ECE 411

Spring 2020

MP3 Final Report

RISC-V Pipelined Processor

TJ Roberts, Arkin Shah, & Yohanes Setiawan

TA: Canlin Zhang

Introduction:

For our final project, MP3 of the course ECE411, we were required to make a pipelined RISC-V processor that is able to execute the basic instruction set (with the exception of FENCE, ECALL, EBREAK and CSRR). The goal was to design and test it using SystemVerilog, such that it synthesises on the Altera FPGA. With the completion of the basic core processor, we were then challenged to implement advanced features that would help make the core execute more efficiently. Such features would go hand in hand with our implementation of both L1 and L2 caches to further speed up the performance.

A pipelined system would ensure that the amount of data throughput is maximised as each 'section' processes data in order, not waiting for the completion of one instruction, but rather multiple instructions together. The more sections there are to a pipeline, the more instructions it can process. However, the only drawback is that if there are loops, it can take the incorrect branch and start executing invalid data, and also sometimes instructions are dependent on calculated data from previous instructions which are not always ready in time. Thus, our implementation of 5 sections (fetch, decode, execute, memory, writeback) also included data forwarding logic and branch predictor. The former forwards data ahead to avoid the issue caused by data not being ready, while the latter takes a chance and predicts which branch to take considering data correctness is the most important thing when implementing efficiency -- we would rather have a slow but correct system, than a fast but incorrect one.

Project Overview:

Due to the pipelined nature of the project, we started from scratch with only key principles and basic modules from MP2 such as registers and arrays and such. We divided the work into two halves, the core and the memory access (caches, arbiter and cacheline adaptor). With 2 people assigned to develop the core and one for the memory access. This would ensure cohesiveness of the project considering the fact that the core should be unaware of the fact that it is attached to caches, as far as the core is concerned, it interacts with the physical memory directly. This also sped up our progress. The core itself is divided into 5 sections as mentioned above, each with an intermediate register in between that holds the data before it is passed onto the next section. We also have a control word that is passed on from the decode section onward to the rest of the pipeline; this control word holds all the necessary information required for the execution and storage of the data, it is designed as a packed struct, this allowed a certain level of abstraction when it was passed on from one section to the next as the separate components can still be accessed, while keeping the word itself, intact. Besides that, the core also included data forwarding using type-defined packages for mux selections that determined where the data would flow to next. Furthermore, to make testing easier, we were also provided with an interface to an RVFI monitor, which allows us to make note of data values and mismatches as execution progresses.

We also implemented an instruction cache and a data cache, with the former connected to the core in the fetch section, and the latter connected in the memory section. These two form the split L1 cache for quick access -- it was designed to be a 2-way cache with each way possessing 8 sets, with each set containing 32 Bytes of data, thus collecting up to 512B for each L1 cache, it utilised the least recently used (LRU) policy to select which way to store the data in. This brings us to the next part - the unified L2 cache, whose specifications depend on which version of our project you are looking at. All versions of our project before checkpoint 4 implemented a 4-way, 8-sets per way, 32B per set L2 cache that utilised the pseudo-LRU (pLRU). However, for our final submission we had to drop it down to 2 ways, 16-sets per way, 32B per set with an LRU policy since we could not figure out why our processor did not work with the pLRU for that specific checkpoint; possibly an edgecase that we could not find on time. Either way, the final size of the L2 was 1KB, which is the summation of the cumulative sizes of the split L1. To connect the split L1 caches with the L2, we also implemented an arbiter that would decide which of the L1's gets to access the L2 and thus the physical memory. Preference was given to the data cache, since we figured that data needs to be accessed first to continue execution. Finally our L2 interacts with the physical memory using a cacheline adaptor that we designed in the MP's prior to this one. Since the caches function in cachelines (each of which is 32B long) and the physical memory only accepts 64 bits at a time, the cacheline adaptor would send bursts of 64 bits to from the L2 to it. Alternatively, it would also convert the bursts from the physical memory to one single burst of 256 bits to send up to the L2.

Our advanced features included designing the 4-way pLRU cache (which was later changed and thus attempted but omitted), an array multiplier (which was attempted but also faced issues in the final submission) and data prefetching (which works and still exists). While the former two were omitted, the final one remains and is still functional and can be found in the arbiter.

Milestones:

I. Checkpoint 1

What did we implement this checkpoint?

We implemented a partial RV32I pipelined datapath for this checkpoint. It handles all the required instructions, without taking into account potential hazards that might occur. The only memory interface it has an interaction with is the given magic memory model. It is not a cache, just a highly idealized memory model. We tested our design using the provided test code for this checkpoint.

What did each team member contribute?

The work was split such that one member created the modules for each stage and intermediate stage register. Another member then designed the top level module connecting all the stages and stage registers. He also set up our top level in the provided testbench. Another completed the control ROM and control word registers. All team members contributed towards debugging the design.

What major problems did we encounter and how did we solve them?

A few major challenges that our team encountered was definitely the fact that we are not able to meet up physically, due to pandemic situations — this took from us the required interactions to teach and learn from each other and also come up with better plans to carry out the project. Other challenges included figuring out what the stage registers actually do, because initially we could not grasp the idea of just passing a control word from one stage to the next and then disassembling that word into the various signals so as to use them in the stage itself. Furthermore, we also struggled with documentation about connecting our datapath to the magic memory because we were not entirely sure how memory interfaces worked in the given files. We also struggled with the correctness of data as well as we compared our code to the results from the auto-grader. We solved most of these issues either by discussing on our group chat or by extensive reading of various sources on the internet. When all else failed, we also resorted to trial and error by going through signals in ModelSim. ModelSim also gave us a lot of trouble as we came across errors that waves could not be added to the simulation. Furthermore, we could not figure out how to make the system halt — we finally figured out all of these after talking with our mentor TA and also going through trial and error. We think that more documentation should be provided about top.sv in the future, to avoid these random problems.

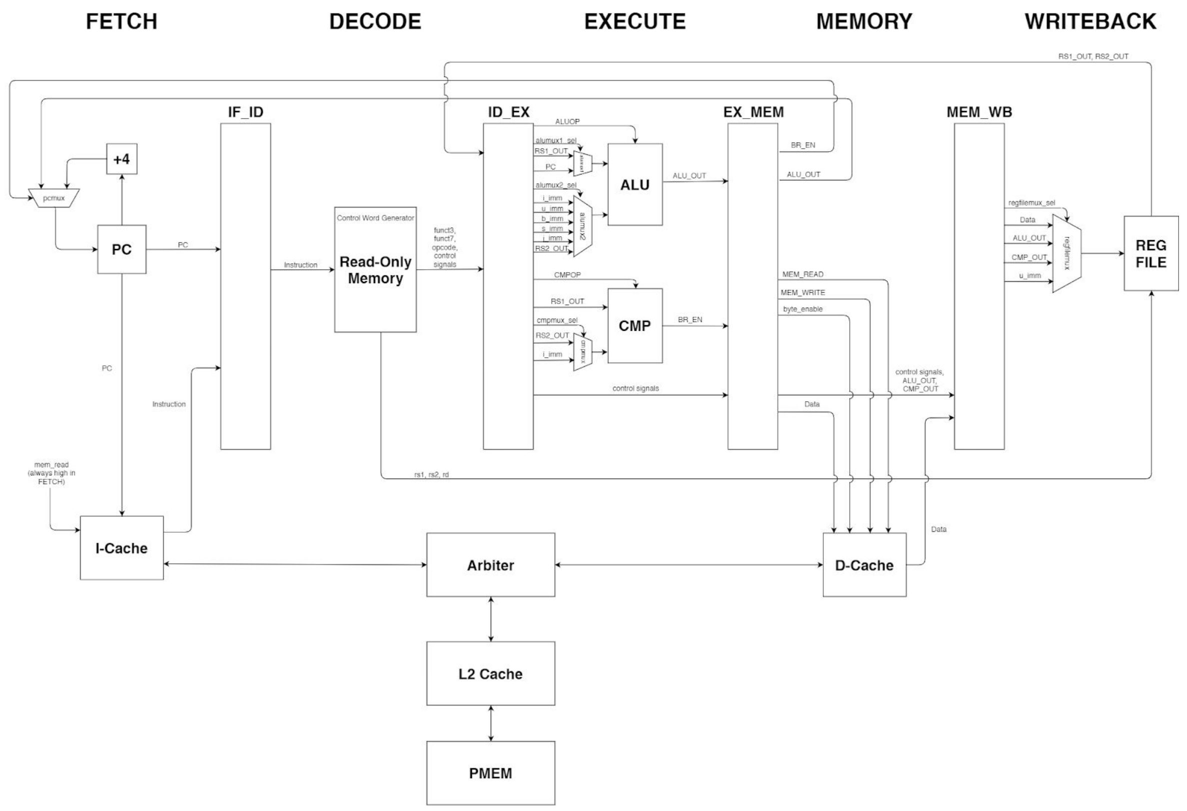

Figure 1: Checkpoint 1 datapath diagram for pipelined CPU

Name	Type	Stage	Origin	Destination
pc_out	rv32i_word	FETCH	PC	IF_ID
pcmux_out	rv32i_word	FETCH	pcmux	PC
pcmux_sel	pcmux_sel_t	FETCH	EX_MEM	pcmux
instr_addr	logic [31:0]	FETCH	PC	I-Cache
instr_rdata	logic [31:0]	FETCH	I-Cache	IF_ID
instr_read	logic	FETCH	Set to 1'b1	I-Cache
instr_byte_enable	logic [3:0]	FETCH	Set to 4'b1111	I-Cache
mem_iresp	logic	FETCH	I-Cache	IF_ID
rom_instr	logic [31:0]	DECODE	IF_ID	ROM
ctrl	rv32i_control_word	DECODE	ROM	ID_EX
rs1	rv32i_reg	DECODE	ROM	REGFILE
rs2	rv32i_reg	DECODE	ROM	REGFILE
rd	rv32i_reg	DECODE	ROM	REGFILE
rs1_out	rv32i_word	DECODE	REGFILE	ID_EX
rs2_out	rv32i_word	DECODE	REGFILE	ID_EX
aluop	alu_ops	EXECUTE	ID_EX	ALU
cmpop	branch_funct3_t	EXECUTE	ID_EX	CMP
alumux1_sel	alumux1_sel_t	EXECUTE	ID_EX	alumux1
alumux2_sel	alumux2_sel_t	EXECUTE	ID_EX	alumux2
cmpmux_sel	cmpmux_sel_t	EXECUTE	ID_EX	cmpmux
alumux1_out	rv32i_word	EXECUTE	alumux1	ALU
alumux2_out	rv32i_word	EXECUTE	alumux2	ALU
cmpmux_out	rv32i_word	EXECUTE	cmpmux	CMP

i_imm	rv32i_word	EXECUTE	ID_EX	alumux2, cmpmux
u_imm	rv32i_word	EXECUTE	ID_EX	alumux2
b_imm	rv32i_word	EXECUTE	ID_EX	alumux2
s_imm	rv32i_word	EXECUTE	ID_EX	alumux2
j_imm	rv32i_word	EXECUTE	ID_EX	alumux2
pc_out	rv32i_word	EXECUTE	ID_EX	alumux1
rs1_out	rv32i_word	EXECUTE	ID_EX	alumux1, CMP
rs2_out	rv32i_word	EXECUTE	ID_EX	alumux2, cmpmux
ctrl	rv32i_control_word	EXECUTE	ID_EX	EX_MEM
alu_out	rv32i_word	EXECUTE	ALU	EX_MEM
br_en	rv32i_word	EXECUTE	CMP	EX_MEM
alu_out	rv32i_word	MEMORY	EX_MEM	ID_EX, MEM_WB
cmp_out	rv32i_word	MEMORY	EX_MEM	MEM_WB
br_en	rv32i_word	MEMORY	EX_MEM	ID_EX
mem_read	logic	MEMORY	EX_MEM	D-Cache
mem_write	logic	MEMORY	EX_MEM	D-Cache
mem_rdata	rv32i_word	MEMORY	EX_MEM	D-Cache
mem_wdata	rv32i_word	MEMORY	D-Cache	MEM_WB
mem_byte_enable	logic [3:0]	MEMORY	EX_MEM	D-Cache
mem_dresp	logic	MEMORY	D-Cache	MEM_WB
ctrl	rv32i_control_word	MEMORY	EX_MEM	MEM_WB
regfilemux_sel	regfilemux_sel_t	WRITEBACK	MEM_WB	regfilemux
regfilemux_out	rv32i_word	WRITEBACK	regfilemux	REGFILE
alu_out	rv32i_word	WRITEBACK	MEM_WB	regfilemux

cmp_out	rv32i_word	WRITEBACK	MEM_WB	regfilemux
u_imm	rv32i_word	WRITEBACK	MEM_WB	regfilemux
mem_data	rv32i_word	WRITEBACK	MEM_WB	regfilemux
rs1_out	rv32i_word	WRITEBACK	REGFILE	ID_EX
rs2_out	rv32i_word	WRITEBACK	REGFILE	ID_EX

Figure 2: Checkpoint 1 datapath signals for pipelined CPU

II. Checkpoint 2

What did we implement this checkpoint?

This checkpoint we implemented a one cycle hit, split, 2-way L1 cache into our pipelined datapath design. We created an arbiter to control the signals from each half of the split cache when a miss occurs. We also modified our datapath to stall until instructions or data has been received from memory. Finally, we finished the functionality of the remaining RV32I load and store operations. Our cache design was thoroughly tested from MP2. We only made minor adjustments to allow for single cycle hits to be functional. Any further debugging was done using Modelsim and the provided test code.

What did each team member contribute?

The work was divided such that one member focused on incorporating the split L1 cache, arbiter, and cacheline adaptor into our design. He went off his own MP2 cache design because we decided his was the best. Another member added the needed functionality for the remaining load and store operations, also using his MP2 design as reference. Another edited the pipelined data path to be cleaner and have less superfluous signals related to the control word. He also added the logic to stall the pipeline when a cache miss occurs.

What major problems did we encounter and how did we solve them?

The first major hurdle for us was converting the MP2 cache design into a one cycle hit cache. We realized that the separate idle state was unnecessary, and we could treat the hit detect state as a pseudo idle state. We ran into additional problems regarding this later since we incorrectly adjusted the next state logic. Initially, all we knew was that the caches were sending response signals at incorrect times. Using Modelsim, we traced the signals from pmem and found no issues. We traced the signals from the cacheline adaptor and found no issues. We traced the signals from the arbiter and found the issue but determined that the arbiter wasn’t the source. That left the cache control signals, where we discovered it would progress to the next state even if read and write were low. By adding additional if statements we fixed this issue. The rest of the major issues came near what we thought was the end at the time. Our code was compiling and giving no errors/warnings in Modelsim and our register values looked good throughout execution. However, the design wasn't passing the AG. Being that we lacked any error information in Modelsim, we needed to use the AG log and Piazza to debug. We found that we had overlooked that the AG sees every cycle where a cache response is as high as a new instruction or new data. It was our fault for not fully reading the Piazza post, but it was still frustrating to realize the issue technically had nothing to do with our actual design. For the future, AG functionality like this should be in the documentation on GitHub not on a Piazza post.

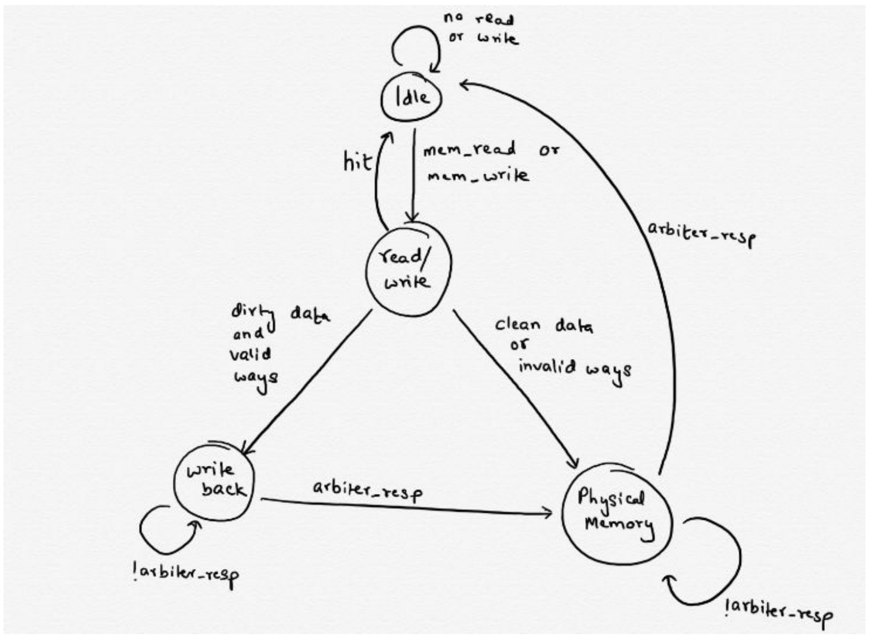

Figure 3: L1 cache state diagram

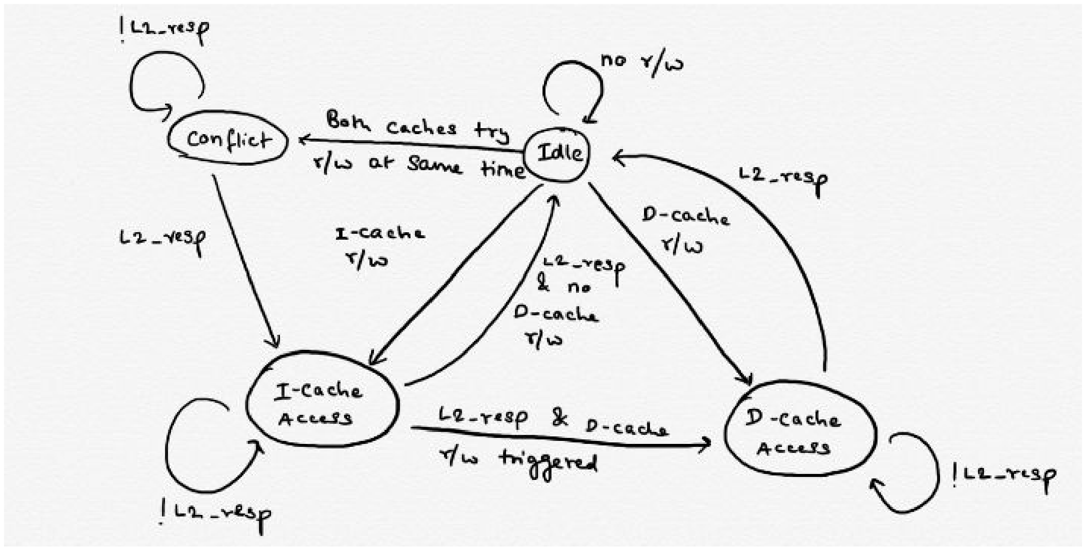

Figure 4: Arbiter state diagram

III. Checkpoint 3

What did we implement this checkpoint?

In this checkpoint, the main features that we implemented were hazard detection, data forwarding and the L2 cache. With those features, our pipeline can now properly detect if an incoming instruction is dependent on an earlier instruction's results and forward the required operands accordingly, reducing stall time and increasing performance. We also implemented a static branch-not-taken for control hazards. In accordance with our roadmap, we decided to implement the Pseudo-LRU in our L2 cache earlier in this checkpoint to reduce workload in checkpoint 4 and eliminate the need to replace the LRU system for the L2 cache. To test forwarding and branch prediction, we used all the given test code including the ones from previous checkpoints. We removed the NOP instructions and looked in Modelsim for any incorrect register values.

What did each team member contribute?

The group member who implemented the L1 cache and arbiter for checkpoint 2 focused on implementing the L2 cache for checkpoint 3 because he would naturally know the best on how to integrate his L1 cache with his L2 cache. The group member who designed the data forwarding and hazard detection implementation for checkpoint 3, worked to implement those features into our pipeline because he understood his design the best. Meanwhile as for the final group member, he was tasked to plan out the advanced design features for checkpoint 4 and how best to implement them while helping debug the features we introduced in checkpoint 3.

What major problems did we encounter and how did we solve them?

Fortunately for us, we did not encounter any major problems during the implementation of data forwarding and hazard detection as TJ was able to add those features into our pipeline without much problem even though we expected it to be the most difficult part of the checkpoint. However, a major hurdle that we encountered was during the implementation of the L2 cache as Arkin encountered shadow memory errors in the course of testing the L2 cache. Even with some debugging, we were eventually stuck on a bug in which only half of the correct data cacheline was being sent from the L2 cache into the arbiter and it was most noticeable when later instructions that are supposed to be read and executed simply were not detected. Using ModelSim, we narrowed down the cause to a problem with our pLRU in which the load LRU signals were incorrectly asserted. Fixing that allowed the L2 cache with pLRU to work correctly.
Another problem we encountered was during the implementation of static branch prediction after data forwarding and hazard detection. While no data mismatch or autograder errors popped up,

we noticed that our registers have incorrect values in the memory stage when we ran CP3 test code with no NOP instructions. Closer scrutiny of the code and results using ModelSim allowed us to trace the error to the forwarded values not being where they are supposed to be and in turn allowed us to notice that our implementation of stall from CP2 was incorrect all along. A quick fix to our stall function and we managed to pass CP3 as a result.

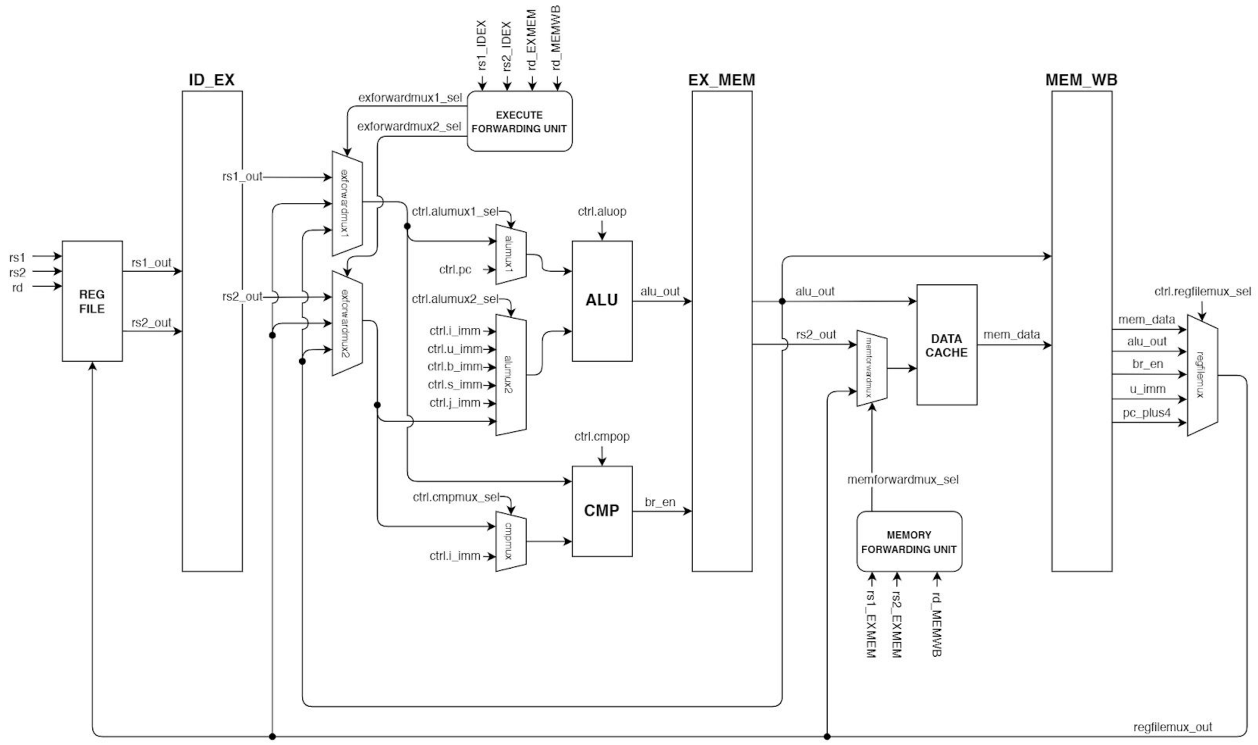

Figure 5: Checkpoint 3 datapath diagram for forwarding

Advanced Design Features:

Feature 1: Pseudo LRU

A. Design

Our initial design of the L2 cache included a 4-way pLRU approach since increasing associativity would ensure that there are more hits and fewer writebacks. We implemented this algorithm using a 3-bit logic tree, where if there are 4 ways named ABCD, the first bit chooses between AB/CD, the second chooses between A/B and the third between C/D. Here, a 0 indicates the term on the left, while a 1 indicates choosing the way on the right. Once chosen, that bit is set. The least recently used is then calculated by calculating the bitwise-NOT of the 3-bit variable. This form is way faster than a true LRU since that would require some sort of a search function, which in turn would increase the complexity of the project and affect both the critical path and synthesis time. This pLRU module interacted with an 8 set array that stores the 3-bit variable outputted for that specific set.

B. Testing

To test the code, we used ModelSim and the waveforms to check how the 3-bit values inside the array changed as we ran the checkpoint test code. To further narrow down our testing procedures, we also wrote our own test code and analysed the values in the array. We came across an interesting bug that bothered us for many hours -- the L1 set index changed while the cache was still trying to fetch data from the L2 or physical memory, thus storing the data in the incorrect place and returning the wrong data to the core. This was because the core was not stalling execution correctly on a cache miss. Fixing the stall logic got rid of the bug, now the entire pipeline halts in the case of a cache miss. However, then we came across other shadow memory errors in checkpoint 4 which are still a mystery to us.

C. Performance Analysis

Oddly enough, not including the pLRU in our final design helped with performance not only due to correctness, but also because the system actually executed faster without the pLRU. We tested this by checking the time at which the first shadow memory error occurred. With the pLRU, it occurs at 797105ns. We then forced the same error with the 2-way 16-set L2 (LRU) cache and it occurred at 660695ns. This completely baffled us. This implied at 1.206x speed up if the pLRU was removed! We are unsure as to why this happened but we suspect that this was likely due to an increased number of writebacks to the physical memory since the rate of cache misses with the pLRU was very high. This is also what our mentor TA told us would happen, since he too tried to implement this previously but found that a random LRU served his project better. This definitely opened our eyes to a scope of improvement, to therefore either increase to more associativity or implement a random LRU the next time we try to make such a system more efficient.

Feature 2: Prefetching

A. Design

We specifically chose to employ basic hardware prefetching in the form of one block lookahead (OBL). Initially, we were going to house the design in the L2 cache. After further thought, we decided that the arbiter would be a better place to prefetch. This way we could control prefetching for instructions and data separately. We did so by only enabling prefetching for instruction misses. Our thinking was that data prefetching would not be as effective since it is much less likely that data from the next block would be useful, and only serve to pollute our L2 cache. Instruction blocks are almost always executed sequentially. Even when they are not the instruction block will be needed eventually (hopefully before being evicted). Our implementation involved a register to hold the address of the instruction miss. This address was incremented by 0x20 in order to address the next block. A prefetch state in the arbiter is reached after the completion of an instruction fetch from L2/PMEM. This state is extremely similar to how the arbiter handles other instruction misses. The main differences are that the incremented address is used, read is set high by the arbiter, and the response signal stops at the arbiter instead of being forwarded to the L1 cache.

B. Testing

To test prefetching we used Modelsim to keep track of the arbiter states as well as the signals being sent from the arbiter to the L2 cache. We could clearly see that during the prefetch state a read signal was being sent along with an address 0x20 greater than the previous instruction access. We confirmed that from the L1 caches' perspective nothing changed. If a miss occurs while the arbiter is prefetching the pipeline correctly stalls and L1 waits until a response signal is received. Once the prefetching is complete, the arbiter can process any queued memory accesses from L1.

C. Performance Analysis

By comparing execution times with and without the prefetching enabled we were able to determine the average speedup provided. Unfortunately, it was quite small at 1.004 times faster with prefetching versus without prefetching. This improvement is likely so small because of how basic our prefetching is. We theorize that more aggressive prefetching such as looking two or three blocks ahead would slightly improve performance. We would have to be careful not to pollute our L2 cache. Eventually, more aggressive prefetching would slow down performance if the prefetched instructions aren't used before getting evicted. Additionally, prefetching will perform better when running code with fewer jumps and branches. This guarantees that prefetched instructions will be used before being evicted by data or other instructions.

Feature 3: Array Multiplier

A. Design

The design for the array multiplier for our pipeline was relatively simple. During the Execute stage of the pipeline, the multiplicand and the multiplier operands, represented by the 32-bit contents of the rs1 register and rs2 register, would be passed into the multiplier module. Inside this module, the multiplier would perform a separate AND operation using each bit of rs1 output from the least significant bit to the most significant bit on every bit of rs2 output that are then shifted to the left by the bit position of the rs1 bit in question, creating 32 64-bit partial products. If rs2 output is to be treated as a signed value by the opcode, then each of these partial products will then be sign extended accordingly. All of these partial products are then added together in order to obtain the 64-bit product of rs1 register output and rs2 register output, however if rs1 output is be treated as a signed value by the opcode, then the partial product corresponding to the most significant bit of rs1 output would instead be subtracted from the final product instead of being added into it. Once all of this is complete, either the lower 32-bit or the upper 32-bit product is loaded into the destination register rd in the regfile depending on the opcode.

B. Testing

To test the code, the multiplier is made to run through all the possible multiplication operations with MUL, MULH, MULHU and MULHSU all being tested. As for the test values, we made sure to test that the signed multiplication operations worked by inputting both negative and positive binary number test cases for each of the signed operations to make sure that the multiplier does work as intended in all possible cases. Using ModelSim to check the inputs and outputs of the array multiplier while cross-checking with the output values from an actual calculator, we verified that the MUL operation does work as intended and so did MULHU operation. However MULH and MULHSU did not completely work as the algorithm we used started outputting incorrect values for test cases where a negative number is multiplied by a positive number and vice versa. Unfortunately due to a lack of time, we were unable to determine the cause of this bug though we theorize it might be caused by an improper addition of partial products for those cases in particular

C. Performance Analysis

Unfortunately, performance analysis for the array multiplier is inconclusive because we were unable to debug and complete it by the end of MP3. It is likely that the multiplier would have caused power and fitting issues due to the large number of AND and adder operations in the implementation of the array multipliers. Another possible problem is how the multiplier uses mainly combinational circuits in its implementation which likely would have taxed the design even more. In hindsight, implementing the array multiplier using a flip-flop derived design could have optimized performance better.

Additional Observations:

Over the course of MP3, a recurring problem that we had was the process of debugging and verification of our design. Throughout the checkpoints, we spent comparatively far more time trying to identify the bugs and investigating the cause of the incorrect output values produced by our pipeline design as opposed to implementing and planning out our design. While we still managed to complete checkpoint 1 and 2 comfortably, this issue became prominent in checkpoint 3 and 4. We ended up spending far more time struggling to identify the bugs in our code, time that we could have used to implement advanced technical features for checkpoint 4 or optimize our code for checkpoint. As a result of this, we were unable to implement our planned array multiplier and also fit our design with the power and timing constraint in particular.

From this, there were three lessons we learned from working on our MP3. One, we should have allocated far more time for MP3 and started on implementing our design for each checkpoint earlier to allow for ample amounts of time for debug and verification process. Another lesson that we learned is that we should have better utilized the resources that were available, in particular by asking the course staff and the other students more frequently on Piazza or during office hours, to potentially save us more time during the verification process of our design. And finally, we should have planned out our pipeline design with optimization in mind and avoid redundant designs as much as possible so that we could fit our design within the power and timing constraints. By the time of checkpoint 4 and checkpoint 5, it was difficult to optimize our mostly completed design and we ran out of time trying to even downsize our design.

Conclusion:

In conclusion, the final project was a success in the grand scheme of things as we learned how to utilise the many tools available to create a pipelined processor that synthesises on the FPGA, along with learning the priceless skill of debugging such a system and most importantly verifying its correctness. Verification is probably the most important part of such a project, and while the three of us were not familiar with it at the beginning, after countless hours, we now know a little more than we did before. On a final note, our team designed this during the COVID-19 pandemic in the year 2020 A.D., while the entire planet was still undergoing many transitions, hence communication was probably the biggest obstacle in our course. Considering all the progress we flushed out purely by working remotely, we got an opportunity to strengthen perhaps the most important aspect of being an engineer - teamwork and collaboration - even though it was virtual and we were hundreds of miles apart.