# CprE 381: Computer Organization and Assembly-Level Programming

# Project Part 3 Report

Team Members: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Alek Norris\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Drew Kearns\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

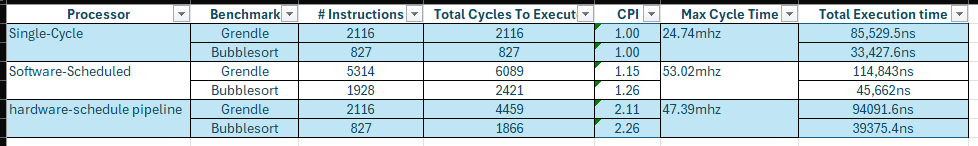
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## Project Teams Group #:\_\_\_\_\_\_\_\_\_Term Proj1\_2s\_07\_\_\_\_\_\_\_

1. **Introduction -** *Write a one paragraph summary/introduction of your term project*

For our term project in CPR E 381, we were tasked with building a basic processor from scratch following the MIPS ISA. The project was divided into five parts. In Part One, we created basic designs in VHDL, such as multi-bit adders and gates, honed our debugging skills, and learned to track and resolve bit-level issues. Part Two escalated our challenge, requiring us to develop a simple adder-subtractor ALU and expand a single D-Flip Flop into multiple D-Flip Flops to store up to 32 bits, forming the foundation of a single register. Additionally, we constructed a register file capable of holding 32 registers, each containing 32 bits. In Part Three, we implemented a single-cycle processor compatible with the MIPS ISA, enabling us to run programs. Parts Four and Five were combined but completed individually; we first transformed our basic single-cycle processor into a four-stage processor and modified our initial programs to include software data and control hazard avoidance techniques. Lastly, we tackled hardware hazard avoidance by integrating a hazard detection and forwarding unit, which could detect instructional hazards and either forward the data to mitigate the hazard or stall the processor and wait for the hazard to resolve that way.

1. **Benchmarking -** *Please generate a table for each of your final single-cycle, software-scheduled pipeline, and hardware-schedule pipeline designs.*

****

1. **Performance Analysis -** *Explain in your own words why the performance was better on one processor versus another or why some applications may have had a smaller difference in performance between processors versus other applications.*

Looking at these results, they all make sense. The maximum cycle time was highest on the software and slowest on the single-cycle processor, but the CPI for the single-cycle was faster, and the overall execution time was quicker. This is because it measures the output from one D-flip flop to the input of another. With the single-cycle, it takes the most time because it must pass through all stages, whereas with the multi-cycle, our slowest time would just be our slowest stage.

The software is slightly faster than the hardware because the multiplexers and the hazard detection unit used for forwarding data in the hardware may have led to a critical path slowdown. However, when examining the CPI and execution time, you can see that the hardware was overall faster despite a significant increase in execution time. This was because in the software, we counted the NOPs (no-operation instructions) as instructions that were placed to avoid data and control hazards. The only time an instruction wasn't counted was when a branch was taken, thus we had additional overhead from 3 instructions that didn’t need to be computed. The software scheduling overall added quite a lot of extra NOP instructions to avoid all data hazards, and because of that, the total number of instructions ended up being considerably more than any other design, leading to an overall slower execution time.

Looking at the hardware, however, we can see that while it had a much higher CPI and a slightly slower cycle time than the software scheduled pipeline, it was faster overall. This was because of its ability to insert stalls and flushes only when necessary, as well as its capability to forward most data hazards, meaning we would have significantly fewer total instructions to cover. As a result, we could finish the program faster, despite having a slower cycle time. Comparing this to the single-cycle processor, however, neither would be enough optimization to beat the original design in these out two benchmarks we compared. As the total number of instructions would have to be much higher, or the jumps would need to be much lower in our compared designs. Since the cycle time is about double, we’d also need to see no more than double the total number of cycles to instructions for the two processors to break even.

The most interesting thing to note here is that even though Grendel was slightly slower to execute the hardware design compared to the single-cycle processor (10% slower) as opposed to Bubble sort, which was only about 17% slower. This discrepancy comes down to the types of instructions and the types of forwards we have in place. In Grendel, there are more data hazards, which we were able to mitigate, but in bubble sort we mostly had jumps. Because we only implemented data hazard forwarders for our design this caused Grendel to perform at a slightly faster speed than bubble sort. If we had a much larger application without jumps, the hardware might be just as fast, if not slightly faster than the single-cycle processor. But with applications that are very dependent on jumps like Grendel or bubble sort, we are slowed down by the need to keep stalling and flushing to avoid all the control hazards, causing the CPI to rise.

When comparing the software, hardware, and single cycle, the overall choice of which design is better would come down to the needs of the user. If they are looking for the fastest version with the highest cycle time, we would recommend the software version. However, it will not be very backward compatible with programs because of its need for software scheduling. If someone were looking for whatever processor is the easiest and will get the job done quickly enough, no matter the scenario, we would recommend the single-cycle processor. Which on paper may appear significantly slower, its overall execution is faster because it doesn’t need to worry about rescheduling for hazards or even trying to detect them. And if someone wanted a processor that would more than likely offer faster execution time overall if the programs being run were optimized to not have a lot of jumping, then they should go with the hardware with forwarding. In the long run, in terms of trillions of instructions, it is more than likely that the gap of the hardware being slower would close.

The great thing about the hardware-scheduled processor is that it can continue to be optimized, making things run even faster. Given enough time, we might have been able to learn how to avoid all hazards in hardware using forwarders, which would speed up the execution time to almost double that of the single cycle. That’s where the single-cycle processor actually loses out, because the hardware is only as slow as its slowest stage, it can continue to be optimized bit by bit. However, with the single-cycle processor, you can only optimize the components so much before you start hitting a wall, sort of like what’s happening right now. We have the basic components, and so we can only speed those up; however, with the hardware, we can use the same components and optimize them to finish each stage faster, as well as minimize our stalls and flushes.

1. **Software Optimization -** *Describe one software optimization (i.e., assembly level software refactoring) that would improve the performance of software on the software scheduled pipeline relative to the others. Provide an estimate of the performance benefit this change could have given your specific benchmarks.*

For software optimization, there are several effective methods. Two approaches that could be particularly beneficial are static branch prediction and loop unrolling. With loop unrolling, instead of executing a single loop, we can replicate the loop multiple times. If we know the exact number of iterations needed, we can directly incorporate that number of loop iterations into the code. While this approach might result in a more cluttered codebase with more static instructions, it would significantly reduce the number of NOP instructions.

Using this method in conjunction with static branch prediction can also greatly enhance efficiency. This involves optimizing the code so that, rather than scheduling NOPs after a branch or jump, we schedule actual instructions. These instructions would be immediately utilized after the jump, in the next loop iteration, or if the branch is not taken, in a way that they do not interfere with any data elsewhere. Employing both of these techniques together could substantially improve our execution time by reducing idle time and minimizing the number of instructions discarded by the processor.

1. **Hardware Optimization -** *Describe at least one different hardware optimization for each design that would improve its performance. Optimization cannot be turning it into one of the other designs. Certain optimizations can be beneficial to more than one design. Choose one design on which you would apply the optimization. Briefly list the specific set of changes you would have to make to your design to accommodate each optimization (a figure would be helpful). Provide an estimate of the performance benefit each optimization could have given your specific benchmarks.*
2. **It Depends -** *Describe your approach to building these programs. If one of these cases is impossible given your designs, argue quantitatively why that is the case.*
3. **Challenges -** *In at least three detailed paragraphs, describe the three most critical challenges your group faced, how you resolved them, and how you could avoid them in the future.*
4. **Demo** - *Each member of the project group will be required to be present for the demo, which will take place during regular lab hours. During this time, you will describe the various design tradeoffs of your project parts, describe how they compare to each other, demonstrate simulations of your benchmarked applications, and discuss potential optimizations.*