ECE 411

Spring 2021

MP4

Core™ i9000
An out-of-order RISC-V Processor

Eric Dong, Michael Kwan , Srikar Nalamalapu

ericd3, mk26, svn3

Lab TA: Abishek Venkit

Table of Contents

1. Introduction

For our MP4, we decided to implement an 32 bit out-of-order RISC-V processor based on the Tomasulo algorithm learned in lecture. The goal was to have a fully functional processor that supports the RV32i ISA. After implementing the base CPU from scratch, we also included various advanced features such as dynamic branch prediction, superscalar processing, prefetching, as well as a N-way L1 cache and a unified L2 cache.

In order to help us verify the processor and obtain metrics in order to expose bottlenecks, we developed our own software processor model that is capable of running assembly code autonomously. Not only could the software model independently run programs, we also programmed it such that it can report metrics such as the number of instructions per cycle and branch prediction accuracy. By using this feature in the software model, we were able to learn what was limiting our CPU and pick the proper advanced feature to implement that can best improve performance.

The reason we decided to implement an out-of-order processor is because we wanted to challenge ourselves and attempt to put the concepts we learned into practice. By building the processor, we can also explore different optimization techniques that may particularly benefit the out-of-order architecture.

2. Project Overview

In order to develop the out-of-order CPU, we split up our project into several checkpoints, with each checkpoint having several deliverables. This was also done to help create deadlines that ensured that we did not fall behind and allowed our TA to keep track of our progress. In total, there were four checkpoints.

We started out our project by assigning tasks to each other to complete by each of the checkpoints, but however we quickly realized how interconnected each module was to each other. It was then nearly impossible for each of us to develop offline separate from each other. A large portion of our working hours were spent together online in the later checkpoints, debugging and catching edge cases in our code that arose. Since this project was done during the COVID-19 pandemic, we used Visual Studio Code’s live sharing capability as well as Discord audio calls and screen sharing to work on ModelSim waveforms.

In the end, each of us could comfortably demonstrate capability of explaining each of the different modules, as we have together traced through the design countless times as instructions go from being initially fetched and then popped off of the reorder buffer, thus completing the lifecycle of an instruction.

3. Design description

3.1 Overview

From the initial design deadline to the final report submission almost 6 weeks later, our design has transformed drastically throughout the process. With the different design decisions and advanced design features to implement, we describe our process in creating such a CPU that can handle a complex processing algorithm with many moving parts.

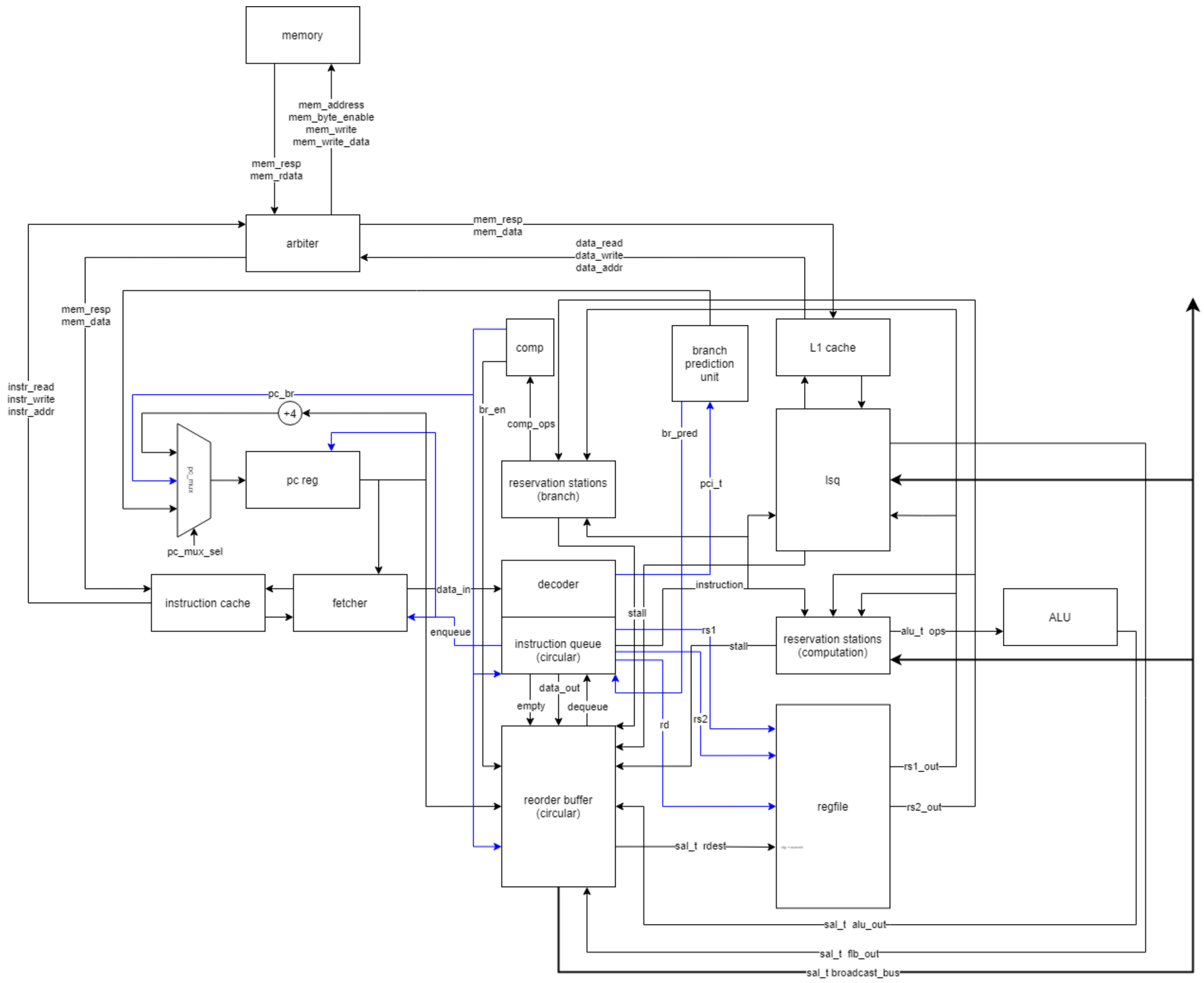

Figure 1: Our top-level design of the OOO CPU after Checkpoint 2, before advanced features.

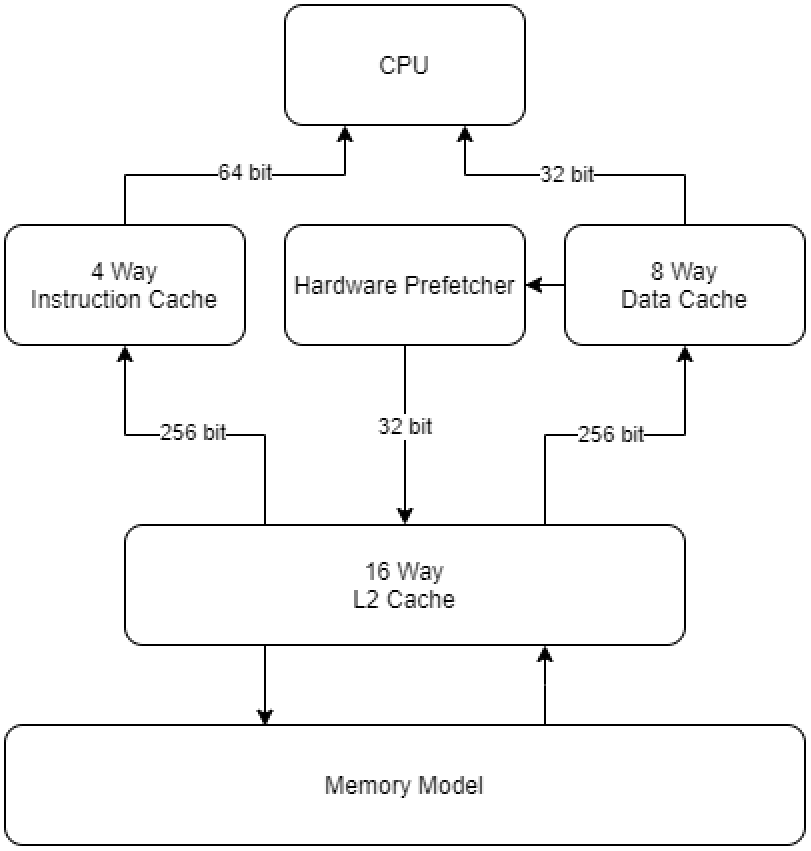

Figure 2: Our top-level design of the OOO CPU after Checkpoint 4, with advanced features.

3.2 Milestones

3.2.1 Checkpoint 1

The first checkpoint did not have any demo points attached to it, just to have a concrete design to follow while constructing the components of the CPU. In order to proceed, we need to design and map out modules such as the load store queue, reservation station, reorder buffer, instruction queue, and arbiter.

The main accomplishment of this checkpoint was to build a reliable circular queue since we would go on to use the design in several of our major modules. We modeled the circular queue from a software version and added a endequeue function which allows simultaneous enqueue and dequeue. After thoroughly testing the circular queue with a custom testbench, we then developed the reorder buffer, instruction queue, and the load store queue based on the base model circular queue.

3.2.2 Checkpoint 2

The second checkpoint required a functioning rudimentary Tomasulo algorithm, but instead replaced branch predictions with no operation (nop) instructions. The CPU should have also been able to run through one program successfully, ```mp4-cp1.s```. The term successfully required the program to reach the halt label, and have the correct register values from 0-31 correct at that time.

For this checkpoint, we integrated all the major modules that were written in the first checkpoint. During this checkpoint, we also began to make custom data structures that would eventually help us greatly. In order to test the CPU, we first started with self written code snippets that would verify basic functionality. As our CPU matured, we brought in ```mp4-cp1.s``` and made sure that worked with our CPU. During this entire process, we ran into countless unexpected bugs accounting for many bizarre edge cases.

3.2.3 Checkpoint 3

The third checkpoint needed two other programs to run correctly, ```mp4-cp2.s``` and ```mp4-cp3.s```. Additionally, branch prediction was needed to be fully implemented, as well as the side effects that came along with an incorrect branch prediction. Our team spent many hours debugging bugs at this stage of the project. Branch prediction proved to be a difficult feature to incorporate because of the many changes that needed to be made to individual modules as well as the edge cases that showed up during testing.

Testing up to this point was done just by monitoring the waveform and seeing where the execution was messing up. Using this debugging technique, we were able to debug ```mp4-cp2.s``` easily without trouble. However, ```mp4-cp3.s``` was a lot more complex and required a new debugging technique. Thus we decided to develop a software model that ran semi-autonomously alongside our CPU. This helped us catch several bugs, and was enough for us to get ```mp4-cp3.s``` working, but we knew there was still room for improvement for our software model.

3.2.4 Checkpoint 4

The fourth and final checkpoint had several deliverables, such as the successful running of all competition codes. Debugging the competition codes was by far the most difficult part of this project due to the large amount of instructions and the added complexity of our own processor. In order to better test and find debugs, we decided to upgrade our software model at this stage. We were able to modify the software model such that it was able to run code completely autonomously. The main difference is that before we were relying on the instructions in the reorder buffer to be correct, whereas now our software model has its own memory module, and is able to keep track of its own PC as well. This decision was the reason we were even able to complete our design. Without a software model or a method to pinpoint where exactly our processor was messing up, we would not have completed the processor.

After designing the initially planned out-of-order processor, additional points were able to be gained by adding several features to our CPU that could be able to decrease the number of cycles needed to run the standardized RISC-V code, or develop the scalability of our processor in the future. Certain features such as Level 2 cache, local branch prediction, and prefetching were chosen as they would be relatively easy to implement and integrate into our existing design.

3.3 Advanced design features

3.3.1 Tomasulo

- Design

The Tomasulo algorithm was the heart of our design. Our Tomasulo mainly follows the design described in the lecture slides which consists of several modules such as the instruction queue, reorder buffer, register file, reservation station, load store queue and the arbiter. The performance increase comes from the ability to execute commands in an out-of-order fashion. For the ROB, instruction queue as well as the load store queue, we decided to go with a circular queue in order to minimize the power consumption as compared to using shift registers.

The ROB is the heart of the processor as it keeps track of all the instructions that are currently being processed by the processor. When a new instruction comes in, the ROB dispatches it to the corresponding unit along with a tag, and then enqueues more instructions while waiting on the previous instruction to finish processing. Once the instruction is ready and has moved its way to the front of the queue in the ROB, it is then committed to the register file. Seeing that the ROB can only commit one instruction at a time, we decided to make the ROB able to commit multiple instructions at a time. Multiple commits happen when an instruction at the front of the ROB, which took a long time to finish (i.e. a load or a store instruction), finally finishes and the instruction behind it has long been ready. Thus we can put all the instructions that are ready onto the commit bus at this time.

Some trade-offs in the Tomasulo algorithm are the size of the ROB and support for multiple commits. The size of the ROB greatly determines the amount of instructions that the processor can handle at a time. The more instructions the processor can process, the faster the CPU can potentially run. The trade-off is that with increase in size of the ROB, the CPU logic will be a lot more complex and also more power hungry. Similarly, having support for multiple commits requires expanding the commit bus, and thus requires more logic and power to drive.

- Testing

Testing the Tomasulo processor was a painstaking process. Initially, we had to debug the CPU by just analysing the waveforms generated by ModelSim. As the programs became more complex, the debugging process became more and more impossible. Thus, we developed the software model which can run instructions in parallel. The software model is able to tell us whether we have PC mismatches, register mismatches as well as memory mismatches. This helped us tremendously since we are able to tell when and where the issue lies. From all the testing and verification that we have done, it is clear that there are many many edge cases that we haven’t thought about initially.

- Performance analysis

For this advanced feature, it was hard to compare the performance increase since we didn't have a pipelined CPU to compare with. Though, judging from the presentations, it was clear that without other optimizations, our performance was comparable and slightly faster to pipelined CPUs.

Another performance metric that is noteworthy was adjusting the size of the ROB. The performance increase was not significant for competition 1 and 2 codes, but was very impressive for competition 3 code. This is due to the amount of load and store instructions which can stall the ROB. With the increase in size, we can enqueue more instructions while waiting for the load/store instruction to finish, thus hiding the latency/stall. The performance increase for expanding the ROB was about 21% for competition 3 code, and negligible for the other two.

3.3.2 Superscalar

- Design

The superscalar feature in our processor means that we should be able to issue and commit multiple instructions simultaneously. In order to achieve this, we had to adjust several modules in our CPU, namely the instruction cache, instruction queue, ROB, reservation station and the register file.

For the instruction cache, we had to expand the data bus so that it can send 64 bits (2 instructions) at a time. We had to compensate for edge cases such as when the CPU tries to fetch the last instruction of the cache line. In that case, we should only allow one instruction to go through, and hardcoded the other instruction to be a nop (which doesn't get enqueued into the ROB).

For the ROB and the instruction queue, we needed to adjust the enqueue and dequeue processes respectively. We modified the register file such that it is able to take 2 tags from the ROB at a time. And for the reservations station, we modified it such that it can also take 2 instructions from the instruction queue. Due to time constraints, we decided not to modify the load store queue as it might create some unexpected problems.

Some trade-offs in the superscalar design was the amount of instructions to enqueue at a time. We decided to go with 2 instructions at a time for several reasons. First, we were on a time crunch, and thought that 2 instructions was enough to prove the point. Second, since one entire cache line was 8 instructions, we thought that if we enqueued more than 4 instructions, this may cause the instruction cache to miss more often, and could potentially starve the data cache. Last, we thought that enqueueing more than 2 instructions may bring more unnecessary instruction dependency bugs and edge cases that we didn't want, so for simplicity we opted for enqueueing 2 instructions instead. Since we only implemented enqueueing 2 instructions at a time, we don't have metrics for a larger number of instruction enqueue design.

- Testing

The testing process for this advanced feature was easy since all we had to do was to run competition 1-3 codes with the help of our trusty software model. The software model reported several edge cases, but we were able to figure them out quite easily.

- Performance analysis

Since we did the superscalar feature last, we only have performance statistics with all other features included. Theoretically, the superscalar feature should boost the performance of all competition codes, and as we can see, across the board, the performance increase for running each competition code was about 20%. There are some cases where the superscalar design could be bottlenecked to only enqueueing one instruction at a time. This would include the case of having consecutive branches since we designed the processor to only enqueue 1 instruction if the first instruction were a branch.

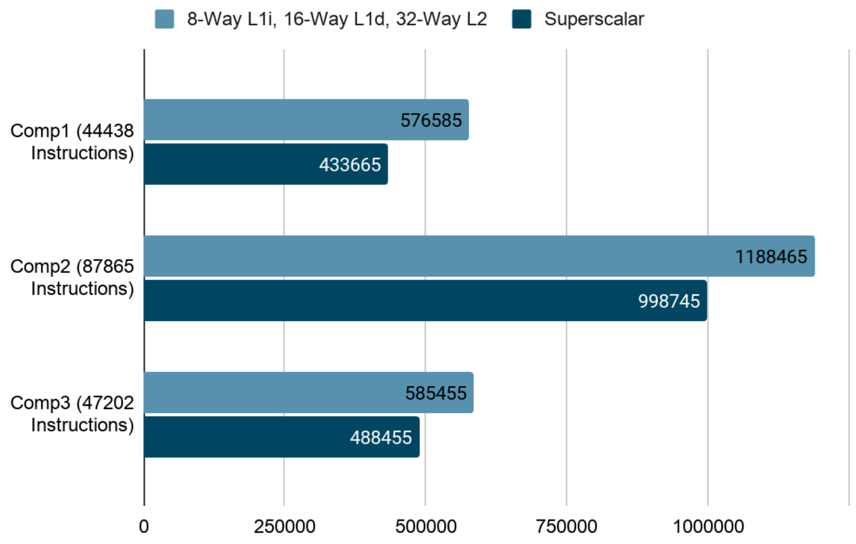

Figure 3: superscalar enabled CPU vs no superscalar CPU (optimized)

3.3.3 Local Branch Prediction

- Design

Branch prediction is a key member to the way a Tomasulo-based CPU functions. Given that the speedup compared to regular pipelines can be found in the multiple instructions being processed at one time, having a prediction system for branches to issue the correct instructions before a branch is resolved is crucial in order to maintain the efficiency of the out-of-order CPU. In our design, we were deciding between two methods of branch prediction, one containing a cached history of the previously encountered branches (N=8), using a pseudo-LRU methodology in order to evict and maintain the order of the cache structure. We also devised another method using a hashing method of the instruction address along with a large hashtable to keep track of the branch counter bits.

Due to time constraints, we decided that the second method described would be easier to implement. The method would probably have a much larger footprint in both power and logic figures, but it would be much more simpler to debug. At first, a hash table with 64 entries would be initialized to branch bits 10 (weakly taken). If a branch is issued, the branch prediction table would take a select number of bits (8th least significant bit - 3rd least significant bit) and check the status of that address in the hashtable. If weakly or strongly taken, the next PC to be issued would be PC specified in the branch instruction, else just PC + 4.

When a branch instruction is evaluated through the reservation station, the result of whether a branch is taken is broadcast to the reorder buffer, as well as the branch prediction table. A branch taken increases the counter, and a branch not taken decreases the counter.

- Testing

The testing for branch prediction was relatively simple, as there are very few moving parts and almost no asynchronous signals to deal with. We created test code that would create multiple combinations of increasing and decreasing counter movement, and then tested it live on the upcoming checkpoint code. This module was one of the easier ones to implement, and there were no changes to it after we implemented it for Checkpoint 3.

- Performance analysis

In order to analyze the efficacy of our solution, we want to know the percentage of the time where our branch predictor changed to the correct address after issuing. This is easily calculated by the number of flushes compared to the number of branches issued when executing code. The following table is the percentage of correct prediction on the competition code:

Competition Code	Branch Prediction Accuracy
Comp 1	86.29%
Comp 2	78.35%
Comp 3	78.46%

Table 1: Branch Prediction Accuracy for Competition Codes

3.3.4 Parameterized (N-Way) Cache

- Design

Initially when we began our project, we had the choice of either using a cache that we designed from MP3 or using the one provided to us. Our MP3 designs were all 2-way set associative caches with hits taking two cycles. The one provided to us was a direct-mapped cache with hits taking a single cycle. We decided to use the provided cache to get better performance with the future goal of implementing higher degrees of associativity.

When the time came, we wanted to minimize the amount of change to the cache design so that it would not get so complicated. First, assuming we were given the specification of an N-way set associative cache, we parameterized our cache by using loops within a generate construct to create N instances of the necessary arrays (data, tag, valid, dirty). We realized that a set associative cache can be treated exactly like a direct-mapped cache if we could figure out which set we were working on. On a read/write, the block we are working on can be given by three different cases prioritized in the following order: on a hit, miss but the cache is not full, or miss and the cache is full. On a hit, the block we would be working on is given by whichever cache way found the hit. If there was a miss, it would be the first invalid cache way for a given set. Finally, on a cache miss and if it were full, the least recently used would be evicted if it were dirty and the new cache line would be placed in the same cache way. Using this scheme, we could find the proper index and work on it with minimal modifications to our cache control and datapath.

Next, we needed to design our LRU. We had a choice of either implementing either a pseudo-LRU algorithm or real LRU. Although it is more costly, we chose to implement a real LRU because it was easier to design. Our cache would have N arrays that stored $\log_2 N$ bits. Each array would correspond to a cache way and would be ranked from 0 to N - 1 based on how recently they were used. On a cache hit, if a cache block with rank X was hit, then every cache block that was valid and had an LRU rank less than X would get incremented. Then the cache block with rank X would get its rank set to 0. The LRU would only be needed on a cache miss and if it were full. We would find the cache way with rank N - 1, replace it with the new cache block, increment the rank of all other cache ways in the set, and finally set the new cache block's LRU rank to 0 when the cache checks again for a hit. On a reset, the LRU arrays would be filled up with N - 1 in order for our algorithm to work.

- Testing

Testing our parametrized cache was easy. First, we made sure our cache could handle multiple cache ways before implementing an LRU. Our design could work without any LRU implementation as long as a cache set did fill up. We tested this by running our design through competition codes and monitoring the waveform. This is an acceptable testing procedure as there are no edge cases for our design. If a cache set was full, then our design would automatically evict the first cache block in a set. Once we confirmed that our cache could work with multiple cache ways when it wasn't full, we began implementing our LRU. Testing this was easier by making sure our LRU ranks were updating properly on cache hits and evictions.

- Performance analysis

Parameterizable cache proved to improve performance but provided diminishing returns after further increasing the degree of associativity. In addition, providing a higher degree of associativity would cost more so implementing such is at the discretion of the purposes.

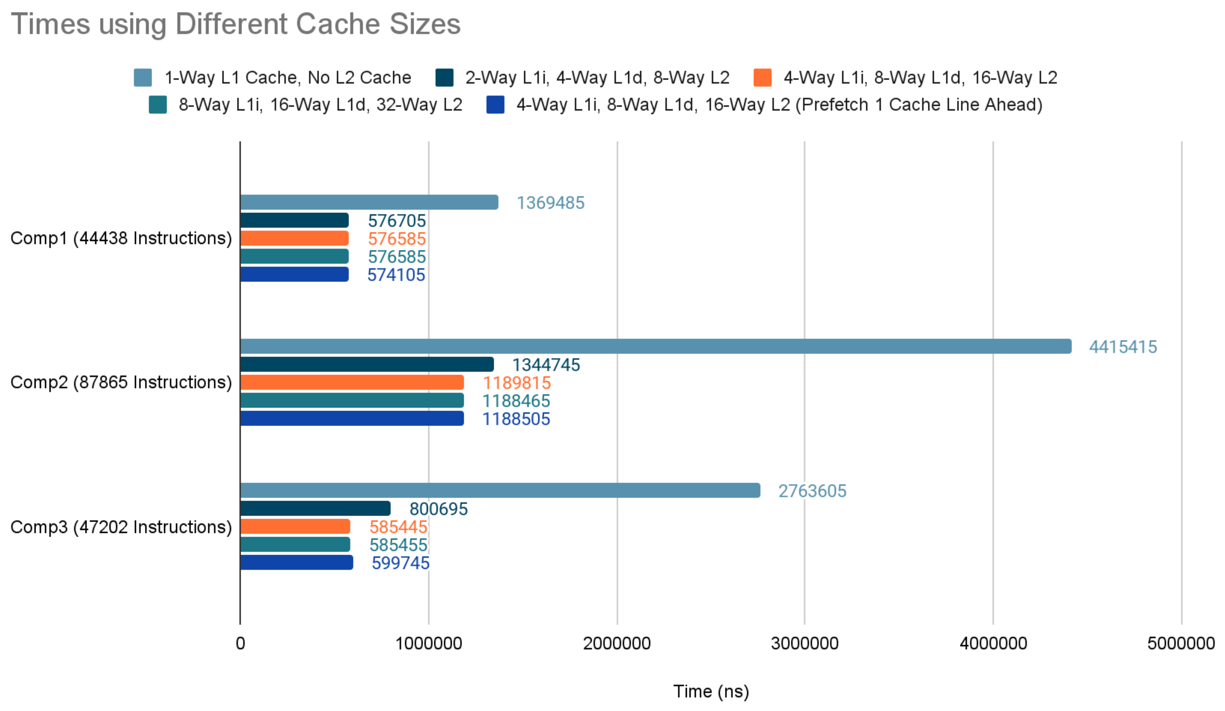

Figure 4: Times using different cache sizes (N-way cache)

3.3.5 L2 Cache

- Design

The L2 cache design was a mere modification of the L1 cache design. The only difference is that the L2 cache allows the L1 cache to read and write 256 bits at the same time, while L1 cache only allows the CPU to read and write 32 bits at a time (for d-cache, 64 for i cache in superscalar). Since we have parameterized the L1 cache, all we had to do was change those parameters in order to get L2 cache to work properly.

Some tradeoffs that we had to take into consideration was the size of the L2 cache. We tested the L2 cache at different sizes, and found that the optimal size of the L2 cache was 16 ways, since increasing the L2 cache to more than 16 ways had no effect on the performance.

- Testing

There was no need to test the L2 cache because we have already verified that the N way cache is fully functioning. All we had to make sure was to properly change the parameters so that we can link it up to the L1 cache.

- Performance analysis

L2 cache is particularly useful when the L1 cache is particularly small (i.e. 1 way or 2 ways). The performance benefits of L2 cache thus diminishes as we increase the size of the L1 cache. This is because when we increase the size of the L1 cache, the L1 hit rate will increase and thus diminish the usage of the L2 cache. This leads to the conclusion that L2 cache is useful when there's either a small L1 cache or when the L1 cache isn't able to hold enough data that the program requires (i.e. competition 3 code has a lot of data reads/writes).

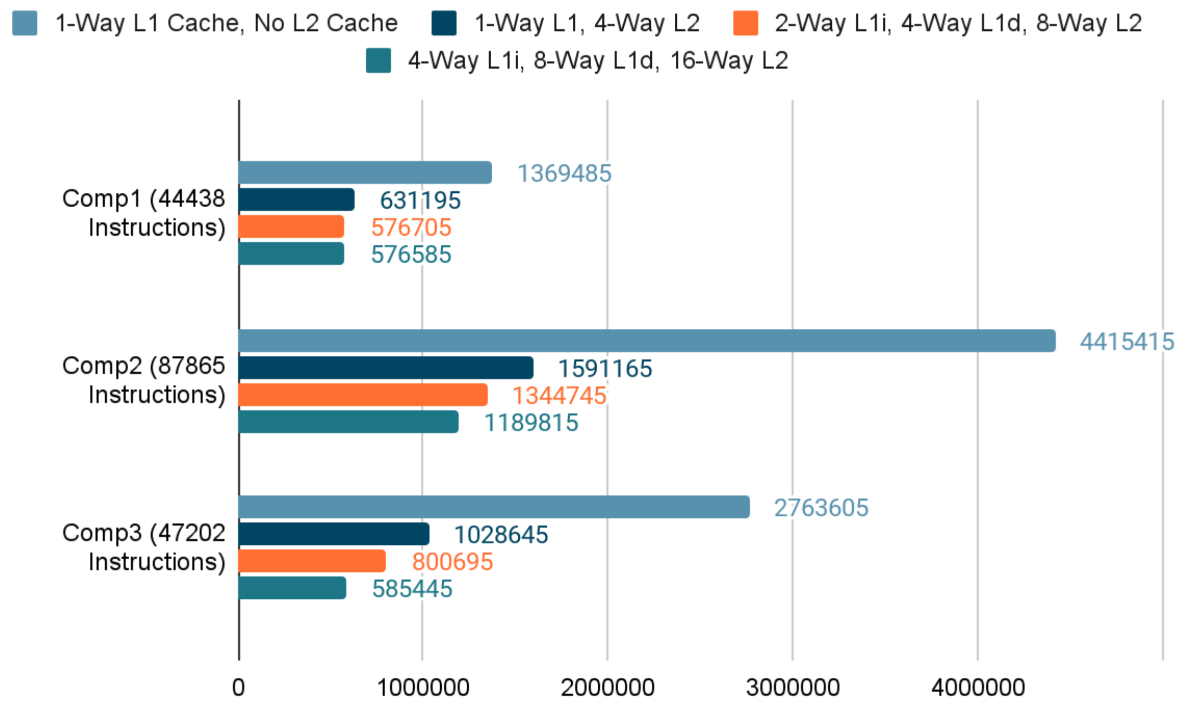

Figure 5: L2 cache performance improvement

3.3.5 Software Verification Model

- Design

One of the major problems with the out-of-order CPU can be found in the many different asynchronous signals and the endless combinations of instructions that could lead to different behaviors of each module. When execution of certain programs lead to waveforms that extend to millions of nanoseconds, it is needed for a way to tell when the execution has gone off track instead of retracing back through pages and pages of signals. A software model can execute instructions parallel to the CPU and compare the register files at any time, raising an error when there is a mismatch.

We designed the module to be semi-independent from the CPU model, loading and operating on its own memory, moving to the next instruction only when that instruction is committed to the register file from the reorder buffer. When dealing with multiple commits, the number of instructions completed per cycle is sent to the software model, and the same number of instructions are decoded and executed. A large case statement is used to deal with all types of instructions writing to a separate register file. The program counter is also checked intermittently to match each other.

We developed this verification module for the Checkpoint 3 deadline, and it was a great help in pinpointing exactly the timestamp that we could go to the waveform to find the edge case.

- Testing

Testing the software model was not easy since whenever it reported an error, we still had to check the warefrom and determine whether it was the software model's fault or our CPU's fault. Fortunately, after catching a few bugs, we haven't had any issues with the software model, and have been able to use it as a proper guide.

- Performance analysis

The software model itself doesn't improve the performance of the processor, but greatly reduces the debugging time. Before having a software model, we would have to estimate the location and time where the error occurred. But after building the model, it would tell us the exact location and time of the error, which helped us tremendously.

3.3.6 Hardware Prefetcher

- Design

We noticed that our arbiter would spend lots of idle time. We postulated that we could use that time to prefetch data into our L2 cache so that a miss at L1 cache would be a hit at L2 cache. However, our instruction cache would only miss at the beginning of the program and hit for the rest of execution. After analyzing the addresses accessed on L1D cache misses, we believed that the competition codes fetched consecutive blocks of memory. Therefore, we decided to only prefetch a parameterizable number of consecutive cache lines if the arbiter was in an idle state after an L1D cache miss.

- Testing

Once incorporating the prefetcher, we inspected if the prefetcher was making requests to the arbiter whenever data cache had a miss. We then counted the number of addresses that it would prefetch before going back into an idle state to make sure it matched up with the parameter that we specified.

- Performance analysis

With our prefetcher we noticed that with a small L2 cache our time got worse as seen in our times for competition 2 and competition 3 with the smaller caches. With smaller caches, the data that is supposed to stay in cache for future requests would get evicted data that is prefetched. With a bigger L2 cache, we wouldn't have to evict L2 cache data as much, therefore causing more L2 cache hits on L1 cache hits. We see this in the slight improvement in competition 3 as this program works more with consecutive data. The difference in times for competition 1 and 2 was negligible, therefore including a prefetcher with a large L2 cache is recommended.

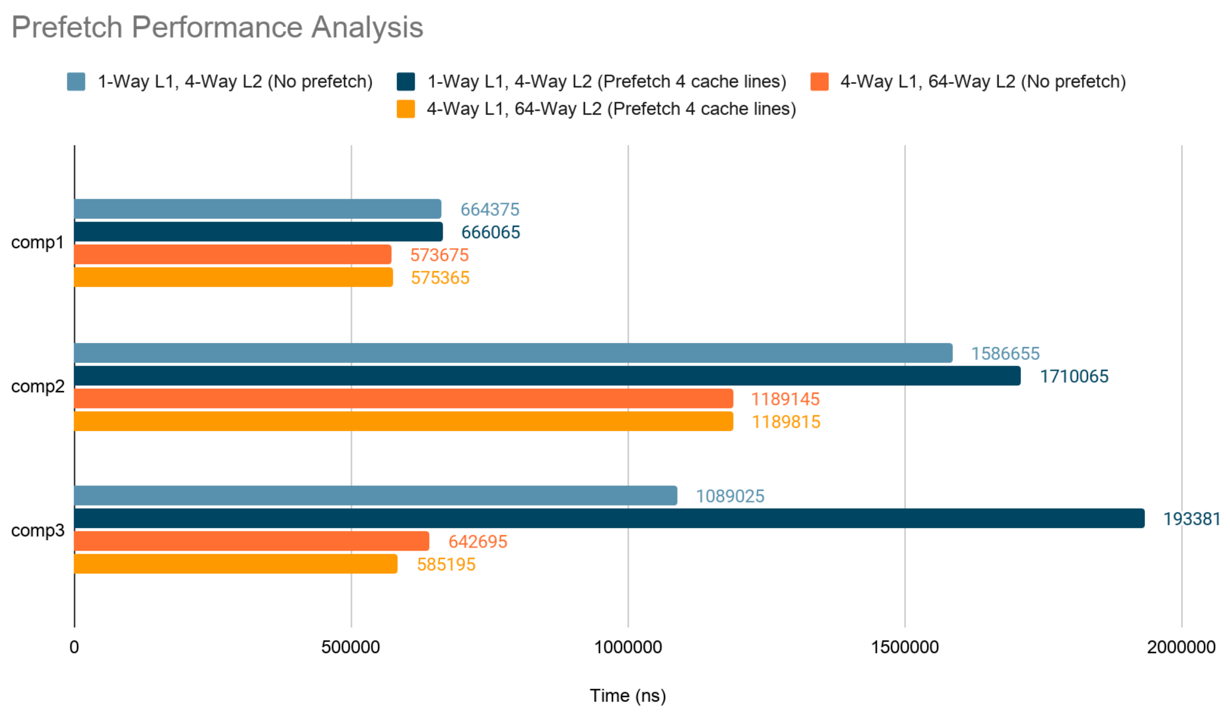

Figure 6: Performance Analysis of our Prefetcher with Different Sized Caches

4. Additional observations

While our design's times were substantially better than the baseline, there is a matching cost to it. Our processor's fmax was unable to reach 100 MHz and was only able to reach 37.5 MHz with an average power rating of 1946.72 mW. We expected the significantly worse fmax and power ratings because of the complexity of our design. Our design required a lot of combinational logic that would add a lot of delay to our clock speed. Furthermore, in order to simplify our design's code, we often made use of hefty operations such as modulo and multiplication. If we had kept this in mind during our design procedure, we could have implemented a design with a higher fmax. The high power rating is expected simply due to the large amount of combinational logic in our design.

In addition, compiling our design in Quartus was a pain-staking process. We were only able to compile our base processor without any advanced features because it took about 5 hours to complete compilation. Compilation with advanced features would either crash Quartus or take at least 14 hours at which point it still would not have completed. If we were able to compile our design quicker or more often, we could have had more attempts to improve our fmax and power.

5. Conclusion

After seven weeks of effort, we accomplished our goal of implementing an out-of-order processor based on the Tomasulo algorithm. In addition, we were able to improve our base performance significantly by adding extra features including a parameterizable cache, L2 cache, superscalar, and hardware prefetching. This was also only possible with the software model that we designed specifically for our out-of-order processor. With all of our advanced features and running the competition codes, our processor recorded times of 433,665 ns, 998,745 ns, and 488,455 ns respectively. This beat the baseline times of a normal pipeline processor by 39.8%, 77.9%, and 86.6% respectively. After designing our own out-of-order processor, we recognize the great improvement that it can provide over a normal pipelined processor.

Throughout this course we have been challenged to design processors and think of ways to improve our design. As a result of adding the extra challenge of designing an out-of-order processor, we have acquired many new skills. Furthermore, learning that current architecture industry giants such as Intel and Apple incorporate immensely more complicated out-of-order designs in their product, we have a newfound admiration for the subject of computer architecture and the engineers that devoted their entire lives into the field. All in all, we will carry on the things we have learned from this class and project into future endeavours.