ECE 411 - Final Report

Team: Xiong Di _OoO

Out-of-Order(Tomasulo Alogorithm)

Team Members:

Jimmy Yu (cmyu3)

Leon Ku (leonku2)

Haoyu Yu(haoyuyu2)

Lab TA: Satvik Yellanki

April 29, 2024

Table of Contents

Table of Contents…………………………………………………………......2

Introduction……………………………………………………………...3

Project Overview……………………………………………………..….3

Design Description……………………………………………....….4

Overview

Diagram……………………………………………………..........................4

Milestones…………………………………………………..……………..**5**

Checkpoint 1 ……...................………………………………..……………5

Checkpoint 2 ……..……………………………………..…………….5

Checkpoint 3 ……………………………………………………………..6

Advanced Feature

Design…………………………………………...7

G Share Branch Predictor……………………………………………..............….…7

-Design 8

-Testing 8

-Performance Analysis 9

I – Cache (Pipeline Cache) …………………………………………………….10

-Design 10

-Testing 10

-Performance Analysis 11

Fully Parametrized Cache ……………….……..…................……….12

Load/Store Forwarding………………………………………………………...13

-Design 15

-Testing 16

-Performance Analysis 17

Next-Line Prefetcher………………………….………………………….......…18

-Design 18

-Performance Analysis 19

Conclusion……………………………………………………..………………...20

Introduction

As computer engineers at UIUC, we entered this school with the end goal of developing a high-level understanding of the inner workings of computers and specifically that of processors in this modern age. This class, and this project specifically, presented us with the unique opportunity of studying, designing, and implementing one of the most widely used and studied instruction set architecture: RISC-V. RISC-V is an open-source instruction set architecture used around the world as one of the standards for efficient and high-performance architecture sets. RISC-V covers register and memory accesses, instruction jumps, branches, advanced memory operations, and more. This instruction set is either used or directly influenced by the ISAs for every computer that we surround ourselves with today. We were able to demonstrate our knowledge of the RISC-V instruction set by designing the basic instruction set along with the out-of-order processor using System Verilog and implementing extra features to improve the overall performance of the baseline ISA.

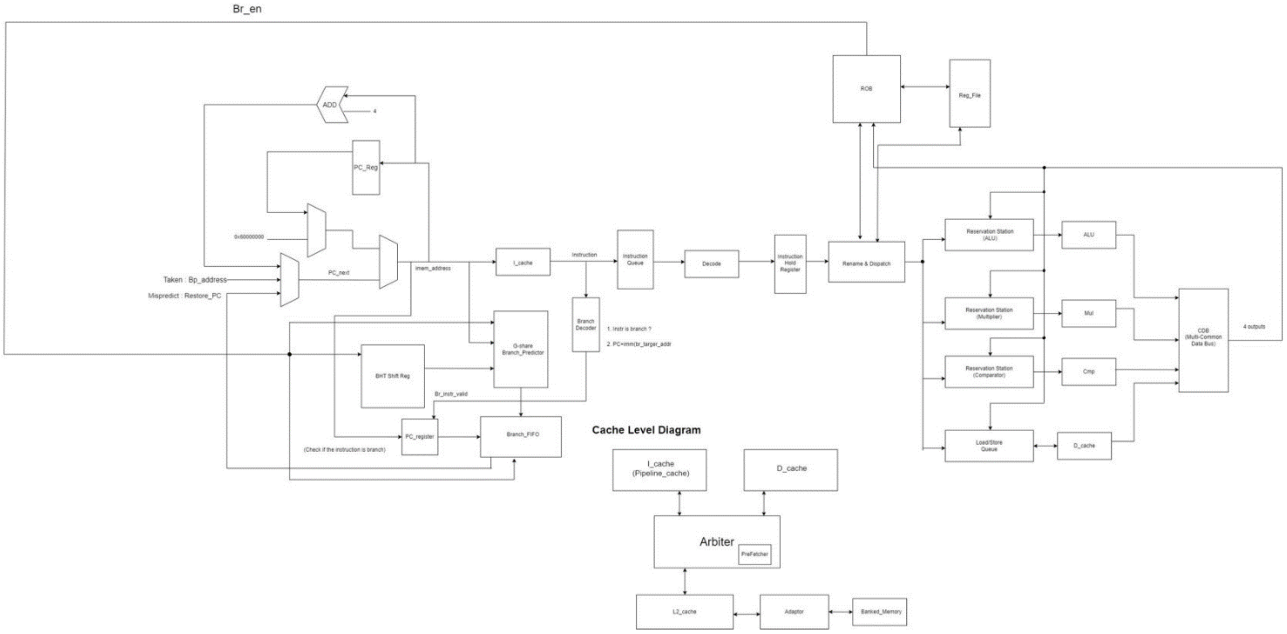
In this report, we hope to cover everything from the basic ideas implemented up including the advanced features that we chose and the reasons behind our decisions. The progressions from each checkpoint and explanation behind our advanced features and design choices will be covered as well.

Project Overview

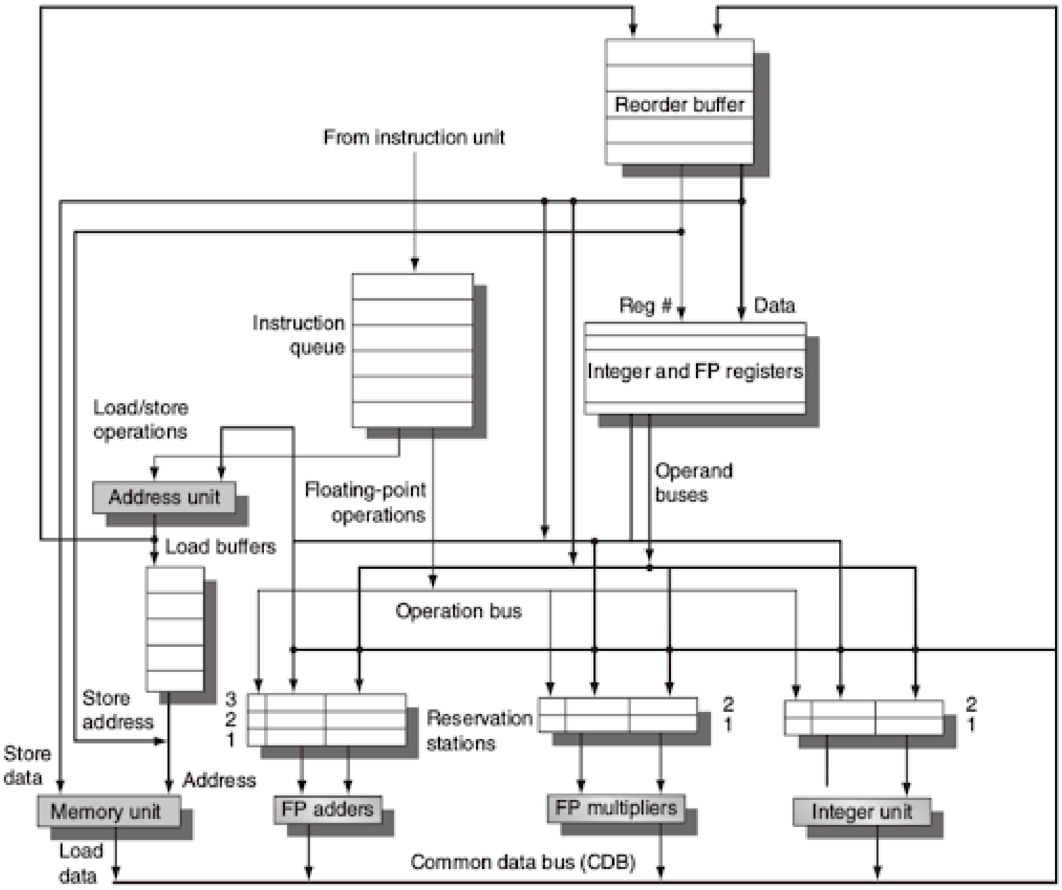
Implementing an ISA of this magnitude presented us with many different obstacles and opportunities to understand how large projects are tackled in the real world. We were able to not only face the technical challenges that came with trying to design such a comprehensive code base such as the RISC-V architecture, but we also had to deal with dividing up work between our group members as well as ensuring that everyone worked together and understood the different parts of the project. The objective of our final project is to design an out-of-order processor using the Tomasulo Algorithm, presenting a significant challenge. We begin by constructing a fundamental out-of-order processor, starting with core components such as Fetch Logic, Decoder, Reorder Buffer (ROB), Reservation Stations, Functional Units, Common Data Bus (CDB), Register File, and Load/Store queue. To optimize performance in terms of Instruction Per Cycle (IPC) and latency reduction, we are implementing several enhancements. These include converting the I-cache to a pipeline cache, replacing the branch-always-not-taken strategy with a G-share Branch Predictor, and integrating a prefetcher to retrieve instructions from memory in advance and store them in the L2-cache (4-ways, 16 sets).

Design Description

Overview Diagram



Tomasulo Algorithm

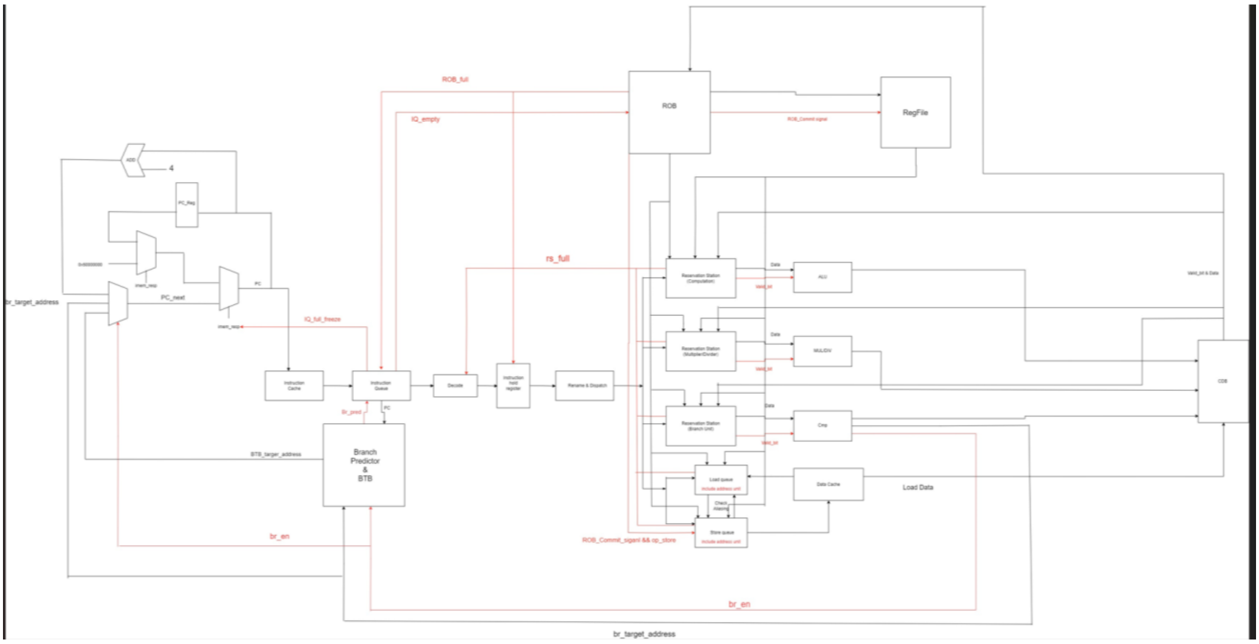


Milestones

Checkpoint 1

Description

For our checkpoint 1 , we were tasked with creating a block diagram to depict the fundamental Out-of-Order (OoO) datapath. This involved breaking down the datapath into two primary components: control logic and the datapath itself. To improve code readability and simplify the integration of advanced functionalities, we organized our datapath into distinct stages, including instruction fetch, instruction queue (FIFO), Decoder, Rename & Dispatch, Reservation Station, Register file, ROB, Functional Units (FU), and Common Data Bus (CDB). Additionally, we were responsible for defining the control word to align with our design specifications. Moreover, we were required to implement parameterizable queues for Reservation Station, Instruction Queue, Reorder Buffer, and Register File.



Checkpoint 2

Description

For checkpoint two, our primary focus is integrating the provided multiplier into our processor as a functional unit, which is a crucial requirement for this stage. Additionally, we are ensuring that our design supports out-of-order execution of arithmetic operations, necessitating the implementation of components such as Reorder Buffer (ROB), Reservation Stations (RS), and Common Data Bus (CDB). We are also validating the correctness of our implementation by running the dependency_test.s and ooo_test.s programs. Specifically, we are ensuring that instructions correctly commit, and writeback to ROB occurs out of order when executing ooo_test.s. These tasks collectively constitute our progress for checkpoint two.

Testing (No branch/Jump/Load & Store/Multiplier)

We tested our checkpoint 2 codebase extensively against the provided tests. These tests primarily focused on verifying the correct propagation of instructions and ensuring proper execution during the Rename/Dispatch phase. Specifically, we validated whether values were retrieved from the Register File and, if not found, were appropriately sourced from the Reorder Buffer based on the ROB tag. Additionally, we verified the successful renaming of the destination register (rds) to the corresponding ROB entry tag for each instruction. Furthermore, we conducted tests to assess the basic ALU operations and Multiplication. Having successfully passed these test cases, we gained confidence in the robustness of our design to progress to subsequent checkpoints.

Checkpoint 3

Description

In checkpoint3, we added a lot of elements, some of which already contain advanced feature functionality, in which we designed a load store queue, load and store are all stored in this queue, store is the oldest of the ROB and lsq must be satisfied at the same time. The store must satisfy both the ROB and the oldest instruction in the lsq before leaving the lsq, the store goes to the D-cache for operation only after the ROB commits, and the load goes to the D-cache for ROB commit only after the D-cache is finished. We also designed address forwarding so that the load can be first out of the lsq as long as it matches the address of the most recent store. We added the function of comparator as reservation station3, and the corresponding CDB3, and lsq as our reservation station4, and the corresponding CDB4. After integrating, we have 4 cdb, 4 rs, and finally, we integrated the cache. We divided the cache into i-cache and d-cache, and we designed an arbiter to prioritize the needs of different caches, and in the arbiter, the d-cache has the highest priority, in addition, we designed the adaptor to interact with the burst memory, and the value we get is then returned to the arbiter, and finally the arbiter gets the value back to the cache.

Testing

We first tested all the functions without instruction with load store branch jump in magic memory, then we tested all the functions without instruction with load store branch jump in burst memory, and then we test the load/store, branch/jump separately after integrating all the , finally ran through coremark, and then finally hooked up the cache and cpu and arbiter and adaptor all together, and tested all the functions together

Advanced Feature Design

1. G Share Branch Predictor

-Design :

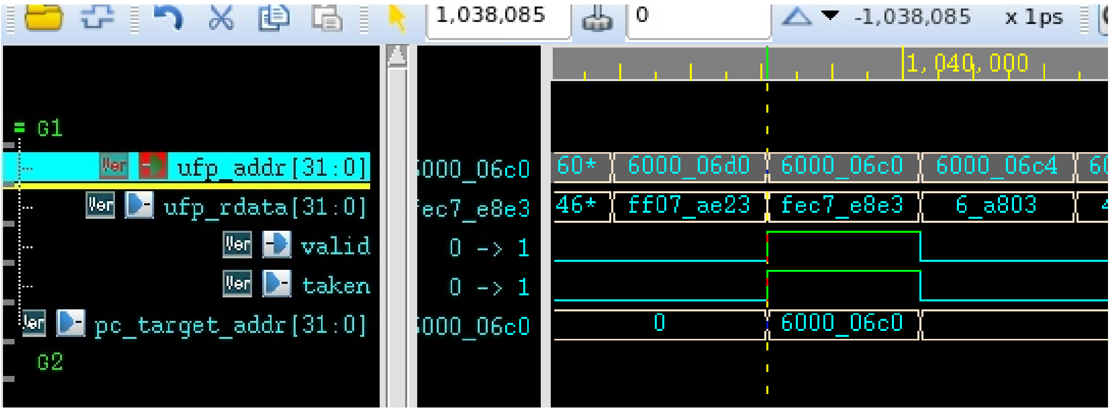
For the Branch Predictor, we decided to do a GShare branch predictor. While we send out imem_addr to I cache. At the same time the addr is sent into a module to obtain an index for Gshare. To do that, we grabbed 5bits of the addr **pc[6:2]** and **xor** it with our **branch history table.** By doing that, we get 5 bits of output, and that will point to the entry to our GShare.

We then read the stored 2bit counter in that entry to decide whether we want our branch to be taken or not taken. However, the problem is we don’t know if the instruction is even a branch. Therefore, the cycle we get back the data from I-Cache. We will have to go through a partial decode to determine its opcode and imm value. Opcode helps identify what type of instruction it is. Imm value helps calculate the branch target addr by doing PC + Imm.

After confirming its a banch, we send out a package to our fetch module that includes our branch decision (taken/not taken), a valid signal (proving it's a branch), and a branch target address. The fetch should correctly set the next imem_addr based on our decision.

This same exact package should be stored in a FIFO. This is because after the branch instruction get committed from ROB, we need to compare the br_en result with our branch decision in the first place. Another thing to check is to see if the branch target addr match each other. If any of these two has a mismatch, we result in a branch mis predict. There are two kinds of mis predict: first is predict not taken, but br_en is 1. This scenario we will jump to the branch_target addr that was sent out during commit. Second is predict taken, but br_en is 0. Under this scenario, we will have to restore the original PC from the package we stored in that FIFO. Both scenarios requires us to flush the system. Therefore, it is crucial to have a high prediction rate to reduce the cycles wasted due to flushing the wrong instructions.

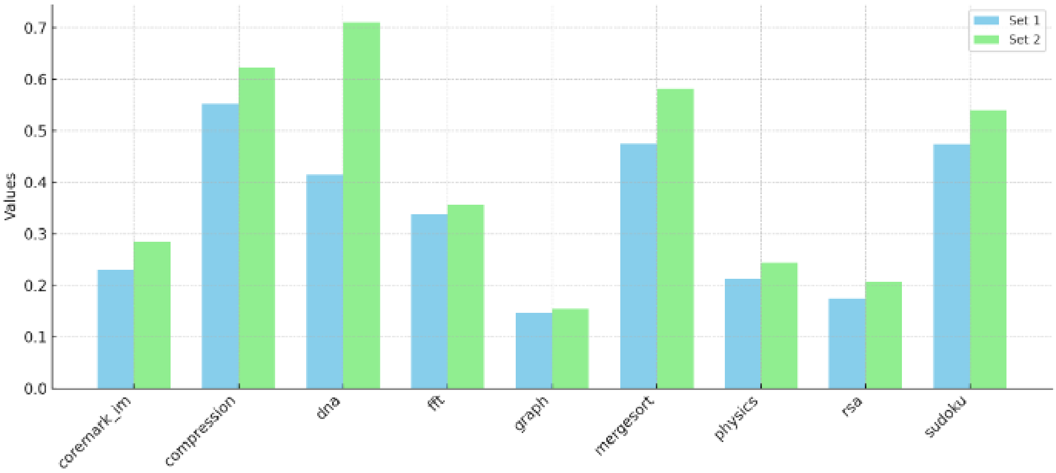
-Testing :



Here is an example of our branch prediction. At first we send out 6000_06d0 to our I-Cache. Next cycle, it returns an instruction fec7_e8e3. After decoding it, we found out its a branch, so valid is high. At the same time the GShare index is calculated and predicted taken. As you can see the branch target addr has also been calculated by doing PC + Imm. Within the same cycle, ufp_addr for I cache has changed to the same addr (6000_06c0) based on this information.

-Performance Analysis:

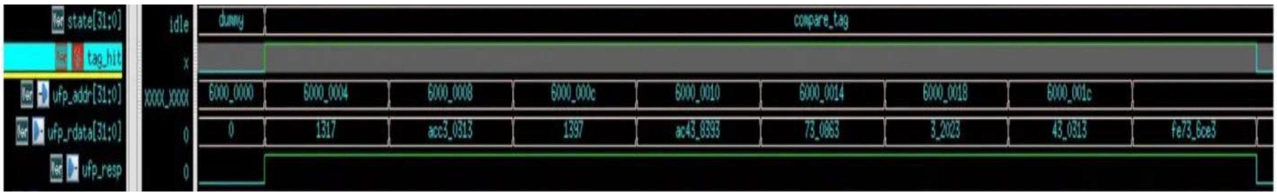
Branch Predictor is one of the best working advanced features performance wise for our project. By having a good branch predictor, we were able to increase our IPC by a lot. In the below graph compares the IPC for each program before and after. We were surprised by how much dna.elf increased in performance.



2. I-Cache (Pipeline Cache)

-Design :

During the compare-tag state, each tag hit maintains the cache in this state, allowing for immediate tag comparison in the subsequent cycle using the instruction memory address. This approach differs from the traditional method, where the cache returns to idle after a tag hit before proceeding with the next instruction comparison. By eliminating this idle time between comparisons, the overall processing time is halved. Consequently, idle time only occurs during the first instruction, as the cache consistently remains in the compare-tag state thereafter. This optimization significantly improves Instruction Per Cycle (IPC) performance while effectively reducing delays.

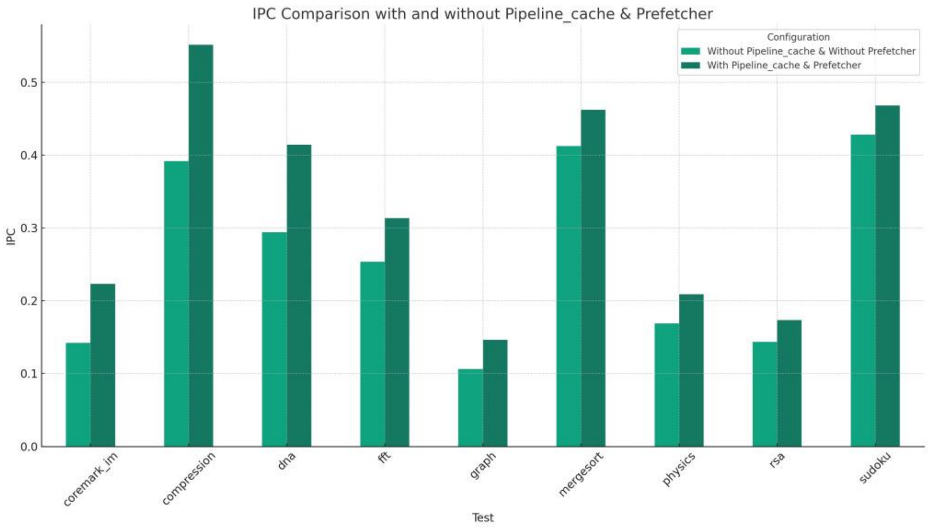


-Testing :

After integrating the pipeline-cache, significant effort was dedicated to rectifying the relationship between the PC and imem_resp in the fetch stage. This was imperative as we needed to dispatch an imem_address to the I-cache after each cycle. Initially, an error arose due to the CPU consistently receiving ufp_rdata one cycle late, resulting in a persistent shadow problem with the PC. Through meticulous investigation, we traced back to the fetch stage to uncover the root cause of the PC mismatch. Eventually, we realized that prior to implementing the pipeline cache, we had introduced a one-cycle delay in passing imem_resp to the fetch stage to resolve a combinational loop. However, with the introduction of the pipeline cache, this delay became unnecessary.

-Performance Analysis:

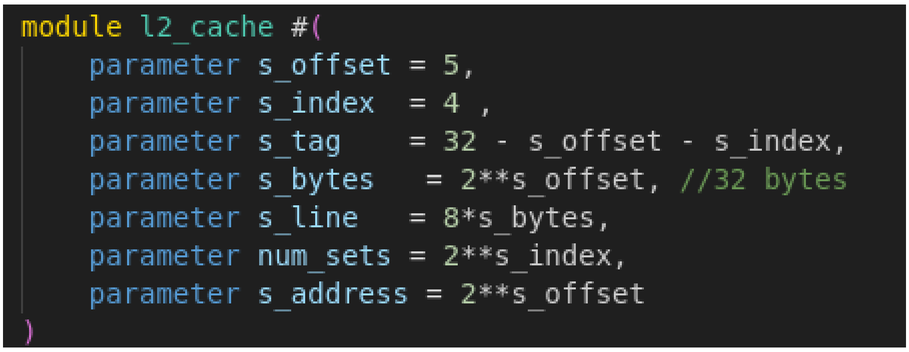
Our performance analysis shows that implementing both a pipeline cache and prefetcher significantly boosts IPC (Instructions Per Cycle). With the pipeline cache, the Cache remains in the compare tag state after each hit, accepting the Fetch stage's address for comparison in the next cycle. This adjustment notably increases IPC (coremark.elf) from a baseline of 0.14 to 0.22. This enhancement improves instruction processing efficiency by eliminating the need to return to the idle state in each cycle.



3. Fully Parametrized Cache(L2 Cache)

-Design :

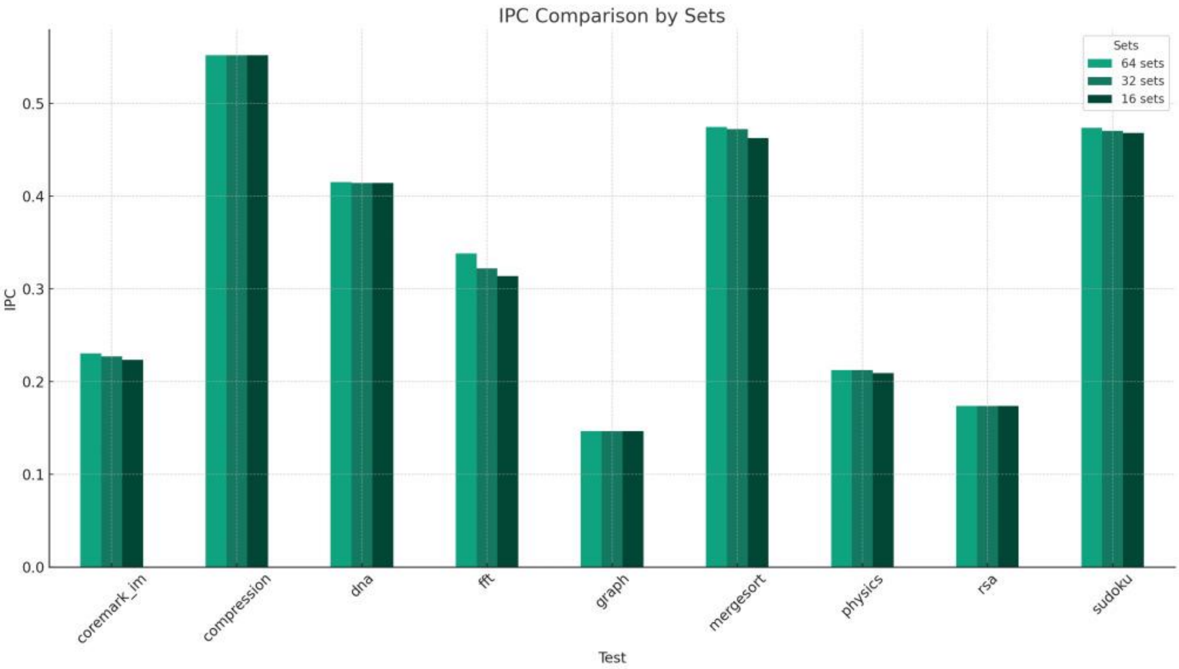
When initially implementing the L2 cache, we could have simply copied the mp_Cache module. However, considering the future need to reduce area, we opted to make our L2_cache module parameterizable. This decision enables easy adjustments to the number of sets as needed for area reduction purposes. Currently, our L2_cache utilizes 4-ways, 16 sets, and PLRU replacement policy. Initially, we used a 64-sets L2_Cache, which did improve IPC to some extent (when combined with a Prefetcher), but it consumed too much area. Therefore, we gradually reduced the sets from 64 to 32 and then to 16. Surprisingly, this reduction in sets did not significantly impact IPC, but it led to substantial area savings. Consequently, we concluded that 16 sets were optimal for our design.



```
module l2_cache #(
    parameter s_offset = 5,
    parameter s_index  = 4 ,
    parameter s_tag    = 32 - s_offset - s_index,
    parameter s_bytes   = 2**s_offset, //32 bytes
    parameter s_line   = 8*s_bytes,
    parameter num_sets = 2**s_index,
    parameter s_address = 2**s_offset
)
```

Performance Analysis :

IPC Comparison between (64 sets, 32 sets, 16 sets)



4. Load/Store Forwarding

-Design :

For our load store instructions, we decided to do a 16-entry unified load store collapsing queue. After each instruction is sent from the dispatch stage, if it's a load or store instruction, it will then be stored into the load store queue in order. It works just like a reservation station. It snoops the CDB every cycle to solve dependency issues. After we obtain values for all source registers, we are able to calculate our data memory address that we wish to access. We then determine what our top instruction is in our load store queue. If it is a load, and all dependencies are solved, we can then fire the instruction. However, if it is a store, we will need to wait until the first instruction in the ROB is also the same store instruction, then we fire it. This is because mis write a memory is very difficult to recover. What if there is a branch Infront of the store? We wrote it into memory, but later realized we had a misprediction for the branch, so the store shouldn't even be executed in the first place. Therefore, we wait until the next instruction committed is a store to send the write request to our D-Cache, to ensure nothing like we mentioned ever happens.

Our load store queue also supports simple memory forwarding. When a store is ready to be issued like mentioned above, we will take its writing address and see if there are loads that should be executed later than the store that plans on reading from the same address. If so, we will directly forward the write_data to the load and send it straight into the CDB. However, there is a rule we have to follow. After the top store is ready to fire, and we start checking the instructions from top to bottom one by one. If there is a different store before a matched address, we would still consider it a fail. Because the unexpected store could be writing to the exact same address with a different data. Therefore, we would be forwarding the wrong value.

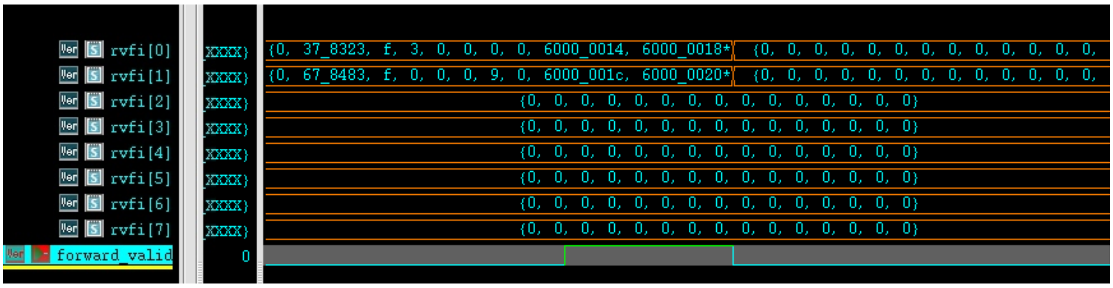
-Testing :

```
sb x3, 6(x15)

sub  x2, x1, x3

lb x9, 6(x15)
```

Here is a snippet of the program that I will be running. In the following instructions, a store is issued to an address calculated by adding 6 to register 15. Two instructions later, a load is issued to read from the exact address.



```
rvfi[0]        XXXX}  {0, 37_8323, f, 3, 0, 0, 0, 0, 6000_0014, 6000_0018*  {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
rvfi[1]        XXXX}  {0, 67_8483, f, 0, 0, 0, 9, 0, 6000_001c, 6000_0020*  {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
rvfi[2]        XXXX}                     {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
rvfi[3]        XXXX}                     {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
rvfi[4]        XXXX}                     {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
rvfi[5]        XXXX}                     {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
rvfi[6]        XXXX}                     {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
rvfi[7]        XXXX}                     {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
forward_valid      0
```

As expected, when both store and load addresses are ready, a forward signal is raised. The store will be sent to the memory unit and dequeued from the queue. The load will be directly forwarded to the CDB with the value and also dequeued from the queue. Causing the load store queue to be empty after they are being issued.

-Performance Analysis :

We did this for cp3, because we thought it would be very painful to change things up after having our basics load store branch done. So we decided to implement this directly. Therefore, we have no before and after comparisons for this advanced feature.

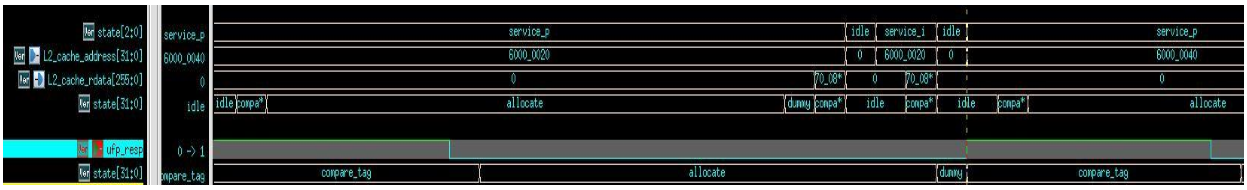
Another thing worth mentioning is that doing memory forwarding with collapsing queue have some cons. First, power consumption is a huge issue because we are constantly moving data from one entry to the other. Second, critical path is a huge bottle neck because collapsing queue tend to have a lot of edge cases where many if else if statements are needed.

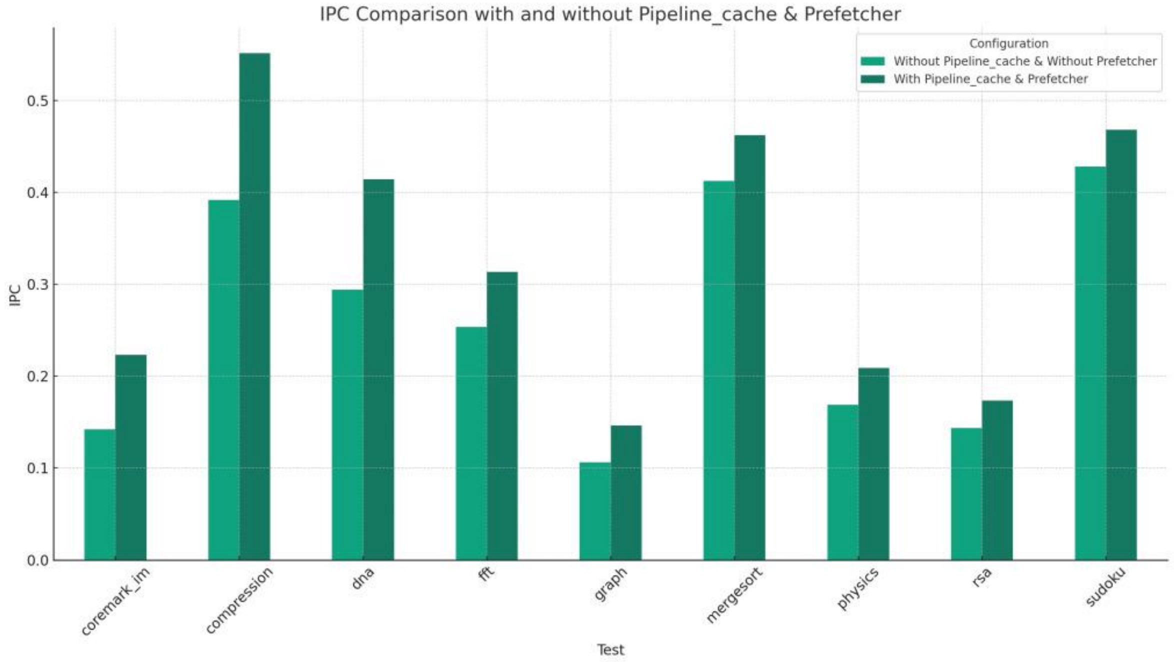
5. **Next-Line Prefetcher**

-Design :

We added an additional state, "service_p," to the arbiter. When the arbiter does not receive requests from the I_cache and D_cache, we initiate prefetching. The address we send for prefetching is calculated based on the I_cache's imem_address plus 32'h20 (the next cacheline). Since we only perform prefetching for the I_cache, when the arbiter is in the "service_i" state, we first compute the prefetch address. Finally, in the "service_p" state, we send out the prefetch address to access the L2_cache.

-Performance Analysis :





Performance Insights:

In benchmarks like **coremark_1m**, **compression**, **dna**, **mergesort**, **rsa**, and **sudoku**, the configuration with pipeline, cache, and prefetcher consistently shows higher IPC, indicating a substantial improvement in processing efficiency due to these enhancements.

For **fft**, **graph**, and **physics**, while the IPC improvement is visible, the increase is less pronounced compared to others, suggesting these benchmarks may be less sensitive to the enhancements or may be bottlenecked by other factors.

Overall Impact:

The enhancements significantly enhance the CPU's performance across most benchmarks, underscoring the importance of pipeline, cache, and prefetch technologies in improving IPC.

The configuration without these enhancements generally shows a lower IPC, highlighting the potential performance loss in more complex or demanding processing tasks.

This data clearly illustrates the benefits of integrating advanced architectural features like pipelining, caching, and prefetching into CPU designs, particularly in handling diverse and computationally intensive tasks.

Conclusion

In the COMPETITION, as of April 26, we secured a first-place finish in DNA with an impressive IPC of 0.78. We also achieved top six finishes in COMPRESSION, RSA, mergesort and SUDOKU. However, in benchmarks like graph and FFT, our IPC was on the lower side, approximately 0.2, positioning us in the lower middle tier of the upper rankings. Overall, we ranked 16th in the weighted average PD3A½. Our total area was 239,396 µm², and we achieved the highest maximum CPU frequency across the benchmarks at 454.44 MHz. Our power consumption ranged from 30-40mW. For future enhancements, we should focus on optimizing the multiplier, incorporating a prefetch structure into the D-cache, and adding registers to the critical path to increase our frequency.