Vishnu Jaganathan
Anirudh Kashyap
Ethan John

AniRISC-V Final Project

Introduction

This report details the design and implementation of a pipelined processor that uses a subset of the RV32I Instruction State Architecture as a final project in ECE 411: Computer Organization and Design at the University of Illinois at Urbana – Champaign. This project was meant to build off previous labs in which a non-pipelined processor and L1 cache were implemented as well as concepts taught during lecture. We designed and implemented a five stage pipelined processor, also adding performance enhancing features including a branch predictor, stride prefetcher, eviction write buffer and an L2 cache. This provided us an open-ended way to learn hardware design and verification and so this was an adequate way to conclude the course. The rest of this report will detail the milestones, design, features, and manner by which the AniRISC-V pipelined processor was created.

Overview

This project is a main intrigue of the ECE411 course, and is customarily taken by students in their last few semesters of undergraduate study. It requires a group, similar to many other of the infamous classes in the ECE curriculum (The namesake of AniRISC-V stems from an AniRUX group name used in ECE391). Upon starting the project, our goal was to first and foremost create a processor that worked seamlessly without any bugs/errors, and from there add features that either increased performance or involved implementing an interesting topic discussed during lecture. One of our main standards that was kept for the duration of the project was putting dedicated time and effort into an initial discussion and paper design of whatever needed to be implemented in order to minimize errors and deliberation whilst coding the design. Due to the COVID-19 pandemic group members were not able to meet, so each milestone started with a video call in which a working design (to our knowledge at the time) was put to paper, and coding responsibilities were assigned based on that design.

Milestones

CP1

The progress and work on the project was split into three checkpoints, the first of which was implementing a basic processor that could handle all the required RV32I instructions, and execute instructions in a pipelined fashion given that there were no hazards.

AniRISC-V's pipeline consisted of five stages (Fetch, Decode, Execute, Memory, Write back). In the Fetch stage, the current instruction was fetched from memory and the next program counter value was calculated. In the Decode stage, the registers that were going to be used were retrieved, and a control word- a struct that consisted of all the control signals for that instruction- was assigned based on the instruction and passed down the pipeline. The Execute stage was where the actual computation occurs and the functional units like the ALU and CMP are used. In the Memory Stage, any load or store operation would use the appropriate mem_address and write data to read or write from memory. And finally in the Writeback Stage, the appropriate data is

selected written to the destination register in the instruction. Buffers or registers were placed in between all the stages in the pipeline to store the operand values, results to be committed to the register file/sent to memory, control signals etc.

The hardware to be used in each stage was taken from a part of the non – pipelined processor that was implemented earlier in the semester. Five stages meant that a maximum of five instructions could be running simultaneously in each stage. Control signals for datapath elements were taken from the instruction's generated control word that is passed down the pipeline. Here is a table and diagram detailing the datapath components in each stage:

Stage	**Datapath Elements**
Instruction Fetch (IF)	PC, memory signals to instruction memory
Instruction Decode (ID)	Register array, logic for generating control word
Execute (EX)	ALU, CMP, multiplexerses for ALU, CMP
Memory (MEM)	Memory signals to data memory
Writeback (WB)	Regfilemux (multiplexers for selecting which result is to be loaded to the register array)

The majority of the work for this checkpoint was ensuring that the implementation of the control word worked properly and that each stage was using the right signals based on the instruction that was in that stage.

Here is a table with the control words for each instruction and all the assigned signals for each stage

Instructions:
(if signals not mentioned they are default 0)

RV32I Instructions	Execute	Memory	Writeback
LUI	N/A	N/A	Regfilemux_sel = u_imm Load_regfile = 1
AUIPC	Aluop = alu_add Alumux1_sel =	N/A	Load_regfile = 1 Regfilemux_sel =

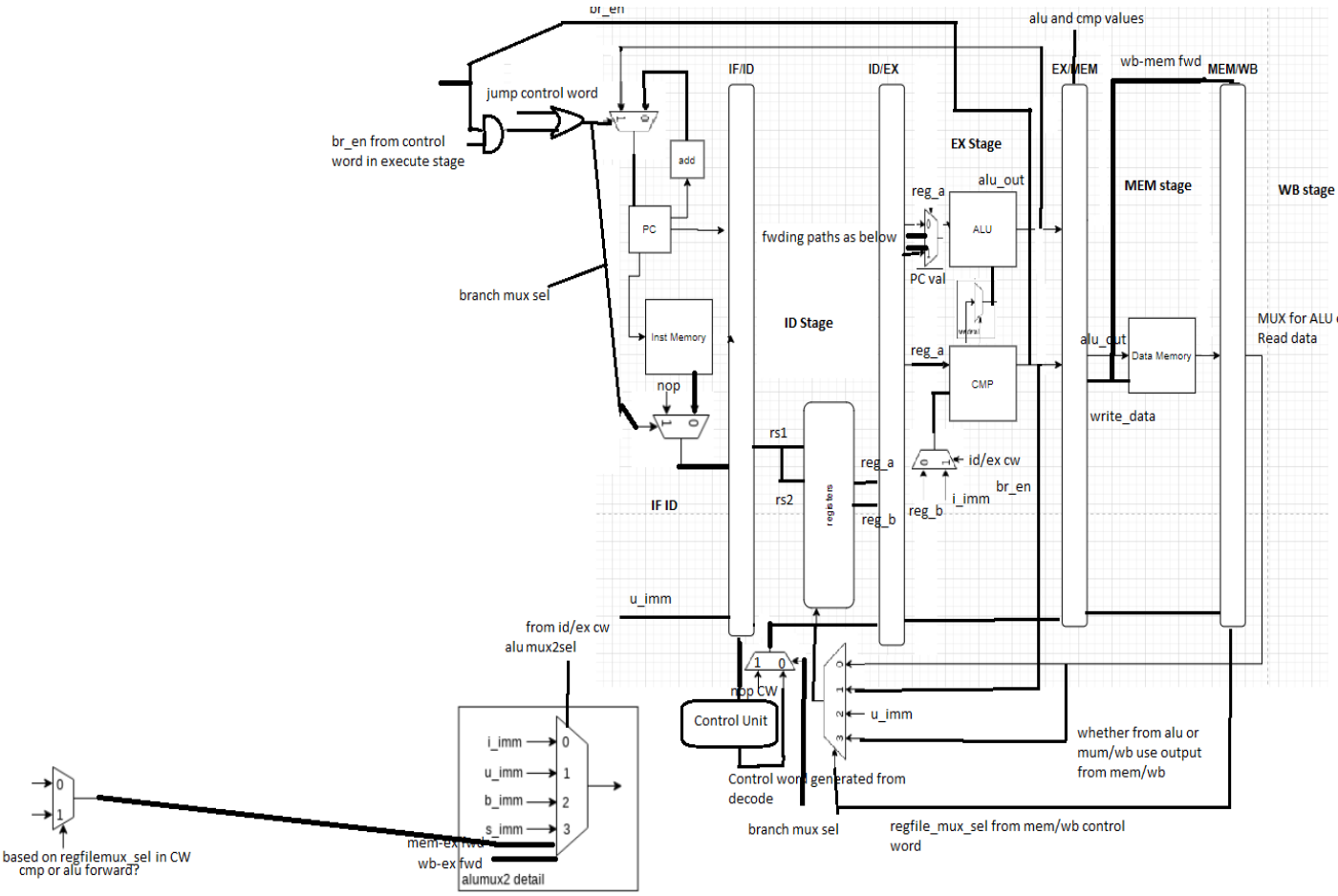
Vishnu Jaganathan
Anirudh Kashyap
Ethan John

	pc_out (old val) Alumux2_sel = u_imm		ALU_buffer
JAL	Alumux1_sel = pc_out Alumux2_sel = j_imm Aluop = alu_add Need to load regfile with new pc and jump here	N/A	N/A
JALR	Aluop = alu_add Pcmuxsel = alu_mod2 Load_pc = 1? Save new pc in regfile	N/A	N/A
BR	Aluop = alu_add Alumux1_sel = pc_out Alumux2_sel = b_imm Cmpop = branch_funct3 Load_pc = 1 Pcmux_sel = {0, br_en}	N/A	N/A
LOAD (all forms)	Aluop = alu_add Load_mar = 1	Mem_read = 1 Load_mdr = 1 (figure out how to freeze pipeline for cp2)	Set regfilemux appropriately based on type of load Load_regfile = 1
STORE (all forms)	Alumux2_sel = s_imm Aluop = alu_add Load_mar = 1 Load_data_out = 1	Mem_write = 1 Set mem_byte_enable depending on type of store (figure out how to freeze pipeline for cp2)	N/A
SLTI	Cmpop = blt Cmpmux_sel = i_mm	N/A	Load_regfile = 1 Regfilemux_sel = br_en_buffer

Vishnu Jaganathan
Anirudh Kashyap
Ethan John

SLTIU	Cmpop = bltu Cmpmux_sel = i_mm	N/A	Load_regfile = 1 Regfilemux_sel = br_en_buffer
SRAI	Aluop = alu_sra	N/A	Load_regfile = 1 Regfilemux_sel = ALU_buffer
Other IMM instructions	Aluop = aluops'(funct3)	N/A	Load_regfile = 1 Regfilemux_sel = ALU_buffer
ADD	Aluop = alu_add Alumux2_sel = rs2_out	N/A	Load_regfile = 1 Regfilemux_sel = ALU_buffer
SUB	Aluop = alu_sub Alumux2_sel = rs2_out	N/A	Load_regfile = 1 Regfilemux_sel = ALU_buffer
SLT	Cmpop = blt	N/A	Load_regfile = 1 Regfilemux_sel = br_en_buffer
SLTU	Cmpop = bltu	N/A	Load_regfile = 1 Regfilemux_sel = br_en_buffer
Other REG instructions	Aluop = aluops'(funct3) Alumux2_sel = rs2_out	N/A	Load_regfile = 1 Regfilemux_sel = ALU_buffer

Vishnu Jaganathan
Anirudh Kashyap
Ethan John



This is the diagram of the pipelined processor datapath for checkpoint 1. The data and execution flows from the stages in a left to right fashion (see datapath diagram on next page), except the write-back stage places a result back to the register file the decode stage, and the PC chooses between an incremented PC and the branch address calculated in the MEM stage. In the diagram, each stage is separated by pipeline registers (IFID IDEX etc… in diagram), which in our design were implemented using the aforementioned control word structure.

CP2

The second checkpoint of the project consisted of adding an L1 I-Cache & D-Cache, adding appropriate stalling logic, static branch prediction, and detecting hazards and implementing forwarding.

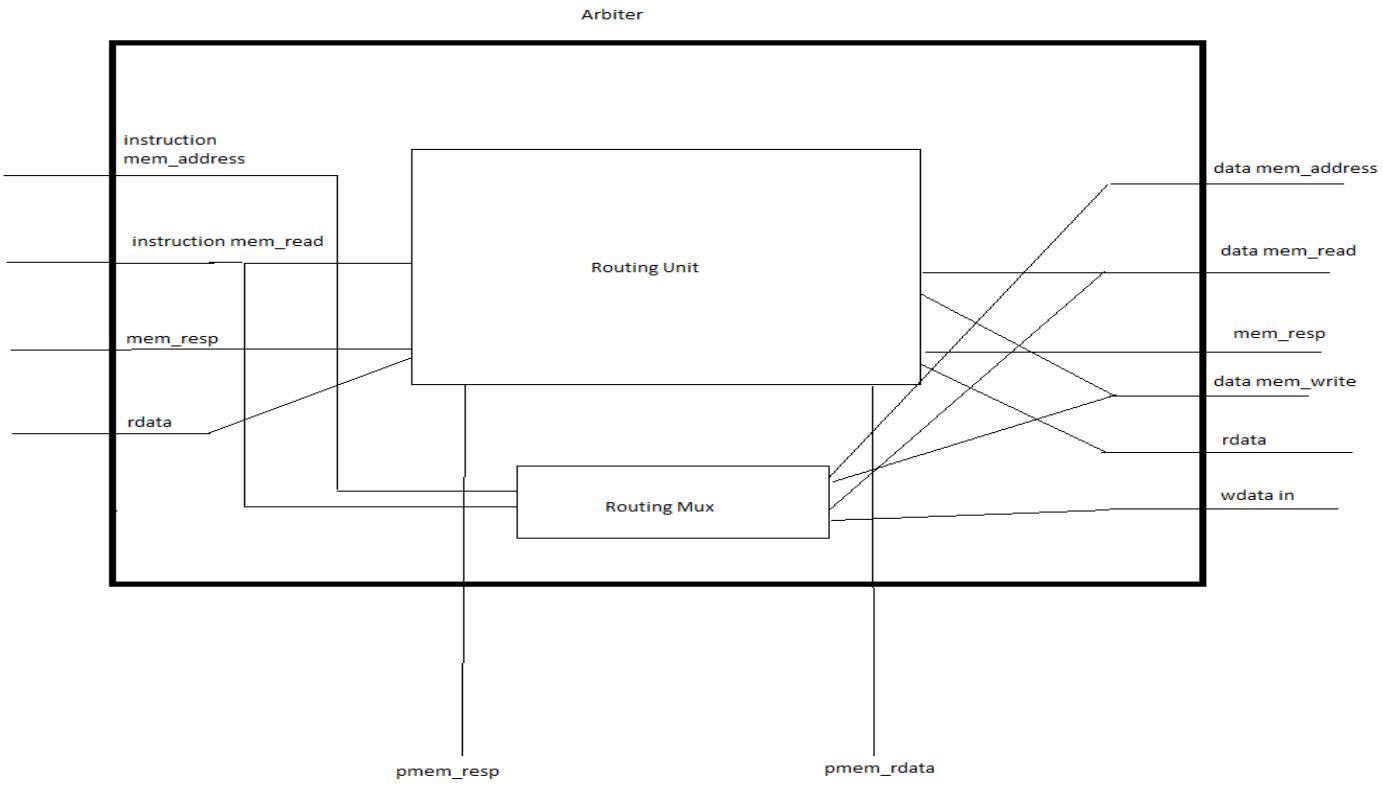
Forwarding was required as after the first checkpoint as while the processor was able to run pipelined instructions, it could not properly run if there were any dependencies between instructions that were in the pipeline that hadn't committed, or written to the register file yet. Hence, these data hazards had to be detected and forwarding logic needed to be implemented.

Vishnu Jaganathan
Anirudh Kashyap
Ethan John

Implementing forwarding consisted of if statements that assigned mux selects for forwarding. Three things were checked: whether an instruction at the further stage in the pipeline has the destination register that is a source register in the current instruction, whether an instruction further down the pipeline is an instruction that actually writes to the register file, and whether the source register is not x0. If they are all true, forwarding from one of the next stages is used. If two stages both have the same destination register that is to be used in forwarding, the most recently computed result was used.

There were 3 forwarding paths implemented: MEM-EX, WB-EX, WB-MEM. As mentioned above, if all the conditions for forwarding were met, a mux type that we created for forwarding called use_fwd was assigned. Else no_fwd was assigned. These forwarding selects were added to the case statements to use the forwarded outputs when required in for ALU multiplexers, MUXes that were used for forwarding regb on stores in the EX stage, and write data and memory address forwarding in the mem stage.

The second half of the checkpoint was to add the L1 caches and appropriate stalling logic. We also had to accommodate for the case where there is a miss in both caches that therefore request memory accesses for both the I and D cache. In order to decide which cache took priority, an arbiter was designed.



The arbiter gave priority to the data cache in the event of both caches reporting a miss.

Vishnu Jaganathan
Anirudh Kashyap
Ethan John

CP3 & Advanced Design Options

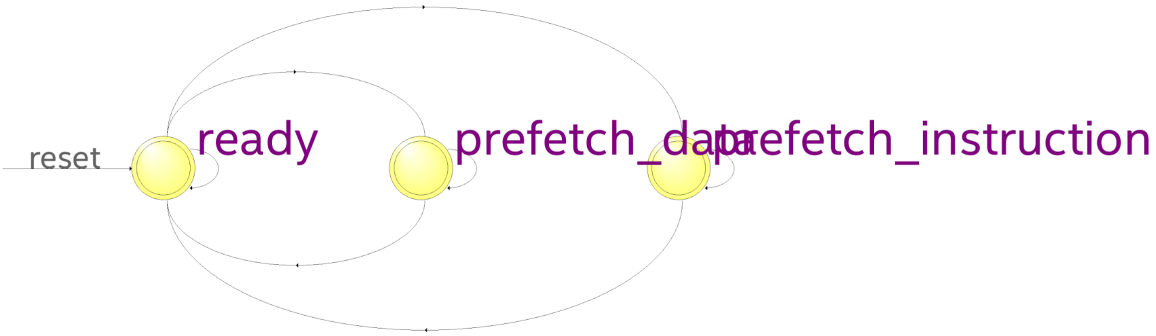
The third checkpoint of the project was mainly dedicated to advanced features. We added an L2 cache, stride prefetching, tournament branch prediction, and an eviction write buffer to the pipelined processor.

Stride Prefetching:

Design

The prefetcher used a stride prediction algorithm where the distance between the previous and current memory access was used to predict the next. Through experimentation that just involved running the testcodes and measuring runtime performance, we found that making a prediction off of one stride yielded the best results. The implementation of this was using registers to hold the values and calculation upon each memory access. This system fell back on a next instruction prefetcher, which just prefetched the next cache line from the current instruction. Both of these prefetching schemes were located in the arbiter which sat between the L1 caches and the L2 cache. When there was neither a data or instruction miss, the arbiter used the prefetcher to load the target data into the L2 cache. Additionally, there were hit signals coming out of the L2 cache to ensure that neither instruction or data that was already present in this cache was fetched again, to not waste cycles.

State Machine



State Name	Description	Transition	Outputs
ready	This state serves instruction or data misses from icache and dcache,	When no memory requests are given, it transitions to prefetch_data if that	Connects memory signals to the appropriate cache, either icache or

	implementing the normal function of the arbiter.	data is not in the L2 cache and prefetch_instruction otherwise. If the instruction is also in the L2 cache then the arbiter stays in the ready state.	dcache, giving dcache priority if both misses occur simultaneously.
prefetch_data	This state prefetches the predicted next data, learning the stride based on the distance from the previous data access to the current.	Once the prefetching is complete, it transitions back to ready.	No outputs, instead it fills the L2 cache with the prefetched data.
prefetch_instruction	This state prefetches the cacheline of the next instruction.	Once the prefetching is complete, it transitions back to ready.	No outputs, instead it fills the L2 cache with the prefetched instructions.

Testing

We tested this implementation with the mp4-cp3.s which was heavy on memory accesses and made sure the correct results were outputted. Additionally, we partitioned this code into many sections and ensured that the expected results were obtained in each section. Also, analyzing the state machine of the arbiter could tell you when the arbiter was serving an instruction or data miss, prefetching the next data, or prefetching the next instruction. We used these waves to see that the expected data was being placed into the L2 cache.

Performance Analysis

We analyzed the performance of this feature by testing a version of the code with and without the modified arbiter across the three competition codes and mp4-cp3.s. We found that on average a 10% improvement occurred in runtime, suggesting that the prefetcher added value. Most of the improvements came from the data prefetching during regular data accesses, while some came from long stretches of code that spanned multiple cache lines. Here, some time was saved by fetching the next instruction.

Vishnu Jaganathan
Anirudh Kashyap
Ethan John

Branch Predictor:

Design

The branch prediction consisted of making a tournament branch predictor that chose between a local and global predictor. The local predictor consisted of storing PC values where there is a br instruction and using the outcomes of the branches at those PC values to update a 2 bit up down counter to choose the appropriate outcome. Up to 4 PC values could be stored. Values are overwritten after that. The global predictor consisted of 16, 32 bit up down counters that incremented when branch was taken and decremented when branch was not taken. Four set index bits were used to index these, with a block size of 8 bits. Essentially, each group of 256 instructions shared a branch predictor, as our assumption was that nearby branches may be correlated with each other. Since this behaved as a directly mapped cache, tag bits were used to evict entries to be replaced by ones further away.

The tournament between these branch predictors was a 2 bit up down counter that moved in the direction of either the local or global predictors. When the local was right and the global was wrong, the state machine moved in the direction of local, and when the global was right and local was wrong, the state machine moved in the direction of global. When both were right or both were wrong, the state machine maintained its current state. The predictions for both predictors were buffered in shift registers to be accessed in the update phase of that branch prediction.

We also implemented a more generalized pipeline flushing logic from the static not taken branch predictor in the previous checkpoint. More specifically, when there is a prediction made we pass the result down the pipeline, and when the branch instruction evaluates in the ALU, we compare the result to the prediction and assign a boolean variable that tells the pipeline to flush the instructions in the previous stages on a misprediction.

Vishnu Jaganathan
Anirudh Kashyap
Ethan John

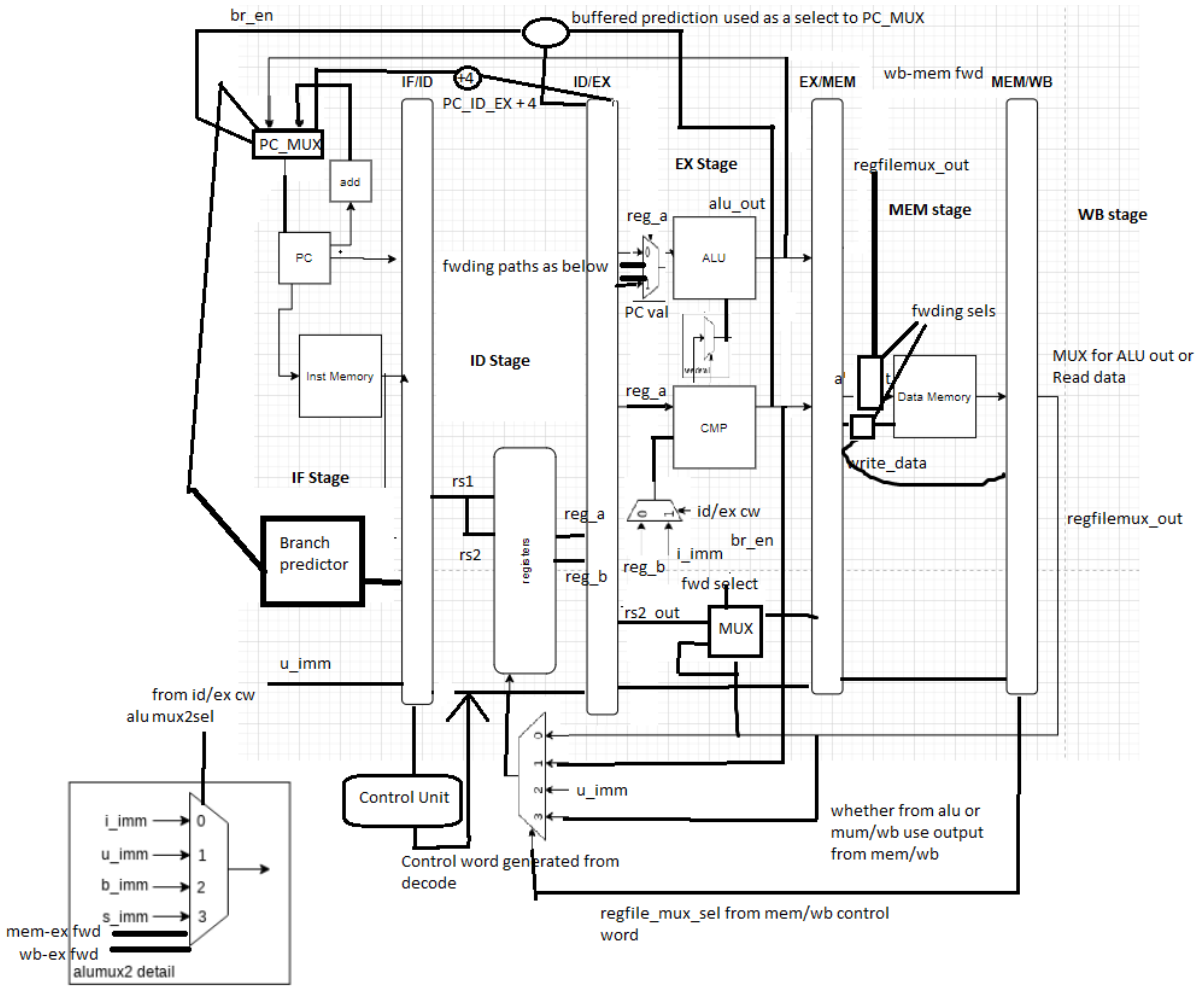


Figure containing Datapath with branch predictor and flushing logic. The prediction stored in IDEX buffer was compared with br_en to assign the appropriate mux select and clear out instructions on a misprediction.

Testing

To test the branch predictor, we executed a simple factorial program that had 3 branch instructions and made note of the expected percentage of predictions that should be taken/not taken, and how many should be hits/misses as it was easy to track. We also ran our branch predictor on other simple programs with not too many lines to compare the results to hand execution.

Performance Analysis

To track the performance of the branch predictor, we augmented our testbench to initialize integers to track the number of branch predictions and correctly predicted branch predictions to 0, and then incremented these variables appropriately whenever there was a branch that was evaluating in the pipeline. While our code did well on the checkpoint 3 code with 80%, the

competition code was around 70%. We realized that this was because we were directly storing PC values in the local predictor instead of actually localizing the predictions based on an indexing scheme, and that the assumptions we made about correlations in the global branch predictor were usually not true.

Eviction Write Buffer:

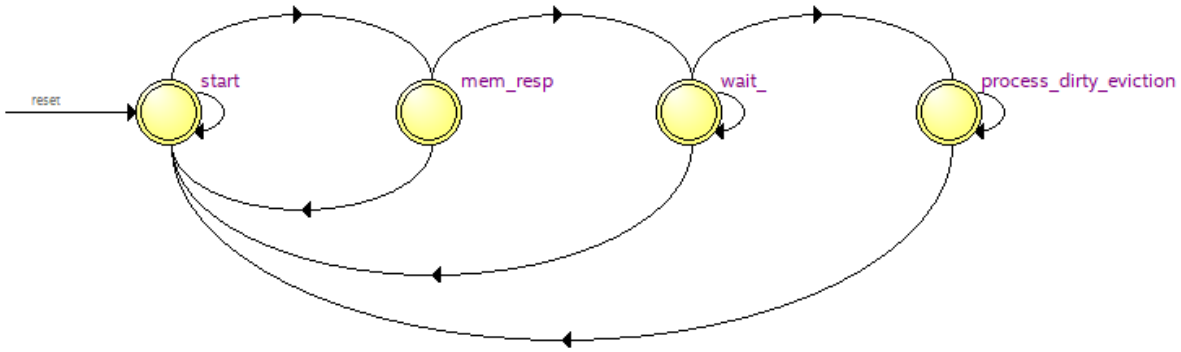
This is a buffer that sits between a cache and the next level cache or memory to store data on an eviction. It takes signals from the cache and sends back mem resp immediately to allow the data to be fetched from memory or the next level cache sooner. After the new data is loaded, the eviction write buffer writes its data to the next level cache.

Design

Since there is essentially no datapath as it is just one 256 bit buffer, the eviction write buffer mainly consisted of designing a 4 state state-machine that took all the signals from the cache and transitioned to the appropriate state, outputting the correct signals to memory and the cache. On a read request from the cache, the write buffer state-machine just stays in the start state and forwarded the read signal. On a write, it transitions to the mem_resp state wherein mem resp is sent back to the cache so that the evicted data write time is avoided. After the mem_resp state, there is an unconditional transition to the wait_ state where the write buffer waits for the read from the next level cache or memory to finish. Finally, the eviction write buffer writes to the next level cache or memory in the process_dirty_eviction state. By doing this, the time of waiting for the write to finish before allowing the pipeline to resume is reduced.

Vishnu Jaganathan
Anirudh Kashyap
Ethan John

State Machine



Memory signals were also manipulated to ensure that there is only one request being processed at once. I.e. if there is another miss while the eviction write buffer is writing to the next level cache or memory, the request is delayed until the write is done.

State Name	**Description**	**Transition**	**Outputs**
start	The default state that the state machine stays in and allows writes of evicted data to buffer on mem_write	On a mem_write, transition to mem_resp state	-Forwards pmem resp from memory/next level cache to cache to allow for reads -Forwards mem_read signals from cache, forces mem_write signals from cache to 0 - Raises signal to allow writes to buffer
mem_resp	State for sending mem_resp high back to cache	Unconditional transition to wait_ state	All memory signals low
wait_	Waits for read to cache to complete	On receiving pmem_resp,	Forwards caches mem_read signal,

		transition to process_dirty eviction state, else stay in wait_ state	forces write to low, forwards pmem_resp to cache
process_dirty_eviction	After read is complete, write back data to next level cache or memory	Waits for pmem_resp, then transitions to start state	mem_resp sent to cache is forced low. If there is another miss, it must wait for the writes to finish. Holds any incoming reads/writes off.

Testing

This consisted of running edge case testcodes like having a cache miss while the write buffer is writing back, having a WaW case to the write buffer etc. We would track the state and buffer values to make sure it was functioning properly. After this, we ran the CP3 test code and checked the output which remained unchanged.

Performance Analysis

For performance analysis, we just noted the difference in runtimes before and after the buffer was added. We noticed a slight speedup in some of the competition testcodes by 2-5%, but saw a slight worse performance in cp3 and one of the competition testcodes. This was because in the process_dirty_eviction state, all reads and write requests are held off until the write is processed which could cause a delay if there are many misses at a time.

L2 Cache:

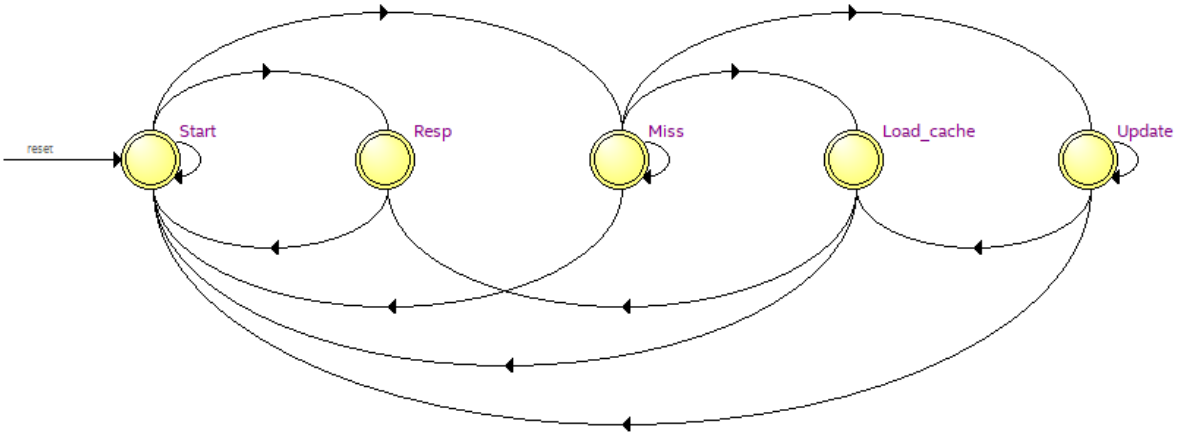
This cache acted as an intermediary between the L1 I-Cache and D-Cache and memory. If implemented properly, it should greatly minimize the need to retrieve data from memory during program execution.

Design

The L2 Cache design was based on a cache implemented in a previous lab in the class by fully parameterizing it. The final product of this L2 cache was a 2 way by 16 set cache. The cache consisted of a five stage state machine to handle read requests and process evictions. Below is a table briefly explaining each state:

Vishnu Jaganathan
Anirudh Kashyap
Ethan John

State	Description
Start	Default state awaiting mem_read or mem_write signals from CPU. If there is a hit, there is immediately a output next cycle
Resp	Chooses appropriate MUX select to get correct data_out and raise mem_resp
Miss	On a miss in the cache, read from memory and load into cache.
Update	If the LRU entry is valid and dirty and needs to be evicted, write to memory. Select cache output to adapter to be the LRU way's data.
Load_cache	After receiving response from memory, load to cache registers, flip LRU



Vishnu Jaganathan
Anirudh Kashyap
Ethan John

Testing

For testing, we executed all competition testcodes and the CP3 test code to ensure that adding the L2 cache did not change the outcome of the execution of the test code. We also analyzed the waveform to make sure that the increased sets and parametrization did utilize the extra sets added.

Performance Analysis

We compared runtimes before and after adding the L2 cache to benchmark performance. We saw a ~55% decrease in runtime across CP3 and all design competition testcodes.

Additional Observations:

During the course of developing the processor, we noticed how important verification was. Without our own verification pipeline set up or using the rvfi monitor, it was very hard to debug things like data forwarding edge cases. We finally ended up using a working MP2 processor and its testbench to write gold commits of valid instructions to a file and comparing it to a file that the MP4 processor wrote its commits to. We also wrote down the times of the commits and used a python script to automatically tell us what time in both MP2 and MP4 this mismatch in commits happened. This took one and a half days to fully setup and verify as functional, and so we lost time in developing the CP3 branch predictor. We underestimated the importance of a verification pipeline in earlier checkpoints, but it was very worth setting up as any edge cases we had to account for were solved in a matter of hours.

Conclusion

Overall we learned a lot about hardware development and the challenges one faces when designing hardware and implementing verification. It cannot be stressed enough how important verification was and how easy it became to debug once verification was set up. Other than this, it was very interesting to consider advanced features and how different branch prediction and prefetching schemes could be designed and change the performance of the processor by a significant amount. Finally, the project also taught us the importance of teamwork and communication. We have become overall more judicious and better developers at the end of this project.