ECE411: Computer Organization and Design
Final Report

Group Name: Malfunctioning-CPU
Group Members: Kristopher Koepcke, Jinrui Hu, Jingxuan Wu
Date: May 10, 2023

Introduction

Our goal for this project was to design and implement a 5-stage pipelined CPU that supports forwarding and hazard detection with a memory hierarchy that includes an instruction cache and data cache that runs the rv32i base instruction set architecture. After meeting this goal, we also extended our CPU with several advanced features to speed up our design. These features include a local history branch predictor with a BTB, a 4-way set associative L2 cache, prefetching support, and the RV32M standard extension. Details of each feature and the challenges involved in their implementation and verification will be provided. Our CPU also includes performance counters that are used to compile performance metrics on a per program basis and are displayed at the end of program execution. These metrics include, cache hits/misses/prefetched blocks, stalls due to memory access penalties and incorrect branch predictions and branch prediction accuracy with misprediction breakdown statistics.

As we got deeper and deeper into the project, we realized that our high level designs often overlooked small details that caused the implementation to become very complicated. Luckily we had each other to collaborate with; overcoming these obstacles, overcoming each other's weaknesses and empowering each other's strengths and pushing forward with the project. A great reminder, that teamwork always makes the dream work!

Project Overview

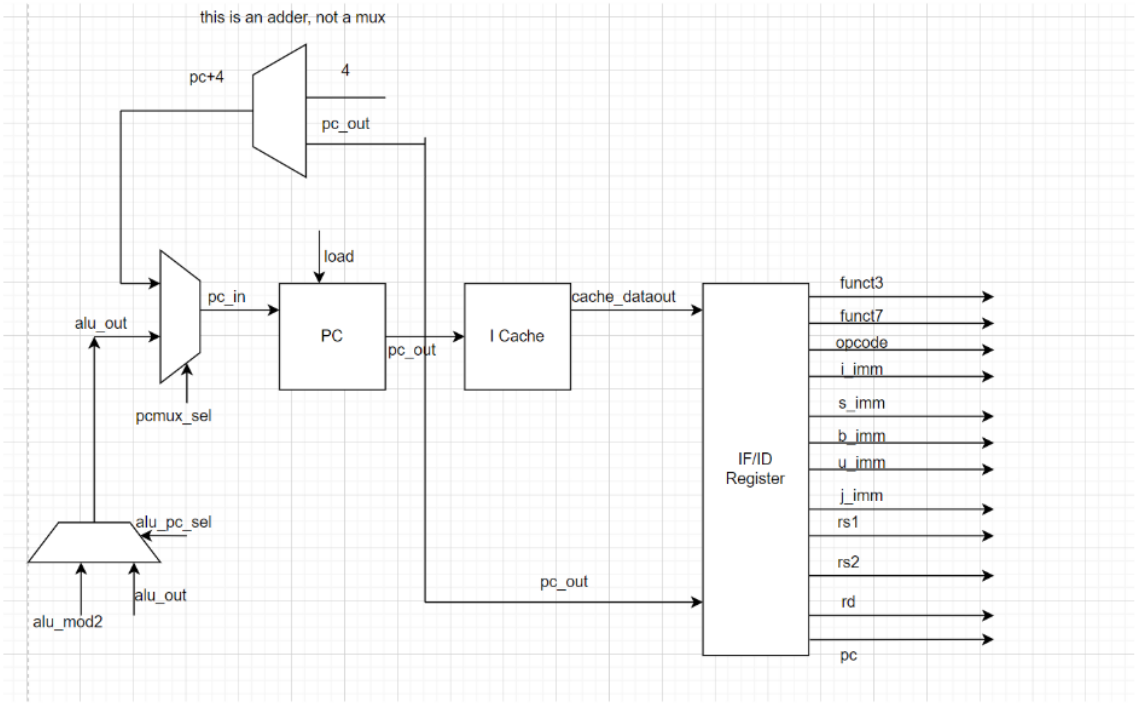
Our design and Checkpoint 1 went smoothly: only encountering 4 bugs. Checkpoint 2 gave us a very hard time. We ended up finding 24 bugs, 15 of which were forwarding issues. We also had complete design components missing from our original design such as forwarding the correct register value on store instructions (which uses rs2 differently than all the other instructions). Countless nights were spent going through each bug one by one, using verdi to track and source the problem. After endless deliberation and some TA support, we were able to correct all the errors that were generated from the given testcode files including coremark's benchmark program. Although we didn't finish this checkpoint on time, the working programs gave us a vast amount of verification coverage for testing our advanced features, which set us up for success in Checkpoint 3. The advanced features were decided and given out based on interest, ability and point value. The main issues that occurred were solving integration errors and file conflicts. We also didn't add performance counters until very late, which caused us to not realize that some advanced features weren't working properly, such as the branch predictor. We also realized at this point that our prefetcher was actually lowering our performance. Checkpoint 4 was mostly data collection and analysis to inform us which features to include or remove based on our power and execution time performance metric. Time was also spent on optimizing the fmax value for optimal execution time performance.

Milestones

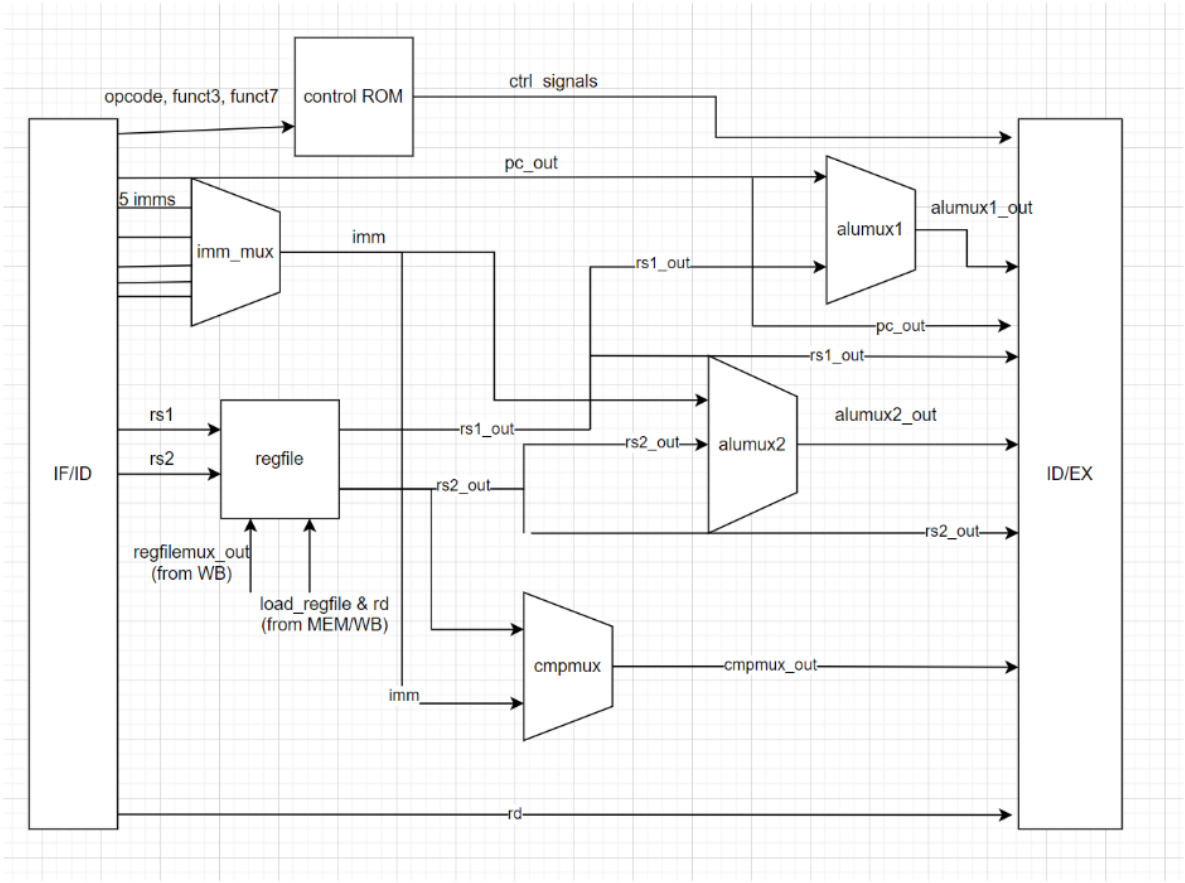
Checkpoint 1

In checkpoint one, we implemented the basic 5-stage pipelined CPU without data forwarding or hazard detection. We named our intermediate registers IF_ID, ID_EX, EX_MEM, and MEM_WB, where IF, ID, EX, MEM and WB were the 5 stages. Below is a diagram per stage showing our initial design.

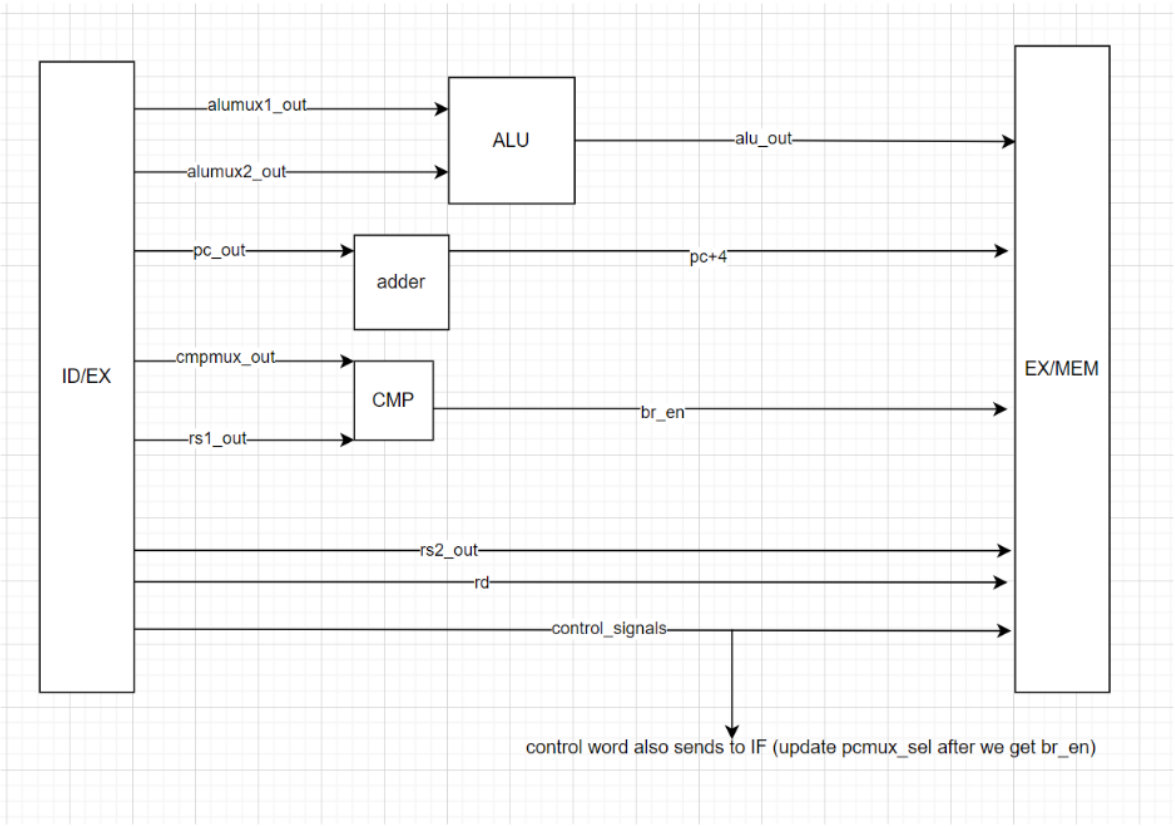
- IF



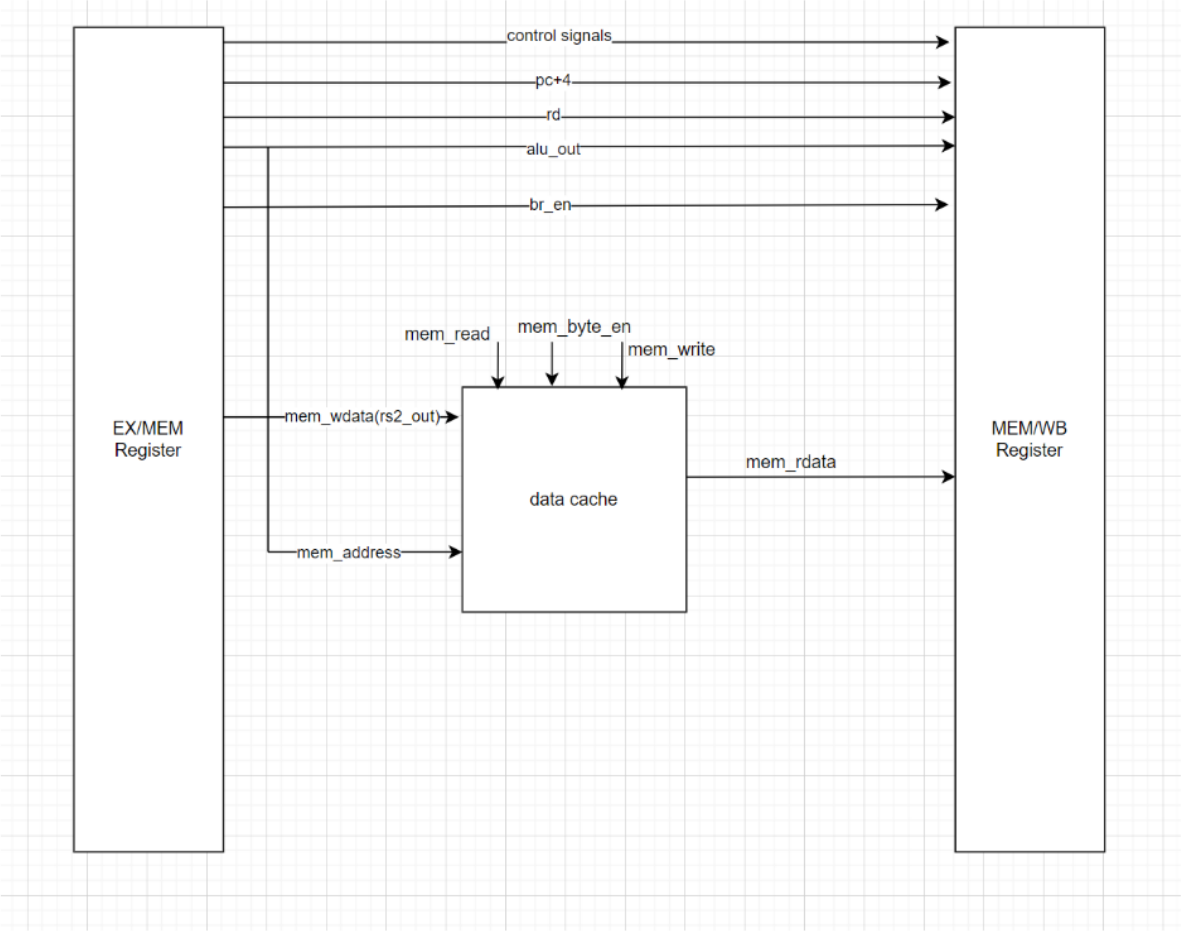
- ID


opcode, funct3, funct7
control ROM
ctrl_signals
IF/ID
pc_out
5 imms
imm_mux
imm
alumux1
alumux1_out
rs1_out
pc_out
rs1_out
rs1
rs2
regfile
rs1_out
rs2_out
alumux2
alumux2_out
ID/EX
rs2_out
rs2_out
regfilemux_out
(from WB)
load_regfile & rd
(from MEM/WB)
cmpmux
cmpmux_out
imm
rd

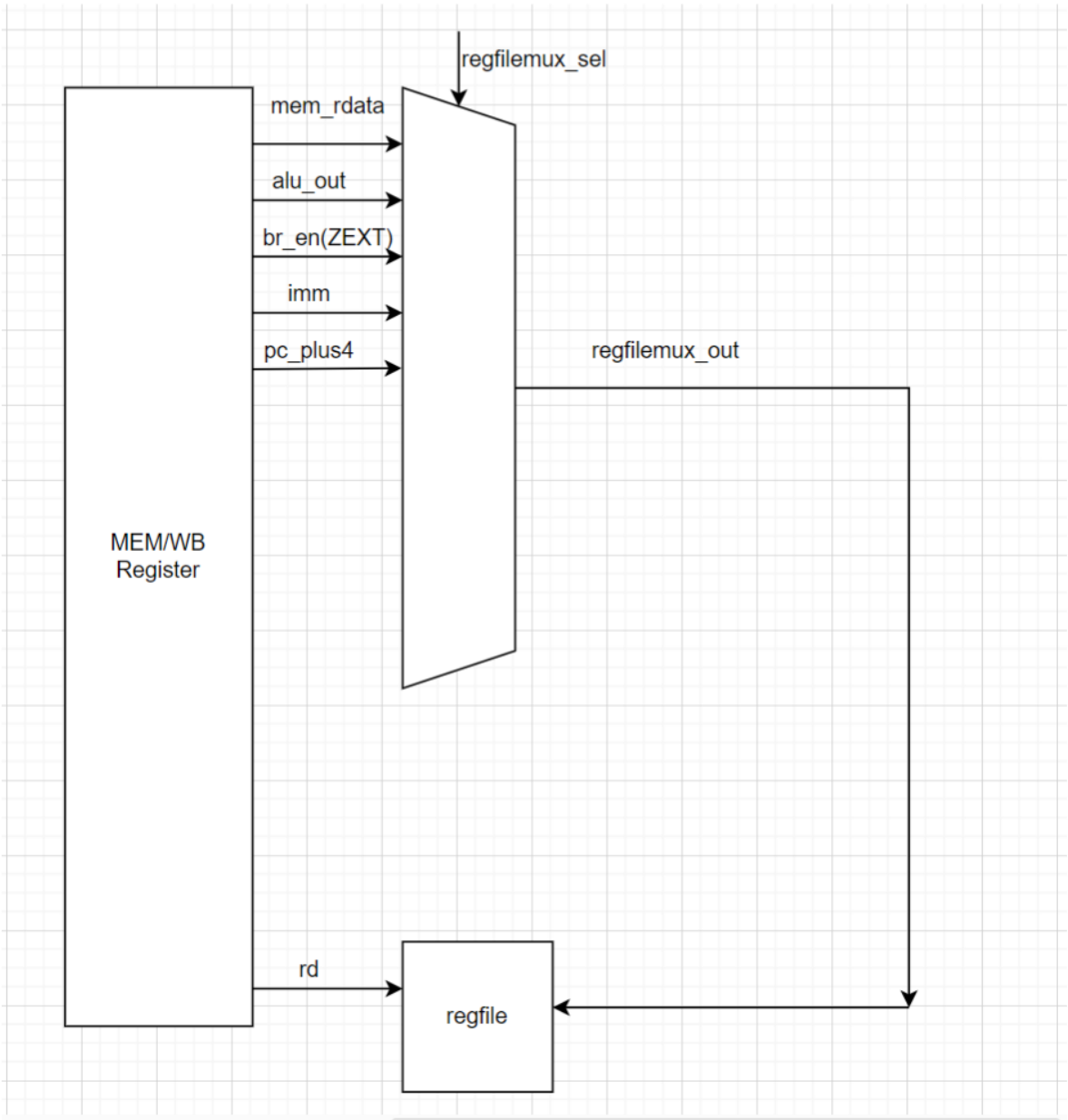
- EX


ID/EX
alumux1_out
alumux2_out
ALU
alu_out
pc_out
adder
pc+4
cmpmux_out
CMP
br_en
rs1_out
EX/MEM
rs2_out
rd
control_signals
control word also sends to IF (update pcmux_sel after we get br_en)

- MEM


control signals
pc+4
rd
alu_out
br_en
mem_read
mem_byte_en
mem_write
EX/MEM
Register
mem_wdata(rs2_out)
mem_address
data cache
mem_rdata
MEM/WB
Register

- WB



We tested our program by running the mp4-cp1.s testcode and comparing the register values with the spike log. We encountered only four bugs and were able to easily debug them using verdi.

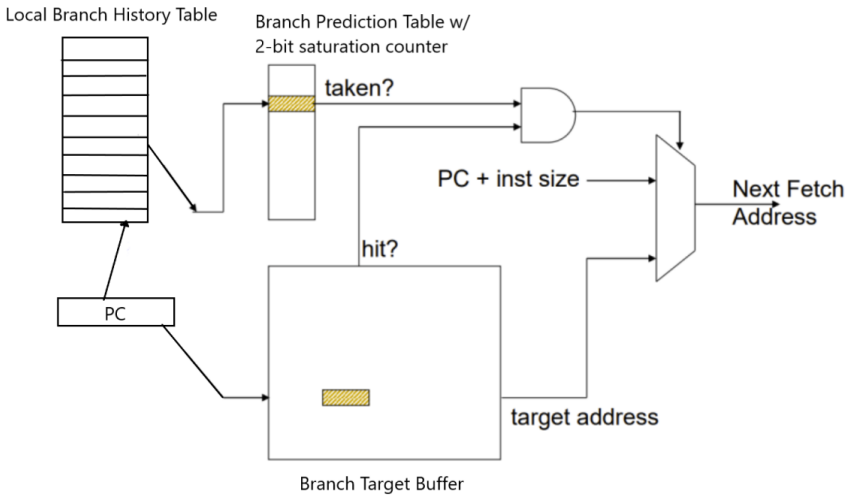
Checkpoint 2

In checkpoint 2, we implemented the arbiter, data forwarding, hazard detection, and connected the RVFI signals. The arbiter decides who should be served first when there are simultaneous requests from the i-Cache and d-Cache. We hooked up the RVFI monitor by creating a separate struct that contains all the required RVFI signals and passed that struct

through each stage in our pipeline, updating any signals along the way that are changed. The data forwarding unit is responsible for sending the correct value of registers that are dependent on an instruction currently being executed in the pipeline. This happens for instructions in the EX, MEM and WB stage. The data forwarding unit caused us the most bugs. One particular case is the special use of rs2 for store instructions which required its own forwarding path separate from the rest. In checkpoint 1, we did not consider that rs2 had to be forwarded on a store instruction, we only considered R and I type instructions. We fixed this bug by creating another forwarding path from our forwarding unit to the EX stage and conditions to check for store instructions. Another critical issue was the immediate value '2' being confused for rs2, triggering a forwarding condition. The data hazard unit is responsible for detecting and avoiding issues that occur as a result of instructions not finishing before other instructions need them to be finished. This includes stalling our pipeline on cache misses and inserting a bubble into our pipeline on read-after-write dependencies. One of the biggest issues we had was inserting a bubble between EX and MEM when it should have been ID and EX to avoid a niche edge case. Our testing method at this point was using the provided testcode: cp2, comp1, comp2, comp3, coremark, and cp3. We figured that if our CPU could run all of these programs without issue, then it is okay to assume the CPU is working properly.

Checkpoint 3

Branch Predictor



- **Design**
  We used a local history branch predictor that uses a BTB, a local branch history table with 5 bits of history and 32 entries, and a direction prediction table that uses a 2 bit saturation counter and 32 entries. Our original design would statically predict a not taken branch path. If it was incorrect, it would have to reset two stages which essentially stalls the pipeline for two cycles. To reduce this stalling, we wanted to improve our prediction accuracy, which is exactly what this predictor does. According to our performance

metrics, the predictor is able to predict with an accuracy of ~70% minimum and as high as 90%.

- **Testing Strategy**
  I used all the provided testcode to test correctness. This includes cp3, comp1, comp2, comp3, and coremark. I also used the performance benchmarks to determine accuracy and I used verdi to inspect the local branch and direction prediction tables. Once all the programs ran without issues I assumed it was working correctly.
- **Performance Analysis**
  I turned out to be wrong with my assumption because the accuracy was ~40% at the very beginning. When looking on verdi I also noticed that the prediction history table was only using 2-4 entries (out of 32). This ended up being a result of incorrect indexing within the branch prediction module. After fixing this and an issue with the BTB, the predictor jumped in accuracy to between 74% and 90% depending on the program.

Branch Target Buffer (BTB)

- **Design**
  The branch target buffer works with the branch predictor. Whenever we have jump instruction, we store the calculated target address (from the EXE stage) in the BTB. In the IF stage, we use all 32 bits of the pc to find a match in the BTB, if there is a hit and the prediction is taken, then we directly jump to the address stored in BTB.
- **Testing Strategy**
  We did not test the BTB before integration with the branch predictor, so the testing was apart of the branch predictor testing: running all the provided test code for wide coverage, using performance counters, specifically for when the BTB stores the wrong target address and using verdi to see if signals are being sent correctly.
- **Performance Analysis**
  Originally, we only had 16 entries in the BTB which caused a lot of evictions because branches that weren't stored there would be overwritten. This in turn caused accuracy to drop because there weren't enough branches being recorded, which caused incorrect targets to be common. To alleviate this, we increased the number of entries to 64 which greatly improved performance.

Return Address Stack (RAS)

- **Design**
  The Return Address Stack is used to store the return address and return value of a function call. Standard calling convention uses r1 as the return address register, and r5 as an alternate link register. The JAL instruction should push the return address onto a return-address stack only when rd=r1/r5. JALR should push/pop from the RAS. The rd of a JALR instruction is used for storing the return value of a function call.

- **Testing Strategy**
  Not Applicable
- **Performance Analysis**
  We decided not to incorporate the RAS as the return addresses of JAL and JALR instructions were already being stored in the BTB so the gain in performance would not be very large, so the time commitment for debugging and verification would not be worth it.

L2 Cache

- **Design**
  The cache miss penalty was by far the largest contributor of stalls. To reduce this penalty, we introduced a shared L2 cache that the i-cache and d-cache both request from. The first instance of our L2 cache was directly taken from mp3, thus it was a 2-way set associative cache with 8 sets. The addition of the L2 cache greatly improved performance, sometimes cutting the execution time in half, but also greatly increasing power consumption.
- **Testing Strategy**
  We mostly relied on the shadow memory and running the suite of test programs to make sure it was working properly. Verdi was used to inspect the waveform when an error was encountered. Once all the code was running without shadow memory errors, I assumed correct functionality.
- **Performance Analysis**
  The hardest part of adding the L2 cache was trying to remember what all the signals and wires were for. But once everything was connected, there were no major issues.

4-way Set Associative Cache

- **Design**
  To improve upon the L2 cache, we decided to make it a 4-way set associative, since we knew that there would be a lot of conflicts due to it being a shared cache between the i-cache and d-cache. This required a massive rewrite of the existing code: renaming things to be more clear, changing variables to be arrays (which will be indexed on a per ‘way’ basis), remaking the pLRU hardware to support 4 ways, and adding conditions in the control for all 4 ways.
- **Testing Strategy**
  Similar to the L2 cache, the coverage provided by the 5 testcode programs were used to find shadow memory errors, if there was an error, verdi was used to debug. Once these programs were working, I also looked in verdi to check and see if all the entries in the cache were being used. After this was checked, I assumed correct functionality.

- **Performance Analysis**
  The biggest issue was making the pLRU logic work with 4 ways. I realized that my understanding of this part of the code was actually incorrect when writing mp3, but it worked out because mp3 only used 1 bit. I first programmed the hardware to store the least recently used entry instead of coding the hardware to point to the most recently used entry and using that to calculate the least recently used entry. Once I figured out my error, the programs worked but there was a combinational loop in our synthesis. This was a major issue that took a long time to fix. The eventual solution was to add a buffer between all incoming and outgoing signals of the L2 cache, and update this buffer on the negative edge. This fixed the issue despite being unconventional. This feature was the most important feature in our whole design, decreasing the execution time by as much as one third and at minimum gave a 45% speed increase. As a result, all other features were built on top of the 4-way shared L2 cache.

M-Extension

- **Design**
  The M-Extension adds 8 more instructions (MUL, MULH, MULHSU, MULHU, DIV, DIVU, REM, REMU) to the RV32i ISA. Based on the opcode, the M_Ext module would choose to either use the multiplier or the divider. Notice that when the M_Ext unit is executing, the whole pipeline will be stalled and it resumes as soon as the execution is complete.
- **Testing Strategy**
  We unit tested the multiplier and divider by giving two 32 bits numbers and allowing the module to calculate the multiplication or division. After we verified the multiplier and divider, we integrated them into our design and ran the coremark to see if they functioned correctly and measure their performance.
- **Performance analysis**
  The m-extension is the second most important feature in our design. Since it only affects code that uses the multiplier or divider (comp2 and coremark), it was a bit specialized offering no benefit to any other type of program, but for the programs that did use the multiplier, we saw at minimum a 26% speedup and at most a 228% speedup.

Prefetching

- **Design**
  This feature is implemented as an upgrade to the L2 cache. The prefetcher fetches the i + 1 block in addition to the ith block everytime there is a cache miss. It starts the prefetch by reading from the physical memory when the state of the arbiter is idle. It improves the performance when the requested blocks are sequential but will degrade the performance when the program contains a lot of loops or lw/sw instructions because the prefetcher will start polluting the cache with data that won't be used.

- **Testing Strategy**
  We tested the prefetcher by running all the testcode and checking the waveform in verdi. By adding cur_state signal, we could see it enter the prefetching states. By additionally adding mm_address, cur_address, and address signals we could verify that prefetching is working. We also looked at performance increases or decreases as a sign of incorrect operation or at least a high level implementation issue.

- **Performance analysis**
  We eventually realized that the prefetcher actually decreases performance. We eventually realized this is because the prefetched data causes a lot of pollution in the cache by evicting potentially useful data. And since it's a shared cache, the i+1 block is not a good predictor of the next block because a lot of accesses are made by the d-cache. To fix this we added a prefetch storage element to hold the data. Unfortunately, this method causes sw/lw instructions to make a lot of useless prefetching requests. The last version of our prefetching included multiple ways in the storage elements for prefetched data which increased the number of useful prefetched blocks. This also came at the cost of power consumption, and the performance benefits ended up being very small, which led us to the decision to remove the prefetcher in the next checkpoint.

Checkpoint 4

The main point of checkpoint 4 was to do whatever you have to do to improve your cpu design for the design competition. The first thing I thought to do was increase the cache sizes, which I did. I doubled the L2 and L1 caches which greatly improved performance but also greatly increased power consumption. Gathering the data needed to come to this conclusion took a lot of time though. This is because I had to synthesize for each design, run the power report 4 times (once for each testcode program: comp1, comp2, comp3 and coremark), record their execution time, record their power consumption, find the average of both of these and use these values in our performance metric: P*D^2*(100/fmax)^2. P is the average power usage, D is the average execution time, and fmax is the maximum frequency. I also removed prefetching because the performance gains did not justify the power usage. Prefetching improved the performance by only ~1% but increased power by ~30%. The last part was optimizing fmax, which required us to set the clock period such that our timing report would calculate 0 slack. The problem is that the smaller the period you give the synthesizer, the harder the synthesizer will work to force the delay to fit within that period. This resulted in running make synth at least 100 times, and each run takes 5-15 minutes. After finishing this process, we were able to more than triple our fmax, giving us a frequency of ~373 Mhz and average execution time for the competition code of 122 microseconds.

OPTIMIZED fmax					
Advanced Features	**comp1 (ns)**	**comp2(m) (ns)**	**comp3 (ns)**	**coremark(m) (ns)**	**clk period (ns)**
normal L2, L1, no prefetch	122002.650	121341.150	190010.730	5289343.710	3.08
normal L2, L1, no prefetch	135513.840	136864.560	201558.000	5698040.880	3.5
normal L2, L1, no prefetch	168373.915	174365.185	230794.995	6713615.705	4.5
normal L2, L1, no prefetch	200808.465	211435.695	259099.815	7711591.185	5.5
normal L2, L1, no prefetch	200808.465	211435.695	259099.815	7711591.185	6.5
dbl L2, L1, no prefetch	104459.700	110135.940	154221.940	3393722.860	2.81
dbl L2, L1, no prefetch	108218.600	114574.600	156815.400	3501149.400	2.94
dbl L2, L1, no prefetch	112665.210	119785.890	160128.570	3629449.110	3.08

OPTIMIZED fmax					
Advanced Features	**comp1 (uW)**	**comp2(m) (uW)**	**comp3 (uW)**	**coremark(m) (uW)**	**clk period (ns)**
normal L2, L1, no prefetch	28,900	28,900	28,900	28,900	3.08
normal L2, L1, no prefetch	25,300	25,300	25,300	25,300	3.5
normal L2, L1, no prefetch	20,600	20,600	20,600	20,600	4.5
normal L2, L1, no prefetch	17,100	17,100	17,100	17,100	5.5
normal L2, L1, no prefetch	9,090	8,170	8,230	7,490	6.5
dbl L2, L1, no prefetch	19,900	16,800	17,500	16,600	2.81
dbl L2, L1, no prefetch	19,100	16,100	16,900	15,900	2.94
dbl L2, L1, no prefetch	18,400	15,600	16,400	15,500	3.08

Additional Observations

One of the advanced features we wanted to implement but were unable to due to time constraints is adding a victim cache for both the i-cache and d-cache. We think that our design would have been one of the most competitive had we been able to integrate this last feature. So for potential improvements of this cpu, the victim cache is the first. We also think using a strided prefetcher would make more sense than the basic i+1 prefetcher because we have a shared L2 cache, so we would want to make prefetching predictions based on a little bit more information rather than blindly requesting the next block which we found out doesn't work very well for the d-cache, and could cause pollution problems, which would affect the i-cache as well. Another improvement would be for the multiplier. We used the basic multiplier, but using a more advanced technique could allow us much higher performance gains. The last feature that could improve performance is implementing the C-extension, which would allow more data to be stored in the caches, thus reducing the total number of cache misses.

Conclusion

We set out to design a risc-v based 5 stage pipelined cpu with hazard detection and data forwarding as well as a split i-cache, d-cache memory hierarchy. We finished with exactly that plus a 4-way set associative L2 cache with prefetching, a local history branch predictor w/ BTB, the M-extension and all the friends we made along the way. Despite our design not including as many advanced features as other groups, we ended up being one of the fastest competition code

cpus in terms of execution time. The time invested in making this project succeed was immense, but seeing our CPU not only work, but also be competitive made it more than worthwhile. The best part of this experience wasn't anything to do with the project itself though. Rather, it was 'that moment' where I looked up a computer architecture research paper about improving prefetching and read about introducing a negative feedback system based on hardware collected statistics about the kinds of data that have been received by the cache to determine how aggressive the prefetcher should be, and I understood everything the research paper said. There is nothing more satisfying after all those endless nights trying to understand every detail about this project, than feeling the fruits of that labor though clear understanding of published research in the field.