ECE 411 MP4 Final Report

Edwin Louie, Ryan Metzger, Sohil Pokharna
edwinml2, rtm4, sohilp2
Three Don’t Cares

I. Introduction

Our team was tasked with designing a five-stage in-order pipeline RISC-V processor. The motivation for doing so was to gain a better understanding of the operating principles behind pipelined processors and have an opportunity to explore some advanced features like caching, branch predictors, and integer multipliers that many modern CPUs employ. Another aspect of the project was to learn more about RISC type ISAs in general and get a better feel for the RISC-V ISA. Recently, RISC-V is gaining a lot of popularity due to it being an open-source architecture which allows people in both academia and industry to try design new processors without paying licensing fees to use other ISAs like x86 or ARM.

II. Project Overview

This project at its core involved implementing and designing various caches, forwarding paths, branch-predictors, and learning how to deal with memory interfaces and handle stalls/flushes in the pipeline. Additionally, there was an area and clock speed constraint that we needed to meet when the last version of the processor was published. Throughout the project there was also a motivation to maximize performance in terms of power-delay product as our processor was pitted against other team's processors in a final design competition. We ended up with a working design that utilized advanced features such as a tournament branch predictor, RISC-V M extension, and an upgraded set-associative Pseudo LRU cache system with a shared L2. The following report will be dedicated to a broad overview of design decisions, an explanation of major milestones, a summary of the advanced features, and ending with some additional commentary on how this project can be expanded/improved upon moving forwards.

III. Design Description

A. Overview

The final five-stage pipeline consists of Fetch, Decode, Execute, Memory, and Writeback stages. It encompasses a two-way set associative single-cycle hit cache for fetching the data and instructions. The L2 cache is a eight-way set associative two-cycle hit cache using pseudo LRU replacement policy. The data and instruction cache share the same L2 cache, so an arbiter decides which cache and when each cache can access the L2. The L2 is connected directly to the main memory through a cacheline adaptor. Our design also incorporates a tournament branch predictor, but still has room for improvement. The processor also supports the RISC-V M-extension; the biggest feature of this implementation is single cycle multiplication. We started with a basic processor and each checkpoint expanded the operation of the pipeline to allow for more advanced features. The milestones below highlight the design journey and serve as a sequencing of the design/debugging process and how our processor evolved over the semester.

B. Milestones

i. **Design Checkpoint:** Our first major checkpoint was designing a datapath of the simplest 5-stage RISC-V pipeline. This pipeline relied on NOPs being inserted into the instructions after any dependent instructions and data hazards as the pipeline had no forwarding paths or stalling implemented. We also designed a control word generator based on the opcodes to send operation/memory/write/select etc. control signals down the pipeline to the appropriate hardware at each stage. At this point, our designed relied on a provided “magic memory” blackbox that allowed the processor single cycle data and instruction memory writes/reads. We designed around this abstraction so that we could focus purely on the basic datapath, leaving more complex memory implementations for future checkpoints.

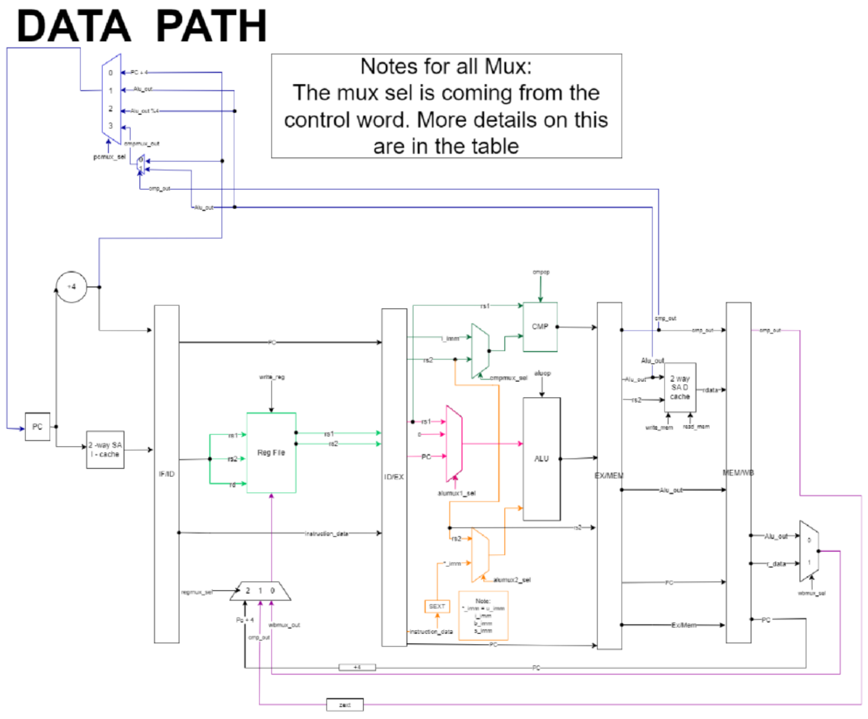


Fig A. Basic Datapath

ii. **Checkpoint 1:** This next checkpoint was the beginning of our basic pipeline processor. We focused on creating the individual components. Edwin and Sohil worked mainly on the five pipeline stages while Ryan worked mainly on the control word generator and its propagation through the entire pipeline. The processor at this stage did not support forwarding, but we were able to test functionality through the MP2 assembly testcode and compare the register files between the non-pipelined MP2 and our processor to confirm that the outputs matched. We had plans to add a randomizer to generate random instructions, but they fell through due to limited time.

After building our basic pipeline, we focused on designing the arbiter, forwarding unit, and static not-taken branch predictor. In the following figures, you can see our arbiter state machine and updated datapath:

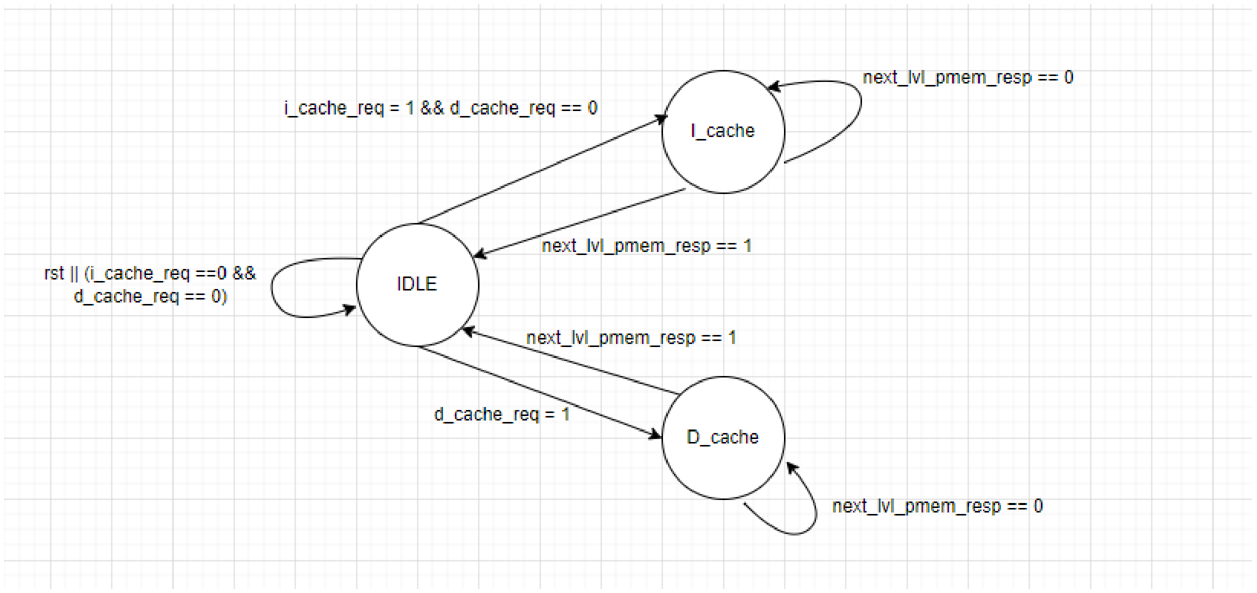


Fig B. Arbiter State Machine Diagram

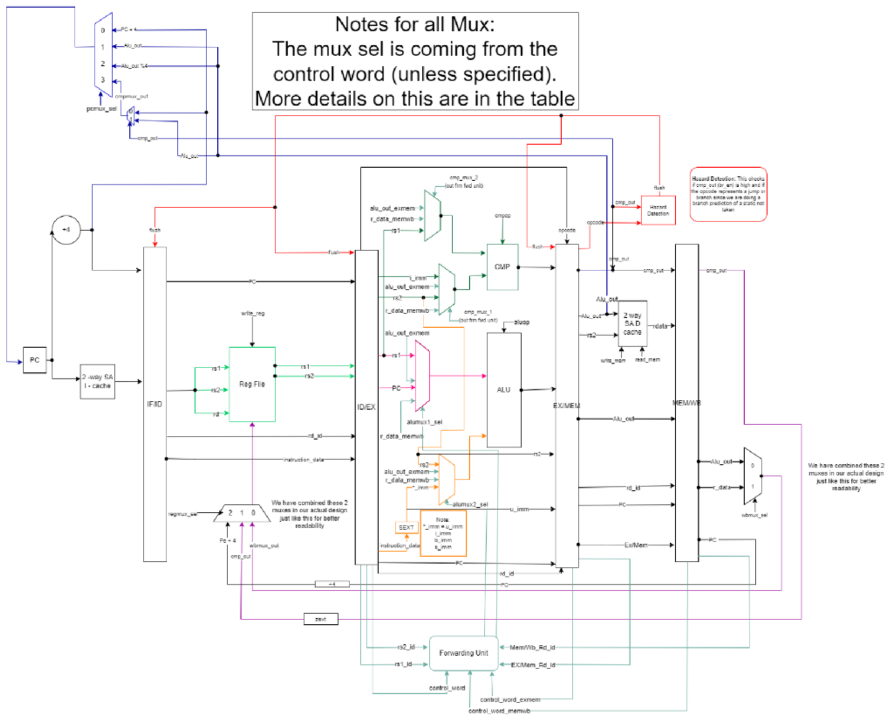

Notes for all Mux:
The mux sel is coming from the
control word (unless specified).
More details on this are in the table
PC
Reg File
IF/ID
ID/EX
EX/MEM
MEM/WB
CMP
ALU
Hazard Detection
Forwarding Unit

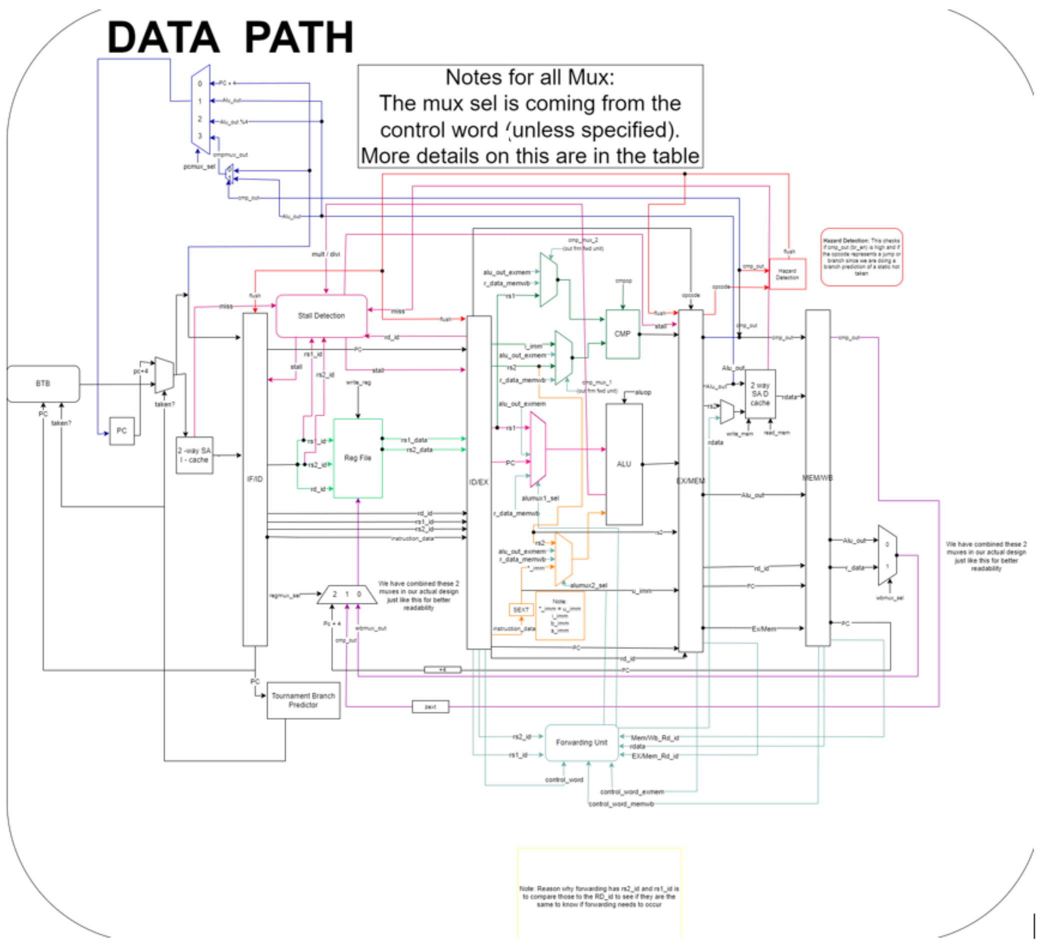
Fig C. Datapath with Forwarding/Branch Prediction

iii. **Checkpoint 2:** In this checkpoint we added more complexity to our datapath, preventing data and control hazards. Ryan focused on the forwarding unit. Sohil focused on the static not-taken branch predictor and the stalling unit. Lastly, Edwin focused on the arbiter and cache connections while also piecing the together the new components designed by the rest of the group.

Our testing strategy during this checkpoint first began with the arbiter and cache system. We ensured that the cache was connected correctly, and the arbiter was functioning by comparing the provided CP1 test code regfile against the MP2. Following that, we added in the static not-taken branch predictor along with the forwarding unit. Through the CP2 testcode, we were able to expose issues plaguing our design. To determine correctness, we used our MP2 processor and stepped through areas that were mismatched in the register file/decoding stages. Once we finished the implementation and our testcode was working properly, we start designing and overviewing the advanced features we planned to do for the next checkpoint. For more information about the designs and diagrams for these advanced features, see the sections below.

iv. **Checkpoint 3:** In this checkpoint, we focused heavily on designing and implementing the advanced features. Sohil, with some help from Edwin, extensively implemented and tested the entire cache system. Ryan focused heavily on the M extension with the advanced multiplier (Wallace Tree) and restoration divider. Edwin resolved the bugs stopping the checkpoint code from accurately running while also implemented and tested the tournament branch predictor. We also planned to do an eviction write buffer which had been designed. However, due to a lack of time we abandoned this advanced feature despite a good possibility it would greatly benefit our performance. For further information regarding the specific advanced features, see the advanced features section. Once each advanced feature was designed, we worked together to connect them in our designs. To ensure correctness, we compared our designs to the expected outputs given by our MP2 processor.

v. **Checkpoint 4:** In our final checkpoint we focused on optimizing the performance of our processor. This entailed choosing which of the advanced features we would include for the design competition. Based on the usefulness and gathered statistics, we concluded to remove the M extension. The M extension would simply be a waste of power and area as it doesn’t play a significant role in the design competition. For the tournament branch predictor, due to the flaws in the meta predictor, the best branch predictor was the one we had originally designed. As for the caches, we decided that a two-way set associative L1 cache would be a better option than a four-way set associative L1 cache. From our final design, our maximum frequency is 218 MHz and our area is 296,740 micrometers squared. The power used by each competition code is 3.36e4 microwatts, 3.57e4 microwatts, and 3.49e4 microwatts respectively.


DATA PATH

Notes for all Mux:
The mux sel is coming from the control word (unless specified).
More details on this are in the table

Hazard Detection: This checks if cmp_out (br_en) is high and if the opcode represents a jump or branch since we are doing a branch prediction of a static not taken

We have combined these 2 muxes in our actual design just like this for better readability

We have combined these 2 muxes in our actual design just like this for better readability

Note: Reason why forwarding has rs2_id and rs1_id is to compare those to the RD_id to see if they are the same to know if forwarding needs to occur

Fig D. Latest Datapath with Advanced Features

IV. Advanced Design Features

1. Tournament Branch Predictor

Design:

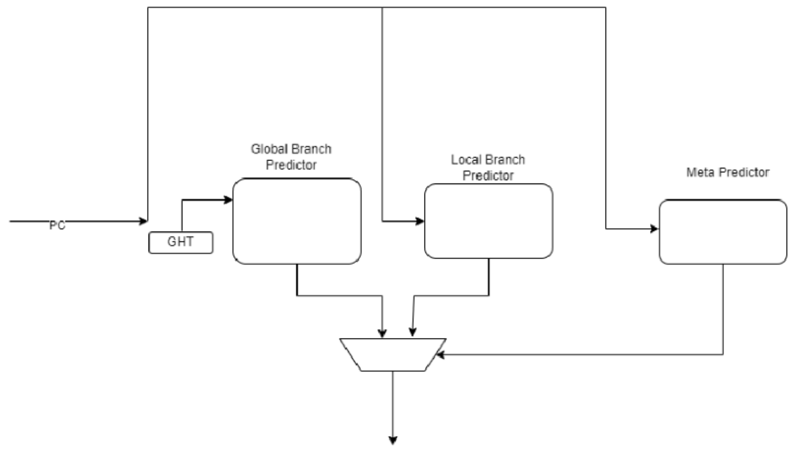


Fig E. Tournament Branch Predictor Structure

The tournament branch predictor consists of three main components. The local branch predictor has two levels that uses the 8 bits of the PC to index into the corresponding 12-bit wide local history register. The value of the local history register points to the index of the corresponding pattern history table entry. The global branch predictor is a Gshare predictor that uses the lowest 10 bits of the PC XORed with the 10-bit global history register. That result is used to index to its corresponding pattern history table entry. All the pattern history tables are two-bit bimodal predictors.

As for the meta predictor, it is a simple design that uses the 10 least significant bits of the PC to index into its version of the pattern history table that instead selects between local and global. All predictors are updated using the corrector/static not taken predictor we created in checkpoint 2. We realized the meta predictor is incorrect and is the root of all of the branch predictor’s accuracy problem. We currently do not provide it with feedback about if the local or global predictor should be taken, and if we had designed it that way, we would see a higher prediction accuracy.

Testing:

We designed the branch predictor piece by piece. We started by building solely the global branch predictor and inserted it in the fetch stage of the pipeline. To test correctness, we would compare the outputs and waveform to the same program ran by the MP2 processor. We repeated the same steps for the local branch predictor development. After the development of the global and local branch predictors, we added the meta predictor and placed the entire branch predictor black box module into our MP4 design and verified correctness through the competition and CP3 testcodes.

Performance Analysis:

The average correct prediction rate for the tournament branch predictor is about 73.7%. Due to the issue with the meta predictor, many of the changes and adjustments to find the best performing branch predictor failed and yielded no significant benefit. If we increased the size of the predictors, it would become too specific to the PC whereas if we decreased the size, it would become too generic. Increasing the size of the predictor leads to a lack of training for each predictor entry. If the branch predictors become too generic, they will be influenced by unrelated branches which would lead to inaccuracies. From some trials, we found that decreasing the width of the global history register increased Comp1's and Comp3's accuracy, but decreased Comp2's accuracy, providing an average of 72.6%. This would mean that Comp1 and Comp3 are more dependent on the local branch predictor. If we increased the size anymore to either components of the branch predictor or decreased the widths, the prediction accuracy would decrease significantly. Much of these low accuracies can be attributed to the original design of the meta predictor (for more information see the design section).

2. M Extension

Design:

The M extension involved expanding our processor to support two main new operations: 32-bit multiplication and division. Specifically, the RISC-V ISA's official list of M-Extension operations: features such as mixed sign operations, remainder calculations, along with unsigned/signed multiplication and division.

Multiplication:

First, we decided on using a Wallace Tree multiplier for the hardware. The motivation was that this type of multiplier was worth much more design points overall than a simple shift-add multiplier design, and has the benefit of being purely combinational (single cycle multiplication).

The design is executed in the following three stages:

1. Generate partial products
2. Compress partial products into two sums
3. Add final sums together using a fast adder

Since we needed to support signed operation, we modified the multiplier slightly - using the Bough-Wooley method to transform the basic Wallace Tree multiplier into a two’s complement multiplier. Bough-Wooley approach involves complementing some of the bits of the input, adding an additional 1 bit into the first layer, and complementing some of the output bits. This clever bit hack allowed us to perform signed multiplication using a modified Wallace Tree structure. One problem remained, however, dealing with mixed sign multiplication. The solution that our team produced was to make a 33-bit multiplier and either zero extend or sign extend the input depending on whether it was signed or not, promoting all multiplication to signed multiplication so that our signed multiplier functioned properly.

After the design was finalized, a python script was created, generating an N-bit Bough-Wooley Wallace Tree in SystemVerilog since our team was not as comfortable with complex SystemVerilog generation schemes. Generating the compression hierarchy was the most difficult part in terms of problem solving but was successful: using a 3:2 compression scheme consisting of Full/Half Adders. For the last stage, a simple 64-bit RCA (ripple carry adder) was used as time was short, but this should have been replaced by something like a CLA (carry lookahead adder) to achieve a higher operating frequency.

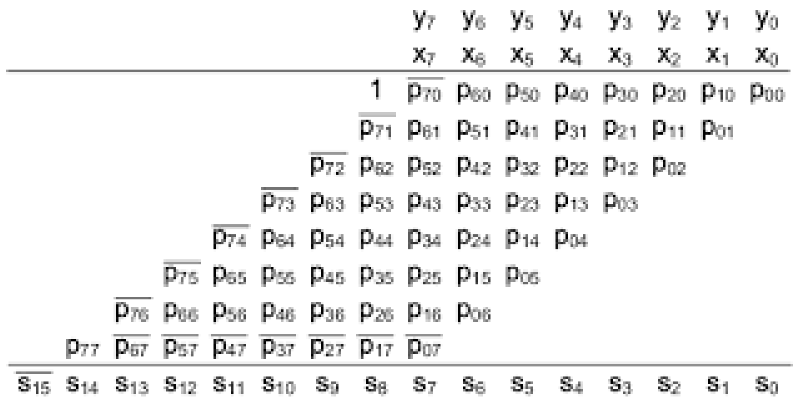

y7 y6 y5 y4 y3 y2 y1 y0
x7 x6 x5 x4 x3 x2 x1 x0
1 p70 p60 p50 p40 p30 p20 p10 p00
p71 p61 p51 p41 p31 p21 p11 p01
p72 p62 p52 p42 p32 p22 p12 p02
p73 p63 p53 p43 p33 p23 p13 p03
p74 p64 p54 p44 p34 p24 p14 p04
p75 p65 p55 p45 p35 p25 p15 p05
p76 p66 p56 p46 p36 p26 p16 p06
p77 p67 p57 p47 p37 p27 p17 p07
s15 s14 s13 s12 s11 s10 s9 s8 s7 s6 s5 s4 s3 s2 s1 s0

Fig F.   Bough Wooley Algorithm

Division:

For the divider we decided to implement a restoration divider. This type of divider takes 2n cycles to complete where n is the number of bits. There are dividers that complete in n cycles, but this divider was chosen because the focus of this extension was on the Wallace Tree generation, and we were short on time. Restoration dividers are one of the simplest types of dividers to design. This divider was designed with a width of 33-bits to support both signed and unsigned division. An absolute value operation was applied to the inputs into the divider, and a sign correction was done at the output of the divider. The basis for restoration division is like long division - keep subtracting from the dividend until the remainder is greater than or equal to zero but less than the divisor. The restoration divider circuit always subtracts, checking after to see if the remainder is non-negative. If the remainder is negative, it undoes the previous subtraction. This two-step process is the reason of 2n cycle time complexity. A 32-bit RCA was used to implement the subtraction step and registers with a basic state machine were used to realize the algorithm. Shown below is a more detailed diagram of the restoration algorithm.

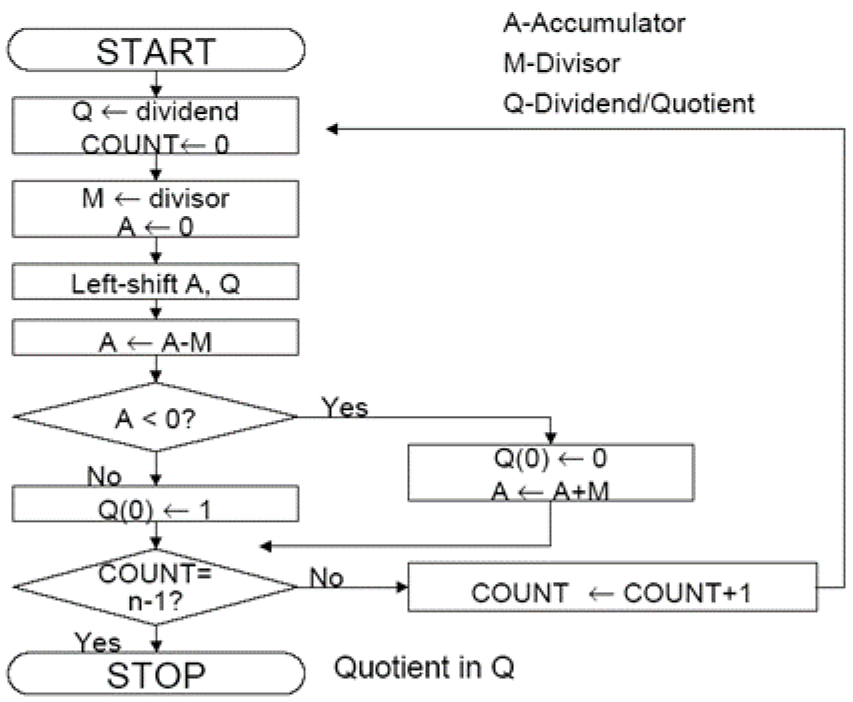


Fig G. Restoration Division Algorithm.

Testing:

There were two stages to testing. Testing the individual multiplier components and testing them integrated into the datapath.

To test each individual component, random multiply and division instructions were generated along with random 32-bit operands. This data was connected directly to the multiplier and divider modules and the output from these modules was compared against SystemVerilog's built in multiply, divide, and modulo operations to check the results' correctness. If a mismatch occurred, an error counter would increase by 1. Over 5,000 randomized tests were generated for both the multiplier and divider, all resulting in 0 error counts. Additionally, edge cases such as division by zero, signed division overflow, and multiplication by zero, were explicitly tested after the randomized section of the testbenches to match RISC-V specifications.

The units were then integrated into the pipeline, adding to the control word generator to support M instructions. The funct7 section indicates that a M type of register operation is occurring, and the func3 specifically chooses one of the eight M instructions to perform. A mux was also added to the original ALU output to select between the ALU, multiplier and divider output. The inputs into the ALU were fed directly into the multiplier and divider units and a stall signal is generated when the divider is performing calculations. We wrote extensive assembly code to test each of the types of operations and edge cases for both multiplication and division operands. We then checked the register file at the end against our expected register file and they matched, indicating success!

Performance Analysis:

The multiplier unit, synthesized outside of the pipeline, had a maximum operating frequency of 327.86 MHz. We strongly believe that this frequency could be pushed much higher if given extra time to implement a faster adder for the last stage. Again, we were short on time, so we used the simpler, yet slower CRA, when a CLA could boost operating frequency. When inserted into the pipeline, the M-extension and its associated hardware dropped the fmax from 218 MHz down to 121.359 MHz. This was a major hit on performance as well because our frequency dropped by 50%, and the Wallace Tree is power hungry, further degrading power-delay performance. Additionally, the inputs into the multiplier are not latched when the multiplier is not in use. This leads to a lot of dynamic power dissipation which can eventually be resolved by only switching the inputs when a multiply instruction is provided. The Wallace Tree a significant area with a whopping 6000 micrometers squared. This is mostly due to the immense amount of logic with 967 Full Adders and 165 Half Adders, in addition to the RCA in the last stage. Also, partial product generation alone required 33*33 = 1089 two input AND gates. A counter was added to the divider state machine to measure performance. The divider performed poorly, taking 66 cycles instead of the alternate 33 cycles with non-restoring operation. This unfortunately means the pipeline stalls for 66 cycles which is a huge hit on runtime as opposed to only stalling for 33. Because of the degradation in performance mentioned above, we decided to exclude the M-extension for the design that we submitted to the competition.

3. Cache Upgrades:

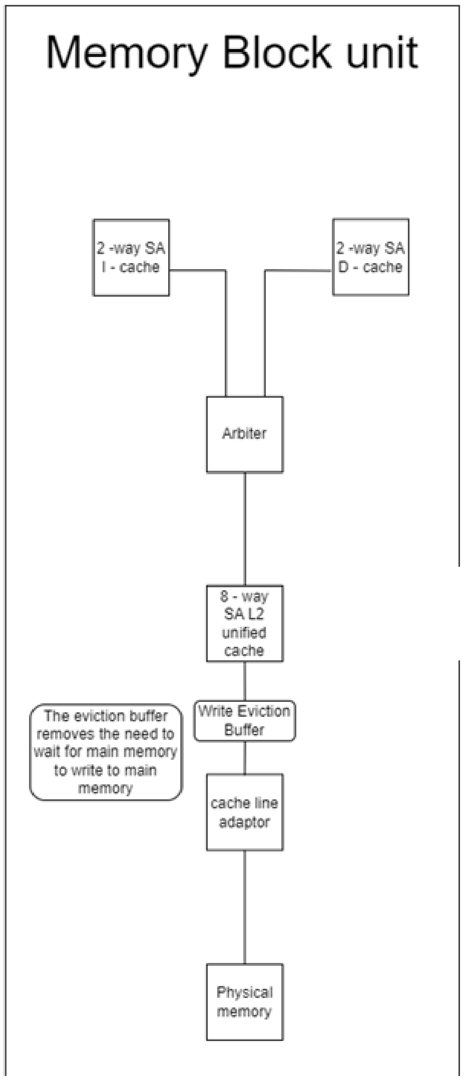


Fig H. Memory Block Box Diagram

Design:

2-way set-associative: This was based off our MP3 code except we had to make it a 1 cycle instead of a 2 cycle hit. This was achieved by removing our idle state and combining the functionality of our idle state in our "hit" state. Now, we are always comparing the cache ways to the address bits passed in even if the read or write signals are low, but we don't do anything with that output until the read or write signals are high.

8-way set-associative: This was again based off our MP3 code except now we increased the number of ways, and we no longer need a write_sel_way mux since we are always replacing the entire cacheline. We also used a Pseudo-LRU replacement policy and the diagram of it can be seen below. We decided to keep the 8 way set associative cache as a 2 cycle hit that way we aren't wasting power on continuously comparing the 8 ways of the cache when we didn't need to.

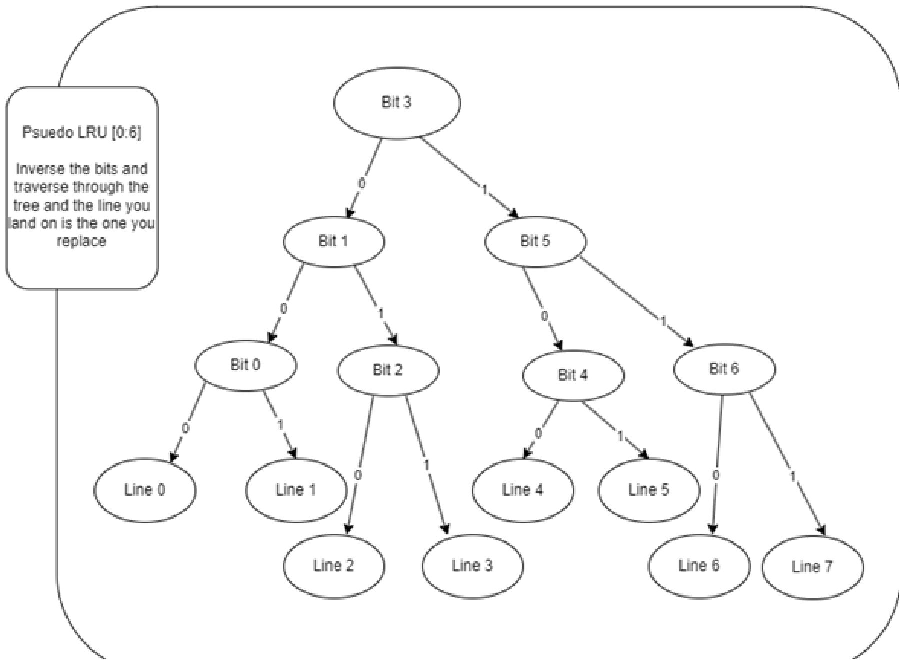


Fig I. Pseudo LRU Overview

Python Code: The python code was used to generate our 4-way set-associative caches with a 1 hit cycle and could create any N-way set-associative cache using the Pseudo-LRU replacement policy. This was achieved by created a binary search tree and then using DFS to traverse it to find which bits aligned with which way. We saved this relationship in a dictionary where the key was the way and the value was a list of the bits that were used to represent the Pseudo-LRU algorithm. Then we just followed the same logic for the other caches we created to generate the script.

Testing: We tested by placing the newly developed 8-way L2 cache and Python generated parameterized caches into MP3 to allow for the Shadow Cache monitor to check for possible errors. We were able to find bugs related to our Pseudo-LRU algorithm utilizing this method. After fixing the errors, we moved forward to test our cache on the various MP4 testcodes to verify they behave exactly as expected and compared it to the MP2 processor output. We then also would replace our MP3 cache with the generated caches, checking the register output to ensure correctness.

Performance Analysis:

Cache hit and miss rates							
		2-Way		4-Way		8 Way	
		D-Cache	I-Cache	D-Cache	I-Cache	2-Way	4-Way
Comp1	Hit Rate	99.75%	99.96%	99.75%	99.67%	0.00%	16.67%
Comp2_i	Hit Rate	99.14%	96.07%	99.45%	99.76%	98.75%	82.39%
Comp3	Hit Rate	97.46%	87.48%	97.46%	99.94%	95.64%	0.95%
Comp1 + Comp2 + Comp3	Average Hit Rate	98.78%	94.50%	98.89%	99.79%	64.80%	33.33%

Table A. Cache Performance Table

Due to lack of time, we mainly wanted to test the difference between a 4-way and 2-way L1 cache. To do this, we compared the hit count against the total access count. In Table A, you can see the different hit rates between the two caches. By using this data, we ultimately decided that there wasn't much difference between the 2- and 4-way cache in terms of performance. However, since both were 1 cycle hits the 4-way cache would have used up more power since we are always comparing the tag bits against the ways, even when we are not trying to read or write from that cache. Furthermore, the 4-way cache would take up more area. Due to all these reasons, we felt that the 2-way set-associative cache was better for our use case.

V. Conclusion

With our limited time, we were still able to accomplish a great amount. We were able to make our design unique through the implementation of a Wallace Tree multiplier along with using Python to create some of our SystemVerilog code. To further improve the ability and performance of our processor, we included several advanced features such as the tournament branch predictor, upgraded cache hierarchy, and M extension in RISC-V.

The overall final product had plenty of room for improvement. With some more time, we would have tested and implemented the eviction write buffer which improve performance during a cache dirty eviction. Moreover, our code was full of inefficiencies which could be refined and rewritten to improve the number of multiplexers and other hardware modules. This would decrease our power consumption and area usage. With more time, we would also investigate the M-extension hardware to improve the hit on fmax. We would replace the ripple carry adder with a faster adder and modify the divider to be non-restoring. Another thing our team discussed was implementing a return address stack. Unfortunately, neglected forwarding paths and edge cases led to wasted hours in debugging, preventing the return address stack implementation.

Throughout this project, we designed and implemented a working RISC-V processor. We integrated many of the topics discussed in class, and this processor reinforced our foundations in computer architecture. Moving forwards, it would be interesting to attempt and implement an out-of-order pipelined processor using Tomasulo. Out-of-order computing/parallelism/multi-core computing is just of the few advanced topics we were interested in but did not have the time to tackle this semester. If our team were to design another processor, we would focus on out-of-order computing as those designs are fascinating to learn about and are most relevant to modern day silicon.