CprE 381: Computer Organization and Assembly-Level Programming

Project Part 3 Report

Team Members: Jake Hafele

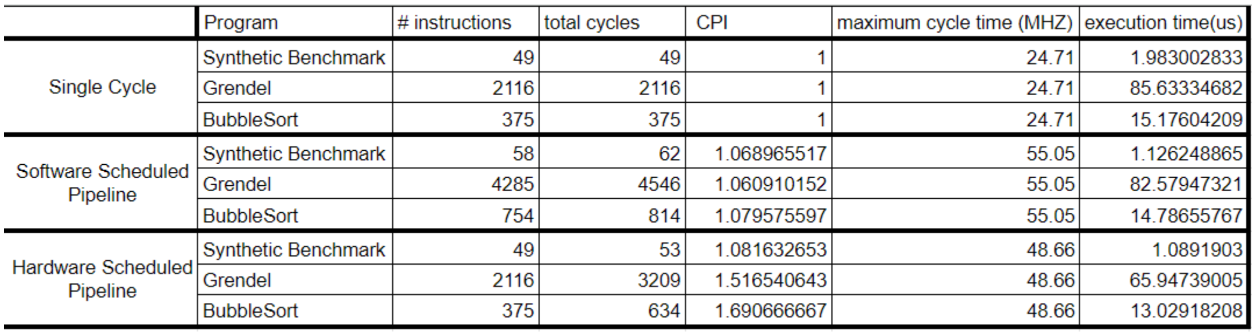
Thomas Gaul

Project Teams Group #: TermProj_2_03 (aka *Marek's little stinkers*)

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

Throughout the semester we have worked on three different processor designs implementing the "basic" MIPS Assembly ISA and a handful of additional branch instructions. As we learned material in class we implemented the concepts into the term project. Starting with a Single Cycle Processor, followed by a Software Scheduled Pipeline, and finally a Hardware Scheduled Pipeline. As we worked through each processor we built off the previous one increasing in its complexity and overall efficiency. To study the performance of each processor we synthesized it and ran a number of test applications on the processor.

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



	Program	# instructions	total cycles	CPI	maximum cycle time (MHZ)	execution time(us)
Single Cycle	Synthetic Benchmark	49	49	1	24.71	1.983002833
	Grendel	2116	2116	1	24.71	85.63334682
	BubbleSort	375	375	1	24.71	15.17604209
Software Scheduled Pipeline	Synthetic Benchmark	58	62	1.068965517	55.05	1.126248865
	Grendel	4285	4546	1.060910152	55.05	82.57947321
	BubbleSort	754	814	1.079575597	55.05	14.78655767
Hardware Scheduled Pipeline	Synthetic Benchmark	49	53	1.081632653	48.66	1.0891903
	Grendel	2116	3209	1.516540643	48.66	65.94739005
	BubbleSort	375	634	1.690666667	48.66	13.02918208

Table 1. Performance Metrics

3. 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.

For our performance metrics (**Table 1**), we determined the number of instructions, total cycles, and maximum cycle time (MHz) for three different software programs. These software programs included a synthetic benchmark using each instruction once, our "final" test that was supplied to us as Grendel, and the BubbleSort process that we developed in Project 1. For the software schedule pipeline in Project 2, we inserted the minimum amount of required NOPs from each assembly instruction. After running each assembly program in the toolflow, we recorded each of the above metrics and calculated the CPI and execution time with the following two equations that we learned in class:

CPI = (Total Cycles) / (# Instructions)
Execution Time = (# Instructions) * CPI * (1/Max Cycle Time)

For our synthetic benchmark program, the Single Cycle processor had the largest execution time, at 1.983 us, while the shortest execution time came from the Hardware Scheduled Pipeline processor with 1.089 us. The software scheduled pipeline took slightly longer than the hardware pipeline, but was relatively close. Throughout our three processors, we were able to attain a speedup of 182% for our basic benchmark of each instruction once. The number of cycles required for the Hardware Scheduled pipeline was 4 cycles more than the single cycle execution due to requiring 4 cycles to fill the pipeline initially. The performance on both pipelined processors was better due to the fact that the pipelining led to **at least** double the maximum clock cycle frequency. Since the number of instructions were relatively close in length, and the CPI was near 1 for each program, the maximum clock frequency was the main impact of execution time. Both pipelined processors were able to increase their cycle time by keeping a similar propagation delay between each pipelined stage, so that one stage would not dramatically increase the cycle time of one stage, in turn delaying the update of the other stages.

In Grendel, it turned out that our single cycle and software scheduled pipeline performed very similarly, with the hardware pipeline dramatically beating both of them in performance. It should be noted that the number of instructions for the software pipeline is double the single cycle pipeline, due to instructions requiring 2 NOPs due to a read after write data dependency or 1 NOP for control hazards. Since there are no NOPs in the hardware pipeline assembly program, the number of instructions matches the single cycle, and instead leads to a higher cycle time due to stalls and flushes for hazards. This leads to a marginal difference in the CPI between the hardware pipeline and the single cycle or software pipeline. Since the hardware pipeline had a low number of instructions and a large maximum cycle frequency, it was able to beat out both the low cycle frequency of the single cycle and the high instruction count of the software pipeline.

Finally, we were able to analyze the BubbleSort program. Similar to Grendel, the single cycle and hardware pipeline both have the same instruction count, with the software pipeline having nearly double the number of instructions due to added NOPs. The CPI for the single cycle was near 1, with the software pipeline following suit. The hardware pipeline had the largest recorded CPI of any program, at 1.69.

These added cycles were due to needing to flush 1 instruction on every branch taken, and having to stall once for lw instructions. While the hardware pipeline still had the fastest execution time, at 13.029 us, the software pipeline and single cycle pipeline were much closer than Grendel, clocking in at 14.786 us and 15.176 us respectively. Due to the hardware pipelines large CPI, the execution time was not able to perform much better than the software pipeline, even with the software pipelines large instruction count. This performance raises concern, since this program included many data hazards and control flow flushes that would be used in any practical assembly program.

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.

A software optimization we could do to improve the performance of the software schedule processor would be to reorder the sequence of instructions to remove data hazards. When rewriting the code to run on the software scheduled pipeline we only inserted nop's when data hazards were present. Reordering would decrease the number of total instructions in the Iron Law for calculating execution time. There is one NOP per control hazard and up to two NOPs for data hazards. We would reason that we could easily remove a quarter of the NOPs by reordering with the rest made up of unavoidable data hazards and control hazards. For instance, in Grendel there are roughly 2000 NOPs which could decrease to 1500, conceivably decreasing the execution time from 82MHz to 73MHz so a 1.12 speedup.

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.

To have the single cycle processor work faster we can break up the critical path. At the moment the critical path moves through the ALU to the DMEM and then to writeback in a load word instruction. To make this run faster we can break apart our ALU to have all portions of it with the exception of an adder to happen in parallel with the data memory. The adder still needs to occur for data memory in address calculations but the shifter can happen at the same time as Dmem. Additionally the adder can be changed from a ripple carry adder to a lookahead carry adder to shorten the critical path more. Currently the critical path goes through the shift register to the DMEM which is a case that will not occur in our processor. Between those two changes I would estimate we could remove about 2 ns from the delay in our critical path. This would make the running time of one cycle to got to 43 ns to 41 ns or about a 1.05 speedup.

$$Speedup = \frac{Running\ Time\ Old}{Running\ Time\ New} = \frac{43.482\ ns}{41\ ns} = 105\%$$

One improvement we could make for both our software pipeline processor is to increase the number of stages from 5 to 6, to reduce the critical path latency. In our software scheduled pipeline, our critical path was in the ID stage, propagating from the register file, to the branch module, then to our prefetch module. The total latency time for this critical path was 21.169 ns. Based on our analysis of the memory modules, both the instruction memory and data memory take around 15 ns to perform. So, if we were able to split the EX stage into two stages, each with a critical path equal or lower to the IF and DMEM stages, then we could increase our maximum clock frequency for the software scheduled pipeline. To do this, we would have to create another pipeline register between the newly added stage and the EX stage. We could split the ALU up into two parts, even splitting the ripple carry adder into two halves of 16 bits per stage. With one extra stage added, this would increase our number of cycles by 1, but would relatively still lead to a CPI of 1 with enough instructions ran in a program such as Grendel. So, the speedup would be as follows:

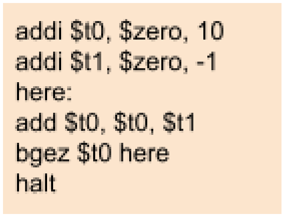
$$Speedup = \frac{(\#\ instructions)\ (CPI\ Old)\ (Max\ cycle\ time\ Old)}{(\#\ instructions)\ (CPI\ New)\ (Max\ cycle\ time\ New)} = \frac{1*(18.16\ ns)}{1*\ (15\ ns)} = 121\%$$

While the estimate of 15 ns for our DMEM and IF stage is ideal, this proves that a speedup could be possible if we improve the performance of our EX stage. After this, it would be expected that our ID stage is on the critical path, where we could then move some of the contents in that stage to our newly added stage to pull the critical path lower and aim towards the 121% speedup.

One change to improve the hardware pipeline performance would be to include forwarding for the RS and RT data inputs to the conditional branch module that is located in the ID stage. If we added forwarding to this module, then we would not be required to stall for 2 clock cycles while the ALU output is buffering through each stage, until they reach the WB stage. This would be beneficial since a branch instruction already requires a flush when a branch is taken, which would help to reduce the CPI of **all** branch instructions, whether a branch is taken or not, since it could determine if a branch is taken earlier. Similar to the ALU input forwarding, a select line would be needed from the forwarding module to drive the input multiplexers for the branch condition inputs of RS and RT data. We would also need to implement two 2-1 MUXes, one each for the respective inputs. Data could be forwarded from the DMEM stage after an ALU output. Since the branch condition is in the ID stage, it would be irrelevant to forward from the WB stage since the conditional can retrieve the proper RS and RT data contents if data is being written to those addresses in the ID stage already. Unfortunately, we may still need to stall once, if there is a read/write dependency for an instruction in the EX stage. It may be possible to feed the ALU output directly to the branch module, but this could lead to a change in critical path and impact our maximum clock frequency. Based on the percent of instructions that are branches, and how many read/write dependencies could occur right after ALU instructions for branches, this change may also be useful.

6. 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.

To achieve a slower runtime on the hardware scheduled pipeline than the single cycle we had the cases where we did not/ could not remove stalling. The two cases we focused on in the program was data hazards with branch instructions and control hazards where the branch is taken. The branching data hazard has two stalls and the control hazard adds 1 stall per loop. With this test case we achieved a runtime of 1.01 us on the single cycle



```
addi $t0, $zero, 10
addi $t1, $zero, -1
here:
add $t0, $t0, $t1
bgez $t0 here
halt
```

and a runtime of 1.25 us for the hardware scheduled pipeline. In each loop there is a control hazard with the bgez instruction and a data hazzard with the addi preceding it causing us to stall 3 times per loop.

Every assembly program that we identified in the benchmarking and performance analysis sections had a better execution time for the hardware pipeline than the software pipeline. As discussed above, this is due to the hardware pipeline having a marginally lower instruction count than the software pipeline, which leads to better performance despite the increase in CPI due to hardware stalling and flushes. The extra instructions included in the software pipeline are due to NOPs but are still slower than the hardware pipeline despite having a larger maximum cycle frequency.

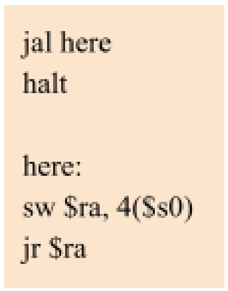
7. Challenges. This term project was challenging for every group. In at least three detailed paragraphs, describe the three most critical challenges your group faced, how you resolved them, and how you could avoid them in the future.

Overall, we were very satisfied with how each project was covered. We both were very proactive in completing our tasks and were ahead of our schedule that we set in our team contract. This class has been extremely rewarding to the amount of effort put in, and each processor taught us something new.

Over the semester, our main challenges included:

1. Determining when to flush/stall to handle control hazards
2. Instructions rolling over halt before it reaches WB stage
3. Debugging SW/LW with forwarding
4. Implementing branch zero instructions in ALU

One of the most rewarding hours of work that came of the semester was when we sat down with a piece of paper and discussed which pipelined processors would be stalled or flushed. We talked about this for a while, since we decided very early that we wanted to move our conditional logic to the ID stage to reduce the amount of NOPs in our software pipeline or the amount of flushes in our hardware pipeline. Right before this discussion, we were working on how to flush and stall our pipelined registers, and this discussion also helped to solidify that. Initially, we were thinking of using an asynchronous reset to clear the register contents. But, as we later covered in class, this was a bad idea since it would wipe away whatever was in the following pipeline register from propagating through on the next clock cycle. To resolve the issue with flushing, we drew a sample set of instructions where instructions would follow after the branch instruction. Then, when the branch instruction was in the ID stage, we drew crosses over the pipeline that needed to be flushed, which was the IF/ID pipeline. We learned that by using one set of test instructions and drawing over our schematic, we could solve general design problems at a top level without getting in the weeds of VHDL.



```
jal here
halt

here:
sw $ra, 4($s0)
jr $ra
```

The program to the left led to an amazing debug session that made us consider every single possibility before the bug was found. Was there a data hazard before the store word? Was there a dependency hazard between sw and jr? Was jump and link an issue? It turned out none of them were! The issue was that after halt was decoded in the ID stage, there were ***<u>TWO</u>*** instructions following this halt control that could propagate through the pipeline before the execution stopped, when halt was in the writeback stage. So, with the above code, in those two instructions before halt was asserted in the wb stage, the final jump register instruction was able to execute ***AGAIN*** when halt was in the writeback stage! So, to fix this, we included the ex and wb stage halt control signals as inputs to the data hazard detection module, asserting that stalls would occur if the halt signal was present and propagating through the pipeline, so we would NOT accidentally branch or jump to another set of instructions accidentally before finishing the program. This bug was prevalent in both our software and hardware pipeline, and was not caught by Grendel. For true evilness, we think this should be added at the end of Grendel. We learned so much about debugging, by meticulously going through each potential issue that could relate to either jump instruction or the store word instruction. We assumed there was an issue with store word, since we had troubles implementing the store word forwarding and load word stalling.

Another major challenge for this project was handling the data storing and loading instructions sw and lw with forwarding and stalling. We did not plan on initially including forwarding for the data being written into the data memory for a store word instruction. After implementing our two planned forwarding multiplexers, for both of the ALU inputs from the DMEM and WB stage, we determined that it would be easier to also include a MUX in the EX stage to forward the data contents for a sw instruction. We also encountered a problem with the fibonacci instruction, with two consecutive sw instructions. If two sw instructions occurred in a row, we were unable to properly forward either instruction, since the content is loaded in the DMEM stage and could only be forwarded in the WB stage, after being written to the DMEM/WB pipeline register. To solve this, we decided to stall one cycle if a lw instruction was detected in the EX stage. To improve the performance of the hardware pipeline, we could perform a hazard detection and only stall if there is a dependency when sw is in the DMEM stage. Similar to before, this issue was worked through by drawing out the flow of each instruction in our hardware pipeline.

One of the first major bugs we ran into was testing the branch instructions of bltzal, bgezal. These were found after running Grendel and it working properly so we concluded our processor was flawless. As it turned out Grendle did not test these instructions and due to their and link properties they were not working quite properly. After digging into the waveforms we found that the instruction would update the ra register regardless of if the branch was taken on not. Eventually the bug was found after digging into the waveforms. This was fixed by putting a mux ahead of the write enable input to the register file and having it only write on those instructions if the branch was taken. To avoid mistakes like this in the future we think about ensuring that we have a complete understanding of full features, quirks, and edge cases. This would help prevent a case like this from occurring again and needing to fix it after a few hours of debugging with a bit of a “band-aid” fix.

The main takeaway from all these issues is the benefit of talking through issues with another person and working through examples on paper or somehow manually working through the processor’s path. Everytime we worked on the processor together, some computer in the lab or TLA got more fingerprints of the screen from us pulling up our schematic and pointing at different locations as we traced instruction’s path through the processor.

Appendix M (Memes):

Steven, you have a mission to rank all of these memes from best to worst. We know you will not fail us.

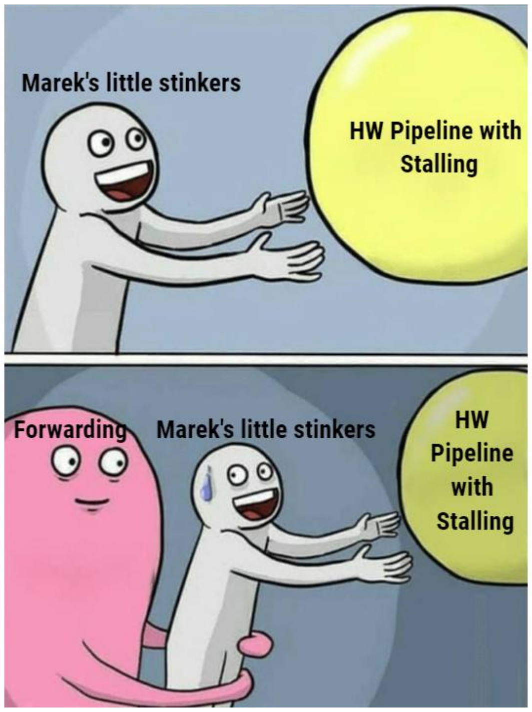


Figure 2. A Tasteful Meme

Figure 3. A Tasty Meme

Figure 4. A Tastier Meme

Figure 5. Thomas also made this one it's not my fault

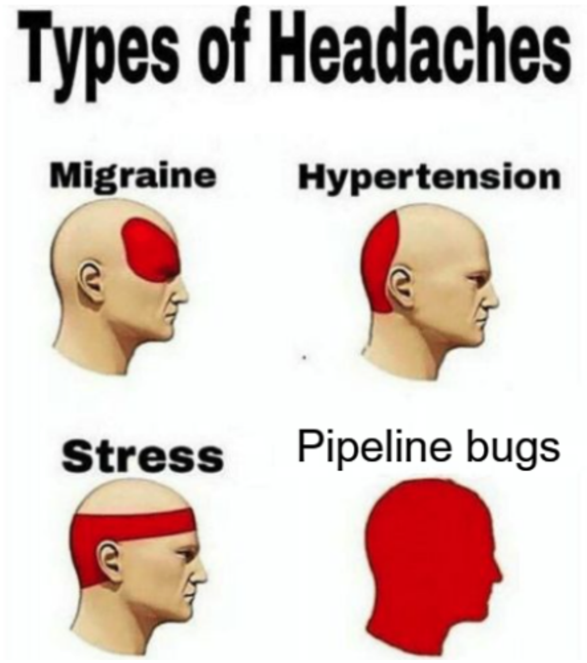


Figure 6. Tastiest Meme

Figure 7. Halt being sneaky

```
JAL HERE
HALT
HERE:
SW $RA, 4($S0)
JR $RA
```