ECE 411 - Final Report

Team NaN

Naveen Nathan (nnathan2)

Neo Vasudeva (neov2)

Anchit Rao (anchitr2)

May 10th, 2021

Table of Contents

Introduction

As computer engineers at UIUC, we entered this school with the end goal of developing a high-level understanding of the inner workings of computers and specifically that of processors in this modern age. This class, and this project specifically, presented us with the unique opportunity of studying, designing, and implementing one of the most widely used and studied instruction set architecture: RISC-V.

RISC-V is an open-source instruction set architecture used around the world as one of the standards for efficient and high-performance architecture sets. RISC-V covers register and memory accesses, instruction jumps, branches, advanced memory operations, and more. This instruction set is either used or directly influenced by the ISAs for every computer that we surround ourselves with today. We were able to demonstrate our knowledge of the RISC-V instruction set by designing the basic instruction set along with the instruction pipeline using System Verilog and also implementing extra features to improve the overall performance of the baseline ISA.

In this report, we hope to cover everything from the basic ideas implemented up including the advanced features that we chose and the reasons behind our decisions. The progressions from each checkpoint and explanation behind our advanced features and design choices will be covered as well.

Project Overview

Implementing an ISA of this magnitude presented us with many different obstacles and opportunities to understand how large projects are tackled in the real world. We were able to not only face the technical challenges that came with trying to design such a comprehensive code base such as the RISC-V architecture, but we also had to deal with dividing up work between our group members as well as ensuring that everyone worked together and understood the different parts of the project.

The main goal of the project was to design and implement the basic five-stage pipeline process for the RISC-V architecture. These stages included FETCH, DECODE, EXECUTE, MEMORY ACCESS, and WRITEBACK. Following this, we were tasked with ensuring handling hazards during execution, including L1 instruction and data caches for faster memory accesses, and static branch prediction to increase the overall speed of the ISA. Finally, we were able to explore our interests by implementing the advanced features we found the most interesting to further improve our microprocessor's performance. We chose to add an 8-way set associative L2 cache to improve our memory access speeds and also improve our branch prediction accuracy by including a perceptron branch predictor along with a 4-way set associative BTB cache.

Design Description

Overview

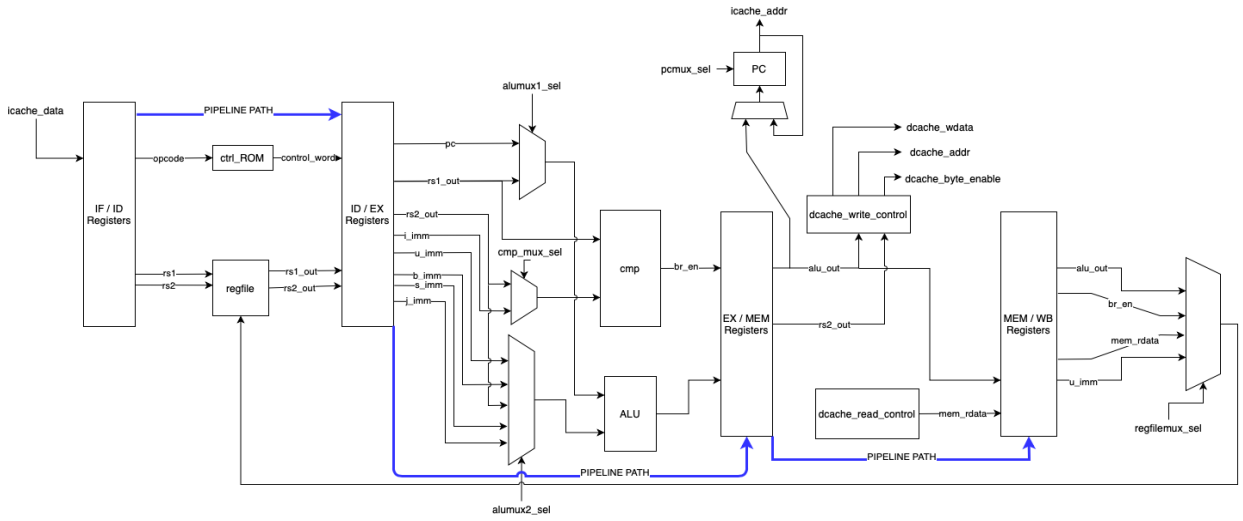
The RISC-V processor we designed included the basic five-stage pipeline process as described above. The processor also had basic hazard detection and included L1 caches to improve memory access speeds. Apart from this basic design, we had the opportunity to include advanced features of our choosing such as perceptron-based branch prediction to improve prediction accuracy for jump and branching instructions, an 8-way set associative L2 cache, and a further expansion of the L1 cache to reduce memory accesses, thus increasing overall runtime speeds.

Milestones

Checkpoint 1

Description

For our first checkpoint, we were tasked with implementing our design as we had drawn from our original design document. This included two main parts: control logic and the datapath. Our datapath design was divided into each specific stage for code legibility and ease of use when implementing our advanced features. We were also responsible for defining our control word how we saw fit for our design, and we chose to include the signals as shown on the right.

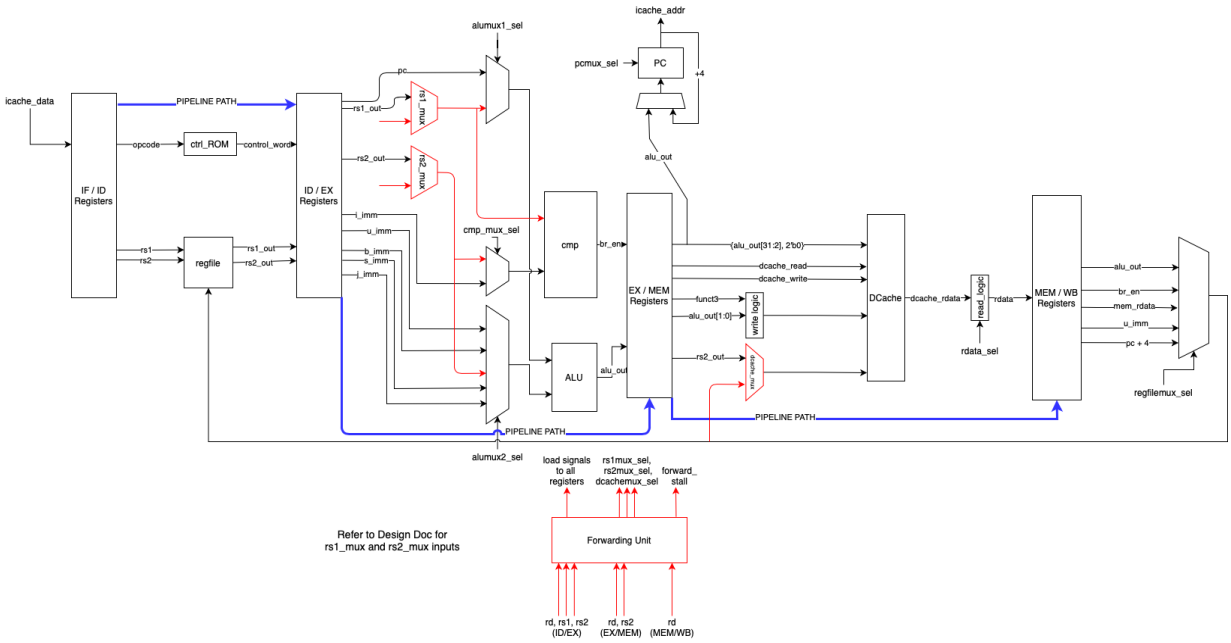


Testing

We were able to test our checkpoint one codebase mostly against the given tests. These tests checked over most basic functionality allowing us to have a good understanding of where our design lacked; however, we did have to implement our own unit tests to cover load/store,

conditional and unconditional jump testing, and register and immediate operations. Once we tested our design against all of these test cases, we were able to safely assume that our design was robust enough to advance in later checkpoints.

Checkpoint 2



Description

For our second checkpoint, we moved on to advance our original design through the addition of forwarding, hazard detection, and inclusion of our arbiter. Forwarding was implemented within all of our pipeline stages. Our design for forwarding heavily followed the information taught in lecture; however, we still had to design and check our implementations with our TA before we settled on our final design.

Testing

We began testing this code against the test cases we covered in checkpoint 1 to ensure there was no broken functionality from the new design additions. To test the new parts of our design, we ran the code against the given checkpoint 2 test code and with the given factorial.s program. This allowed us to test our forwarding logic with conflicting instructions that ran back to back. In this portion, we realized we had many problems with stalling through our design, and we were able to go back and make changes to fix flaws in our design logic. Finally, before we progressed on to adding our arbiter to the design, we also tested the code with the magic memory to ensure full functionality. To ensure that the design would work in all edge cases, we modified the given test cases to include back-to-back branching to test how robust our code would be in a complex code scenario.

Checkpoint 3

Description

For checkpoint three we were able to implement our advanced features of our choosing. These proved to be much more of a struggle compared to the original design since we had to design our components using external papers. This required much more research on our parts.

One of our advanced features we chose to implement was a perceptron branch predictor using a 4-way set associative BTB cache. Originally, we faced some trouble increasing our branch prediction accuracy over 70% for the majority of our code; however, we were able to fix this bug through various improvements later on. We chose to implement our perceptron branch predictor with a history length of 12 and a table size of 32 perceptrons by basing these values on a paper written by professors at UT Austin.

The second advanced feature that we implemented was an inclusive 8-way set associate L2 cache. We chose this improvement as it would help to increase speeds for memory accesses on top of our normal L1 cache. We chose to include this inclusive cache as we believed that the additional complexity of writing an exclusive cache would not have been worth the small improvements that it would bring to our microprocessor's performance. This L2 cache allowed for much greater storage speeds and improved hit accuracy between the Instruction Cache and the Data Cache.

Furthermore, for this checkpoint, we also hooked up the RVFI monitor to help with debugging and further find any problems with our design functionality. Since the RVFI monitor runs simultaneously with the core of our microprocessor, it can identify points at which the design strays from the ISA specification.

Finally, we also improved on one portion of our design that was not considered an advanced feature. We converted our 8-set directed mapped L1 cache into a 16-set direct-mapped L1 cache to further improve speed and accuracy on memory hits for our microprocessor.

Testing

This checkpoint presented us with many obstacles regarding testing, as we were not given any test cases to check our unique advanced features. For this reason, we had to develop a broad variety of unit tests to check edge cases when compared to our baseline implementation. After testing basic functionality with these small unit tests, we moved on to test each advanced feature separately with the tests from the previous checkpoints that were described above. Additionally, we more comprehensively tested our checkpoint 3 code using assembly unit tests from checkpoint 2. Only after testing each individual checkpoint with a plethora of test cases were we able to combine the advanced features into a single codebase and re-run all the test cases individually. Splitting up the process in this fashion ensured that we would have the most robust code design even with the advanced features we implemented.

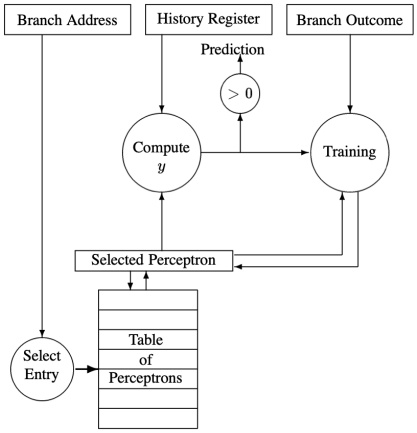
Advanced Design Options

Perceptron Branch Predictor with a 4-way Branch Target Buffer (BTB)

We chose to implement a perceptron branch predictor for the benefits of increased prediction accuracy during jumps and branches during instruction processing. Branch predictors use speculation to decrease the frequency of flushing the pipeline resulting in fewer lost clock cycles due to wrong instruction execution. Perceptron branch prediction uses a very simple neural network (*perceptron*) to process branch prediction. This predictor using neural networks uses longer history lengths to increase accuracy when compared to the more common two-bit counter methods.

Branch prediction methods are used to speed up the performance of jumps and branches in the average case. By executing instructions of a given path, stalling times can be greatly reduced. However, as the number of stages grows, misprediction penalties increase drastically. For this reason, action must be taken to reduce mispredictions as much as possible.

Design



The branch prediction design that was followed had few deviations from the research paper it was based on. Minor adjustments were made to improve speed in our case; however, the overall design was fairly similar. The design works to train perceptron accuracy correlation by increasing the value of a given weight and performing the dot product of weights with the global branch history. Every branch prediction that corresponds to a correct branching leads to an output of 1, while incorrect branches lead to an output of -1. As the weight becomes larger and larger, the neural network becomes more sure of branches taken due to a positive correlation between the weight and previously taken branches.

Our branch predictor also uses a table of perceptrons to increase performance. After every branch, the selected perceptron is trained by the algorithm defined above to increase or decrease the weight for the given perceptron. For our design, we chose a global branch history length of 12 based on the formula identified in the paper below. We also chose a table size of 32 perceptrons to minimize aliasing and power usage.

We also implemented a 4-way Branch Target Buffer following a similar implementation to our L2 cache. However, in the BTB the design allowed for reads and writes to occur simultaneously. Furthermore, this cache implementation included a pseudo-LRU replacement policy for efficient removals of unused values.

Testing

Our perceptron branch predictor was mainly tested using the competition codes to check branching accuracies. We needed to ensure that the testing code still ran accurately and completed successfully in shorter time periods. We began by testing all the competition codes

but heavily relied on competition code 1 due to it having a large number of branches. We also relied on small test benches for unit tests during development testing.

Performance Analysis

Our performance analysis for our branch predictor was through identifying our BTB hit percentages as well as our branch predictions on conditional and unconditional branches. Originally, we had poor performance on our branch predictor when compared with the power consumption; however, we were able to lessen this later by increasing the accuracy of our branch predictor. In the end, our perceptron branch prediction along with the addition of the BTB cache allowed us to increase our score by 6.15% when compared to the baseline pipeline with no branch prediction. This small percentage is mostly due to the poor performance of the branch predictor. We spent some time trying to improve its accuracy but were limited by the time constraints.

Resources Used

Our perceptron branch predictor was heavily based on a design paper by Daniel A. Jiménez and Calvin Lin from the Department of Computer Science at the University of Texas at Austin. More information can be found at: https://www.cs.utexas.edu/~lin/papers/hpca01.pdf.

8-Way Set Associative L2 Cache

We chose to include an inclusive 8-way set associative L2 cache for reducing the number of memory accesses to allow for better performance from our processor. We followed a similar implementation to the given L1 cache.

Design

For our L2 cache, we chose a data line size of 256 bits, a tag size of 24 bits, a pLRU size of 7 bits, and also assigned a valid and dirty bit. Although we debated between an inclusive and exclusive cache design, our choice of inclusive L2 cache was due to the much higher complexity of implementing an exclusive cache while having only minor performance improvements.

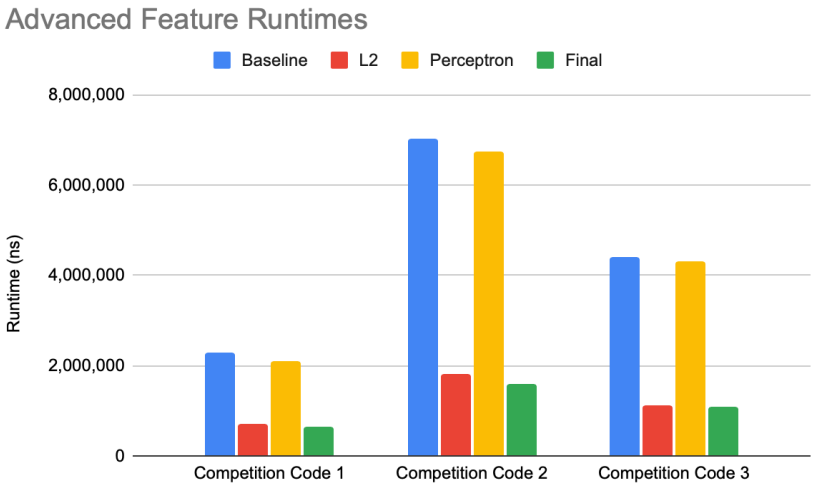
Testing

Originally, our addition of the L2 cache led to many issues dropping our f_max under the 100 threshold. This led to the score dropping drastically under what we had hoped; however, after optimizing and finding some problems without our implementation we were again able to increase our f_max. Even with these changes, our f_max was still lower than we had hoped due to long critical paths when the L2 cache was added to the perceptron code. Our testing for the L2 cache was fairly similar to the testing for the perceptron. We first tested the L2 cache on the tests from the original checkpoints, and then followed this up with comprehensive testing by using the given competition code. We also made sure to test the L2 cache through our MP3 implementation before moving it over to our MP4 code. This ensured that the L2 cache would work reasonably and we could narrow down any issues we face more easily. Unit tests were

also made for testing during the development of the cache; however, these were used minimally and mainly made for brief sanity checking at the time.

Performance Analysis

The main downside of implementing our L2 cache was the drastic increase in power usage. Although the runtime of the code greatly decreased, the power still rose from the baseline due to the additional cache added. Our performance analysis for our L2 cache was driven by benchmarks on the given competition code. Our L2 cache hit accuracy was above 93% in all the competition code benchmarks. This high accuracy drastically improved our score by considerably decreasing the number of direct accesses to memory, vastly improving the runtime of the competition code. For this reason, we were able to achieve score improvements of 3151% when compared to our baseline pipeline implementation. This may have been due to the small size of the L1 cache; had the L1 cache been larger, the score improvement that we would have seen would likely not have been as high.



Conclusion

Overall, our goal for this class was to achieve a solid understanding of the inner workings of a RISC-V microprocessor, and we believe we were able to achieve this. By designing and implementing the pipeline stages of the RISC-V architecture along with hazarding, forwarding, and advanced features of our choosing, we were able to run a microprocessor with strong performance scores on any given test code.

Although we ran into many troubles during our design process, we were able to push past the obstacles and achieve a high-performance working microprocessor. By approaching this design step by step, we were able to ensure minimal obstacles during our development process and were able to quickly identify problems in our design.

After implementing all the advanced features explained above, as well as making minor improvements to our initial design, we were able to see an impressive improvement of 3,349.35% in our score when compared to our original baseline implementation. Although we saw an increase in power consumption when running, the decrease in runtime was exponential compared to this small amount. For this reason, competition code runs had up to 4.36 times speed improvements when compared to the original baseline implementations. However, there were still many improvements we hoped to make with our branch predictor to optimize

performance and hopefully further improve our final design. However, due to time constraints, we were unable to fully implement these changes.

Overall, this project was an exceptional and unique opportunity that we were presented with, and our group was able to learn a lot about large-scale code development as well as developing as a team.