EECS 470 Computer Architecture
Final Project: Out-of-Order Processor Design Report

Cadin Cross, Deric Dinu Daniel, Andreas Demoor, Mason Miller, Daniel Werner, Eric Zhang

Abstract

This is the final project report for University of Michigan course EECS 470 Computer Architecture. Our group designed an N-way scalar, R10k-based out-of-order processor, with early branch resolution, early tag broadcast, and a GShare branch predictor.

I. Introduction

Designing and implementing a processor is a perfect introduction to computer architecture. The project posed an immense challenge to our group, especially considering the limited timeframe of 11 weeks. Our processor still has a wide array of improvements we would have hoped to implement given more time. However, given the time allocated, we managed to design an N-way processor with early branch resolution, early tag broadcast, and a Gshare branch predictor.

In this report, we will describe the design, implementation, and performance of our processor in detail. In part II, we will provide an overview of the processor and the design choices made while developing this processor. Parts III, IV, and V will discuss specific features included in our processor. In part VI, we showcase our visual debugger web application. Then, in the final parts of our report, we provide test data and performance analysis on bottlenecks, potential improvements, and reflections.

II. Design Overview

Our group built an out-of-order, N-way superscalar, 32-bit processor based on the R10K microarchitecture, with early branch resolution, fast branch recovery, early tag broadcast, and a Gshare branch predictor, Non-Blocking DCache. Our processor supports the RV31IM 32-bit ISA variant of RISC-V, without fences, division, CSR operations, and system calls.

TABLE 1: Critical Design Specification

RS	ROB	PR	FUs	I$	SQ	D$	BP	BS
16 Entries	32 Entries	32 Regs	3 ALU, 2 Mult, 3 Load, 3 Store, 1 Branch	256 Bytes	8 Entries	256 Bytes	8 Entries	8 Entries

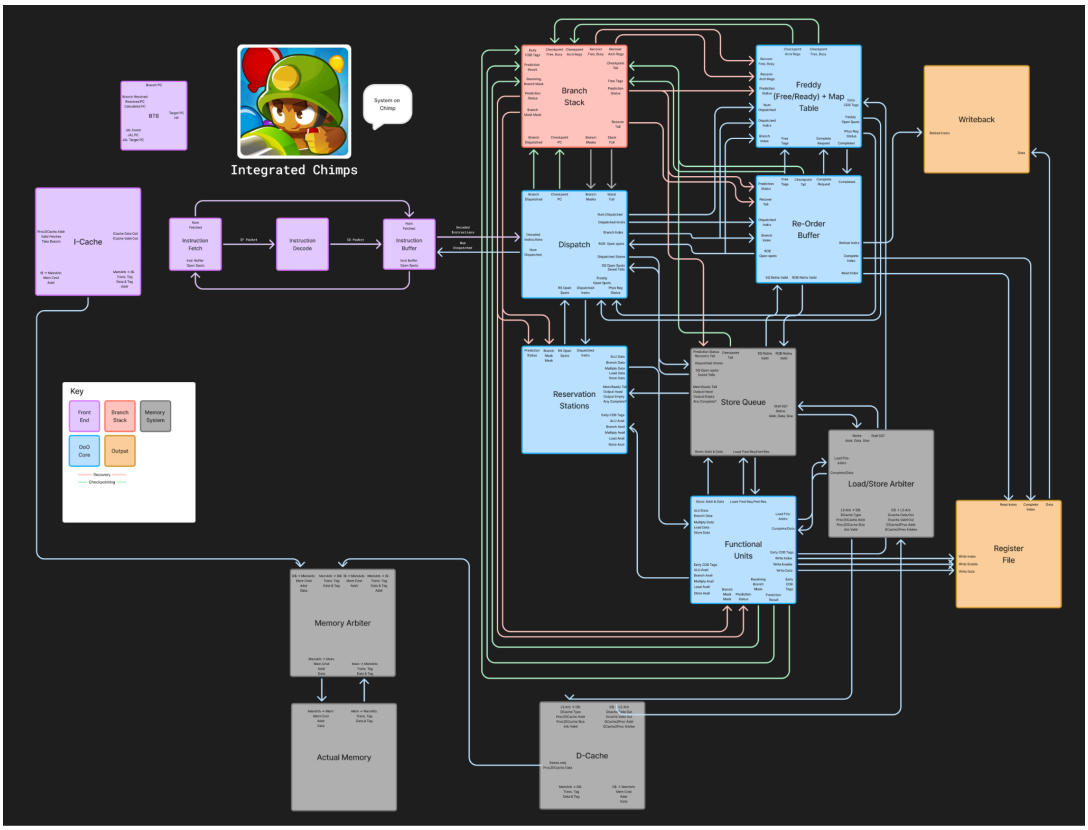


Fig. 1: Top-level Overview of our R10k-based Processor

III. Basic Features

A. Fetch Stage

The fetch stage implements a superscalar design for instruction delivery. The stage consists of a fetcher module working with an instruction buffer. The fetcher maintains an array of program counters for tracking multiple instruction addresses, enabling parallel instruction retrieval from the cache. These fetched instructions enter a circular instruction buffer that acts as an intermediary between fetch and dispatch stages. The buffer, sized at three times the processor width, uses head and tail pointers to manage instruction flow. This organization decouples instruction retrieval from dispatch, handling varying execution rates while supporting mis-speculation recovery. The buffer can accept and retire up to N instructions per cycle, where N is the processor width, maintaining efficient instruction delivery to subsequent pipeline stages.

B. Reservation Stations

The reservation station employs a 16-entry buffer that manages instruction dependencies and issues operations to functional units. The issue logic routes instructions to eight parallel execution units: three ALUs for arithmetic and store address computation, two multipliers, one branch unit, and three load units. The RS matches this diverse functional unit mix with dedicated selection logic for each operation type. When dispatching instructions, the RS allocates entries for up to N instructions per cycle, tracking their physical register dependencies and immediate values. The RS monitors early completion tags through both the CDB broadcast and early tag broadcast, updating dependency information to enable rapid issue once source operands become available. During branch misprediction recovery, the RS uses branch masks to identify and invalidate speculative instructions from the mispredicted path. Upon analysis of our N way issue logic we found that removing it and allowing full way issue hurt our CPI, for N=2 superscalar width we found that the geometric mean of the CPI across all programs increased from 2.9 to 3.8. We would love to do analysis of this data to potentially find a reason why it does hurt our CPI, our speculative reason is that higher issue logic causes worse cbd arbitration. If the functional units are always full, crucial instructions may be delayed on the arbitration hurting the CPI.

C. ROB

The reorder buffer (ROB) is a circular buffer with 32 entries that maintains program order for our out-of-order execution architecture. Each ROB entry tracks the instruction's metadata, including physical register mappings (T_old and T_new), completion status, and branch/store flags. The ROB can process N instructions per cycle for both dispatch and retirement, where N is the processor width. For retirement, the ROB checks completion status and memory ordering - stores can only retire when signaled by the store queue, and branches must retire before subsequent stores. During branch misprediction, the ROB invalidates all entries after the mispredicted branch and restores the tail pointer to recover architectural state. This maintains program correctness while enabling out-of-order execution.

D. Frizzy Table

The register renaming system consists of the Frizzy Table, which combines the free list and ready bit table. The free list tracks 64 physical registers, with architectural registers marked as not free on reset and remaining physical registers available for renaming. The table maintains a count of free registers and can allocate up to N registers per cycle for dispatch, with availability signaled through a priority selector network. The ready bit table tracks completion status of physical registers, updating through both the CDB broadcast for newly completed values and T_old signals from the ROB for freed registers. For branch prediction, the table creates checkpoints of both the free and ready states, which can be restored during misprediction recovery. The free and ready states in the branch stacks are updated along with the actual state. The system supports internal forwarding - newly freed registers can be reallocated in the same cycle, and just-completed values on the CDB are immediately marked as ready. For register mapping, the design maintains a map table with checkpointing capability for branch recovery, allowing us to not need an architectural map tap.

E. Functional Units

The functional units execute operations for different instruction types, providing parallel execution capabilities. The processor contains eight functional units: three ALUs for arithmetic, logic, and store address generation; two multipliers for multiplication operations; two load units for memory reads; and one branch unit for control flow operations. The ALUs complete operations in a single cycle, implementing standard arithmetic operations (add, subtract), logic functions (and, or, xor), and shifts through a combinational circuit. The multipliers use a 4-stage pipeline design to compute 32-bit multiplication results, with partial products generated and accumulated each cycle. The branch unit evaluates control flow conditions in a single cycle, determining branch outcomes for conditional branches and

calculating target addresses for jumps. Each functional unit connects to the common data bus for result broadcasting and receives operands from both the physical register file and immediate values. The units also participate in branch recovery by tracking branch masks and invalidating operations from mispredicted paths.

IV. Advanced Features

A. Early Branch Resolution

The branch recovery system employs an early branch resolution mechanism to minimize the performance impact of mispredictions. The processor maintains a branch stack with 4 entries that stores checkpoint information for each outstanding branch instruction. When a branch is dispatched, a unique branch mask identifier is assigned to the branch and to all subsequent instructions, enabling quick identification of dependent instructions. Branch execution can begin as soon as source operands are available, allowing branches to resolve well before they reach the head of the reorder buffer. Upon misprediction detection, the system triggers an immediate recovery process - invalid instructions are cleared from the reservation stations and ROB based on their branch masks, and architectural state is restored using the checkpointed data from the branch stack in a single cycle. Through early tag broadcast, the processor can also eagerly free physical registers and invalidate dependent instructions as soon as the branch outcome is known, rather than waiting for normal pipeline completion and retirement.

B. Early Tag Broadcast

The early tag broadcast mechanism optimizes data flow dependencies by broadcasting destination register tags as soon as instructions enter functional units. The processor employs a dual-tag broadcast system - an early tag broadcast that happens when instructions begin execution and a regular tag broadcast when results are available. Each functional unit can request access to both broadcast paths through a priority selector network that arbitrates between parallel requests. For ALUs and branch units which complete in a single cycle, early tags are broadcast at the same time those instructions are issued. Multi-cycle units like multipliers broadcast early tags several cycles before completion. This system allows dependent instructions to recognize when their source operands are being computed, enabling them to monitor the common data bus for incoming results. By alerting the reservation stations early, this mechanism reduces effective instruction latency and improves throughput, particularly for instruction sequences with tight dependencies. One large improvement we found while implementing ETB was back pressure within the multiplier.

This allows instructions to continue to feed through the front of the multiplier while the back stalls, mitigating the impacts of not being arbitrated.

C. Branch Predictor

The branch prediction system employs a dual-predictor approach combining a branch target buffer (BTB) for target address prediction and a Gshare predictor for direction prediction. The BTB is implemented as a 1024-entry direct-mapped cache with 8-bit tags, storing target addresses for branches and JALR and JAL instructions. When a branch or jump instruction is fetched, the BTB is accessed in parallel using the low-order bits of the PC for indexing, providing the predicted target address if a hit occurs. The Gshare predictor uses a 3-bit global branch history register (BHR) XORed with branch PC bits to index into a pattern history table (PHT) containing 2-bit saturating counters. The PHT starts with weakly not-taken predictions and updates based on actual branch outcomes. For branch recovery, the system maintains checkpoints of both the BHR state and branch target information. On a misprediction, the processor restores the checkpointed BHR, updates the relevant PHT entry based on the actual outcome, and updates the BTB with the correct target address. The integration of BHR checkpointing with the branch stack enables efficient recovery while maintaining accurate prediction state.

D. Store Queue

The store queue maintains program order for memory operations with a 8-entry circular buffer managed by head and tail pointers. When dispatching stores, the queue assigns sequential positions to maintain program order and provides these indices to load instructions, enabling them to track their ordering relative to outstanding stores. The queue supports store-to-load forwarding by allowing loads to search for completed stores targeting the same address, with support for the same data widths (byte, halfword, word). Upon receiving completed store addresses and data from the store functional units, the queue marks entries as ready for retirement. Stores can only retire when they reach the head of the queue and receive explicit retirement permission from the ROB. During branch misprediction recovery, the queue leverages checkpointed tail pointers to restore precise state by invalidating speculative stores past the recovery point.

V. Memory Interface

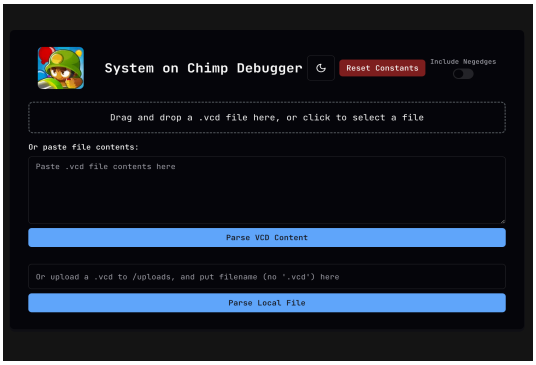
A. Instruction Cache

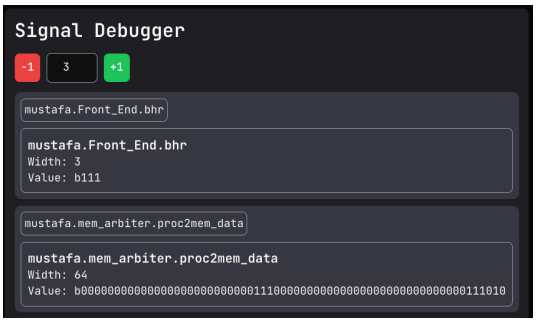
The instruction cache is designed as a direct-mapped cache with 32 entries that supports parallel access for multiple instructions. Each cache line contains a 64-byte memory block and a tag field with a valid bit. To support superscalar fetch, the cache provides up to N instructions per cycle where N is the processor width. When a cache miss occurs, the cache issues a load command to memory and blocks further fetch requests to that line until the memory response returns. The cache handles branch mispredictions by invalidating outstanding memory requests and clearing the valid bit of affected cache lines, preventing stale instructions from being fetched after recovery.

B. Data Cache

The data cache implements a direct-mapped write-back design with 32 entries and an MSHR (Miss Status Holding Register) to provide non-blocking functionality for handling multiple outstanding misses. The cache supports byte, halfword, and word operations with proper alignment checking. For loads, the cache checks for hits in parallel and returns data immediately if found, otherwise allocating an MSHR entry and issuing a memory request. Store data updates the cache line if present, otherwise allocating an MSHR entry to fetch the line from memory. The cache coordinates with the store queue for proper memory ordering, ensuring stores only modify cache lines once they are ready to commit. Stores to memory are delayed by only writing to memory when previous loads to memory evict current cache entries.

VI. Visual Debugger





The visual debugger is a full stack application built using Flask with Python on the backend, and Next.js (React.js framework), in which our team was able to visualize all values and states of everything, enabling rapid bug detection and integration.

A. User Steps

To be able to use the debugger, the user must enable dumping of signal values to .vcd files by adding $dumpfile(“<filename.vcd>”); $dumpvars(0, <testbench name>);. This will tell the simulation to dump all signal values everytime a signal changes in the specified module and all children modules. The user can then upload the .vcd file to debugger by copying it over to the specified folder, and hit the parse button to begin the VCD parsing. Once the VCD is done parsing, the site will redirect to the debugger page with all the modules.

The header provides a link back to the home page, theme switcher, constants editor, cycle control buttons, include negative edge toggle, tag searcher, and cycle count. The theme switcher lets users control the color theme. The constants editor lets users edit constants if needed, but this is not really needed as constants are auto-detected from the signal lengths. The cycle control buttons control which cycle is currently being displayed. The negative edge toggle enables debugging negative edges (very helpful for combinational logic like dispatch).

The tag searcher allows the user to type in a physical register tag, and see it highlighted everywhere the tag appears, to follow an instruction through the CPU. Finally, the cycle count displays the max number of cycles parsed, and current cycle displayed.

In the actual debugger display, there are two main sections. First there are all the module displays. There is a debugger for (almost) every module present in the CPU. Within each module, the inputs, outputs, relevant internal data structures and signals are all displayed. Each module has unique, custom built visualizations, color highlighting (valid, ETB hit, and several more), and are all collapsible. The second section at the bottom includes a raw signal debugger which displays the raw bit representation of any signal, and below that a hierarchy of all signals.

B. How It's Built

We found a Python package to parse VCD files into a hierarchical structure, so we chose to use Flask/Python on the backend. The backend provides a parse API to parse the VCD files which uses the package to parse into memory, then break the VCD down into cycle by cycle data and stored in the Flask cache database.

The frontend is built in Next.js (a React framework), Typescript, Tailwind, and Shadcn UI. The frontend sends a file to get parsed and then requests data for any cycle. For each cycle, the backend provides the data of every single signal in the same hierarchy. The frontend can then parse the raw binary signals into Typescript objects which can be used to display the content as desired.

C. Effects on Group Productivity

The debugger immensely accelerated our group's ability to find bugs for writing individual modules and especially for integration. The easy visualization, color highlighting, tag search, raw signal debugger, and overall ease of use made it a worthwhile investment. It did take a lot of time to write, took away a lot of a person's time from writing the actual processor, and made us depend on it. However, it was fully worth it as the very subtle bugs we found would've taken much longer to find without it.

VII. Testing

This section of the report will detail the overarching test strategies and debugging workflow that our group employed to ensure and maintain correctness during development.

A. Test Strategy

The testing framework was built on a modular development approach, where each component is first validated independently before integration. This method allowed us to isolate and fix issues early in the development cycle while maintaining a clear understanding of each module's behavior. The workflow begins with individual module implementation, followed by extensive simulation testing to verify functionality. After passing simulation tests, modules undergo synthesis verification to ensure timing and resource constraints are met. Only after a module has passed both simulation and synthesis validation is it integrated into the larger system, where integration testing verifies correct interaction with other components. During the integration phase, we reuse our units when making changes to modules to ensure regression doesn't occur. In addition, we utilize integration testbenches to test large integrated components, as well as simple assembly programs to test entire pipeline integration. Finally, we use regression tests on large C and assembly programs to ensure our changes do not cause issues.

B. Testbenches

Our testbench suite consists of two types of verification environments: module-specific testbenches and system integration testbenches. Module testbenches, exemplified by our Gshare predictor tests, implement targeted test cases that verify specific functionality and edge cases. These tests include state transition validation, error handling, and performance under various conditions. The comprehensive CPU integration testbench serves as our system-level verification platform, coordinating memory interfaces, instruction execution, and architectural state verification. This testbench includes mechanisms for monitoring internal state, tracking cycle-accurate behavior, and collecting performance metrics including CPI calculations and memory state verification.

C. Test Cases

Our testing approach combines comprehensive public test cases with targeted unit tests designed to isolate specific functionalities. The initial test suite encompasses various

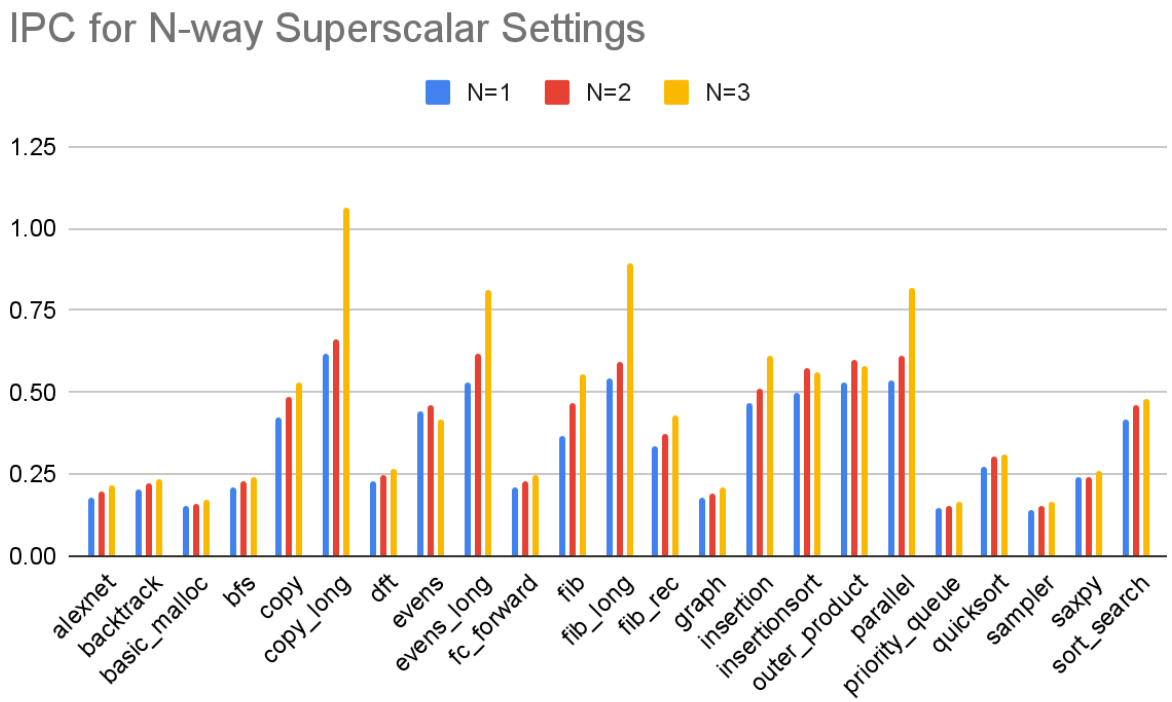
computational benchmarks, including matrix multiplication, sorting algorithms, and memory-intensive programs that exercise different aspects of the processor architecture. Upon encountering failures in these complex test cases, our team develops focused micro-benchmarks to reproduce and debug specific issues. For example, to validate store-to-load forwarding logic, we created concise test programs containing carefully ordered loads and stores that exercise edge cases in memory ordering and data forwarding. We applied this same methodology to develop targeted tests for other critical components, including branch prediction, memory interfaces, and early tag broadcast. These minimal test cases, typically consisting of 5-10 instructions, provide clear visibility into the processor's behavior and enable rapid debugging cycles. Through the combination of broad system tests and focused unit tests, our testing methodology ensures both comprehensive coverage and efficient debugging capabilities.

D. Shell Scripts

We developed a suite of parameterizable shell scripts to automate and streamline our testing process. These scripts enable rapid testing across multiple configurations by automatically varying key parameters like superscalar width (N) and module sizes. The automation framework compiles test programs, runs simulations, and compares outputs against golden reference files. Our primary test script tracks both register writeback and memory output correctness while reporting CPI metrics. The scripts include features for continuous integration testing, with capabilities for automated regression testing whenever code changes are detected. This automation significantly improved our testing efficiency and helped maintain consistent verification coverage throughout development.

VIII. Performance Analysis

A. N Way



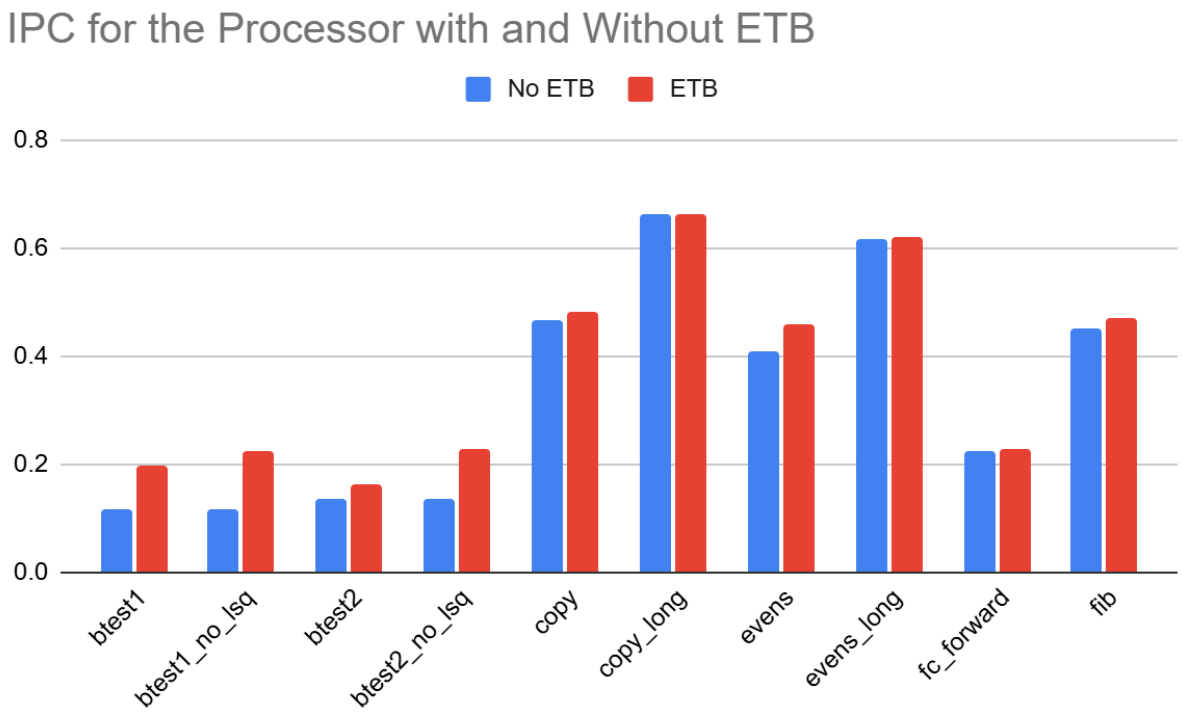
We implemented N-way superscalar in order to improve our instructions per cycle. The graph above shows the IPC for the public test cases at different values of N. The IPC increases overall from N = 1 to N = 3. The IPC increases significantly for large programs such as copy_long, fib_long, and even_long, as these programs have many noop instructions which do not cause any stalling. The IPC of some programs does not increase significantly for some programs such as alexnet, backtrack, and basic_malloc. This bottleneck could be caused by our branch prediction being too low due to us only fetching and dispatching one branch at a time. Another bottleneck to our N-way superscalar implementation is that we only perform one access to memory per cycle, whether it is instruction fetches, load instructions, and store instructions.

B. Early Branch Resolution

While we did not know of a way to disable early branch resolution and compare a processor that uses EBR to one without it, we believe that implementing early branch resolution greatly improves our performance. EBR allows us to complete instructions out of order so that we do

not need to wait for branch instructions to retire in order. In addition, using checkpointing allows us to recover from a mispredict in one cycle, instead of needing to do a serial rollback.

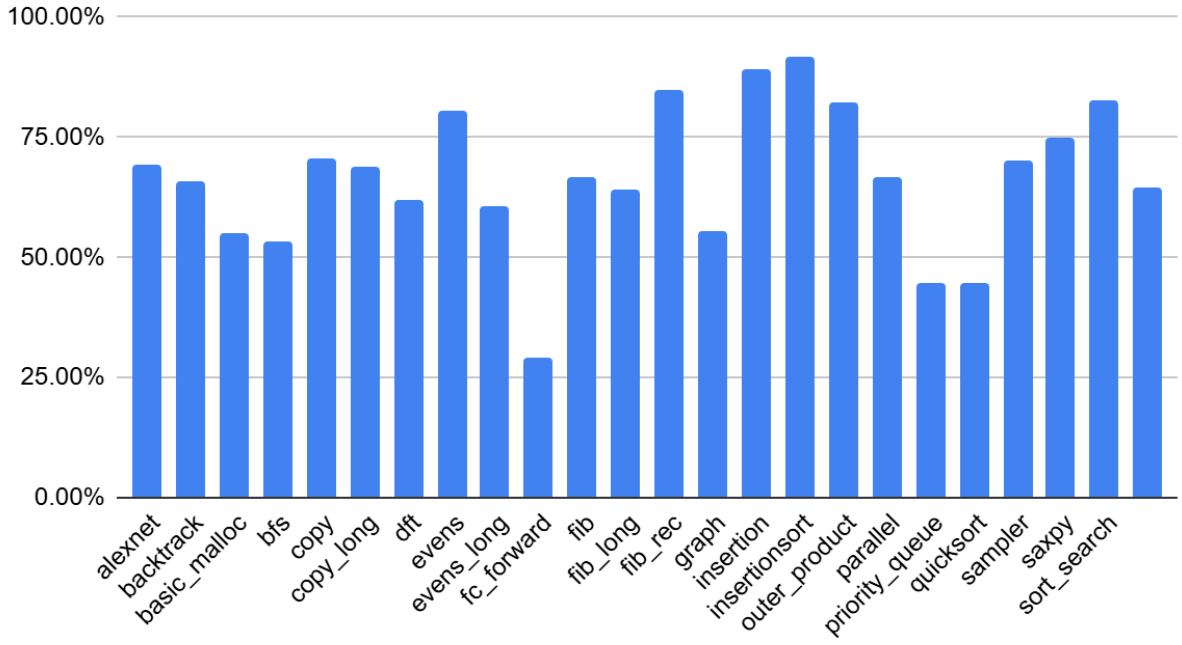
C. Early Tag Broadcast



This graph compares the IPC of various tests without ETB (blue) and with ETB (red) when our superscalar width (N) is 2.

Our initial decision to do Early Tag Broadcast was motivated by the promise of IPC gains. In order to properly test ETB's impact on our processor, we created a one cycle delay for tag broadcasts to simulate what our processor would perform like without ETB. Generally, we found that for all programs, a processor with ETB performed better than one with ETB removed.

D. Branch Predictor



This graph compares the accuracy of our branch predictor across various tests.

Our Gshare branch predictor was able to achieve an average prediction accuracy of 65.17% for N=2. We have an extremely high accuracy of 91% for insertion sort, due to too many predictable for loops. We have an extremely low accuracy of 29.27% for fc_forward due to our branch predictor alternating between taken and not taken.

IX. Further Optimization

A. Optimize Critical Path

Our processor passed correctness, but we did not have enough time to do further optimization to our critical path. Ideally we would be able to find the critical path and work to reduce the logic flow within that section if we had more time. We presume that the critical path exists within the store queue-data cache handshake before data is forwarded to the load functional units.

B. Byte-Level Forwarding

With more time, we could have implemented byte-level forwarding in our Store Queue. Currently, our processor only forwards data from Stores to Loads when they are of the same size. When a Load has a matching address to an entry in the Store Queue, and it is of a different type, it stalls the load until the conflicting stores have been retired from the Store Queue and ROB.

C. Aggressive Prefetching

We had the structure in place for an aggressive prefetcher that would continue to fetch until another memory access was placed. We removed this feature from our processor in order to ensure correctness by the deadline. Given more time we would like to integrate this feature in and see what the performance benefits would be. We expect them to be drastic due to the fact that dispatch stalls primarily on the instruction fetcher instead of any other component.

X. Team Logistics

For the first milestone, we did not set clear goals for the milestone. We had decided on our weekly meeting time, but did not properly plan out time for testing. For the second milestone, we decided to create a Gantt chart detailing all of the tasks for the milestone. We allocated plenty of time for testing and module integration in order to meet the milestone on time. For milestone 2, we met each week and modified our plan for the remainder of the milestone. Throughout the milestone, we encountered collaboration and communication issues as at times, multiple team members would work on the same task without another person knowing, causing tasks to be redone. For milestone 3, we made sure to strictly outline which individual was assigned to what task, and to clearly communicate this to all team members if plans change. A breakdown of tasks completed by each team member is shown below.

- Cadin Cross (16.6%): Early Tag Broadcasting, Reservation Station, Instruction Cache, Dispatch, Branch Target Buffer, and Front End Integration
- Deric Dinu Daniel (16.6%): Visual Debugger, ROB, Out-of-Order Core Integration, Store Queue Design, and Load & Store Functional Unit Design
- Andreas Demoor (16.6%): Branch Stack, Freddy Table, Out-of-Order Core Integration, Store Queue, Memory Integration, and Debugging
- Mason Miller (16.6%): Gshare Branch Predictor, Branch Predictor Integration, Front End Integration, Test Infrastructure, Early Branch Resolution Testing, and Debugging
- Daniel Werner (16.6%): GShare Branch Predictor, ROB, Out-of-Order Core Integration, Memory Integration, and Debugging
- Eric Zhang (16.6%): Data Cache, Freddy Table, ROB, Memory Arbiter, Branch Stack, Load Store Arbiter, Memory Integration, and Debugging

Appendix A
Public Testcase List

Below is a list of all public assembly and C programs used to test our final pipeline.

- alexnet.c
- backtrack.c
- basic_malloc.c
- bfs.c
- btest1_no_lsq.s
- btest1.s
- btest2_no_lsq.s
- btest2.s
- copy_long.s
- copy.s
- dft.c
- evens_long.s
- Eve s.s
- fc_forward.c
- fib_long.s
- fib_rec.s
- fib.s
- graph.c
- haha.s
- halt.s
- insertion.s
- insertionsort.c
- matrix_mult_rec.c
- mergesort.c
- mult_no_lsq.s
- mult_orig.s
- no_hazard.s
- omegalul.c
- outer_product.c
- parallel.s
- priority_queue.c
- quicksort.c
- sampler.s
- saxpy.s
- sort_search.c

Appendix B
Example Script

```
#!/bin/bash

echo_color() {
    if [ -t 0 ]; then tput setaf $1; fi;

    echo "${@:2:$#}"

    if [ -t 0 ]; then tput sgr0; fi
}
module=$1

for i in {1..4}
do
    for j in {16,32,64}
    do
        if [ $j -ge $i ]; then
            echo "Starting $module test with N=$i, ROB_SZ=$j"
            cmd="make -B build/$module.simv N=$i ROB_SZ=$j"
            echo "Compiling simulation as: '$cmd'"
            $cmd &> /dev/null

            cmd="make $module.out N=$i ROB_SZ=$j"
            echo "Running testbench as: '$cmd'"
            output=$($cmd | grep "@@@")

            if [[ $(echo $output | grep "Passed") ]]; then
                echo_color 2 $output
            else
                echo_color 1 "@@@Failed"
            fi
        fi
    done
done

```

This Bash script streamlines the testing workflow for processor modules by automating the compilation and execution of test cases across a range of parameter configurations. Specifically, the script iterates over combinations of the superscalar width and reorder buffer sizes, filtering out invalid configurations where the ROB size is less than the superscalar width. For each valid configuration, it compiles the simulation using a make command, executes the corresponding testbench, and extracts the results from the output. Test outcomes are presented in a color-coded format—green for "Passed" and red for "Failed"—to enhance readability and facilitate quick issue identification. This automated process ensures consistent test coverage, reduces manual effort, and accelerates the debugging and verification of processor module functionality.

Appendix C

Example Test Case

```
# Test 7: Multiple stores
test_store_sequence:
    li x2, 0x1000       # Base address
    li x1, 0x42
    sb x1, 0(x2)        # Store 0x42 to byte 0
    li x1, 0x76
    sb x1, 1(x2)        # Store 0x76 to byte 1
    li x1, 0x89
    sb x1, 2(x2)        # Store 0x89 to byte 2
    li x1, 0xAB
    sb x1, 3(x2)        # Store 0xAB to byte 3
    lw x3, 0(x2)        # Should load 0xAB894276
    wfi
```

Our team developed targeted test cases to validate the complex interactions between the store queue and load unit, especially focusing on store-to-load forwarding and memory ordering. One key test validates the processor's ability to handle multiple byte-sized stores followed by a word-sized load to the same memory location. This test begins by storing four individual bytes (0x42, 0x76, 0x89, and 0xAB) to consecutive memory addresses starting at 0x1000, followed by a load word instruction targeting the same address range. This sequence exercises several critical aspects of our memory subsystem: the store queue's ability to coalesce multiple byte stores into a single memory location, the load unit's forwarding logic when encountering multiple preceding stores, and the proper handling of different access granularities (byte stores versus word loads). The expected result of 0xAB894276 verifies that all stores are correctly ordered and that the load unit properly reconstructs the full word from the individual byte stores, whether through forwarding from the store queue or accessing memory directly. This test proved particularly valuable in identifying edge cases in our forwarding logic and ensuring precise memory ordering semantics in our out-of-order execution environment.

Acknowledgements

Finally, we would like to thank the course staff for an amazing semester. We appreciate the endless support and encouragement we received throughout the project, and especially towards the end!