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 for 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, softwarescheduled pipeline, 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.

   This report covers the comparisons of the three(in our case, two…) MIPS-based processors we have developed over this semester. The first processor is the single cycle processor that performs most basic MIPS instructions, including arithmetic, logic, shifting, loads/stores, and jumps/branching. It's rather simple, but it works. Our next processor is our MIPS Software Scheduled processor that is simply just a pipelined version of our single-cycle processor, meaning that we have to go into our test files and manually fix (i.e., add NOPs/bubbles) into our programs to avoid data/control hazards. Generally, we need three NOPs to avoid a data hazard created when a given consuming instruction is sequentially right after a producing instruction, and we need two NOPS after any jump/branch instruction we may have, so that unintended instructions aren't loaded into the pipes. Finally, we have a hardware-scheduled processor that detects data and control hazards, but there's no forwarding.

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. 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

Program	# Instructions	Total Cycles	CPI	Max Cycle	Execution Time
Proj1_base_test	39	39	1	42.294ns	1649ns
Proj1_bubblesort	696	696	1	42.294ns	29436.624ns
grendel	2116	2116	1	42.294ns	89494.104ns

Software Processor

Program	# Instructions	Total Cycles	CPI	Max Cycle	Execution Time
Proj1_base_test	57	61	1.07	23.386ns	1426.55ns
Proj2_bubblesort	1739	2023	1.16	23.386ns	47309.878ns
grendel(NOP)	5703	6221	1.09	23.386ns	145484.306ns

Hardware processor

Program	# Instructions	Total Cycles	CPI	Max Cycle	Execution Time
Proj1_base_test	39	59	1.51	22.988ns	1356.292ns
Proj1_bubblesort	696	1796	2.58	22.988ns	41286.448ns
grendel	2116	5645	2.67	22.988ns	129767.26ns

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.

First of all, the pipelined processor is generally faster, but that's not a hard-set rule. First of all though, we need to go into how we actually calculated our values. To find our execution time, we simply took the total number of cycles, multiplied that by the max cycle time, and made sure to keep the proper units, nanoseconds. Rather straightforward, and here's a nicer equation:

Execution Time = Total Cycles * Time per Cycle.

This is essential to all of our performance calculations, and will remain heavily used. If we look at the results, it's clear that there isn't a true "better" processor between our two, but in programs that need lots of bubbles/NOPS inserted, the software-scheduled pipeline really suffers in comparison to the single cycle processor. It's cycle time is half of what the single-cycle's is, but it requires too many extra instructions/bubbles, and it can't make up the difference. The hardware-scheduled processor is a little faster than the software-scheduled processor, but not by a a massive margin. Another perk it has is that having the processor itself automatically detect data/control hazards saves time for the programmer, and removes some human error from the equation by *potentially* avoiding pointless NOPs, though unfortunately, the hardware-scheduled processor has wrinkles of its own. For starters, we’re assuming there aren’t any issues with hazard detection triggering when it shouldn’t. Second, there’s *no way* to mitigate control hazards without some kind of prediction, which in itself isn’t a 100% successful or guaranteed improvement. As such, even with our hardware-scheduled processor working, it isn’t really much better than our software-scheduled processor. The base test was the single test that both software and hardware actually beat single-cycle in, and that’s because there weren’t many data hazards that needed to be mitigated. As such, its total runtime was 1427 and 1356 ns on the software and hardware processors, respectively. On the single-cycle processor, it was 1649ns, a significant difference.

Software
Effective speed up = 1649/1427 = 15.56%

Hardware
Effective speed up = 1649/1356 = 21.61%

A speed up of up to 21.61% is quite large, but we can take it further if we write a program that has *no* hazards whatsoever. It doesn’t matter what kind of operation we perform, as long as there’s just one per cycle, so we’ll run a program that’s just 25 NOPs, which would accomplish the same thing as 25 SLLs that shift register 0 by 0.

Single-Cycle: Insts: 26, Cycles 26, Time = 26 * 42.294 = 1099.644ns

Software: Insts: 26, Cycles 30, Time = 30 * 23.386 = 701.58ns

Hardware: Insts: 26, Cycles: 30, Time = 30 * 22.988 = 689.64ns

This is an unrealistic example, but it demonstrates *how* you could maximize performance on pipelined processors; simply reduce the amount of hazards to the smallest possible degree. On the other hand, how does one optimize for a single-cycle processor? The answer is, you don’t. The single-cycle processor is the baseline, there’s no tricks to make it run faster, all you can do is reduce the critical path and increase the frequency.

In summary, the fewer hazards you have, the better the program will run when pipelined, whereas if you have more, it can and tends to run better on single-cycle processors if you don’t have sufficient mitigation techniques. The biggest thing that could be done to make the hardware-scheduled processor more competitive with the single-cycle processor would be the addition of forwarding, which enables the processor to handle *most* data hazards without delaying to compensate. This enables one to keep the fast frequency of a pipelined processor AND the instructional efficiency of a single-cycle processor. Here’s another example:

Let's say we have a test program (purely hypothetical as we DON'T have forwarding working, but bear with us here), that contains 50 instructions and 30 data hazards, 15 of which require 3 NOPs, 9 of which require 2 NOPs, and 6 of which require a single NOP. In total, this means we have an extra (15 * 3 + 9 * 2 + 6 * 1) 72 dummied out instructions on a software-scheduled processor. Now, let's calculate our execution times, but for our hardware calculation, ignore the added NOPs.

Single-Cycle: Insts: 50, Cycles 50, Time = 50 * 42.294 = 2114.7ns

Software: Insts: 122, Cycles 126, Time = 126 * 23.386 = 2934.0ns

Hardware (ignoring hazards): Insts: 50, Cycles: 54, Time = 54 * 22.988 = 1241.352ns

This is another cherry-picked example, but it's far more realistic, there just aren't any control hazards. Of course, the critical path of our hardware processor would likely change, but likely not to an extreme extent that would render this jump in performance moot.

4. **Software Optimization.** Identify and describe one software optimization (i.e., assemblylevel software refactoring) that would improve the performance of software on the softwarescheduled pipeline relative to the others. Provide an estimate of the performance benefit this change could have given your specific benchmarks.

Software optimization is very open-ended, so I'll provide a few examples.

One of the most direct ways to improve your program's raw performance is by reducing jumps. By reducing the number of jumps we take in a program, we can reduce the number of unavoidable NOPs we have to eat when we run a program on a pipelined processor. There are lots of ways this can be done, and it depends on how the program is structured. In a program with lots of small individual loops, if possible, one could combine said loops into one loop to reduce the number of jumps. If one was writing a rather short program that utilized jumps, one could instead just write the necessary instructions in sequential order, as if the jump wasn't there (THIS IS NOT TAKING DATA HAZARDS INTO ACCOUNT).

For example:

```
addi $t1, $t1, 0
j bingus:
sll $t0, $t1, 8

bingus:
sra $t0, $t1, 8
```

Can be rewritten as:

```
addi $t1, $t1, 0
sra $t0, $t1, 8
```

You want code that's effectively the same, even if it takes up more space in instruction memory; our concern at the moment is speed, not space efficiency. Here's a better example that I've written up that's less of a sample:

Here's the base program:

```
addi $t0, $t0, 3
addi $t1, $t1, 1

second:
add $t2, $t0, $t1
sub $t0, $t0, $t1
beq $t0, $t1, kill
j second

kill:
sll $zero, $zero, 0
halt
```

Now, here's the modified one:

```
addi $t0, $t0, 3
addi $t1, $t1, 1

add $t2, $t0, $t1
sub $t0, $t0, $t1
add $t2, $t0, $t1
sub $t0, $t0, $t1
add $t2, $t0, $t1
sub $t0, $t0, $t1

sll $zero, $zero, 0
halt
```

It loops three times, no big deal. However, by getting rid of the loop portion and the jump, we can save 4 jump/branch instructions, saving us 4 * 2 = 8 NOPs. This isn't a method that could or should be used all the time, but it's a way to shorten some things. As long as the modified program does effectively the same thing, it's all good. This is not a viable option for programs that have massive, lengthy loops, as the program will end up becoming very bloated.

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: Critical Path Reduction:

Improving performance in a single-cycle processor comes down to improving the speed, or coming up with new instructions that are potentially faster. The latter is not preferred, as MIPS is a RISC language, so any instructions that aren't strictly necessary are not preferred. So, we need to find ways to reduce our critical path. In our case, rewriting our ALU and adding a separate ALU control module would work wonders. Additionally, our branch selection consists of a bunch of sequential MUXes, which does bode well for performance. These could be consolidated into one 4t1 MUX, as we already have a 4t1 MUX and a 2t1 MUX that are sequential, so said control bits (J and JR) can wired to use that extra 4th input that's currently just a copy of the third input. Assuming a 2t1 MUX and a 4t1 MUX have the same delay, this would reduce our total delay in our jump checking logic (all located near the ALU, it's all in the EXECUTE stage in a pipelined processor) by roughly half.

Software: Negative Edge Registers:

One way to improve performance in our pipelined processor (this does include the hardware-scheduled processor as well!) is by clocking the registers to operate off of a negative edge. This enables the registers to update half a cycle ahead, which results in values getting pushed through and updated sooner. Most obviously, this would reduce the number of NOPs needed for data hazard prevention by one, knocking it down to a maximum of 2. To estimate this improvement, all we'd need to do is use a mock program that has maxed data hazards.

For our benchmark, we'll assume we have a program with 20 instructions, and 10 max (3 NOPs) data hazards. For our modified software processor, we'll just cap the NOPs at 2 instead.

Control Software-Scheduled Processor:
20 instructions = 24 cycles + 30 cycles from NOPs = 54, 54 * 23.386 = 1262.844ns

Modified Software-Scheduled Processor:

20 instructions = 24 cycles + 20 cycles from NOPs = 44, 44 * 23.386 = 1028.984ns

1263/1029 = 22.74% speed increase!

Finally, to implement this, we'd just need to adjust our pipe registers to operate on negative clock edges, and if we wanted to take this further and use it in our hardware-scheduled design, we'd need to remove our write stage hazard detection functionality, as it'd be redundant and triggering when it shouldn't.

Hardware: Branch Prediction: There's no way to 100% remove control hazards, but you can try. Branch prediction is an attempt to mitigate them, and it's relatively conceptually simple. Here's how we'd set it up and have it function:

When a branch is taken, increment a counter by 1. This is our "confidence value."
If a branch is taken *again* before any branches aren't taken, increment the counter to 2, and now start treating all branches as if they're guaranteed, only squashing the registers after checking and seeing if a branch wasn't supposed to be taken. If this streak is broken, reset the confidence value to 0. If there are consecutive branches that *aren't* taken, then increase the confidence value for each one until it reaches 2, and then assume that those branches *won't* be taken. Of course, keep checking throughout the pipeline, and if a branch was supposed to be taken, squash IFID and IDEX and reset said confidence value.

If we wanted to implement this into our processor, we'd need to add said counter, and add a condition to our squash signals that would block activating them on a branch when the processor is "confident" that there won't be a branch.

Performance-wise, the first two branches wouldn't be any better, but assuming the trend held (if we were in a long loop, [like 10+ iterations]), we'd save 2 NOPS each time. Here's a formula for loop branch performance: LoopCount * 2 – 2 = Saved NOPs

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 softwarescheduled 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.

   If we look back at the previous tests, there's a pretty clear pattern. Every program, grendel, proj1 test, proj1 bubblesort, all ran better on our hardware-scheduled processor versus our software-scheduled processor. If we had forwarding up and running, this difference would be even more stark. The biggest reason they all run better is because the hardware-scheduled processor has a better critical path, and it's less prone to human-error thanks to unecessary NOPs. If we were to write a program to mimic this behavior, we'd simply need to make one that features plenty of data hazards, as If we had forwarding, it'd only further compound with the reduced critical path delay.

   When it comes to better performance on a single-cycle processor, grendel is a clear example. 89494ns on the single-cycle processor, versus 129767ns on the hardware processor. It's a slowdown of (89494/129767 – 1) 31%. This is of course because grendel has a very large number of jumps, and plenty of data hazards. Even if forwarding was up and running, it seems unlikely that the difference between the two would be skewed much in hardware's favor, given that there's still no way to avoid the +2 NOPs for *every* jump in grendel. If we were to build a program that was designed to run better on a single-cycle processor than even a hardware-scheduled processor with forwarding, we'd need a very large amount of jumps executed, at least enough to add enough NOPs to add enough redundant cycles to just about double the total instruction count to even the difference between the two.

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 major challenge our group faced was a frustrating issue related to file organization that blocked our ability to run the final test scripts. When our Project 1 processor was fully implemented and ready for verification, we encountered an error saying work.awesome could not be found, even though the design compiled manually in QuestaSim without problems. We spent considerable time troubleshooting compile orders and library mappings, but the issue persisted during automated script runs. Eventually, we realized the real problem was the physical location of the awesome.vhd file. The files was mistakenly placed inside the TopLevel directory rather than in the main project directory where the script expected it. Once we moved the file to the correct location, everything compiled and all tests passed. This taught us a valuable lesson about adhering strictly to directory structures and verifying file paths early, especially when using automated build and test frameworks that rely on precise folder organization.

   Another challenge we faced was optimizing the number of NOPs in our Grendel program for the software-scheduled pipeline processor. Initially, to ensure correctness and avoid hazards, we added excessive NOP instructions after nearly every instruction. While this method guaranteed correct execution, it led to extremely poor performance and inflated instruction count. As we progressed, it became clear that many of these NOPs were unnecessary and could be removed through better scheduling and understanding of data hazards. We spent significant time carefully reviewing instruction dependencies and gradually eliminating redundant NOPs while ensuring that the program still executed perfectly and produced correct results. This process taught us the importance of balancing correctness with performance, and in the future, we would take a more methodical approach to hazard analysis before blindly inserting large numbers of NOPs.

   Finally, data hazard detection. Control hazards were rather easy to detect, but it took a very long time to actually understand how to detect control hazards properly. Eventually, we sat down, wrote out the signals, and figured out what signals we actually needed to run into the haz detection unit, and what the logic needed to look like. Eventually, we finally got it done! It took a long while, but once we changed our approach from “how do we mitigate data hazards” to “how do we create bubbles,” it all fell into place. Focusing on *how* you do something is easier than *why* you’d do it, and once we got to that point, it was easy.

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.