Final Report

SnapDragon 411: Daniel Marks, Jason Liu, and Jincheng Liu

1. Introduction

The purpose of this project was to optimize a standard RISC V processor to increase the overall performance of benchmarking programs. We approached this design by first implementing a five-stage pipelined CPU, then integrating advanced design choices like complex caching and branch prediction to reduce runtime inefficiencies. This project gave us a deeper understanding of computer architecture by allowing us to engage with the design process and understand tradeoffs between design choices while challenging us to evaluate our execution through the implementation and analysis of performance metrics.

2. Project Overview

This project was largely successful in terms of the goals we set out to achieve. We were able to implement a fully pipelined five-stage CPU that could perform all of the standard RISC V ISA from which we built out the remainder of the features. After implementing a forwarding unit to prevent and exploit data hazards and a flushing unit to protect from incorrect branch predictions, we linked a 2-way set associative cache for both data and instruction memory operations. Immediately we saw two strong weaknesses in the processor's branch prediction and caching mechanisms.

Branch predictions were especially costly as they both caused flushing to stall upon improper prediction as well as stalling to calculate the target PC value after correct predictions. To create a very strong branch predictor, we opted to combine the benefits of a local branch history table for localized loops as well as a global predictor for the detection of larger patterns within a program. On the other hand, we found that memory operations were very costly as well as frequent with a 2-way caching scheme. We chose to implement a pipelined 4-way set associative cache to reduce the number of read and write operations, which also improved the fmax of our system by implementing BRAM modules.

For the first checkpoint, our group worked collectively to pipeline our original MP2 RISC V processor. From this baseline, we were able to divide up work on each new feature among the group members. For checkpoint 2, Jason worked on the instruction and data caches, while Jincheng designed and implemented the forwarding unit. Daniel then worked on debugging these two features, as well as implementing the flushing unit for correcting branch predictions. After completing these features, we moved on to the advanced features of our design. For checkpoint 3, Jason designed the pipelined cache, Jincheng designed the victim cache and L2 cache, and Daniel designed the tournament branch predictor. During the implementation of

these designs, Daniel worked on debugging the tournament branch predictor and pipelined cache, while Jason debugged the victim cache with Jincheng.

3. Design description

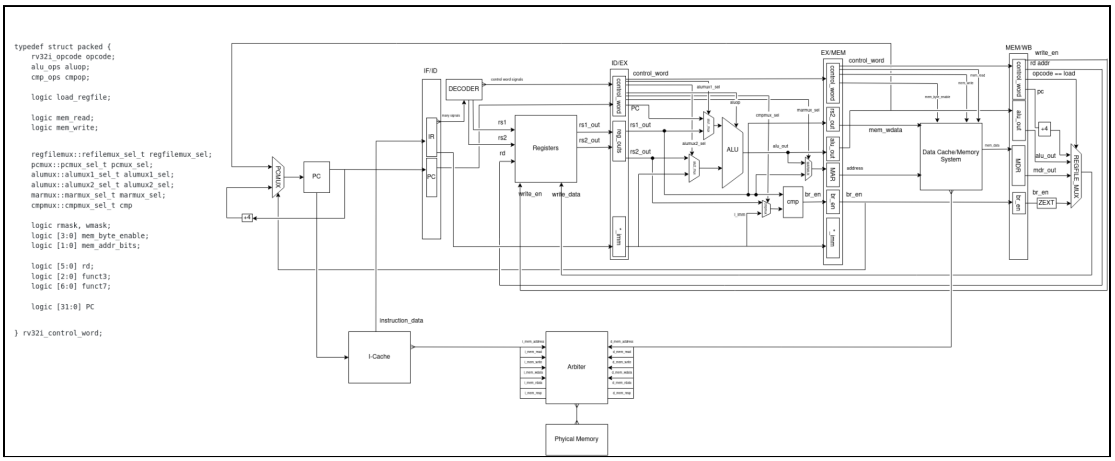
(a) Overview:

I. The design of our project was based on a five-stage pipelined RISC V processor with a forwarding unit and flushing unit for protection against control and data hazards. The advanced features implemented were a tournament branch predictor, pipelined instruction cache and 4-way data cache, an L2 cache, and a victim cache.

(b) Milestones:

I. Checkpoint 1:

We designed basic 5-stages pipelined datapath. We divided the execution of instruction into 5 stages and implemented a data struct called a control_word which would be decoded from the IR output in the control_rom to pass signals from stage to stage. In each stage, useful data from the previous stage is registered (i.e. rs2_out in the execution stage for mem_wdata in the memory stage). Additionally, any useful control signals for each stage were extracted from that stage's control word. We verified that this design worked for non-hazardous programs by running the mp4-cp1.s code.

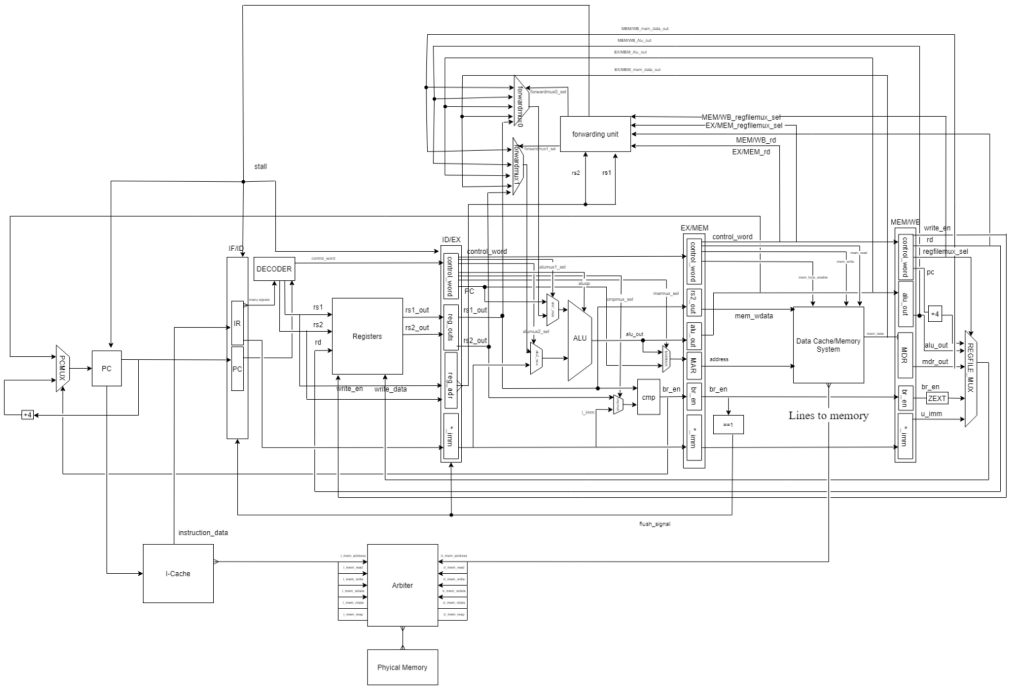


The original design for the non-hazard protected pipelined processor

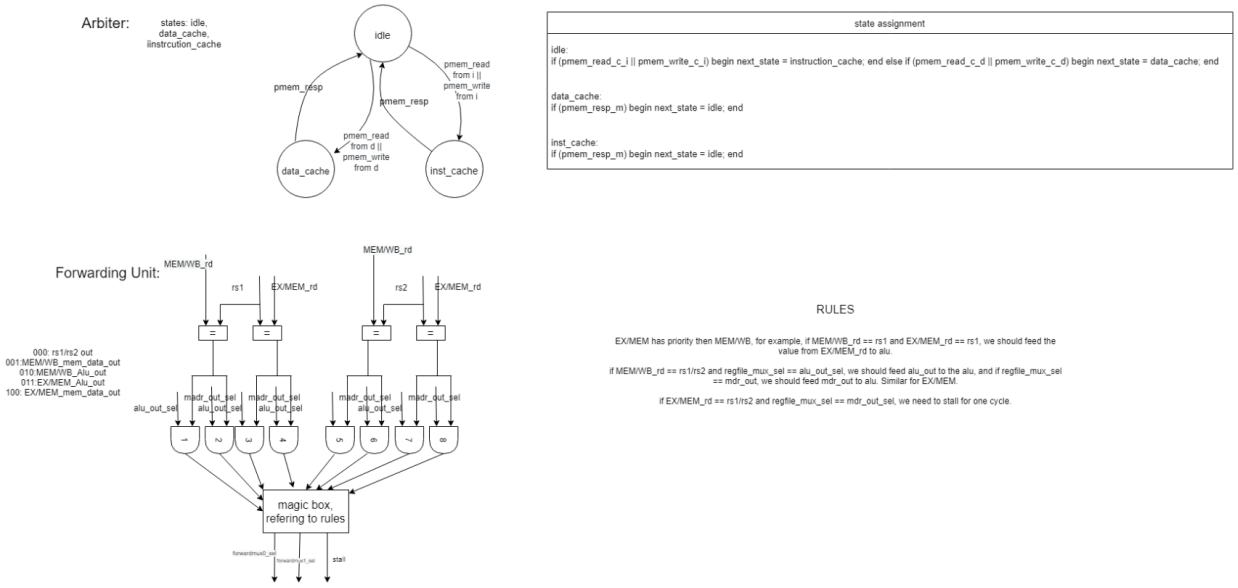
II. Checkpoint 2:

We designed a forwarding unit that passes data from the memory and writes back stages to rs1_out or rs2_out in the case of data hazard. Our forwarding module consisted mostly of some dependency checking signals as well as several muxes to forward the appropriate signal to the ALU and CMP modules.

We divided L1 cache into a data cache and an instruction cache because the memory address space for the instructions and memory data rarely overlap, and we needed to be able to perform a memory operation at the same time as another instruction fetch. We used an arbiter in between the two caches and the main memory to determine which one should be connected to the main memory since it is one main memory port versus two cache ports.



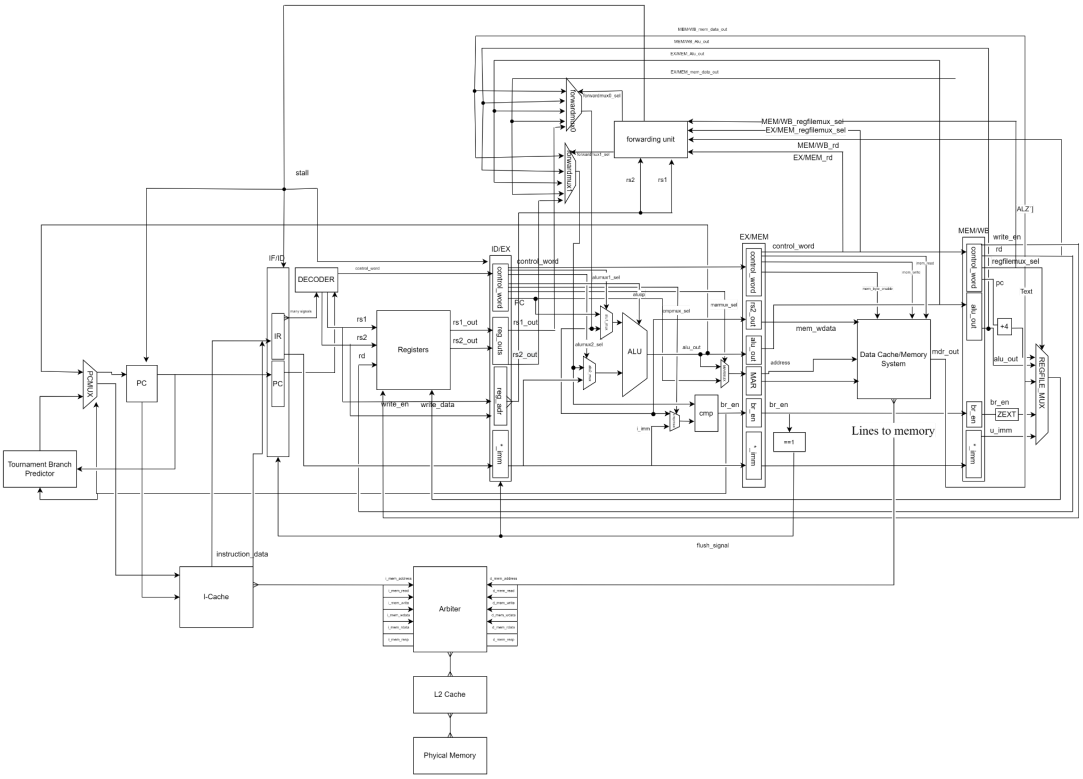
The pipelined CPU datapath with forwarding unit and caching mechanisms.



The state machine for the memory arbiter (top) and the datapath for the forwarding unit (bottom)

III. Checkpoint 3:

For this checkpoint, we implemented several advanced design features to improve the performance of our CPU. We chose to implement pipelined instruction and 4-way data caches, an L2 cache, a victim cache, and a tournament branch predictor. We tested these features using both the mp4-cp3.s code, all three competition codes, as well as custom test code to check the behavior of each state of the caches (i.e. writebacks). Upon scanning through the waveform and checking the final register values of these test programs, we were confident that our features worked properly. Additionally, we verified the performance gains from these solutions by implementing performance trackers such as a cache miss count, correct prediction count, and victim cache hit count.



Finalized datapath for the pipelined CPU with advanced features

(C) Advanced design:

I. L2:

The L2 cache provides more cache space to the CPU which significantly reduces the miss rate if L2 cache is fully used. However, if L1 is big enough, L2 will drag down the performance as L2 increases the time to fetch data from the main memory.

In our design, adding L2 cache doesn't provide significant speed up. Our theory is that since our L1 cache is big (as it is 4 ways), the extra cache space that L2 provides is excessive.

L2 cache can be implemented by changing the ports of L1 cache; since L2 sends read data from main memory and receives writeback data from L1, the read/write data will all be 256 bits long. And since L2 only gets a written request when L1 needs to write back a whole block, we also don't need the membyte_enble as we just need to write everything into a block. So for L2, we just replace 32 bits read/write data ports of L1 cache with physical write/read data and set the membyte_enble to 0xffffffff.

II. Victim cache:

Victim cache is a technique that we used to reduce conflict misses. Whenever a higher level cache misses, the data being evicted is stored in the victim cache. Since these lines were evicted because of a higher level cache's capacity limitation, there is a good chance of them being referenced again. It is much faster to retrieve it from the victim cache than from the main memory because it only takes one cycle for misses in higher-level cache and hits in the victim cache.

Our victim cache is 16 way set associative and one set per way with a Pseudo LRU replacement policy. We didn't implement it using pipelined although we should for the sake of reducing power and logic consumption. Pipelined cache is another level of complexity added to the already difficult victim cache design.

There were three locations that we deemed reasonable to place the victim cache, that is, between L1 data-cache and arbiter(Since we tested that there were much more conflict misses from L1 data-cache as opposed to L1 instruction-cache), between arbiter and L2, or between L2 and main memory. For the ease of connecting, we end up with placing the victim cache between L2 and main memory, it is probably not the best place to put it since we have a very big L1 cache written with pipelined cache so we don't have too many duplicated misses, therefore, victim cache performance is not the best.

We tested the victim cache separately without pipelined cache and L2 cache and connected it between the given data-cache and arbiter. Using a counter, we record a total of 1413 misses in L1 data-cache and a total of 1011 hits in the victim cache. Out of the 1011 hits in victim cache, 419 of them were L1 write misses. It is a big performance boost since it shrinks the 30+ cycle penalties of read/write misses to 1 cycle penalties.

III. Pipelined cache:

The pipelined caches improve performance by improving the fmax of our system. Since it was using the separate Bram module, we were able to implement a very big L1 cache with 4 ways and 32 sets per way without hurting the frequency too much, which significantly decreased the miss rates of both the instruction-cache and the data-cache. We only have 1-2% of misses in data-cache and 0.05% misses in instruction-cache when running competition tests.

We faced challenges with this approach when updating the caches to spread over two stages of the pipeline. This changed the forwarding path for the memory stage, and required us to utilize bubble states to stall part of the CPU pipeline for load operations followed by instructions that generated a destination register dependency. Additionally, we had to implement a mechanism to handle cache stalls since the two cycle cache can only detect a hit on the next cycle from the request.

IV. 4-way Set Associative Cache:

Adding more ways can increase the hit rate of cache as it improves the temporal locality. For example, in a loop, if we access three data with the same index in each iteration, one of them will be kicked out in the cache in 2 ways. Therefore, cache has to re-fetch the data from the main memory for every iteration. 4-way set associative resolves this situation as it can hold up to 4 data with the same index in a set at the same time. However, 4 ways has its drawbacks. Increasing the number of ways to increase the search time of tag.

Implementing a 4-way set associative cache provides decent speed up to our design, as it reduces the miss-rate at the cost of a small amount of extra search time.

In our implementation, we first add 2 more sets of data arrays as well as other signal arrays to a 2-way cache. Then we use pseudo LRU to indicate which data in the set should be replaced by the new coming data.

V. Tournament Branch Predictor

The tournament branch predictor consists of both a local branch history table and a global branch history table. The local table takes the 6 least significant bits from the PC as input and hashes a branch prediction and a target PC which the PC loads as its next value. The global branch prediction takes 5 bits from the PC concatenated with 4 bits which store the outcome of the previous 4 branches as the hashing mechanism for the table. To choose the correct prediction between these two tables using a two bit predictor. This predictor functions similarly to a conventional two bit predictor, instead of using the states strongly-global, weakly-global, weakly-local, and strongly-local. As the predictor generates more missed predictions, it shifts to the states which use the other table. Overall, our branch predictor was able to correctly predict the correct branch PC 77% correctly on average in the three competition codes.

4. Conclusion :

We have successfully implemented a five-stage pipelined CPU with tournament branch predictor and with that we build a complex cache structure including pipelined L1 instruction cache and L1 data cache, arbiter, L2 cache, and victim cache. It is very hard to test each component out separately and it is even harder to test the combination of different components but we finished it in the end. The concept of balancing frequency and logic cost are crucial in this assignment and we sadly didn't reach our goal in that category. However, our project consists of an overall working CPU and cache system. We are proud of what we've accomplished.