Cache Me Outside
EECS 470 F25 Final Project Report

Guanlin Li, Carolyn Liu, Yousef Maitah, Lawrence Han, Venkatkrishnan Iyer
{guanlin, carolliu, yousefm, hanlb, venkiyer}@umich.edu

December 12, 2025

Abstract

Our project for the University of Michigan's Fall 2025 EECS 470 Computer Architecture course is an implementation of an R10K-style processor for the RISC-V ISA. We designed a fully synthesizable, N-way superscalar processor with a tournament branch predictor, return address stack, early tag broadcast, and out-of-order load issuing. Our reservation station uses dependency-based FIFO queues described in Palacharla et al.'s 1997 paper to issue instructions into functional units.

1 Introduction

This semester, the majority of our efforts were spent creating a functional processor. We started with the dispatch stage, then issue, then the functional units, complete, and retire. Afterwards, we designed the caches, fetch, and memory systems. This report will describe our implementations and the design decisions we made along the way. It will conclude with an analysis of the processor's overall performance and our rationale for choosing the final parameters.

2 Design Overview

We have designed an out-of-order, N-way superscalar R10K-style processor with advanced features such as a dependency-based reservation station, out-of-order load issues, byte-level store-to-load forwarding, early tag broadcast, non-blocking multi-banked instruction and data caches, and a writeback buffer. A high-level overview of our data flow can be seen in figure 1.

2.1 Instruction Fetch

Because our processor is N-way superscalar, we ideally fetch N instructions every cycle. Spaces in the instruction buffer are allocated in the fetch stage as well.

For non-branch instructions and branches predicted not-taken, the next value of the program counter is determined by how many lines were hit in the instruction cache and how many spaces the instruction buffer has free. For `jal` instructions or when a branch is predicted taken, the fetch stage stops on the taken branch and sets the next program counter to the predicted target of the branch. For `jalr` instructions, the fetch stage stops as for a taken branch and sets the next program counter to the top of the return address stack.

2.2 Instruction Cache

We have implemented an instruction cache with a configurable number of banks. Each bank is fully associative, and we use a tree-based pseudo-LRU eviction policy. It is non-blocking but the number of lines the cache attempts to prefetch is tied to the number of banks in the cache.

2.3 Branch Target Buffer

Our branch target buffer (BTB) is a direct-mapped cache with a configurable number of lines. It provides target prediction for instructions with both static (conditional branches and `jal`) and dynamic (`jalr`) targets by caching the taken target of a branch. The BTB is updated on instruction complete for better hit rates on branches with static targets.

2.4 Map Table

Our map table is implemented as an array of 32 numbers, mapping architectural registers to physical

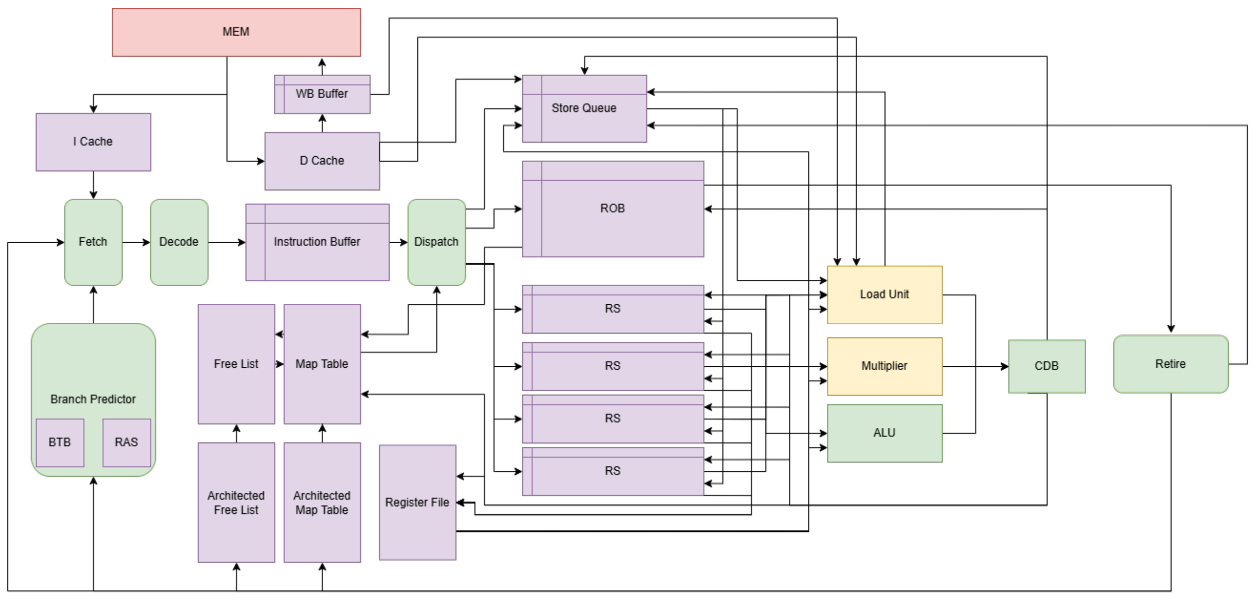

Figure 1: Processor Overview

ones. The mappings are updated during dispatch, with forwarding within a cycle ensure that all N instructions which can be dispatched obtain the correct physical register number.

Parallel to the map table, there is an architectural map table which is updated only when an instruction retires. On a branch mispredict, the architectural map table is copied over to the map table.

So that the destination physical register can be used as a unique identifier for all in-flight instructions, instructions with a destination register of 0 or which do not write to the register file are given a physical register, but the map table does not change.

2.5 Free List

Our free list is implemented as a bit vector indicating which physical registers are free. An N-grant priority selector is used to issue free physical registers to instructions in dispatch. The reorder buffer maintains a record of the new and old physical register which map to the architectural destination register of an instruction.

On retirement, the old physical register is freed and the new physical register is commited to the architectural free list which is maintained in parallel to the regular free list. If the instruction writes to architectural register 0 or does not write to the register file, the newly mapped physical register will be freed instead. On a branch mispredict, the architectural free list is copied over to the regular free list.

2.6 Reorder Buffer

Our re-order buffer (ROB) has a configurable amount of entries and is implemented as a circular buffer. The ROB keeps track of a head and tail pointer, and is able to receive between 0 and N instructions on dispatch and provide between 0 and N instructions on retire. The ROB contains a bit vector of valid entry spots which is used to determine the number of empty spaces remaining in the buffer using an N-grant priority selector. Due to early tag broadcast, our ROB has internal forwarding from complete to retire.

The ROB has additional data for handling load instructions, and these will be discussed along with the load unit in section 3.6.

2.7 Reservation Station

Our reservation station (RS) is based on the dependency-based FIFO queue design described in Palacharla et al.'s 1997 paper. The heuristic for dispatching an instruction into the queue is as follows:

- If an instruction has a data dependency with an instruction in one queue, insert into that queue.
- If an instruction has a data dependency with

instructions in two queues, insert into either queue arbitrarily.

- If an instruction has no data dependencies with instructions in the RS, insert into an empty queue.

If the insertion cannot be completed due to a structural hazard, the stage stalls.

3 Advanced Features

We implemented various advanced features. Their status is shown in table 1. Every advanced feature which was implemented was integrated while creating the processor modules, so they are difficult to remove.

Feature	Integrated
N-way Superscalar	✓
ETB	✓
Palacharla RS	✓
Tournament Predictor	✓
Prefetching	✓
Banked Cache	✓
Associative Cache	✓
Non-Blocking Cache	✓
Store Queue	✓
Store-to-Load Forwarding	✓

Table 1: Advanced Feature Status

3.1 N-way Superscalar

Our processor can fetch, decode, and dispatch N instructions every cycle. Our processor can also complete and retire N instructions every cycle. This allows for increased exploitation of ILP.

3.2 Early Tag Broadcast

We implemented early tag broadcast (ETB) by having the CDB arbiter broadcast combinationally which tags would be placed onto the CDB on the next cycle. This required the implementation of internal forwarding in the ROB and physical register file.

3.3 Palacharla Reservation Station

This feature was already discussed in 2.7.

3.4 Tournament Branch Predictor

We implemented a tournament branch predictor between gShare and a predictor with a per-PC history register and global pattern table. This decreased branch mispredict rates for a wider set of patterns.

3.5 Store Queue

Our store queue has a configurable amount of entries and tracks the address and data of a store and whether or not it has retired. On dispatch, stores are placed into the store queue, ROB, and RS. Stores are treated as an integer addition operation by the RS, so they will be issued into one of the ALUs.

When a store issues, the store queue will copy the value of `rs2` of the store and place it into the queue. When the store completes, the store queue will save the address of the store from the CDB and realign the value stored in the queue to where it would be within the word. For example, a `sh` instruction writing the value `0xBEEF` to the upper half of a word would shift the data within the queue to `0xBEEF0000`. This is to facilitate faster writing to the memory as well as more efficient byte-level store-to-load forwarding.

When the store retires from the ROB, it will be marked as ready to be sent to the data cache, and retired stores will be popped off the queue in order. A detailed depiction of the data flow for store instructions can be seen in figure 2.

3.6 Load Unit

Our load unit is a 3-stage fully pipelined functional unit. The work for loading from memory is split into three stages:

1. Address calculation
2. Store-to-load forwarding and D-cache access
3. Data alignment and complete request

The reservation station includes both a ready bit and a bit which marks whether or not all the older stores have left the store queue. The former bit determines whether or not a load is ready to issue based on its checkpointed tail pointer, and the latter bit determines whether or not to request data from the store queue. This is to prevent race conditions with out-of-order stores and loads.

In the second stage, as in the store queue, the data to be loaded is provided in the locations they are within a word, so the data is aligned to the base of the register while requesting access to the CDB. This

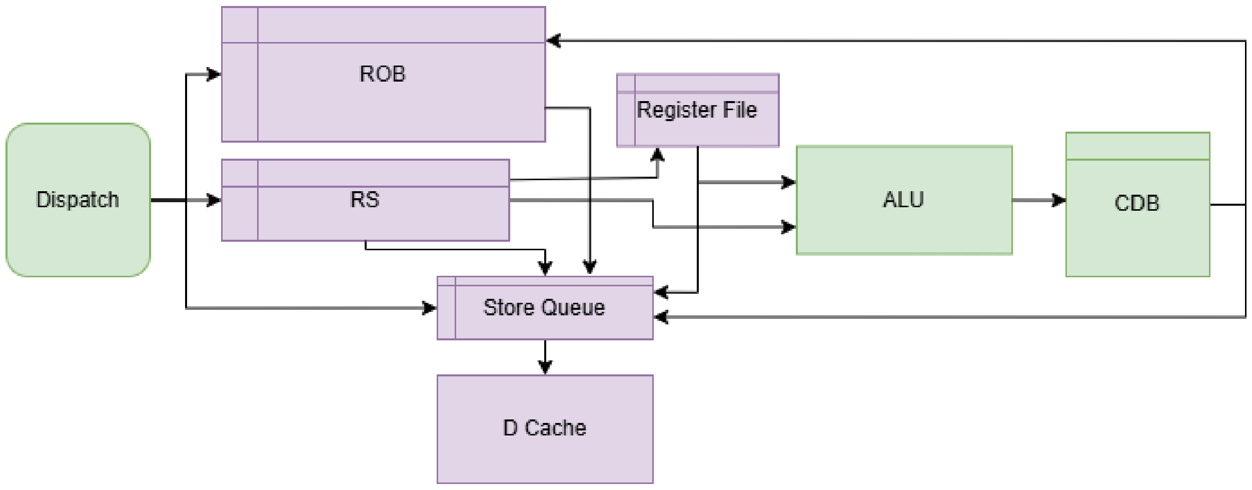

Figure 2: Dataflow of Stores

allows for faster store-to-load forwarding, as both the load unit and store queue have a bit vector of which bits within a word an instruction wants and has respectively, and can compare those directly when forwarding.

3.7 Data Cache

Our data cache, like the instruction cache, has a configurable amount of fully associative banks and uses a tree-based pseudo-LRU eviction policy. The data cache allows for multiple load and store hits on a single line in a bank per cycle.

The data cache is write-back, meaning it maintains a dirty bit to mark whether or not an evicted block should be sent back to memory. Evicted blocks which are dirty are first sent to a writeback buffer where they wait for the data cache to be free of memory requests before being sent to memory.

The writeback buffer has first priority on the memory bus when it is full, so there is always a spot to place evicted blocks into.

4 Testing and Evaluation

As we only achieved a fully correct processor very late in the semester, we did not have much time to conduct testing and evaluation of our processor.

4.1 Testing

Our testing strategy was to run the processor on the provided test programs with varying parameters to expose new edge cases in our processor which may fail, and we are confident that the processor functions correctly for $N \in \{1, 2, 3, 4\}$ and various ROB and RS sizes.

4.2 Performance Evaluation

With the time that we had, we compared the time per instruction of various sizes of ROB, RS, and SQ, as well as various amounts of ALUs and multipliers. We also attempted to vary the number of banks in the caches. Another thing we did to evaluate the performance of our processor was add a variety of signals out of the CPU module which could be used to export a CSV of functional unit and queue usage for each cycle of a program.

Parameter	Value
Clock Period	11.12ns
N	3
ROB Size	16
SQ Size	4
RAS Size	16
#ALU	2
Multipliers	1
Multiplier Stages	4
Load Units	2
RS Queues	4
RS Queue Size	4
ICache Banks	4
DCache Banks	2

Table 2: Final Processor Parameters

The parameters we chose for our final submit are show in table 2. We initially chose these arbitrarily while working on the processor, but our performance evaluation has shown them to be optimal. Although the actual minimum clock period of the processor was lower, 11.12ns was chosen to avoid the decrease in CPI associated with having to wait an extra cycle for memory, which would have decreased our overall performance.

4.2.1 Usage Analysis

The functional unit and queue usage statistics for a run of the `quicksort` program for our final submission processor are shown in figure 3. Graphs for other programs are generally similar. As can be seen, we rarely ever achieve 100% utilization of our functional units or queues, so we decided not to increase them to maintain the clock period.

We concluded that the main bottleneck in our processor was the instruction cache. Increasing the number of prefetched cache lines greatly improves our CPI, but due to the way it is implemented, also increases our clock period which offsets the CPI gains. Given more time, we would have implemented a proper prefetcher.

4.2.2 Critical Path Analysis

Currently, the critical path of our processor is from memory to the data cache. The data cache is currently designed so that when the memory provides data, any stores that were waiting on that line can forward their data on the same cycle. This improves CPI at the cost of clock period, but because prefetching more lines also increases our clock period, we kept this tradeoff because we determined that it gave us an improved TPI.

With some other configurations of the processor, the critical path is from the reservation station issue to the load unit. This is because we only determine whether or not the load unit must stall, and send that data back to the reservation station, after a cache hit. This path was overlooked until after the submission deadline, so it was not changed.

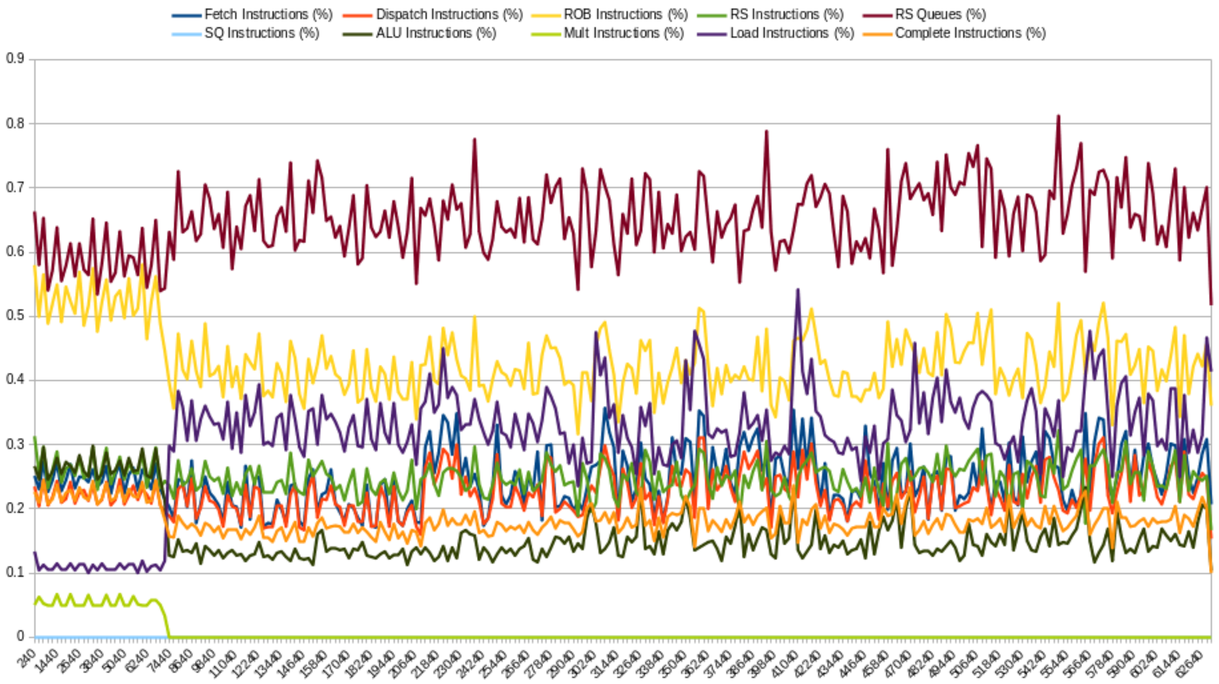

Figure 3: Usage Statistics for Functional Units and Queues averaged every 120 cycles