CprE 381 – Computer Organization and Assembly Level Programming

Project Part 3

[Note: This is the final and summative component for your term project. The purpose of this assignment is to put the other components of your project into context and allow you to relate the detailed implementations you have done to the higher-level concepts we have interacted with in other aspects of the course. As such, ***this report is expected to be of higher quality than your previous reports, in particular with regard to its clarity, analysis, and readability****. Please make an effort to provide context for all figures and some flow through the paper (i.e., don't just copy the report template and respond directly). You have three working processor designs. Congratulations on getting to this step!*

Disclaimer: Due to this project being due during ~~Dead~~ Prep Week, there will be no extensions granted. Please turn in whatever work you have completed by the due date.]

0. **Prelab.** Review your notes, evaluations, and feedback regarding your single-cycle, software-scheduled, and hardware-scheduled pipeline. Make sure you have the three designs ready to evaluate. In particular, ensure that you have your synthesis results handy.

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

   Our term project involved creating a MIPS processor starting with a single-cycle design and improving on it to build a software-scheduled pipeline and finally a hardware-scheduled pipeline processor. Our implementation of the mentioned processors mainly used Structural, some Dataflow, and no Behavioral VHDL. The objective of this project was to implement all the modules learned so far in this course to improve on the previously built single-cycle processor by converting it into a pipelined processor. New modules used in this lab include the four pipeline registers IF/ID, ID/EX, EX/MEM, and MEM/WB registers. Furthermore, a forwarding unit has been implemented to prevent data dependency hazards, while stalling has been implemented as a part of a new hazard detection unit to avoid load-use hazard.

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-scheduled pipeline designs. The rows should correspond to your synthetic benchmark (i.e., the one with all instructions), Grendel (provided with the testing framework), Bubblesort, and, for teams >4, Mergesort. The columns should be # instructions (count using MARS), total cycles to execute (count using your Modelsim simulations), CPI (using the previous two columns to calculate), maximum cycle time (from your synthesis results), and total execution time (using the appropriate previous columns). Note that the applications used to benchmark the single-cycle and hardware-scheduled pipeline applications should be identical and thus the same number of instructions, while the software-scheduled pipeline programs should be modified to work on the software-scheduled processor and thus should have more instructions. Count software-inserted NOPS as instructions. Make sure to include units and double-check that these results make sense from your first principles!

Single Cycle Processor Benchmark Table:

Test	# Instructions	Total Cycles	CPI	Max Cycle Time (ns)	Total Execution
proj2_base_test	41	41	1.0	41.8	1,713.8 ns
grendel	2116	2116	1.0	41.8	88,448.8 ns
Bubblesort	732	732	1.0	41.8	30,221.4 ns

Software Scheduled Pipeline Processor Benchmark Table:

Test	# Instructions	Total Cycles	CPI	Max Cycle Time (ns)	Total Execution
proj2_base_test	73	81	1.11	19.2	1,555.8 ns
grendel_hazardFree	5519	5780	1.05	19.2	111,263.0 ns
Bubblesort_hazardFree	856	913	1.07	19.2	17,585.7 ns

Hardware Scheduled Pipeline Processor Benchmark Table:

Test	# Instructions	Total Cycles	CPI	Max Cycle Time (ns)	Total Execution
proj2_base_test	41	51	1.24	23.1	1,174.4 ns
grendel	2119	2852	1.35	23.1	66,081.0 ns
Bubblesort	732	912	1.25	23.1	21,136.5 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. This section should reference the above performance table and describe how it was generated, including any formulas you used. The section should be about five to seven substantial paragraphs.

Each processor ran three different applications: base_test, which executes all the instructions once; Grendel, a large application that tests numerous data hazards; and bubblesort, an application commonly used in the real world, featuring multiple stores, loads, and loops. The single-cycle processor and the hardware-scheduled processor utilized the exact same file for all three tests. The software-scheduled processor maintained the same functionality for each test, only using reordering and no-operations (nops) to ensure compatibility with a pipeline processor that lacks hazard prevention. The MARS simulator reported each program's instruction count (IC) and cycle count. The cycles per instruction (CPI) can be calculated by dividing the instruction count by the cycle count. The cycle period for each was reported by the synthesis and calculated by taking the inverse of the maximum frequency. Finally, the total execution time for each application was determined by multiplying IC by CPI and then by cycle time.

At a quick glance at the results, the single cycle is the worst design since it is the slowest for the base_test and bubblesort, while placing second for Grendel. Although its CPI is the lowest of the three, its cycle time is nearly double that of either of the pipeline designs. One could also assume the hardware-scheduled to be the best because it had the fasted time for Grendel and base_test, but interestingly, it came in second for bubblesort, making it worse than the software-scheduled in certain scenarios.

When we compare our next design, the software-scheduled pipeline, to our single-cycle processor, we observe a higher CPI and instruction count in exchange for a significant reduction in cycle time. The increase in CPI is quite small and is primarily affected by the instruction fill at the beginning of each application. This explains why the CPI is lower for a larger application like Grendel and higher for the simpler base_test. Even though the CPI is nearly the same and the cycle time is less than half as long, Grendel on the single-cycle processor runs faster. This is due to the considerable number of nops required within Grendel for the software-scheduled pipeline, which is reflected in the more than double instruction count. Grendel has numerous data hazards back-to-back, many of which cannot be rearranged. It is no surprise that Grendel had over double the instructions, considering around half the instructions needed two to three nops in between them. Looking at the other two applications, the software-scheduled versions did not need as many nops between them, making instructions count less than double. This allowed the low cycle time to take advantage and finish the program faster.

This benefit improved further with the hardware-scheduled pipeline design. The hardware-scheduled pipeline outperformed the single-cycle processor across all programs. Although this processor had a slightly longer clock period compared to the software-scheduled one, it remained significantly faster than the single-cycle design. Furthermore, while the increased CPI made bubblesort slower than the software-scheduled version, avoiding nops aligned the instruction count with the single-cycle processor, resulting in the magnitude of increased cycles being less than the magnitude of reduced cycle time.

When comparing the hardware-scheduled and software-scheduled pipelines, we observe that the hardware-scheduled pipeline outperformed it for base_test and Grendel, though not in bubblesort. Base_test encountered only a few data hazards, most of which required just a single forward. This means that these data hazards did not require stalling and did not affect CPI. Analyzing the instruction count in relation to the cycle count for the hardware-scheduled pipeline, there are only 10 more cycles than instructions. Four of these cycles are due to instruction fills, which means that only six stalls or flushes were needed. In contrast, our software-scheduled pipeline requires two to three nops for each data hazard (we can disregard control hazards since both hardware and software require one flush and one nop, respectively). Consequently, the software-scheduled pipeline ends up adding significantly more nops than the required stalls or flushes, enabling the hardware-scheduled pipeline to complete more quickly, even with a slightly longer cycle period. Since the software-scheduled cycle period is about 1.2 times faster than the hardware-scheduled pipeline, reducing the instruction count down to around 60 instructions would make the software-scheduled pipeline faster. This could be achieved by using different registers and reordering to reduce data hazards and nops.

We have already discussed why the hardware-scheduled processor outperformed the single-cycle processor for Grendel and why the single-cycle processor outperformed the software-scheduled processor. However, the software-scheduled pipeline managed to execute faster than the hardware-scheduled pipeline for bubblesort. Looking at the total cycles, the hardware-scheduled and software-scheduled processors are essentially equal. Even though the software-scheduled pipeline has more instructions, the CPI is lower, and its faster clock period enables quicker execution times. This makes sense because bubblesort involves a nested loop with multiple loads and stores executed back-to-back. These create load-use hazards, necessitating three nops for software-scheduled tasks and numerous stalls for hardware-scheduled tasks. I am curious about why so many stalls occurred. In a double data hazard that requires two stalls and a forward, the hardware-scheduled processor should only stall for three cycles, which is the same number of nops needed. One possibility could be reordering. In the bubblesort_avoidHazards file, we reordered several instructions to reduce data hazards or reduce the number of nops needed. However, the hardware-scheduled pipeline ran the same file as the single-cycle pipeline without any reordering. This suggests that the software-scheduled pipeline might have reordered an instruction so that no nops would be necessary, but the hardware-scheduled pipeline did not reorder and therefore stalled. Assuming this is the rationale, the hardware-scheduled pipeline could outperform the software-scheduled pipeline if reordering and unnecessary data hazard avoidance were incorporated into its applications.

4. **Software Optimization.** Identify and 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.

   Instruction reordering is one software optimization that could improve the performance of the software-scheduled pipeline relative to others. This software optimization means rearranging different types of instructions in a program in such a way that you are able to minimize the number of NOPs needed and effectively use the pipeline stages in our software-scheduled pipeline. In our case, we were not able to minimize any stalls, but we still managed to improve our instruction count by reordering the instructions. This resulted in faster execution time. Without reordering in Grendel and Bubblesort, almost every instruction would need 3 NOPs, but because of reordering, we were able to bring that number down. If we further reordered Grendel, we could possibly make Grendel faster than our single-cycle processor.

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.

The worst-case path for both hardware and software pipelines was the Execution stage. This is from the ALU, specifically the ripple-carry adder. If implementing a carry-lookahead or even faster type of adder, this could reduce the cycle period, possibly making a different stage become the worst-case path. The ALU adder took about 14 ns in the pipeline processors, so a faster adder (like carry-lookahead) could reduce that to around 7 ns. From the signal-cycle synthesis, the DMem took about 11 ns to complete. Assuming it is the same for the pipeline processors, that could make the MEM stage the worst-case cycle period for the software-scheduled pipeline if we move the ALUSrc mux to the ID stage. However, for the hardware-scheduled pipeline, a 2x faster adder would most likely have the worst-case cycle period still be EX stage, since it has the additional latency from forwarding muxes and memToReg mux. Even though the worst-case period is the EX stage, the reduced adder time could still greatly increase the cycle period. This would also benefit the single-cycle design since the worst-case path is loads and stores, which also go through the ALU adder.

For the hardware-scheduled, if the worst-case path is still in EX after changing the ALU adder to a faster design, moving the byte select, halfword select, and some of the memToReg muxs to the MEM stage will help lower the worst-case path based on forwarding. Specifically, the latency added from forwarding from the WB stage is about 3 ns because of those WB stage muxes. Moving those to MEM stage (as long as we are not making the MEM stage the new worst-case path) could reduce the time that the forwarding muxes are waiting for the new memToReg values. This could improve the hardware-scheduled cycle period by up to 3 ns.

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.

The approach to designing a program that performs better on a single-cycle processor involves various load-use hazards and control hazards. These hazards rely on stalls and flushes, which increase the CPI of the hardware-scheduled pipeline. For instance, a loop may include a load that writes to a register that another load reads from in the subsequent instruction. Each branch would require a flush, and every load-use hazard requires a stall. If the CPI is around 2, then the single-cycle processor executes more quickly.

To create a program that runs faster on a hardware-scheduled pipeline than on a software-scheduled, we can refer to Grendel and base_test. A program must exhibit multiple data hazards in such a way that reordering cannot be done to reduce the nops in the software-scheduled pipeline while having data hazards that can be forwarded by the hardware-scheduled pipeline. Forwarding does not require a stall and will not decrease CPI, but it still requires multiple nops for the software-scheduled pipeline.

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.

   One of our group's challenges was during the software scheduled pipeline. We found it more difficult than the single-cycle processor implementation because it was easy to make small errors like forgetting to propagate signals and values through each pipeline. Because each signal needed to be duplicated for each stage it went through, the software-scheduled pipeline was the hardest processor to design. In terms of difficulty, the software scheduled was easier to debug but was harder to implement when compared to the hardware scheduled.

   Another critical challenge was during the hardware scheduled pipeline, where we ended up slowly testing all the test cases and fixing our hazard detection and forwarding unit. The hardest part of this was going through all the data hazards possible. At first, it seemed very overwhelming, but thinking about where each instruction uses or produces a value allowed me to create groups of consuming instructions, producing instructions (how they propagate through each pipeline), and a list of four producers and six consumers. From this, I thought of every scenario where each consuming instruction group would be placed right after a producing instruction group or separated by one instruction. I also thought of how forwarding would work from each scenario, or if a stall was required. After writing them all out, they were easy to implement with a forwarding unit and muxes.

   The overall most difficult challenge occurred after we implemented our forwarding unit and multiplexers. We conducted all of our tests, which passed except for Grendel. This was particularly challenging since Grendel was lengthy, and the traces could not be utilized effectively. This initiated a prolonged search for a data hazard we had yet to implement, until we ultimately discovered it was due to a stall and a forward needing to happen simultaneously. Both actions could execute correctly, but the forwarding data would exit the WB stage in the next cycle and could not be forwarded again. This issue forced us to consider a scenario we had not covered in class or discussed in the textbook. Fortunately, Guru had already encountered the idea of adding an extra register to hold WB data. From this concept, I designed a solution to forward to EX when a stall and WB forward were detected in the previous cycle.

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.