Final Project Report

Johan Ortega-Rios, Geitanksha Tandon, Quinn Doud

always_comb @(posedge what)

1 General comments

Your project report should be written in a reasonably formal format by adhering to paragraphs, punctuation, proper grammar, expressions, and possibly figures. In short, assume that you are writing a report that is to be read by a group of people that you don't know, but that you would like to impress. Your objective is to write a clear and informative report so that whoever reads it understands exactly the points you want to make. At the same time, the style and format of your report should be such that it leaves a positive impression on the reader.

1. Introduction
2. Project overview
3. Design description
    (a) Overview
    (b) Milestones
        i. Checkpoint 1
        ii. Checkpoint 2
        iii. Checkpoint 3
    (c) Advanced design options
        i. Option 1 (e.g., branch prediction)
            A. Design
            B. Testing
            C. Performance analysis
        ii. Option 2 ...
4. Additional observations (if any)
5. Conclusion

2.1 Introduction

Present the main technical thrust of the project and a very brief organization of the report. Typically, the introduction is broad and also gives a motivational explanation about the importance/impact of the work reported. In our case, you may want to outline the architectural level description of your work and describe the importance of completing this design as part of learning about computer architecture. Assume that you address an audience that is familiar with the broad subject but not with your specific work.

1. *Introduction:*
   Our group designed a pipelined RISC-V Processor which was split into 5 different stages. We had to design the processor from its very base, implementing its datapath and having a Control ROM Capable of decoding its instructions and controlling the datapath. This would help us understand the actual working of a RISC Processor and further our inherent understanding of Computer Architecture, developing skills in the context of knowing the mechanisms to aid us in out future endeavors. It would give us broad knowledge about the subject but also help us understand the tradeoffs between designs which will help us be good computer programmers and architects in the future.

2.2 Project overview

Describe your project and provide insight into what was involved. Give the goals of the project and talk about what influenced the choice of goals. Use this section to describe the organizational and administrative aspects of your project, such as work sharing, project management complications, or notable achievements.

2. *Project Overview:*
   The project had us work on the processor over a span of 4 weeks, and broke down the work we needed to complete by each checkpoint. We mainly worked together for the first two checkpoints, trying to divide the work equally but splitting the main task into subsections. We would always have one person keeping track of the deadlines and start the project documents in order to optimize time. We struggled a little to complete all tasks exactly on time but overall made sure everything was submitted before the deadline.

3. *Design Description:*

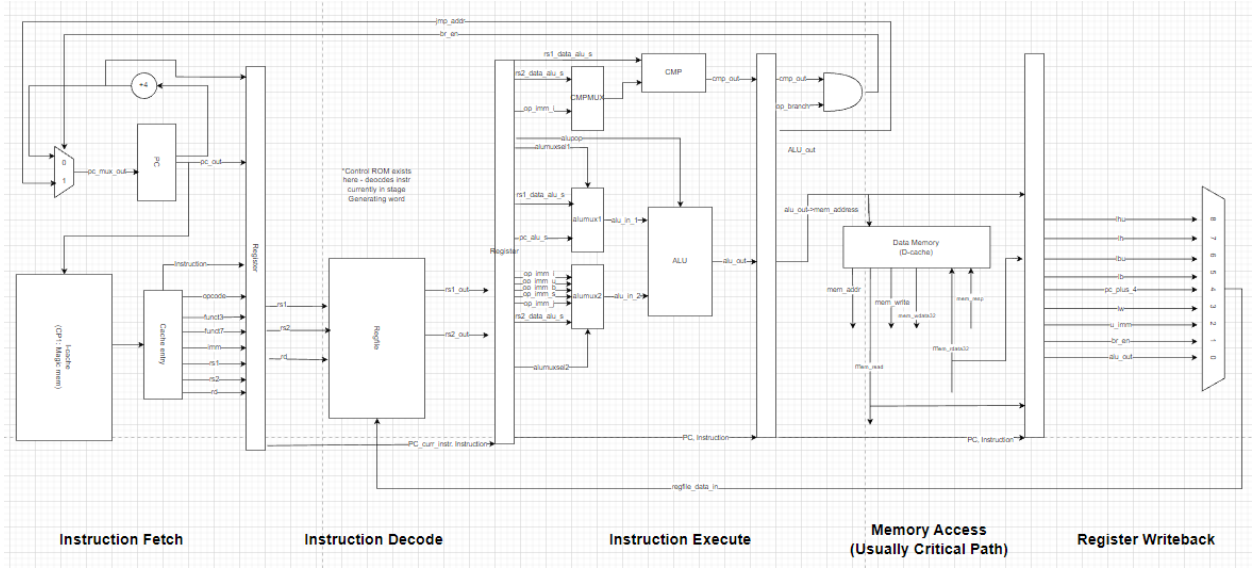
   (a) Overview:
   We designed the baseline of the processor first implementing 5 stages, and then slowly worked our way up toward making it pipelined and capable of handling multiple instructions, then handling the cases of hazards, solving it via the usage of forwarding and stalling. We implemented Forwarding to solve Data Hazards, used a separate instruction and data memory to solve Structural Hazards, and had a stalling mechanism for Control Hazards, and when caches faced a miss and needed to wait for the main memory.

2.3 Milestones

Organize your progress reports for each of the checkpoints in chronological order to demonstrate the evolution of your processor. For each checkpoint, provide an overview of your design and present detailed descriptions of all aspects of your design so far. Include paper designs, diagrams, schematics, or charts where appropriate. Include how you tested different components and verified that they work according to specification. Transition to each checkpoint within this section to demonstrate the progress of your MP.

(b) Milestones:

- Checkpoint 1: For this checkpoint, we only had to convert our previous processor designs into a multi-stage pipelined design. This involved separating the original design into 5 stages, namely: Fetch, Decode, Execute, Memory, and Writeback which each completed different steps of the instruction process.
- Fetch performed logic relating to grabbing new memory data and updating the PC value
- Decode grabbed the relevant Regfile memory data
- Execute performed any numerical operations (add, cmp, etc).
- Memory wrote to or loaded data from physical memory.
- And Writeback returned data to the Regfile.
- Since the code for this was essentially equivalent to debugging our original design with only a few modifications, our debugging process was relatively simple as well. So we ran test code and checked Verdi to find out what was going wrong. Most of our errors wound up being small bugs which came from typos or incorrect signal names. Additionally, once we were able to pass the basic provided test code, we wrote our own to ensure that it properly handled every supported instruction.



- Checkpoint 2: For this checkpoint, we worked on addressing hazards as well as integrating more realistic memory with single-cycle caches instead of the magic memory that was used in checkpoint 1. This meant changing many small things that our original checkpoint 1 design relied on– for example, stalling on a memory write only took 1 cycle in the previous design, whereas now we had to be able to stall indefinitely.

- Checkpoint 3: For this checkpoint, we worked on our advanced design features: Cache, Global BHT, Eviction Write Buffer, L2 Cache, and Simple Hardware Prefetcher. We additionally fixed bugs present in our CPU that were not revealed in the CP3, 4 test code.

- Checkpoint 4: For this checkpoint, we worked on integrating all of our implemented features into our working baseline CPU. As well as fixing issues in our baseline so that we could run the entirety of coremark with matching spike traces.

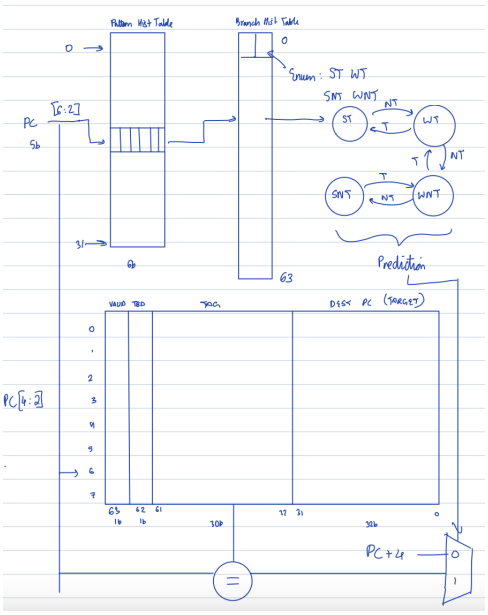
2.4 Advanced Design Features

For each advanced design option, describe design trade-offs and provide some performance analysis on the performance impact of the option. Ideally, note the speedup (or slowdown!) of adding each advanced feature to your processor. Describe how you implemented these features. Comment on the theoretical workloads that the feature would help with and some workloads that it may handle poorly.

(c) Advanced Design Features:

i. Local Branch History Table

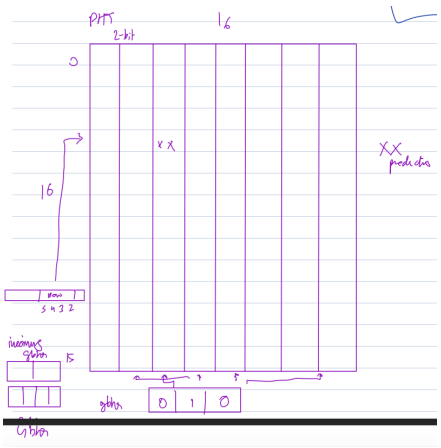
A. Design: The Local BHT had a pattern history table recording the previous 6 histories of that specific branch. Those histories indexed into a 2^6 Branch History table that stored 2 values of state and updated it with a Finite State Machine depending on whether the branch was taken or not in the execute stage. For the fetch state, it simply returned a prediction based on the current bits, and in the execute stage updated the FSM as well as sent whether the original prediction was a misprediction or not.



B. Testing: The code was first tested as a DUT, where it was initialized manually and the FSM checking was done to ensure that it was correctly updated. Then, it was integrated into the pipeline and provided a branch prediction scheme and was checked against the provided TA Code. It matched and so was used for prediction.

C. Performance analysis: Alone, the BHT did not produce much benefit as the address calculation for the jumping address was only provided in the execute stage. However, coupled with the Branch Target Buffer, it proved to have an improvement in performance. In terms of accuracy, it was very accurate, with over 80% accuracy in most codes and above 70% accuracy in all codes.

ii. Global BHT:



A. Design: This Branch History Table actually only used a Global Branch History Register that was 3 wide (tried with 4 as well, not useful) and updated the register based on the branch of any branch instruction in the entire code. Based on that, the GBHR checked the specific “way” of the prediction history table to find the 2 state bits it would use for prediction. It used some bits of the PC to index into the PHT. Similar to the BHT, it had an FSM that was updated in the Execute Stage.

B. Testing: Just like the BHT, it was tested as a DUT and we modified the Testbench to accommodate for the PHT entries now, but tested one way and 4 indexes, after which the algorithm would remain the same. Then, we directly tested it on the codebase and it had correct results when coupled with the BTB.

C. Performance Analysis: It also does not help without a BTB to assist it, but accuracy wise it had 80% accuracy in 2 code bases and over 70% in the rest.

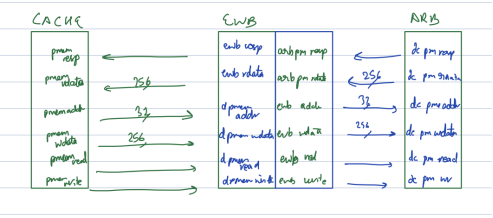
iii. Branch Target Buffer:

A. Design: The BTB simply stores the target address of any branches seen prior, and saves the 30 bits of the PC as tag to compare to current PC that will help us detect where to jump to in case the same branch was seen before. While a traditional BTB is usually Fully Associative, this one was a DM cache with index of 8 and tag of 30 bits of current pc, storing valid and Target of the PC to jump to.

B. Testing: The BTB was tested using older testbenches as a DM cache, and it matched all the requirements. Hence, it was simply integrated with the BHT schemes one at a time.

C. Performance Analysis: The BTB was too small to provide significant amount of performance benefits but helped cut off 2 cycles for some of the codes and improved performance slightly.

iv. Eviction Write Buffer:



A. Design:
The EWB was simply inserted between the arbiter and the Data Cache. It works by simply storing the evictions of the cache above it and working on sending it to main memory whilst the main cache can continue servicing other requests.

B. Testing: For the EWB, creating a DUT was unnecessary as it worked with the code that was provided already, which would not have worked in case it was incorrect. Hence, due to its inherent simplicity, extensive testing was unnecessary.

C. Performance Analysis: The EWB was unsynthesizable as it did not cooperate with the given cache but it helped improve performance as the cache did not have to wait to write to main memory, it simply wrote to the less latency oriented EWB and continued, reducing time of stalls.

v. L2 Cache

A. Design
The L2 Cache follows the same principles as the L1 Cache. Implementation-wise, it is the exact same module as the L2 Cache with indexing modifications made in order to facilitate 64 direct mapped cache lines. There were also additional modifications made in order to account for the writeback of dirty cache lines

from the L1 cache.

Because all cache line modifications would only be done in the L1 cache, our L2 cache does not need to support the same write granularity as the L1 cache. As such, the L2 cache's write mask was fixed so that all bits were always high. This is because the L1 cache will always be writing the entire cache line back to the L2 cache. So support for writing to specific segments of bytes was not necessary. The line adapter was also removed due to this.

B. Testing

For testing, we verified that the correct values were being written back and stored into the L2 cache after writebacks. Additionally, we verified that the spike logs for all the checkpoints/competition code matched with the matching golden trace.

C. Performance Analysis

In order to analyze the performance, we used a performance counter that would record and increment the number of "dead cycles" that the L1 spent waiting for data from the L2-cache. Our results were as follows:

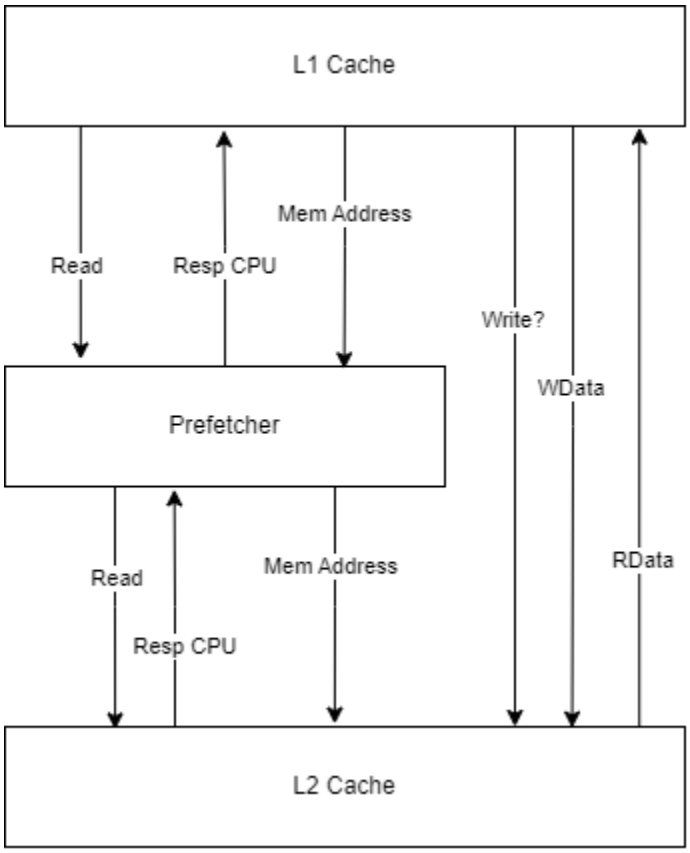
	CP2 Design	With L2 Cache
CP1	676	712
CP2	534	558
CP3	359702	12698
Comp1	23732	2708
Comp2	235532	2275
Comp3	119309	12911

With the addition of the L2 cache, we saw incredible performance improvements compared to the baseline CP2 design. There was such a decrease in idle cycles, that the L1 cache had to wait less than **1% of the cycles it previously did** (0.9658% in Comp2). However, it is important to note that there was actually a slowdown with the CP1 and CP2 test code. This is because of the additional overhead that the L2 Cache requires, and the length of the test code. Because CP1 and CP2 are short programs, instruction-wise, we simply don't read enough from physical memory to see any benefit from the L2 cache. Additionally, there is not a significant

amount of data loads in the test code that would benefit from the L2 cache.

vi. Simple Hardware Prefetcher

A. Design

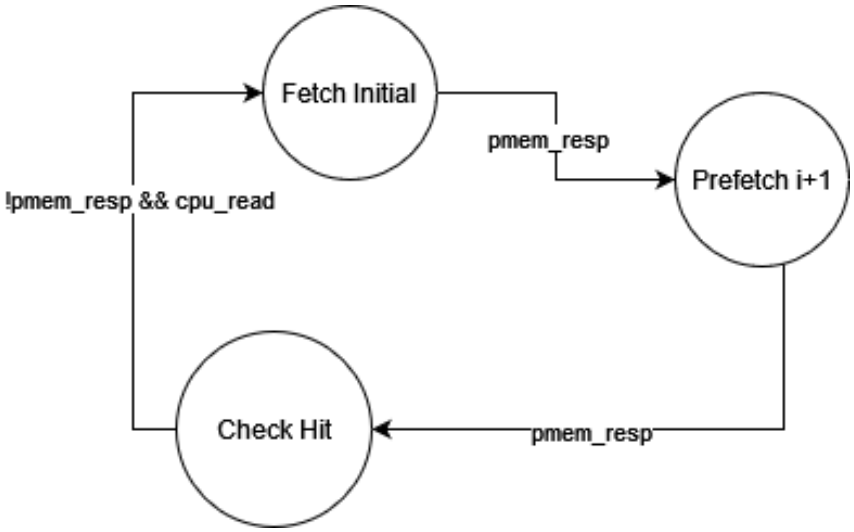


The simple prefetcher functions by prefetching the i+1th cache line in the case of a cache miss. The prefetcher was designed and implemented such that it would act as a middle-person when attempting to read from physical memory (the L2 Cache here).

The reason that it is between the L1 and L2 Caches and not anywhere else came down to convenience, and ease of implementation.

If the prefetcher were placed between the arbiter and physical memory, there would have to be much more signal manipulation in order for the line to be properly prefetched. Additionally, if it were placed between the cache and the arbiter, then a line buffer would have to be implemented to hold the previously fetched cache line. There were difficulties in implementing this, so we opted for the design that would require less re-inventing of our already present modules (cache).

Additionally, implementing with the L2 cache allows us to avoid having data processing logic in the prefetcher (cache line buffer). The Prefetcher functioned according to the following FSM:



While in the “Check Hit” stage, it would pass all the control signals provided by the L1 cache through to the L2 cache. Upon detecting a miss, it enters the fetch initial stage.

In the fetch initial state, the L1 cache’s control signals still pass through, but the prefetcher will begin as soon as the L2 cache is ready to receive another request.

In the Prefetch state, the Prefetcher is now providing its own control signals to the L2 cache, and not allowing L1’s to pass through. Upon receiving a mem resp from the L2 cache, the prefetcher returns to the check hit stage and allows the L1 cache’s signals to pass through once again.

B. Testing
For testing, we verified that our spike logs matched those of the golden trace. Additionally, we verified through Verdi that the states were transitioning as they were supposed to. As well as cross-checking with the disassembled code to verify that the correct data was being prefetched into the L2 cache.

C. Performance Analysis
In order to analyze the performance, we used a performance counter that would record and increment the number of "dead cycles" that the L1 spent waiting for data from the L2-cache. Our results were as follows:

	CP2 Design	With L2 Cache + Read Prefetching
CP1	676	683
CP2	534	483
CP3	359702	10973
Comp1	23732	2576
Comp2	235532	1804
Comp3	119309	5532

By using the prefetcher in tandem with the L2 Cache, we were able to reduce the number of idle cycles in our L1 Cache up to **4%** of what they were previously (In the case of Comp3). We saw a huge performance improvement with this, particularly when running programs that had high spatial locality – as in Comp3.

2.5 Additional observations

Make observations about aspects of the project that do not fit in any of the above sections. This section is optional.

2.6 Conclusion

Succinctly give a summary of your design objectives and achievements and summarize any novel or interesting aspects of the work. Give any punchline conclusions if applicable.

4. Conclusion:

Overall our design objectives were focused on improving the latency of memory, since the physical memory was by far the slowest part of the original design. Therefore it was natural for our advanced design options to reduce that delay. Upon implementing memory design options, we saw incredible performance improvements compared to our baseline.