CprE 381: Computer Organization and Assembly-Level Programming

Project Part 3 Report

Team Members: __________________Alek Norris___________________

__________________Drew Kearns__________________

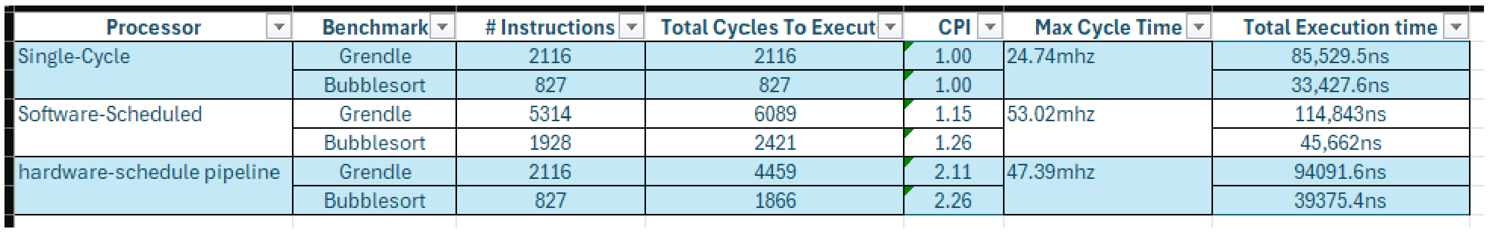
____

Project Teams Group #:___________Term Proj1_2s_07________

1. **Introduction -**

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.

2. **Benchmarking -**



Processor	Benchmark	# Instructions	Total Cycles To Execut	CPI	Max Cycle Time	Total Execution time
Single-Cycle	Grendle	2116	2116	1.00	24.74mhz	85,529.5ns
	Bubblesort	827	827	1.00		33,427.6ns
Software-Scheduled	Grendle	5314	6089	1.15	53.02mhz	114,843ns
	Bubblesort	1928	2421	1.26		45,662ns
hardware-schedule pipeline	Grendle	2116	4459	2.11	47.39mhz	94091.6ns
	Bubblesort	827	1866	2.26		39375.4ns

3. **Performance Analysis -**

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.

4. **Software Optimization -**

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.

5. **Hardware Optimization -**
    a. **Single-Cycle Processor**
        i. In our processors we implemented Byte and Word decoders that come after our DMEM. These essentially add two shifters, an and-gate, and a MUX each, but since this is run synchronously it basically only adds a another shifter to the critical path. This really increases our critical path frequency. If we were to implement these in the ALU using the preexisting shifter that is in there we would be able to shorten the critical path and decrease execution time. This wouldn't shorten it by much but with longer tests it would greatly help the performance.
    b. **Software-Scheduled Multi-cycle Processor**
        i. For this processor we did not implement forwarding, so the biggest help for this processor would be to move the control logic (branching and jumping) into the decode stage. This would allow for branches and jumps to only have to flush 1 stage of the pipeline as opposed to 3 stages which would significantly reduce the CPI for the control instructions. To accommodate this we would have to move the branch adder from the execute stage and the muxes from the memory stage into the decode stage. This would likely change our critical path but would greatly reduce overall execution time.
    c. **Hardware-Scheduled Multi-cycle Processor**
        i. For the hardware-scheduled pipeline we would add branch prediction. Branch prediction would help a lot for tests like grendle where there are multiple loops. This would allow us to not flush any stages as the prediction would happen in the fetch stage if the prediction was correct. If the prediction was incorrect then we would still only have to flush one stage as we would also have moved the branching and jumping into the decode stage with the software-scheduled pipeline.

6. **It Depends -**

As seen in part 2 of this report our single-cycle processor actually performs better on both grendle and bubblesort when compared to the hardware-schedule pipeline. The cycles per instruction is over two times as much as the single-cycle for both tests. We believe that it is impossible for our hardware-schedule pipeline to perform better than our single-cycle pipeline for any program unless the program has absolutely no data hazards in it. If the program has no data hazards in it then our processor should have a lower CPI than the single-cycle processor because we would only need to stall/flush when we are jumping or branching. If there are data hazards then we typically have to stall to 1 CPI for each noncontrol instruction. Then when we get a control instruction it will increase the CPI because it has to flush every stage behind it which increases the CPI for a control instruction to 7 CPI.

Comparatively our software-scheduled pipeline is much worse. It did not outperform our hardware-scheduled pipeline at all. This makes sense because our software pipeline did not have any forwarding capabilities so it had to stall for every hazard. Since our hardware pipeline was basically the software pipeline but improved with forwarding it is impossible for our software pipeline to ever outperform the hardware pipeline. Whenever the hardware pipeline has to flush, the software pipeline has to flush, but the hardware pipeline does not have to stall every time the software pipeline has a nop.

7. Challenges -

Our first critical challenge was balancing the workload of the project not only with this classes other assignments but with our other classes as well. We were both taking classes that had projects outside of this class with other partners. We had to coordinate with multiple different people to work on projects this semester which was not an easy task. The way we coordinated with each other was constant communication about when we were working on the projects or when we planned to. If there was ever any unexpected events that prevented us from working on the project we let the other teammate know ASAP. This is a pretty unavoidable challenge but we persevered through it regardless.

Our second critical challenge was making sure we had our parts of the project done when we said we would get them done. When working with a partner it is important to finish your tasks when you say you will especially if the other partner is needing parts of the project from the other partner in order to continue their own part of the project. As I stated in our first challenge, we were both taking other classes that we had to work with teammates. So similarly to that challenge we were in constant communication about possible delays that would come up with our parts of the project and reaching out for help if we needed it. To avoid this in the future we could do more thinking about how parts of the project will connect before assigning parts to each other. I feel like this would have been beneficial to us as one partner would not be waiting on the other to implement certain sections of the project and instead that one partner could do the parts that correspond with one another as the other partner works on other sections.

Our last challenge was trying to get all of the parts of our processor to work together. This challenge mostly applies to project 2. For project 2 we seemed to have lots of issues getting our different units to work together. This is partly due to the taking apart and putting back together we had to do to introduce pipelines. I feel like one way we could have fixed this was to make each pipeline stage it's own unit so we could test each unit separately and more thoroughly in their own test benches. This would have allowed us to catch issues faster even though putting them in their own units would have taken a while. I also don't know if that would have been possible with all the specific wire names we had to use. The testing stage of the processors was undoubtedly the hardest part of the project for all parts. To avoid this in the future we could implement better test benches initially so that when we put it all together we won’t have to debug as much when things break.

8. **Demo -**