CprE 381 – Computer Organization and Assembly Level Programming

Project Part 3

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

This project provides a comprehensive look at how to make a single and multi-cycle processor and what tradeoffs you can make to improve/change a processor. For much of this lab, we were forced to learn the hard way. The best way to see what might slow down your processor is by making slow components that actually slow your processor down, and the best way to figure out that process statements refuse to synthesize is by including process statements in nearly every component. Jokes aside, we found this lab enjoyable and interesting but severely demanding. Debugging took more time than the actual planning and building most of the time.

2. Benchmarking. Now we are going to compare the performance of your three processor designs in terms of execution time. Please generate a table for each of your final single-cycle, software-scheduled pipeline, and hardware-schedule pipeline designs

Single Cycle	Synthetic bench	Grendel	Bubble sort
# instructions	42	2116	976
Cycles to ex	42	2116	976
CPI	1.0	1.0	1.0
Max cycle time	25.53 MHz (39.1696 ns)	25.53 MHz	25.53 MHz
Total ex time	1645.12 ns	82882.88 ns	38229.53 ns

The Software scheduled pipeline struggled to run Grendel, and we suspect we are missing nops somewhere, so Grendel is not included in the table.

Software	Synthetic bench	Grendel	Bubble sort
# instructions	97	N/A	2199
Cycles to ex	106	N/A	2430
CPI	1.09	N/A	1.11
Max cycle time	52.41 MHz (19.08 ns)	52.41 MHz	52.41 MHz
Total ex time	2022.48 ns	N/A	46365.1975 ns

The hardware pipeline processor was the last thing we did, and it had trouble with branching and jumping. So we used “base tests” which has no branch logic. We can not compare things like Grendel and Bubble sort because they do not run on our hardware processor. We understand this is not what the project wants us to do, but we felt it was important to include some sort of metric to compare the 3 processors, and base tests was one that they all could run.

Base Tests	Hardware	Software	Single Cycle
# instructions	23	48	23
Cycles to ex	48	51	23
CPI	2.09	1.06	1.0
Max cycle time	52.84 MHz (18.925 ns)	52.41 MHz (19.08 ns)	25.53 MHz (39.1696 ns)
Total ex time	908.4 ns	973.08 ns	900.9 ns

3. Performance Analysis. Analyze the performance of the three applications on the three processors. 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.

The single-cycle processor was better than the hardware and software pipelined at everything we tested, but the hardware-implemented pipelined processor was nearly as fast as the single-cycle processor for base tests. This leads me to believe that the hardware pipelined processor could have been the fastest-performing processor if we had gotten the branch and forwarding functions to work. For programs that included more hazards, the single-cycle processor performed better. For programs with fewer hazards, both the software and hardware pipelined processors performed better due to their ability to process multiple instructions at once.

4. Software Optimization. Identify and describe one software optimization (i.e., assembly-level software refactoring) that would improve the performance of the 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.

If there was some way for jumps to automatically happen in IF, they wouldn't have to be accounted for with nops or special hardware. The addition of a software jump that could change the address itself would save us 3 nops per jump, which could add up. By removing 3 no ops from jump instructions in a 200-cycle program that is 50% jump instructions, you would have 125 cycles out of the previous 200 cycles because each jump would only need 1 cycle (jump) instead of 4 (jump + 3 nops).

5. Hardware Optimization. Identify and describe at least one different hardware optimization for each design that would improve its performance. The 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.

Single cycle: I am using a ripple carry adder for my ALU because it seemed to be the easiest to implement. However, I now understand how long that takes because you require the answer to the last addition to know the next addition due to the carry bit being part of every addition. By using a Carry look-Ahead adder, the hardware complexity would increase, but the ALU could add multiple bits at the same time in parallel, potentially speeding up the adder's time to execute. A potential downside of this could be the hardware becoming very complex if we attempted to add 32 bits at once.

Software multi-cycle: For our software multi-cycle processor, our fetch executes in our write-back stage. However, if we were to move this up, branch and jump instructions could execute one cycle faster, which could offer a significant performance boost in branch/jump-heavy programs. Potential downsides of this implementation would be the additional hardware necessary to branch. For example, the equals comparator the fetch can use from the ALU would have to be implemented in the ID stage to supply logic for the branching operations. By removing a cycle from branch cycles in a 100-instruction program free of read-after-write hazards that is 50% branch instructions, you would have 200 cycles out of the previous 250 cycles because each branch would only need 3 instructions (branch + 2 nops) instead of 4 (branch + 3 nops).

Hardware multi-cycle: For our hardware multi-cycle processor, we were unable to implement forwarding due to time constraints and issues with flushing branch logic instructions, so forwarding would be the upgrade to make. We could forward write-back data to the ID, MEM, and EX stages to reduce the number of nops needed. For example, If an instruction needs to use a register that hasn't been written yet in ID, the forwarding could send the value of the data to a mux in the decode stage so that the processor does not have to use so many nops to wait for the value to be stored. Potential downsides to this hardware implemented forwarding could be a slow down of cycles because of the addition of muxes, and the complexity and time and cost of the processor would definitely increase with the introduction of forwarding.

6. It Depends. Given the above discussion, you should now understand the interaction between the programs and your hardware designs in terms of performance. Identify or write a program that performs better on a single-cycle processor versus a hardware-scheduled pipeline and another one that performs better on the hardware-scheduled pipeline versus the software-scheduled pipeline. Describe your approach to building these programs. If one of these cases is impossible given your designs, argue quantitatively why that is the case.

Given our errors and incomplete design, the hardware pipeline never performed to its full extent, and as such, every program we tested favored the single-cycle processor in performance time. Without forwarding, the hardware pipeline could not compete with the single cycle, but If we were to re-write programs with the intent of minimizing nops at the cost of additional instructions, we could make programs favor the multi-cycle processors.

7. Challenges. This term project was challenging for every group. 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.

The first challenge for us was understanding how VHDL worked and what the correct syntax was. Just to get test benches working was a struggle, and throughout lab 1 and lab 2, I went to most office hours and even some additional labs just to understand how to use VHDL better. Once we built most components in lab 2, we were comfortable enough with the logic portion. But then we needed to figure out how the tool flow worked, and that required changing our code. Next, synthesis required us to get rid of all process statements and start some components almost from scratch. And once we figured out how to get the processors to synthesize, we had to change the formatting and file locations to get it to work with the auto-grader. For every hour we spent on design, we spent another on formatting and syntax. Knowing what we know now about how picky synthesis and testing can be, I would have attempted them in advance and tried to build my components to be synthesizable from the start.

The next challenge for us was debugging and chasing signals. At the beginning of Project 1, we used some code from Lab 2. We did not do a good enough job testing the register in lab 2, and the register had edge cases and behaviors that we had not accounted for. This led to painful debugging in Project 1, chasing signal after signal leading to errors in individual projects. If we were to do this again, we would take the extra time to thoroughly test the components before attempting to merge them together.

The final challenge we would mention would be naming signals and understanding what they do. All of these problems we are mentioning we were warned about, but it is hard to understand the warning without seeing the problem. I think we had to learn the hard way for many of these lessons. When we started Project 1, we would incrementally add signals. For example, we made a branch signal, then a branch logic signal, then a branch and link signal, etc., until it was hard to understand which signal did what, especially if they were generically named. Knowing what we know now, trying to plan for signals at the start, and being deliberate with our signal names could have saved us much time debugging and untangling signals every time we worked on the processor.vhd file of the project.

8. Demo. You will be expected to demo your benchmarking process to the Professor and TAs on the project due date. 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

We demoed to Duwe at 12:15 after Steven left.