ECE 411 Final Report

Team 3

Rauhul Varma (rvarma2), Kyung Jae Lee (klee152), W Kim (wkim53)

Introduction

The goal of this project was to create a traditional five-stage pipelined processor with split L1 caches and a unified L2 cache. Additionally, we had the stretch goals of adding more advanced features such as dynamic branch prediction, etc. In deciding these advanced features, we took into consideration the impact each would have on our performance, alongside the additional complexity introduced by the feature.

Our processor ended up supporting the entire LC3b ISA excluding the RTI instruction; we first added basic instructions like ADD, AND, NOT, as well as simple memory instructions like LDR and STR, then went on to add the support for the rest of the ISA. The processor features split L1 caches with a unified 8-way associative L2 cache with an eviction write buffer paired with a pseudo LRU. Additionally, the processor supports full data hazard detection/forwarding, ensuring the fastest execution possible with minimal stalling.

The processor also features performance counters to track many metrics such as cache hits/misses, stalls, and both incorrect and correct branch predictions. Moreover, these performance counters are accessible via memory-mapped I/O. We also chose to implement two advanced features commonly seen in industry processors.

We noticed our processor was wasting a large percentage of cycles flushing after incorrect predictions. As a result we implemented a dynamic branch predictor with a branch history table coupled with per-address pattern history tables; this dynamic predictor allowed us to increase our prediction accuracy to just below 90% and consequentially greatly improved our performance. To further improve our processor's performance we added counters for the number evictions from each cache and realized our data cache was evicting often. As a result we implemented an N-way set associative cache that allowed us to easily experiment with different cache sizes; we eventually settled on an 8-way L2 cache. These two features were aimed to address the biggest slowdowns we saw throughout the development process.

Project overview

Due to the scope of the project and large amount of work required to make a successful processor, very early on we realized that having a good method of balancing the workload would be required. We used agile methodology since gitlab extends itself well to this development process. We leveraged the built in milestones and issues to create all the tasks required for each checkpoint and followed task's progress using the built-in scrum boards.

The plan was to create a bunch of issues or tasks at the beginning of each checkpoint and each teammate would opt-in to the ones they felt best able to accomplish. This approach let us split up the work easily without directly needing to assign tasks to one-another.

However, this approach broke down during the first checkpoint as most of the burden fell on Rauhul to complete. Unfortunately this continued throughout the project; while Rauhul created a bunch of tasks to complete, he ended up completing them himself. Given the opportunity to redo the project, our work split would have been greatly improved if all team members had worked on the tasks for each milestone.

Design Description

Overview

Over the next couple subsections, we cover how our processor evolved over the semester, highlighting the major milestones. We discuss the design decisions, their testing strategies, and the advanced design features.

Milestones

Checkpoint 1

During this checkpoint, we completed all the basic instructions in the LC3b ISA, ADD, AND, NOT, LDR, STR, and BR. Additionally, we added data paths through the pipeline support for most of the rest of the instructions in the specification. We did not handle any control hazards or data hazards in this checkpoint, as specified in the checkpoint documentation.

We implemented the pipeline by creating a set of `stage` and `barrier` modules which plug directly into one another forming the pipeline. These modules were each used once in our larger CPU module. This architecture of modules allowed us to functionalize each of the pipeline components and greatly simplified the debugging process.

See Appendix A for the initial pipeline diagram.

Testing

We created test code for each of the basic instructions that rigorously exercised each instruction. We used these test cases to debug potential pipeline implementation issues and as regression tests for future features.

After ensuring that the basic instructions worked properly, we used the checkpoint 1 code as additional test code for the LDR, STR, and BR instructions. We wrote down what every relevant signal was before stepping through the program execution in modelsim and checked against our hand simulation. This allowed us to very quickly identify errors and fix them.

Checkpoint 2

By this checkpoint, we had implemented full support for all LC3b ISA instructions, including the more challenging instructions such as the indirect memory accesses. We implemented the STI and LDI instructions by stalling the memory stage individually allowing for two memory accesses per indirect memory instruction. We also added support for PC modifying instructions such as

JMP, JSR/JSRR, etc. For these instructions we updated the write back and fetch stages to allow for the PC to be updated from multiple sources.
In this checkpoint we also added split L1 caches to our processor (i cache and d cache). These caches fed into a basic cache arbiter that allowed both L1 caches to have access to physical memory.

See Appendix B for the initial cache design diagrams.

Testing

Just like checkpoint 1, created test code for each of the new instructions. We used all the previously written tests as regression tests when adding support for instructions. Once all the instructions passed our tests we then ran through the mp3cp2b test code; we annotated every register value change and insured our processor performed each one of these register changes as expected. After adding the caches we tested each instruction again, we re-ran through the mp3cp2b test code ensuring all tests still passed.

Checkpoint 3

During checkpoint 3, we updated the pipeline to detect and handle control, data, and structural hazards; we introduced a forwarding controller that detected hazards between instructions in the execute, memory, and write back stages to make this possible.

We also refactored the cache design so the number of lines and associativity could be chosen during module instantiation. As a result we could easy configure all the caches in the CPU and expanded the d cache to contain 8 cache lines with 4-way set associativity. Additionally, we leveraged this parameterized cache to easy add a unified L2 cache with 8 cache lines with 8-way set associativity.

Moreover, during this checkpoint we refactored the way stalls and resets are propagated through the pipeline. We added a stall arbiter module to take `pipeline control requests` from various pipeline sub components that can modify the pipeline's execution; This stall arbiter allowed us to propagate these control request based on preassigned priority and exclusivity requirements. This structure allowed us to easily add modules that can stall or reset various pipeline components without requiring major re-architecting of the pipeline.

Testing

During this checkpoint, we used all the tests from previous checkpoints as regression tests. We also identified the locations in the checkpoint 3 code, that would cause issues in checkpoint 2 CPU and used them as small unit tests for our checkpoint 3 implementation.

Checkpoint 4

During checkpoint 4, we added 3 features to our pipeline: a set of software-visible performance counters, an eviction write buffer and a static not-taken branch predictor.

We added the performance counters to our memory stage, and introduced a small module that intercepted reads and writes to memory addresses assigned to each counter. This module re-routed memory accesses these address to the performance counters making them accessible to the CPU. The CPU could load the values in the counters and reset them via a store.

We added an eviction write buffer to our L2 cache, this addition allowed us to read new cache data before the write back occurs, effectively allowing the write back to complete asynchronously with the execution of the main pipeline. Additionally, we added miss and hit signals coming from all the caches to be consumed by the performance counters.

Lastly we refactored the branch controller to take a prediction from a branch predictor module and added a queue of predictions to handle sequential correct and incorrect predictions. Like the caches we also added signals to report correct and incorrect predictions again to be consumed by the performance counters.

Testing

For this checkpoint we created a new test case that exercised each of the new code paths and use it to debug implementation issues. The test case included code to check our branch flushing and static branch prediction as well; this was done by putting placing instructions that should be skipped when the branch is eventual resolved. This allowed us to see that the instructions that should be skipped made no irreparable modifications and were flushed correctly. We also re-ran all previously created tests and checkpoint code to ensure no previous features broke.

Advanced design options

As mentioned previously when deciding advanced features to implement, we took into account the impact each feature would have on our performance and the additional complexity they would introduce. We used the performance counters implemented in checkpoint 4 to determine which CPU features would benefit the most from a more sophisticated implementation. We settled on improving the associativity of our caches, due to the large number of memory accesses in some of the competition code, which caused a multitude of conflict misses. Following the same decision-making process, we also determined our branch predictor was another major component with large room for improvement.

Configurable N-Way Set Associative Cache

Typically, increasing the size of a cache by increasing associativity decreases the frequency of conflict misses decrease. However, because implementing a true least recently used eviction policy requires a large amount of additional circuitry, high associativity caches must make due with other eviction policies that emulate the true LRU policy. Many policies can yield highly accurate emulation, however none are perfect and suffer from an increased frequency of conflict misses compared to the true LRU policy. Moreover, since the cache is still not fully associative, an equal sized set associative cache again suffers increased frequency conflict misses. Due to the cumbersome nature of implementing true LRU eviction policy we decided to support our increased associativity cache with a pseudo-LRU policy.

After determining we would use a pseudo-LRU policy, we realized the actual implementation was fairly trivial for a given N-way associativity, but we were not sure what N would actually yield the best results; we also did not know the best number of sets for each cache. These two factor combined drove us refactor our caches to support any power of two (2^n) associativity and (2^m) number of sets.

In order to make this possible we first had to refactor many of the supporting modules used throughout our pipeline design. We updated all muxes and demuxes in our CPU to leverage a new parameterized design that allowed the number of inputs and size of the inputs to be determined during instantiation. The `cache way` module was subsequently updated to support the `number of sets` parameter and the `cache` module was updated to instantiate the `associativity` quantity of these `cache way` modules. The main difficulty came from creating a parameterized pseudo-LRU module which required extensive use of `generate` blocks.

See Appendix C for partial whiteboarding of the parameterized pseudo-LRU design.

Once the design was fully complete, we tested each of the sub modules ensuring they worked correctly individually. We first tested the LRU module by manually expanding the generate blocks for various values of the `associativity` parameter; we leveraged Sublime Text to make expanding these blocks easier, using the multi-cursor features alongside the Text Pastry package. We then wrote targeted test cases for the cache to ensure all signals had been refactored correctly. These test cases allowed us to quickly identify issues in the parameterization such as: incorrectly adjusting the number of tag bits in response the to `number of sets` parameter. Finally, once the cache components passed these tests, we re-ran our full set of previously written tests to ensure the pipeline did not break as a result of the changes.

Since our new cache was fully configurable we were able to very easily try out different sizes and structures for each of the caches in the CPU. We found that across the 3 competition test cases we could reduce the L1 d cache miss percentage from 14% to 6% and the unified L2

cache from 42% to 26%. Combined these improvements decreased (improved) our competition score by 13%.

Configurable Branch Predictor

Conditional branches are a very large source of stalls in CPUs without a high accuracy predictor and serve as major performance obstacles. A wide variety of branch prediction schemes have been proposed with a very large range of accuracy and consistency. These predictors attempt to dynamically determine during run time if a branch should be taken and can be extremely large. While profiling our CPU's static predict taken and static predict not-taken schemes, we found that across the 3 competition codes our predictors were (a) not very accurate and (b) highly inconsistent. However, similar to the cache designs mentioned above, we did not know what schemes would yield the best results.

With this in mind we set-out to create an easily configurable branch predictor that allowed us to quickly test many configurations. In order to make this possible we first implemented a `branch waterfall queue` module to record all encountered branches and the misprediction PC in case the branch prediction was incorrect. We additionally implemented a branch controller that took an input from another branch predictor module to make switching out predictors extremely easy.

We first started with a simple 2-bit saturating predictor, this predictor improved our accuracy from roughly 52% for our static not taken predictor to 64%. (Note all percentages are based on weighted averages across all 3 competition codes). However, we did not see any improvement in variance prediction accuracy across codes. Following the simple 2-bit predictor we implemented a parameterized global N bit branch history register with a global 2^n pattern history table containing 2-bit saturating predictors. We further extended the predictor adding support for M per address pattern history tables and K branch history registers forming a per address branch history table.

These parameters together allowed us to try a very large set of branch predictor configurations; ultimately, we found that using a global 3-bit BHR combined with a per address PHT using PC bits [5:4] predicted with 70% accuracy and an extremely low test to test variance of 0.089% compared to ~5% for the static predictor. Additionally, we found that we could achieve 90% accuracy with 2.47% variance using a global 6-bit BHR and a global PHT. We believe these results are due to the largest competition code doing matrix-matrix multiplication, which is not a typical workload for general applications; we would expect using a pa-BHT and pa-PHT to yield the best results.

For our competition code we used the 6-bit predictor mentioned above which yielded a 21% improvement in our competition score.

Conclusion

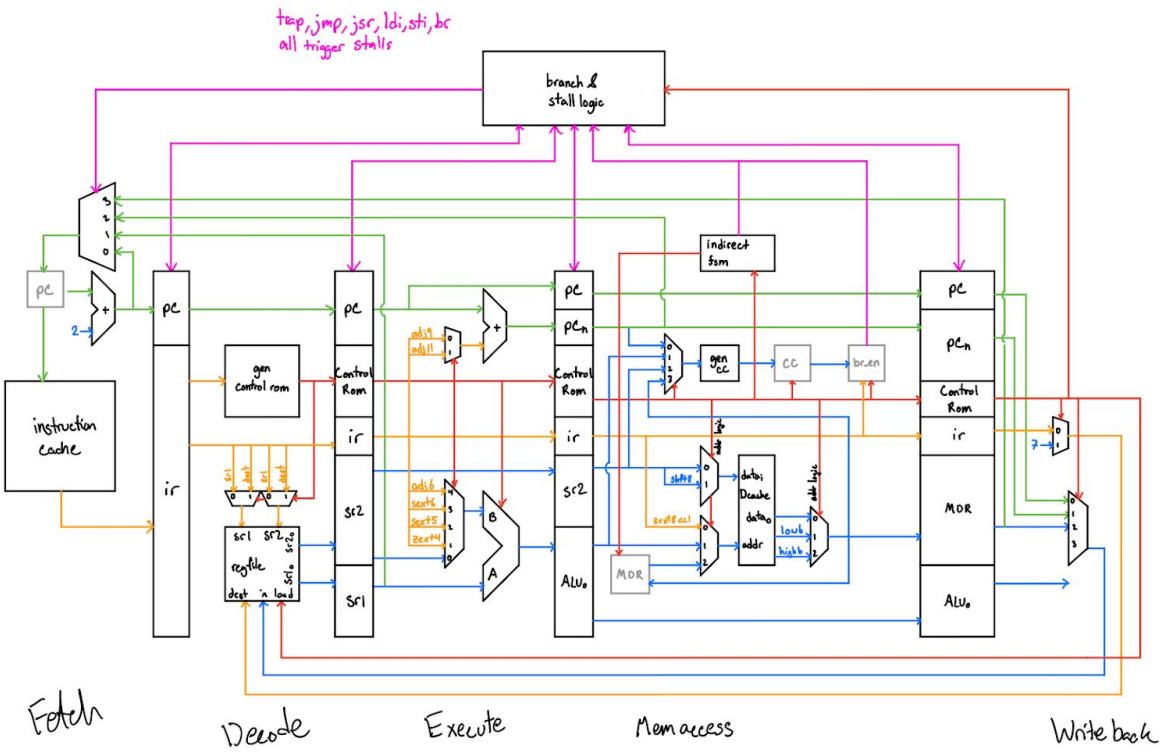
We sought to design a 5 stage pipelined processor that met all the requirements of the project as well as our own goals and successfully implemented the pipeline along with data, control, and structure hazard handling. We implemented additional features beyond the requirements and designed in such a way as to allow for future features to be implemented easily. The project very different from any other project we had worked on; the project required much more time than any other and required a completely different design and verification paradigm.

Unfortunately due to the limited time in the project, we did not have enough time to implement more advanced design features. We would have liked to add fully asynchronous write backs to our caches to allow for hits to be serviced during the write back. Additionally, we wanted to experiment with block based writes into the d-cache to optimize performance of highly sequential workloads. Along similar lines we also wanted to add a WIDTH parameter to our configurable cache to allow for more experimentation. However, given the limited scope we are content with the quality and quantity of advanced features were able to implement.

After finishing implementing features mentioned above, our processor had an fmax of 72.3 MHz, which was much lower than what we had wanted to reach. Given more time, we would definitely work on reducing the length of the combinational paths that caused the fmax decrease. However, while our fmax was lower than desired we were able to correctly predict branches with just below 90% accuracy and had a multilevel cache hierarchy that very effectively reduced main memory accesses. Combined these features allowed us to drastically reduce the number of required cycles to complete most programs.

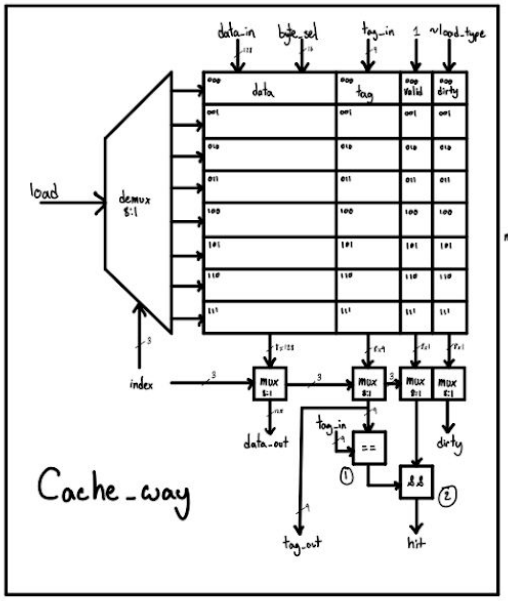
At the beginning of the project we did not expect the project be as complex as it eventually became. However, as the project concluded there is a newfound respect for computer architecture and the work that does into the design and verification of processors. We started the semester skilled at programming and ended with the ability to design in an entirely new paradigm.

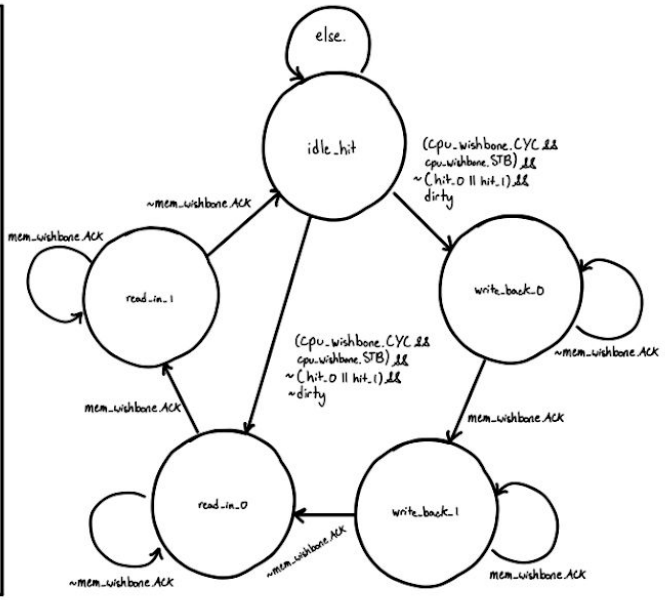
Appendix A: Initial Pipeline Diagram

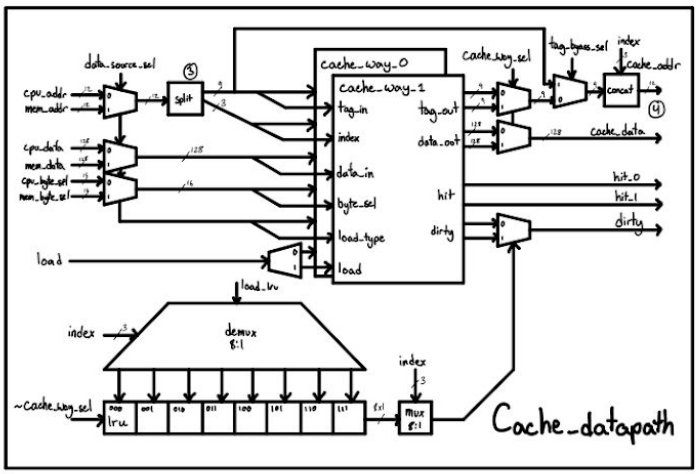


Appendix B: Initial L1 Cache Implementation

(Pre-Parameterization)







Appendix C: Parameterized Pseudo-LRU Design In-Progress