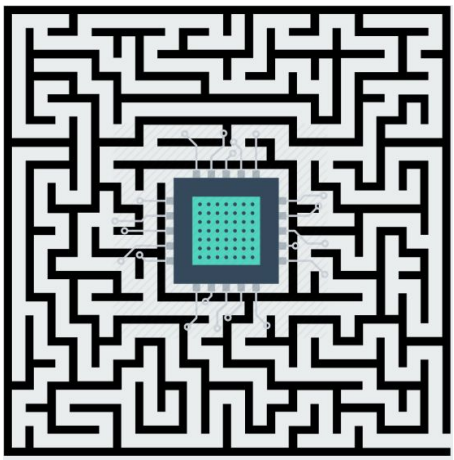

ECE 411 MP3

Maze Processor

By-

Arnav Agarwal

Arvind Arunasalam

Shivam Bharuka

1. Introduction

As a capstone project, we incorporated the concepts learnt in ECE 411 to design a pipelined microprocessor. Our task for this machine problem was to design a 5-stage pipelined processor that supported the LC3b Instruction Set Architecture(ISA) and to incorporate features that helped our processor to efficiently run the test codes without compromising our processor's frequency.

2. Project overview

The processor consists of 5-stages in its pipeline, namely: Instruction Fetch, Instruction Decode, Execute, Memory Access and Write Back. The processor also contains a cache hierarchy with a split L1 cache and a unified L2 cache. It also performs hazard detection (control, structural and data) as well as incorporates data forwarding and branch prediction.

3. Design description

3.1 Overview

The datapath contains registers between each stage to provide instruction-level parallelism in a single processor. Following are the stages of the pipeline with their respective functionality:

a. Fetch Stage

We read the instruction from the memory whose address is specified by the program counter. It also contains a branch prediction unit which assists in predicting which way a branch will go before it is known definitively.

b. Decode Stage

We decode the instruction and access the register file for the registers used in the instruction. Our pipeline is controlled by using a read-only control memory which is propagated through the different stage registers and is accessed in the decode stage.

c. Execute Stage

All arithmetic calculations for each LC3b instruction happens in the execute stage. The execute stage also contains a data forwarding unit which receives values from the memory stage as well as the write back stage.

d. Memory Stage

The memory stage accesses the memory in the case of a load, store and trap instruction. It also generates a stall signal when it waits for a response from memory as well as houses a forwarding unit that gets data from the write back stage.

e. Write Back Stage

The final stage is the write back stage, which commits the values to the register unit as well as resolves the branches and sets the program counter in the case of a mis-prediction.

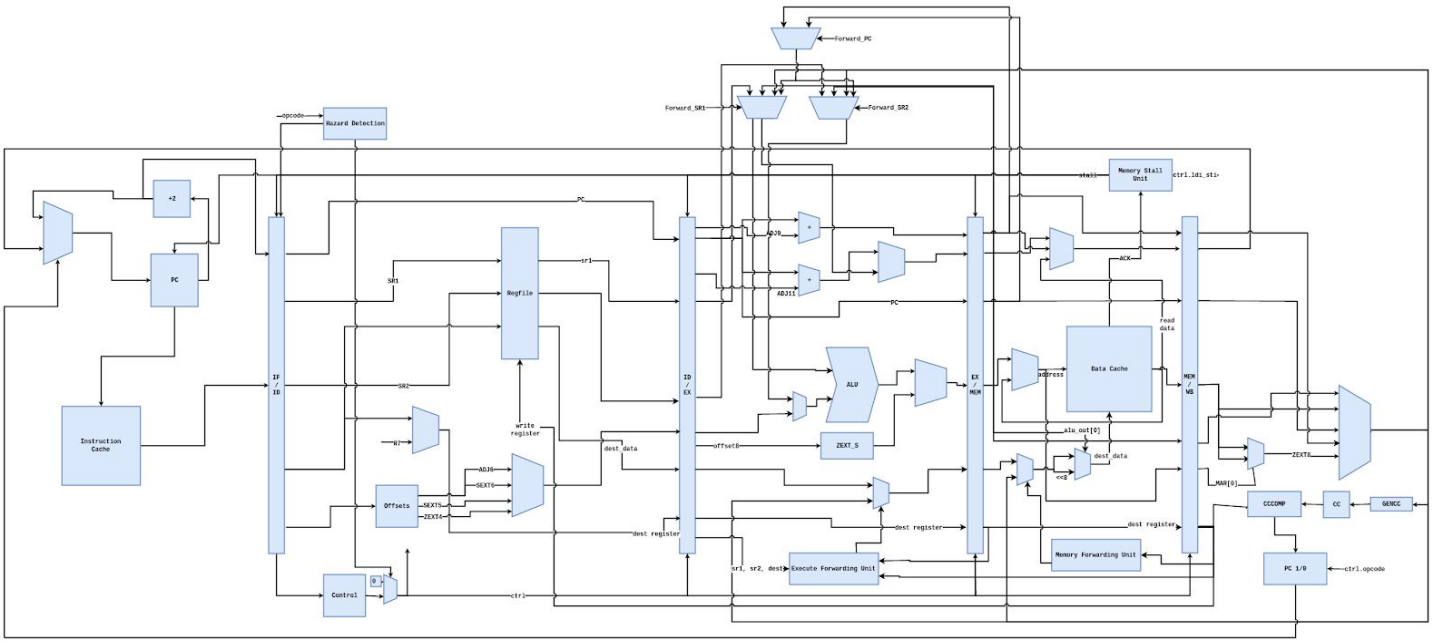

Fig 1: Block Diagram of the Maze CPU

3.2 Milestones

a. Checkpoint 1

We implemented the SystemVerilog code for our pipelined processor. We were supposed to support the LC3-bα ISA, but we also wrote code for all instructions except STI and LDI. We tested the provided test code as well as our own test cases, and they worked as expected. We also discussed and designed the cache hierarchy for the processor and how it works with the shared wishbone even though the L1 cache has separate wishbone master ports. We decided to implement the 4-way set associative L2 cache with LRU policy.

b. Checkpoint 2

We completed the coding for supporting the full LC3b ISA, while also implementing the split L1 cache with the data and instruction cache both being 2-way set associative. We also modified our datapath design to support data-forwarding and hazard detection. We also completed our planning for the implementation of the L2 cache as well as made progress on our advanced design features discussion.

c. Checkpoint 3

We completely integrated our two-level cache hierarchy to the pipelined CPU along with implementing structural, control and data hazard detections to ensure proper functioning of our processor. We also implemented static branch prediction in this checkpoint itself. Out L2 cache is 4-way set associative with a pseudo-LRU.

3.3 Advanced design options

3.3.1 Tournament Branch Prediction

a. Design

We incorporated a tournament branch predictor which tried to guess which way a branch will be resolved. It uses a selector to choose the prediction outcome of either a local or a global branch predictor. The global prediction unit is based on gshare where the history table is indexed by xor of branch PC and outcome of last four branches. The local prediction unit is indexed by 6 LSBs of branch PC. We use a 2-bit prediction scheme in both the local and global history table so

that prediction only changes if it gets mispredicted twice. On branch resolution, the selector is updated based on the prediction correctness of the two predictors.

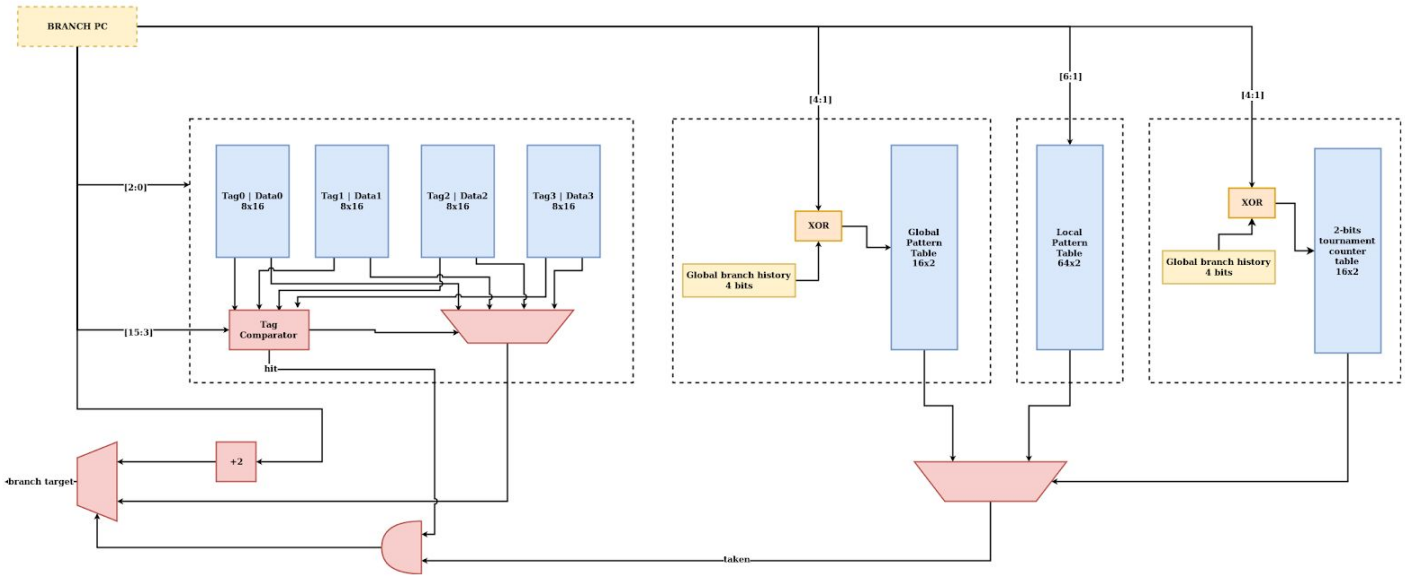

Fig 2: Block Diagram of the Branch Prediction Unit

b. Testing

We performed unit test for both local and global branch predictor using loops in our test code. After verifying the functionality of both the predictors, we performed tests for tournament predictor to see that the right predictor outcome was selected.

c. Performance Analysis

We saw a major improvement in the competition code due to correct prediction in case of both conditional and unconditional branches.

3.3.2 4-way Branch Target Buffer

a. Design

We used a 4-way branch target buffer to store the predicted target of a program counter based on previous outcomes. In case of a branch taken prediction by the tournament predictor and a hit in BTB, the predicted PC is loaded into the next PC. It is updated on branch resolution if the branch is resolved to be taken.

b. Testing

We performed unit tests for BTB without integrating it with tournament predictor by assuming the branch to be always taken in case of a hit in BTB. After verifying the functionality of BTB, we performed integration tests for the whole branch predictor unit.

c. Performance Analysis

We saw a major improvement in the competition code when we performed unit test for BTB.

3.3.3 4-Way Set Associative Unified L2 Cache

a. Design

We decided to use a 4-way set associative write-back cache with a pseudo-LRU policy for our unified L2 cache. Our cache had 16 sets, with each line of size 128 bits, the size of our wishbone, which made the size of our L2 cache 1kB. Our split L1 cache connected to the L2 cache used an interconnect which gave priority to the instructions and serviced data requests when the instructions were being read from the L1 instruction cache. Our decision to use only 16 sets was taken keeping in mind that a larger cache size could potentially reduce our frequency.

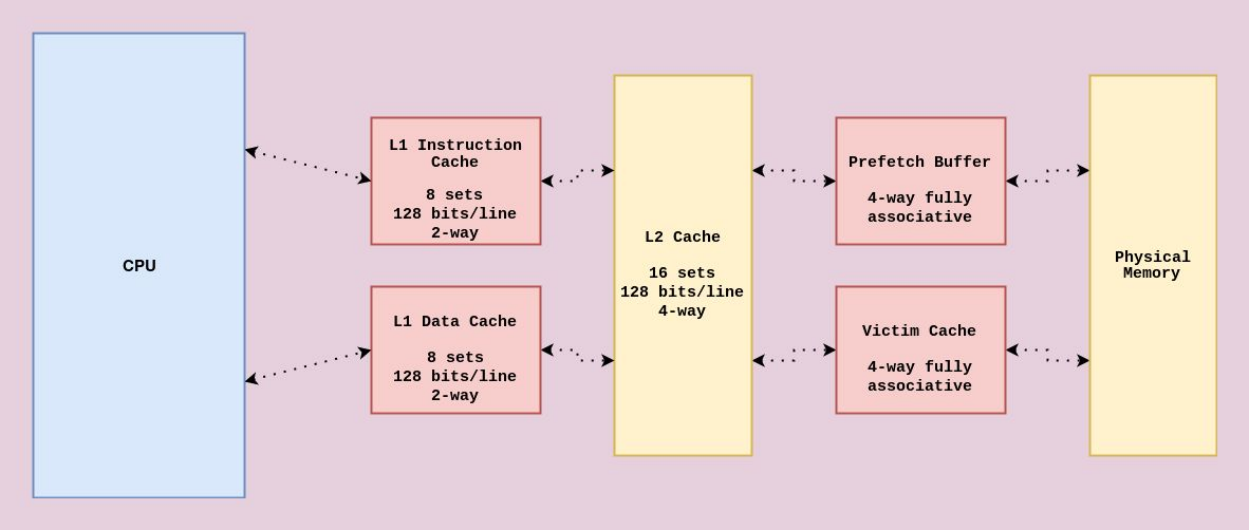

Fig 3: Block Diagram of Cache Hierarchy

b. Testing

Our testing of the cache was done by writing code that accessed several memory locations using load and store instructions. Our testing also verified that the interconnect did not starve either the instruction cache or the data cache.

c. Performance Analysis

Since we implemented our 4-way set associative L2 cache in checkpoint 3 itself, we considered this to be our baseline performance.

3.3.4 4-Way Fully Associative Victim Cache

a. Design

We modified the eviction write buffer to a victim cache so that it stored non-dirty evicted entries in L2 as well. The victim cache is a fully-associative 4-way unit that is designed to decrease conflict misses in L2. It gets populated when data is evicted from L2. If data needs to be evicted from L2, victim cache stores the evicted line. If the victim cache is full then it first evicts the least recently used data to physical memory before storing the evicted data from L2. Otherwise, it stores the evicted L2 data in an empty slot. On a miss in L2, victim cache is looked up. If it's a hit then the data is sent back to L2 with only one cycle penalty. Otherwise, physical memory is accessed to send the data to L2.

b. Testing

We extrapolated our test code for the L2 cache to access more conflicting memories to see if the victim cache worked as expected and stored both dirty as well as non-dirty data from the L2.

c. Performance Analysis

Using the performance counters, we could see that the victim cache did service some requests in all the provided codes and therefore reduced cycle during a miss.

3.4 Other Advanced Design Options

We added few other advanced design options but excluded them from our final submission due to insignificant performance improvement and decrease in clock frequency.

3.4.1 Hardware Prefetching

a. Design

We added a hardware prefetcher in between our L2 cache and physical memory. The responsibility of this unit is to perform sequential prefetching and fetch instructions before they are needed. We use a stream buffer based on one block lookahead (OBL) prefetch which initiates a prefetch for line i+1 whenever the line i is accessed and results in a cache miss. Using a stream buffer, also prevents cache pollution by storing prefetched data in a separate buffer. We use a scheduler to service either the hardware prefetcher request or the victim cache request. The priority is given to victim cache and only when there is no request from victim cache, request from prefetcher is served.

b. Testing

We tested our hardware prefetcher by looking at the stream buffer after a cache miss to verify the next instruction line is prefetched in it.

c. Performance Analysis

We didn't see any performance improvement by adding hardware prefetcher. Our analysis was that whenever the prefetch request is serviced, there is a cycle delay due to a delay in servicing the victim cache request.

3.4.2 Memory Stage Leapfrogging

a. Design

We modified our CPU pipeline to allow independent instructions to jump past the memory stage when there is a data cache miss. This required us to add a unit in the execute stage which looked at the condition codes and forwarded an instruction directly to write-back stage if there is a stall due to memory stage. We also modified our forwarding unit to account for register updates in case of a leapfrogged instruction.

b. Testing

We tested our design by first making sure that all the previous test codes worked correctly. After our initial verification, we performed tests by stalling the memory stage using load and store instructions followed by independent instructions.

c. Performance Analysis

Our goal behind implementing memory stage leapfrogging was to reduce the running time of competition code C. But we didn't see any major improvement in the test code since most of the load/store instructions which resulted in a cache miss were followed by dependent instructions. This design also resulted in a decrease in our frequency so we excluded it from our final submission.

4. Evaluation

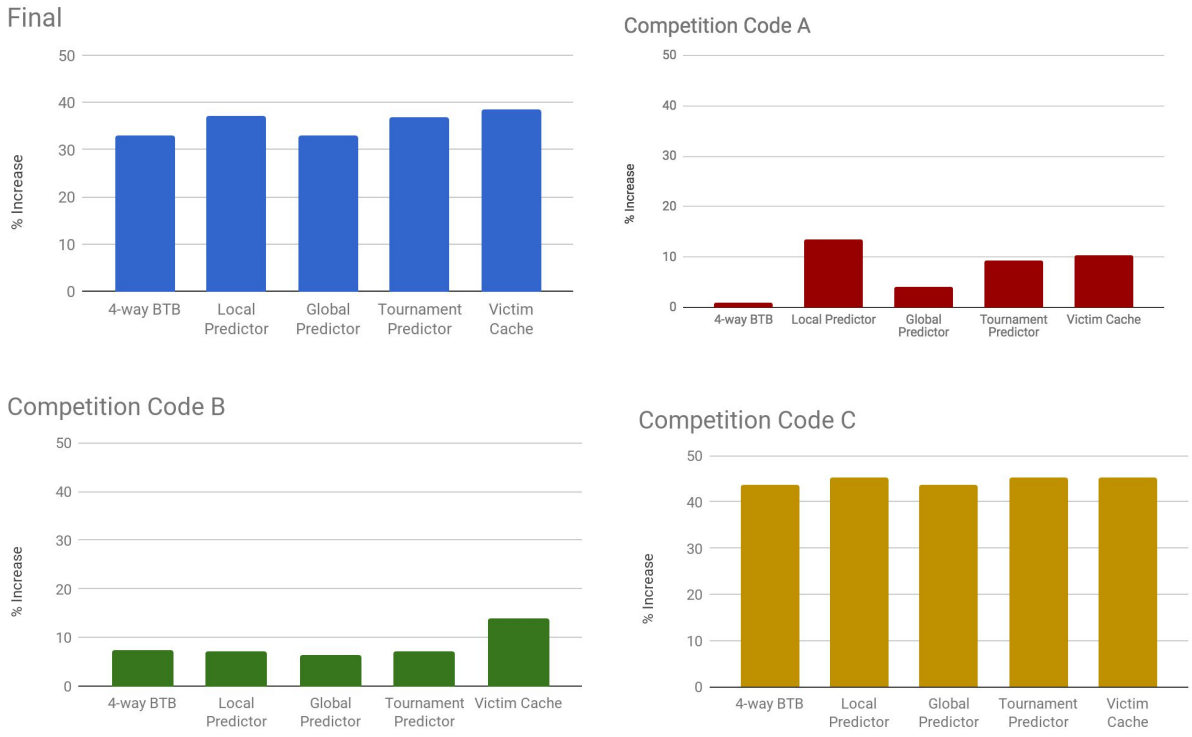

Fig 4: Performance evaluation of four test codes. Each graph shows the incremental increase in performance compared to the base performance obtained after checkpoint 3.

5. Conclusion

Our processor managed to run all the test codes successfully with a frequency of 105.4 MHz. All in all, we were quite happy with our processor's performance.

.