ECE 411: MP3 Report

A Pipelined Implementation of the RV32I Processor

Group null

Zhili Luo (zhilil2), Michelle Kang (dykang2), Ni Zhang (nizhang2)

Introduction

In this project, our goal was to create a 5-stage pipelined processor with split L1 caches and unified L2 cache. The processor would support all of the RV32I instruction set with the exception of FENCE, ECALL, EBREAK, and CSRR instructions.

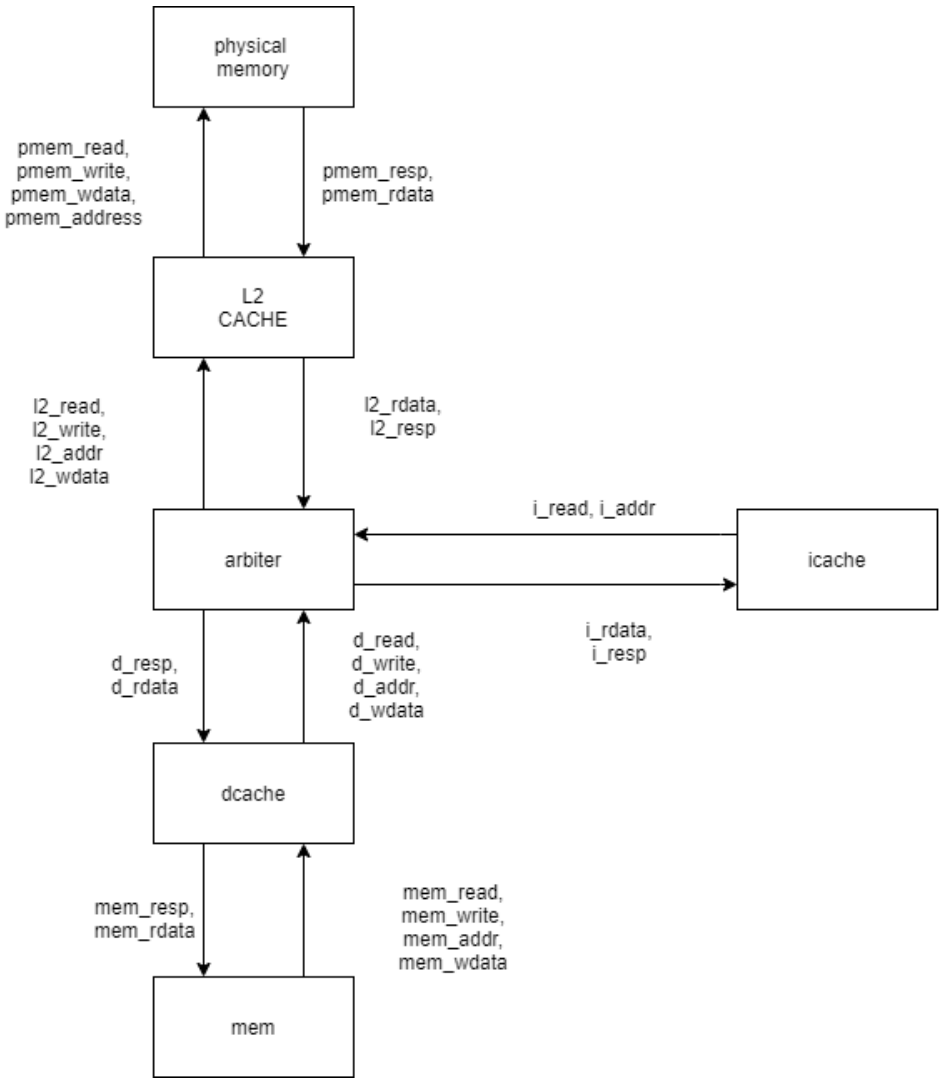
In the first machine problem, we have implemented a RV32I processor. In the second machine problem, we have implemented a unified 2-way set-associative cache. In this project, we were able to revisit MP1 and MP2 and improve the performance of the processor by introducing and implementing new features that we learned in class.

For the CPU part of the processor, we implemented a simple pipeline based on the previous machine problem. Then we added hazard detection and forwarding. For the memory part of the processor, we implemented 2-way 8-set instruction cache, data cache, L2 cache, and an arbiter in L1.

From this project, we were able to test and challenge our knowledge gained from the class. We could have hands-on experience with computer architecture, and we could also discuss the tradeoffs of different design implementations. Throughout this report, we will discuss the design ideas we came up with, the hardships we have faced, and the solutions to overcome the hardships.

Project Overview

The project was divided into five checkpoints. For each checkpoint, our goals were to come up with a feasible design that successfully passes the given test code. Each group member agreed to split the work and worked on the fairly distributed workloads. For the first three checkpoints, we designed and built the base of our processor; we implemented split L1 cache (instruction and data cache), unified L2 cache, pipelined datapath, and data forwarding. For the last two checkpoints, we designed and built advanced features to achieve better performance.



Memory Hierarchy

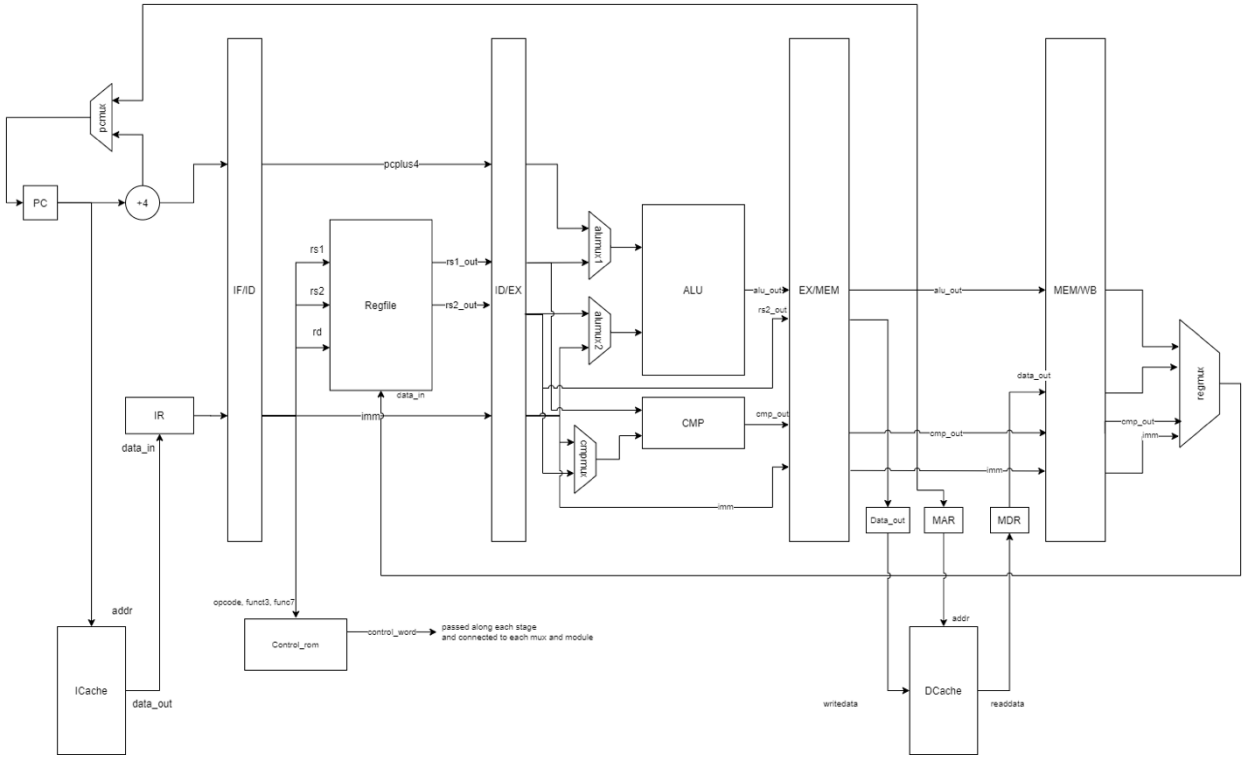
Design Description

Overview

In this section, we will cover the progress of each checkpoint from building the processor to implementing advanced features.

Milestones

Checkpoint 1



Basic datapath (no data hazard)

In this checkpoint, the objective was to implement a partial 5-stage pipeline and complete the design of two L1 caches and arbiter.

Based on our design prior to checkpoint 1, we implemented a five-stage pipeline with control rom that takes inputs of opcode, pc, rd, funct3, funct7, imm. It outputs a control word that contains various control signals for each stage of the pipeline. In the fetch stage, we obtain PC by incrementing it by four or from the memory stage if the instruction is to branch. Then we use the PC to load instruction into IR. In the decode stage, we use information from the PC and feed the

input into the control rom and load the source and destination registers in the regfile. In the execute stage, we use signals from the control word and feed them into ALU and CMP. In this stage, we perform the arithmetic operations and determine if it is a branch. In the memory stage, we access memory with signals from data cache. In the write back stage, we feed control signals into the write back mux and write the desired result back to regfile. To test the design, we used the provided test code and checked that the registers values were correct.

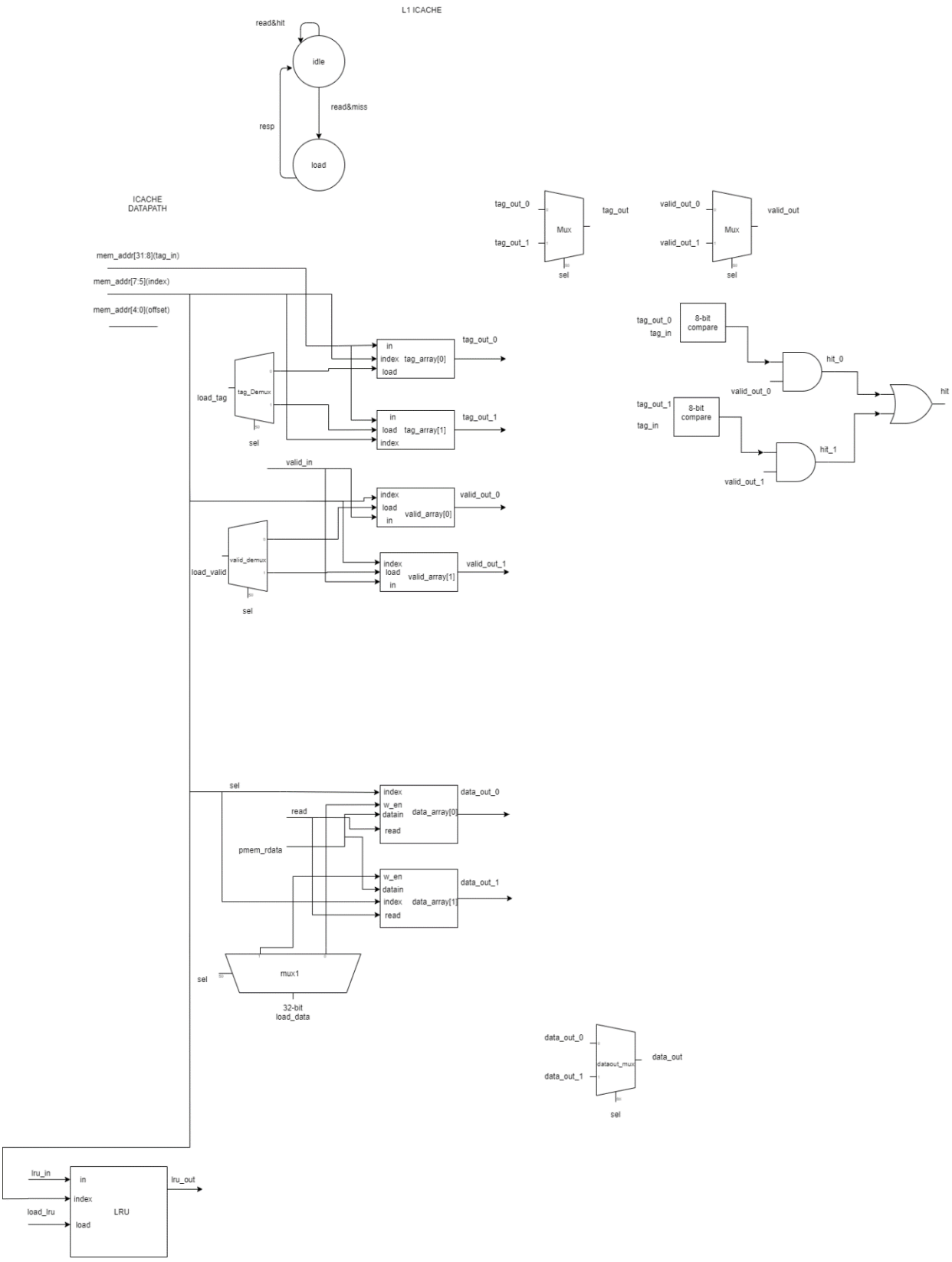
For the cache design, we decided to start with the 2-way 8-set cache we built in mp2 and modify the state machine so that the both caches could support single cycle hit detection. The two caches were connected to the arbiter.

Checkpoint 2

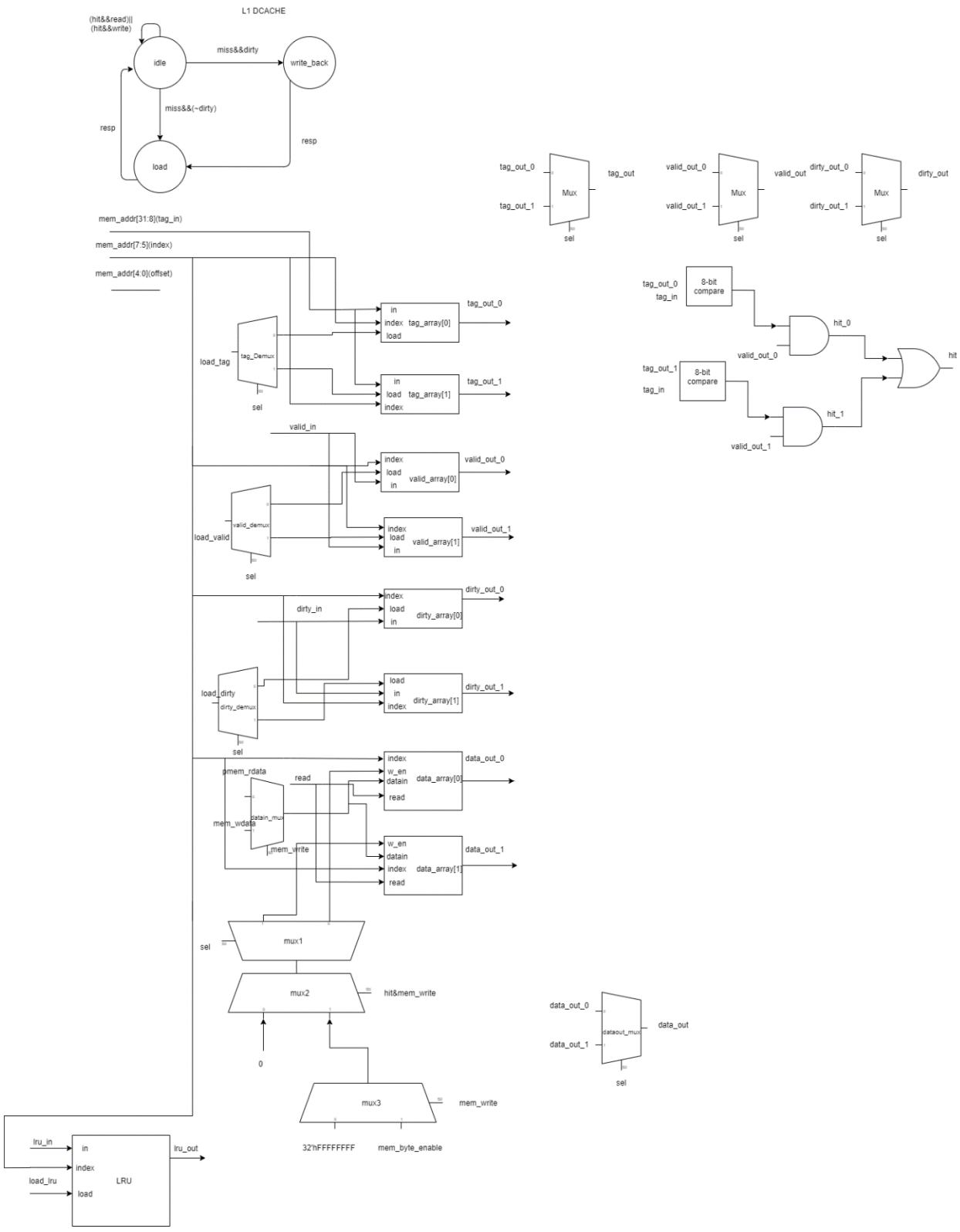
In this checkpoint, the objectives were to implement the rest of the pipeline, L1 instruction cache, data cache and arbiter. We also designed hazard detection and forwarding in the pipeline.

For the pipeline implementation, we added the features that detect structural hazards and pause when the instruction or data are not ready to be used. If in the execution stage or memory stage, the destination register is the same as in either of the source register in the decode stage; we would stall the input into the execute stage. When we are reading or writing to cache, if the response signals are not ready, we pause the pipeline by feeding the output of each stage back to the input. In the decode stage, we do not load the PC for the next instruction if we are stalling the pipeline.

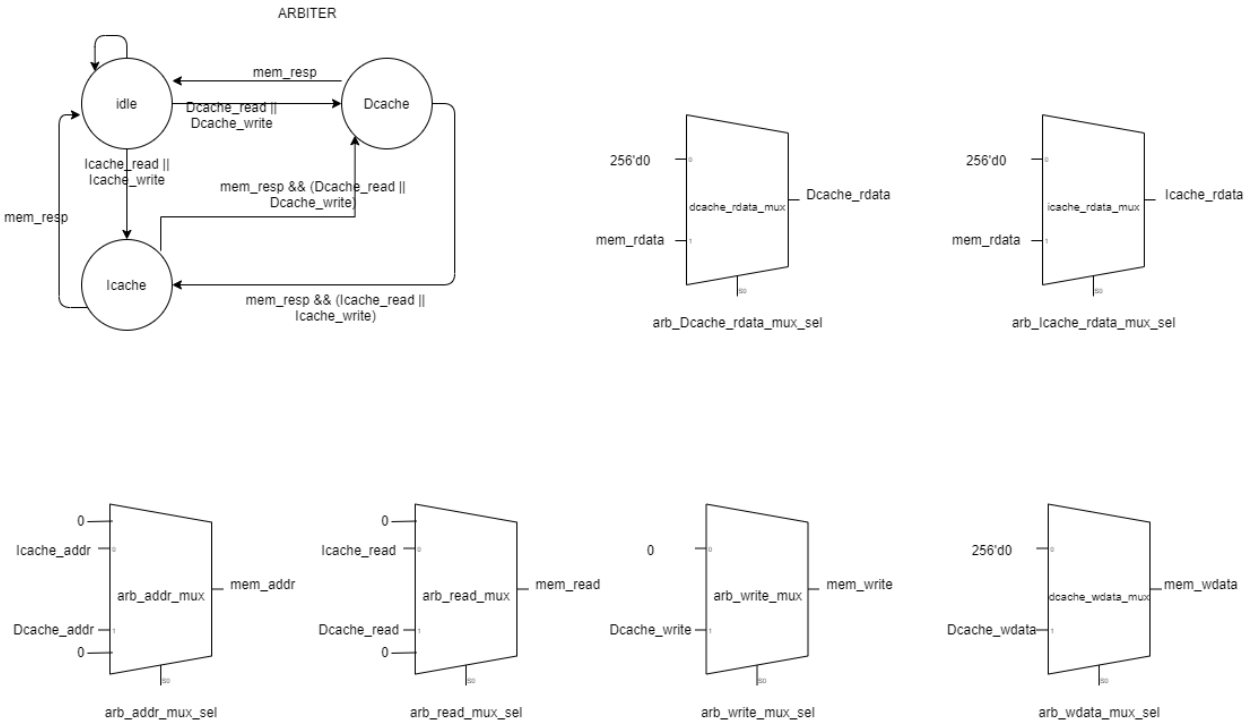
For L1 data cache, we implemented 2-way, 8-set write back cache with LRU replacement policy and 32-byte cacheline.



ICache state diagram and datapath



DCache state diagram and datapath



Arbiter state diagram and datapath

In cache control, there are three stages. In the decode stage, we check read, write, hit and dirty signals. If the read or write signal is not high, or it is a cache hit, we stay at the decode stage and set response signal to high. If it is a miss and the cache is dirty, we go to the write back stage. If not, we load the new data to cache. In the write back stage, we write back the cacheline back to memory by changing the physical memory address and setting the physical memory write signal to high. After we get the response signal from the physical memory, we move to the load data stage. In this stage, we set the physical memory read signal to high. After we get the response signal from physical memory, we load the data into cache by sending load data and valid signal to the datapath.

In the cache datapath, we maintain two data arrays, two tag arrays, two dirty arrays, two valid arrays and a LRU array. For the LRU array, we load the array if there is hit and feed the hit result of the first way into the input. Then the array outputs the least recently used way. If there is a hit and it was not from the first way, we know that the first way is not the recently used way and vice versa. We store the hit result by comparing tags in both ways to the input tag if valid signals in each way are high. When evicting a cacheline, if we obtain a load data signal from control and if it was the least recently used way, we set load and write enable signal to high to write a new cacheline. When sending data from cache to cpu, if hit signal and write signal are both high, we load the data array and set write enable to the input memory byte enable.

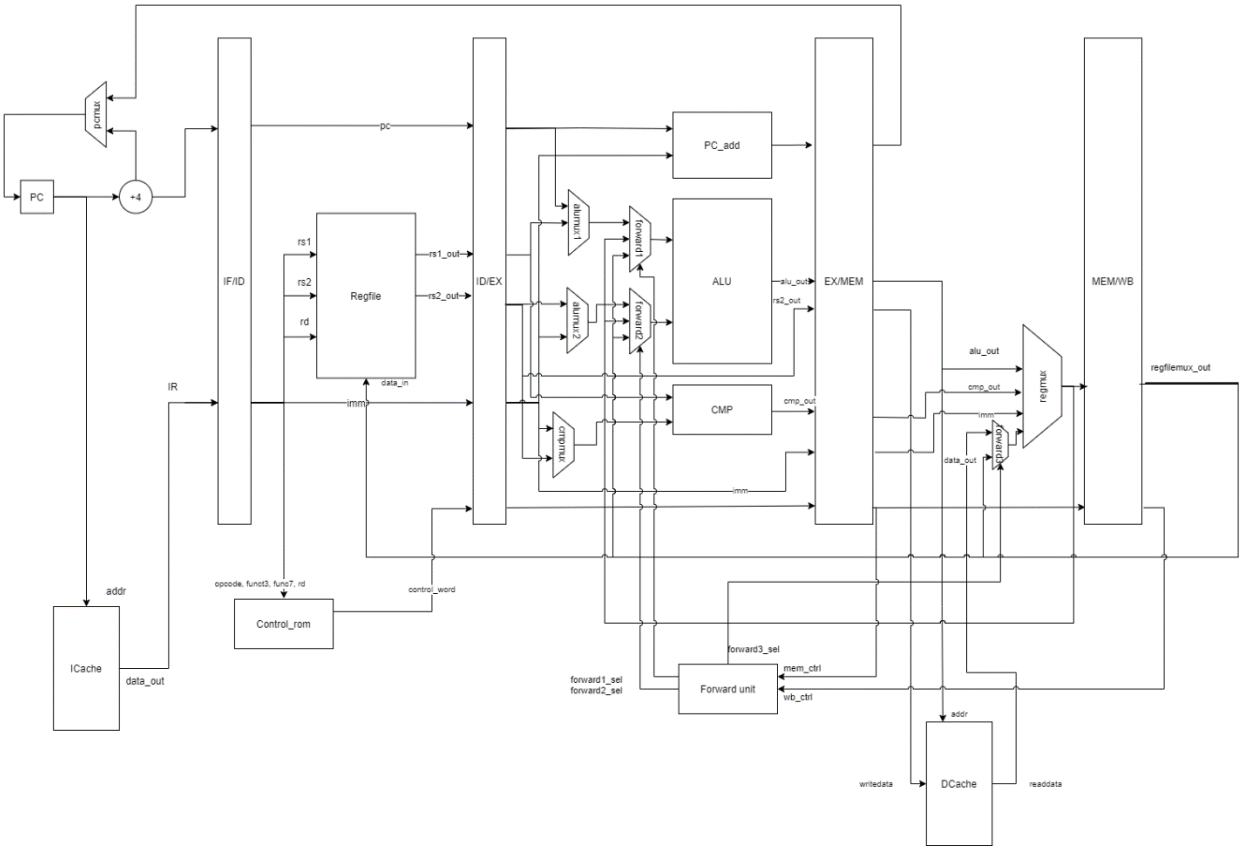
For the instruction cache, we use the same implementation as that of data cache but remove all signals related to cache write.

We also implemented arbiter between the instruction and data L1 cache to prevent conflicts that might happen when both data and instruction cache send signals to physical memory. There are three stages in arbiter control. In the idle stage, we detect if there are signals coming from instruction cache or data cache. If not, we stay in the idle stage. In the instruction cache stage, we connect the signals of the instruction cache to physical memory. If the physical memory response and data cache read or write signals are high, we move to the data cache stage and connect data cache signals to physical memory. Otherwise, we return to the idle stage.

Checkpoint 3

In this checkpoint, the objectives were to finish the basic pipeline and memory hierarchy, which include implementing a unified L2 cache, enabling data forwarding and static branch mispredict.

In the pipeline, we implemented three types of forwarding by adding forwarding logic and three forwarding muxes and revised the datapath to support them. When the destination register of the memory stage input is the same as the source register of the execute stage input, we enable execute to the execute forwarding by selecting the result of the execute output in the forwarding mux to the execute input. When the destination registers of write back stage input are the same as either of the source register in the execute stage input, we enable memory to execute forwarding by selecting the result of the memory output in the forwarding mux to the execute input. If it is a load instruction and the destination register in write back stage input is the same as the source register in memory stage input, we enable memory to the memory forwarding by selecting the result of memory output in the forwarding mux to the memory stage input.



Revised datapath (with data forwarding)

We also implemented static branch mispredict. For the memory stage input, if we have jump instructions or if we have branch instruction and the branch enable signal is high, we flush the pipeline by setting decode, execute, memory stage inputs to 0.

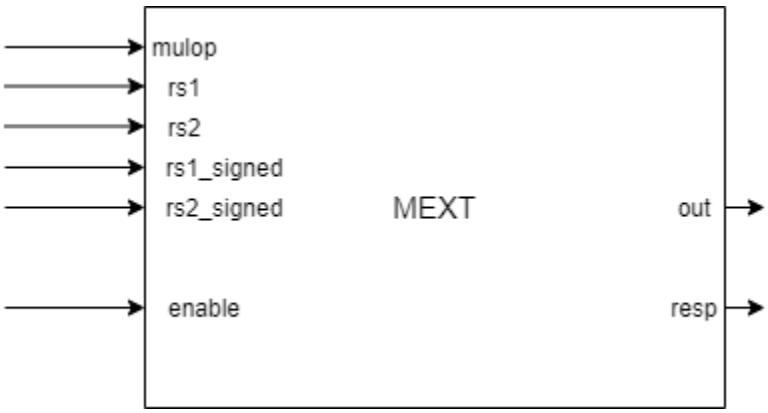
For the L2 cache, we used the same implementation as the data cache and changed the number of sets to 32.

Advanced Design Features

To improve performance, we implemented M-extension, basic prefetch, and tournament branch predictor. We also used a performance counter that updates the value of total branch count, mis-predict branch count, stall count, prefetch count, cache hit, and cache miss every clock cycle.

RISC-V M-Extension:

This feature enables signed multiplication and division operations in the processor. We use the basic add-shift multiplier from the previous machine problem. We implemented M extension in the execute stage of the pipeline. The pipeline pauses when the arithmetic operation is performed.



M Extension module

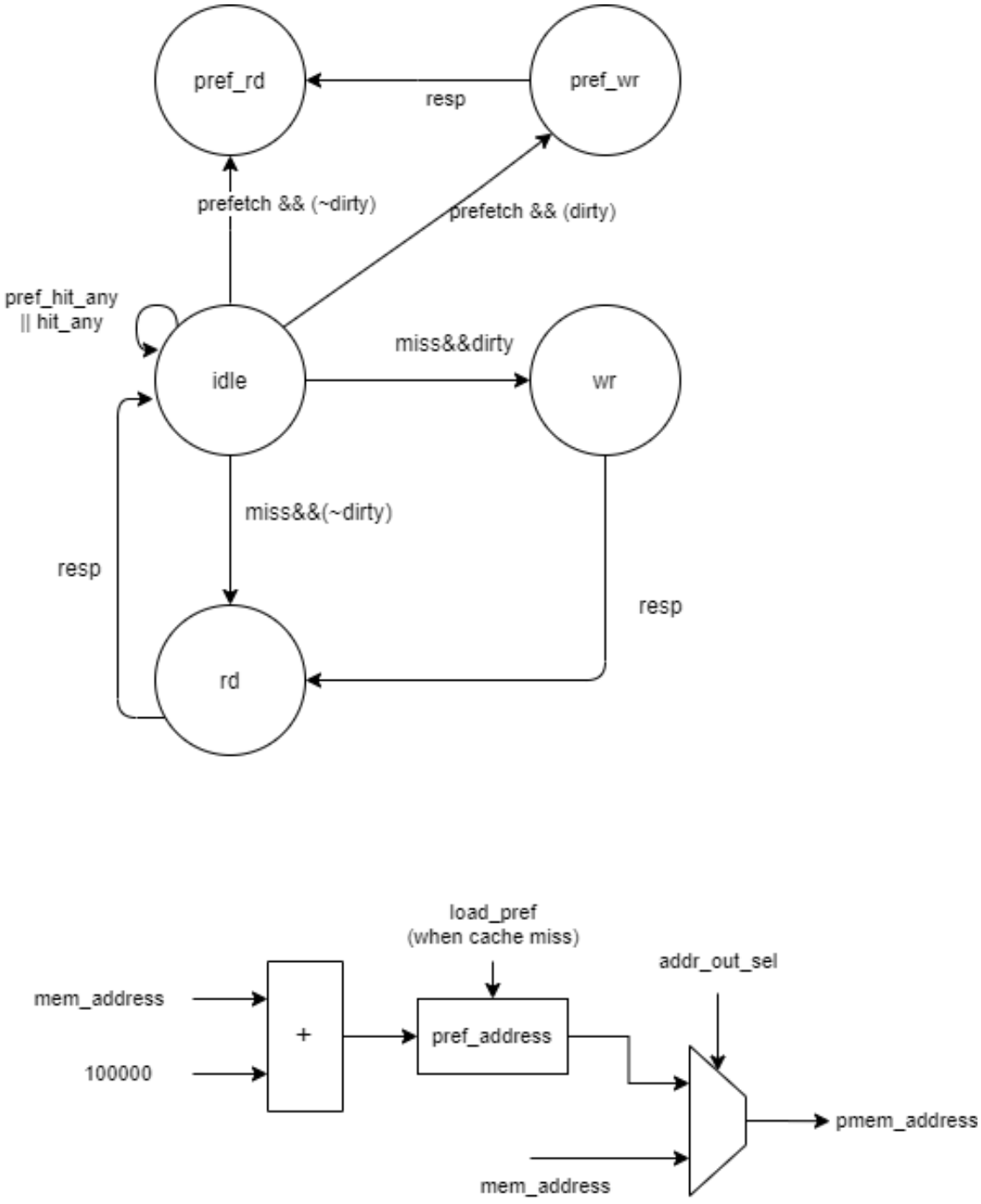
We found that this implementation significantly improved the performance. When executing the competition test code, it took 3759505ns without M-extension and 816345ns with M-extension. Initially, we found out that the long combinational path from data cache to the M-extension module caused Fmax drop to 36MHz. However, we were able to fix this problem by using two cycle hit data cache.

Comp2 time without M Extension (ns)	*Comp2* time with M Extension (ns)
3759505	816345

Basic Hardware Prefetching:

We implemented basic prefetch in the data cache. The spatial locality would benefit sequential prefetching, which will improve the program performance. We generated a prefetch signal in the cache datapath, and updated it with the state machine. In the cache control, load the prefetch signal and prefetch response signals to datapath.

We added two states in the cache control, prefetch read and prefetch write. In the decode stage, we loaded prefetch and checked whether it is a cache miss. When there are no read or write signals and the prefetch signal is high, prefetch is performed at the prefetch address by incrementing the 6^{th} bit of the memory address. If it is a cache miss and the cache is dirty, we move to prefetch write back stage. In the prefetch write back stage, we write back the data of the prefetch address. After the response signal from physical memory is received, we move to the prefetch read stage. In this stage, we read data from memory and set the prefetch response signal to high. In the datapath, the prefetch related signals are updated every clock cycle. If the prefetch response signal is low, the prefetch signal is set to low. If the load prefetch signal is high, set the prefetch signal to high and update prefetch address.



Prefetch state diagram and datapath

With prefetch test code, we were able to reduce the performance time from 16615 ns to 13805 ns. However, with cp4 test code, we found that the decrease in time is not proportional; it had small speedup. It was because the prefetching process itself takes up time and space. Non sequential memory reads results in data not used being fetched and stored in cache.

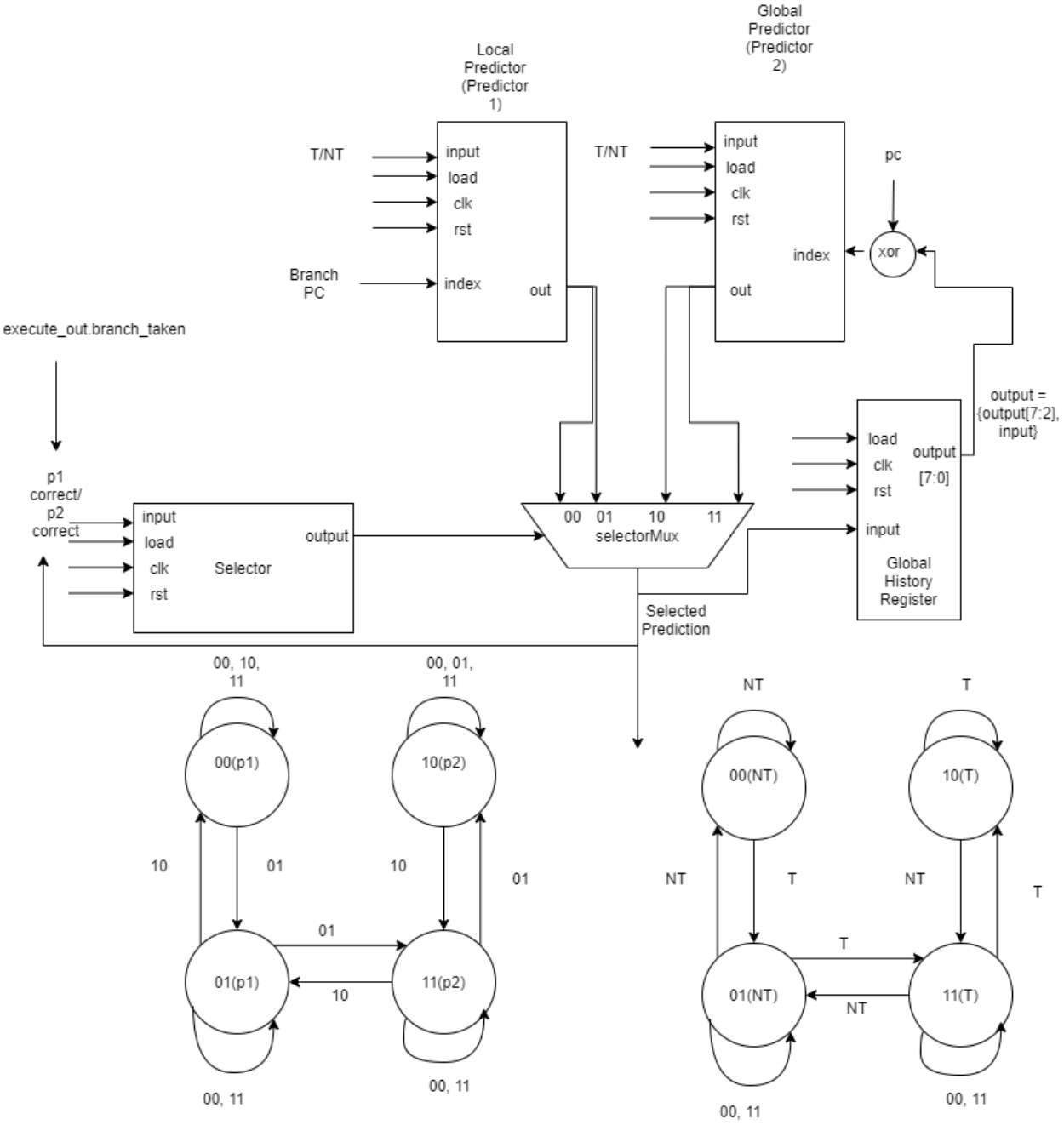
Prefetchtest time without Basic Prefetch (ns)	*Prefetchtest* time with Basic Prefetch (ns)
16615	13805

Tournament Branch Predictor:

We implemented tournament branch predictor by maintaining a local branch predictor and a global predictor. Their predictions and the actual branch taking results from the pipeline are fed into the state machine of a selector. The selector makes a decision to choose local or global predicting results based on the correctness of the two predictors.

The local branch predictor is implemented with a branch history table and a 2-bit counter. In the branch predict history table, we use a 16-entry data array to store the 2-bit predict result. For the input predictions, bit 5 to bit 2 in the execution stage PC are used to store the result; for the output prediction, the data stored in bit 5 to bit 2 in decode stage PC of the next instruction are used. When the operation in memory stage input is branch, the counter is updated. The counter takes an input from the memory stage in the pipeline and the predict table. The prediction is updated based on whether the branch is actually taken in the execution stage. Then, the result in the predict table is stored. Also, the prediction in the local predictor is initialized to strongly not taken.

Global branch predictor is implemented with the same branch history table, 2-bit counter and an additional global history register. In the global branch predict table, the prediction is initialized to strongly taken. In the global history register, a 4-bit left shift array is used to store the actual result of branch-taking. The data from GHR is used to get a more accurate index of the predication.



By using the short test code that had each branch instruction with a long interval in-between, we were able to test our implementation and obtain a branch prediction rate higher than 80 percent. Testing with the competition code, with 2000 branch counts being issued, ours had around 400 mispredict counts with static not taken prediction. With the predictor, our design reduced the mispredict counts for more than 100. When the branch instruction was constantly issued, the accuracy of the predictor drastically decreased. Specifically, the accuracy of both global and local predictors became zero and stayed at the same state. We suspect the problem lies in the situation when we did not make the branch update signal to only take one clock cycle for each

instruction. Therefore, the prediction was updated when the same branch operation was still being processed.

Mispredict count without TBP	Mispredict count with TBP
428	309

Conclusion

Our group was able to obtain better understanding in the course material through this project. We were able to learn about the processor, cache, and memory and their relationships in more detail. Sometimes, our group could not finish work as planned, and we sometimes faced difficulties. However, we eventually built the working processor and implemented advanced designs in addition. Even though the project was very challenging, it was very rewarding and meaningful at the end. We learned about something that we would not have if we only stayed in the lecture room.