MP4: Team Tux
RV32I Instruction Set Pipelined Microprocessor

Raj Baala
Ayan Deka
Darshan Desai

Introduction

The goal of this project is to fully implement a RISC V 5-stage pipelined microprocessor with split L1 caches. Specifically, the processor had an L1 instruction cache and L1 data cache. We also decided to enhance our processor design by adding features such as implementing the M extension, implementing a 2 level global branch predictor with a branch target buffer, hardware prefetching, parameterized caches, and multi level caches. Due to the numerous stalls that occurred from branching as well as load/store instructions, we knew that we wanted to implement features that specifically helped in boosting cache and branch prediction performance. The specifications of the final processor included a 2 way 2 set instruction cache, 2 way 2 set data cache, arbiter between both caches, cache line adapter, 2 level global branch predictor with a branch target buffer, hardware prefetching, and the RISC V M extension.

In the report, we provide details on metrics that we measured such as the number of cache stalls, branch stalls, multiply stalls, and divide stalls as well as the advanced features we implemented in order to affect our performance in a more positive manner. Regarding computer architecture as a whole, it was crucial to determine what are some components that contribute to major slowdowns in a processor and determine some extra features to implement in order to address these performance issues. In addition, it was also important to learn more about the implementation of required schemes such as data forwarding and stalling in order to ensure the smooth pipelining of the processor across all 5 stages.

Project Overview

Due to the timeline of the project, we knew that we needed to leverage each other's strengths and determine the most feasible way to divide up the project so that we can not only meet deadlines in a sufficient manner but all team members are comfortable and interested in the parts that they are assigned. Specifically, at the beginning of the project, we divided the components in a way such that Ayan would work on the cache components of the processor due to his efficient cache design while Raj and Darshan worked on the overall CPU components of the processor.

Initially, we began the design of our processor by implementing more of a flat design. However, we immediately ran into issues and realized that it was taking too much time to debug issues regarding timing of signals through each stage. Towards the end of CP1, we transitioned from a flat design to a more modular design which ended up being very beneficial for the rest of the project timeline as it was much easier to detect any issues as we worked through checkpoints. This modular design consists of breaking up each of the 5 stages into its own modules. Due to the extensive issues that arose from data forwarding and cache stalling, there was not much thought into which advanced features to implement. However, we knew it was crucial to make advanced design features to the caches as well as branch prediction since they contribute to major slowdowns.

Design Description

Overview

For this section, we describe the work that was completed at each checkpoint. Specifically, we describe design schematics that were created, implications that we had to face, and verification designs to ensure that each component worked as intended.

Milestones

Checkpoint 1

For checkpoint 1, the processor had to be a basic pipelined processor that did not handle any data and control hazards. In this checkpoint, dual-port “magic” memory was used so that cache misses or memory stalls did not need to be handled. The basis of this checkpoint was to have the basic pipeline structure working as well as having a working connection with RVFI. Some implications that we ran into with this checkpoint was the transition from a flat pipeline design to modular pipeline design which was described in the previous section as well as an issue with store instructions. However, with sufficient testbench code, we were able to fix our issue with store instructions. In addition to the overall basic pipeline design, we created paper designs for the arbiter as well as data forwarding and branching. The paper designs are shown in Appendix A. The basic design of the arbiter is the idea that this unit sits between both the I cache and D cache and the physical main memory in order to determine what access request to service. For the design of our arbiter, we chose the design that would prioritize requests from the instruction cache rather than the data cache. In general, data forwarding includes checking to see if the destination register in the memory or writeback stage matches any of the source registers in the execute stage. If this condition holds true, then data must be forwarded from either stage to the corresponding source register in the execute stage otherwise the corresponding source register will not hold the correct data.

Checkpoint 2

For checkpoint 2, We had to extend the design to include hazard detection and forwarding, including static-not-taken branch prediction. Specifically, the intended behavior was to flush the previous two instructions in the pipeline if the branch is taken in the execute stage. This contributed to major slowdowns which were later addressed when we implemented the branch target buffer and 2 level global branch predictor. Regarding hazard detection and forwarding, we used the book as a reference to create a forwarding module that checks if any of the source registers in the execute stage matches any of the destination registers in the memory or writeback stage. If this condition matches, then we need to forward the alu output from either corresponding stage to the corresponding source register in the execute stage. After covering general cases, we checked specific hazard detections regarding load/store and branch/jump instructions and forwarding the appropriate output values. In addition to hazard detection and forwarding, we also had to create stall conditions as we needed to wait for the response signals from the data cache or instruction cache before proceeding through the pipeline. On the cache side, we also implemented the arbiter. Our design for the arbiter prioritizes instruction cache services over data cache services if conflicting services occur. After implementing the arbiter and integrating our cache from the previous MP, we no longer used magic memory that was given to us. The most challenging aspect of this MP was ensuring that all hazard detection and forwarding cases were covered and flush/stall signals were properly implemented in our processor. Overall, our area was way over the threshold which was addressed in later checkpoints.

Checkpoint 3

This checkpoint consisted of implementing our advanced design features. We first started with the M extension as well as multilevel and parameterized caches. Regarding the M extension, we worked on creating the DADDA multiplier which allowed our processor to complete multiply instructions in just 3 cycles. In addition to the DADDA multiplier, we implemented the basic subtract shift divider as there are no other advanced dividers. With our subtract shift divider, our processor completed divide instructions in 34 cycles. Regarding the method of testing for the M extension, we first started with creating a testbench that ensured timing was matched and the instruction was traversing through the correct states. After verifying the computation as well as edge cases, we ran final tests by creating assembly tests. We also implemented a similar style of testing for our other design features. After coremark m was successfully running, we moved on to implementing a dynamic branch predictor. We initially implemented a BTB with a 2 level global predictor. However, this made it harder to debug branch prediction issues. So we decided to first implement and integrate a BTB with a 1 bit prediction counter before adding the 2 level global predictor. On the cache side, we implemented design features such as multi level caches and parameterized caches as well as hardware prefetching.

Advanced Design Options

Option 1: BTB branch prediction / 2 level global predictor

Design

For the branch target buffer, we made the buffer as a 4 way direct mapped cache. The buffer stores 4 PC addresses as well as the corresponding branch target address. We decided on the 4 way design as we wanted to maintain our 3 bit PLRU scheme. For the 2 level global predictor, we implemented a 3-bit global branch history register that indexes into 8, 2-bit saturating counters in the pattern history table.

Testing

For testing, we ran a branch instruction testbench where the code repeats the same branch instruction 16 times and then after the 16th time, the branch will not be taken and progress through the rest of the testbench. This allows us to ensure that the PC address is correctly stored in the branch target buffer and the number of branch stalls are reduced. This also allowed us to ensure that the global history two-level branch predictor was updating correctly after each branch instruction as we implement 2 bit saturated counters.

Performance analysis

Through our implementation, we saw a significant improvement in branch prediction. Specifically, the implementation of the branch target buffer cut down the number of branch stalls by half. However, we saw very minimal improvement with the integration of the 2 level global branch predictor with the branch target buffer.

Option 2: M extension

Design

For the divider extension specifically, we implemented a 32 cycle shift subtract divider. Through this divider, we saw a somewhat significant slowdown as one divider instruction takes 34 cycles as there is one cycle for the idle start state, one cycle for the end of computation state, and 32 cycles of subtract and shift. In addition, we ensured that we checked overflow conditions as well as dividing by zero cases where the divisor is 0. In addition, to speed up the time it takes to complete an instruction, we also checked to see if the dividend is 0 which is not necessary to complete all 32 cycles.

For our multiplier design, we decided to go with the Dadda Multiplier. This multiplier is better than other common multiplier algorithms as it reduces the number of total adders required. The overall overview of the algorithm is to first compute all partial products by ANDing each bit *w1* with each bit *w2* to yield a result of *l1 * l2*, where *l1* and *l2* are the bit lengths of each operand. Since we are doing 32x32 bit multiplications, this would yield 1024 ANDed results. The second step is to arrange these results into a tree where each result bit corresponds to a weighted column. The next step is partial product reductions in multiple stages, where each stage has its corresponding maximum column height of bits to reduce down to. The formula for the maximum height of each stage's columns is d1 = 2 and dj+1= floor(1.5dj). The initial value of j is chosen as the largest value such that dj < min(n1,n2) where n1 and n2 are the number of bits in the multiplicand and multiplier. For 32x32 bit multiplication, there are a total of 8 stages where the reduction heights of each stage are 28, 19, 13, 9, 6, 4, 3, and 2 respectively. If the height of the current column is <= dj, then move to the next column. If height of the current column = dj + 1, add top two elements in a half adder, putting the sum bit at the end of the current column and carry bit at the start of the next column. If height of the current column > dj + 1, add the top three elements in a full adder, placing the sum at the bottom of the column and place carry bit at the bottom of the next column. The final stage is a 64-bit addition to get the final 64-bit result. A python script was made to generate the SV file for the dadda multiplier datapath.

Testing

Testing for both divider and multiplier were done through their own testbenches following a UVM testbench format. Multiple tasks were used to test each multiply and division operation via assertion statements. Since testing every 32x32 bit combination would take very long, each testbench ran a subset of important inputs that would satisfy the verification of the multiplier and divider units. After DUT testing was done, we tested the units hooked up to the CPU and checked if spikes matched on all multiply and divide operations.

Performance analysis

The dadda multiplier only requires 3-4 cycles for a multiply operation to finish, making it very fast at completing multiply operations in the m-extension coremark. As for the divider, it takes up to 32 cycles for the operation to complete depending on the size of the input. The final stage of the dadda multiplier addition requires a 64-bit addition which would become the critical path of the circuit if not optimized. Since multiple instructions only require either the lower or upper 32-bits of the result, the carry bit from the lower 32-bit addition can be latched to reduce the critical path. When the upper half result is needed, you will pay an extra cycle penalty to load the carry bit, but comes with significantly reducing the critical path length. Thus, the min cycles to complete a multiplication operation is 3 cycles while the max is 4 cycles.

Option 3: multilevel caches / parameterized caches

Design

For parameterized caching, we decided to do up to 8 ways with an eviction scheme of Pseudo LRU. Additionally, we parameterized the set index and associativity of our caching. The way we did this is by simply including these numbers in the parameter of our caching module. After that, we made sure to include these parameters when setting up our data, tag, pseudo LRU, valid, and dirty arrays so that it would correctly fit what we wanted. Once we created these parameterized caches, we implemented multi level caching, where L1 was the smaller cache and L2 was the bigger cache.

Testing

We simply ran our test cases that we used for checkpoint 3, but extended it to be up to 8 ways. We used randomized seeds to test different addresses, as well as different reads and writes to a certain set index.

Performance analysis

The performance of our multilevel caching greatly improved our instructions per cycle. The final design we had to do to meet the area constraints was a two set, two way PLRU cache. This gave us an IPC at around .3. However, when we decided to disregard the area constraints, we had a 16 set, 3 way PLRU. This increased our IPC to around .76, with the expense of area.

Option 4: Hardware Prefetching

Design

We decided to implement prefetching between our L1 and L2 caches. The design of the prefetch was fairly simple, as it was a control based module. We would have two states, a normal read and a prefetch state. In the normal read state, we would just pass the signal down. In the prefetch state, we would fetch the next line immediately after the current read. Additionally, we would block the new read during the prefetch.

Testing

For testing, we simply made tests that would be around the two states that we implemented. For example, we would test the read while prefetching and seeing if the memory address lines up with a prefetch memory address. Additionally, we would make sure the correct signals would be passed down, and that the value of these signals line up with the expected value.

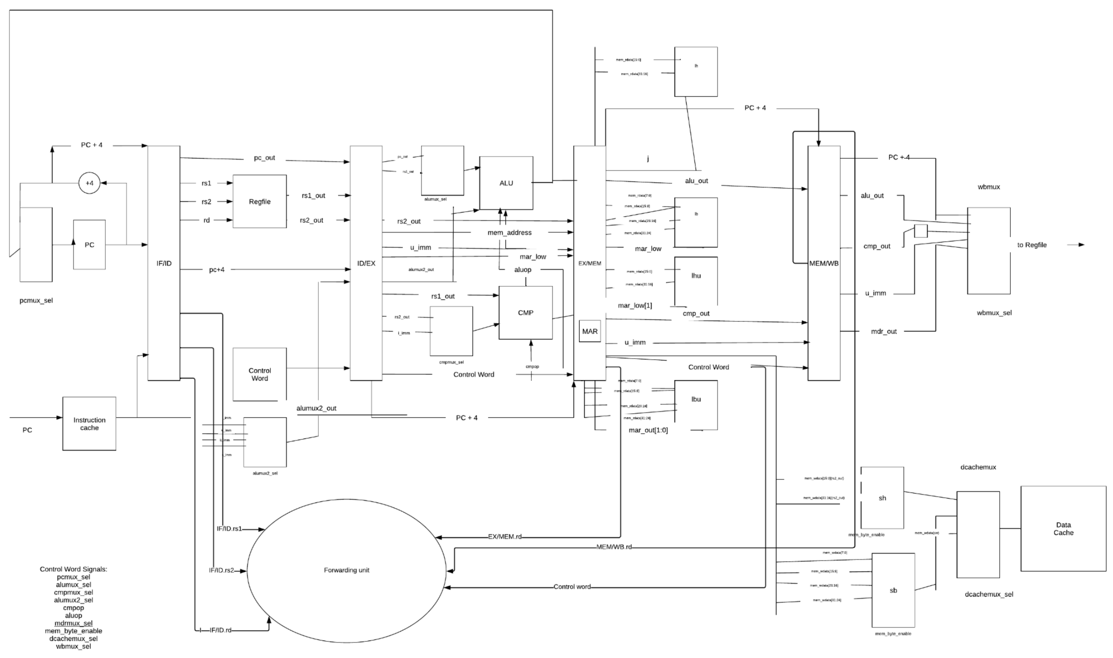
Performance analysis

When having this prefetching between the two caches, it improved our performance slightly. It went from a maximum IPC of .76 to .78. However, since this prefetching requires an L1 and L2 cache to be between, the area had increased drastically when implementing this.

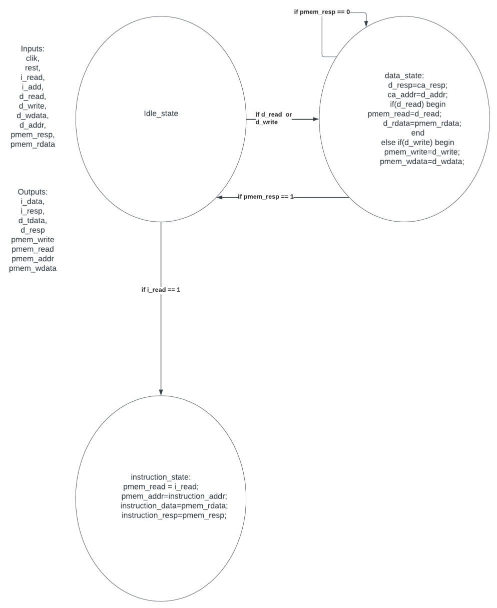
Additional Observations / Conclusion

Overall, this project has allowed us to learn a lot about CPU architecture, design strategies, as well as design tradeoffs(performance, area, and power). There were a lot of challenges along the way of fixing small bugs and edge cases, but allowed us to get better at debugging and identifying quickly where potential issues may be within the design. The hardest part of this project was optimizing for best performance while maintaining a small area and power. Especially with the smaller area constraint this semester due to the introduction of the SRAM, it was very difficult to push out more performance when our cache area was limited from flip flop usage. Going back, we would have attempted to pipeline the cache in order to gain a significant boost in performance while having low power and area as well. Something else that we would go back and spend more time on is optimizing the critical path of the CPU so that we could drive the frequency of our clock a bit higher, allowing us to decrease our delay metric. Moving a few muxes to different stages could have helped to optimize some more.

Appendix A



Data forwarding and data hazarding design



```
data_state:
    d_resp=ca_resp;
    ca_addr=d_addr;
    if(d_read) begin
pmem_read=d_read;
    d_rdata=pmem_rdata;
        end
  else if(d_write) begin
    pmem_write=d_write;
  pmem_wdata=d_wdata;
```

```
instruction_state:
pmem_read = i_read;
pmem_addr=instruction_addr;
instruction_data=pmem_rdata;
instruction_resp=pmem_resp;
```

Arbiter Design