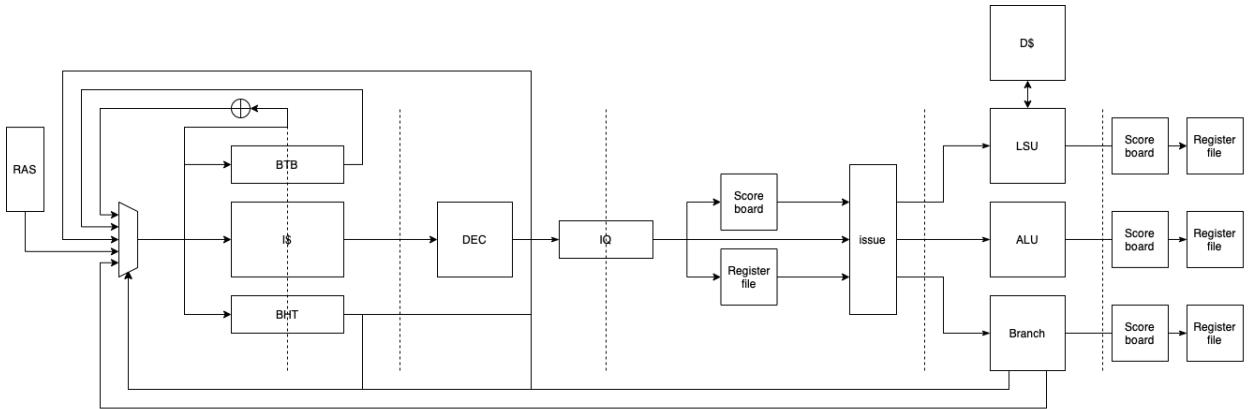


ECE 411 FINAL PROJECT REPORT



1. Introduction. Computer architecture is a dominant subfield of computer engineering. This project served as a learning experience for CPU design and computer architecture. One of the most up-and-coming architecture's today is RISC-V, so we set our goal to design an efficient, pipelined RISC-V CPU. RISC-V is an open standard instruction set architecture (ISA) based on reduced instruction set computer principles. RISC-V is also distributed under open-source licenses. This report will explore the development process and explain the project timeline, process, design choices, results, and challenges.

2. Project Overview. For this project we focused on making a CPU that was optimized for performance and power consumption. We wanted to minimize our power consumption and minimize the time that it takes to process instructions. Our other goal for this CPU was to participate in our class's competition for best performing CPU which was based on metrics relating to power and timing. The competition led to a lot of our goals and we picked advanced design features for our CPU based on what would make the competition codes run faster. The first thing we did was look at all the instructions that each competition code had and then plan out which design features would optimize them. In terms of organization for our project, each week the team would meet together to discuss. In these meetings we looked over what we had to do for the week and assigned work based on people's interests. Then at the end of the week we would delegate someone to write the weekly report and mark all our progress down. At these meetings we also frequently came up with design diagrams and decisions together. Overall this worked pretty well for us and led to us finishing a working CPU with a decent amount of advanced design additions.

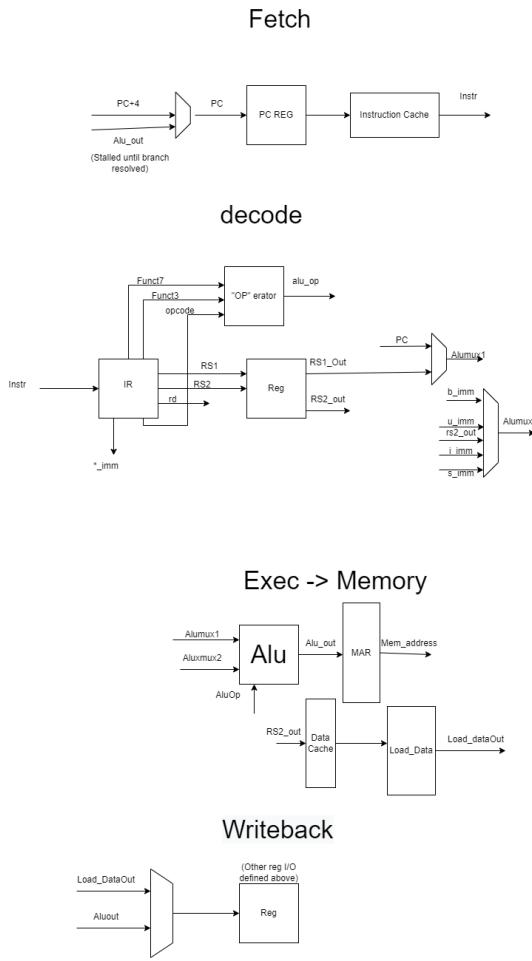
3. Design Description.

(a) **Overview.** Overall we made an in-order pipelined CPU that runs the RISC-V ISA. Some notable features of our design compared to a basic in-order pipeline CPU is that we have a scoreboard and an instruction queue. We put these in place since originally we wanted to make a superscalar CPU but ran out of time before we could finish it. Our design also incorporates pipelined L1 Caches, GShare Branch predictor, Short Forward

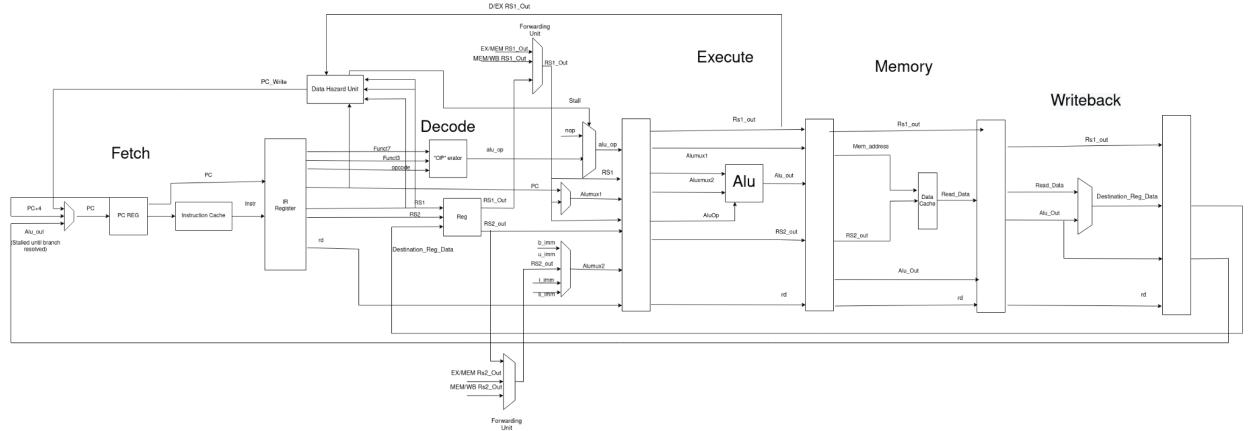
Optimization, and the ZBB extension.

(b) **Milestones.**

i. Checkpoint 1. For this checkpoint we began by making a design based on our datapath diagram that we created in the original Design Checkpoint. We went off by splitting the work based off of the pipeline stages. Splitting it into the decode stage, the alu/branch/load-store units, regfile and setting up magic memory. We also worked on the datapath diagram for checkpoint 2.



ii. Checkpoint 2. For this checkpoint we began to think about our advanced design features. We decided to go with superscalar, the zbb extension and a multiplier but also had backup ideas in mind. We also finished the arbiter, forwarding and data hazard detection units in this stage. We also wrote the proposal for our advanced design features as well as updated our datapath to resemble what we currently have in place.



iii. **Checkpoint 3.** For this checkpoint we mostly worked on the advanced design features and were getting ready to formalize our design for checkpoint 4. As well as getting into the swing of verifying things. This is where we actually hooked up RVFI in order to verify our CPU. We also began making advanced design features such as the zbb extension, gshare prefetcher, pipelined caches, multiplier, superscalar, and short forward optimization. We unfortunately did not finish the superscalar part by the end of this checkpoint or the multiplier. We did incorporate some elements of the superscalar design such as the queue as well as the scoreboard. The instruction queue was helpful as it allowed us to hide latency between the front and backend of the CPU.

iv. **Checkpoint 4.** For this checkpoint we focused on making our presentation as well as finishing up our written report. Beyond that we were also working on integrating all of our components together. We also were verifying that all the parts of our CPU were working in conjunction. We found a few bugs initially with the cache as well as jump logic that did not appear in the earlier checkpoint codes but did in the competition codes. We mostly work equally as a group on all parts of this checkpoint. (Datapath for this and checkpoint 3 are at the top of the report)

(c) **Advanced Design Options.**

i. Pipelined L1 Data and Instruction Cache.

A. **Design.** Cache design choices are multi-dimensional. First and foremost, the choice of a custom cache vs the given cache was easy to make. Max read and write throughput vs the given cache would be 2x and 1.5x respectively. We additionally decided to go with a pipelined cache to take advantage of the Fmax and capacity benefits of BRAM.

We also had to decide between direct mapping, set-associativity, or full associativity. We opted for direct mapping for both of our caches. For the instruction cache, we profiled the code and found that we could fit all the competition programs fully into cache, so there was no need to introduce

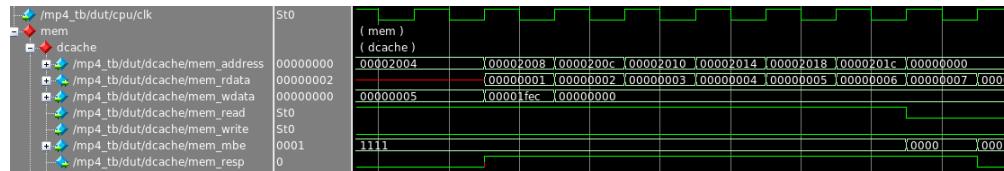
complexity with more complex mapping. For the data cache, we left it as a future option to introduce set-associativity, but didn't set it as a high priority.

Our instruction and data caches ended up very similar in design, with the exception of not implementing write back for the datacache. We did end up going with 2KiB for the I1\$ and 1KiB for the D1\$. Configuring this was simple since both caches were parameterized. These capacities turned out to be the best mix of Fmax and IPC for us.

State was not allowed, so implementing the caches was a bit of a challenge. We designed the data cache incrementally and chronologically in terms of what behavior should happen. The biggest challenge was figuring out how to set signals without being a function of state, but eventually combinational logic was enough to do the job. The instruction cache was a simplified variant of the data cache, since no CPU writes are needed.

B. Testing. We tested behavior by checking test code resulting registers against the baseline of the given cache.

C. Performance Analysis. We anticipated that the pipelined caches would help with consecutive read/writes and increasing Fmax. Upon profiling, we indeed found that consecutive read hits went from 1 per 2 cycles to 1 per cycle and consecutive write hits went from 1 per 3 cycles to 1 per 2 cycles. We have shown a demonstration of consecutive read hits below. We also measured Fmax to be increased from 143MHz to 185MHz, a 29% speedup.



iii. GShare Branch Predictor.

A. Design. GShare is a branch prediction technique which aims to make use of both PC and global branch history data in predicting the next branch. Ideally, it should be able to predict branches which are correlated or uncorrelated. GShare branch predictors are fairly simple structures.

In terms of hardware, GShare BPs consist primarily of a large table of 2 bit counters and a hashing mechanism. The hashing mechanism combines low order bits of the PC and global branch history into a single bit vector with which to index the counter table. Ideally, the hashing mechanism should ensure there are minimal conflicts between different PC and BH combinations. Typically, the hashing mechanism is a simple XOR.

Our GShare implementation consisted of a 1024 entry counter table. This was

indexed with 10 bits of branch history XORed with the last 10 bits of PC. Accesses were 1 cycle pipelined and our table was stored on BRAM.

B. Testing. We tested correctness while integrated with the full design and ensuring that misprediction rates were reduced.

C. Performance Analysis. We tested the unit's performance by creating performance counters to track correct and incorrect predictions. These counts were further divided into branch taken and not taken predictions. Additionally we ensured that runtime had reduced.

iv. Short Forward Optimization.

A. Design. Short forward optimization is an optimization technique used in architectures which lack conditional move instructions. It works by detecting “short forward branches” or branch instructions with small positive offsets.

The instructions which would be skipped if the branch were to resolve taken are instead tagged conditional. These conditional instructions are only allowed to write back if the branch under which they are shadowed resolves not taken.

This technique is useful for preventing pipeline flushes due to unpredictable instructions. Ideally, a design should only trigger SFO for branches which are difficult to predict. It should only shadow instructions with short latencies which do not have any side effects.

Our SFO implementation shadowed branches with offsets of 8. This enabled our design to avoid predicting branches which only skipped a single instruction. These instructions were common in comp1.

B. Testing. We tested our shadowing implementation by integrating it into our design. Our design was hooked into RVFI and thus able to detect incorrect PCs.

C. Performance Analysis. Unfortunately, short forward optimization made it difficult to correctly maintain the global branch history. Because of this, our conventional branch predictor never “learned” the patterns for short forward branches.

Our SFO was configured to only kick in for low confidence branches, but since our BP never learned our short forward branches all of them remained low confidence.

Therefore, SFO would be applied to all short forward branches and not just the unpredictable ones. In the end, our implementation only increased comp1 scores by 5% and other test codes by even less. We ended up removing SFO from our final core since the improvements were lower than the reduction in Fmax.

SFO would probably be a more viable technique if we had been able to correctly

maintain the global branch history for SFOed branches.

v. Multiplier.

A. Design. For the multiplier we did a lot of research on what type of multiplier would be the easiest to implement while retaining significant performance upgrades from a standard multiplier. We took note of three prominent designs that we came across in research papers. Array multiplier, Dadda multiplier, and Wallace multiplier. We ended up going with an Array multiplier since it seemed the easiest to implement. An array multiplier works by essentially calculating multiple partial products at a time and then shifting over the partial products by the correct powers of 10 and adding the partial products together. We started on an array design but did not finish by the time of the competition. Therefore foregoing adding one to our overall design.

vi. Zbb Extension (Bit Manipulation).

A. Design. Zbb is the base set of the RISC-V Bit Manipulation extension. This extension includes many common bitwise manipulation extensions such as logical negate (andn, orn, xnor), single instruction min/max instructions and most importantly for us, a byte swap instruction. This instruction (rev8) reverses the endianness of a word.

B. Testing. Zbb was tested integrated in the CPU (with RVFI) running testcodes which made use of the instructions.

C. Performance Analysis. Byteswap is implemented in software in comp3, so rev8 allows us to skip a significant amount of work for that test code. When recompiled with Zbb, comp3 also makes use of min and max instructions. The Zbb extension allowed us to increase our performance in comp3 by 60%. We did not recompile any other testcodes with the extension.

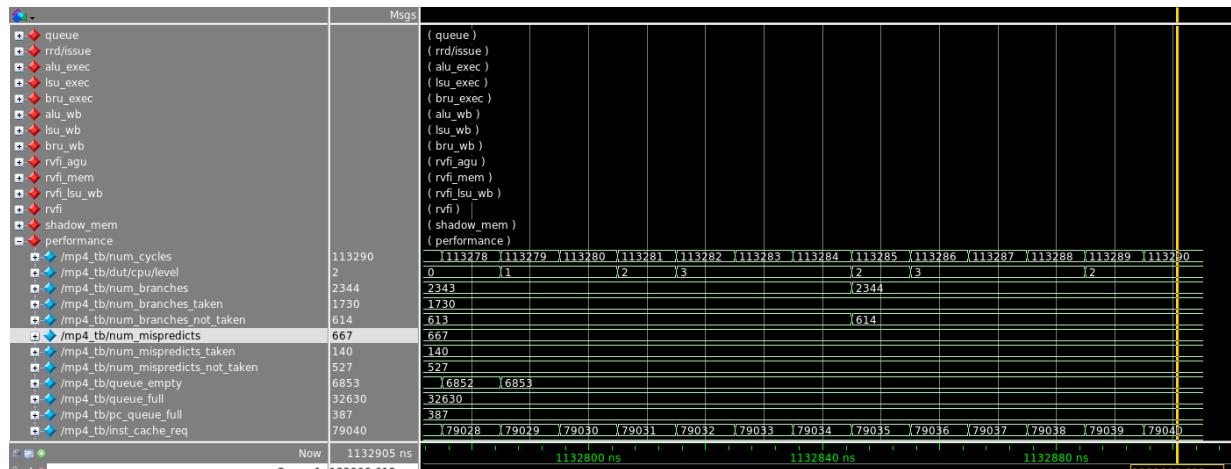
vii. Superscalar.

A. Design. This was the more ambitious part of our CPU. We were planning on making a 2-wide issue superscalar CPU. We designed our CPU assuming a later upgrade to superscalar and this shows in our architecture. We included a scoreboard in our design which would have been able to handle simultaneous issue by keeping track of register states and functional unit execution state. We also separated our frontend and backend with a pop queue. This queue allowed us to increase throughput in situations where either the frontend or backend could be operating faster at any given time.

We did not end up actually implementing superscalar in the end, due to a lack of time and other design priorities.

4. Additional Observations. There were some issues when it came to making our design. I think that one major blockage was that we did not set up RVFI early enough into our work. RVFI really helped reveal some errors that we did not see with the checkpoint 1 and 2 code. Apparently our jump logic had some errors and sometimes we would go to incorrect memory locations. Also RVFI really helped for checkpoint 4 and verifying that all our instructions were working correctly. Additionally we greatly underestimated the debugging process in general. Specifically the time aspect of debugging. We were late with our submissions on some checkpoints because we were still debugging. However we eventually got our designs working which was good. For our final note we will be listing our competition codes timing and power since it is not listed anywhere else on this report.

Competition Code	Time	Cycles	Total Power
comp1.s	448827.815 ns	80579 cycles	738.37 mW
comp2.s	840142.595 ns	150833 cycles	730.74 mW
comp3.s (Zbb)	432591.265 ns	77664 cycles	631.67 mW
comp3.s (no Zbb)	695862.885 ns	124930 cycles	653.37 mW



5. Conclusion. Overall we did end up achieving many of our goals. We made a fast and efficient CPU. We definitely beat the baseline scores. We were able to include pipelined caches, the Zbb extension and a GShare branch predictor, all of which demonstrably increased the performance of our target workload. In this project we learned a lot about computer architecture. In particular we came to better understand the techniques, optimizations, and trade offs engineers look at when architecting designs.