Charlie King

Justin Casali

5/4/17

ECEN 4593: Computer Organization

**Final Project Report:**

**How to Run Simulation:**

In order to run our programmed MIPS simulator, simply extract the contents of our provided .zip file, compile the code with the included .h files, and then execute the main.cpp file. Although the code should already be written to execute Program 2, this can be changed in the main.cpp file by changing the name of the desired file to be read in the readProgram() function. Also, if it is desired to change the cache functionality, this can be done in the initial lines of code of the cacheFuncs.h file.

**Self-Assessment:**

With the code we have written, we have implemented several features in an attempt at optimizing the functionality of our instruction execution. First, we made sure that our initial pipeline was implemented properly by running it as a single-stage pipeline, where only one instruction is processed at a time. Then, we implemented the multi-staged pipeline that will be processing up to five different instructions in one clock cycle. In order for this revised pipeline to function properly, we had to implement a combination of forwarding, stalling, and branch and jump detection.

Once the pipeline had been successfully implemented and the program was returning the correct information, we decided to implement the instruction cache, initially with no early restart. We found that doing so greatly reduced the cycle count of the program, as we were no longer having to access main memory nearly as often. We then implemented the early-start functionality and found that the clock cycles were reduced quite a bit.

Then, we implemented the data cache with a write-through procedure, where main memory was written to with every implementation of the cache. Although the cache still writes to memory quite a bit, we found the cycle count reduced quite a bit as the data could be read from the cache and not directly from memory.

We then decided the implement the write-back data cache, which greatly reduced the cycle count, as the program was reading and writing from main memory *much* less often. Although troublesome at times to code, we eventually achieved full functionality with this cache.

Ultimately, we managed to get every aspect of the project working, including the basic pipeline without caches, the instruction cache with and without early start, and then finally both caches working together with both write-through and write-back working properly.

**Test Results:**

After completing our MIPS processor emulation, we can test it with programs that have been compiled to assembly. In doing so, we can measure certain statistics such as the CPI, d-cache hit rate, and i-cache hit rate in order to determine the best configuration of parameters pertaining to the cache.

After running the provided Program 1 for all of the desired parameters, the results are as follows:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| PROGRAM 1 | | | | | | Instructions: | 474137 |
| i-cache size | d-cache size | block size | wb/wt | i-hit rate | d-hit rate | CPI | total cycles |
| no cache | no cache | n/a | n/a | n/a | n/a | 3.744 | 1775275 |
| 128 | 256 | 16 | wt | 99.84% | 97.81% | 2.583 | 1224805 |
| 128 | 256 | 16 | wb | 99.84% | 97.81% | 1.616 | 766207 |
| 128 | 256 | 4 | wt | 99.83% | 93.45% | 2.561 | 1214473 |
| 128 | 256 | 4 | wb | 99.83% | 93.45% | 1.593 | 755135 |
| 128 | 256 | 1 | wt | 99.73% | 74.90% | 2.891 | 1370907 |
| 128 | 256 | 1 | wb | 99.73% | 74.90% | 1.926 | 913059 |
| 64 | 1024 | 16 | wt | 94.98% | 99.54% | 4.745 | 2249855 |
| 64 | 1024 | 16 | wb | 94.98% | 99.54% | 3.778 | 1791257 |
| 64 | 1024 | 4 | wt | 99.34% | 99.74% | 2.378 | 1127603 |
| 64 | 1024 | 4 | wb | 99.34% | 99.74% | 1.409 | 668265 |
| 64 | 1024 | 1 | wt | 98.13% | 98.98% | 2.462 | 1167363 |
| 64 | 1024 | 1 | wb | 98.13% | 98.98% | 1.496 | 709515 |

These results provide us with some very useful information as to the performance of our emulated processor. First of all, we can see just how much of a difference having a properly configured cache can make. We can also see that the write-back caches seem to be *much* more effective than the write-through caches overall. We can also see that optimization isn’t quite as simply as using the biggest cache possible. In our case, the most optimal configuration was an i-cache size of 64, a d-cache size of 1024, a block size of 4, and a write-back cache. We can also see that, for each cache size, a block size of 4 is always the most optimal. With this, you can see just how much more efficient our processor was, as the program was completed in almost a third of the clock cycles without any cache at all.

Next, we can look at Program 2, the results of which can be seen below:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| PROGRAM 2 | | | | | | Instructions: | 12153 |
| i-cache size | d-cache size | block size | wb/wt | i-hit rate | d-hit rate | CPI | total cycles |
| no cache | no cache | n/a | n/a | n/a | n/a | 2.329 | 28303 |
| 64 | 512 | 16 | wt | 88.14% | 94.52% | 7.215 | 87679 |
| 64 | 512 | 16 | wb | 88.14% | 94.52% | 6.833 | 83037 |
| 64 | 512 | 4 | wt | 83.90% | 94.75% | 4.325 | 52565 |
| 64 | 512 | 4 | wb | 83.90% | 94.75% | 3.955 | 48061 |
| 64 | 512 | 1 | wt | 69.83% | 81.93% | 4.545 | 55231 |
| 64 | 512 | 1 | wb | 69.83% | 81.93% | 4.224 | 51335 |
| 128 | 256 | 16 | wt | 92.07% | 57.56% | 7.236 | 87933 |
| 128 | 256 | 16 | wb | 92.07% | 57.56% | 7.061 | 85811 |
| 128 | 256 | 4 | wt | 99.72% | 84.35% | 1.852 | 22513 |
| 128 | 256 | 4 | wb | 99.72% | 84.35% | 1.531 | 18605 |
| 128 | 256 | 1 | wt | 99.04% | 73.79% | 1.860 | 22599 |
| 128 | 256 | 1 | wb | 99.04% | 73.79% | 1.540 | 18719 |
| 256 | 128 | 16 | wt | 99.90% | 57.33% | 3.707 | 45053 |
| 256 | 128 | 16 | wb | 99.90% | 57.33% | 3.537 | 42985 |
| 256 | 128 | 4 | wt | 99.73% | 84.24% | 1.853 | 22519 |
| 256 | 128 | 4 | wb | 99.73% | 84.24% | 1.540 | 18717 |
| 256 | 128 | 1 | wt | 99.04% | 73.79% | 1.860 | 22599 |
| 256 | 128 | 1 | wb | 99.04% | 73.79% | 1.561 | 18967 |

From this information, we can again see that the most optimal configuration isn’t necessarily the largest cache. We can also see here that there were several configurations of the cache that actually had a higher cycle count and CPI than without a cache at all, so it’s interesting to see that using a cache could potentially slow down a program. Overall, it can seen that the most optimal configuration here is an i-cache size of 128, a d-cache size of 256, a block size of 4, and a write-back cache. We can also see here that, again, write-back is always a better policy than write-through. Also, for all of the different cache sizes, a block size of 4 is always the most optimal.

Overall, looking at our results, we can see some interesting results. First of all, it’s clear just how much the implementation of caching can change the runtime of a program, as the number of cycles and the CPI changed *drastically* for different configurations. Also, we can see that write-back always seems to be a better cache policy than write-through. In addition, for whatever reason, a block size of 4 seems to always provide the most optimal CPI and cycles count. Otherwise, as mentioned earlier, we can see from these results that, contrary to what one may think, it’s not best to just make the cache as large as possible in order to optimize performance.

**Lessons Learned:**

Throughout this project, we learned a great deal about the MIPS processor. Namely, we understood why much of its functionality is so necessary. Implementing it from scratch, you come to understand the processor incredibly well, and it was interested to create a pipeline, observe that errors were occurring in the pipeline, and realize that aspects such as forwarding and branch and jump detection are required for full functionality, reinforcing what was taught in lectures and in the book. Learning why these aspects are necessary through implementation was a great way of reinforcing the ideas.

We also learned about how powerful a reasonably simple idea can be when implemented on such a large scale. The idea of the pipeline itself isn’t massively complicated, and it’s reasonably easy to run through its execution for a few instruction at a time. It’s then impressive to know that the pipeline will hold up for thousands and thousands of instructions without any failure, and that they can all execute so quickly. Seeing the simplicity of a processor implemented to a such a large degree is truly impressive, and serves as a great learning experience.

Overall, the project was an incredibly interesting way of developing an understanding of the MIPS architecture and everything that must be implemented for a successfully functioning pipeline.