ECE411
5 Stage Pipelined RISC-V CPU
Team: MP4_100%

1. Introduction:

In the rapidly evolving field of computer science, designing a high-performance CPU is an essential task. As part of our ongoing project, we developed a 5-stage pipelined CPU that could correctly execute RV32i ISA. Our pipelined CPU divides the processing of instructions into five stages (instruction fetch, decode, execute, memory operation and data write back), allowing for faster and more efficient processing of instructions. Our project aims to develop a pipeline CPU that is capable of handling a wide range of workloads and achieving faster processing times. This report provides an overview of our project, including the design and implementation of our pipelined CPU, the advanced features we have incorporated, and the testing strategies we have used to verify the functionality of our CPU.

2. Project Overview:

Our project involved designing and implementing a 5-stage pipeline CPU that improves the performance and efficiency of computing systems. The primary goal of our project was to create a pipeline CPU that can handle a wide range of workloads and achieve faster processing times. To achieve this goal, we had to consider several factors such as processor architecture, memory management, cache design, branch prediction, and testing strategies. We chose these goals based on the current industry standards and requirements for high-performance computing. With the increasing demand for faster and more efficient computing, a pipeline CPU that can handle a wide range of workloads is crucial. Moreover, our project aimed to incorporate advanced features such as cache systems and branch prediction to further improve the performance of our CPU. Organizational and administrative aspects of our project involved work sharing and project management. We divided the tasks based on the expertise of each team member. Jiamao Xu worked on the local branch history table, global 2-level branch history table, and tournament branch prediction. Jerry Wang worked on the L2+ cache system, basic and strided prefetching. Garen Hu worked on the parameterized cache. Each team member was responsible for implementing their part of the design and ensuring that it was integrated seamlessly into the overall CPU design.

3. Design Description:

(a) Overview:

In this project, we implemented several advanced features in a 5-stage pipeline CPU to enhance its performance and efficiency. One of the main features we added was a parameterized cache, which allowed us to adjust the cache size and associativity based on the workload requirements. This feature resulted in improved CPU performance, as it adapted to different data access patterns and workloads.

We also incorporated a level 2 (L2) cache, which acted as a buffer between the main memory and the level 1 (L1) cache, thereby reducing memory access times. Additionally, we implemented basic prefetching and PC-based stride prefetching to further optimize memory access and reduce cache misses.

To minimize the impact of branch instructions on the CPU's performance, we implemented two types of branch prediction strategies: local and global. The local branch history table (LBHT) prediction used the branch history of the current program execution to predict the outcome of a branch. The global two-level branch history table (BHT) prediction, on the other hand, stored the history of all branches in the program and used this information to predict the outcome of a branch. We also incorporated a tournament branch predictor that combined the LBHT and BHT strategies to achieve higher accuracy in predicting branch outcomes.

Implementing these advanced features required a coordinated effort from the team members. We assigned specific tasks to each team member based on their skills and experience. We also had regular team meetings to track progress, discuss any issues or challenges, and ensure that everyone was on the same page. Notable achievements include the successful implementation and testing of all features, which significantly improved the performance and efficiency of the CPU.

(b) Milestones:

i. Checkpoint 1:

We created a basic pipeline capable of handling all RV32I instructions except FENCE*, ECALL, EBREAK, and CSRR. We used a dual-port "magic" memory that always sets mem_resp high immediately to avoid cache misses or memory stalls. We also provided paper designs for data forwarding, hazard detection, and an arbiter to interface the instruction and data cache with main memory.

ii. Checkpoint 2:

At checkpoint 2, we added hazard detection and forwarding, as well as static-not-taken branch prediction for all control hazards. We avoided stalling or forwarding for dependencies on register x0 or when an instruction does not use one of the source registers. We also implemented an arbiter and integrated it so that both split caches (I-Cache and D-Cache) connect to the arbiter, which interfaces with memory. The arbiter

determines the priority for cache requests when both caches miss and need to access memory on the same cycle. For RVFI, we made sure it's working at this checkpoint.

iii. Checkpoint 3:

Our cache module has been upgraded to a two-level structure consisting of L1 and L2 caches, with up to 8-way set-associative systems for better caching performance. It is also parameterized, allowing for easy adjustment of cache size and associativity. A branch prediction module has also been added, using both LBHT and global BHT strategies to minimize the time penalty associated with branching. These advanced features enhance the CPU's performance by adapting to varying workloads and data access patterns.

(c) Advanced design analysis:

i. Tournament Branch Prediction:

A. Design:

We first construct a local pattern history table with 32 entries. This table stores the recent branch result from the CMP module. Here we have the simplest branch prediction unit. Next, we construct a global branch history table and a branch target buffer. The branch target buffer will store the PC address with its branch target address. Here we have a global 2-level branch predictor. Finally, we build a counter to select between local branch predictor and global 2-level branch predictor. This is the tournament branch prediction we have.

B. Testing:

Since the tournament predictor is composed of local and global predictors. We first test them separately and make sure their accuracy is above 80% by tuning the parameter for the predictor. Then we combine them together and test them using test code.

C. Performance analysis:

	Local Branch History Table	Global Branch History Table	Tournament branch predictor
Branch count	6364	6364	6364
Branch misprediction count	529	906	528
Accuracy	91.69%	85.76%	93.18%

ii. Parameterized cache:

A. Design:

To create a parameterized N-way N-set associative cache, we use a generate block to create N instances of the data, tag, valid, dirty, and lru arrays. This will allow for easy configuration of the cache's associativity. We make use of the provided cache and treat the set associative cache like a direct-mapped cache, focusing on the set being accessed. On a read/write operation, if a cache is hit, we will perform the operation on the cache way that found the hit. On cache miss we will perform the operation on the first invalid cache way for a given set if the cache is not full. If the cache is full, we will evict the least recently used (LRU) block if dirty and place the new cache line in the same cache way. In order to parameterize our LRU logic, we choose to implement a real LRU logic for its simplicity in design. We create N arrays with log2(N) bits each, corresponding to a cache way, and rank them from 0 to N-1 based on their recent usage. On a cache miss, if there is a need for eviction, we find the cache way with highest rank and replace it with the new cache block, and increment the ranks of other cache ways in the set by 1, and reset the evicted cache block's rank to 0. In order to parameterize the set, we modify the arrays so that in can take in parameterized indexes and we modify the indexing bits for the set to be mem_address[(5+log2(Sets) - 1:5] and tag to be mem_address[31:(6+log2(Sets) - 1)].

B. Testing:

We first ensure that the cache can handle changes in the number of cache ways and sets by running the checkpoint 3 test code and monitoring the data being loaded from or stored in the cache. After that, we test the parameterized LRU by checking the updates of our lru array in order to verify that the correct cache line is evicted and updated on a write back.

C. **Performance analysis:**

With more ways and sets, the cycles that are needed to perform read and write with the physical memory decrease significantly and the overall delay to execute checkpoint 3 test code also decreases significantly.

Without Prefetch ONLY L1	2 way 8 sets	2 way 16 sets	4 way 8 sets	8 way 8 sets	8 way 16 sets
Pmem Read cycles count	80630	57935	59395	23005	14235
Pmem Write cycles count	50975	36385	38445	7245	0
Time (ns)	1 971 525	1 573 855	1 611 675	899 955	731 765

iii. L2 cache:

A. **Design:**

In order to implement a L2 cache system for our processor. We reuse our L1 cache datapath and control while taking out the bus_adaptor as the L2 cache will be taking in 256 bits of data instead of 32 bits for L1. Then we modify the input ports to be 256 bits and place our L2 cache between the arbiter and the cache line adaptor for the physical memory.

B. **Testing:**

We first ensure that the cache can correctly handle memory operations by running the checkpoint 3 test code and monitoring the data being loaded from or stored in the cache. We also implemented performance testing in order to measure the performance of our L2 cache. We validate the cache hit rate and access latency both with and without the L2 cache system to see any improvements, and also to ensure that the cache can handle higher loads and corner cases.

C. **Performance analysis:**

With the addition of L2 cache, the cycles that are needed to perform read and write with the physical memory decrease significantly and the overall delay to execute checkpoint 3 test code also decreases significantly.

Without Prefetch	L1(2 way 8 set)	L1(2 way 8 set) and L2(4 way 8 set)
Pmem Read cycles count	80630	46180
Pmem Write cycles count	50975	37235
Time (ns)	1 971 525	1 488 285

iiii. Prefetcher:

A. Design:

We implemented a one-block-lookahead hardware prefetcher for our i cache as well as a pc-based-strided hardware prefetcher for our d cache. The reason is that we noticed that for most of the times when there are no branch or jump instructions, the next instruction address is always pc + 4. While the data address varies based on the instruction data. Therefore, with the hardware prefetchers, we could prefetch the data that might be needed for the next load instruction from the physical memory and place them in our cache system so that it could reduce the memory access stall time. For one-block-lookahead hardware prefetchers, we always initiate a prefetch for the next instruction line (pc + 4) whenever the current pc is fetched and accessed and results in a cache miss. If the next pc data is already cached, no memory access is initiated. For PC based strided prefetching. We record the distance between the memory addresses referenced by a load instruction (stride of the load) in an array with parameterizable entries and we also keep track of the last address referenced by the load.When the same load instruction is fetched, we initiate a prefetch for the last address + stride.

B. Testing:

We first ensure that the cpu can correctly execute all the instructions by running the checkpoint 3 test code and monitoring both instruction and data address that are being read from the memory and the data being committed after an instruction completes. After that, we implemented performance testing in order to measure the performance of our prefetchers. We validate the delay to execute the checkpoint3 test code decreases after the addition of our prefetchers.

C. Performance analysis:

With the addition of both prefetchers, the cycles that are needed to perform read and write with the physical memory increase as more memory requests are being made while the overall delay to execute checkpoint 3 test code decreases.

L1 (2 way 8 set) L2 (4 way 8 set)	Without Prefetch	OBL (pc + 4)	With PC_based stride Prefetch	OBL for i cache Plus PC_based stride Prefetch for d cache
Pmem Read cycles count	46180	46295	46155	46270
Pmem Write cycles count	37235	37235	37210	37210
Time (ns)	1 488 285	1 482 485	1 487 785	1 481 985

4. Conclusion

In conclusion, our 5-stage pipeline CPU project involved implementing various advanced features to optimize the performance and functionality of the CPU. Our team successfully incorporated a parameterized cache, L2 cache, basic prefetch, PC-based stride prefetch, local and global branch prediction, and tournament branch predictor to enhance the CPU's capabilities. Through rigorous testing and evaluation, we were able to verify the effectiveness of each feature and identify areas for improvement. Our team worked collaboratively and effectively, with each member taking on specific tasks and responsibilities to ensure the project's success. Overall, our CPU's advanced features improve its performance by reducing the time penalty associated with cache misses and branch operations, while also ensuring flexibility in handling different workloads. The incorporation of a 5-stage pipeline with hazards and forwarding also enables efficient instruction execution, further enhancing the CPU's overall performance. As a team, we are proud of our achievements and the successful implementation of these advanced features. Our project has provided valuable insights into CPU architecture and design, and we believe that our work will contribute to the development of even more advanced and efficient CPUs in the future.