ECE 411 Final Project Report

LMP

Lakshya Goyal (lgoyal3), Mridul Verma (mverma9), Peter Li (shifang2)

Introduction

This project in Computer Organization and Design was about implementing a 5-stage pipelined CPU along with additional features that we chose to implement in order to improve performance. Other than correct functionality, performance was a priority in this design, as well as power. This project helped us learn about and implement these features in our own CPU while trying to achieve some given benchmark performance goals. This type of design helped us think more carefully about how and what features to implement, as well as how they impact execution time of different types of workloads.

Project Overview

The design of this project consisted of various stages and iterations. The initial stages were to get a basic pipelined CPU working. Next, the goal was to improve the performance by adding specific features, tuned to work best for the given programs. To organize our project, we relied heavily on the Git version control to allow us to seamlessly work on different aspects of the project at the same time, without interfering with each other's work, and then integrating along the way.

Design Description

Overview

During the project, most of our design work began with diagramming and planning the RTL design and then discussing this with the team so that we could all come up with any critiques or suggestions before implementation. This process allowed us to first think through as many potential issues before hand and see it with the rest of the CPU in place. The following milestones outline each of the stages in the project this semester.

Milestones

Checkpoint 1

This milestone was the first, arguably the most important in setting a strong base for our design in terms of correct functionality. This involved converting work from the multi-stage CPU to a 5-stage pipeline, currently without forwarding and hazard units. This is also where we started formulating our project habits such as our Git branches and correct working strategies. We created the pipelined registers using SystemVerilog interfaces and modules for the appropriate pipeline stages. We then worked on splitting up the datapath and controller logic into their respective stages.

The testing strategy for this stage was to compare the outputs of the CPU with the outputs of the CPU from previous MPs. Specifically, MP3 was the main code used for comparison. We ran the provided Assembly test code, as well as past test code, to check for correct functionality.

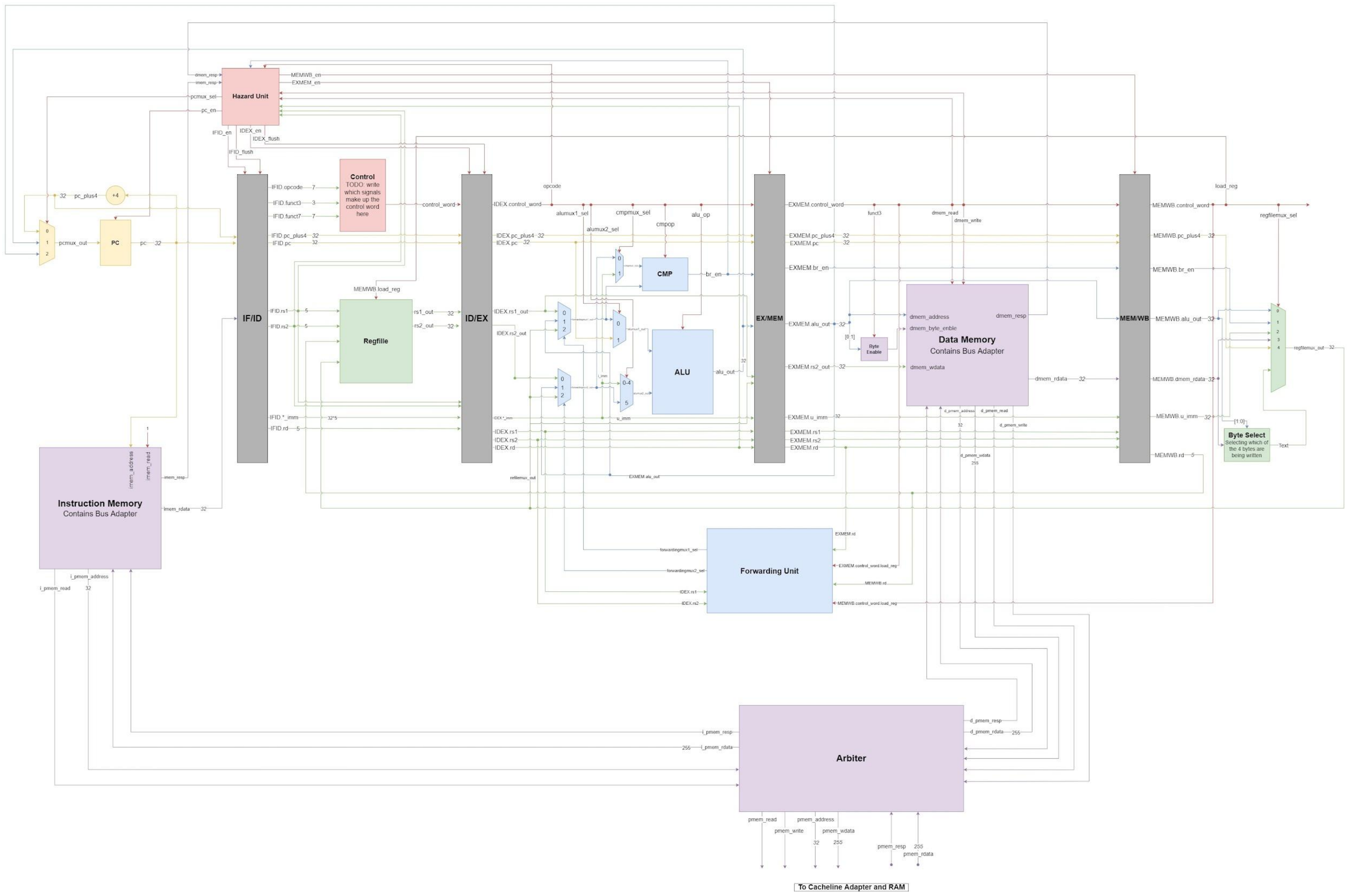
Checkpoint 2

For this milestone, we worked on designing and implementing the forwarding and hazard units for the pipeline, as well as the arbiter and static branch prediction. This milestone was about improving the pipeline to work more efficiently for control flow type instructions and reduce the number of stalls due to them. Without these modules implemented, our CPU would be very slow due to the large number of bubbles that would need to be added for various data dependencies and control hazards. At this stage, we also added static Not-Taken branch prediction which allowed the regular flow of our CPU. Although this caused many branch mispredictions, this was not an issue at this milestone since we had not yet implemented a predictor.

Additionally, we added the connections to the RVFI monitor and Shadow Memory. These were important aspects of our verification because these would help us check our memory accesses and the execution of our CPU. Once these were added and any bugs were fixed, we were more confident in the correct functionality of our design implementation. The provided Assembly test code for this checkpoint, as well as past test code were all used to simulate our design, now with the RVFI Monitor and Shadow Memory modules attached.

Below is the final 5-stage pipeline with forwarding and hazard units, as well as the split I- and D-caches and the arbiter

- The grey blocks are the pipeline registers
- The purple blocks are the memory modules (caches)
- The red (in the top) is the Hazard Unit
- The blue (at the bottom) is the Forwarding Unit


Hazard Unit
dmem_resp
imem_resp
pcmux_sel
pc_en
MEMWB_en
EXMEM_en
IFID_en
IDEX_en
IDEX_flush
IFID_flush
Control
TODO: write which signals make up the control word here
control_word
+4
pc_plus4
32
PC
pc
pcmux_out
0
1
2
IF/ID
IFID.opcode
IFID.funct3
IFID.funct7
IFID.pc_plus4
IFID.pc
IFID.rs1
IFID.rs2
IFID.*_imm
IFID.rd
7
3
5
32*5
Regfille
MEMWB.load_reg
rs1_out
rs2_out
ID/EX
IDEX.control_word
IDEX.pc_plus4
IDEX.pc
IDEX.rs1_out
IDEX.rs2_out
IDEX.*_imm
IDEX.rs1
IDEX.rs2
IDEX.rd
opcode
alumux1_sel
alumux2_sel
cmpmux_sel
cmpop
alu_op
CMP
br_en
ALU
alu_out
u_imm
i_imm
0-4
5
regfilemux_out
EXMEM.alu_out
EX/MEM
EXMEM.control_word
EXMEM.pc_plus4
EXMEM.pc
EXMEM.br_en
EXMEM.alu_out
EXMEM.rs2_out
EXMEM.u_imm
EXMEM.rs1
EXMEM.rs2
EXMEM.rd
funct3
dmem_read
dmem_write
[0:1]
Byte Enable
Data Memory
Contains Bus Adapter
dmem_address
dmem_byte_enble
dmem_wdata
dmem_resp
dmem_rdata
d_pmem_address
d_pmem_read
d_pmem_write
d_pmem_wdata
255
MEM/WB
MEMWB.control_word
MEMWB.pc_plus4
MEMWB.br_en
MEMWB.alu_out
MEMWB.dmem_rdata
MEMWB.u_imm
MEMWB.rd
load_reg
regfilemux_sel
regfilemux_out
[1:0]
Byte Select
Selecting which of the 4 bytes are being written
Text
Instruction Memory
Contains Bus Adapter
imem_address
imem_read
imem_resp
imem_rdata
1
i_pmem_address
i_pmem_read
Forwarding Unit
forwardingmux1_sel
forwardingmux2_sel
IDEX.rs1
IDEX.rs2
EXMEM.rd
EXMEM.control_word.load_reg
MEMWB.rd
MEMWB.control_word.load_reg
Arbiter
i_pmem_resp
i_pmem_rdata
d_pmem_resp
d_pmem_rdata
pmem_read
pmem_write
pmem_address
pmem_wdata
pmem_resp
pmem_rdata
To Cacheline Adapter and RAM

Checkpoint 3

The purpose of the milestone was simply implementing the Advanced Features. The details are written below in the **Advanced Design Options** section.

Checkpoint 4

The purpose of the milestone was to test for performance and timing of our original design. The object was to test our design's execution time, and tweek it in order to make the design more efficient and faster. We would run code that was given to us in the form of a design competition to determine the team in the class with the most efficient design. To quantify our performances, our scores were calculated with the formula of $PD^2 * (100/Fmax)^2$, or energy * $(delay * 100/Fmax)^2$, where 100/Fmax was to normalize our score based on our machine's processor speed.

Advanced Design Options

The five main features we completed were the Branch History Table, the Branch Target Buffer, the RISC-V M-extension, the parametrized cache, and the pipeline cache. We had originally considered implementing the Eviction Write Buffer as well; however, this was incomplete and therefore not in our final design and project.

Branch Prediction

Design: The Predictor that was implemented is a dynamic Local Branch History Table using a 2-bit bimodal predictor along with a corresponding Branch Target Buffer. This type of predictor chooses between taken or not taken for a branch or jump instruction depending on the history of that particular instruction. This is chosen based on a hash function. For this BHT, we implemented it such that it was parameterized for the number of sets. This allowed us to change the size depending on the available resources, power and predictor collisions. The hash function that was used was the lower order bits of the PC (not including the lowest 2 bits) to index into the BHT and BTB.

Additionally, in order to improve the predictor, we implemented a Global History Register as well to prove greater context to the predictor in determining the next result. This was parameterized as well to change the number of bits of history. We then tested a different hash function that used the XOR of the lower bits of the PC with this global history register. However, this type of predictor actually did worse than our basic bimodal predictor so it was not used.

Another improvement that was made was checking for the opcode before using the predictor. Initially, the BHT and BTB were referenced for every instruction, including non branch/jump instructions. Adding a check for those types of instructions helped improve prediction accuracy since the BHT and BTB wouldn't be cluttered with other instructions since those should always be 'not taken'.

The Predictor would help with most types of workloads since about 15-20% of instructions are usually control flow types. Programs with many loops that have a large number of iterations each would benefit the most.

Testing: To test the predictor, I started with a small BHT size of 128 entries and ran all the existing test code through our CPU. I looked for three things. One is that the correct functionality is not hindered and there are no errors reported by the RVFI or the Shadow Memory. I also checked to make sure that the final register values were consistent with what is expected. Lastly, I checked to see if the execution time has decreased, in general. If the predictor is working correctly it should reduce the number of miss-predictions compared to the static prediction. I also performed a manual check of the predictor for a small test case using the waveforms to check that the values of the predictor are updating. Once these criteria were passed for all the test assembly files, we could be confident that the predictor was working. I then tested for sizes of 256 and 512 sets as well. Although 512 or larger did not fit on the FPGA due to resource constraints.

Performance Analysis:
Below are the results of the prediction before and after adding the opcode checking. Here you can see that adding this check improved the predictor.

Metric	**Basic Prediction**			**Checking Instruction Opcode**		
	comp1.s	comp2_m.s	comp3.s	comp1.s	comp2_m.s	comp3.s
Clock Cycles	55,885	27,763	72,665	49,537	26,152	68,835
BR Accuracy	84.0 %	60.8 %	56.6 %	89.3 %	70.5 %	67.5 %
J Accuracy	99.9 %	78.3 %	~100.0 %		86.5 %	
Total Accuracy	**87.8 %**	**69.1 %**	**75.1 %**	**91.9 %**	**77.7 %**	**81.8 %**

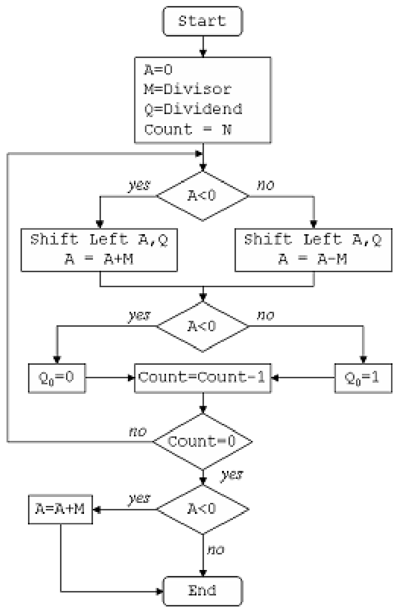
The speedup achieved is between 4.65-12.40% depending on which workload was executed. This is on top of the improvement from not having dynamic prediction at all. Adding the global history caused a slowdown in the total execution time of an estimated 5-10% and so it was not used. This is likely due to the resulting hash function being much more sporadic and random, leading to the same branch instruction's prediction being stored in different indexes of the BHT. Hence there were more miss predictions. Lastly, a large portion of the core power was due to the BTB due to its size and because it used lots of registers that are constantly being clocked. This also limited the size we could use.

To improve this performance further, we would likely need to look into using a tournament predictor, increasing the associativity, and using BRAM.

M-extension

Design: The RISC-V M-extension consists of a multiplier, divider, and the additional control logic associated with them. For the multiplier, we implemented a Wallace Tree multiplier. The multiplier uses the Wallace Tree algorithm which includes 8 stages of single bit full adders and one stage of 64-bit addition. This is significantly faster than the add shift multiplier, which would require 32 stages of 64-bit additions. The multiplier will compute a 64-bit value for the results. Depending on the instruction type, either the upper 32-bit or the lower 32-bit is sent to the pipeline as the final result.

For the divider, we implemented a non-storing divider, which can also calculate the remainder. Again depending on the instruction type, either the quotient or the remainder is sent to the pipeline. The logic for the divider is shown below.



This divider needs 32 stages of 32-bit additions. We wrote each stage into a task, so that we can easily adjust how many stages we want to fit into one clock cycle by changing the times that the task repeats in a clock cycle. For the rest of the control logic, we added a dedicated compute unit parallel to the regular ALU to compute multiplication and division. This unit is initiated by a start signal from the control unit, and outputs a done signal when the results are ready. The hazard unit will stall the pipeline using these two signals when the results are not ready.

Testing: We wrote two dedicated testbenches for the multiplier and divider. The testbenches loop through integers in a certain range and send them to the multiplier or divider as inputs, and then compares the outputs to the numbers obtained by using the built in “*” and “/” functions. We also manually tested some edge cases to make sure that they work as expected. Once we ensured that the multiplier and the divider are both working properly, we integrated them into the pipeline along with the additional control logic.

We used the given comp2-m.s test codes along with the RVFI monitor to do the final testing and debugging. After we made sure that the entire M-extension worked correctly, we also tested the timings and found out some problems. Originally we implemented the multiplier such that it only takes one clock cycle to compute. However, we could only get about 50MHz that way, which is only half of what we need. Therefore, we added some registers in the multiplier to save the intermediate results and made it into two cycles. Also, we realized that for the divider we could fit four stages of the task described earlier into one clock cycle without impacting the timings. In the end, we achieved a 2-cycle multiplication and 8-cycle division.

Performance Analysis: Because this feature adds the capability of direct multiplication and division to our CPU, we can expect that any programs that require multiplication and division would be significantly faster. However, programs that do not use multiplication and division may be slightly slower due to the slightly longer critical path causing lower frequencies. Below are the results comparing comp2-i.s to comp2-m.s.

Metrics	comp2_i.s	comp2_m.s
Clock Cycles	119,570	25,924
Execution Time	1,195,755 ns	259,295 ns
# of Instructions	116,080	11,082
Cycles per Instruction	**1.03**	**2.34**

Here we can see that even though the CPI is slightly longer since it takes longer to do multiplication or division instructions than regular instructions, but because of the drastically reduced number of total instructions, the same program runs 4.6 times faster with the M-extension.

Parametrized cache

Design: We started on the parametrized cache by modifying the cache we wrote in MP3. We parameterized the cacheline size, number of index bits (number of sets), and number of ways. These parameters can be separately adjusted for the icache and dcache in the top level module. We used the generate block in SystemVerilog to generate the number of modules defined in the parameters. As for the LRU, we implemented a parameterized pseudo LRU, which uses (Number of Ways - 1) bits for each set.

Testing: The testing strategy we used for the parameterized cache is simply trying to run the given test codes with different cache configurations. We can then verify the results based on the RVFI monitor and the regfile values.

Performance Analysis: In general, we expect that larger cache would improve performance when there are frequent reads and writes across a wide range of memory addresses. It mostly shouldn't impact performance for programs with minimal read and write operations except that larger caches may lead to longer critical paths, but we did not observe this for the cache sizes we tested. We tested the cache for up to 64 sets, 4 ways. However, we did not see any performance improvements above 32 sets for all three of the competition test codes. Below are the results for some of the cache configurations.

Metric	**8 Sets, 2 Ways**			**16 Sets, 4 Ways**			**32 Sets, 4 Ways**		
	comp1.s	comp2_i.s	comp3.s	comp1.s	comp2_i.s	comp3.s	comp1.s	comp2_i.s	comp3.s
Clock Cycles	50,377	385,383	341,533	50,057	237,319	72,665	50,057	119,570	72,665
I-Cache Hit Rates	99.9%	91.0%	82.8%	99.9%	95.6%	99.9%	99.9%	99.9%	99.9%
D-Cache Hit Rates	99.7%	98.9%	89.6%	99.7 %	99.8%	95.2%	99.7%	99.5%	95.2%

As we can see in the table, the performance generally would increase as we increase the cache sizes. At 32 sets, 4 ways, we can see that the cache hit rates are almost 100%, so that is probably why we don't see any performance improvement beyond that. Comparing to the original 8 sets, 2 ways cache, the 32 sets, 4 ways provides up to 4.7 times performance improvement in comp3.s and up to 3.2 times in comp2_i.s.

<u>Pipelined cache</u>

Design: In our cache from MP3, the cache would take two cycles to respond if there is a hit, which means the CPU has to stall for at least one cycle for every instruction. The solution is to pipeline the cache such that as the cache is responding to the previous request, it can accept a new request at the same time. Therefore, when there are consecutive cache hits, the CPU doesn't need to stall. In order to achieve this, we have to modify both the cache and the CPU datapath. In the cache, we need to have new registers to hold information for the previous request so that they wouldn't get overwritten by the new request. Specifically, for cache read we need to save the memory address, and for cache write we need to save both the address and the write data. In the cache state machine, we changed the next state logic such that if there is a hit, the cache would stay in the "LOOKUP" state, in which the cache would check for new hits with the new address.

Even though now the cache can receive requests in consecutive cycles, each request would still take two cycles to complete. In order to not stall the CPU, the cache needs to receive the request "early". Therefore, we modified the CPU datapath and let the cache get the request data before it latches into the registers (PC register for icache, EXMEM register for dcache). In this way, the CPU doesn't need to stall to wait for the cache when there are consecutive cache hits.

Testing: The testing strategy for the pipelined cache is also simply running the given test codes with the pipelined cache and checking the modelsim waveforms. Specifically, we checked the imem_resp and dmem_resp signal for consecutive cache hits. They should be always high when there are consecutive cache hits. Also, we checked the address and the data to make sure that the cache is receiving the correct request and responding with the correct values. Last but not least, we checked the regfiles values of each test code the programs run correctly overall.

Performance Analysis: We did not run the competition test codes before we implemented the pipelined cache, so we do not have any data to compare with. However, it is very easy to estimate the performance improvement. The pipelined cache simply allows cache hits to complete in a single cycle instead of two cycles. In other words, the execution time for cache hits is cut in half. Therefore, we can always expect a performance improvement unless the program doesn't access anything in the cache. To calculate the improvement, we can just take the cache hit rates and divide it by 2, then we can get the execution time reduction percentage. Obviously this would vary with different cache sizes and different programs, but as we can see in the previous table, the cache hit rates are almost 100% with a large cache. Thus we can safely estimate that the pipelined cache provides 50% reduction in execution time or a two times performance improvement.

Additional Observations

The last remaining advanced feature that was attempted was the Eviction Write Buffer. Although not fully implemented and thus left off of the final design, we can still discuss some of the details of how the design would have been.

Within our normal design, a dirty eviction would result in the evicted block being written to the next cache level, with the missed address then being fetched. This buffer would instead let the block be written in such a way that the missed address could be fetched first, and then write the block to the next level only when the level is free. The CPU thus gets the missed data much faster instead of waiting because the block needed to be written first. Design wise, the address in and address out are 12-bit values that correspond to the location of the data blocks, which themselves hold a 128 bit value. These values are stored in a temporary buffer that only becomes permanent when the next cache level is ready, instead of making the CPU wait until the next level would become ready.

Conclusion

Our original goal was to create a 5-stage pipelined CPU with some advanced features that would be designed with performance in mind. We overall succeeded in our goal, adding branch history and cache edits to help increase the performance of the processor. Implementing the M-extension originally made the design slower (although it made actual multiplication much, MUCH faster), until we chose to edit the design to be a Wallace tree that used a temporary register; the moral of the story here is that two different designs can have similar functionality but far different execution times. The project was an overall success as we managed to meet our goals of balancing performance and capabilities of the CPU.