CPRE 3810

Sam Burns and Isaiah Aldiano

5/3/2025

Project 3 Report

# Comparative Performance Analysis

[**Comparative Performance Analysis 1**](#_vdzy54g0n8o5)

[Introduction 2](#_egcoxrag6hrv)

[Benchmarking 2](#_3kkcfz6y7j97)

[Performance Analysis 3](#_bq20pjl4zpfi)

[Software Optimization 5](#_buiwxi4q589h)

[Hardware Optimization 6](#_eprn12vknve5)

[It Depends 6](#_h7es1txlfvxh)

[Challenges 7](#_bson6mjqe2j6)

[Conclusions 8](#_gthx7xxjmvqd)

## Introduction

This report will provide a detailed performance analysis of the three MIPS processors we have designed and implemented during the semester. We will provide both quantitative performance results and in-depth explanations (for design decisions that affect performance outcomes) in the following sections. In addition to performance analysis, we will also detail improvements that could be made- at both the hardware and software level- to improve the performance of our processors. Finally, we will also describe three major challenges we faced working through these projects in the last half of the semester.

## Benchmarking

In this section we start the quantitative comparison of our processors. In order to accurately assess and demonstrate the performance of our three processors, we used the following major benchmarks: Bubblesort and grendel. We will use Instruction count, CPI, and cycle time to determine the execution time of these programs- which will be the metric we use to qualify processor performance. Below the performance of each processor on these programs is organized into separate tables.

| **Single Cycle Processor** | Instruction Count | Cycle Count | CPI | Cycle Freq. | Execution Time |
| --- | --- | --- | --- | --- | --- |
| Bubblesort | 83619 | 83619 | 1.0 | 25.56MHz | 3.271ms |
| grendel | 2116 | 2166 | 1.0 | 25.56MHz | 0.083ms |
| allinstr | 65 | 65 | 1.0 | 25.56MHz | 0.00254ms |

| **Software Pipeline** | Instruction Count | Cycle Count | CPI | Cycle Freq. | Execution Time |
| --- | --- | --- | --- | --- | --- |
| Bubblesort | 288583 | 310124 | 1.07 | 54.32MHz | 5.709ms |
| grendel | 7299 | 7906 | 1.08 | 54.32MHz | 0.146ms |
| allinstr | 65 | 70 | 1.08 | 54.32MHz | 0.00128ms |

| **Hardware**  **Pipeline** | Instruction Count | Cycle Count | CPI | Cycle Freq. | Execution Time |
| --- | --- | --- | --- | --- | --- |
| Bubblesort | 83619 | 203457 | 2.43 | 51.44MHz | 3.955ms |
| grendel | 2116 | 4846 | 2.29 | 51.44MHz | 0.094ms |
| allinstr | 65 | 93 | 1.43 | 51.44MHz | 0.00181ms |

## Performance Analysis

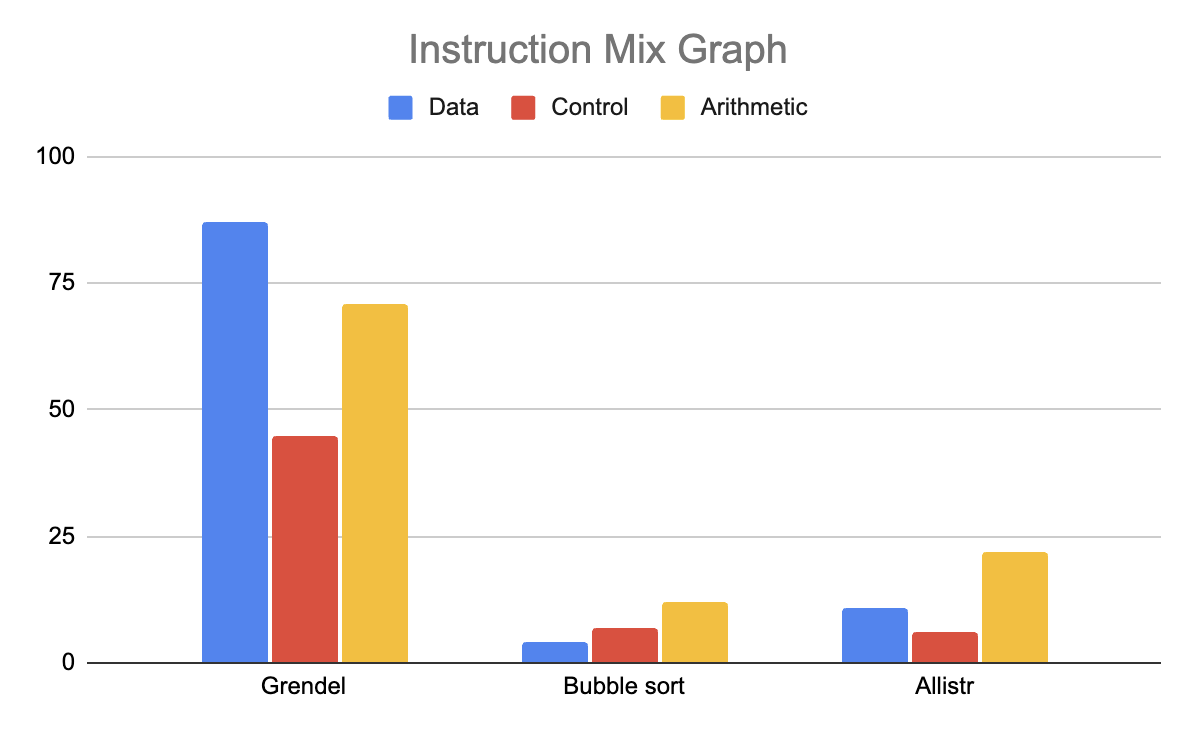


Figure 1. Instruction Mix Graph

*This instruction mix graph represents the mix of data, control, and arithmetic instructions that comprised the three benchmark programs that we ran on the three different processors implemented during the duration of this project.*

*Single Cycle processor*

*Software-scheduled processor*

*Hardware-scheduled processor*

Our benchmark testing of bubblesort, grendel, and allistr reveals significant performance differences between the three implementations of the processor. The CPI of 1.0 for every instruction across all three programs reflects the single-cycle processor representing the most simplistic design. This concurs with the simple design of the processor that doesn't account for instruction overlapping or data dependencies—additionally, the single cycle design results in a lower cycle time of 25.56 MHz. Observing the execution times of the three programs executed on the single-cycle processor, we will consider them as baseline times to reference when compared to the software and hardware pipelined processors. Because these results represent no pipelining, we can assume that any changes to the processor's design are directly related to an improvement or decrease in performance.

The software-scheduled pipelined processor attempted to enhance the single-cycle processor's performance by introducing pipelining between the different stages of the processor. We successfully split up the single-cycle processor into five stages by implementing D-flip flops that stored and output values in sync with the processor clock. The implemented instruction parallelism in our processor allowed multiple instructions to be using different stages of the processor at a time. Subsequently, the software-pipelined processor's cycle time improved by 112.5%, and CPI remained nearly identical to the single-cycle processor. Though we increased cycle time significantly, our instruction count, cycle count, and execution time suffered due to the software pipeline's inability to handle data and control hazards properly. For example, when looking at bubble sort results, the instruction count increased by 245%, the cycle count increased by 270%, and the execution time increased by 79%.

To allow for our software-scheduled pipelined processor to complete the required programs, we had to adjust each program by rescheduling instructions and adding NOOPS in between instructions with data and/or control hazards so that correct values reached the required stages in time. The actual bubble sort program had 50 NOOP instructions inserted into the file. Similarly, we had to add NOOPS and reschedule instructions to grendel and allistr in order for them to function on our software-pipelined processor. However, looking at allistr, we can see that only 26 NOOP instructions were added to the program file, significantly less than the required amount for bubble sort. This is primarily because we efficiently scheduled instructions so values being written did not cause data hazards. As a result, the execution time of allistr resulted in a 49.6% decrease. The significantly increased cycle time paired in the software pipelined implementation and efficient instruction scheduling results in improved processor efficiency. With our test programs it would be fair to assume this deduction applies more so towards simpler programs with less frequent control flow instructions.

The hardware-scheduled pipelined processor built upon the software schedule processor by adding and adjusting components to stall and flush D-flip flop values and forward data hazard values when required. In the Challenges section, we will discuss our group's inability to implement data forwarding properly. However, our processor successfully implemented a data/control flow hazard component that accurately detected when instructions in the ID stage were dependent on values in the EX stage. Our hardware-scheduled processor saw an increased cycle time of 101.18% compared to the single-cycle processor and a 5.3% decrease compared to the software-scheduled processor. This decrease in cycle time can be attributed to the high amount of stalling our processor utilizes to avoid data/control hazards. Consequently, our CPI suffers significantly, with an average of 2.05, approximately a 50% increase compared to the previous processors.

Testing our hardware-scheduled processor was similar to testing the single-cycle processor, as both processors can run programs without adjusting the software running on them. The instruction count on the single cycle and hardware-scheduled processors are identical because of no software intermediate adjustments. However, the cycle counts vary, with a 143% increase on the hardware-scheduled processor. Compared to software-scheduled, our hardware-scheduled processor has 34.38% fewer cycle counts because the hardware takes responsibility for stalling instead of intermediate software adjustments. Along with a lower cycle count, our hardware-scheduled processor performs significantly faster in execution time when comparing results of bubble sort between hardware and software. With approximately 30% fewer cycle counts and similar cycle times, the hardware schedule saw a direct decrease of 30% in execution times. Additionally, the software and hardware processors performed similarly running grendel and allistr despite the hardware having different microarchitecture.

Overall, we saw a substantial increase in cycle times for the hardware and software scheduled processors compared to the base cycle time of the single-cycle processor. Despite the lack of forwarding in our hardware processor, we could still observe multicycle processors' improvements over single-cycle processors in cycle and execution time. The varying microarchitectures substantially changed the way each processor handled programs. Results from running programs with the same functionality but different implementations gave insight into the pros and cons of software intervention and when to allow hardware to handle core functionality. In the following two sections, we will discuss what kinds of hardware and software optimizations could improve performance.

## Software Optimization

One obvious change that could be made to improve performance of the software pipeline is instruction reordering. Using this method, programs may be able to run more efficiently, preventing the use of avoidable nops. However, this method is highly program specific, and would only improve performance in codes that have avoidable load-use hazards. Programs that have unavoidable load-use hazards would not benefit from this technique. One technique that would show noticeable performance improvements in the benchmarks used in this report would be loop unrolling. Loop unrolling would allow programs such as grendle and Bubblesort to execute without the need to jump. Quantitatively, the direct performance improvement would be decreasing the instruction count by the number of jumps and branches taken during execution.

## Hardware Optimization

As for Hardware optimizations, starting with the single cycle processor, we could decrease our slack by an estimated 2ns if we redesigned our barrel shifter for speed. This would lead to a 5% increase in frequency. This increase in frequency would directly correspond to a 5% reduction in execution time for our single cycle processor. The reason that we identified the barrel shifter as an area of improvement, despite the memory modules contributing far more to slack, is that these memory modules were provided to us by the instructor, and changing their form/function would likely conflict with the testing toolflow. However, without the constraint of the toolflow issues, this would be an area where improvements could lead to a massive reduction in cycle time.

Next, the critical path of the software scheduled pipeline was through the ripple carry adder of the ALU. Changing our adder design could lead to a significantly faster cycle time. The optimization we recommend to make is to change the ripple carry adder to a carry lookahead adder. This would reduce the latency in the execution stage by so much that it would be guaranteed to no longer be on the critical path. Therefore, our new cycle time would be set by the next slowest component which - in our case - is the IMEM with a latency of ~12.1ns. This would increase our clock frequency to ~82.5MHz. This increase in frequency would directly correlate with a 52% decrease in execution time.

Finally, there are several improvements that should be made to improve performance of the hardware scheduled pipelined processor. Firstly, and most importantly, we need to implement forwarding. We were not able to implement a successful forwarding unit, opting to just stall when a data dependency instead. Had we been able to implement the forwarding unit, we would be able to reduce the CPI by about ~40% in the worst case. This would make our hardware scheduled design perform better than the single cycle design and software pipeline in a vast majority of programs. However, as we will see in the next section, the best performing design is highly dependent on the structure of the program being tested.

## It Depends

As mentioned at the end of the previous section, the speed of our processors is highly dependent on the structure of the MIPS program being tested. On a program that does not jump or branch, our pipelined processors will be nearly twice as fast as the single cycle processor. However, once a program implements jumps and branches, the software and hardware pipelined processors become noticeably slower than the single cycle processor. This point is backed up by the measured execution time of the programs found in the benchmark section.

The allinstr program uses only a handful of jumps and branches, as it is not a looping program, and therefore only stalls a few cycles. Additionally, there are few data dependencies in the program; the allinstr program requires few stalls, and therefore the number of cycles is much closer to the instruction count than the other programs. Because of these factors, the hardware and software pipeline processors perform much better than the single cycle processor.

However, because we do not implement forwarding on our hardware pipeline processor, the single cycle processor runs Bubblesort nearly twice as quickly. Being unable to implement forwarding on our hardware pipelined processor forces programs to stall until data dependencies are resolved, causing cycle count of looping programs (such as Bubblesort and grendel) to increase geometrically in cycle count. Despite having a cycle period less than half that of the single cycle processor, the increase in cycle count causes the two programs to run slower on the hardware pipelined design than on the single cycle processor.

Despite not having a forwarding unit, the hardware pipelined processor is still able to perform better than the software scheduled pipeline in both grendel and bubblesort. The reason for this is that we changed DFFs in the register file to write on the negative edge of the clock. This allowed us to ignore load use-stalls once they reached the writeback stage, as the data would be written to the register file in the same clock cycle once this change was made. This change allowed us to stall far less in these two benchmark programs, which is reflected in the difference between the total number of cycles required to execute the benchmarking programs.

## Challenges

The projects in this class were not without their struggles. Designing a system of this scale was far beyond the scope of the first two labs, so learning the skills necessary to work through these projects was learning experience in and of itself. On top of this, these projects required both partners to have a full understanding of each component in the processor. Achieving such a level of understanding took both time and effort, but we were able to get through these challenges by talking things out together. Overall, the most critical struggles we had were designing the control unit, writing the software scheduled pipeline tests, and testing the hardware scheduled pipeline.

When starting project 1, we felt very overwhelmed trying to come up with a high level sketch of our design. It seemed as though we would need to have our design entirely thought through before we even wrote a single line of code. This led us to spin our wheels a bit, spending time trying to completely design each module at the high level so we would have a straightforward path when it came to implementing our design in VHDL. This was especially true for the design of the control signal module. The control signal spreadsheet we needed to fill out was very intimidating. This was the point where we needed to assign the operation each piece of hardware would be performing, at every step in our processor, for every MIPS instruction. Working through this challenge took a lot of textbook reading and group decision making to make sure we were on the same page.

As we moved on to the beginning of project 2, we found it fairly simple to implement the software scheduled pipeline. We were on the same page for what needed to be done, and most of our debugging simply involved making sure signals were hooked up correctly. However, as we started trying to write the test programs for the processor we struggled to get them to run. We still struggled to understand where data dependencies would crop up, especially between jumps and branches. After some practice revising homework questions and reading the textbook, we were able to write some test codes that ran on our software scheduled pipeline.

Our team may have struggled to implement stalling, flushing, and forwarding more than any other part of the two projects. After we had our initial design done, almost none of the tests worked besides addiseq, showing we hadn’t necessarily broken anything, but anytime there was a data dependency our processor was completely unable to handle it. After talking through our design with the TAs, we realized that we did not correctly understand stalling. After implementing this fix, more tests worked, but other issues cropped up. We also realized that we were taking the wrong input into our branch address calculation. Finally, we spent an enormous amount of time trying to debug forwarding and get it working, but were unsuccessful. We spent a lot of time addressing the symptoms of our improperly functioning unit, rather than addressing the root cause of the issue. We spent time making the forwarding cases too specific, and not implementing a ruleset for how the module should perform.I believe that if we had correctly implemented stalling and flushing earlier, and taken a more principled approach to designing the forwarding unit, we would have been able to successfully implement it.

## Conclusions

Overall, these projects have turned out to be incredible learning experiences. Not only did we get to make plenty of design decisions with our processor design and observe their implications. But we also got to reflect on how these design decisions manifest in real programs and compare how different architectures run on different implementations in this final project. Aside from skills directly related to the class, we both improved at using our team-based workflow through git, and in our communication skills to notify each other of updates to the code or design as a whole. CPRE 3810 has provided us with many opportunities for growth this semester.