EECS 470 Final Project Report

Group 1 - o3o (Out of Order Onion)

Biro Shin
University of Michigan
biroshin@umich.edu

Ibrahim Musaddequr Rahman
University of Michigan
iamr@umich.edu

Joseph Goslak
University of Michigan
jgoslak@umich.edu

Kai Jun Han
University of Michigan
kaijun@umich.edu

Lani Quach
University of Michigan
laniq@umich.edu

Wayne He
University of Michigan
waning@umich.edu

***Abstract*– This report details the design and implementation of an N-way superscalar out-of-order processor based on the RISC-V instruction set architecture. The processor features early branch resolution, early tag broadcast, a tournament branch predictor with G-share and a per-PC local history predictor, a 4-way associative data cache, and a direct-mapped instruction cache with 10-line prefetching. Our implementation achieves significant performance improvements over the 5-stage in-order pipeline with an average CPI of approximately 1.661 and a clock period of 12.75ns. This report outlines the architecture, discusses the effectiveness of design decisions, evaluates performance metrics, and reflects on project management, providing a comprehensive overview of the development process and outcomes.**

I. INTRODUCTION

Modern computing demands high-performance processors capable of executing complex workloads efficiently. Out-of-order execution, which allows instructions to be processed as their operands become available rather than strictly in program order, has become a cornerstone of advanced processor designs. This project aimed to design and implement an out-of-order RISC-V processor, drawing inspiration from MIPS R10K architecture.

II. DESIGN OVERVIEW

We have designed an out-of-order N-way superscalar R10K-style processor, with advanced features to improve performance such as a tournament branch predictor, instruction prefetching, 4-way set associative data cache, early branch resolution, early tag broadcast, a branch target buffer, and a load buffer-store queue with internal forwarding. We feature a high-level overview of our processor data flow in Figure 1.

A. Fetch & Decode Stage

Our fetch stage consists of two modules – the instruction fetch module and the instruction cache (Icache). Using the current PC, instructions are fetched from memory and stored in the instruction cache for future reference. Decoding annotates these instructions using two additional modules, the branch predictor and the branch target buffer.

Instruction Fetch

Our fetch module is responsible for retrieving instructions from memory by tracking the current Program Counter (PC) and calculating the next PC. For our N-way superscalar design, we fetch N instructions (or one cache line) per cycle whenever possible.

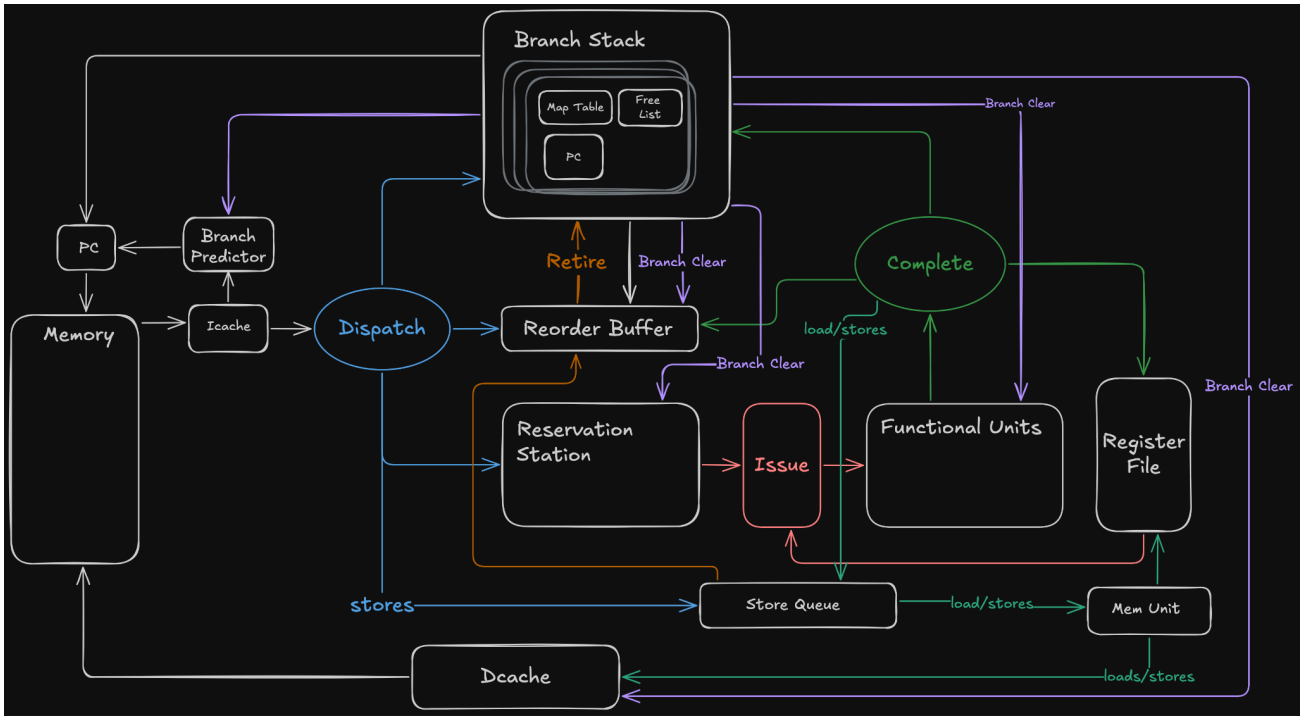


Figure 1. High-Level Overview of Our Processor

Resultantly, the fetch module includes an array of N decoders which help decode N instructions in parallel to enable superscalar execution. Because our fetch module does not lie upon the critical path, all instruction decoding occurs within it prior to dispatch.

On the final processor, our fetch width was limited by the throughput of the Instruction Cache. Our fetch unit currently fetches two 32-bit instructions at once (64 bits). For simplified branch handling, only one branch instruction can be dispatched at a time, and it takes up the full width in the pipeline.

If the fetch module encounters a branch, it will signal the branch stack to save a checkpoint of current state, and annotate the instruction with a corresponding branch ID. If the branch is predicted to be taken, and an address is contained in the BTB, it will jump to the target address. Otherwise, the PC will advance by 4 * (number of instructions fetched).

Instruction Cache

The instruction cache (Icache) acts as the interface between the fetch module and main memory, storing recently requested instructions for lower latency. Our Icache is direct-mapped with 32 8-byte lines and non-blocking. While blocking on a miss, the Icache will continuously prefetch up to 10 lines of instructions from memory to improve latency for future instruction fetches. Icache memory accesses will stall if the Dcache requests memory.

Branch Predictor

Our branch predictor module implements a hybrid predictor scheme that combines both Gshare and local history techniques, with arbitration handled by a tournament predictor. The module receives at most one branch instruction per cycle from the fetch-and-decode stage for prediction.

The Gshare predictor consists of a global branch history register (BHR) implemented as a 8-bit FIFO shift register which is XORed with the PC to index into a 256 line pattern history table (PHT). Each entry is a 2-bit saturating counter.

Our local predictor consists of two components – a 256-lines local history table, with each entry storing an 8-bit history pattern specific to a given PC, and a

corresponding local pattern history table with 256 entries outputs a 2-bit counter to generate predictions.

The tournament predictor is a 256 entry table, indexed by a subset of the PC and mapped to a prediction table which decides which prediction method to use based on a 2-bit saturating counter.

On branch resolution, both the Gshare and local predictors are updated, ensuring correct state at all times, obviating the need for misprediction recovery. If both the Gshare and local predictors predict the same, then the tournament predictor is not updated. However if they differed, the tournament predictor will update to reflect bias towards the correct predictor.

Branch Target Buffer

Our branch target buffer (BTB) serves as a cache for target addresses. When a branch instruction has been fetched and decoded, it is sent to the BTB module to provide quick access to the predicted target address, allowing instruction fetching to continue without waiting for branch resolution. Our BTB is implemented as a 16-line fully associative cache with a FIFO-like replacement policy that updates and evicts only on branch resolution in the Complete Stage.

B. Dispatch Stage

The dispatch stage is a critical component in an out-of-order processor pipeline and serves as the bridge between in-order front-end stages (fetch/decode) with the out-of-order execution backend. Our dispatch stage processes incoming instructions by first performing register renaming in the map table before sending off the instructions to the reservation stations (RS), reorder buffer (ROB), store queue (SQ) and branch stack all in the same cycle.

Dispatch

The roles of our dispatch module are four-fold: (1) It implements register renaming to eliminate false dependencies between instructions by using a priority selector to allocate available physical registers which are tracked in the Map Table and Free List. (2) It determines the appropriate functional unit for each instruction and assigns appropriate metadata at the point of dispatch. (3) It tags instructions with the appropriate branch masks to enable misprediction handling. (4) To handle pipeline stalls, the dispatch module monitors the availability of the RS, ROB and SQ at all times and generates back-pressure when needed.

Branch Stack

The branch stack manages multiple architectural state checkpoints that enable the processor to recover from branch mispredictions efficiently. We implemented the branch stack as the main driver for consistency for Early Branch Resolution (EBR). At its core, the branch stack consists of three main components – the Map Table, Checkpoint Storage Array, and Free List.

Our processor maintains 32 architected registers and 64 physical registers which are tracked by the Map Table through logical-physical register mapping as well as the Free List, which are passed into the branch stack from the dispatch module.

The Checkpoint Storage Array saves the current architectural state (both the Map Table and Free List) at the point when a branch enters the pipeline. This allows for quick misprediction recovery by restoring the processor state to the appropriate checkpoint.

At the same time, the branch stack sends out branch resolution signals by indicating which instructions need to be squashed (on a branch mispredict) or which branch IDs need to be removed from tracking (on a branch hit). By doing so, the branch stack serves as the recovery mechanism that makes speculative execution of branches possible.

Reorder Buffer

The Reorder Buffer has 64 entries and functions as a circular buffer that tracks all in-flight instructions from dispatch until retirement. The ROB is capable of receiving and retiring N instructions per cycle. Each instruction is allocated an entry in the ROB at dispatch time and these entries maintain execution status and necessary metadata.

The ROB uses head and tail pointers to manage this circular structure. New instructions enter at the tail while

old instructions are retired from the head pointer on commit. On a branch mispredict, the ROB walks through all entries and squashes all instructions dependent on that branch (tracked using the branch mask) and rolls back the tail pointer.

For store instructions, the ROB works closely with the store queue by controlling when stores actually update memory through a one bit pop_store signal which allows the ROB to retire at most one store instruction per cycle.

C. Issue Stage

Reservation Station

Our reservation station (RS) has 32 entries and is implemented as a centralized, content-addressable buffer that holds instructions while waiting for their operands to become available for issue. The RS receives N valid instructions from dispatch and allocates entries to store them along with their operand dependency information.

The RS watches the Common Data Bus (CDB) for completing instructions and marks operands supplied by these instructions as ready. Instructions with all operands ready are marked as executable and exposed to the issue module.

Issue

The issue module serves as an interface between our reservation station (RS) and functional units in the execute stage. It represents the transition point from the in-order front-end to the out-of-order execution engine.

The issue module achieves parallel selection of ready instructions through multiple priority selectors that issue ready instructions to their respective functional units.

The issue module will issue as many instructions as the functional units can take, and will recycle ones that were not selected by the Complete logic. Upon issue, store instructions will be sent to the store queue to register the destination address and data and will be marked ready to retire on the ROB in the next cycle.

The issue module implements special handling for load instructions; Load instructions require all stores that are older than it to be marked ready in the store queue before issuing, and the oldest load instruction in the RS will have priority in order to prevent deadlocks. Upon issuing a load instruction, it will compute its address in the FU stage and will be sent to the store queue.

Store Queue

Our store queue (SQ) module has 16 entries and is implemented as a circular buffer which manages memory store operations and provides store-to-load forwarding capabilities. Upon dispatch of store instructions, the tail entry is allocated but left empty.

Upon receiving issuing store instructions, the SQ will mark the matching entries ready and record their address and data. Upon receiving issuing load instructions, the SQ will attempt to forward data from the youngest ready entries that are older than the load.

Forwarding happens at the byte-level such that each load byte undergoes independent logic to derive store data writing to its specific byte-level address. After the forwarding attempt, the load instruction is sent to the memunit, communicating which bytes were not forwarded and need to be read from memory.

Finally, upon retirement of a store instruction, the SQ will pop the head entry and send the store instruction to the memunit to write to memory.

D. Execute Stage

Functional Units

Our functional units contain 3 general purpose ALUs, 2 8-stage multipliers, 1 load ALU, 1 store ALU, 1 branch ALU, 1 stallable branch ALU, and 1 memunit. The general purpose ALUs are used to handle arithmetic instructions (e.g. addi), while the specialized ALUs are used to calculate branch conditions or memory addresses. The branch and stalling branch ALUs resolve the addresses for B-type (e.g. beq) and J-type instructions (e.g. jalr) respectively.

E. Complete Stage

Complete

The complete module manages how completing instructions' destination registers are propagated throughout the processor.

At the heart of our complete module is a priority selector that arbitrates which instructions complete in the current cycle. The selector ensures that the process can complete up to N instructions per cycle and broadcasts their tags on the Common Data Bus (CDB). The complete module sends corresponding stall signals to FUs and the issue module for instructions that were not selected.

Branch instructions that do not write to a register do not need to be placed on the CDB and will always be completed, allowing for N+1 completions in the case of a full CDB and branch resolution occurring at the same cycle. Branch instructions that do write to a register are classified separately and must be selected by the CDB in order to complete. While these instructions can theoretically be stalled by the CDB, we designed the module such that branches are located at the lowest index of the priority selector input bus and loads on the highest index so that the two types will always get priority due to the alternating nature of the priority selector scheme.

Combined with the invariant that there may only be one branch instruction being dispatched at a time, this satisfies the simplification that only one branch is resolved at a time.

Physical Register File

Our Physical Register File (PRF) has 64 entries and supports reading two source registers at a time for each reservation station entry. It has N write ports to support our N way superscalar complete stage. A crucial function of the PRF is its CDB forwarding logic, which allows our processor to select between forwarding newly computed values from the CDB to the RS or returning the stored register value, allowing instructions to receive operands as soon as they are computed, even before they are written back into the register file.

F. Memory Stage

Memunit Module

The memunit module serves as an interface between the processor's execution pipeline and the memory hierarchy. It manages load and store operations, handles store-to-load forwarding and coordinates with the caches to enable proper memory ordering while supporting speculative execution.

Our memunit module operates through a four-state finite state machine that processes memory operations and provides a clean interface to the cache system for handling loads and stores with different memory addresses and sizes (byte/half/word) while taking into account branch squashes and clears.

When receiving a memory instruction from the pipeline, it performs one of three actions: (1) For fully forwarded loads, it returns data immediately (2) For loads requiring cache access, it issues a read request and manages the response from the data cache and (3) For stores, it simply sends write requests to the cache. Once load data is received from the cache, the memunit forwards this data packet to completion and uses a ready-valid handshake to accept incoming instruction from the store queue.

For load instructions arriving at the memunit, they arrive with the memory address to load from, a bitmap indicating which bytes need memory access and partial data that is already forwarded from the store queue. Using the bitmap, a byte level mask is generated which indicates what bytes of data is needed to be merged from the cache and the final value is sign extended based on the load type before being sent to the completion stage.

Data Cache

The Data Cache(Dcache) is the interface that sits between the memory unit and main memory. In our final processor, the Dcache is a 32-line 4-way set associative, write-back cache with a true LRU replacement eviction policy. The Dcache is parameterized to enable 2^k-way set associativity. On eviction from our Dcache, the cache block is written back into main memory based on the dirty bit.

In the event of a cache hit, the processor updates the LRU (Least Recently Used) bits to reflect the access order and modifies the dirty bits of the corresponding cache line if the operation is a store, ensuring accurate tracking of cache state.

Conversely, if the access results in a cache miss, indicating the data is not present in the cache, the processor proceeds to allocate a Miss Status Handling Register (MSHR) entry to track the process of fetching this miss from memory. The Dcache will repeatedly try to request the missed data from memory, then store the transaction tag in the MSHR. Upon data arrival, the Dcache will clear the MSHR and find the least recently used line to evict.

As our data cache is blocking, it will only accept requests after this evict and its potential dirty writeback to memory.

G. Writeback Stage

The retire stage represents the final phase in the processor pipeline where instructions complete their execution. The Reorder Buffer (ROB) governs whether an instruction is eligible for commitment, allowing up to N instructions at the head of the ROB to be committed in a single cycle. The retirement is straightforward: once instructions are flagged as having completed execution by the ROB, they are eligible for commitment without additional constraints. For store instructions, retirement is analogous to writing to memory. Therefore, store instructions may only retire if the memunit is vacant and there is no load instruction being issued.

III. ADVANCED FEATURES

N-way Superscalar

Our processor was implemented as a parameterized N-way superscalar which enables it to dispatch, issue, execute, complete and retire up to N instructions per cycle. In our final submission, our processor was limited to N=2 due to a single read port on the Icache(each line stores two instructions). Higher width could have been achieved by multibanking the Icache, but we determined that the performance increase was not worth the development time.

Instruction Prefetching

Because the reads from instruction memory occur in a regular pattern, latency can be reduced by prefetching multiple instruction lines while Icache is blocked upon a miss. Upon a miss in the Icache, our processor prefetches up to 10 lines of instructions ahead of the current PC value. Similar to the Dcache, transaction tags for each prefetch access are stored in MSHRs, and deallocated once the data is returned by memory. While the Icache is servicing hits, no additional lines are prefetched. Additionally, all MSHRs are cleared upon a branch squash or taken prediction, to invalidate prefetches for irrelevant instructions.

Advanced Branch Predictor

Initially only Gshare was implemented, resulting in a lower branch prediction accuracy than expected. In order to increase our accuracy a tournament predictor was added, as shown by the table "*Comparison of different Branch Predictor Implementations accuracy"*. This approach balances the strengths of each method: Gshare, which refines the global predictor by blending branch PC with global history to reduce aliasing, captures correlations across branches effectively, while the local predictor excels at identifying branch-specific patterns but falters with broader dependencies and requires more storage.

Early Tag Broadcast

Every execution module anticipating completion within that cycle submits a request for a slot on the Common Data Bus (CDB), which updates on the next cycle. With the implementation of ETB, the tags are driven combinationally to mark dependent instructions ready to issue on the next cycle. On the following clock edge, the CDB is updated with its data and intercepts all read requests from issuing instructions awakened by the early tag broadcast bus and forwards the new data.

Early Branch Resolution

Our Early Branch Resolution (EBR) implementation is centered around the branch stack module, which

maintains architectural state checkpoints for speculative execution. When a branch instruction is dispatched, the branch stack allocates an entry containing a map table snapshot, PC information, prediction data, and a branch mask indicating dependencies on other speculative branches.

Each branch is assigned a unique branch ID (bid) that is used throughout the pipeline to track dependencies, with the current set of active branch IDs maintained in a global branch mask. This implementation supports multiple concurrent speculative branches with precise dependency tracking to manage nested speculation efficiently.

When a branch instruction completes execution in the ALU, the branch stack immediately processes the result without waiting for retirement, comparing the actual target address with the predicted one. For correct predictions, the branch stack broadcasts a "clear" mask to remove branch dependencies from other instructions while maintaining speculative state. For mispredictions, it generates a "squash" mask, restores the processor state to the appropriate checkpoint, and provides the correct target PC for instruction fetching.

This selective recovery mechanism cancels only instructions dependent on the mispredicted branch while preserving independent speculative execution. The map table in each checkpoint is continuously updated with register readiness information as instructions complete on the Common Data Bus, ensuring accurate state restoration during recovery. When a branch that contains other branches is squashed, all checkpoints related to the nested state are freed.

TUI Debugger

To aid in debugging the processor pipeline, we developed a debugger with a Terminal User Interface (TUI) that reads in the generated Value Change Dump (VCD) files, and displays the simulated data in a visually aesthetic terminal written entirely in Rust. The TUI debugger displays the contents of many modules per clock cycle and allows for arbitrary signal access. [See Appendix A]

IV. EXPERIMENTAL FEATURES

Due to time constraints or adverse performance effects, there are some features we implemented that we decided not to include in the final design.

Pseudo LRU eviction policy for BTB

The eviction system for the branch target buffer in the final system is a FIFO circular queue which works on BTB sizes that are powers of 2. For more flexibility, we implemented a bit-PLRU policy[1] to vary the size of the BTB freely. However, we concurrently found that in our tests, we were rarely reaching the BTB's max capacity, so the eviction policy was not needed. Additionally, our Dcache already had a true LRU policy, so this eviction policy was never used anywhere.

V. EVALUATION & TESTING

Systems Integration & Testing

Our testing strategy involved two major system integrations. The first phase focused on validating our core pipeline without memory operations. For this test, we created a testbench with a simulated fetch stage that directly provided instructions to the decode stage, deliberately using programs without loads or stores. This simplified setup included a basic branch predictor (always predicting "not taken") that allowed us to verify our branch misprediction handling mechanisms. After successfully confirming this functionality, we moved forward with designing additional individual modules.

The second integration phase was significantly more complex, incorporating memory operations through instruction cache, data cache, and load store queue implementations. Despite the challenges, we were able to get a correct working processor by Milestone 3.

We verified the correctness of our processor by comparing the register write back order, final memory state, and memory writeback order of our processor with the corresponding files generated by the correct Project 3 processor. Our processor passes this verification for all

[1]A. Abel, "Automatic Generation of Models of Microarchitectures," Ph.D dissertation, Saarland, Germany, 2020.

provided test files on all compiler optimizations flags as well as on test files we created to test certain behaviors (e.g. memory operations near the end of the program).

Performance Evaluation

We achieved a final clock period of 12.75 ns with final critical paths in the Branch Stack (for branch resolution) and Store Queue. Pipelining on these modules is possible, but due to time constraints, we were unable to fully implement this in our final design.

Our average CPI across all public tests with no compiler optimizations is 1.661. This, and all further CPI numbers in the report, was calculated by summing the total clock cycles and dividing it by the total instructions retired in all programs.

Summarized Parameters for Our Final Submission

[See Appendix A]

These parameters were selected based on the performance improvements we saw while testing and their effects on synthesis results/critical path length.

CPI across test suite

[See Appendix B]

Our CPI varied based upon the particular workload tested. The highest CPI occurred on test cases that were predominantly branches, as Icache was frequently invalidated. Our best CPI was on test cases with repeated loop patterns, and independent store instructions that could retire in parallel with other execution.

Comparison of Dcache associativity

Ways	CPI	Hit %
2	1.692	93.72%
4	**1.661**	**94.65%**
8	1.658	94.65%
16	1.659	94.60%

Past a certain associativity, the Dcache's hit rate does not change significantly. The parameter selected in our final submission (4) does have a worse CPI compared to an 8-way associative Dcache, but due to the LRU policy used for our cache it was not feasible to create more associative caches without affecting clock period adversely or using different eviction policies.

Comparison of Superscalar Widths

N	CPI	% cycles completing N instructions
2	**1.661**	**13.00%**
3	1.645	3.56%
4	1.669	0.69%
5	1.661	0.03%

Ultimately, our processor has the best performance on N=3, though we did not include this in our final submission because of time constraints. At larger N values we see that the number of cycles where the processor takes advantage of the increased width decreases significantly, diminishing performance gains.

Comparison of different Branch Predictor Addresses accuracy

[See Appendix D]

From our analysis on indexing from different parts of the branch PC for our local and Gshare PHTs, we found the optimal PC address came from indexes [15:8].

Capacity Utilization of ROB/RS

ROB	RS	% full ROB cycles	% full RS cycles
64	8	0.02%	13.54%
64	16	0.08%	9.82%
64	**32**	**0.77%**	**9.29%**
64	64	12.17%	0.01%
128	64	0%	0%

[See Appendix E]

We find that with our clock period, our average ROB and RS utilization is low for most test cases, going up to 49 on test cases with higher percentages of

load and multiply instructions. The optimal ROB/RS size of 128/64 became a critical path, thus the smaller 64/32 was selected as the best at our clock period.

Accuracy of Branch Predictor Implementations

Predictor Scheme	Overall	Large tests
Tournament	**0.854**	**0.917**
Gshare	0.781	0.834
Local	0.787	0.847

On all test cases, our processor had the highest branch predictor accuracy with the tournament style predictor, choosing between Gshare and the local bimodal predictor. On larger test cases such as Alexnet and InsertionSort, the trend followed with the tournament predictor having the best branch predictor accuracy.

ICache Hit Rate

[See Appendix F]

The hit rate of the Icache had varying success rates across the different programs. Programs with more compact instruction footprints such as basic_malloc, matrix_mult_rec, and copy exhibited very high hit rates. This is expected, as their instruction working sets are small enough to remain entirely within the instruction cache.

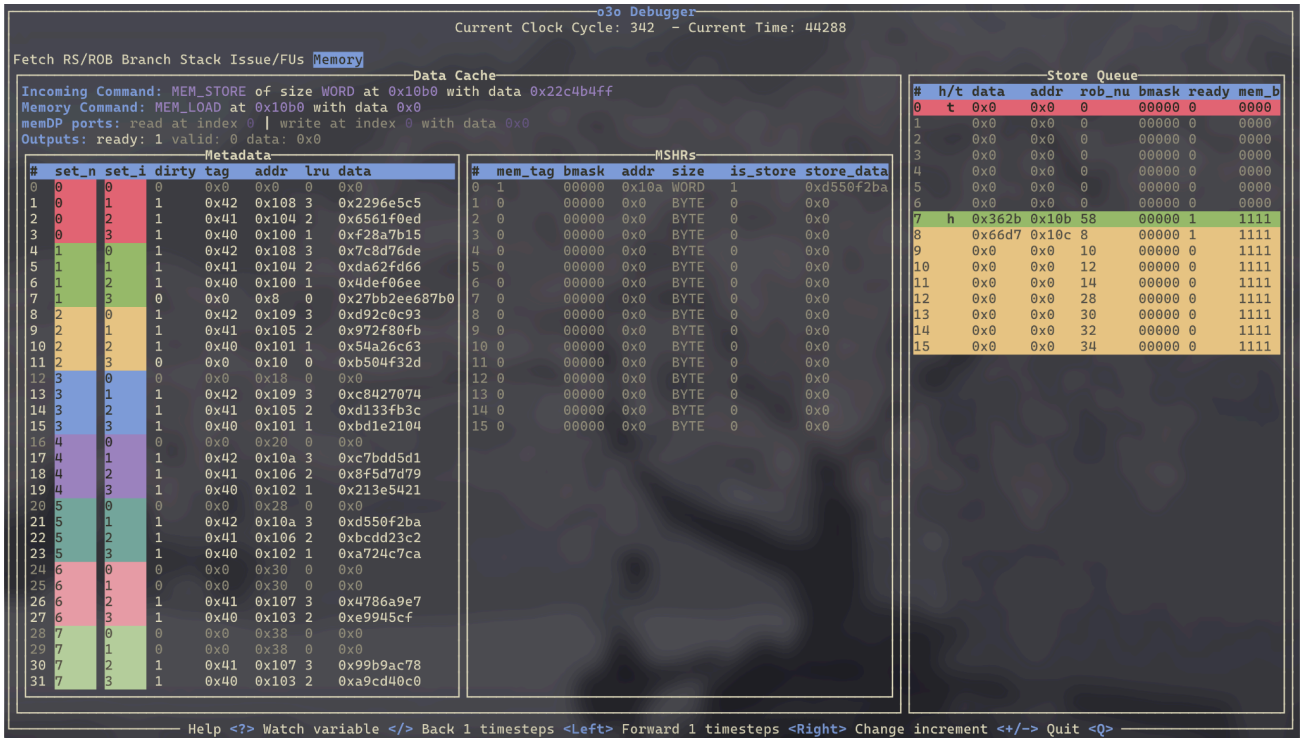
DCache Hit Rate

[See Appendix G]

The hit rate of the Dcache is relatively high on the test cases which run for >10000 instructions, as expected. For test cases which perform many different accesses to memory, like matrix_mult_rec, the capacity of the Dcache becomes a problem. On greater associativities, the hit rate for this test remains approximately the same.

VI. ACKNOWLEDGEMENTS

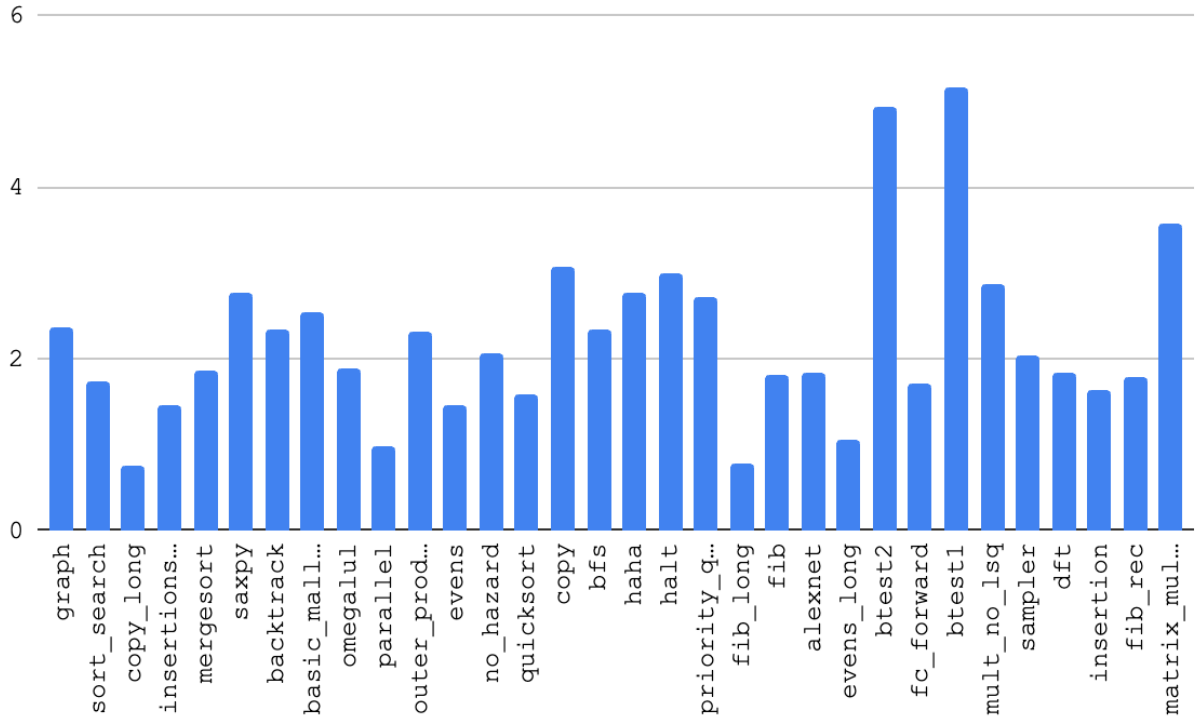
We would like to express our heartfelt gratitude to Professor Dreslinski and Professor Flautner, as well as Bradley, Mustafa, Jaccob, and Jonah for their exceptional guidance and technical support throughout this memorable semester.



Appendix A: Full Screenshot of TUI Debugger

Parameter	Value	Parameter	Value
Superscalar Width	2	*Dcache Lines*	32
RS Size	32	*Dcache Line Size*	8 bytes
ROB Size	64	*Dcache Associativity*	4-way
Physical Registers	64	*Dcache Total Size*	256 bytes
Branch History Register	8 bits	*Icache Lines*	32
Store Queue Size	16	*Icache Line Size*	8 bytes
ALUs	3	*Icache Associativity*	direct-mapped
Multiplier FUs	2	*Icache Total Size*	256 bytes
Multiplication Stages	8	*Load Units*	1
Store Units	1	*Branch Units*	1

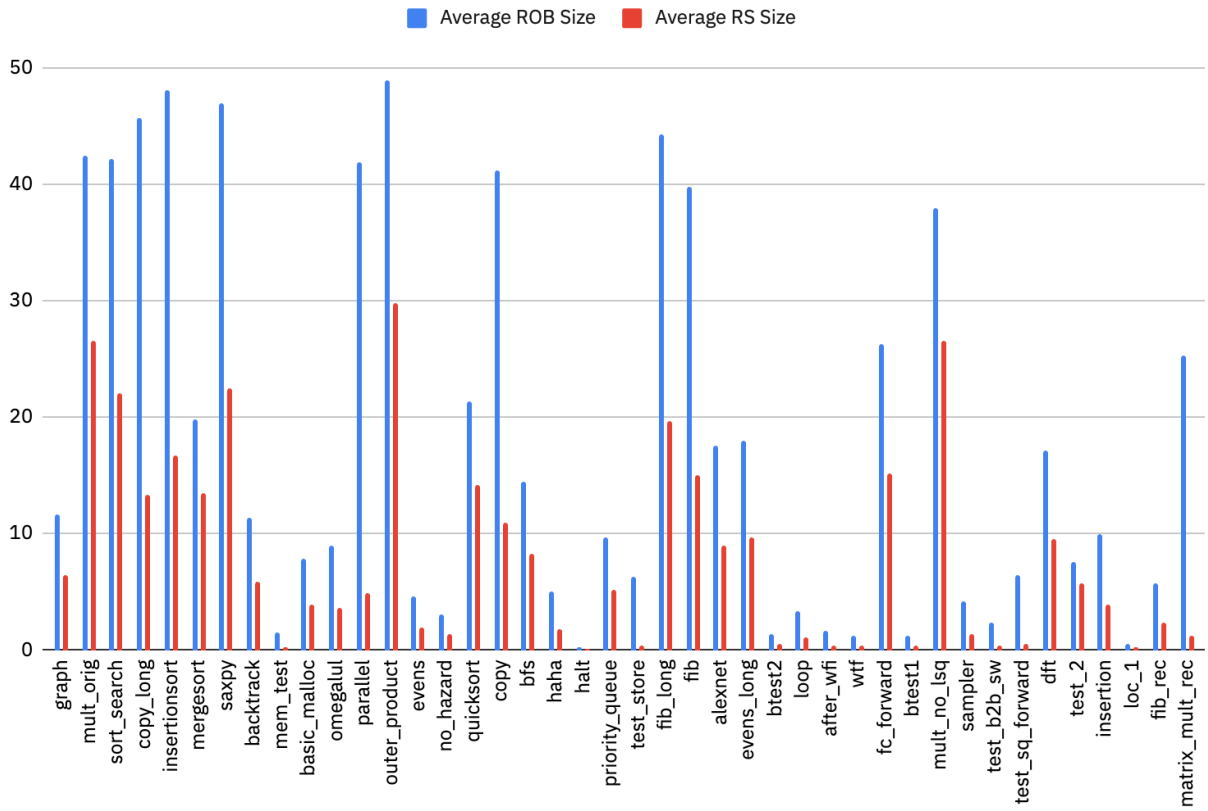
Appendix B: Final Submission Parameters



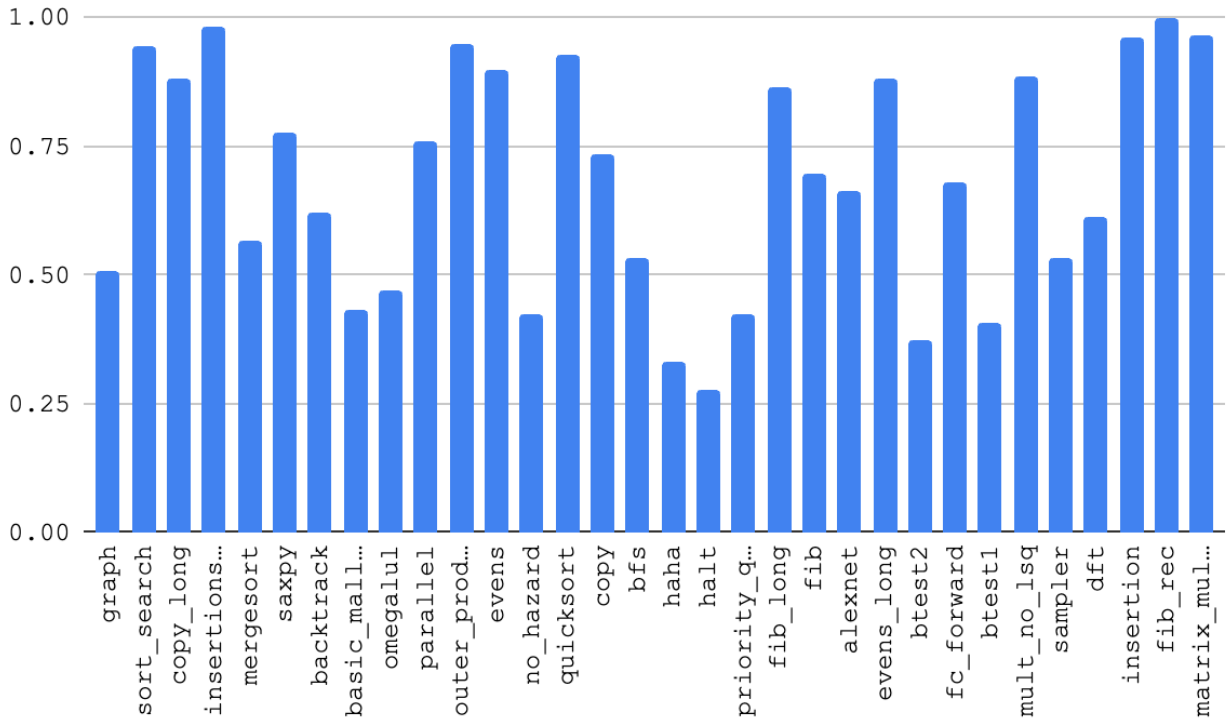
Appendix C: Average CPI Across Our Test Suite

	Tournament [15:8]	Tournament [7:0]	Tournament [19:12]	Tournament [11:4]
alexnet	0.913	0.908	0.479	0.921
backtrack	0.797	0.792	0.623	0.803
basic_malloc	0.779	0.750	0.555	0.826
bfs	0.781	0.736	0.558	0.781
btest1	0.688	0.667	0.667	0.667
btest2	0.839	0.667	0.667	0.667
copy_long	0.938	0.938	0.938	0.938
copy	0.938	0.938	0.938	0.938
dft	0.820	0.796	0.791	0.836
evens_long	0.697	0.667	0.667	0.697
evens	0.697	0.697	0.697	0.697
fc_forward	0.935	0.935	0.935	0.935
fib_long	0.929	0.929	0.929	0.929
fib_rec	0.944	0.880	0.880	0.937
fib	0.929	0.929	0.929	0.929
graph	0.895	0.825	0.673	0.910
insertionsort	0.953	0.945	0.944	0.951
insertion	0.737	0.548	0.548	0.539
loop	0.967	0.967	0.967	0.967
matrix_mult_rec	0.983	0.983	0.971	0.982
mergesort	0.769	0.640	0.670	0.757
mult_no_lsq	0.938	0.938	0.938	0.938
mult_orig	0.941	0.941	0.941	0.941
outer_product	0.950	0.950	0.943	0.950
parallel	0.933	0.933	0.933	0.933
priority_queue	0.847	0.795	0.527	0.851
quicksort	0.865	0.854	0.814	0.870
sampler	0.379	0.414	0.379	0.379
saxpy	0.947	0.947	0.947	0.947
sort_search	0.882	0.892	0.892	0.882
Average:	**0.854**	**0.827**	**0.774**	**0.843**

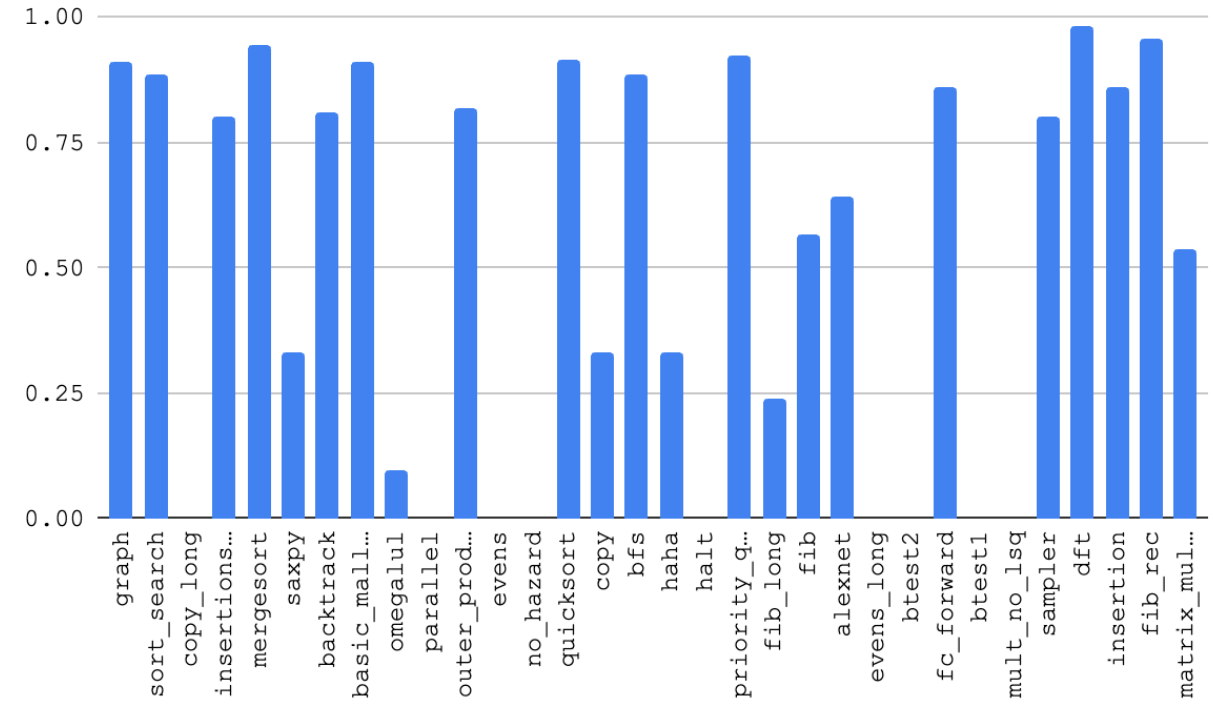
Appendix D: Comparison on the Accuracy of Different Branch Predictor Addresses (% Hit Rate)



Appendix E: Capacity Utilization of ROB/RS (ROB/RS Capacity Against Number of Cycles)



Appendix F: ICache Hit Rate (%)



Appendix G: DCache Hit Rate (%)