ECEN 4593: Computer Organization Final Project

Andrew Lockwood, Austin Alberts

May 5, 2017

Contents

1 Introduction

The goal of the Computer Organization final project was to teach us how a pipelined processor functions, how to address and mitigate various hazards that arise in such a pipelined architecture, how a processor interacts with data and instruction caches, and how the CPI of a processor can be greatly effected by the addition of all of these levels of abstraction. Over the course of this project, we have had the opportunity to have a hands-on experience with these various interconnecting principles, and have had the chance to demystify computer architecture. This report will encompasses the results of our simulations, and above all, what we have learned from the computer architecture process.

2 Running the Simulation

2.1 Installing and Running Simulation

The code was developed on a Mac/Linux platform using C++. We thought C++ would be the best way to approach this project for its inherent ability to have underlying functions, parameters and classes. We will both be working in object oriented programming languages after graduation, so we had determined that C++ would be in ou best interest moving forward.

2.2 User Modifiable Variables

Our simulation is configured for usability. We have provided the user the ability to configure caches to be either on or off, the ability to change the size of the caches, and also the ability to select which of the two programs the user would like to run. These variables can be found in *main.cpp*, and begins on line 36.

When configuration changes are desired, caches can be toggled by the variables iCACHEON and dCACHEON defined on lines 38 and 39 in *main.cpp*. Set each of these two variables to true to turn the caches on, or set to false to turn them off. The first function in int main located on line 49, is responsible for loading the desired program into main memory. To change between the two, the user must change that function from transfer_Program2() to transfer_Program1().

2.3 Compiler and Environment

In order to compile and execute our program, we created a makefile to assist. The makefile will look for all header files in the inc directory and all source code in the src directory. Running *$ make* in a terminal with a g++ compiler installed will create an executable called output, which will run the simulation.

2.4 Caches and Simulator Implementation

The I-Cache and D-Cache are instances of the same cache class, defined in *cache.hpp*. Their cache size and words per block can be updated through the iCACHE_SIZE, dCACHE_SIZE, and WORDS variables defined at lines 40, 41, and 42 respectively in *main.cpp*. Our project is correctly implemented, except for the D-Caches, as they are still not returning the desired results. We included branch detection in the decode stage, hazard detection, a simulation of pipeline stalls, forwarding, and instruction cache. The code runs all combinations of the I-Cache on Program 1 and Program 2. The program returns the correct values when D-Cache is configured for WORDS = 1 and write through for Program 1.

3 Self-Assessment of the Simulator

3.1 The Pipeline

The pipeline in our simulation was close to what we expected from it. The pipeline had the branch detection occurring in the decode stage of the pipeline (found in *Instr_ID.cpp*). The pipeline also implemented delayed branching, to minimize the amount of stalls that must occur. Forwarding is simulated in our pipeline, with the cases where stalls must be inserted to remove hazards located in the *Instr_MEM.cpp* file.

In order to keep our pipeline simulation as close to an actual pipeline as possible, we calculate the load and store addresses in the execute stage (*Instr_Exe.cpp* for pipeline stage and *exe.cpp* for actual functions), and handle the acquiring or storing into memory in the Memory Stage (*Instr_MEM.cpp*). The Write Back stage occurs before the decode stage of the next instruction, as it should be in an actual pipeline. All of the #defines for instruction types, bit masks, and opcodes and function types are located inside the *Instr_IF.hpp* header file.

3.2 Loads and Stores

In our simulation, we used word addressable memory, and that provided several challenges. The first being that loads and stores needed to be handled in a special case. The address calculated in the execute stage of the pipeline needed to be shifted left by 2. This is because the address is in bytes, where as the memory location we use is in words. When storing bytes as well, we learned that the byte has to be added to the byte location in memory, and it should not overwrite the entire word.

3.3 Forwarding

When it came to forwarding, we had to think about more than just one instruction after the other. To determine all of the instances where a stall is needed, or where a nop needs to be inserted. In our simulation, we only kept track of the previous instruction. To cover, for instance, a load and then a branch two instructions away, we learned that we needed to add a structure to cover two instructions back. Once we did this, we could then track these errors and insert nops where they were needed, and bringing our total clock cycles closer to what was expected of the simulations.

3.4 Branches and Jumps

We learnt much about how to deal with branches and jumps during the course of this project. In regards to branches, we learned the reason behind moving branch detection to the decode stage of the pipeline. We learned that the logic behind it and also the logic behind delayed branches are to eliminate stalls should a branch be in question. Something we also learned in regards to branches, is that it is not just the program counter added to the immediate 16 field, but rather the immediate 16 value added to the program counter plus 1 (for word memory, 4 for byte addressable memory).

Jumps were also tricker with word addressable memory. In byte addressable memory, the immediate 26 field is shifted left by 2, and then given the 4 highest order bits of the program counter as their own. In our word addressable simulator, we learned that you could not approach jumps in this way. Instead of shifting it left by 2, you just add the program counter's highest 4 bits, shift that value right by 2, and then add it into the jumps instructions highest order bits, allowing it to access words instead of bytes.

3.5 Working with Caches

One of the most difficult lessons learned in this project was how to integrate caches into a pipeline. Conceptually, we spent most of our time attempting to optimize the simulation to allow all cache size block to correctly function. We arrived at those equations for the I-Cache after serious examination of caches, and after tweaks, were able to get accurate results for it.

While the I-Cache took a large amount of time to conceptualize, the D-Cache was much, much more. The most difficult part of the D-Cache was figuring out how to store the information correctly in the cache, and how to implement the write back policy. Despite having a strong theoretical understanding of how the cache should work, and the difference between the write policies, we could not get the code to work as we wanted it to. These "setbacks" made us further our knowledge of how caches interact with their environment, and in general, understand their purpose more than when the project initially began.

4 Program 1 Test Results and Conclusion

Due to the fact that our D-Cache was not fully operational, we were not able to acquire the full results for Program 1. In the table below, we have the results for the simulation with the D-cache scenarios we could perform, and **solely with the I-Cache for the other results**.

4.1 Program 1 Results Table

Program 1 Results							
I-Cache Size	D-Cache Size	Block Size	Write Policy	I-hit rate	D-hit rate	CPI	Clock Cycles
None	None	N/A	N/A	N/A	N/A	1.28	607,129
128	256	16	WT	99.88	N/A	1.312	622,335
128	256	16	WB	99.88	N/A	1.312	622,335
128	256	4	WT	99.87	N/A	1.294	613,494
128	256	4	WB	99.87	N/A	1.294	613,494
128	256	1	WT	99.73	41.74	1.917	909,353
128	256	1	WB	99.73	N/A	1.299	616,049
64	1024	16	WT	92.70	N/A	3.187	1,511,463
64	1024	16	WB	92.70	N/A	3.187	1,511,463
64	1024	4	WT	99.19	N/A	1.364	646,782
64	1024	4	WB	99.19	N/A	1.364	646,782
64	1024	1	WT	97.63	97.26	1.47	697,745
64	1024	1	WB	97.63	N/A	1.446	685,809

4.2 Program 1 Results Discussion

As previously stated, only some of these results contain the D-Cache in a fully operational state, however, as seen from the clock cycles of the no cache model, the results are very close to what they should be.

It can also be seen that the clock cycles value for the no-cache configuration varies slightly from what the expected result should be. The reason for the variation from the provided no cache model is that, in our pipeline, the insertion sort loop is being run an extra 8 times. This is resulting in at least 100 clock cycles being added to our total clock cycle value. It also results in an excess 96 extra I-Cache hits, which is due to the fact that it is sitting in a loop, and is therefore triggering more hits. These factors cause the results to be slightly higher than expected, but would be very accurate if the pipeline was working 100% as expected.

If we had been able to fully integrate the D-Cache, then the clock cycles would be very close to the expected results with the D-Cache. Based on the behavior of the I-Cache, and with our knowledge of how D-Caches should work, we can make an approximation as to which configurations are the best. We would say that the 4 word configuration are the best suited cache configurations for this program. The fact that this application has a large use of loads and stores furthers this conclusion.

5 Program 2 Test Results and Conclusions

The results and issues faced for Program 2 were similar to Program 1. As we could not fully integrate the D-Cache, **these results are only with the I-Cache implemented, and the no-cache implementation for further reference**.

5.1 Program 2 Results Table

Program 2 Results							
I-Cache Size	D-Cache Size	Block Size	Write Policy	I-hit rate	D-hit rate	CPI	Clock Cycles
None	None	N/A	N/A	N/A	N/A	1.189	14,440
64	512	16	WT	83.87	N/A	5.218	63,389
64	512	16	WB	83.87	N/A	5.218	63,389
64	512	4	WT	86.31	N/A	2.582	31,364
64	512	4	WB	86.31	N/A	2.582	31,364
64	512	1	WT	69.67	N/A	3.309	40,200
64	512	1	WB	69.67	N/A	3.309	40,200
256	128	16	WT	99.88	N/A	1.216	14,778
256	128	16	WB	99.88	N/A	1.216	14,778
256	128	4	WT	99.64	N/A	1.225	14,880
256	128	4	WB	99.64	N/A	1.225	14,880
256	128	1	WT	98.78	N/A	1.274	15,480
256	128	1	WB	98.78	N/A	1.274	15,480

5.2 Program 2 Results Discussion

From the no-cache implementation, we can see that the clock cycle count is very close to the desired results (within 0.2% of the desired value). As with Program 1, the results were with the I-Cache only. We can see from the results that the clock cycles and CPI would also be very close to the desired results had the D-Cache been fully operational.

From our knowledge of how caches work, and how they effect the pipeline, we could say that the 4 word configurations for the caches would be the most optimal configuration for this program. The fact that is program emphasizes a balance between loads/stores and arithmetic instructions furthers this conclusion.

However, from the results that we could see in the table above, we would get that the most optimal result would be the 16 word configuration with the I-Cache having a size of 256 Bytes and the D-Cache having a size of 128 Bytes. If we had fully integrated the D-Cache, we would expect the results to fall more in line with what our theoretical logic would say.

6 Simulation Results Summary

Pipelining is a process that has been around since 1913 with Henry Ford's assembly line. It was just a matter of time before this technological breakthrough was applicable to computers. Pipelining in general is a good idea to implement. The only trade-off comes with latency. If the design requires a quick start up, then the implementation of a pipeline would need a serious analysis. Otherwise, general computations work excellent in a pipeline because the throughput is dramatically increased. This is a characteristic that most people care about. Usually they do not mind a lag in the first result because the process happens for a long period of time. This initial delay is compensated for by the run time of the procedure.

Even though we were not able to implement many desired configurations, we were able to learn that all coding can benefit from caching, but the extent to which they benefit varies dramatically on the code. Repeatability, and the size of the repeatable code, is a huge influence on the effectiveness of caching. If an aspect of the code is repeated numerous times with a lower amount of coding lines, a small cache will greatly increase performance, but adding more memory to the cache will only provide subtle improvement. The best way to implement caching is to thoroughly understand the code, and the performance characteristics that could be improved upon.

7 Lessons Learnt

Over the course of the project, we had the opportunity to learn much about the inner-workings of a MIPS processor, the relationship between a processor and separate data and instruction caches, and the hazards and dependencies that occur in a 5-stage-deep pipeline. Working through this project provided many instances where we could learn in more depth about the inner-workings of a MIPS processor.

An aspect of this project that difficulties arose was the beginning design. We feel that mid-way through our project we may not have been using the best approach, but due to major time considerations due to debugging the code, we had to push forward with the design instead of being able to adopt a new process.

One huge lesson learned was the use of unit testing. Anyone could attempt to code this whole process and then run it at the end. This would produce errors that are almost untraceable. Every new feature that we added to our project we had to consider the most effective way to test this feature before adding anything else in the project. One of the biggest issues was not being able to see the memory array during this unit testing. The only way for us to view memory was to print out the memory buffer after every execution of an assembly statement. We instead printed out registers to an excel sheet that was tab delimited to determine what registers were changing with each instruction to determine if this was correct.

Overall the biggest lesson learned was how much humans, including us electrical engineers, take for granted when we pick up a computer or cell phone. The amount of design, unit testing, integration testing, verification and validation that go into a mobile device was seen and appreciated through this effort.