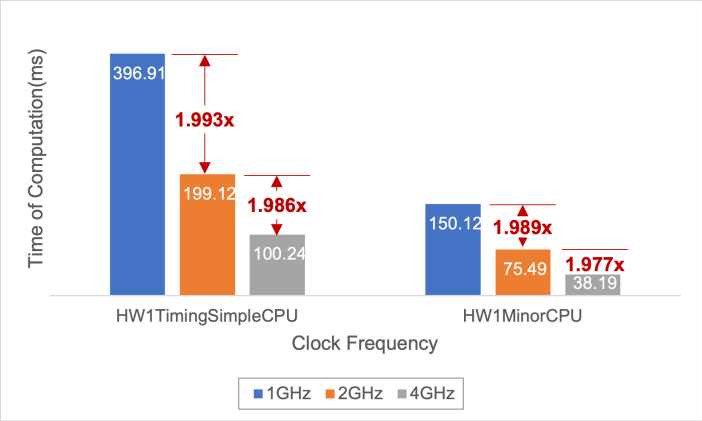

Figure 4: Speedup Comparison Under Different Clock Frequency

3 Analysis and Simulation: Step II

1. What would be the impact on the overall performance of a computer system if we were to enhance the memory bandwidth and reduce its latency?

Before simulation:

Improving memory bandwidth and reducing memory latency can lead to better overall performance of a computer system due to the following two main reasons:

- **Faster Data Access**: Reduced memory latency means that the processor can access data from memory more quickly. This leads to minimized response delays to memory requests and quicker data retrieval times, benefiting applications that rely on frequent memory access.
- **Higher Throughput**: Increased memory bandwidth allows the system to transfer a larger volume of data between the memory and the processor in a given time. This can lead to higher overall throughput, particularly in memory-bound tasks.

After simulation:

The table below displays the peak bandwidth and latency (measured by row cycle time (tRC)), for various memory models provided in this assignment:

Memory Model	Bandwidth (Peak)	Latency (tRC)
HW1DDR3_1600_8x8	12.8GB/s	48.75ns
HW1DDR3_2133_8x8	17.057GB/s	46.09ns
HW1LPDDR3_1600_1x32	6.4GB/s	58ns

Table 1: Bandwidth and Latency of Different Memory Models

The comparison reveals that in terms of bandwidth, the order is: HW1DDR3_2133_8x8 $>$ HW1DDR3_1600_8x8 $>$ HW1LPDDR3_1600_1x32; while for latency, the order is: HW1DDR3_2133_8x8 $<$ HW1DDR3_1600_8x8 $<$ HW1LPDDR3_1600_1x32. Therefore, according to the analysis in the "Before Simulation" part, the performance hierarchy should be: HW1DDR3_2133_8x8 $>$ HW1DDR3_1600_8x8 $>$ HW1LPDDR3_1600_1x32.

Based on the simulation results presented in Figure 5, Figure 6, and Figure 7, the time of computation for HW1TimingSimpleCPU is recorded as 100.24ms, 100.18ms, and 100.64ms for HW1DDR3_1600_8x8, HW1DDR3_2133_8x8, and HW1LPDDR3_1600_1x32, respectively. For HW1MinorCPU, the corresponding times are 38.19ms, 38.13ms, and 38.43ms

for the same memory models. Although these values appear quite similar, the performance hierarchy aligns with our initial analysis in the ”Before Simulation” section.

To delve deeper into the results, applying the Iron Law involves examining the number of instructions and CPI since the clock frequency remains constant across all scenarios. It is observed that the number of instructions is very similar across different memory models. Notably, HW1DDR3_2133_8x8 exhibits a slightly lower CPI than other memory models for both HW1TimingSimpleCPU and HW1MinorCPU.

The minimal differences in performance among various memory models may be attributed to the intricacies of DRAM design. While alterations in memory characteristics can influence performance, the impact may be limited by other system components, particularly the cache hierarchy, which could serve as the dominant bottleneck, mitigating the effects of different memory models on overall performance.

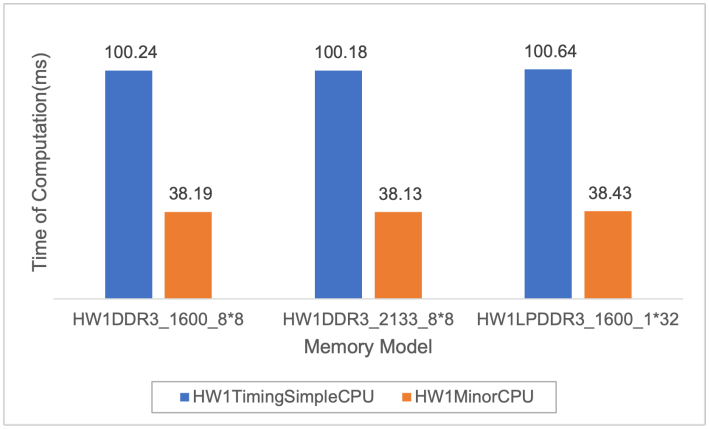

Figure 5: Comparison of Time Among Different Memory Models

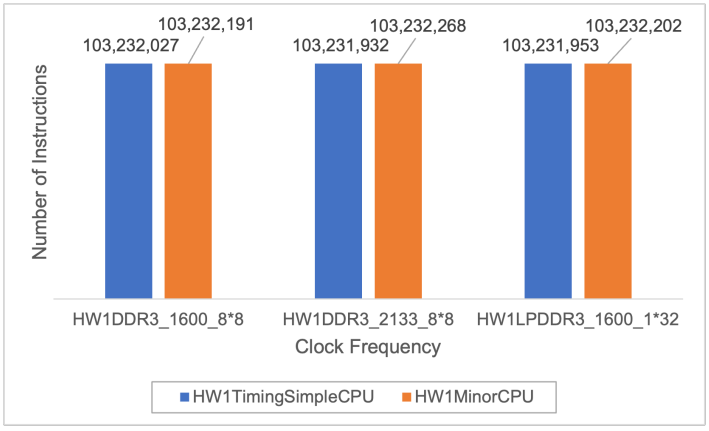

Figure 6: Comparison of Number of Instructions Among Different Memory Models

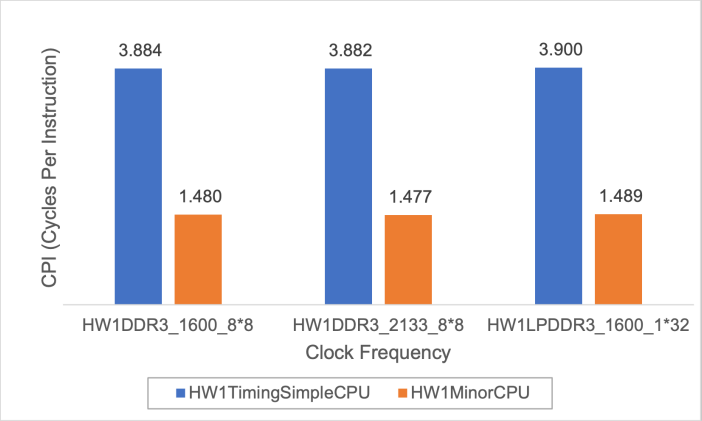

Figure 7: Comparison of CPI Among Different Memory Models

2. Which CPU model (between HW1TimingSimpleCPU and HW1MinorCPU) will benefit more from improving memory performance? Why?

Before simulation:

While both CPU models can benefit from improved memory performance, the in-order pipelined CPU, with its complex pipeline structure and dependence on efficient instruction flow, stands to gain more in terms of overall system performance.

To be specific, in-order pipelined CPUs have multiple stages in their pipelines, allowing for concurrent processing of multiple instructions. Efficient memory access is crucial to keep the pipeline stages fully utilized. Since in-order pipelines aim to overlap the execution of multiple instructions to improve throughput. This requires a continuous supply of instructions and data. Faster memory access, reduced latency, and increased bandwidth contribute to maintaining a consistent flow of instructions through the pipeline stages, reducing the impact of stalls, which is particularly important for in-order pipelined CPU.

After simulation:

Based on the simulation results shown in Table 2, for HW1TimingSimpleCPU, the time of computation achieves a speedup of 1.0006x when transitioning from HW1DDR3_1600_8x8 to HW1DDR3_2133_8x8 and 1.0046x when transitioning from HW1LPDDR3_1600_1x32 to HW1DDR3_2133_8x8. In comparison, for HW1MinorCPU, the time of computation experiences a speedup of 1.0015x when transitioning from HW1DDR3_1600_8x8 to HW1DDR3_2133_8x8 and 1.0079x when transitioning from HW1LPDDR3_1600_1x32 to HW1DDR3_2133_8x8. While the speedup is very minimal for both CPUs, HW1MinorCPU demonstrates a higher speedup compared to HW1TimingSimpleCPU, aligning with the analysis presented in the ”Before Simulation” section.

	HW1DDR3_1600_8x8	HW1LPDDR3_1600_1x32
HW1TimingSimpleCPU	1.0006x	1.0046x
HW1MinorCPU	1.0015x	1.0079x

Table 2: Speedup of HW1DDR3_2133_8x8 Compared to Different Memory Models

4 Analysis and Simulation: Step III

1. Which program do you think will perform better?

Before simulation:

The program compiled with the -O3 optimization flag is likely to perform better. The -O3 flag enables more aggressive optimizations, including the use of advanced instructions like fmadd.s (floating-point multiply and add). This instruction combines multiplication and addition in a single step, which can significantly reduce the number of instructions executed and potentially improve execution speed.

After simulation:

Based on the simulation results shown in Figure 8, Figure 9, and Figure 10, it is evident that employing -O3 yields a speedup of approximately 5.31x compared to -O0, indicating a much faster execution. A reduction of approximately 82.99% in the number of instructions executed, suggests that the -O3 flag leads to more compact and efficient code generation. The CPI values for -O0 and -O3 are 3.52 and 3.90, respectively, indicating that -O3 experiences a higher CPI, possibly attributed to more aggressive optimizations under the -O3 flag. Overall, the data demonstrates that the -O3 optimization flag provides a substantial performance advantage over the non-optimized -O0 flag.

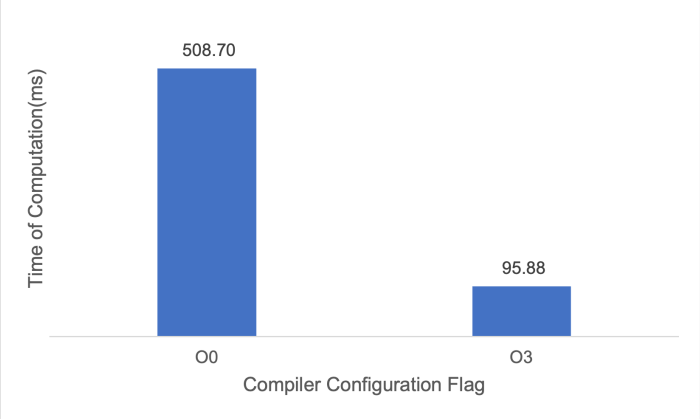

Figure 8: Comparison of Time Between Different Compiler Flag

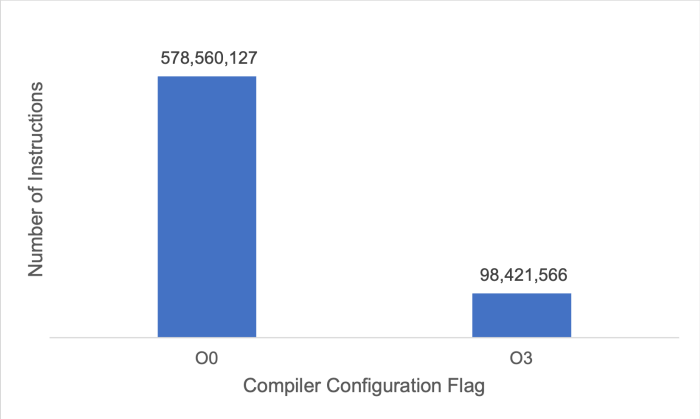

Figure 9: Comparison of Number of Instructions Between Different Compiler Flag

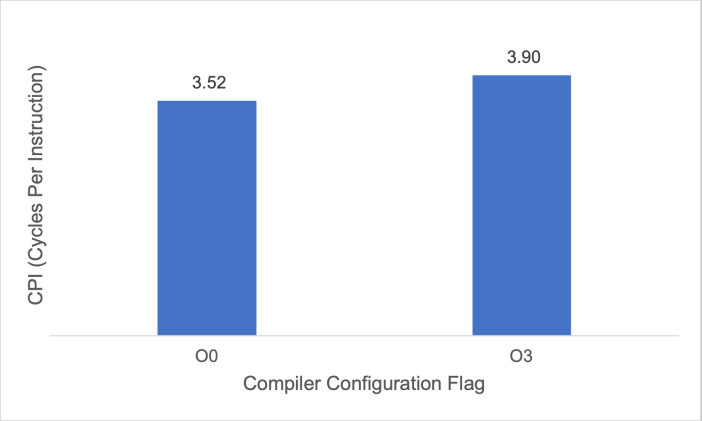

Figure 10: Comparison of CPI Between Different Compiler Flag

2. What part of the Iron Law are you optimizing in this step?

Before simulation:

The -O3 optimization reduces the number of instructions needed to perform the same computation, which is the "Instructions per Program" component. The use of complex instructions like fmadd.s combines multiplication and addition into a single instruction.

After simulation:

Flags	Number of Instructions	CPI	Cycle Time
-O0	578,560,127	3.517014	250ps
-O3	98,421,566	3.896640	250ps

Table 3: Iron Law Components of Different Compiler Optimization Flags

From the results, there is a significant reduction (approximately 83.01%) in the number of instructions when using the -O3 flag compared to -O0. The CPI is slightly higher with -O3 compared to -O0. This could be due to the more complex nature of the instructions used in the optimized code, which may take more cycles on average to execute. And the cycle time remains the same.
The primary optimization achieved through the use of the -O3 compiler flag is in the "Instructions per Program" component of the Iron Law.

3. Do you think if you use CISC ISA, the results will further improve? Why?

Using CISC ISA may not further improve the result. CISC ISAs, characterized by their complex instructions, may reduce the "Instructions per Program" component of the Iron Law of Processor Performance. However, this benefit might be offset by longer "Cycles per Instruction" due to the complexity of these instructions. Also, compilers are quite adept at optimizing code for various ISAs. The -O3 optimization flag utilizing the fmadd.s instruction, is already effectively optimizing for the current RISC.

5 Analysis and Simulation: Step IV

1. If you were to use a different application, do you think your conclusions would change? Why?

The conclusions derived from our simulations and analyses, while comprehensive for the tested scenarios, might exhibit variations when applied to different types of applications:

CPU Architecture Performance Variability: The observed performance differences between single-cycle and in-order pipelined CPUs are application-dependent. Applications that primarily involve sequential processing and exhibit limited instruction-level parallelism might not fully utilize the advantages of an in-order pipelined CPU, possibly resulting in a smaller performance gap compared to single-cycle CPUs.

Clock Frequency Sensitivity: The impact of clock frequency changes can be application-specific. Applications that are more constrained by I/O operations or network latency might show minimal performance differences with varying clock frequencies. This variability could affect the perceived sensitivity of both CPU types to changes in clock frequency.

Memory Performance Enhancements: The benefits of enhanced memory bandwidth and reduced latency depend on an application's memory access patterns. Applications with intensive memory access might show notable improvements, whereas compute-bound applications could experience only marginal gains.

Compiler Optimization Effects: The response to compiler optimizations, such as the -O3 flag, varies among applications. While some applications may already be optimized and show little benefit from further optimization, others with less efficient initial code could see significant performance improvements.

CISC vs. RISC for Compiler Optimizations: The performance implications of using CISC or RISC architectures can vary significantly based on an application's computational tasks and algorithms. Some applications may be better suited to the complex instructions of CISC, while others might benefit more from the simplicity and efficiency of RISC instructions.