ECE 411 MP4 Report

Fall 2023

5-Stage Pipelined RV32i Processor

Team: Superscalar

Members: Dongming Liu, Elijah Ye, Tracy Miao

NetIDs: dl35, gaohany2, mmiao2

(1) Introduction

This project presents the development of a five-stage pipelined CPU capable of accurately executing the RV32i instruction set architecture (ISA). Our pipelined architecture deconstructs instruction processing into five distinct stages: instruction fetch, decode, execute, memory operation, and data write back. This compartmentalization facilitates a swifter and more efficient instruction throughput. The overarching objective of this project is to realize a pipelined CPU adept at handling diverse workloads and achieving superior processing speeds. This report offers a comprehensive overview of the project, encompassing the design and implementation of our pipelined CPU, the integration of advanced features, and the testing methodologies employed to validate its functionality.

(2) Project Overview

This project endeavored to design and implement a five-stage pipeline CPU, aiming to enhance performance and efficiency in computing systems. The primary motivation stemmed from the need for a pipeline architecture capable of tackling diverse workloads and achieving improved processing speeds. To this end, we meticulously examined critical factors including processor architecture, memory management, cache design, branch prediction, and testing methodologies. Our selection of these areas aligned with current industry standards and demands for high-performance computing. Recognizing the increasing pressure for faster and more efficient processing, we deemed a pipeline CPU adept at handling diverse workloads to be crucial. Furthermore, our project sought to incorporate advanced features, including cache systems and branch prediction, for further performance optimization. The project's organizational and administrative aspects involved collaborative work and effective project management. In terms of advanced features, we initially brainstormed a list of potential candidates. Subsequently, each team member chose the topic that most resonated with their interests. Elijah (gaohany2) focused on multilevel caching and prefetching, Tracy (mmiao2) tackled the eviction buffer, and Dongming (dl35) pursued the M-extension. Notably, both Tracy and Dongming contributed to the parameterized cache design.

(3) Design Description

(3.1) Overview

In this project, we implemented several advanced features in our baseline pipeline processor to improve the performance of our processor. To reduce program execution time, we mainly focused on cache system optimizations and additional execution units. For cache optimizations, we implemented a two-level cache structure, fully parameterizable cache, eviction buffer, and next-line prefetcher. For the execution unit, we implemented the RV32 m extension, supporting additional RV32 instructions for multiplication, division, and modulo.

(3.2) Milestones

(3.2.1) Checkpoint 1

We implemented a basic pipeline processor that could handle all RV32I instructions except FENCE*, ECALL, EBREAK, and CSRR instructions. To avoid the stalling coming from instruction and data cache misses, we used the "magic" dual-port memory that guaranteed every memory access could be done in 1 cycle. We set up the RVFI monitoring for debugging and verifying the outputs of our processor. In addition, we provided our designs for the next checkpoint, featuring data forwarding, control hazard detection, an arbiter that controlled memory accesses of instruction cache and data cache, and a cacheline adapter.
Note: checkpoint 1 design `ECE411MP4CP1Design.png` is in our team repository, located at `main` branch, directory `docs/pics.`

(3.2.2) Checkpoint 2

For data and control hazards, we implemented data forwarding, control hazard detection units, and a static-not-taken branch prediction. We modified our register file so that it could handle data forwarding for the writeback stage. We implemented a one-cycle-hit instruction cache, a 4-way set associative data cache, and an arbiter that controlled and served the requests from both caches. The arbiter prioritizes data cache requests if both instruction cache requests and data cache requests are presented simultaneously. Because we use the burst memory in this checkpoint, we connected a cacheline adapter to our arbiter to help transfer the data to/from the burst memory.
Note: checkpoint 2 design `ECE411MP4CP2Design.png` is in our team repository, located at `main` branch, directory `docs/pics.`

(3.2.3) Checkpoint 3

We upgraded our cache system to support a two-level cache structure, which consists of L1 caches (instruction and data cache) and a unified L2 cache. To make the cache system flexible, we had all the cache parameterizable. To hide the memory latency, we implemented an eviction buffer for data cache writeback and a next-line prefetcher for instruction cache. In addition, we implemented the m extension in RV32 that consisted of a fast multiplier (dadda tree multiplier) and a simple divider (shift-subtract divider). With the m extension, programs do not need to have software solutions for multiplications and divisions. These advanced features enhance the overall performance by reducing main memory accesses and providing more instruction options.

3.3 Advanced Design Options

3.3.1 RV32 M Extension:

The M Extension unit we implemented can support all the instructions under RV32M standard. The multiplication instructions are done by the dadda tree multiplier, and the division and remainder instructions are done by the simple subtract-shift (or subtract-shift) divider.

3.3.1.1 Design:

Algorithm of Dadda Tree:

Note: j is the stage index, i is the column index

We first need to compute the stage required for 32-bit by 32-bit multiplication.

d_j is defined by: d_1 = 2, d_(j+1) = floor(1.5 * d_j)

The value of j is chosen as the largest value such that d_j < 32.

d_1 = 2, d_2 = 3, d_3 = 4 , d_4 = 6, d_5 = 9, d_6 = 13, d_7 = 19, d_8 = 28

So the value j = 28

For each stage from j to 1, reduce each column start at the lowest weight column c_0 according to the following rules:

1. If height of c_i <= d_j , not reduction, move to c_(i+1)
2. If height of c_i <= d_j , add the top two elements in a half-adder, placing the result at the bottom of the column and the carry at the bottom of column c_(i+1), then move to column c_(i+1)
3. Else, add the top three elements in a full-adder, placing the result at the bottom of the column and the carry at the bottom of column c_(i+1), restart step 1

There are two rows of bits at the end

The result of the multiplication is the sum of the two rows of bits

We used a Python script to simulate the algorithm above and generate the dadda tree module. For more information about simulation and generating script, we include it in our GitHub team repository under a branch called `m_ext_fixing`. The file is called `dadda_tree.ipynb` under the directory `dadda_tree`.

Algorithm for divider:

We use the long division algorithm for our divider

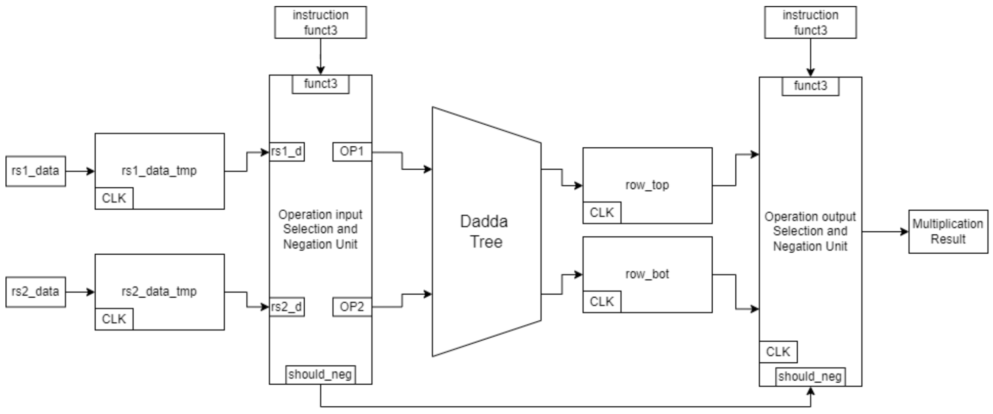
D = divisor, N = dividend, Q = quotient, R = remainder, i = the bit index of N

```
if D = 0 then raise division by 0 error
Q = 0
R = 0
for i = n − 1 … 0 do
```

```
R = R << 1    # Left-shift R by 1 bit
R(0) = N(i)     # Set the LSB of R to bit i of the numerator
if R >= D then
        R = R − D
        Q(i) = 1
```

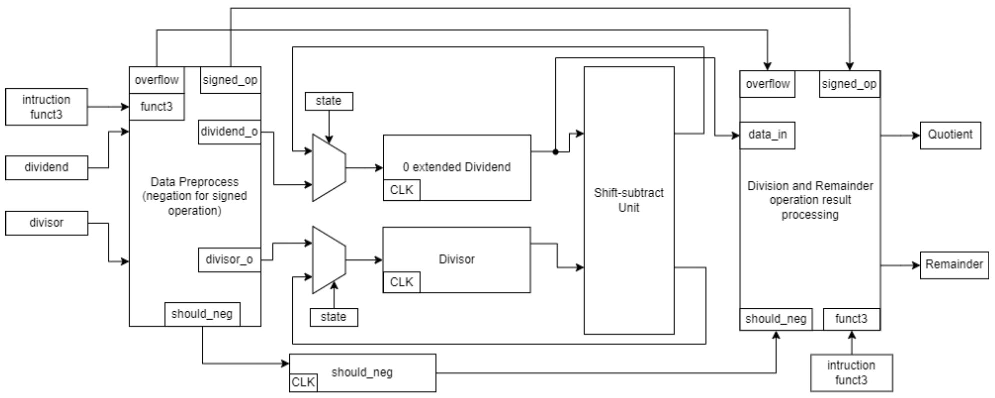
Datapath diagrams:

Note: the control logic (FSM) is not shown.



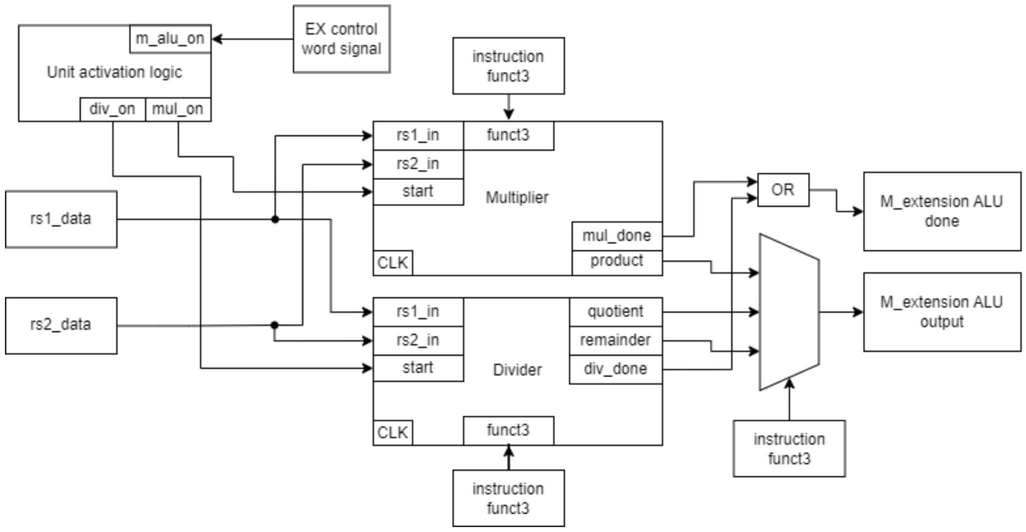
Multiplier datapath

Note: the outputs of the dadda tree are two 64-bit unsigned numbers, which means that row_top and row_bot are 64-bit registers.



Divider datapath

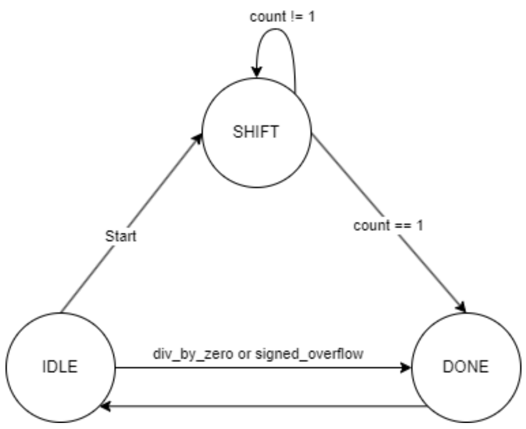
Note: the zero-extended dividend register contains 64-bits. After the division is completed, the lower 32-bit is the quotient and the upper 32-bits is the remainder.



M Extension ALU

Control Logics:

The multiplier and divider use two different control logics. The multiplier uses a binary up counter to keep track of the number of cycles required for operations, whereas the divider uses a complicated FSM to control. During the implementation of the multiplier, we discovered that the result for MUL instructions was not related to the upper 32-bit of the outputs from dadda tree. So we were able to complete the operation within two cycles. Other multiplications used the upper 32-bits. This indicated that we could either use a 64-bit full adder or pipelined two 32-bit additions to obtain the results. We chose the latter because the 64-bit full adder creates a long critical path for our design. The disadvantage of pipelined two 32-bit additions is that the multiplier requires one more cycle to compute the results. For MUL, the multiplier sets the target count of its counter to 2. For other multiplication instructions, the multiplier sets the target count to 3. The division algorithm we used is the long division algorithm. The shift-subtract divider FSM simply mimics the control logic of the long division. For a 32-bit division, the divider needs 32 cycles to complete the division based on the long division algorithm. The FSM is the following:



The FSM for divider

IDLE state: no operation request is sent to the divider
SHIFT state: the divider does bit shifting and subtracting based on the zero-extended dividend and divisor
DONE state: the divider finishes the division operation, and quotient and remainder are ready

3.3.1.2 Testing:
We wrote a separate testbench specifically for our m extension ALU. We tested all the multiplication, division, and modulo functionalities of our m extension ALU against the built-in SystemVerilog multiplication, division, and modulo operations. After that, we integrated the m extension to the baseline processor. We wrote a separate testing assembly code that touches on signed-signed and unsigned-unsigned operations for multiplication, division, and modulo. Because the m extension multiplication has an unsigned-signed operation, we wrote some codes for that as well. We compared the results of our processor against the golden spike.

3.3.1.3 Performance Analysis:
Both implementations run at a clock period of 10000 ps. Vanilla is our baseline implementation.

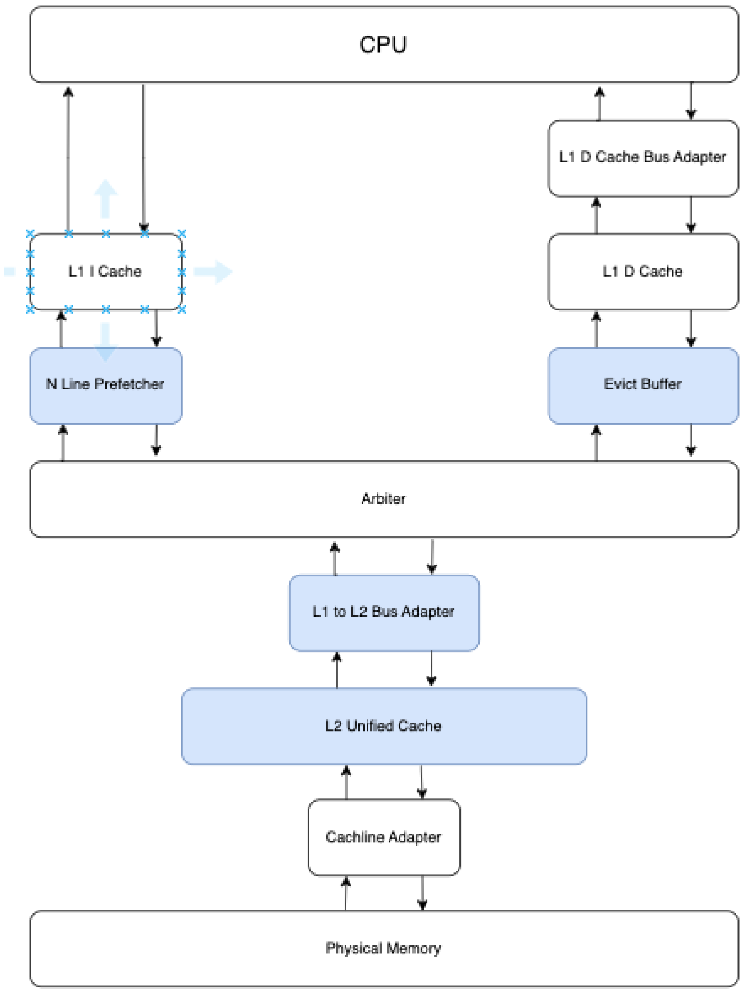
	Area	Power (uW)	Stop Time (ns)
vanilla	77335.11475	3.10E+03	15890000
vanilla with m_extension	86177.48676	3.50E+03	7517430

Although the area and power increase, the program execution time is sped up by 2.11x. With that, the PD^2 and PD^3 are much better than the baseline version.

(3.3.2) L2 Cache
A key approach to enhance overall CPU performance is optimizing the cache hierarchy by reducing miss rates and miss penalties. Implementing a second-level unified L2 cache augments the L1 instruction and data caches to form a multi-level cache system. The L2 functions as a victim cache, caching lines evicted from the smaller L1 due to capacity or conflict misses. Sitting

closer to the processor, the L2 cache can provide data and instructions at lower latency compared to main memory. By exploiting additional locality, the augmented hierarchy better supplies the demands of the L1 caches. This system-level integration of an extra cache level allows leveraging access locality properties across a wider working set and timeline to boost hit rates. Overall application performance sees gains from the decreased miss overhead through synergy between the coordinated caches.

(3.3.2.1) Design



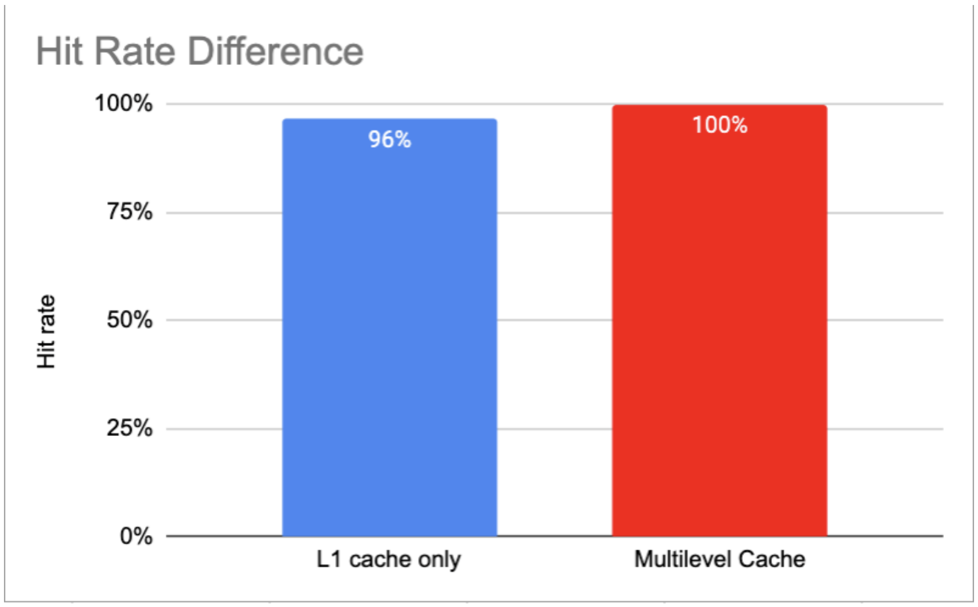
Memory System Overview

Our processor integrates a unified L2 cache between the main memory arbiter and cacheline adapter. On an L1 I-cache or D-cache miss, requests get forwarded to the arbiter, which prioritizes D-cache misses over I-cache misses when scheduling memory access. By servicing data misses first, program execution can resume sooner. The corresponding memory response then flows into the unified L2 cache rather than directly to the L1 caches. By caching the returned data, the L2 cache can service future requests directly, avoiding additional arbiter traversal. Any subsequent requests then hit in the now populated L2 cache line, achieving lower latency hits compared to main memory. These hits propagate upwards, filling L1 cache lines until data ultimately reaches the CPU to complete the miss. Overall this multi-level hierarchy with a unified victim L2 cache aids performance by exploiting locality that may be outside L1 working sets.

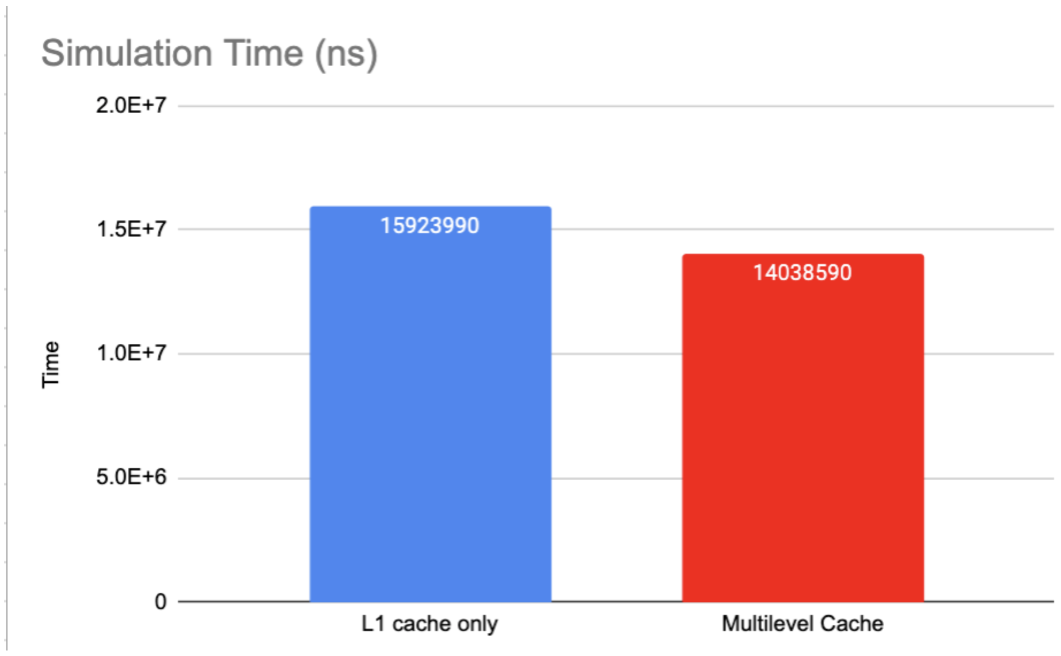
(3.3.2.2) Testing

We test our multilevel cache by running coremark and comparing the results against the results from the golden spike, using Verdi to observe cache behaviors in terms of different parameter settings.

(3.3.2.3) Performance Analysis



Hit Rate for L1 Cache and Global Cache



Simulation Time for L1 Cache and Global Cache

Expanding to a multilevel cache hierarchy significantly improved overall program performance despite high baseline L1 hit rates. The L2 cache builds on the L1 strength by providing a secondary level to capture additional locality. Together, their aggregated hit rates neared 100% by leveraging complementary working set coverage. However, the implementation exceeded area constraints for a practical design. Generous sizing enabled strong performance but unrealistic capacity. Employing parameterized modeling and tuning the hierarchy to meet strict area limits consequently impacted metrics. With more constrained capacity, hit rates, miss penalties and ultimately application speed suffered. This illustrates key system-level tradeoffs between chip real-estate, cache size, hit ratios and total performance. Extensive design space exploration allows optimally balancing factors like area versus speed.

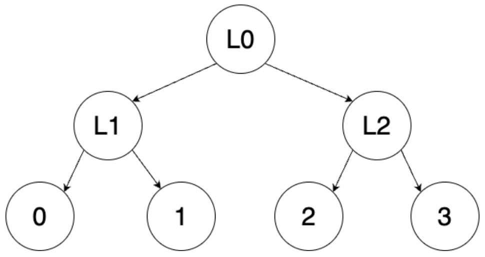
(3.3.3) Parameterized Cache

We implemented fully parameterized cache in terms of numbers of ways, sets and cache line sizes in order to do convenient parameters tuning for meeting the area constraint. Due to the simplicity of parameterizing cache sets, we prioritize on discussing the approach of parameterizing ways and cache line sizes, of which the most significant parts are parameterized PLRU and L2 bus adapter.

(3.3.3.1) Design

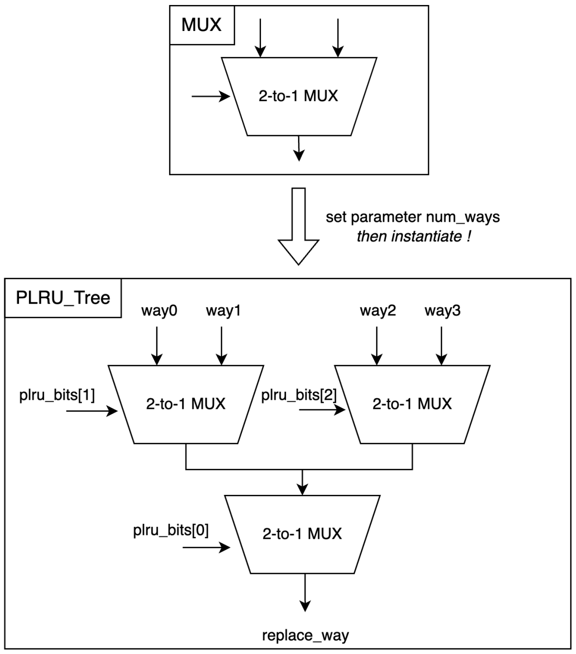
Parameterized PLRU:

We discovered that the PLRU tree is a perfect binary tree. For 4-way, we structured PLRU bits as [L2, L1, L0], and we built the binary tree the following way:



PLRU searching tree

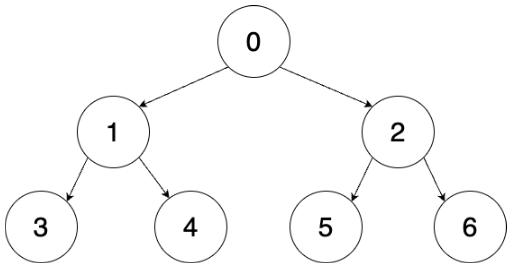
The rules are the following: if the bit is 0, take the left branch; if the bit is 1, take the right branch. Two-to-one MUX can help us implement such a rule, so the hardware structure of the PLRU searching tree looks like the following:



PLRU searching tree in hardware

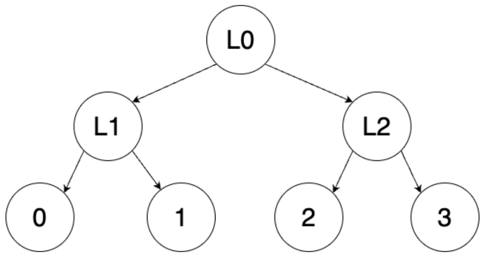
After seeing the pattern from 4-way, we can construct the PLRU searching tree for 2-way, 8-way, and even more, as long as the number of ways is a power of 2 and greater than 1.

Like the searching tree, the update logic for PLRU bits is required if and only if the number of ways is a power of 2 and greater than 1. The PLRU update logic was tricky to implement in SystemVerilog, so we decided to write a Python script to generate the PLRU bits update logic. Because PLRU tree is a perfect binary tree, we can construct an array as the following:
For 4-way, the perfect tree array T = [0,1,2,3,4,5,6,7]



Binary tree diagram for array T

Assuming that we are currently at index i:
To walk the left branch, we can update i to i * 2 +1.
To walk the right branch, we can update i to i * 2 + 2
The rules for PLRU are still applied here: if the bit is 0, take the left branch; if the bit is 1, take the right. Because PLRU bits tell us the path, we only need to care about the internal nodes, which means only [0, 1, 2] in the tree are the important ones. We can treat nodes 0, 1, and 2 as the same as L0, L1, and L2.

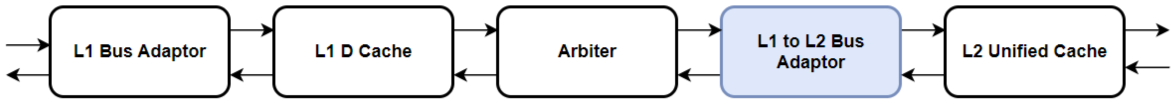


PLRU searching tree

To get to 0, the PLRU bits [L2, L1, L0] = [x, 0, 0], 0 in binary = 00
To get to 1, the PLRU bits [L2, L1, L0] = [x, 1, 0], 1 in binary = 01
To get to 2, the PLRU bits [L2, L1, L0] = [0, x, 1], 2 in binary = 10
To get to 3, the PLRU bits [L2, L1, L0] = [1, x, 1], 3 in binary = 11
Note: x is don't-care.
The reverse of the binary string for each number is the same as combining the PLRU bits without the x. We know that the update logic flips non-x control bits. In our generating script, we can use the binary string of a number, apply the software indexing rule, and find out what bit position in the PLRU bits needs to perform a bit flip (negating the path used to find the current way hit). For 4-way the software array is T = [0, 1, 2], hardware PLRU packed array is plru_bits = [L2, L1, L0] (L2 is the MSB). The mapping is 0 → L0, 1 → L1, and 2 → L2.
Generated update logic example for 4-way:
0: new_plru_bits = {plru_bits[2], ~hit_way[0], ~hit_way[1]}
1: new_plru_bits = {plru_bits[2], ~hit_way[0], ~hit_way[1]}
2: new_plru_bits = {~hit_way[0], plru_bits[1], ~hit_way[1]}
3: new_plru_bits = {~hit_way[0], plru_bits[1], ~hit_way[1]}

Note: new_plru_bits is the update. hit_way is a packed array; it is the way that cache hits
More details about the update logic generation Python script is in our team GitHub repository under the `main` branch. The file `plru_update_generate.py` is located at directory `mp4`.

Parameterized L2 Bus Adapter:
Because the L2 cache is directly connected to the burst memory and might have a different cache line size than the L1 cache. We designed a bus adapter similar to the one connected between L1 and our processor and placed it between our arbiter and the L2 cache, which is illustrated by the diagram below. By implementing the L2 bus adapter, we are able to parameterize the cache line sizes of our L1 instruction cache, data cache and L2 unified cache.



(3.3.3.2) Testing
We test our parameterized cache by running coremark and comparing the results against the results from the golden spike, using Verdi to observe cache behaviors in terms of different parameter settings.

(3.3.3.3) Performance Analysis
Groups of tests are experimented to see the data memory stalls caused by different parameter settings of our L1 cache. Vanilla is our baseline model, which is a non M-extended CPU with 4 way L1 data cache (16 sets and 256 bits cache line size). The detailed experimental results in terms of caches with different numbers of ways are shown below. In addition, "vanilla with 2 ways" means the baseline model with adjusting the number of ways of L1 data cache to 2 and keeping other parameters unchanged:

	Data Memory Stall Cycle Counts	Data Cache Miss Cycle Counts
vanilla (4 ways, 16 sets)	85447	135
vanilla with 2 ways	90609	397
vanilla with 8 ways	84377	87

	Data Memory Stall Cycle Counts	Data Cache Miss Cycle Counts
vanilla (4 ways, 16 sets)	85447	135
vanilla with 32 sets	84397	88
vanilla with 2 ways, 32 sets	85858	156

From tests above, it can be seen that increasing cache capacity by adding more ways or sets can reduce miss counts, which make the data memory stall less frequently. Compared with making 2x the number of sets, doubling the cache ways can avoid more conflict cache misses, which is coherent with what we learnt from lectures. In addition, we can apparently observe that increasing cache size cannot make more improvements if the cache size is large enough to deal with the current workload.

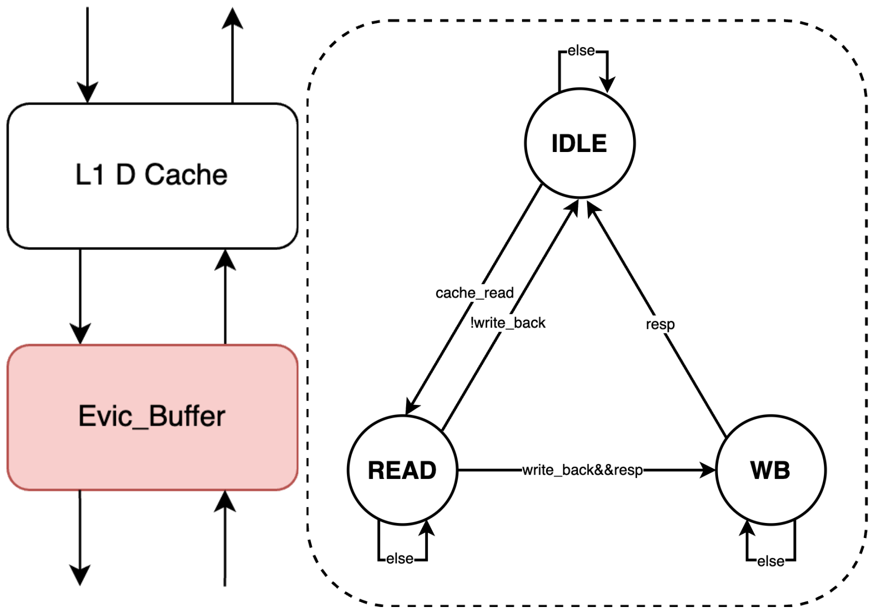
Moreover, parameterizing the cache line size gives up great help when we want to keep the L2 unified cache (which takes a large area) and meet the area constraint at the same time. By tuning the cache line size of L1 data cache as 128 bits, we allow the unified L2 cache which improves a lot in terms of the instruction memory stalls as well as extending the frequency limitation of our CPU in comparison with merely L1 data cache with a large size.

(3.3.4) Eviction Buffer

The eviction buffer is a small specialized buffer placed between the cache and memory which is mainly functionalized as a temporary storage for evicted data and then writing it back to the main memory at background, which is able to help improve the cache efficiency by reducing the write back waiting cycles.

(3.3.4.1) Design

As shown in the left diagram below, we implement our eviction buffer below the L1 data cache to temporarily store evicted data from the L1 data cache and then do the write back operation in the background. In addition, our eviction buffer is only one-set due to simplicity for implementation and area constraint concerns.
The right part diagram is a finite state machine for an eviction buffer depicted in the diagram begins in an IDLE state, where it awaits either a cache read operation or a command to write back data to main memory. Upon a cache_read signal, the machine transitions to the READ state, indicating that data is currently being fetched from the cache. Should a write-back command be issued during a read, and a response be required, the machine moves to the WB (Write-Back) state to perform data write-back operations. The WB state is designed to handle the transfer of data back to the main memory. Once these operations are complete, and no other conditions are met, the machine reverts to the IDLE state, signaling it is ready for the next operation. The 'else' paths signify default transitions back to IDLE whenever other specific conditions for remaining in or transitioning to a new state are not met, maintaining the eviction buffer's readiness for new read or write-back requests.



(3.3.4.2) Testing
We test our eviction buffer by running coremark and comparing the results against the results from the golden spike, using Verdi to observe writing back behaviors.

(3.3.4.3) Performance Analysis
In terms of the performance of the eviction buffer, we focus more on how many writeback cycles are saved compared with baseline Vanilla. The table below shows the statistics of data memory stall cycles and L1 data cache write back cycles of the comparison between baseline and baseline with eviction buffer.

	Data Memory Stall Cycle Counts	L1 Data Cache WB Cycles
vanilla	85447	565
vanilla with eviction buffer	85118	60

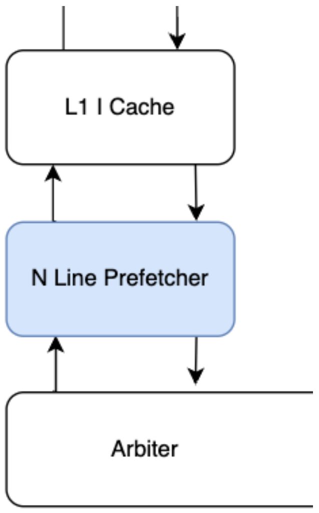
As we can see from the table, implementing the eviction buffer does really save most of the writeback waiting cycles. However, the corresponding side effect is that placing the eviction buffer between the CPU and memory can lengthen the critical path then hamper the frequency performance, therefore the overall performance increase in the data memory stall cycles is not as many as the amount of saved writeback cycles.

To maximize CPU performance, we conducted experiments to assess whether the writeback cycle reduction benefits provided by the eviction buffer outweigh its impact on frequency limitations. Our tests revealed that without the eviction buffer, the CPU could achieve a frequency of 740MHz, with power delay product (PDP) values of PD2 at 19.94 and PD3 at 23.8. Conversely, while retaining the eviction buffer improved performance at equal frequencies, it restricted the CPU to a lower frequency of 714MHz, with corresponding PDP values of PD2 at 21.25 and PD3 at 26.33. Based on these findings, we decided to remove the eviction buffer, enabling a higher frequency and superior performance, which is crucial for the CPU competition.

(3.3.5) Next-line Instruction Prefetcher

The next-line prefetcher is a common hardware performance enhancement unit that attempts to prefetch instructions from memory into the cache based on spatial locality principles. By speculatively loading instruction cache lines that sequentially follow recently fetched addresses, a next-line prefetcher helps mitigate pipeline stalls if those instructions are executed in upcoming cycles. This works well for code with largely linear control flow across neighboring cache lines. Essentially, the prefetcher makes an educated guess that the next cache line needed is directly after the last one accessed.

(3.3.5.1) Design



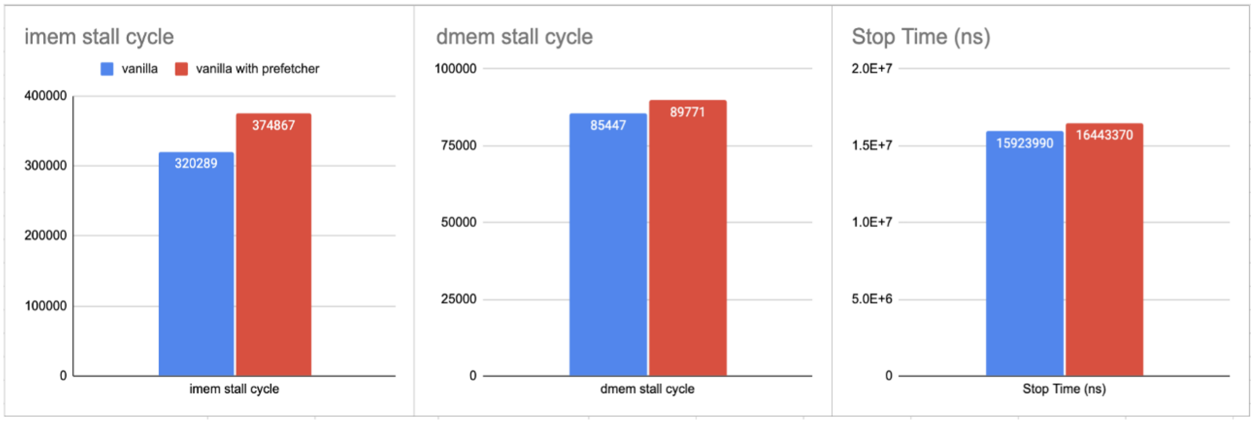
Next-line Prefetcher in Memory System

In our design, an instruction prefetcher was inserted between the L1 I-cache and memory arbiter to speculatively request the next sequential cache line on an instruction fetch. It exploits spatial locality in control flow by assuming linear code execution across cache lines. When the arbiter returns data for an I-cache miss, the prefetcher triggers, incrementing the original address to generate the next line to fetch concurrently. This hides Miss Under Miss latency if the next instructions needed are located there, avoiding a second miss to lower level memory. Instead, the prefetched line will be ready in the L1 I-cache, improving performance of linearly executing code.

(3.3.5.2) Testing

We test our Next-line Instruction Prefetcher by running coremark and comparing the results against the results from the golden spike, using Verdi to observe cache behaviors in terms of different parameter settings.

(3.3.5.3) Performance Analysis



Performance with Next-line Prefetcher

Our Next-line Prefetcher was worse in all performance metrics. After analyzing the performance counter and the code. We discovered that our Next-line Prefetcher was competing for memory bandwidth with the main program and thus made the overall performance even worse. One way to solve this issue is by moving the whole prefetcher into the arbiter. This way, we can fully utilize the IDLE state in the arbiter and thus minimize the memory contention issue with the main memory, hence improving the overall performance.

(4) Additional Observations

In this analysis, we aim to highlight key areas for future enhancements in our processor design. A notable omission in our current model is the branch predictor. Its absence has led to a significant increase in instruction memory stalls, adversely affecting the IPC. Implementing a branch predictor could thus markedly improve IPC efficiency. Furthermore, our prefetching strategy warrants refinement. The current prefetcher, instead of offering benefits, has inadvertently contributed to inefficiencies. By developing a more effective prefetching mechanism, we can further mitigate instruction memory stalls. Additionally, there is substantial scope for optimizing our eviction buffer. Due to area constraints, we had to reduce cache size, resulting in increased write-back cycles. An improved eviction buffer, if designed to avoid becoming a bottleneck in frequency during final competition stages, could significantly enhance our CPU's overall performance.

Collectively, these improvements hold the promise of elevating our processor's efficiency and competitiveness in the market.

(5) Conclusion

In conclusion, our 5-stage pipelined CPU encompassed several integral techniques to bolster performance and versatility—including M-extension to enable multiplication and division, fully parameterized caches, a unified L2 cache, next-line instruction prefetching, and writer eviction buffering. Rigorously developing and evaluating each facility verified their efficacy and illuminated future refinements. Dividing endeavors across our synergistic team facilitated concurrently engineering the sophisticated features. Specifically, augmenting the hierarchy with an additional cache level diminished aggregate miss overheads. Prefetching leveraged spatial locality to mitigate control hazard stalls. Buffer writing postponed external traffic without blocking in-order commitment. Driving extensive simulations assessed robustness across diverse codebases. In unison, the thoughtfully selected mechanisms enhanced throughput, efficiency, and flexibility. Our project imparted profound processor architecture insights. The accumulated experiences equip us to continue working on the next generation CPUs.